Dive into
Deep Learning
ADVANCED
Alexander J Smola · Rasool Fakoor

Dive into Deep Learning（2026.11.27 版）。

Table of contents

発展 **1**

9 最適化アルゴリズム **5**
9.1 ランドスケープと勾配降下法 6
9.1.1 最適化ランドスケープと条件数 7
9.1.2 臨界点、鞍点、プラトー 10
9.1.3 降下保証と極小点の安定性 12
9.1.4 安定性の縁 14
9.1.5 ニュートン法と前処理 17
9.1.6 まとめ 18
9.1.7 演習問題 19
9.2 確率的勾配降下法とミニバッチ勾配降下法 20
9.2.1 確率的勾配降下法 20
9.2.2 ミニバッチ 23
9.2.3 まとめ 27
9.2.4 演習問題 28
9.3 2 つのテストモデル 28
9.3.1 言語モデル TinyLM 29
9.3.2 画像分類器 FashionCNN 32
9.3.3 訓練ハーネスと評価補助関数 33
9.3.4 ミニバッチ SGD のベースライン実行と実験プロトコル 35
9.3.5 まとめ 38
9.3.6 演習問題 38
9.4 モメンタム 39
9.4.1 モメンタム法 39
9.4.2 加速と臨界減衰 41
9.4.3 データ上でのモメンタム 44
9.4.4 ネステロフのモメンタム 45
9.4.5 まとめ 47
9.4.6 演習問題 47
9.5 Adam と AdamW 48
9.5.1 座標ごとの適応的学習率 49
9.5.2 言語モデルにおける Adam 52
9.5.3 分離型重み減衰（AdamW） 56
9.5.4 まとめ 60
9.5.5 演習問題 60
9.6 Muon 62

9.6.1 ノルムと最急降下 . . . 62
9.6.2 ニュートン・シュルツ反復による直交化 . . . 64
9.6.3 ゼロからの Muon の実装 . . . 67
9.6.4 まとめ . . . 72
9.6.5 演習問題 . . . 72
9.7 スケジュール . . . 72
9.7.1 ノイズのある谷における学習率減衰 . . . 73
9.7.2 言語モデルにおける減衰形状 . . . 75
9.7.3 学習率ウォームアップ . . . 77
9.7.4 ウォームアップ・安定・減衰（WSD）とチェックポイント分岐 . . . 78
9.7.5 まとめ . . . 80
9.7.6 演習問題 . . . 81
9.8 バッチサイズ . . . 81
9.8.1 勾配ノイズスケール . . . 82
9.8.2 バッチサイズ、ステップ効率、学習率のスケーリング . . . 84
9.8.3 実験的検証：目標到達ステップ数とスケーリング則 . . . 85
9.8.4 まとめ . . . 90
9.8.5 演習問題 . . . 91
9.9 スケールアップ . . . 91
9.9.1 標準パラメータ化における学習率と幅 . . . 92
9.9.2 最大更新パラメータ化（μP） . . . 95
9.9.3 Transformer とスペクトルオプティマイザへの拡張 . . . 99
9.9.4 まとめ . . . 100
9.9.5 演習問題 . . . 100
9.10 実践 . . . 101
9.10.1 勾配クリッピングとスパイクからの回復 . . . 101
9.10.2 重み平均化（EMA） . . . 103
9.10.3 エンドツーエンド設定の実例 . . . 106
9.10.4 まとめ . . . 109
9.10.5 演習問題 . . . 109
9.11 関連文献 . . . 109
9.11.1 基礎的な教科書、サーベイ、プレイブック . . . 110
9.11.2 損失曲面 . . . 110
9.11.3 古典的な 2 次手法と分散低減手法 . . . 111
9.11.4 行列オプティマイザと構造的オプティマイザ . . . 111
9.11.5 スケジュール、平均化、大規模ベンチマーク . . . 112

10 アテンション **113**
10.1 クエリ、キー、値 . . . 114
10.1.1 微分可能な参照としてのアテンション . . . 114
10.1.2 固定カーネルによるアテンションプーリング . . . 115
10.1.3 診断手段としてのアテンション重み . . . 117
10.1.4 まとめ . . . 118
10.1.5 演習問題 . . . 118
10.2 アテンションのスコアリングとマスキング . . . 120
10.2.1 ドット積アテンション . . . 121
10.2.2 マスキング . . . 124
10.2.3 ミニバッチに対するアテンション . . . 128
10.2.4 学習されたアライメントとしてのアテンション . . . 130
10.2.5 まとめ . . . 132

10.2.6 演習問題 . . . 133
10.3 自己アテンションと複数のヘッド . . . 135
10.3.1 自己アテンション . . . 135
10.3.2 単一ヘッドの限界 . . . 138
10.3.3 マルチヘッドアテンション . . . 139
10.3.4 クロスアテンション . . . 145
10.3.5 まとめ . . . 145
10.3.6 演習問題 . . . 145
10.4 位置情報 . . . 147
10.4.1 置換同変性 . . . 147
10.4.2 絶対位置埋め込み . . . 148
10.4.3 回転位置埋め込み . . . 150
10.4.4 距離ペナルティとエンコーディングなし . . . 152
10.4.5 訓練長を超える外挿 . . . 152
10.4.6 まとめ . . . 158
10.4.7 演習問題 . . . 158
10.5 アテンションのコスト . . . 160
10.5.1 二次の時間とメモリ . . . 160
10.5.2 行列を保持しない厳密なアテンション . . . 162
10.5.3 まとめ . . . 166
10.5.4 演習問題 . . . 166
10.6 ウィンドウアテンションと線形アテンション . . . 168
10.6.1 スライディングウィンドウアテンション . . . 168
10.6.2 線形アテンション . . . 172
10.6.3 機構の選び方 . . . 176
10.6.4 まとめ . . . 176
10.6.5 演習問題 . . . 177
10.7 アテンション回路 . . . 178
10.7.1 回路と残差ストリーム . . . 179
10.7.2 コピーと誘導 . . . 181
10.7.3 回路の裏付け . . . 184
10.7.4 コンテキスト内学習 . . . 189
10.7.5 説明としてのアテンション重み . . . 191
10.7.6 まとめ . . . 191
10.7.7 演習問題 . . . 191
10.8 関連文献 . . . 194
10.8.1 ルックアップとその正規化 . . . 194
10.8.2 自己アテンションとヘッド . . . 195
10.8.3 位置 . . . 196
10.8.4 コスト . . . 196
10.8.5 大規模モデルにおける回路 . . . 197
10.8.6 推奨される参考資料 . . . 198

11 Transformer **199**
11.1 Transformer ブロック . . . 200
11.1.1 Transformer ブロックの構成要素 . . . 200
11.1.2 正規化 . . . 201
11.1.3 フィードフォワードネットワーク：GELU から SwiGLU へ . . . 208
11.1.4 設定可能な `TransformerBlock` . . . 209
11.1.5 まとめ . . . 210

11.1.6 演習問題 . . . 210
11.2 デコーダのみの言語モデル（GPT） . . . 211
11.2.1 ブロックから言語モデルへ . . . 211
11.2.2 WikiText-103 におけるバイトレベル GPT の訓練 . . . 215
11.2.3 統制された比較 . . . 218
11.2.4 モデルからのサンプリング . . . 220
11.2.5 事前学習済み GPT-2（124M）の読み込み . . . 221
11.2.6 現代的なレシピ . . . 223
11.2.7 まとめ . . . 224
11.2.8 演習問題 . . . 225
11.3 KV キャッシュとキャッシュ削減 . . . 226
11.3.1 再計算からキャッシュへ . . . 226
11.3.2 メモリ容量とハードウェアのルーフライン . . . 232
11.3.3 KV キャッシュの縮小 . . . 235
11.3.4 キャッシュ削減手法の比較 . . . 240
11.3.5 まとめ . . . 240
11.3.6 演習問題 . . . 240
11.4 エンコーダ、デコーダ、およびクロスアテンション . . . 242
11.4.1 エンコーダ、デコーダ、およびエンコーダ・デコーダアーキテクチャ . . . 242
11.4.2 双方向文脈とマスクトークン予測 . . . 242
11.4.3 エンコーダ・デコーダにおけるクロスアテンション . . . 246
11.4.4 インターフェースとしてのクロスアテンション：固定サイズの潜在ボトルネック . . . 250
11.4.5 3 つの配線の使い分け . . . 251
11.4.6 まとめ . . . 252
11.4.7 演習問題 . . . 252
11.5 Vision Transformer（ViT） . . . 253
11.5.1 トークンとしてのパッチ（PatchEmbedding） . . . 253
11.5.2 Vision Transformer のアーキテクチャ . . . 254
11.5.3 Fashion-MNIST での訓練 . . . 256
11.5.4 パラメータ数を揃えた畳み込みネットワークとの比較 . . . 259
11.5.5 まとめ . . . 262
11.5.6 演習問題 . . . 262
11.6 スケーリング則 . . . 263
11.6.1 パラメータ数（$N \approx 12Ld^2$）と FLOPs（$C \approx 6ND$）の算出 . . . 263
11.6.2 WikiText-103 における小規模スケーリング実験 . . . 265
11.6.3 計算量最適な配分（Chinchilla スケーリング） . . . 268
11.6.4 まとめ . . . 271
11.6.5 演習問題 . . . 271
11.7 混合エキスパート（MoE） . . . 273
11.7.1 条件付き計算：保持パラメータとアクティブパラメータ . . . 273
11.7.2 混合エキスパート層 . . . 274
11.7.3 ルーティング崩壊と負荷分散 . . . 278
11.7.4 混合エキスパート言語モデル . . . 282
11.7.5 まとめ . . . 285
11.7.6 演習問題 . . . 285
11.8 関連文献 . . . 286
11.8.1 正規化の配置と残差ストリームの派生形 . . . 286
11.8.2 フィードフォワードネットワークと活性化関数 . . . 287
11.8.3 長文脈推論とキャッシュ圧縮 . . . 288

11.8.4 エンコーダ、クロスアテンション、ビジョンバックボーン 288
11.8.5 スケーリング則と混合エキスパートの設計 289

12 状態空間モデル **291**
12.1 ゲート付きリカレントネットワーク 292
12.1.1 ゲート 292
12.1.2 長・短期記憶 293
12.1.3 ゲート付きリカレントユニット 295
12.1.4 ゲート付きネットワークの訓練 297
12.1.5 まとめ 305
12.1.6 演習問題 305
12.2 系列対系列学習 307
12.2.1 エンコーダ・デコーダの抽象化 307
12.2.2 翻訳のスコアリング 308
12.2.3 固定状態のボトルネック 310
12.2.4 まとめ 313
12.2.5 演習問題 313
12.3 線形漸化式と並列スキャン 314
12.3.1 線形漸化式 314
12.3.2 並列スキャン 315
12.3.3 線形リカレント言語モデル 317
12.3.4 アテンションとのコスト比較 320
12.3.5 まとめ 322
12.3.6 演習問題 322
12.4 状態空間モデル 323
12.4.1 連続時間から漸化式へ 323
12.4.2 対角状態空間層 327
12.4.3 まとめ 331
12.4.4 演習問題 331
12.5 選択的状態空間モデル 332
12.5.1 選択的コピー 333
12.5.2 選択的ダイナミクス 335
12.5.3 Mamba 層 336
12.5.4 訓練済み Mamba モデル 340
12.5.5 まとめ 344
12.5.6 演習問題 345
12.6 行列状態漸化式 346
12.6.1 連想メモリ 346
12.6.2 減衰 349
12.6.3 双対形式 350
12.6.4 ゲート付き線形アテンション層 354
12.6.5 まとめ 359
12.6.6 演習問題 359
12.7 デルタ則 361
12.7.1 加算的書き込み 361
12.7.2 勾配ステップとしての書き込み 362
12.7.3 Gated DeltaNet 365
12.7.4 回帰としてのメモリ 371
12.7.5 まとめ 373
12.7.6 演習問題 373

12.8 ハイブリッドアーキテクチャ 374
12.8.1 固定サイズの状態の限界 374
12.8.2 1つのアテンション層による想起 376
12.8.3 同規模での言語モデリング 378
12.8.4 推論時のメモリ 382
12.8.5 本章のメモリの比較 383
12.8.6 まとめ 384
12.8.7 演習問題 384
12.9 関連文献 385
12.9.1 再帰とアテンションの組み合わせ 386
12.9.2 行列メモリへの書き込み 386
12.9.3 状態空間モデルと長い畳み込み 387
12.9.4 言語を超えて 388
12.9.5 参照実装とチュートリアル 388

13 計算性能 **389**
13.1 性能モデル 390
13.1.1 演算密度とルーフライン 390
13.1.2 GPU コードの時間計測 393
13.1.3 3つのレジーム 394
13.1.4 訓練ステップのプロファイリング 399
13.1.5 まとめ 401
13.1.6 演習問題 401
13.2 ハードウェア 402
13.2.1 メモリ階層 403
13.2.2 演算性能とメモリ帯域幅 406
13.2.3 GPU 408
13.2.4 GPU 外のデータ経路 410
13.2.5 ルーフライン上のワークロード 411
13.2.6 まとめ 412
13.2.7 演習問題 413
13.3 計算グラフとコンパイル 414
13.3.1 グラフキャプチャ 415
13.3.2 演算融合 418
13.3.3 訓練ステップのコンパイル 419
13.3.4 カーネルリプレイ 420
13.3.5 コンパイルの限界 422
13.3.6 まとめ 422
13.3.7 演習問題 423
13.4 メモリと精度 424
13.4.1 訓練ステップのメモリ 424
13.4.2 混合精度 429
13.4.3 活性化チェックポイント 431
13.4.4 勾配累積 432
13.4.5 メモリ手法の選択 433
13.4.6 まとめ 434
13.4.7 演習問題 434
13.5 第一原理からのマルチ GPU 436
13.5.1 3つの分割方法 436
13.5.2 手作業によるデータ並列 436

13.5.3 リング allreduce . . . 440
13.5.4 通信コストとスケーリング . . . 442
13.5.5 まとめ . . . 446
13.5.6 演習問題 . . . 447
13.6 データ並列の実践 . . . 448
13.6.1 実用データ並列 . . . 448
13.6.2 4 基の GPU でのスケーリング . . . 453
13.6.3 まとめ . . . 457
13.6.4 演習問題 . . . 457
13.7 訓練状態のシャーディング . . . 458
13.7.1 オプティマイザ状態、勾配、重みの分割 . . . 459
13.7.2 メモリと通信 . . . 460
13.7.3 2 基の GPU によるシャーディング訓練 . . . 461
13.7.4 まとめ . . . 463
13.7.5 演習問題 . . . 463
13.8 ケーススタディ：Transformer の最適化 . . . 465
13.8.1 実験設定 . . . 465
13.8.2 単一 GPU . . . 467
13.8.3 データ並列 . . . 473
13.8.4 結果 . . . 476
13.8.5 まとめ . . . 478
13.8.6 演習問題 . . . 478
13.9 関連文献 . . . 480
13.9.1 計測とモデル化 . . . 480
13.9.2 ハードウェアと数値形式 . . . 481
13.9.3 コンパイラとカーネル . . . 482
13.9.4 メモリ . . . 483
13.9.5 通信とスケール . . . 483

14 強化学習 **485**
参考資料と関連文献 . . . 486
14.1 マルコフ決定過程 . . . 487
14.1.1 モデル . . . 489
14.1.2 収益・割引・ホライズン . . . 491
14.1.3 状態の選び方 . . . 495
14.1.4 報酬設計 . . . 496
14.1.5 2 つの特別な場合 . . . 498
14.1.6 まとめ . . . 500
14.1.7 演習問題 . . . 500
14.2 動的計画法 . . . 503
14.2.1 価値とベルマン方程式 . . . 503
14.2.2 価値反復法 . . . 506
14.2.3 方策評価と方策反復法 . . . 515
14.2.4 滑る氷の上での最適方策 . . . 518
14.2.5 報酬シェイピングと最適方策 . . . 521
14.2.6 まとめ . . . 523
14.2.7 演習問題 . . . 524
14.3 時間差分学習 . . . 526
14.3.1 サンプルによるバックアップ . . . 527
14.3.2 滑る湖における Q 学習 . . . 531

14.3.3 オフポリシー学習 . . . 537
14.3.4 学習されたモデルにおける計画 . . . 538
14.3.5 まとめ . . . 540
14.3.6 演習問題 . . . 540
14.4 探索 . . . 543
14.4.1 多腕バンディット問題 . . . 544
14.4.2 楽観主義と事後サンプリング . . . 548
14.4.3 MDP における探索 . . . 552
14.4.4 検証器タスクにおける探索 . . . 556
14.4.5 まとめ . . . 558
14.4.6 演習問題 . . . 558
14.5 デモンストレーションからの学習 . . . 560
14.5.1 行動クローニング . . . 561
14.5.2 分布シフト . . . 565
14.5.3 複合誤差 . . . 568
14.5.4 対話的なデータ収集 . . . 570
14.5.5 まとめ . . . 574
14.5.6 演習問題 . . . 574
14.6 最適制御、LQR、モデル予測制御 . . . 576
14.6.1 線形二次制御 . . . 577
14.6.2 平衡点周りでの非線形システムの線形化 . . . 581
14.6.3 反復 LQR による非線形軌跡最適化 . . . 583
14.6.4 移動ホライズンモデル予測制御 . . . 586
14.6.5 まとめ . . . 590
14.6.6 演習問題 . . . 590
14.7 方策勾配 . . . 592
14.7.1 方策のパラメータ化 . . . 593
14.7.2 方策勾配 . . . 594
14.7.3 共通例題における REINFORCE . . . 597
14.7.4 方策勾配定理 . . . 603
14.7.5 コストと限界 . . . 604
14.7.6 まとめ . . . 607
14.7.7 演習問題 . . . 607
14.8 ベースラインと分散低減 . . . 610
14.8.1 ゼロ平均恒等式 . . . 611
14.8.2 残りの累積報酬、ベースライン、制御変量 . . . 612
14.8.3 中心化、スケーリング、正規化 . . . 615
14.8.4 5 つの推定量の比較 . . . 620
14.8.5 まとめ . . . 627
14.8.6 演習問題 . . . 628
14.9 強化学習における関数近似 . . . 631
14.9.1 テーブルに代わるネットワーク . . . 631
14.9.2 CartPole における REINFORCE . . . 635
14.9.3 連続行動 . . . 638
14.9.4 状態間の汎化 . . . 643
14.9.5 手法の見取り図 . . . 645
14.9.6 まとめ . . . 647
14.9.7 演習問題 . . . 647

15 深層強化学習 653

15.1 アクター・クリティック法と複数ステップ収益 654
15.1.1 時間差分アドバンテージ 655
15.1.2 アクター・クリティックの更新 657
15.1.3 複数ステップ収益と GAE 664
15.1.4 λ 収益のバイアスと分散 666
15.1.5 まとめ 671
15.1.6 演習問題 671
15.2 信頼領域と近接方策最適化 674
15.2.1 パラメータ空間と方策空間 675
15.2.2 重要度サンプリング 676
15.2.3 信頼領域 678
15.2.4 PPO の実践 681
15.2.5 まとめ 693
15.2.6 演習問題 693
15.3 KL 正則化方策最適化 697
15.3.1 比較からの報酬の学習 697
15.3.2 当てはめ報酬の最適化 701
15.3.3 KL 正則化目的関数 702
15.3.4 ソフトバックアップ 706
15.3.5 まとめ 710
15.3.6 演習問題 710
15.4 深層 Q ネットワーク 712
15.4.1 不安定性の要因 714
15.4.2 経験再生とターゲットネットワーク 716
15.4.3 過大評価 726
15.4.4 まとめ 729
15.4.5 演習問題 730
15.5 ソフトアクタークリティック 733
15.5.1 最大エントロピー目的関数 734
15.5.2 スクワッシュされたガウス方策 737
15.5.3 アルゴリズム 741
15.5.4 サンプル効率と較正 747
15.5.5 まとめ 754
15.5.6 演習問題 754
15.6 学習されたモデルと計画 757
15.6.1 データからのダイナミクスモデル 758
15.6.2 モデル内での計画 763
15.6.3 方策改善としての探索 768
15.6.4 まとめ 773
15.6.5 演習問題 773
15.7 オフライン強化学習 775
15.7.1 固定された記録からの学習 775
15.7.2 悲観主義 784
15.7.3 表形式を超えて 786
15.7.4 まとめ 788
15.7.5 演習問題 788
15.8 系列生成のための強化学習 791
15.8.1 軌跡としての応答 791
15.8.2 検証器・報酬モデルとその抜け穴 793
15.8.3 事後学習の更新則 795

15.8.4 まとめ . . . 798
15.8.5 演習問題 . . . 798
15.9 関連文献 . . . 800
15.9.1 実践における方策最適化 . . . 800
15.9.2 DQN 以降の価値学習 . . . 801
15.9.3 2018 年以降の連続制御 . . . 802
15.9.4 モデル、想像、探索 . . . 802
15.9.5 大規模なオフライン手法 . . . 803
15.9.6 事後学習の変種 . . . 804
15.9.7 本章の先の話題 . . . 804
15.9.8 参考書、ベンチマーク、コードベース . . . 805
15.9.9 プロジェクト . . . 805

16 敵対的生成ネットワーク 807
記法 . . . 808
16.1 敵対的生成ネットワーク . . . 808
16.1.1 尤度から 2 サンプル比較へ . . . 809
16.1.2 対数損失の目的関数 . . . 810
16.1.3 生成器の勾配 . . . 813
16.1.4 2 次元ガウス分布での理論の検証 . . . 816
16.1.5 Fashion-MNIST での MLP GAN の訓練 . . . 822
16.1.6 まとめ . . . 824
16.1.7 演習問題 . . . 824
16.2 敵対的目的関数とダイバージェンス . . . 826
16.2.1 一般的なテンプレート . . . 827
16.2.2 分類損失と f ダイバージェンス . . . 827
16.2.3 積分確率指標 . . . 833
16.2.4 データから遠い領域と近い領域における勾配 . . . 834
16.2.5 まとめ . . . 842
16.2.6 演習問題 . . . 843
16.3 相対論的目的関数 . . . 844
16.3.1 ペアのスコア付け . . . 844
16.3.2 ペアリングダイバージェンス . . . 846
16.3.3 ランキングとモード被覆 . . . 848
16.3.4 まとめ . . . 851
16.3.5 演習問題 . . . 852
16.4 勾配ペナルティと収束 . . . 853
16.4.1 Dirac-GAN . . . 853
16.4.2 ゼロ中心勾配ペナルティ . . . 855
16.4.3 勾配ペナルティが価値関数に及ぼす影響 . . . 858
16.4.4 R3GAN レシピ . . . 860
16.4.5 まとめ . . . 865
16.4.6 演習問題 . . . 865
16.5 敵対的画像生成 . . . 866
16.5.1 DCGAN アーキテクチャ . . . 867
16.5.2 正規化なしバックボーン . . . 873
16.5.3 統制された損失の比較 . . . 876
16.5.4 定量的評価指標 . . . 880
16.5.5 潜在補間とクリティックの過学習 . . . 884
16.5.6 まとめ . . . 885

16.5.7 演習問題 . . . 886
16.6 条件付き生成 . . . 887
16.6.1 条件付き目的関数 . . . 888
16.6.2 条件付けのアーキテクチャ . . . 888
16.6.3 CIFAR-10 におけるクラス条件付き生成 . . . 890
16.6.4 条件付き生成器の評価 . . . 894
16.6.5 まとめ . . . 898
16.6.6 演習問題 . . . 898
16.7 GAN を超える敵対的損失 . . . 899
16.7.1 条件付き平均と多峰性ターゲット . . . 900
16.7.2 拡散蒸留における識別器 . . . 903
16.7.3 本章の敵対的目的関数の比較 . . . 906
16.7.4 まとめ . . . 907
16.7.5 演習問題 . . . 907
16.8 関連文献 . . . 909
16.8.1 推奨される入門文献とコードベース . . . 909
16.8.2 目的関数とミニマックスダイナミクス . . . 909
16.8.3 固定された差異と学習される差異 . . . 910
16.8.4 アーキテクチャ、特徴空間、データ効率 . . . 910
16.8.5 条件付け、画像変換、現代のハイブリッドシステム . . . 911

17 拡散モデル **913**
17.1 ノイズ除去とスコア . . . 914
17.1.1 通し例 . . . 914
17.1.2 最適ノイズ除去器 . . . 918
17.1.3 単一ノイズレベルの限界 . . . 922
17.1.4 画像のノイズ除去 . . . 927
17.1.5 まとめ . . . 929
17.1.6 演習問題 . . . 930
17.2 ガウス経路：拡散とフローマッチング . . . 932
17.2.1 データからノイズへの経路 . . . 932
17.2.2 周辺場への回帰 . . . 938
17.2.3 まとめ . . . 946
17.2.4 演習問題 . . . 946
17.3 積分によるサンプリング . . . 948
17.3.1 確率質量の輸送 . . . 949
17.3.2 確率フロー ODE に対するオイラー法 . . . 951
17.3.3 祖先連鎖と逆 SDE . . . 955
17.3.4 両サンプラからのサンプル . . . 957
17.3.5 まとめ . . . 959
17.3.6 演習問題 . . . 959
17.4 画像の拡散モデル . . . 961
17.4.1 ノイズ除去 U-Net . . . 961
17.4.2 訓練とサンプリング . . . 965
17.4.3 サンプル品質の測定 . . . 969
17.4.4 ガイダンス . . . 972
17.4.5 まとめ . . . 978
17.4.6 演習問題 . . . 978
17.5 サンプラ、逆変換、編集 . . . 980
17.5.1 決定論的サンプラと確率的サンプラ . . . 981

17.5.2 ステップ数の削減 . . . 983
17.5.3 逆変換と編集 . . . 994
17.5.4 まとめ . . . 999
17.5.5 演習問題 . . . 999
17.6 尤度、重み付け、設計空間 . . . 1001
17.6.1 尤度境界 . . . 1002
17.6.2 重み付けとしての目的関数 . . . 1011
17.6.3 設計空間 . . . 1014
17.6.4 まとめ . . . 1023
17.6.5 演習問題 . . . 1023
17.7 潜在拡散と Transformer . . . 1025
17.7.1 知覚圧縮 . . . 1025
17.7.2 潜在空間での拡散 . . . 1030
17.7.3 Transformer バックボーン . . . 1035
17.7.4 表現とバックボーンの選択 . . . 1041
17.7.5 まとめ . . . 1042
17.7.6 演習問題 . . . 1042
17.8 少数ステップ生成 . . . 1043
17.8.1 長いオイラーステップ . . . 1044
17.8.2 フロー写像 . . . 1045
17.8.3 フロー写像の訓練 . . . 1050
17.8.4 分布マッチング . . . 1057
17.8.5 まとめ . . . 1058
17.8.6 演習問題 . . . 1059
17.9 離散拡散 . . . 1060
17.9.1 トークンの破損 . . . 1061
17.9.2 マスク拡散の変分上界 . . . 1063
17.9.3 マスク解除によるサンプリング . . . 1066
17.9.4 訓練されたノイズ除去器 . . . 1069
17.9.5 隣接するモデルと振り返り . . . 1076
17.9.6 まとめ . . . 1078
17.9.7 演習問題 . . . 1078
17.10 関連文献 . . . 1081
17.10.1 専門書、講義、コード . . . 1081
17.10.2 エネルギーベースモデルとスコアマッチング . . . 1082
17.10.3 画像ネットワークとガイダンス . . . 1083
17.10.4 評価、暗記、汎化 . . . 1083
17.10.5 サンプラ、編集、逆問題 . . . 1083
17.10.6 スケジュール、重み付け、尤度 . . . 1084
17.10.7 潜在表現、テキスト、動画 . . . 1084
17.10.8 少数ステップ生成 . . . 1084
17.10.9 離散拡散 . . . 1084
17.10.10 事後学習と視覚・テキスト以外の応用 . . . 1085

参考文献 **1087**

発展

「**基礎**」パートの締めくくりでは、H. G. Wells の『The Time Machine』を用いてリカレントニューラルネットワークを訓練した。「**The Time Traveller**」という言葉をプロンプトとして与えると、この小さなモデルは英語として認識できる単語や句読点こそ出力するものの、十数文字もすると文法的な一貫性が崩れてしまう。対照的に、今日の最前線にある基盤モデル（foundation model）は、パラメータ化された層、損失関数、自動微分（automatic differentiation）、勾配ベースのオプティマイザといったまったく同じ基本的な構成要素から築かれているにもかかわらず、立ち入った技術的な対話を長く続け、ソフトウェアを書いてデバッグし、競技数学の問題を解き、1 文のテキストから写実的な画像や動画を合成する。

教室で作るプロトタイプと現代の最前線のシステムを分かつものは何か。その隔たりを越えるには、トイデータセット上の小さなモデルでは決して直面しない 4 つの根本的な課題を解決する必要がある。

1. **最適化の安定性とスケーリング**。「**基礎**」パートでは、ミニバッチ確率的勾配降下法、学習率スケジュール、バッチサイズを主に便利なレシピとして扱ってきた。しかし、ネットワークが数百万から数十億個のパラメータへと大規模化するにつれ、幅の狭い層を滑らかに訓練できる学習率であっても、より幅広い、あるいはより深いネットワークでは瞬時に発散を引き起こすことがあり、悪条件の損失曲面によって数百万ステップにわたり進行が停滞することもある。
2. **長距離の記憶と並列な系列処理**。古典的なリカレントネットワークは、どの細部が後から重要になるかわからないうちに、それまでに読み込んだすべてを固定幅の単一の隠れベクトルへと圧縮せざるを得ない。しかも、各タイムステップが直前のタイムステップの完了を待つため、再帰的な訓練では現代の GPU が備える大規模な並列性を活かせない。
3. **ハードウェアのボトルネックと分散システム**。現代のアクセラレータは、オフチップメモリとオンチップレジスタの間、あるいは GPU 間のネットワークリンク越しにテンソルを移動させる速度よりも、はるかに高速に浮動小数点演算を実行する。大規模環境では、モデルの実環境における訓練速度や推論速度は、単純な演算回数と同程度に、メモリ帯域幅、数値精度、通信トポロジーによって左右される。
4. **静的なラベルを超えた学習**。教師ありの交差エントロピーは、すべての入力に一意な正解ラベルがある場合には機能する。しかし、環境内で行動する自律エージェントは、自身の方策（policy）が変化するにつれて分布がシフトする、遅延した疎なスカラー報酬から学習しなければならない。また、高解像度画像を合成する生成モデルには一致させるべき単一の目標ラベルが存在しないため、確率分布全体を比較する目的関数や、連続的な破壊過程を逆転させる目的関数が必要となる。

PyTorch と JAX の両方で実装された本パートの 9 つの章では、これら 4 つの課題に体系的に取り組む。

- **最適化とスケーリング（第 9 章）**。まずは、悪条件の曲率、鞍点（saddle point）、ミニバッチ勾配のノイズといった要因によって高次元の非凸最適化（nonconvex optimization）がな

ぜ難しくなるのか、そして現代のオプティマイザがそれらの障壁をいかに克服するのかを検証する。確率的勾配降下法（stochastic gradient descent）からモメンタム、AdaGrad、RMSProp、Adam、分離型重み減衰（decoupled weight decay、AdamW）への発展を、ウォームアップやコサイン学習率スケジュールと組み合わせてたどり、最後に、臨界バッチサイズや最大更新パラメータ化（maximal update parametrization、μP）を通じてバッチサイズや幅の拡大に伴って学習率をどう変化させるべきかを明らかにし、完全な訓練プロトコルで締めくくる。

- **アテンション機構と Transformer（第 10 章と第 11 章）**。第 10 章では、キー、値、クエリに対する微分可能な内容参照型検索（content-addressable lookup）である**アテンション**を導入し、古典的な系列モデルにおける固定長ベクトルの記憶ボトルネックを打破する。Nadaraya–Watson カーネル回帰から出発し、加法アテンションとスケール化ドット積アテンション、系列対系列のアライメントのための Bahdanau アテンション、そしてマルチヘッド自己アテンションを構築する。続く第 11 章では再帰構造を完全に排し、自己アテンション、位置ごとのフィードフォワードネットワーク、層正規化（layer normalization）、残差接続、および回転位置埋め込み（rotary position embedding、RoPE）を含む位置エンコーディングを組み合わせて、双方向エンコーダ（BERT）、因果的自己回帰デコーダ（GPT）、Vision Transformer（ViT）という 3 つの基本的な **Transformer** パラダイムへと統合する。あわせて、長文脈アテンションの計算上のボトルネック、生成時のキー・値キャッシュ（KV キャッシュ）、スパースな混合エキスパート（mixture of experts、MoE）層も掘り下げる。
- **現代のリカレントネットワークと状態空間モデル（第 12 章）**。すべてのトークンが他のすべてのトークンに注意を向けなければならない場面では二次の自己アテンションが主流となるが、数十万トークンにわたって肥大化し続けるキー・値キャッシュを保持することは推論時に高コストとなる。そこで第 12 章では、「**推論時には固定サイズの状態を保ちながら、系列のすべての位置にわたって並列に訓練するにはどうすればよいか**」という新たな問いを携えて再帰構造へと立ち返る。状態の非線形な再帰を取り除くことで結合的な**スキャン**（associative scan）による厳密な並列評価が可能になることを示し、線形漸化式（linear recurrence）を連続時間の**状態空間モデル**（state space model、SSM）や HiPPO 多項式射影と結びつけ、さらに行列値の状態（線形アテンション）、入力依存の**選択的**（selective）状態空間モデル（Mamba）、誤り訂正型のデルタ則（DeltaNet）へと進む。
- **計算性能（第 13 章）**。第 13 章では、訓練ステップの時間とメモリがどこに費やされるのかを問う。ルーフラインモデルによって算術演算に律速される演算とメモリトラフィックに律速される演算とを切り分け、さらに 3 つ目の制約である、すべての演算を発行するホスト側の処理によって、細かな演算がなぜ GPU をアイドル状態のまま遊ばせてしまうのかを説明する。そのうえで、コンパイル（`torch.compile` と `jax.jit`）、bf16 演算、より大きなミニバッチ、活性化チェックポイント（activation checkpointing）を組み合わせることで、1 基の GPU 上における GPT の訓練ステップのスループットを約 2 倍に高める。続いて複数の GPU へと拡張し、手書き実装と実用ライブラリの両方によるデータ並列（data parallelism）、allreduce とそのコストモデル、通信と逆伝播のオーバーラップ、そして単一のカードには収まらないモデルを訓練するための重み・勾配・オプティマイザ状態のシャーディング（ZeRO と FSDP）を扱う。
- **強化学習と深層強化学習（第 14 章と第 15 章）**。第 14 章では、不確実性の下での意思決定に関する数学的・アルゴリズム的基礎を築く。具体的には、マルコフ決定過程（Markov decision process）、ベルマン最適方程式、価値反復法と方策反復法、モデルフリーの時間差分学習（Q 学習）、多腕バンディット問題における探索戦略、模倣学習、既知のダイナミクスの下での連続最適制御（倒立振子における LQR、iLQR、モデル予測制御）、REINFORCE と分散低減ベースラインによる直接的な方策最適化、およびニューラル方策勾配を扱う。続く第 15 章では強化学習を深層関数近似へとスケールさせ、深層 Q ネットワーク（DQN）における経験再生とターゲットネットワークによって不安定性の「死の三要素」（deadly triad）に対処したうえで、一般化アドバンテージ推定（generalized

advantage estimation、GAE）を備えたアクタークリティックのアーキテクチャ、近接方策最適化（proximal policy optimization、PPO）、最大エントロピーに基づくソフトアクタークリティック（soft actor-critic、SAC）、オフライン強化学習とモデルベース強化学習、KL 正則化制御、さらには自己回帰言語モデルのための強化学習（GRPO）を展開する。

- **敵対的生成ネットワークと拡散モデル（第 16 章と第 17 章）**。最後の 2 つの章では、現代の深層生成モデリングを発展させる。第 16 章では、生成サンプルと本物のサンプルを見分けるよう学習するクリティックと生成器との間のゲームとして生成を捉える。そこでは、各クリティックの損失がどのダイバージェンス（Jensen–Shannon、他の f ダイバージェンス、MMD、W_1）を測っているのか、また本物から離れた生成器にもなお勾配が得られるのかを問い、本物と生成サンプルをペアでスコア付けするクリティックを比較するとともに、ゼロ中心の勾配ペナルティである R_1 と R_2 がなぜ勾配による訓練をデータ近傍へと収束させるのかを小さなゲーム上で示す。さらに、条件付きモデルを含めた画像生成器を構築してスコア付けし、今日なおボコーダ、画像オートエンコーダ、および拡散モデルから蒸留された 1 ステップ生成器の訓練に用いられている敵対的項で締めくくる。続く第 17 章では、今日の主力となる画像・音声・動画モデルの背後にある連続時間の生成パラダイムを展開する。信号対雑音比（signal-to-noise ratio）によってパラメータ化されたガウス確率経路に沿ってデータを破壊し、クリーンなデータ、ノイズ、スコア関数、あるいは**速度場**（velocity field）を予測するよう深層ノイズ除去器（denoiser）を訓練し、逆時間の確率微分方程式（stochastic differential equation、SDE）または確率フロー ODE を積分して新しいサンプルを生成する。その過程で、ノイズ除去拡散（DDPM/DDIM）と**フローマッチング**を統一し、分類器なしガイダンス（classifier-free guidance）を備えた U-Net および Diffusion Transformer（DiT）のバックボーンを構築し、一貫性（consistency）と平均速度（average velocity）の蒸留によってサンプリングをわずか数ステップへと高速化する。

これらの章は、必ずしも先頭から順番どおりに読む必要はない。どの章がどの章を前提としているかは、まえがきの依存関係図（Figure 1）に示されている。これらの章が一体となって、「**言語モデル**」パートおよび「**画像モデル**」パートが大規模な基盤モデルに応用するアーキテクチャ・数学・システム面のツールキットを提供する。

Chapter 9

最適化アルゴリズム

前章までに登場したすべての深層ネットワークは、モメンタム 0.9 のミニバッチ SGD や、線形ウォームアップとコサイン減衰を組み合わせた AdamW といった標準的なレシピで訓練してきた（7.6 節）。訓練の実行中に値が nan へ発散すれば学習率を下げ、進行が停滞すればスケジュールやバッチサイズを調整する。本章ではオプティマイザのブラックボックスを開き、3 つの実践的な問いに答える。安全に使える最大の学習率とそれ未満での収束速度は何によって決まるのか、現代的な訓練レシピの各要素は更新ステップをどのように作り変えるのか、そしてモデルやバッチサイズが大きくなるにつれてそれらのハイパーパラメータをどのようにスケーリングしなければならないのか。

勾配降下法そのものは現代のコンピュータより古く、1847 年に Augustin-Louis Cauchy が惑星の軌道を計算するために提案し、1951 年には Herbert Robbins と Sutton Monro がノイズを含む観測のもとで減衰するステップサイズを用いた求根問題を解析した。古典的な設定と現代的な設定のいずれにおいても、勾配ベースのあらゆるオプティマイザは次の 2 つの根本的な障害によって制約される。

1. **悪条件性（不均一な曲率）：**損失曲面がある方向には急激に曲がり、別の方向には緩やかに曲がっているとき、最も急な方向で安定を保てるほど小さい学習率を選ぶと、最も平坦な方向への進行が極端に遅くなる。
2. **勾配ノイズ：**勾配はデータセット全体ではなく有限サイズのミニバッチから推定されるため、各更新にはサンプリング分散が伴い、一定のステップサイズでは反復点が最小点に落ち着かない。

Figure 9.1 は、同一の 2 次元の谷におけるこれら両方の障害を示している。現代のオプティマイザにおけるあらゆる設計上の選択は、次の 3 つの制御軸を通じてそのいずれかに対処する。

- **ステップ方向：**モメンタム（9.4 節）、Adam や AdamW のような座標ごとの適応的手法（9.5 節）、および Muon のような行列レベルで直交化する手法（9.6 節）は、生の勾配を前処理（preconditioning）や平滑化を施した方向に置き換えることで、悪条件（ill-conditioned）な谷に沿ってより速く進む。
- **ステップサイズスケジュール：**学習率のウォームアップ、コサイン減衰や線形減衰、およびウォームアップ・安定・減衰（warmup–stable–decay、WSD）スケジュール（9.7 節）は、序盤の速やかな移動と終盤の低いノイズとの間のトレードオフを調整する。
- **ミニバッチサイズとスケーリング：**臨界バッチサイズ（critical batch size、9.8 節）は、ミニバッチを大きくしても最適化ステップ数がそれ以上減らなくなる境界を定める。一方、最大更新パラメータ化（maximal update parametrization、μP、9.9 節）を用いると、小規

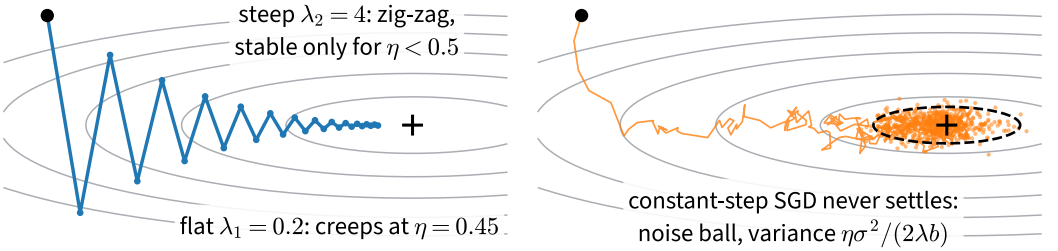


Figure 9.1: 谷を横切る方向の曲率が谷に沿う方向の 20 倍大きい谷（$\kappa = 20$）における 2 つの障害。左：勾配降下法は急な方向をジグザグに行き来しつつ、平坦な方向には少しずつしか進まない。右：ノイズを含む勾配と一定のステップサイズのもとでは、反復点は最小点に到達するものの決して落ち着かず、ノイズ球（破線の楕円：2 標準偏差）の内部をさまよい続ける。

模なプロキシモデルで調整した学習率をはるかに幅の広いネットワークへそのまま転移できる。

本章では各手法を展開するにあたり、条件数（condition number）$\kappa = 20$ の 2 次元の二次関数の谷という解析モデルと、9.3 節で導入する 2 つのニューラルネットワークベンチマーク（畳み込み分類器 `FashionCNN` とバイトレベルの Transformer 言語モデル `TinyLM`）での体系的な実証計測とを組み合わせる。最後に 9.10 節では、勾配クリッピング、重み平均化、ハイパーパラメータのスケーリングを統合し、大規模な訓練の実行を設定・診断するためのエンドツーエンドのプロトコルを構築する。

本章では、勾配、対称行列の固有値、および基本的なミニバッチ SGD（2.1.2.3 節）に関する知識を前提とする。ハイパーパラメータの選択に直結する箇所では簡潔な導出を示し、より長い数学的証明は第 35 章に譲る。教科書、論文、および実務者向けプレイブックに関する注釈付きの参考文献は 9.11 節にまとめている。

9.1 ランドスケープと勾配降下法

モメンタム、適応的な前処理（preconditioning）、学習率スケジュールといった違いはあるものの、深層学習における標準的なオプティマイザはすべて、損失の勾配を評価してその逆方向へ進むという同じ基本操作を反復している。では、なぜその 1 ステップがある目的関数では急速に前進し、別の目的関数では這うようにしか進まないのだろうか。また、学習率（learning rate）をあるしきい値より大きくすると、なぜ突然発散してしまうのだろうか。

d 個のパラメータをもつネットワークは、経験リスク（empirical risk）$f : \mathbb{R}^d \to \mathbb{R}$ を d 次元のパラメータベクトル $\mathbf{x}$ 上に定める。汎化（generalization）は最終的に未知データに対する期待リスクに依存するものの（2.6.1 節）、最適化アルゴリズムの主たる役割は、経験目的関数 $f(\mathbf{x})$ を可能な限り速く確実に減少させることにある。従来の非凸最適化の説明では、質の悪い極小点に陥る危険性が強調されることが多い。しかし高次元のニューラルネットワークでは、訓練ダイナミクスを支配するのは別の 2 つの幾何学的特徴、すなわち**悪条件性**（ill-conditioning、方向によって曲率が大きく異なること）と**鞍点**（saddle point）である。

本節ではまず、悪条件な 2 次元の二次関数の谷における勾配降下法（gradient descent）を解析し、曲率、安定な最大学習率、反復回数の間の厳密な関係を導く。次に、高次元の損失ランドスケープ（loss landscape）における臨界点（critical point）・鞍点・プラトーを調べ、滑らかな関数に対する一般的な降下保証を証明する。さらに、学習率が極小点をその鋭さ（sharpness）

によって選別する仕組み（これがニューラルネットワークの訓練における**安定性の縁**（edge of stability）につながる）を示し、悪条件性への対処法として**前処理**を導入する。

9.1.1 最適化ランドスケープと条件数

不均一な曲率の影響を切り分けるため、次の 2 次元の二次目的関数を考えよう。

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2, \tag{9.1}$$

この関数の唯一の最小点は原点にある。二階導関数（曲率）は $\lambda_1 = 0.2$（平坦な軸 x_1 方向）および $\lambda_2 = 4$（急峻な軸 x_2 方向）であり、細長い楕円状の等位集合（level set）が狭い谷を形成している。

学習率 $\eta > 0$ の勾配降下法は、反復点を $\mathbf{x} \leftarrow \mathbf{x} - \eta\nabla f(\mathbf{x})$ により更新する。$\nabla f(\mathbf{x}) = (0.2x_1, 4x_2)$ であるため、2 つの座標は互いに分離し、各ステップで乗法的に発展する。

$$\begin{aligned} x_1 &\leftarrow (1 - 0.2\eta)\, x_1, \\ x_2 &\leftarrow (1 - 4\eta)\, x_2. \end{aligned}$$

それぞれの座標は、η に対して相反する制約を課す。

- **安定性の上限（急峻な方向）：**急峻な座標 x_2 が縮小するのは $|1 - 4\eta| < 1$ のとき、かつそのときに限り、ここから厳格な安定性の上限 $\eta < 0.5$ が課される。
- **収束の速さ（平坦な方向）：**安定な任意のステップサイズ $\eta < 0.5$ に対して、平坦な方向の縮小率は $1 - 0.2\eta > 0.9$ を満たす。つまり、各ステップで取り除かれるのは残存誤差の 10% 未満（x_1 方向）である。

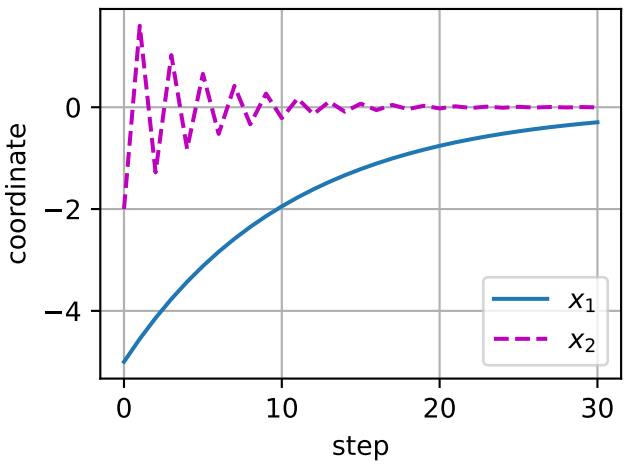
谷の目的関数 `f_valley`、その勾配 `valley_grad`、2 次元の軌跡実行関数 `train_2d`（初期点 $(-5, -2)$）、および通常の勾配降下法の 1 ステップ `gd_2d` を `d2l` に保存し、後続の節ですべてのオプティマイザをまったく同じ曲面上でベンチマークできるようにしておく。

```
VALLEY_CURVATURES = (0.2, 4.0)
h1, h2 = VALLEY_CURVATURES  # Curvatures of the valley along x1 and x2

def f_valley(x1, x2):
    """The ill-conditioned quadratic valley h1/2 x1^2 + h2/2 x2^2, with
    (h1, h2) = VALLEY_CURVATURES."""
    h1, h2 = VALLEY_CURVATURES
    return h1 / 2 * x1 ** 2 + h2 / 2 * x2 ** 2

def valley_grad(x1, x2):
    """Gradient of f_valley."""
    h1, h2 = VALLEY_CURVATURES
    return h1 * x1, h2 * x2

def train_2d(trainer, f_grad, steps=20, **hparams):
    """Run a 2D update rule from (-5, -2) and return the iterates."""
    x1, x2, s1, s2 = -5, -2, 0, 0  # s1, s2: state of stateful optimizers
    results = [(x1, x2)]
    for _ in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad, **hparams)
```

```
        results.append((x1, x2))
    print(f'step {steps}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def gd_2d(x1, x2, s1, s2, f_grad, eta):
    """One step of gradient descent in 2D; the state stays zero."""
    g1, g2 = f_grad(x1, x2)
    return x1 - eta * g1, x2 - eta * g2, 0, 0
```

安定性の上限のわずかに下である $\eta = 0.45$ では、急峻な方向の乗数は $1 - 4(0.45) = -0.8$ となるため、x_2 は谷底を行き過ぎて毎ステップ符号を反転させながら、絶対値で 20% ずつ縮小する。一方、平坦な方向の乗数は $1 - 0.2(0.45) = 0.91$ であるため、30 ステップを $(-5, -2)$ から進めた後でも、平坦な座標は依然として $-5 \times 0.91^{30} \approx -0.3$ 付近にとどまる。

```
h = np.array([0.2, 4.0])  # Curvatures h1, h2 along x1 and x2

def gd_valley(eta, steps, x0=(-5.0, -2.0)):
    """Gradient descent on the valley; returns all iterates."""
    xs = [np.array(x0)]
    for _ in range(steps):
        xs.append(xs[-1] - eta * h * xs[-1])  # The gradient is h * x
    return np.stack(xs)

xs = gd_valley(0.45, steps=30)
d2l.plot(np.arange(31), [xs[:, 0], xs[:, 1]], 'step', 'coordinate',
         legend=['$x_1$', '$x_2$'])
```



η を $(0, 0.5]$ の範囲で変化させると、全体の収束時間（両方の座標を原点からの初期距離の 1% 以内に収めるまでに要するステップ数で測る）が、これら 2 つの因子のバランスにどう依存するかがわかる。

```
print('  eta   factor x1   factor x2   steps to 1%')
for eta in [0.1, 0.2, 0.3, 0.4, 0.45, 2 / 4.2, 0.49, 0.5]:
```

```
    xs = gd_valley(eta, steps=1000)
    done = np.all(np.abs(xs) <= 0.01 * np.abs(xs[0]), axis=1)
    steps = int(np.argmax(done)) if done.any() else 'never'
    print(f'{eta:5.3f}   {1 - 0.2 * eta:9.3f}   {1 - 4 * eta:9.3f}   '
          f'{steps:>11}')
```

```
  eta   factor x1   factor x2   steps to 1%
0.100       0.980       0.600           228
0.200       0.960       0.200           113
0.300       0.940      -0.200            75
0.400       0.920      -0.600            56
0.450       0.910      -0.800            49
0.476       0.905      -0.905            47
0.490       0.902      -0.960           113
0.500       0.900      -1.000         never
```

最適なステップサイズは平坦な方向と急峻な方向の乗数の絶対値を等しくする条件 $1-0.2\eta = -(1-4\eta)$ から求まり、$\eta^\star = 2/4.2 \approx 0.476$ となる。このとき両方の因子の絶対値は 0.905 となり、1% への収束には 47 ステップを要する。安定性のしきい値付近における急激な非対称性に注目しよう。$\eta^\star$ よりわずかに小さい $\eta = 0.45$ に下げてもステップ数は 2 ステップ増えるだけ（49 ステップ）であるのに対し、わずかに大きい $\eta = 0.49$ に上げるとステップ数は 2 倍以上に跳ね上がる（113 ステップ）。

この解析は、任意の d 次元の正定値二次関数 $f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top \mathbf{H}\mathbf{x}$（対称なヘッセ行列 $\mathbf{H} \succ \mathbf{0}$ をもつ）へ直ちに拡張できる。$\mathbf{x} = \sum_{i=1}^{d} c_i \mathbf{v}_i$ と $\mathbf{H}$ の正規直交固有基底（$\mathbf{H}\mathbf{v}_i = \lambda_i \mathbf{v}_i$ かつ $0 < \lambda_{\min} = \lambda_1 \le \cdots \le \lambda_d = \lambda_{\max}$）で展開すると、勾配降下法のステップは各固有ベクトルに沿って分離する。

$$(\mathbf{I} - \eta \mathbf{H}) \sum_{i=1}^{d} c_i \mathbf{v}_i = \sum_{i=1}^{d} (1 - \eta\lambda_i)\, c_i \mathbf{v}_i. \tag{9.2}$$

安定であるためには $|1-\eta\lambda_i| < 1$ がすべての i について成り立つ必要があり、これは最大固有値によって決まる。

$$\eta < \frac{2}{\lambda_{\max}}. \tag{9.3}$$

この上限のもとでは、最も平坦な固有ベクトル $\mathbf{v}_1$ に沿った進み具合は、次の**条件数**（condition number）のみに依存する。

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{9.4}$$

目的関数 f を定数 $c > 0$ 倍に再スケーリングすると、すべての固有値は c 倍になり、安定な学習率は c で割られるため、軌跡は不変に保たれる。無次元の比 κ こそが、曲面の本質的な幾何学的難しさを捉えている。両端の縮小率をつり合わせる条件 $1-\eta\lambda_{\min} = -(1-\eta\lambda_{\max})$ から、最適な定数学習率 $\eta^\star = 2/(\lambda_{\min}+\lambda_{\max})$ と、1 ステップあたりの最悪ケースの縮小率 $(\kappa-1)/(\kappa+1) \approx 1 - 2/\kappa$ が得られる（35.1.2 節）。したがって、誤差を ϵ 倍に減らすのに必要なステップ数は次式となる。

$$t \approx \frac{\kappa}{2} \log \frac{1}{\epsilon} \tag{9.5}$$

この谷（$\kappa = 4/0.2 = 20$、$\epsilon = 0.01$）に対して、この式は $10 \log 100 \approx 46$ ステップを予測し、上で観測された 47 ステップと一致する。t は κ に比例して増大するため、いかなるスカラーの学習率を用いても悪条件性を克服することはできない。収束を加速するには、更新方向そのものを変える必要がある。

9.1.2 臨界点、鞍点、プラトー

凸二次関数の椀型の曲面とは異なり、ニューラルネットワークの損失ランドスケープは非凸であり、$\nabla f(\mathbf{x}) = \mathbf{0}$ となる多数の**臨界点**や、$\|\nabla f(\mathbf{x})\|$ が極めて小さくなる**プラトー**をもつ。2 回微分可能な関数 $f : \mathbb{R}^d \to \mathbb{R}$ の任意の臨界点において、ヘッセ行列 $\mathbf{H} = \nabla^2 f(\mathbf{x})$ の固有値は局所的な幾何構造を次のように分類する。

- **極小点**：d 個の固有値がすべて真に正である（$\mathbf{H} \succ \mathbf{0}$）。
- **極大点**：d 個の固有値がすべて真に負である（$\mathbf{H} \prec \mathbf{0}$）。
- **鞍点**：$\mathbf{H}$ が正と負の両方の固有値をもち、ある固有ベクトルに沿っては上向きに曲がり、別の固有ベクトルに沿っては下向きに曲がる。原点における $f(x, y) = x^2 - y^2$ がその例である（Figure 9.2）。
- **退化臨界点**：$\mathbf{H}$ が 0 の固有値をもち、かつ異符号の固有値をもたない点（degenerate critical point）であり、極値点（x^4 など）か変曲点（x^3 など）かは高階の導関数によって決まる。

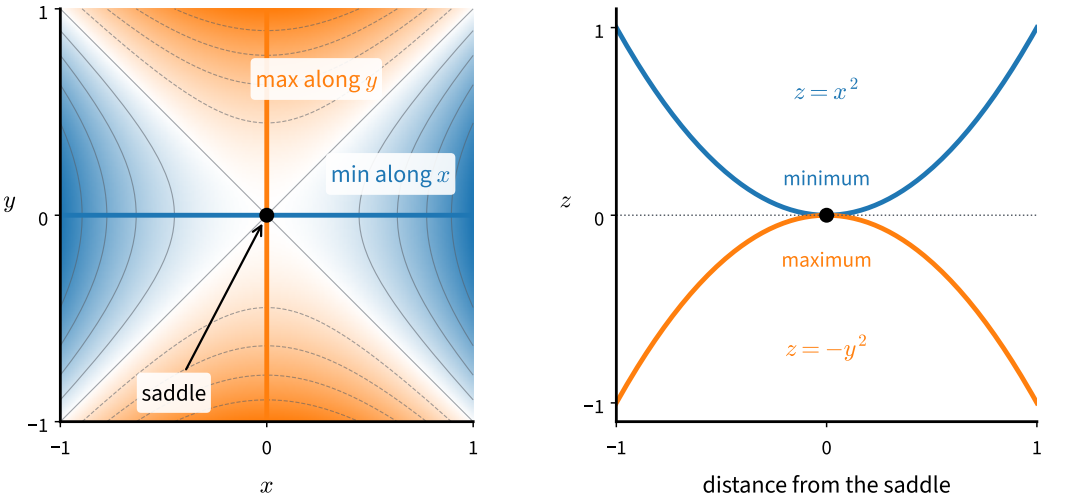


Figure 9.2: 鞍面 $z = x^2 - y^2$ は x 方向に極小、y 方向に極大をもち（左）、原点を通る 2 つの断面にもそれが現れている（右）。

高次元では、鞍点の数が極小点を圧倒的に上回る。臨界点が極小点であるためには、d 個のヘッセ行列の固有値がすべて正でなければならない。単純なランダム符号のヒューリスティックのもとでは、d 個すべての固有値が正となる確率は 2^{-d} にすぎない（またランダム行列理論によれば、基準となる損失が十分に低くなるまでは、対称ランダム行列の固有値は 0 の周りに均等に分かれることが確認されている。演習問題 4 を参照）。ニューラルネットワークにおけるヘッセ行列

の実測でも、高い損失値にある臨界点は負の固有値を多数もつ鞍点で占められており、負の固有値の割合が 0 に近づくのは低い損失値の領域に限られることが確かめられている (Dauphin et al. 2014)。

重要なことに、**狭義鞍点**（strict saddle point、少なくとも 1 つの負の固有値 $\lambda < 0$ をもつ臨界点）は勾配降下法を閉じ込めない。曲率 $-\mu < 0$ で下向きに曲がる固有ベクトルに沿っては、勾配ステップのたびに変位が $(1 + \eta\mu) > 1$ 倍に増幅される。狭義鞍点へ収束するような初期値の安定多様体（stable manifold）は測度 0 であるため、ランダム初期化からの勾配降下法は確率 1 で鞍点から脱出する (Lee et al. 2016)。尾根から距離 δ の位置から出発すると、不安定方向に沿って距離 1 に達するまでのステップ数は $\log(1/\delta)/\log(1+\eta\mu) \approx \log(1/\delta)/(\eta\mu)$ となる。これは δ については対数的であるが、負の曲率の大きさ μ には反比例する。さらに、ミニバッチ勾配のノイズは $\delta = 0$ の場合であっても反復点を尾根から積極的に弾き出す（演習問題 1）。

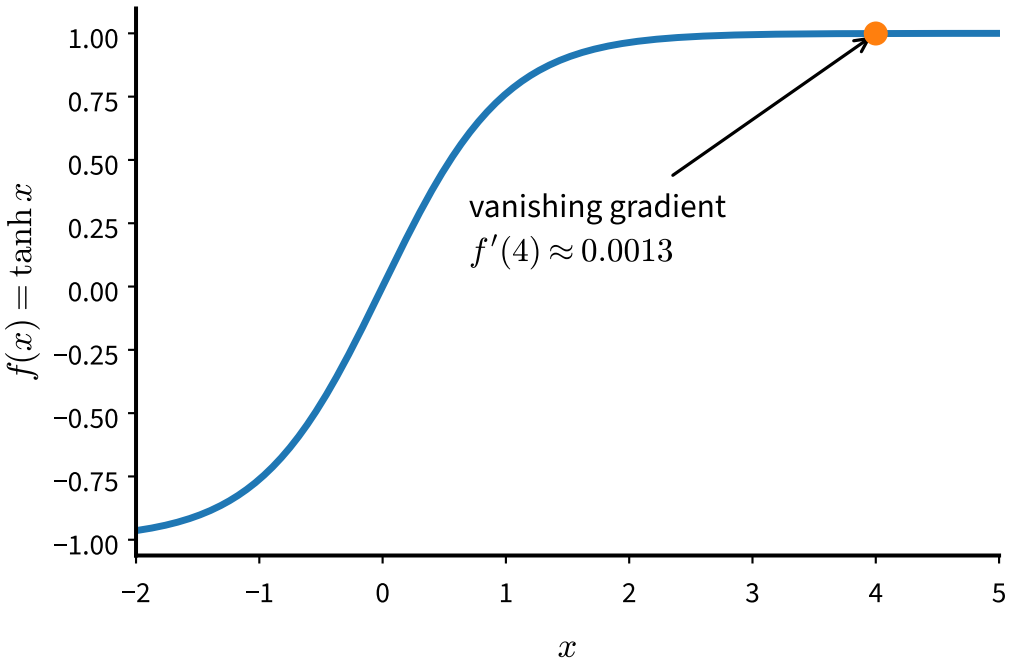


Figure 9.3: $f(x) = \tanh x$ を $x = 4$ から最小化すると、臨界点が一切ないにもかかわらず、$f'(4) \approx 0.0013$ となる活性化プラトー上で停滞する。

対照的に、**プラトー**は活性化関数の導関数が飽和するとき、臨界点がなくても生じる。たとえば $f(x) = \tanh(x)$ の $x = 4$ 付近（Figure 9.3）では、$f'(4) = 1 - \tanh^2(4) \approx 0.0013$ であるため、$\eta = 1$ の勾配降下法は 370 ステップもかけて $x = 4$ から $x = 2$ まで這うように進むことになる。深層ネットワークにおいて、このような勾配消失（4.4 節）は主にアーキテクチャの設計、すなわち非飽和型の活性化関数（4.1.4 節）、正規化層（7.3 節）、残差接続（7.4 節）によって解消されており、オプティマイザにとっての中心的な課題として残るのは悪条件性である。

ニューラルネットワークの損失は大域的には非凸であるものの（たとえば、ある層の d 個の隠れユニットを並べ替えると同一の損失をもつ $d!$ 個の異なるパラメータベクトルが得られ、それらの平均はより損失の高い単一ユニットのネットワークに潰れてしまう。演習問題 3）、任意の反復点の周りにおける局所的な幾何構造はそのヘッセ行列のスペクトルによって決まる。深層ネットワークの実測スペクトルは、0 付近に密集する膨大な固有値のバルクと、少数の大きな正の外れ値固有値を示す (Ghorbani et al. 2019)。**鋭さ** $\lambda_{\max}$ と呼ばれる最大固有値が安定な学習率の上限を

決める一方、平坦なバルクは $\eta\lambda_i \ll 1$ の速さでしか進まず、高次元空間において狭い谷のボトルネックが再現される。

9.1.3 降下保証と極小点の安定性

非二次の目的関数では、曲率は場所ごとに変化する。それにもかかわらず、なぜ $2/\lambda_{\max}$ という規則が依然としてステップサイズの安定性を支配するのだろうか。

1 次のテイラー展開 $f(\mathbf{x}+\epsilon) \approx f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x})$ とコーシー・シュワルツの不等式により、負の勾配 $-\nabla f(\mathbf{x})$ は局所的な最急降下方向となる。有限のステップ $\mathbf{x} - \eta \nabla f(\mathbf{x})$ が実際に f を減少させることを保証するため、f が L **滑らか**（L-smooth）、すなわちその勾配が L リプシッツ連続であると仮定しよう（1 変数の場合は $|f'(y) - f'(x)| \le L|y-x|$、あるいは $|f''(x)| \le L$）。ステップ ϵ に沿って傾きの差を積分すると、f は曲率 L の接する放物線によって上から抑えられる。

$$
\begin{aligned}
f(x+\epsilon) - f(x) - \epsilon f'(x) &= \int_0^{\epsilon} \big(f'(x+s) - f'(x)\big)\, ds \\
&\le \int_0^{\epsilon} L s \, ds = \frac{L}{2}\epsilon^2 .
\end{aligned}
$$

ここに勾配ステップ $\epsilon = -\eta f'(x)$（あるいは $\epsilon = -\eta \nabla f(\mathbf{x})$、すなわち d 次元で $\|\nabla^2 f(\mathbf{x})\|_2 \le L$ の場合。35.1.1 節）を代入すると、次の**降下補題**（descent lemma）が得られる。

$$
f(\mathbf{x} - \eta \nabla f(\mathbf{x})) \le f(\mathbf{x}) - \eta\Big(1 - \frac{\eta L}{2}\Big)\|\nabla f(\mathbf{x})\|^2 . \tag{9.6}
$$

$0 < \eta < 2/L$ である限り、臨界点以外の任意の点で各勾配ステップは f を真に減少させ、1 ステップあたりの保証減少量は $\|\nabla f(\mathbf{x})\|^2/(2L)$（$\eta = 1/L$ のとき）で最大となる。

ここで視点を逆転させ、曲率の異なる複数の極小点をもつ損失ランドスケープ上で学習率 η を固定してみよう。任意の極小点 $x^\star$（曲率 $\lambda = f''(x^\star) > 0$）の近傍では、1 回の勾配ステップによって微小な変位 $x - x^\star$ が $1 - \eta\lambda$ 倍になる。その結果、次のことが成り立つ。

- $\lambda < 2/\eta$ ならば $|1 - \eta\lambda| < 1$ となり、この極小点は**線形安定**（linearly stable）である（近傍の反復点を引き寄せる）。
- $\lambda > 2/\eta$ ならば $1 - \eta\lambda < -1$ となり、この極小点は**不安定**である（厳密に $x^\star$ 上に着地しない限り、あらゆる反復点を弾き出す）。

したがって、学習率 η は極小点を深さではなく**鋭さ**によって選別する。すなわち、勾配降下法が収束できるのは鋭さが $\lambda_{\max} \le 2/\eta$ を満たす極小点に限られる。この選別は $f(x) = x\cos(0.15\pi x)$ 上で直接観察できる。この関数は $x = 7.27$ 付近に深くより鋭い極小点（$f'' \approx 1.81$）をもち、$x = -1.83$ 付近により浅く平坦な極小点（$f'' \approx 0.98$）をもつ（Table 9.1 にまとめる）。

Table 9.1: 局所曲率 f'' と 1 ステップあたりの変位の乗数 $1 - \eta f''$（$f(x) = x\cos(0.15\pi x)$ の 2 つの極小点における値）。

極小点	曲率 f''	$\eta = 0.5$ での乗数	$\eta = 2$ での乗数
深い極小点（$x = 7.27$）	1.81	0.09（安定）	-2.62（不安定）
浅い極小点（$x = -1.83$）	0.98	0.51（安定）	-0.96（安定）

まず、1 次元の勾配降下法を実行する関数 gd と軌跡を描画する関数 show_trace を定義する。

```
def gd(eta, f_grad, x=10.0, steps=10):
    """Gradient descent from x; returns all iterates."""
    xs = [x]
    for _ in range(steps):
        x = x - eta * f_grad(x)
        xs.append(float(x))
    return xs

def show_trace(f, *runs, legend=None):
    """Draw the iterates of one or more runs on the graph of f."""
    n = max(abs(x) for xs in runs for x in xs)
    grid = d2l.arange(-n, n, 0.01)
    d2l.plot([grid, *runs], [f(grid), *[[f(x) for x in xs] for xs in runs]],
             '$x$', '$f(x)$', legend=legend, fmts=['-', '-o', '-s'])
```

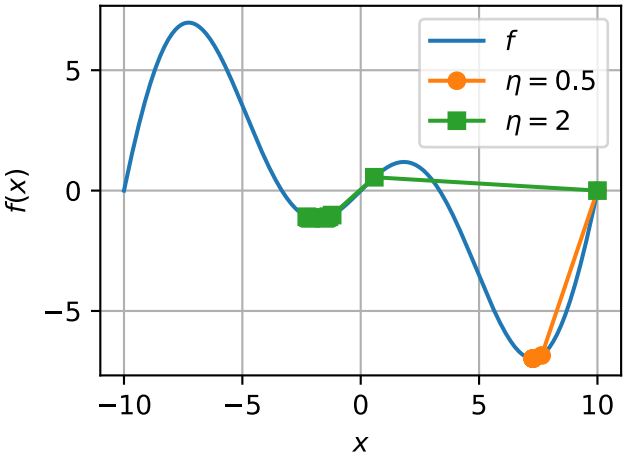
次に、$f(x) = x\cos(0.15\pi x)$ とその一階導関数 f_cos_grad および二階導関数 f_cos_hess を定義する。

```
c = d2l.tensor(0.15 * np.pi)

def f_cos(x):  # Objective function
    return x * d2l.cos(c * x)

def f_cos_grad(x):  # Its derivative
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

def f_cos_hess(x):  # Its second derivative
    return -2 * c * d2l.sin(c * x) - c**2 * x * d2l.cos(c * x)
```

2 つの極小点で f_cos_hess を評価し、gd を $x = 10$ から $\eta \in \{0.5, 2\}$ で実行すると、安定性の上限 $2/\eta$ が深く鋭い極小点と浅く平坦な極小点のどちらを選ぶかがよくわかる。

```
minima = [gd(0.5, f_cos_grad, x=x, steps=50)[-1] for x in (10.0, 0.0)]
print(', '.join(f"f''({x:.2f}) = {float(f_cos_hess(x)):.2f}" for x in minima))
runs = {eta: gd(eta, f_cos_grad, steps=100) for eta in (0.5, 2)}
for eta, xs in runs.items():
    print(f'eta {eta}, 2/eta = {2/eta:.0f}: x = {xs[10]:.3f} after 10 steps, '
          f'{xs[100]:.3f} after 100')
show_trace(f_cos, *[xs[:11] for xs in runs.values()],
           legend=['$f$', r'$\eta = 0.5$', r'$\eta = 2$'])
```

```
f''(7.27) = 1.81, f''(-1.83) = 0.98
eta 0.5, 2/eta = 4: x = 7.269 after 10 steps, 7.269 after 100
eta 2, 2/eta = 1: x = -1.528 after 10 steps, -1.821 after 100
```



$x = 10$ から出発すると、$\eta = 0.5$（$2/\eta = 4 > 1.81$）での実行は $x = 7.269$ の深い極小点へ急速に収束する。一方、$\eta = 2$（$2/\eta = 1 < 1.81$）では深い極小点が不安定（$1 - \eta f'' = -2.62$）となり、反復点は代わりに $x = -1.83$ 付近のより平坦な極小点（$f'' = 0.98 < 2/\eta$）へと落ち着く。

9.1.4 安定性の縁

ニューラルネットワークの訓練では、鋭さ $\lambda_{\max}$ と安定性の限界 $2/\eta$ の関係は動的なものになる。Cohen et al. (2021) は、フルバッチ勾配降下法のもとで損失が下がるにつれてヘッセ行列の鋭さ $\lambda_{\max}(\nabla^2 f(\mathbf{x}_t))$ が着実に上昇し（**漸進的鋭化**（progressive sharpening））、やがて安定性のしきい値 $2/\eta$ に達することを示した。その後、訓練は発散するのではなく、残りの訓練期間を通じて鋭さが $2/\eta$ 付近にとどまり、損失は非単調に減少し続ける。この動作領域は**安定性の縁**として知られている。

この現象は、パラメータ数 25 の単一隠れ層 tanh ネットワークを 16 個の点（$y = \sin(3x)$、区間 $[-2, 2]$）に当てはめる実験で直接測定できる。$d = 25$ であるため、厳密な 25×25 のヘッセ行列とその最大固有値 $\lambda_{\max}$ を 500 ステップごとに、10 個のランダム初期化にわたって $\eta = 0.4$（$2/\eta = 5$）および $\eta = 0.25$（$2/\eta = 8$）のそれぞれについて計算できる。

```
X = torch.linspace(-2, 2, 16)
Y = torch.sin(3 * X)

def mlp_loss(p):  # 8 hidden units: weights, biases, output weights, bias
    W1, b1, W2, b2 = p[:8], p[8:16], p[16:24], p[24]
    return 0.5 * ((torch.tanh(X[:, None] * W1 + b1) @ W2 + b2 - Y) ** 2).mean()

def sharpness(P):  # Largest eigenvalue of the Hessian, for each row of P
    H = torch.func.vmap(torch.func.hessian(mlp_loss))(P)
    return torch.linalg.eigvalsh(H)[:, -1]

def gd_sharpness(eta, P, steps=20000, every=500, last=1000):
    """Full-batch gradient descent from each row of P. Returns the sharpness
    every `every` steps, the final loss, and the fraction of the last `last`
    steps on which the loss rose."""
    grad_and_loss = torch.func.vmap(torch.func.grad_and_value(mlp_loss))
```

```
        sharp, losses = [], []
        for k in range(steps + 1):
            if k % every == 0:
                sharp.append(sharpness(P))
            g, loss = grad_and_loss(P)
            losses.append(loss)
            if k < steps:  # The last pass only measures
                P = P - eta * g
        losses = torch.stack(losses[-last - 1:])  # (last + 1, starts)
        return (torch.stack(sharp, 1), losses[-1],
                (losses[1:] > losses[:-1]).float().mean(0))

    P0 = 0.5 * torch.randn(10, 25)  # Ten random starts, shared by both runs
```

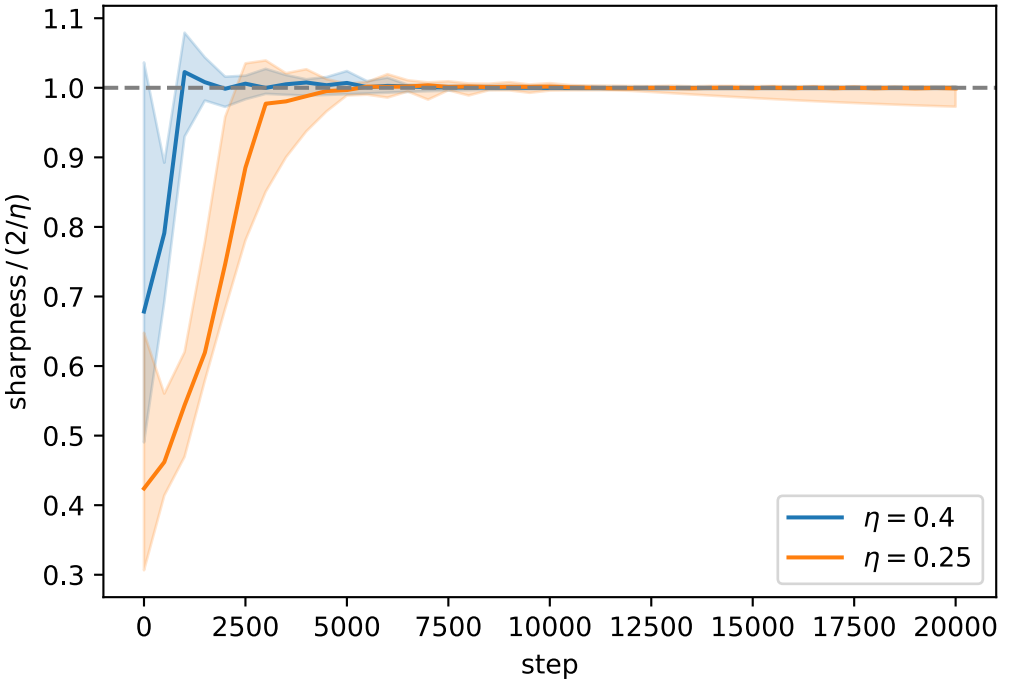
関数 gd_sharpness を共通の 10 通りの初期化 P0 から $\eta = 0.4$ $(2/\eta = 5)$ および $\eta = 0.25$ $(2/\eta = 8)$ で実行し、初期および最終の鋭さ、最終損失、ならびに終盤のステップのうち損失が上昇した割合を表にまとめよう。

```
    runs = {eta: gd_sharpness(eta, P0) for eta in (0.4, 0.25)}
    print('      start      |' + '|'.join(
        f'    eta {eta}, 2/eta = {2 / eta:.0f}     ' for eta in runs))
    print('sharpness   loss |' + '|'.join([' sharpness    loss  rising '] * 2))
    for i in range(len(P0)):
        print(f'{float(runs[0.4][0][i, 0]):9.2f} {float(mlp_loss(P0[i])):6.3f} |'
              + '|'.join(f' {float(s[i, -1]):9.2f} {float(l[i]):7.1e} '
                         f'{float(r[i]):6.0%} ' for s, l, r in runs.values()))
```

```
      start      |    eta 0.4, 2/eta = 5     |    eta 0.25, 2/eta = 8
sharpness   loss | sharpness    loss  rising | sharpness    loss  rising
     2.92  0.540 |      5.00 3.3e-04    50% |      8.00 3.5e-04    50%
     2.45  0.257 |      5.00 1.9e-03    50% |      7.78 1.0e-04     0%
     5.18  0.669 |      4.99 3.9e-04    50% |      7.98 1.4e-04    50%
     3.46  0.323 |      4.99 4.1e-04    50% |      7.98 2.9e-04    49%
     3.33  0.736 |      4.99 4.2e-04    50% |      8.01 2.0e-04    50%
     4.62  0.805 |      4.99 3.9e-04    50% |      8.02 1.7e-04    50%
     2.95  1.170 |      5.01 3.2e-04    50% |      7.98 7.4e-05     0%
     3.98  1.408 |      5.00 3.0e-04     0% |      8.00 3.9e-04    50%
     4.06  0.706 |      5.00 2.9e-04    50% |      8.00 2.3e-04     1%
     3.19  0.376 |      5.00 2.7e-04    50% |      7.98 4.4e-05     0%
```

安定性のしきい値で正規化した鋭さ $\lambda_{\max}/(2/\eta)$ を全 20,000 ステップにわたってプロットすると、どちらの学習率でも鋭さが 1 の安定性の縁に向かって上昇していく様子がわかる。

```
    curves = {rf'$\eta = {eta}$': eta / 2 * s for eta, (s, _, _) in runs.items()}
    d2l.plot_curves(curves, 'step', r'sharpness$\,/\,(2/\eta)$',
                    x=np.arange(0, 20001, 500), reference=1)
```



まったく同じ初期化から出発しても、正規化された鋭さ $\lambda_{\max}/(2/\eta)$ は 1 まで上昇してそこで頭打ちになる。すなわち、$\eta = 0.4$ の実行は鋭さ $\lambda_{\max} \approx 5.0$ で停止する一方、$\eta = 0.25$ の大半の実行は $\lambda_{\max} \approx 8.0$ まで上昇する。すべての実行がそこに到達するわけではなく、$\eta = 0.25$ ではいくつかの初期値が鋭さ 8 に達する前にデータに適合し、しきい値のわずかに下にとどまったまま、損失が上昇するステップをほとんど伴わずに降下する。しきい値 $2/\eta$ に達した実行では、損失は典型的に約 50% のステップで上昇する。これは、反復点が最も鋭い固有ベクトルに沿って振動する（$1 - \eta\lambda_{\max} \approx -1$）一方で、残りの 24 個のより平坦な方向に沿っては降下し続けるためである。$\lambda_{\max}$ が一時的に $2/\eta$ を超えるたびに、最も鋭い固有ベクトルに沿った振動の増大が損失の 3 階導関数と結合してより平坦な領域への復元ドリフトを引き起こし、$\lambda_{\max}$ を $2/\eta$ に安定化させる (Damian et al. 2023)。この自己安定化は、訓練開始時に η を徐々に引き上げる手法（**学習率のウォームアップ**、9.7 節）によって、ピーク学習率に耐えられるほど平坦な領域へとネットワークが誘導される理由も説明している (Kalra and Barkeshli 2024)。

この実験ではフルバッチ勾配降下法を用いたが、本章の残りの節ではモメンタム、Adam、およびミニバッチを用いて訓練する。安定性の縁はこれら 3 つすべてにおいても保たれ、それぞれ異なるしきい値をもつ。係数 β のヘビーボールモメンタム（9.4 節）は、安定性の限界を $\eta\lambda_{\max} < 2$ から $\eta\lambda_{\max} < 2(1+\beta)$ へと引き上げ、それに応じて鋭さも $2(1+\beta)/\eta$ に落ち着く (Cohen et al. 2021)。Adam（9.5 節）はステップの各座標をそれぞれの勾配スケールで割るため、定常値に落ち着く量は同じ再スケーリングを施した後のヘッセ行列の鋭さ、すなわち**前処理後の鋭さ**（preconditioned sharpness）となる。Adam で通常用いられるモメンタム係数 $\beta_1 = 0.9$ の場合、そのしきい値は約 $38/\eta$ である (Cohen et al. 2022)。ミニバッチ勾配を用いると、鋭さは $2/\eta$ より手前で上昇を止め、バッチが小さいほどより早い段階で止まる。いずれの場合も、学習率が大きいほど鋭さの上限は低くなるため、ウォームアップに関する議論はこれ以降で用いるオプティマイザにもそのまま当てはまる。

9.1.5 ニュートン法と前処理

9.1.5.1 二次ステップとニュートン法

なぜ勾配降下法は谷 Equation 9.1 で苦戦するのだろうか。点 $(-5, -2)$ において、負の勾配 $(1, 8)$ は原点 $(0, 0)$ ではなく、ほぼ真横の急峻な壁を向いている。急峻な座標 x_2 が 1 ステップで谷底に到達するには $\eta = 1/4 = 0.25$ が必要であるのに対し、平坦な座標 x_1 には $\eta = 1/0.2 = 5$ が必要となる。

ニュートン法（Newton's method）は、$\mathbf{x}$ の周りの 2 次テイラー近似を最小化することで、各方向にその局所曲率の逆数を与える。

$$f(\mathbf{x} + \epsilon) \approx f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2} \epsilon^\top \mathbf{H} \epsilon. \tag{9.7}$$

ϵ に関する勾配を 0 とおくと、ニュートン法の更新式が得られる。

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}^{-1} \nabla f(\mathbf{x}). \tag{9.8}$$

任意の正定値二次関数では 2 次展開が厳密に一致するため、条件数 κ にかかわらず 1 回のニュートンステップで最小点に到達する。一般の滑らかな関数の正則な極小点の近傍でも、ニュートン法は**二次収束**（quadratic convergence）を達成し、ステップごとに有効桁数が倍増する（35.2 節）。

このような高速な局所収束性にもかかわらず、厳密なニュートンステップは次の 3 つの理由から深層ニューラルネットワークには非現実的である。

1. **計算コスト：**ヘッセ行列・ベクトル積（34.5.3.2 節）を用いて $d \times d$ のヘッセ行列（要素数 10^{18} 個、$d = 10^9$ の場合）の保存を回避したとしても、$\mathbf{H}\epsilon = -\nabla f(\mathbf{x})$ を反復法で解くには 1 回のパラメータ更新あたり数十回の誤差逆伝播法の実行が必要になる。
2. **負の曲率：**$\mathbf{H}$ はすべての臨界点に対して対称に作用するため、ダンピングや絶対値スペクトルへの射影による修正（35.2.1 節）を加えない限り、負の固有値がステップ方向を反転させ、反復点を鞍点や極大点に向かって上り方向へと駆動してしまう。
3. **ミニバッチノイズへの感度：**0 付近のノイズを含んだ曲率推定値の逆数をとると、サンプリングノイズが壊滅的に増幅される。

9.1.5.2 対角前処理

実用的な深層学習のオプティマイザは、ヘッセ行列全体の逆行列を求める代わりに、各座標（または層の行列）を安価な曲率推定値で再スケーリングするという二次手法の手頃な中核部分だけを残す。一般に、$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathbf{P}^{-1} \nabla f(\mathbf{x})$ の形をとる更新は**前処理付き勾配降下法**（preconditioned gradient descent）と呼ばれ、ここで**前処理行列**（preconditioner）$\mathbf{P}$ は $\mathbf{P}^{-1/2} \mathbf{H} \mathbf{P}^{-1/2}$ の有効条件数を小さくする役割を果たす。

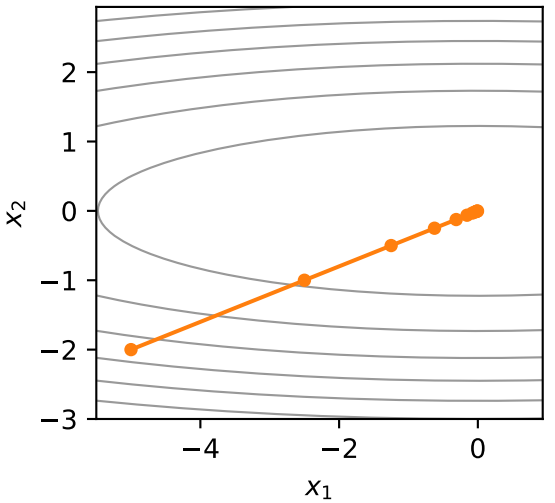
最も単純な選択は**対角前処理行列**（diagonal preconditioner）$\mathbf{P} = \operatorname{diag}(\mathbf{H})$ であり、勾配の各座標をそれぞれの二階導関数で割る。

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \operatorname{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}). \tag{9.9}$$

$\operatorname{diag}(\mathbf{H}) = (0.2, 4)$ である軸に沿った谷では、Equation 9.9 は毎ステップ両方の座標を同一の因子 $1 - \eta$ で縮小させ、有効条件数を $\kappa = 20$ から 1 へと引き下げる。$\eta = 0.5$ のとき、両方の座標が初期値の 1% 未満に下がるまでのステップ数は、わずか 7 ステップ（$0.5^7 \approx 0.0078$）となり、従来の 47 ステップから大幅に短縮される。

```
def precond_gd_2d(x1, x2, s1, s2, f_grad, eta):
    g1, g2 = f_grad(x1, x2)
    return x1 - eta * g1 / h1, x2 - eta * g2 / h2, 0, 0  # diag(H) = (h1, h2)

def steps_to_1pct(xs):  # First step with both coordinates within 1% of start
    return next(t for t, (x1, x2) in enumerate(xs)
                if abs(x1) <= 0.01 * 5 and abs(x2) <= 0.01 * 2)

plain = train_2d(gd_2d, valley_grad, steps=100, eta=2 / (h1 + h2))
precond = train_2d(precond_gd_2d, valley_grad, steps=10, eta=0.5)
print(f'steps to 1%: plain at its best rate {steps_to_1pct(plain)}, '
      f'preconditioned {steps_to_1pct(precond)}')
d2l.show_trace_2d(f_valley, precond)
```

```
step 100, x1: -0.000225, x2: -0.000090
step 10, x1: -0.004883, x2: -0.001953
steps to 1%: plain at its best rate 47, preconditioned 7
```



深層ネットワークでは $\mathrm{diag}(\mathbf{H})$ が直接与えられるわけではなく、主曲率の方向が座標軸と一致しているわけでもない（谷を 45° 回転させると $\mathbf{H}$ の対角成分が等しくなり、対角前処理は効果を失う。演習問題 7）。後続の節ではこの両方の限界に対処する。すなわち、Adam（9.5 節）は二乗勾配から座標ごとのスケールをオンラインで推定し、モメンタム（9.4 節）と Muon（9.6 節）は座標軸に沿わない方向への進行を加速する。ただしその前にまず、厳密な勾配をノイズを含んだミニバッチ推定値に置き換えたときに何が起こるかを調べよう（9.2 節）。

9.1.6 まとめ

ヘッセ行列の固有値 $\lambda_{\min} \leq \cdots \leq \lambda_{\max}$ をもつ正定値二次関数において、勾配降下法が安定であるのは $\eta < 2/\lambda_{\max}$ のとき、かつそのときに限られる。また、相対精度 ϵ に最適な定数ステップサイズ $\eta^\star = 2/(\lambda_{\min} + \lambda_{\max})$ で到達するには $t \approx (\kappa/2)\log(1/\epsilon)$ ステップを要する（ここで $\kappa = \lambda_{\max}/\lambda_{\min}$ は条件数である）。高次元の非凸ランドスケープでは、高い損失域にある臨界点は主として狭義鞍点であり、勾配降下法は負の曲率をもつ固有ベクトルに沿ってそこから脱出する一方、活性化プラトーは現代的な活性化関数、正規化層、残差接続によって概ね防がれている。

任意の L 滑らかな目的関数に対しては、降下補題により $0 < \eta < 2/L$ である限り単調減少が保証される。逆に、学習率 η を固定すると $2/\eta$ より鋭い極小点はすべて不安定化し、フルバッチでのニューラルネットワークの訓練では、$\lambda_{\max}$ がこの**安定性の縁** $2/\eta$ 付近にとどまるまで漸進的鋭化が進む。曲率に基づく前処理行列によって勾配を再スケーリングすると有効条件数が減少し、対角前処理は $O(d)$ のメモリで座標軸に沿った悪条件性を解消するため、9.5 節で扱う座標ごとの適応的手法の動機となる。

9.1.7 演習問題

1. [code] **鞍点からの脱出時間**。鞍面 $f(x, y) = x^2 - \mu y^2$（$\mu > 0$、下向きの曲率 2μ）を考え、学習率 η の勾配降下法を $(1, \delta)$ から開始する。
   1. $|y| \geq 1$ に達するまでのステップ数が約 $\log(1/\delta)/(2\eta\mu)$（$\eta\mu$ が小さいとき）となることを示せ。この推定値を $\mu \in \{1, 0.01\}$、$\delta = 10^{-2}$、$\eta = 0.1$ について数値的に検証し、曲率 2μ をもつ正曲率の座標を δ 倍に縮小するのに要するステップ数と比較せよ。
   2. $\mu = 1$、$\delta = 0$ とし、各勾配評価に標準偏差 $\sigma \in \{10^{-3}, 10^{-1}\}$ をもつ平均 0 のガウスノイズを加えて脱出時間を測定せよ。得られた結果を、9.1.2 節における測度 0 の安定多様体の結果と整合的に説明せよ。
2. [code] **学習率と鋭さ**。2 層のスカラー線形ネットワークは、ab を入力 1 に対する出力として予測し、二乗損失 $\ell(a, b) = \tfrac{1}{2}(ab - 1)^2$ をとるため、双曲線 $ab = 1$ 上のすべての点が大域的最小点となる。
   1. 最小点 (a, b)（$ab = 1$ 上）におけるヘッセ行列の最大固有値が $\lambda_{\max} = a^2 + b^2$ であることを示せ。最も平坦な最小点はどれか。
   2. 200 ステップの勾配降下法を $(a, b) = (4, 0.1)$ から $\eta \in \{0.05, 0.1, 0.15, 0.2\}$ で実行し、最終的な鋭さ $\lambda_{\max}$ を安定性のしきい値 $2/\eta$ と比較せよ。
3. **置換対称な極小点と非凸性**。●$\theta^\star$ を、$d \geq 2$ 個の隠れユニットをもつ単一隠れ層 MLP の極小点とし、どの 2 つの隠れユニットも同一の入力重みを共有していないものとする。
   1. 隠れユニットを置換すると同一の損失をもつ $d!$ 個の異なるパラメータベクトルが得られることを示し、もし損失が凸であればそれらの平均もまた単一隠れユニットのネットワークと等価な大域的最小点になることを証明せよ。
   2. tanh 活性化関数を用いる場合について、各隠れユニットの符号反転対称性を特定し、その結果として生じる等価な極小点の総数を求めよ。
4. **対称ランダム行列の固有値**。●$\mathbf{M}$ を $k \times k$ の対称ランダム行列とし、その独立な成分 $M_{ij} = M_{ji}$（$i \leq j$）が対称分布 $p_{ij}(x) = p_{ij}(-x)$ から抽出されるものとする (Wigner 1958)。
   1. 一様に選ばれた固有値 λ（$\mathbf{M}$ の固有値）が $P(\lambda > 0) = P(\lambda < 0)$ を満たすことを証明し、区間 $[-2\sigma, 2\sigma]$ 上のウィグナーの半円則を用いて、$\mathbf{M}/\sqrt{k}$ の十分大きな 1 つの実現値における負の固有値の極限割合を求めよ。
   2. $k = 2$ かつ $M_{ii} \sim \mathcal{N}(0, 1)$、$M_{12} \sim \mathcal{N}(0, 1/2)$ のとき、両方の固有値が正となる厳密な確率を計算し、独立なコイン投げによる推定値 2^{-2} と比較せよ。
5. **多数ステップにわたる降下保証**。f を L 滑らかで下に $f^\star$ で有界な関数とし、勾配降下法を $\eta = 1/L$ で $\mathbf{x}_0$ から実行する。
   1. Equation 9.6 を T ステップにわたって足し合わせることで、$\min_{0 \leq t < T} \|\nabla f(\mathbf{x}_t)\|^2 \leq 2L\,(f(\mathbf{x}_0) - f^\star)/T$ を示せ。この上界に凸性は必要か。
   2. 谷の目的関数において初期点 $(-5, -2)$、$T = 100$ としたとき、この上界と反復点が実際に到達する二乗勾配ノルムとを比較せよ。
6. [code] **到達可能な極小点と安定性の縁より下での動作**。
   1. $f(x) = x\cos(0.15\pi x)$ の区間 $(-20, 30)$ における 4 つの極小点を求め、それぞれにおける f'' と安定性のしきい値 $2/f''$ を計算し、各極小点から 0.01 離れた初期値からしきい値の $0.9\times$ および $1.1\times$ の学習率で勾配降下法を試せ。

2. パラメータ数 25 のネットワーク（9.1.4 節）において、`gd_sharpness(0.5, P0)` を実行する前に最終的な鋭さを予測せよ。次に $\eta = 0.1$ で 50,000 ステップ（$2/\eta = 20$）試行し、$2/\eta$ に達する前にデータへ適合した実行がなぜ単調減少のまま（`rising = 0%`）となるのかを説明せよ。

7. [code] **回転下での対角前処理とニュートン法の収束次数。●**
   1. 谷を角度 θ だけ回転させ、$\mathbf{H}_\theta = \mathbf{R}_\theta \,\mathrm{diag}(0.2, 4)\mathbf{R}_\theta^\top$ とする。対角前処理を施したヘッセ行列 $\mathbf{D}^{-1/2}\mathbf{H}_\theta\mathbf{D}^{-1/2}$（ただし $\mathbf{D} = \mathrm{diag}(\mathbf{H}_\theta)$）の条件数を θ の関数として計算し、$\theta = 45^\circ$ において対角前処理が単なるスカラー学習率の変更に帰着し $\kappa = 20$ のまま変わらないことを数値的に検証せよ。
   2. 1 次元のニュートン法 $x_{t+1} = x_t - f'(x_t)/f''(x_t)$ を $f(x) = \cosh(0.5x)$ に対して $x_0 = 10$ から実行し、局所的な収束次数 p（$|x_{t+1}| \approx C|x_t|^p$、0 の近傍）を推定せよ。さらに、$p = 3$ となることを $x - 2\tanh(0.5x)$ のテイラー展開から導出せよ。

9.2 確率的勾配降下法とミニバッチ勾配降下法

9.1 節では、すべてのパラメータ更新に目的関数の厳密な勾配を用いたため、ステップごとに訓練セット全体にわたって損失を評価する必要があった。深層学習において、フルバッチ勾配降下法は計算コストの面から現実的ではない。ImageNet の 1.28 百万枚の画像で ResNet-50 を 90 エポックにわたり 256 枚のミニバッチで訓練すると 450,000 回の更新を要するが (Goyal et al. 2017)、もしフルバッチで 450,000 ステップ実行すればデータセット全体を 450,000 周走査することになり、5,000 倍もの計算量が必要になる。

その代わりに、**確率的勾配降下法**（stochastic gradient descent、SGD）はランダムにサンプリングした 1 つの例、あるいは b 個の例からなる小さな**ミニバッチ**から勾配を推定する（2.1.2.3 節）。この推定量は不偏であり、その分散は $1/b$ に比例して減少するが、サンプリングによるノイズは最適化の終盤の挙動を根本的に変えてしまう。すなわち、一定の学習率 η のもとでは、反復点はある 1 点の最小値へ収束せず、定常的な**ノイズ球**（noise ball）の内部で揺らぎ続ける。

本節では、勾配ノイズ（gradient noise）と不均一な曲率を組み合わせて考察する。9.1.1 節の 2 次元の谷を用いて、一定ステップサイズの SGD における各曲率方向の定常分散を導出し、そこから得られる過剰損失（excess loss）の予測を実際の回帰データセットで検証する。さらに、非復元サンプリング（sampling without replacement）とハードウェアの並列性がミニバッチの選択をどのように左右するかを調べ、学習率減衰（9.7 節）およびモメンタム（9.4 節）を導入する動機を明らかにする。

9.2.1 確率的勾配降下法

9.2.1.1 サンプリングされた勾配と谷

経験リスク（empirical risk）$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x})$ が n 個の訓練例に対して定義されるとき、厳密な勾配は例ごとの勾配の平均となる。

$$\nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{9.10}$$

1 例ずつ用いる SGD は、各ステップでインデックス $i \in \{1, \dots, n\}$ を一様ランダムに引き、次のように更新する。

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}). \tag{9.11}$$

$\mathbb{E}_i[\nabla f_i(\mathbf{x})] = \nabla f(\mathbf{x})$ であるため、確率的勾配（stochastic gradient）は $\nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}) + \xi$ と分解できる。ここで、勾配ノイズ ξ の平均は 0 である。

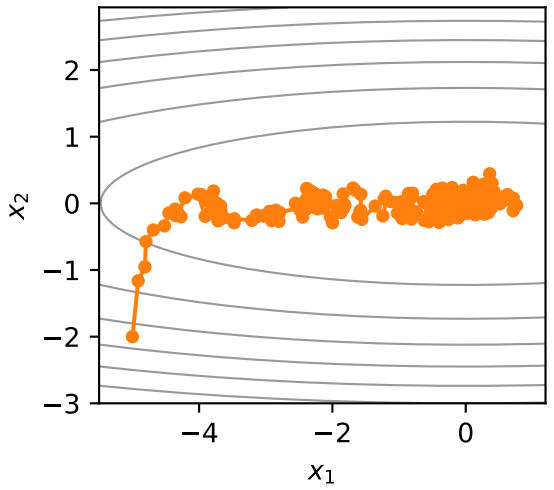
ξ が曲率とどのように相互作用するかを確かめるため、まず勾配の両座標に平均 0・分散 1 の独立なガウスノイズを加える sgd_2d を定義する。

```
def sgd_2d(x1, x2, s1, s2, f_grad, eta, rng):
    g1, g2 = f_grad(x1, x2)
    n1, n2 = rng.normal(size=2)  # Unit-variance gradient noise
    return x1 - eta * (g1 + n1), x2 - eta * (g2 + n2), 0, 0
```

続いて、300 ステップの sgd_2d を谷の関数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$（9.1.1 節）上で、学習率 $\eta = 0.1$、初期点 $(-5, -2)$ から実行する。

```
rng = np.random.default_rng()
trace = d2l.train_2d(sgd_2d, d2l.valley_grad, steps=300, eta=0.1, rng=rng)
d2l.show_trace_2d(d2l.f_valley, trace)
```

```
step 300, x1: -0.371201, x2: -0.077858
```



最初の ~ 100 ステップでは、決定論的な勾配が支配的となり、反復点は原点へと近づく。原点付近に達すると、ノイズの振幅が一定のままであるのに対し $\|\nabla f(\mathbf{x})\|$ は小さくなるため、反復点は平坦な軸 x_1 に沿って引き伸ばされた細長い雲の中に閉じ込められる。

9.2.1.2 ノイズ球

この雲の大きさは解析的に計算できる。曲率 $\lambda > 0$ をもつ固有ベクトルの方向に沿って、平均 0 の独立な勾配ノイズ ξ_t（分散 σ^2）が加わるとき、SGD の 1 ステップは次のように時間発展する。

$$x_{t+1} = x_t - \eta\,(\lambda x_t + \xi_t) = (1 - \eta\lambda)\,x_t - \eta\,\xi_t. \tag{9.12}$$

両辺を二乗して期待値をとると（$\mathbb{E}[x_t \xi_t] = 0$ を用いる）、二次モーメント（second moment）の漸化式が得られる。

$$\mathbb{E}[x_{t+1}^2] = (1-\eta\lambda)^2\,\mathbb{E}[x_t^2] + \eta^2\sigma^2. \tag{9.13}$$

任意の安定なステップサイズ $0 < \eta < 2/\lambda$ に対して、平均 $\mathbb{E}[x_t]$ は 0 へ減衰し、二次モーメントは決定論的な縮小とノイズの注入が釣り合う定常分散へと収束する。すなわち $\mathbb{E}[x_\infty^2] = (1-\eta\lambda)^2\mathbb{E}[x_\infty^2] + \eta^2\sigma^2$ より、次式が成り立つ。

$$\mathbb{E}[x_\infty^2] = \frac{\eta\,\sigma^2}{\lambda\,(2-\eta\lambda)} \approx \frac{\eta\,\sigma^2}{2\lambda} \quad \text{for } \eta\lambda \ll 1. \tag{9.14}$$

Equation 9.14 は**ノイズ球**の性質を特徴づけている（有限時間における完全な軌跡の導出は 35.4.1 節を参照）。

1. **安定性の上限の維持：** $\eta \to 2/\lambda$ に近づくにつれて、分母は $(2-\eta\lambda) \to 0$ となり、分散は発散する。
2. **曲率に反比例するスケーリング：** 定常分散は曲率 λ に反比例する。谷の関数において $\eta = 0.1$、$\sigma^2 = 1$ とすると、定常分散は $0.1/(0.2 \times 1.98) \approx 0.2525$（平坦な座標 x_1 方向、std ≈ 0.50）および $0.1/(4 \times 1.6) \approx 0.0156$（急峻な座標 x_2 方向、std ≈ 0.125）となる。平坦な方向は二重の不利を被る。過渡期には収束が遅く、定常状態では大きく揺らぐからである。
3. **ステップサイズに対する線形スケーリング：** $\eta\lambda \ll 1$ のとき、学習率 η を半分にすると、あらゆる方向で定常分散も半分になる。これは 10,000 本の独立な軌跡をシミュレーションすることで確認できる。

```
def noise_ball(eta, lam, num_runs=10000, num_steps=2000):
    """Variance of SGD iterates on sum(lam * x**2 / 2) with unit noise."""
    rng = np.random.default_rng()
    x = np.zeros((num_runs, len(lam)))
    for _ in range(num_steps):
        x = x - eta * (lam * x + rng.normal(size=x.shape))
    return x.var(axis=0)

lam = np.array(d21.VALLEY_CURVATURES)  # Curvatures of the valley
for eta in (0.1, 0.05):
    print(f'eta={eta}: measured {noise_ball(eta, lam).round(4)}, '
          f'predicted {(eta / (lam * (2 - eta * lam))).round(4)}')
```

```
eta=0.1: measured [0.2544 0.0157], predicted [0.2525 0.0156]
eta=0.05: measured [0.1275 0.007 ], predicted [0.1256 0.0069]
```

9.2.1.3 学習率の減衰

一定ステップサイズの SGD は根本的なトレードオフに直面する。学習率 η を小さくすればノイズ球は $\propto \eta$ で縮小するが、1 ステップあたりの縮小率 $\eta\lambda$ が下がるため初期の収束が遅くなる。ノイズフロアを取り除くには、時間とともに学習率 η_t を減衰させる必要がある。Robbins and Monro (1951) は、勾配の分散が有界である滑らかな強凸目的関数（strongly convex objective）に対し、ステップサイズが次の条件を満たせば SGD が厳密な最小点へ収束することを示した。

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \tag{9.15}$$

第 1 の条件は累積ステップ総量が無限大であることを保証し、任意の初期化から最適点へ到達できるようにする。一方、第 2 の条件は注入される累積ノイズ分散 $\propto \sum_t \eta_t^2$ が有限にとどまることを保証する。$\eta_t = c/(t+t_0)$（ただし $c > 1/(2\lambda_{\min})$）のような多項式スケジュールは両方の条件を満たし、期待二乗誤差を $\mathbb{E}[\|\mathbf{x}_t - \mathbf{x}^\star\|^2] = O(1/t)$ のレートで減少させる（35.4.2 節、演習問題 2）。ステップ数 T の固定された訓練予算のもとでは、実用的な深層学習は漸近的な $1/t$ スケジュールに代えて、コサイン、線形、あるいはウォームアップ・安定・減衰（warmup–stable–decay）スケジュールを用いる（9.7 節）。

9.2.2 ミニバッチ

9.2.2.1 勾配の分散とバッチサイズ

1 つの例を用いる代わりに、ミニバッチ SGD はランダムな部分集合 $\mathcal{B}_t$（b 個の例からなる）にわたって勾配を平均する。

$$\mathbf{g}_t = \frac{1}{b}\sum_{i \in \mathcal{B}_t} \nabla f_i(\mathbf{x}_t). \tag{9.16}$$

例ごとの勾配の共分散行列を $\Sigma(\mathbf{x}) = \mathbb{E}_i\big[(\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x}))(\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x}))^\top\big]$ とおく。そのトレース $\operatorname{tr}\Sigma = \mathbb{E}_i\|\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})\|^2$ は、例ごとの勾配の分散の総和を表す。独立に復元サンプリングされた b 個の例に対して平均をとると、共分散は b で割られる。

$$\mathbb{E}\big\|\mathbf{g}_t - \nabla f(\mathbf{x}_t)\big\|^2 = \frac{\operatorname{tr}\Sigma}{b}. \tag{9.17}$$

$\sigma^2 \leftarrow \sigma^2/b$ を Equation 9.14 に代入すると、曲率 λ の方向に沿ったミニバッチ SGD の定常分散が得られる。

$$\mathbb{E}\big[x_\infty^2\big] = \frac{\eta\,\sigma^2}{b\,\lambda\,(2 - \eta\lambda)} \approx \frac{\eta}{b}\cdot\frac{\sigma^2}{2\lambda} \quad \text{for } \eta\lambda \ll 1. \tag{9.18}$$

ステップサイズが小さい領域 $\eta\lambda \ll 1$ では、学習率とバッチサイズはそれらの比 η/b のみを通じて定常的なノイズ球を支配する。すなわち、バッチサイズ b を 2 倍にすることは、ノイズフロアに対して学習率 η を半分にするのと同じ効果をもつ。

9.2.2.2 データにおける過剰損失

ノイズ球内部での揺らぎは、訓練損失をその最小値 $f(\mathbf{x}^\star)$ からどれだけ押し上げるだろうか。k 番目の固有ベクトル（ヘッセ行列 $\mathbf{H}$ の固有値 λ_k、例ごとの勾配の分散 σ_k^2 をもつ方向）に沿った分散 $v_k \approx \frac{\eta\sigma_k^2}{2\lambda_k b}$ の定常的な変位は、二次損失の期待値に $\frac{1}{2}\lambda_k v_k$ だけ寄与する。

$$\frac{1}{2}\lambda_k\left(\frac{\eta\,\sigma_k^2}{2\lambda_k b}\right) = \frac{\eta\,\sigma_k^2}{4b}. \tag{9.19}$$

ここで曲率 λ_k が約分されて消えることに注目しよう。平坦な方向ほど広い半径にわたって揺らぐが（$\propto 1/\lambda_k$）、平坦な方向への単位変位が損失を押し上げる量は小さい（$\propto \lambda_k$）ためである。

直交する d 個すべての固有ベクトルにわたって和をとると（$\sum_{k=1}^{d} \sigma_k^2 = \operatorname{tr}\Sigma$）、定常状態の**過剰損失**が得られる。

$$\mathbb{E}\big[f(\mathbf{x}_\infty)\big] - f(\mathbf{x}^\star) \approx \frac{\eta \operatorname{tr}\Sigma(\mathbf{x}^\star)}{4b}. \tag{9.20}$$

Equation 9.20 を NASA 翼型自己騒音（airfoil self-noise）回帰データセット（$n = 1{,}500$ 件の風洞測定値からなり、標準化された 5 個の特徴量と標準化された 1 個の音圧ターゲットをもつ）で検証する。

```
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')


def get_airfoil(batch_size=10, n=1500):
    """Load the standardized airfoil self-noise data as minibatches."""
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    dataset = torch.utils.data.TensorDataset(
        *(t.to(d2l.try_gpu()) for t in (data[:n, :-1], data[:n, -1])))
    sampler = torch.utils.data.BatchSampler(  # Whole minibatches of indices
        torch.utils.data.RandomSampler(dataset), batch_size, drop_last=False)
    data_iter = torch.utils.data.DataLoader(dataset, sampler=sampler,
                                            batch_size=None)
    return data_iter, data.shape[1]-1
```

二乗損失を用いた線形回帰に対し（バイアスは定数特徴量に対する 6 番目の重みとして表す）、厳密なヘッセ行列 $\mathbf{H} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$、閉形式の最小点 $\mathbf{w}^\star$、最大曲率 $L = \lambda_{\max}(\mathbf{H}) \approx 2.10$、および例ごとの勾配の分散 $\operatorname{tr}\Sigma(\mathbf{w}^\star) \approx 3.24$ を計算する。

```
data_iter, num_features = get_airfoil(batch_size=1500)
X, y = map(d2l.numpy, next(iter(data_iter)))  # All 1,500 examples
X = np.hstack([X, np.ones((len(X), 1), np.float32)])  # Bias as a 6th weight
H = X.T @ X / len(X)  # Hessian of the loss
w_star = np.linalg.lstsq(X, y)[0]  # The minimizer, in closed form
G = X * (X @ w_star - y)[:, None]  # Per-example gradients at w_star
tr_Sigma = (G ** 2).sum(axis=1).mean()
print(f'L = {np.linalg.eigvalsh(H).max():.2f}, tr Sigma = {tr_Sigma:.2f}')
```

```
L = 2.10, tr Sigma = 3.24
```

この予測に対して経験的な過剰損失を測定するため、`sgd_runs` は復元サンプリングによるベクトル化されたミニバッチ SGD の軌跡を `num_runs=400` 本シミュレーションし、`excess_loss` は $\frac{1}{2}(\mathbf{w}-\mathbf{w}^\star)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}^\star)$ を評価する。

```
def sgd_runs(X, y, w, eta, b, num_steps, rng, num_runs=400):
    """Run minibatch SGD on the squared loss num_runs times from w, drawing
    examples with replacement; return every iterate, (steps, runs, d)."""
    W, trace = np.tile(w, (num_runs, 1)), []
```

```
        for idx in rng.integers(0, len(X), (num_steps, num_runs, b)):
            r = np.einsum('rbd,rd->rb', X[idx], W) - y[idx]  # Residuals
            W = W - eta / b * np.einsum('rbd,rb->rd', X[idx], r)
            trace.append(W)
        return np.array(trace)

    def excess_loss(W, w_star, H):
        """f(w) - f(w_star) of the quadratic loss, for every w in the last axis."""
        D = W - w_star
        return 0.5 * np.einsum('...d,de,...e->...', D, H, D)
```

初期点を $\mathbf{w}^\star$、バッチサイズを $b = 10$ として、3 エポック目（150 ステップ分）にわたる `excess_loss` の平均を $\eta \in \{0.05, 0.025\}$ のそれぞれで求め、$\eta \operatorname{tr}\Sigma/(4b)$ と比較する。

```
rng = np.random.default_rng()
for eta in (0.05, 0.025):  # b = 10, three epochs of 150 steps from w_star
    trace = sgd_runs(X, y, w_star, eta, 10, 450, rng)
    measured = excess_loss(trace[-150:], w_star, H).mean()  # The last epoch
    print(f'eta={eta}: excess loss {measured:.4f}, '
          f'predicted {eta * tr_Sigma / (4 * 10):.4f}')
```

```
eta=0.05: excess loss 0.0042, predicted 0.0041
eta=0.025: excess loss 0.0021, predicted 0.0020
```

$b = 10$ において、実測された過剰損失は、約 0.004（$\eta = 0.05$ のとき）および 0.002（$\eta = 0.025$ のとき）となり、予測値 $\eta \operatorname{tr}\Sigma/(4b)$ とよく一致している。ここで、モデルがすべての訓練例に厳密に適合できるだけの表現力をもち、$\nabla f_i(\mathbf{x}^\star) = \mathbf{0}$ がすべての i について成り立つ**補間レジーム**（interpolating regime）では何が起きるかに注目しよう。この極限では $\operatorname{tr}\Sigma(\mathbf{x}^\star) = 0$ となり、反復点が $\mathbf{x}^\star$ に近づくにつれてノイズ球は消失し、一定ステップサイズの SGD は厳密な最小点へと幾何級数的に収束する。多数のエポックにわたって訓練される過剰パラメータ化された分類器はこの補間レジームに近づく一方、巨大なコーパスを 1 パスだけ走査して訓練される大規模言語モデルは、$\operatorname{tr}\Sigma(\mathbf{x}^\star) > 0$ となるノイズ制限レジーム（noise-limited regime）にとどまる。

9.2.2.3 非復元サンプリング

実際には、データローダはミニバッチを独立に復元サンプリングするのではなく、各エポックの開始時にデータセットをランダムに並べ替え、互いに素なサイズ b のミニバッチへと分割する。b 個の異なる例をサイズ n の有限母集団から非復元サンプリングすると、例ごとの勾配誤差の間に負の相関が生じる（$\sum_{i=1}^{n}(\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})) = \mathbf{0}$ であるため）。その結果、ミニバッチ勾配の分散 Equation 9.17 には有限母集団修正係数（finite-population correction factor）が掛かる（演習問題 3）。

$$\mathbb{E}\left\|\mathbf{g}_t - \nabla f(\mathbf{x}_t)\right\|^2 = \frac{\operatorname{tr}\Sigma}{b} \cdot \frac{n-b}{n-1}. \tag{9.21}$$

$b \ll n$ のとき（10 個を 1,500 個から選ぶ場合）、この係数はほぼ 1 に等しく（0.994）、$b = n$ のときはフルバッチ勾配降下法で要請されるとおり分散が 0 になる。さらに 1 エポック全体を通してみると、ランダムな再シャッフル（random reshuffling）によってエポック内の n/b ステップにわたるミニバッチ勾配誤差が 1 次のオーダーで打ち消し合うため、学習率が小さいときには復元サンプリングよりも定常誤差が小さくなる (Mishchenko et al. 2020)。

9.2.2.4 バッチサイズとハードウェアのスループット

$\eta\lambda \ll 1$ であり、かつ η/b を一定に保つとき、サイズ b のミニバッチに対して学習率 η で 1 ステップ更新すると、期待誤差は $(1-\eta\lambda)$ 倍に縮小し、1 例ずつ b ステップを学習率 η/b で進めた場合と同じノイズ球に到達する。これは $(1-\eta\lambda/b)^b \approx 1-\eta\lambda$ が成り立つためである。例ごとの勾配評価の回数だけで数えるなら、この線形レジームの範囲内で b を大きくしても、学習は速くも遅くもならない。

ミニバッチを不可欠なものにしているのは、**ハードウェアによる償却**（hardware amortization）である。各パラメータ更新には b に依存しない一定の Python ディスパッチおよびカーネル起動のオーバーヘッドが生じる上、ベクトルプロセッサ（CPU や GPU）はモデルの重みをメモリから 1 度読み出すだけで b 個の例を並列に評価できる（第 13 章）。翼型データセットにおいて、1 エポックあたりの実経過時間と処理した例数に対する損失の両方をベンチマークするため、`train_sgd` は $\mathbf{w}=\mathbf{0}$ から線形回帰に対してミニバッチ SGD を実行し、更新ごとのデータセット全体の損失と 1 エポックあたりの秒数を返す。

```
def train_sgd(lr, batch_size, num_epochs=2):
    """Minibatch SGD on linear regression from w = 0; return the loss after
    every update and the seconds per epoch."""
    data_iter, num_features = get_airfoil(batch_size)
    X, y = (torch.cat(t) for t in zip(*data_iter))  # The whole dataset
    w = torch.zeros(num_features + 1, device=X.device, requires_grad=True)
    loss = lambda X, y: ((X @ w[:-1] + w[-1] - y) ** 2).mean() / 2
    loss(X, y).backward()  # Warm-up; the first update clears the gradient
    ws = []

    def epoch():
        for X_b, y_b in data_iter:
            w.grad = None
            loss(X_b, y_b).backward()
            with torch.no_grad():
                w.sub_(lr * w.grad)
            ws.append(w.detach().clone())

    seconds = d2l.Benchmark(epoch, warmup=0, repeats=num_epochs).time
    W = torch.stack(ws)  # The weights after every update, (updates, 6)
    losses = ((X @ W[:, :-1].T + W[:, -1] - y[:, None]) ** 2).mean(0) / 2
    return losses, seconds
```

それでは、`train_sgd` を $b \in \{1, 10, 100\}$ に対して一定の比 $\eta/b = 0.005$ で実行し、あわせてフルバッチ勾配降下法（$b = 1{,}500$、$\eta = 0.9 < 2/L \approx 0.95$）も実行しよう。

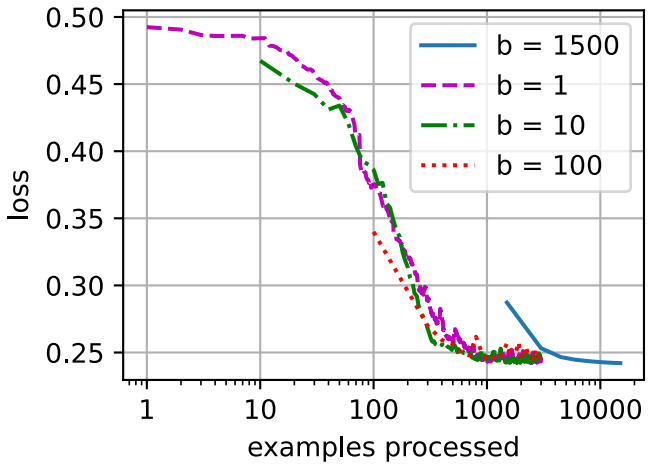
```
runs = {}
for b, lr, num_epochs in ((1500, 0.9, 10), (1, 0.005, 2), (10, 0.05, 2),
                          (100, 0.5, 2)):  # lr / b = 0.005 for SGD
    runs[b], seconds = train_sgd(lr, b, num_epochs)
    updates = 1500 // b
    print(f'b={b:4d}: {updates:4d} updates/epoch, {seconds:.3f} s/epoch '
          f'({1000 * seconds / updates:.1f} ms/update), '
          f'last-epoch loss {float(runs[b][-updates:].mean()):.3f}')
```

```
b=1500:    1 updates/epoch, 0.003 s/epoch (2.7 ms/update), last-epoch loss 0.242
```

```
b=   1: 1500 updates/epoch, 1.825 s/epoch (1.2 ms/update), last-epoch loss 0.245
b=  10:  150 updates/epoch, 0.191 s/epoch (1.3 ms/update), last-epoch loss 0.245
b= 100:   15 updates/epoch, 0.023 s/epoch (1.5 ms/update), last-epoch loss 0.249
```

処理した総例数（$b \times$ updates）に対して訓練損失をプロットすると、4 つのバッチサイズすべてのサンプル効率を重ねて比較できる。

```
d2l.plot([b * np.arange(1, len(losses) + 1) for b, losses in runs.items()],
         [d2l.numpy(losses) for losses in runs.values()],
         'examples processed', 'loss', legend=[f'b = {b}' for b in runs],
         xscale='log')
```



$b = 100$ 個の例に対する 1 回の更新は、$b = 1$ での更新とほぼ同じ実経過時間で完了するため、$b = 100$ の 1 エポックは約 50–100× 高速になりながら、処理した例数あたりではまったく同じ損失曲線をたどる。同時に、線形スケーリング $\eta \propto b$ がどこで破綻するかにも注目しよう。$\eta/b = 0.005$ を保ったまま $b = 100$ にすると学習率は $\eta = 0.5$ まで上がり、$\eta L = 0.5 \times 2.10 = 1.05$ となって 2 と比べて十分に小さいとは言えなくなる（その結果、Equation 9.14 の分母の逆数 $(2 - \eta\lambda)^{-1}$ が増大する）。さらに $b \approx 190$ を超えるバッチサイズでは、安定性の上限 $\eta < 2/L \approx 0.95$ に違反してしまう。この臨界バッチサイズ（critical batch size）の遷移については、9.8 節で体系的に解析する。

9.2.3 まとめ

ミニバッチ SGD は b 個の例から勾配を推定し、サンプリングによる分散を $\mathrm{tr}(\Sigma)/b$（非復元サンプリングではこれに $(n-b)/(n-1)$ を掛けた値）まで低減しつつ、ステップごとのディスパッチおよびメモリ帯域幅のコストをミニバッチ全体で償却する。一定の学習率 η のもとでは、反復点は定常的な**ノイズ球**へと収束し、曲率 λ の方向に沿った座標分散は $\frac{\eta\sigma^2}{b\lambda(2-\eta\lambda)} \approx \frac{\eta\sigma^2}{2\lambda b}$ となって最も平坦な方向で最も広くなる。全モードにわたって二次損失への寄与を足し合わせると λ が相殺され、定常状態の期待**過剰損失** $\mathbb{E}[f(\mathbf{x}_\infty)] - f(\mathbf{x}^\star) \approx \frac{\eta\,\mathrm{tr}\,\Sigma(\mathbf{x}^\star)}{4b}$ が残る。この過剰損失が 0 になるのは、最適点においてモデルがすべての訓練例を補間する場合に限られる。$\eta L \ll 1$ が成り立つ限り、1 例あたりの過渡的な進捗と定常ノイズフロアはいずれも η と b にその比 η/b のみを通じて依存するが、b を $\eta \propto b$ に従って増やし続けると、やがて決定論的な曲率の上限 $\eta < 2/L$ に近づきこの性質は成り立たなくなる。

9.2.4 演習問題

1. **ランダムなデータセットとしてのノイズ。** 分散 1 のガウス勾配ノイズを伴う `sgd_2d` を $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 上で実行することは、$f(\mathbf{x}, \mathbf{w}) = 0.1(x_1 - w_1)^2 + 2(x_2 - w_2)^2$（ただし $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{C})$）上で 1 例ずつの SGD を実行することと等価であることを示し、共分散 $\mathbf{C}$ を求めよ。

2. [code] **$1/t$ スケジュールの定数。**●曲率 λ の 1 次元二次関数に対し、分散 σ^2 の勾配ノイズと学習率 $\eta_t = c/(t + t_0)$ のもとで以下を行え。

   1. $e_t = \mathbb{E}[x_t^2]$ に対する厳密な漸化式を書き下し、漸近的な仮定形 $e_t \approx A/t$ を代入することで、$A = c^2\sigma^2/(2c\lambda - 1)$ が（$c > 1/(2\lambda)$ のとき常に）成り立つことを示せ。どの c の値が A を最小化するか。
   2. e_t の決定論的な漸化式を $\lambda = 0.2, \sigma = 1$、初期値 $e_0 = 25$、$t_0 = 20$、$c \in \{1, 2.5, 5, 10\}$ として $t = 10^6$ まで反復計算し、両対数軸上で $t\,e_t$ を t に対してプロットせよ。さらに、単一の c では $\lambda_1 = 0.2$ と $\lambda_2 = 4$ の両方に対して同時に最適になり得ない理由を説明せよ。

3. [code] **有限母集団係数と再シャッフル。**

   1. $\mathbf{e}_i = \nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})$ とおく。$\sum_{i=1}^{n} \mathbf{e}_i = \mathbf{0}$ を用いて $\mathbb{E}_{i \neq j}[\mathbf{e}_i^\top \mathbf{e}_j] = -\operatorname{tr}\Sigma/(n-1)$ を示し、Equation 9.21 を導出せよ。さらに翼型データセットの例ごとの勾配 `G` を用い、$b \in \{10, 100, 500, 1000\}$ を復元サンプリングおよび非復元サンプリングした場合について、この公式を実験的に検証せよ。
   2. $n = 100$ 個のベクトル $\mathbf{w}_i$ を演習問題 1 の分布から生成し、復元サンプリングとエポックごとの再シャッフルによる 20 エポックの SGD の軌跡を $\eta \in \{0.1, 0.01\}$ で比較せよ。

4. [code] **訓練中の勾配の分散。**●

   1. 翼型の線形モデルにおいて、$\operatorname{tr}\Sigma$ の値を $\mathbf{w} = \mathbf{0}$ と $\mathbf{w} = \mathbf{w}^\star$ で比較せよ。
   2. 2 層の ReLU MLP（隠れユニット数 64）を翼型データで 10 エポック（$b = 10, \eta = 0.05$）訓練し、$\operatorname{tr}\Sigma$（`W1`、`b1`、`W2`、`b2` ごとの内訳）を初期化時点と訓練後とで比較せよ。

5. [code] **バッチサイズと学習率のスケーリング。**●翼型の線形回帰モデルにおいて $\eta = 0.05$（$b = 10$ のとき）を基準に固定し、$b \in \{1, 10, 100, 1500\}$ を線形スケーリング（$\eta \propto b$）と平方根スケーリング（$\eta \propto \sqrt{b}$）の両方で試せ。$L = 2.10$ を用いてどの設定が発散するかを説明し、さらに `H` の全 6 個の固有ベクトルにわたる厳密な定常過剰損失を Equation 9.14 から計算して、$b = 100, \eta = 0.5$ のフロアが $b = 10, \eta = 0.05$ よりもわずかに高くなる理由を説明せよ。

9.3 2 つのテストモデル

2 次元の二次関数の谷のような解析的モデルは、曲率と勾配ノイズが単純な更新則をどのように支配するかを説明してくれるが、現代のオプティマイザを評価するには実際のニューラルネットワーク上での経験的な測定が欠かせない。なぜ AdamW は Transformer では大差で SGD を上回る一方、畳み込みネットワークでははるかに小さな優位性しか示さないのだろうか。こうした問いに答えるには、統制された実験プロトコルのもとで実際のモデルを訓練する必要がある。

確率的な訓練の実行結果はランダム初期化やミニバッチの抽出ごとに変動するため、2 つのオプティマイザ設定のあいだの差が意味を持つのは、それが単一の設定における実行間のばらつきを上回る場合に限られる。本節では、本章の残りの部分を通して使用する 2 つのベンチマークアーキテクチャ、すなわちバイトレベルの Transformer 言語モデル `TinyLM`（パラメータ数 470,784）

と畳み込み分類器 FashionCNN（パラメータ数 421,642）を定義し、あわせて共通の訓練ループ、学習率スイープ用ハーネス、ベースラインとなる SGD の測定結果、および有意性のしきい値を導入する。

```
import os
import pandas as pd
import random
import textwrap
```

9.3.1 言語モデル TinyLM

TinyLM は、次の UTF-8 バイト（$V = 256$）を連続する 64 バイトのコンテキストウィンドウから予測する自己回帰型のバイトレベル言語モデルである（8.2 節）。標準的な 2,000 ステップの実行では、64 個のウィンドウからなるミニバッチを用いて $2{,}000 \times 64 \times 64 \approx 8.2 \times 10^6$ 個のターゲットバイトを処理する。訓練例の繰り返し（最適化の速度と暗記を混同させてしまう）を避けるため、訓練には WikiText-103 の第 1 シャード（Wikipedia 記事 $\sim 2.7 \times 10^8$ UTF-8 バイト分）を用いる (Merity et al. 2016)。各訓練ミニバッチはランダムなバイトオフセットから 64 個のウィンドウをサンプリングするため、2,000 ステップの実行で目にするのはコーパスのわずか $\sim 3\%$ にとどまり、すべての訓練ステップで新しいテキストを評価することになる。検証パープレキシティは、1.1×10^6 バイトの検証分割から切り出した連続する非重複ウィンドウ上で測定する。

```
class WikiText103(d2l.DataModule):
    """WikiText-103 as UTF-8 bytes, cut into windows of num_steps bytes."""
    def __init__(self, batch_size, num_steps, num_train=None, num_val=None):
        super().__init__()
        self.save_hyperparameters()
        self.vocab = d2l.BPETokenizer(vocab_size=256, specials=())  # Bytes
        self.train = self._read('train-00000-of-00002',
                                '3cbc792dfc0eeedf8d7333acdc15a9f536032d9a')
        self.val = self._read('validation-00000-of-00001',
                              '49d15019b0476afc752bd5f010b5a2aab2de89ca')
        self.train, self.val = self.train[:num_train], self.val[:num_val]

    def _read(self, split, sha1):
        url = ('https://huggingface.co/datasets/Salesforce/wikitext/resolve/'
               f'main/wikitext-103-raw-v1/{split}.parquet')
        fname = d2l.download(url, os.path.join(self.root, 'wikitext-103'),
                             sha1)
        lines = pd.read_parquet(fname)['text']  # Each line ends with '\n'
        return np.frombuffer(b''.join(s.encode() for s in lines), np.uint8)

    def windows(self, corpus, starts):
        """Inputs and next-byte targets of the windows at `starts`."""
        idx = np.asarray(starts)[:, None] + np.arange(self.num_steps + 1)
        array = torch.from_numpy(corpus[idx].astype(np.int64))
        return array[:, :-1], array[:, 1:]

    def train_dataloader(self):  # A pass of minibatches at random offsets
        n = len(self.train) - self.num_steps
        for _ in range(n // (self.batch_size * self.num_steps)):
```

```
            yield self.windows(self.train,
                               torch.randint(n, (self.batch_size,)).numpy())

    def val_dataloader(self):  # Consecutive, non-overlapping windows
        starts = np.arange(0, len(self.val) - self.num_steps, self.num_steps)
        for i in range(0, len(starts), self.batch_size):
            yield self.windows(self.val, starts[i:i + self.batch_size])
```

`WikiText103(batch_size=64, num_steps=64)` をインスタンス化すると、訓練シャードと検証シャードがダウンロードされ、コーパス先頭の 300 バイトがデコードされる。

```
data = WikiText103(batch_size=64, num_steps=64)
print(f'{len(data.train):,} training bytes, '
      f'{len(data.val):,} validation bytes')
text = data.vocab.decode(data.train[:300])
print('\n'.join(textwrap.fill(line, 72) for line in text.splitlines()))
```

```
269,630,498 training bytes, 1,144,248 validation bytes
 = Valkyria Chronicles III =
 Senjō no Valkyria 3 : Unrecorded Chronicles ( Japanese : 戦場のヴァルキュリア 3 ,
lit . Valkyria of the Battlefield 3 ) , commonly referred to as Valkyria
Chronicles III outside Japan , is a tactical role @-@ playing video game
developed by Sega and Media.Vis
```

9.3.1.1 アーキテクチャとパラメータグループ

`TinyLM` は 2 ブロックのデコーダのみの Transformer（第 11 章）であり、埋め込み幅は $d_{\text{model}} = 128$、各ブロックは 2 個の因果アテンション（causal attention）ヘッドと、隠れ層幅 $4d_{\text{model}} = 512$ の 2 層 GELU MLP を持つ。各サブ層の前には LayerNorm（7.3.4.1 節）が置かれて残差接続で囲まれ、最後に最終 LayerNorm と 256 個のバイトロジットへ射影する線形出力ヘッドが続く。埋め込みの各行は $\mathcal{N}(0, 1)$ から初期化され、すべての線形層の重みとバイアスは $[-1/\sqrt{\text{fan-in}}, 1/\sqrt{\text{fan-in}}]$ から一様に抽出される。

```
class TinyLM(nn.Module):
    """A small decoder-only transformer language model."""
    def __init__(self, vocab_size, d_model=128, num_heads=2, num_blks=2,
                 max_len=64):
        super().__init__()
        self.num_heads = num_heads
        self.token_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)
        self.blks = nn.ModuleList([nn.ModuleDict(dict(
            norm1=nn.LayerNorm(d_model),
            qkv=nn.Linear(d_model, 3 * d_model),
            proj=nn.Linear(d_model, d_model),
            norm2=nn.LayerNorm(d_model),
            mlp=nn.Sequential(nn.Linear(d_model, 4 * d_model), nn.GELU(),
                              nn.Linear(4 * d_model, d_model))))
            for _ in range(num_blks)])
        self.norm = nn.LayerNorm(d_model)
```

```
        self.head = nn.Linear(d_model, vocab_size)

    def attention(self, blk, X):
        B, T, D = X.shape
        q, k, v = blk['qkv'](X).chunk(3, dim=-1)
        q, k, v = (u.reshape(B, T, self.num_heads, -1).transpose(1, 2)
                   for u in (q, k, v))
        Y = F.scaled_dot_product_attention(q, k, v, is_causal=True)
        return blk['proj'](Y.transpose(1, 2).reshape(B, T, D))

    def forward(self, X):
        H = self.token_emb(X) + self.pos_emb(torch.arange(X.shape[1],
                                                          device=X.device))
        for blk in self.blks:
            H = H + self.attention(blk, blk['norm1'](H))
            H = H + blk['mlp'](blk['norm2'](H))
        return self.head(self.norm(H))
```

オプティマイザの観点からは、TinyLM の 470,784 個のパラメータは構造的に異なる 3 つのグループに分かれ、後の節ではそれぞれ異なる扱いを受ける（たとえば 9.5.3 節の分離型重み減衰（decoupled weight decay）や 9.6 節の行列直交化など）。

- **埋め込み**（9%）：ルックアップテーブル（token_emb、pos_emb）であり、その各行はトークン頻度に比例したスパースな勾配を受け取る。
- **重み行列**（90%）：密な 2 次元射影行列（各ブロックの qkv、proj、2 つの MLP 層、および head）である。
- **1 次元ベクトル**（$< 1\%$）：線形層のバイアスおよび LayerNorm のスケール・シフトベクトルである。

```
# nn.Linear and nn.Embedding initialize as described above by default;
# nn.GELU is the exact GELU
model = TinyLM(vocab_size=len(data.vocab))
kinds = {'embeddings': 0, 'matrices': 0, 'vectors': 0}
print(f'{"parameter":<22} {"shape":<14} {"count":>7}')
for name, p in model.named_parameters():
    kind = ('embeddings' if 'emb' in name else
            'matrices' if p.ndim == 2 else 'vectors')
    kinds[kind] += p.numel()
    print(f'{name:<22} {str(tuple(p.shape)):<14} {p.numel():>7}')
for kind, count in kinds.items():
    print(f'{kind:>37} {count:>7}')
print(f'{"total":>37} {sum(kinds.values()):>7}')
```

```
parameter              shape           count
token_emb.weight       (256, 128)      32768
pos_emb.weight         (64, 128)        8192
blks.0.norm1.weight    (128,)             128
blks.0.norm1.bias      (128,)             128
blks.0.qkv.weight      (384, 128)      49152
blks.0.qkv.bias        (384,)             384
blks.0.proj.weight     (128, 128)      16384
blks.0.proj.bias       (128,)             128
```

```
blks.0.norm2.weight    (128,)            128
blks.0.norm2.bias      (128,)            128
blks.0.mlp.0.weight    (512, 128)      65536
blks.0.mlp.0.bias      (512,)            512
blks.0.mlp.2.weight    (128, 512)      65536
blks.0.mlp.2.bias      (128,)            128
blks.1.norm1.weight    (128,)            128
blks.1.norm1.bias      (128,)            128
blks.1.qkv.weight      (384, 128)      49152
blks.1.qkv.bias        (384,)            384
blks.1.proj.weight     (128, 128)      16384
blks.1.proj.bias       (128,)            128
blks.1.norm2.weight    (128,)            128
blks.1.norm2.bias      (128,)            128
blks.1.mlp.0.weight    (512, 128)      65536
blks.1.mlp.0.bias      (512,)            512
blks.1.mlp.2.weight    (128, 512)      65536
blks.1.mlp.2.bias      (128,)            128
norm.weight            (128,)            128
norm.bias              (128,)            128
head.weight            (256, 128)      32768
head.bias              (256,)            256
                    embeddings   40960
                      matrices  425984
                       vectors    3840
                         total  470784
```

9.3.2 画像分類器 FashionCNN

Transformer におけるオプティマイザのランキングは畳み込みネットワークにおけるランキングと大きく異なるため、2 つ目のベンチマークとして 28×28 の Fashion-MNIST 画像（3.2 節）に対する分類器 `FashionCNN` を導入する。2 段の 3×3 畳み込み + ReLU + 2×2 最大プーリングのステージ（32 チャネルと 64 チャネル）によって空間解像度を $64 \times 7 \times 7$ に縮小し、その後に 2 つの線形層（$3{,}136 \to 128 \to 10$）を続ける。すべての層で一様分布 $\pm 1/\sqrt{\text{fan-in}}$ による初期化を用い、パラメータ数は合計 $421{,}642$ となる。

```python
class FashionCNN(nn.Module):
    """A compact CNN for Fashion-MNIST: two conv-pool stages, two linear."""
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, X):
        X = F.max_pool2d(F.relu(self.conv1(X)), 2)
        X = F.max_pool2d(F.relu(self.conv2(X)), 2)
        return self.fc2(F.relu(self.fc1(X.flatten(1))))
```

`d2l.num_params` を用いて、`FashionCNN` が $421{,}642$ 個の訓練可能なパラメータを持つことを確認する。

```
# nn.Conv2d and nn.Linear draw from U(-a, a), a = 1/sqrt(fan-in), by default;
# images arrive as (batch, channel, height, width)
print(f'{d2l.num_params(FashionCNN()):,} parameters')
```

```
421,642 parameters
```

バッチサイズ 256 の場合、2,000 ステップの実行では ∼ 8.5 周にわたって 60,000 枚の Fashion-MNIST 訓練画像を走査することになるため、evaluate_accuracy を用いて訓練損失とあわせてホールドアウトのテスト正解率も報告する。

```
fashion = d2l.FashionMNIST(batch_size=256)

def evaluate_accuracy(model, batches):
    """Fraction of the examples in batches that model classifies correctly."""
    device = next(model.parameters()).device
    training, correct, total = model.training, 0, 0
    model.eval()  # Dropout off, batch norm on running statistics
    with torch.inference_mode():
        for X, y in batches:
            y_hat = model(X.to(device)).argmax(dim=1)
            correct += (y_hat == y.to(device)).sum()
            total += len(y)
    model.train(training)
    return float(correct) / total
```

9.3.3 訓練ハーネスと評価補助関数

どちらのモデルもミニバッチをロジットへ写像し、一定のステップ予算にわたって平均交差エントロピー損失を最小化する。Table 9.2 に、d2l に保存される訓練ループと評価ユーティリティをまとめる。

Table 9.2: d2l に保存される共通の訓練・評価関数。

関数	目的
train_steps	num_steps ステップ（または stop(step) が成り立つまで）訓練し、ステップごとの交差エントロピー損失を返す
lm_perplexity	バッチのイテレータ上でトークンごとのパープレキシティ $\exp(\text{mean cross-entropy})$ を計算する（8.3 節）
evaluate_accuracy	バッチのイテレータ上で分類正解率を計算する
final_loss	直近 $k = 100$ ステップの損失の平均（発散した場合は inf）
smooth	プロット用の連続する $k = 25$ 個の損失ウィンドウにわたるブロック平均
steps_to_reach	25 ステップ平滑化損失が target 以下に下がる最初のステップ（到達しない場合は None）
lr_sweep	同一の初期重みから学習率のリストにわたって訓練し、損失曲線と評価スコアを返す

まず train_steps を実装する。この関数は、num_steps 個のミニバッチにわたって（パディングラ

ベル `-100` を無視しつつ)、あるいは任意の述語 `stop(step)` が成り立つまで、平均交差エントロピーを最小化する。

```
def train_steps(model, data, optimizer, num_steps, *, stop=None):
    """Minimize the mean cross-entropy of the logits model(X) for the labels
    Y over the minibatches (X, Y) of data, leaving out the labels -100, for
    num_steps steps or until stop(step) is true; return every step's loss."""
    device = d2l.try_gpu()
    model.to(device)
    losses, step = [], 0
    while step < num_steps:
        for X, Y in data.train_dataloader():
            X, Y = X.to(device), Y.to(device)
            logits = model(X)
            loss = F.cross_entropy(logits.reshape(-1, logits.shape[-1]),
                                   Y.reshape(-1), ignore_index=-100)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            losses.append(loss.detach())
            step += 1
            if step >= num_steps or (stop is not None and stop(step)):
                return torch.stack(losses).tolist()
```

次に、`lm_perplexity` はミニバッチのイテレータ上で言語モデルを推論モードで評価し、トークンあたりの $\exp(\text{mean cross-entropy})$ を返す。

```
def lm_perplexity(model, batches):
    """Perplexity of a language model on the minibatches (X, Y) of batches."""
    device = next(model.parameters()).device
    modes = {m: m.training for m in model.modules()}  # Restored at the end
    model.eval()  # Dropout off while we measure
    total, count = 0, 0
    with torch.inference_mode():
        for X, Y in batches:
            logits = model(X.to(device))
            total += F.cross_entropy(logits.flatten(0, 1),
                                     Y.to(device).flatten(), reduction='sum')
            count += Y.numel()
    for m, training in modes.items():
        m.training = training
    return math.exp(float(total) / count)
```

ノイズを含むステップ損失の履歴を要約・プロットするため、`final_loss` は直近 $k = 100$ ステップの損失を平均し、`smooth` は 25 ステップごとのブロック平均を計算し、`steps_to_reach` は平滑化された損失が `target` に到達する最初のステップを求める。

```
def final_loss(losses, k=100):
    """Mean of the last k losses; infinity if the run diverged."""
    v = sum(losses[-k:]) / k
    return v if math.isfinite(v) else float('inf')
```

```
def smooth(losses, k=25):
    """Means over consecutive blocks of k losses, for plotting."""
    return [sum(losses[i:i + k]) / k
            for i in range(0, len(losses) - k + 1, k)]

def steps_to_reach(losses, target, k=25):
    """First step at which the smoothed losses reach target; None if they
    never do."""
    for i, v in enumerate(smooth(losses, k)):
        if v <= target:
            return k * (i + 1)
    return None
```

最後に、`lr_sweep` は同じ乱数 seed から初期化した新しいモデルを `lrs` 内の各学習率で訓練し、指定があれば `evaluate(model)` を `every` ステップごとに実行する。

```
def lr_sweep(make_model, make_optimizer, data, lrs, seed,
             num_steps=2000, *, evaluate=None, metric='score', every=None,
             verbose=True):
    """Train make_model() with make_optimizer(model, lr) for each lr, from
    one initialization; return the loss curves and the evaluations."""
    every = every or num_steps
    curves, scores = {}, {}
    for lr in lrs:
        torch.manual_seed(seed)  # The same initialization for every lr
        model = make_model()
        optimizer = make_optimizer(model, lr)
        curves[lr], scores[lr] = [], []
        for _ in range(num_steps // every):
            curves[lr] += train_steps(model, data, optimizer, every)
            if evaluate is not None:
                scores[lr].append(evaluate(model))
        line = f'lr {lr:g}: final loss {final_loss(curves[lr]):.3f}'
        if scores[lr]:
            line += f', {metric} {scores[lr][-1]:.3f}'
        if verbose:
            print(line)
    return curves, scores
```

9.3.4 ミニバッチ SGD のベースライン実行と実験プロトコル

ここで、定数学習率のミニバッチ SGD（2,000 ステップ、モメンタムも重み減衰もなし）によるベースラインを確立しよう。`TinyLM` では、未訓練のモデルは一様分布の交差エントロピー $\ln 256 \approx 5.55$ nats/byte よりわずかに高い値から出発する（初期ロジットの分散のため ~ 5.71 nats/byte となる。演習問題 1 を参照）。WikiText-103 における参照用のランドマークとして、ユニグラムのバイト頻度テーブルは ~ 3.2 nats/byte、バイグラムのバイトペアテーブルは ~ 2.5 nats/byte を達成する（演習問題 6）。Figure 9.4 はこれらのランドマークを 1 つの尺度上に並べたものであり、本章におけるすべての損失はこの尺度に照らして読み取ることができる。

まず、モデルと素の SGD を生成するファクトリクロージャ `make_lm`、`sgd`、`new_seed` を定義する。

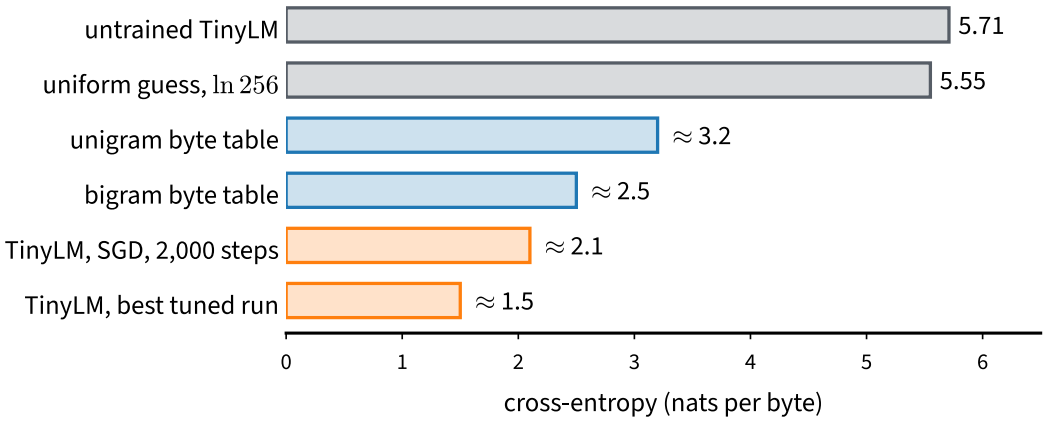


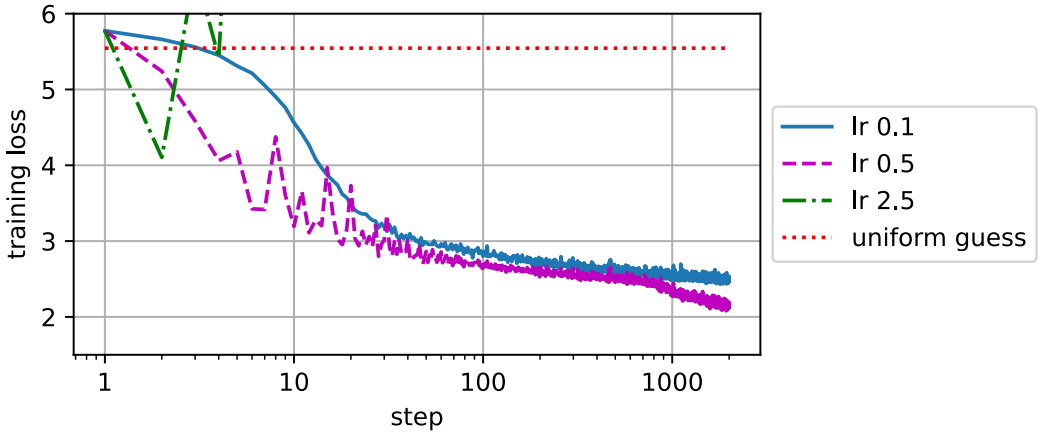
Figure 9.4: WikiText-103 におけるバイトレベル言語モデリングの損失ランドマーク。灰色：未訓練の TinyLM と一様な当て推量。青：ユニグラムおよびバイグラムのバイトテーブル。オレンジ：以下で測定する 2,000 ステップの SGD 実行後の TinyLM、および本章後半で最もよく調整された実行後の結果。

```
make_lm = lambda: TinyLM(len(data.vocab))
sgd = lambda model, lr: torch.optim.SGD(model.parameters(), lr)
new_seed = lambda: random.getrandbits(31)
```

lr_sweep を TinyLM 上で $\eta \in \{0.1, 0.5, 2.5\}$ にわたって実行し、ステップごとの損失を一様な当て推量のベースライン $\ln 256$ とともにプロットすると、典型的な 3 つの学習率領域が現れる。

```
lm_sgd, _ = lr_sweep(make_lm, sgd, data, lrs=[0.1, 0.5, 2.5], seed=new_seed())
d2l.plot(list(range(1, 2001)),
         list(lm_sgd.values()) + [[math.log(256)] * 2000],
         'step', 'training loss', xscale='log', ylim=[1.5, 6],
         legend=[f'lr {lr:g}' for lr in lm_sgd] + ['uniform guess'],
         legend_loc='outside', figsize=(5, 2.5))
```

```
lr 0.1: final loss 2.492
lr 0.5: final loss 2.151
lr 2.5: final loss inf
```



- **高すぎる場合**（$\eta = 2.5$）：最初のステップの後に局所的な安定性しきい値 $2/L$ を超え、10 ステップ以内に `inf` へと発散する。
- **低すぎる場合**（$\eta = 0.1$）：滑らかだがゆっくりと降下し、到達するのは $\sim$ 2.5 nats/byte（バイグラムの水準、ステップ 2,000 時点）にとどまる。
- **適切に調整された場合**（$\eta = 0.5$）：ネットワークが初期の高曲率領域を通過する際に序盤で一時的なスパイクを経験するが、その後回復して約 2.1 nats/byte に到達する。

`FashionCNN` では、$\eta = 0.3$ の素の SGD が訓練損失 $\sim$ 0.22、ホールドアウトのテスト正解率 $\sim 90\%$ に到達する。

```
test_acc = lambda model: evaluate_accuracy(model, fashion.val_dataloader())
cnn_sgd, _ = lr_sweep(FashionCNN, sgd, fashion, lrs=[0.3], seed=new_seed(),
                      evaluate=test_acc, metric='test accuracy')
```

```
lr 0.3: final loss 0.221, test accuracy 0.901
```

9.3.4.1 比較プロトコルと有意性のしきい値

Deep Learning Tuning Playbook (Godbole et al. 2023) に従い、2 つのオプティマイザを公平に比較するには、学習率 η を手法ごとに独立にスイープすべき**局外ハイパーパラメータ**（nuisance hyperparameter）として扱い（共通の初期重みから対数等間隔グリッド上で `lr_sweep` を実行する）、任意の共通学習率で 2 つのオプティマイザを比較することは避けなければならない (Schmidt et al. 2021)。各オプティマイザの最良の学習率で達成された最小損失とあわせて、その**使用可能な学習率範囲**（usable learning-rate range：最適値から 0.15 nats 以内で終了するグリッド点の範囲）も追跡する。

適切に調整された学習率のもとで 5 個の独立な乱数シードにわたって実行すると、末尾 100 ステップの訓練損失（`final_loss`）および検証損失は `TinyLM` では $\sim$ 0.01–0.015 nats の範囲で変動し、一方 `FashionCNN` では訓練損失で $\sim$ 0.02、テスト正解率で $\sim 1\%$ 変動する。保守的な $3\times$ の範囲マージンを採用し、単一実行での差を統計的に有意とみなすのは、`TinyLM` で 0.05 **nats**、または `FashionCNN` で 0.06 **の訓練損失（3% のテスト正解率）**を超える場合に限ることとする。それより小さな差は、複数のシードにわたって確認されない限り引き分けとして扱う。

9.3.5 まとめ

本章の経験的な比較では、相補的な 2 つのニューラルネットワークベンチマークを組み合わせる。1 つ目は `TinyLM`（470,784 パラメータ：90% が 2 次元射影行列、9% が埋め込み、< 1% が 1 次元ベクトル）であり、ランダムな 64 バイトウィンドウを WikiText-103 から切り出すシングルパスのストリーミング設定で訓練するため、2,000 ステップの実行において毎ステップ新しいテキストを評価できる。2 つ目は `FashionCNN`（421,642 パラメータ）であり、~ 8.5 エポックにわたり 256 枚の画像からなる Fashion-MNIST のミニバッチ上で訓練し、訓練損失とホールドアウトのテスト正解率の両方で評価する。また、閉形式の一様・ユニグラム・バイグラムのバイトエントロピーによって、訓練を始める前の段階で `TinyLM` の損失スケールに基準を与える。アルゴリズム本来の改善をハイパーパラメータやシードに対する感度から切り分けるため、すべての比較において同一の初期重みからオプティマイザごとに学習率 η を独立にスイープし、複数シードに基づく有意性のしきい値である 0.05 nats（`TinyLM`）および損失 0.06 または正解率 3%（`FashionCNN`）に照らして、最適損失と使用可能な学習率範囲の両方を報告する。

9.3.6 演習問題

1. **訓練前の初期損失。**
   1. ロジット $z_1, \dots, z_V$ がターゲットとは独立に $\mathcal{N}(0, s^2)$ から独立同分布に抽出されるとき、$\sum_{i=1}^{V} e^{z_i} \approx V\mathbb{E}[e^{z_i}] = Ve^{s^2/2}$ を用いて、初期交差エントロピーの期待値がおよそ $\ln V + s^2/2$ になることを示せ。
   2. 平均 0、分散 1 の LayerNorm 出力と、$\mathcal{U}(-1/\sqrt{128}, 1/\sqrt{128})$ から抽出されたヘッドの重み・バイアスが与えられたとき、$s^2 = 129/384 \approx 1/3$ かつ $\ln 256 + 1/6 \approx 5.71$ nats となり、これがステップ 1 における `TinyLM` の損失と一致することを示せ。ヘッドの重みとバイアスを 0 で初期化した場合、初期損失はいくらになるか。

2. [code] **`FashionCNN` の損失曲線の診断。** `FashionCNN` を $\eta \in \{0.003, 0.03, 0.3, 1.0\}$ にわたって素の SGD で訓練し、ステップ損失を対数スケールのステップ軸上で一様分布のベースライン $\ln 10 \approx 2.3026$ とともにプロットせよ。$\ln 10$ 付近で停滞する実行について、1 ミニバッチにわたる死んだ ReLU の活性化値の割合を調べ、なぜ勾配が消失するのかを説明せよ。

3. [code] **`TinyLM` における発散の境界。** `TinyLM` 上で素の SGD を $\eta \in \{0.7, 1.0, 1.4, 2.0\}$ および 5 個のシードにわたってスイープせよ（`lr_sweep(..., verbose=False)`）。生存率、発散する実行が破綻するステップ、および発散しなかった実行における最小損失を記録せよ。

4. [code] **複数シードのばらつきと標準誤差。**●20 個のシードで `TinyLM` を AdamW（9.5 節）により $\eta = 3 \times 10^{-3}$（`weight_decay=0`）で実行し、`final_loss` の標本標準偏差 σ を正規分布の範囲推定量 range$/3.7$ と比較したうえで、必要なシード数 k（0.005 nats の差を標準誤差の 2 倍 $\sigma\sqrt{2/k}$ の水準で検出できる数）を求めよ。

5. [code] **訓練損失と検証損失の比較および小規模コーパスでの過学習。**
   1. なぜ `final_loss`（直近 100 訓練ステップの平均）がわずかに（0.02–0.03 nats ほど）ステップ 2000 終了後の `TinyLM` の検証損失より高くなるのかを説明し、最終チェックポイントをホールドアウトの訓練ウィンドウ上で評価して確認せよ。
   2. コーパスを 100,000 バイトに制限し（`WikiText103(batch_size=64, num_steps=64, num_train=100000)` により ~ 80 エポックを 2,000 ステップで回る設定）、訓練損失と検証損失を 250 ステップごと（`every=250`）にプロットせよ。

6. [code] **カウントベースの n グラムランドマーク。** 経験的なバイトユニグラムのエントロピー（~ 3.2 nats）、ならびに 1 加算平滑化を用いたバイグラム（~ 2.5 nats）およびトライグラムの検証分割上での交差エントロピーを、先頭 10^7 訓練バイトの `WikiText103` から計

算し、2,000 ステップの SGD（約 2.1 nats）および AdamW（1.59 nats、9.5 節）による `TinyLM` の損失と比較せよ。

9.4 モメンタム

9.1 節では、通常の勾配降下法が悪条件（ill-conditioned）の谷で苦戦する理由を見た。最も急峻な曲率 $\lambda_{\max}$ によって安定性の上限 $\eta < 2/\lambda_{\max}$ が課されるため、最も平坦な方向 $\lambda_{\min}$ はステップごとに高々 $O(1/\kappa)$ しか縮小しない（ここで $\kappa = \lambda_{\max}/\lambda_{\min}$ は条件数である）。通常の勾配降下法が遅いのは、各ステップがその時点の勾配 $\mathbf{g}_t$ のみに依存しているからである。しかし、直近の勾配の履歴には明確な幾何学的情報が含まれている。谷底に沿っては連続する勾配が一貫して同じ方向を指すのに対し、急峻な壁を横切る方向では反復点が左右にオーバーシュートするため勾配の符号が交互に反転する。

Polyak (1964) は、過去の勾配の指数重み付き和である**速度**（velocity）に沿ってステップを進める手法を提案した。これにより、平坦な方向に沿う持続的な成分は蓄積され、急峻な方向を横切る振動成分は打ち消し合う。摩擦を受けながら溝を転がり落ちる重い球になぞらえられる**ヘビーボールモメンタム**は、パラメータごとに単一の速度バッファを保持するコストと引き換えに、二次ボウル上での最悪ケースの反復回数を $O(\kappa)$ から $O(\sqrt{\kappa})$ へと削減する。

本節では、速度の漸化式、その実効ステップサイズ（effective step size）と記憶の時間スケール、および臨界減衰（critical damping）による $\sqrt{\kappa}$ 加速則を解析する。続いて、フルバッチのソフトマックス回帰と `FashionCNN` のミニバッチ訓練でモメンタムを検証し、ネステロフの先読みモメンタムを導入したうえで、β と η を調整するための実践的な指針を確立する。

```
import random
```

9.4.1 モメンタム法

9.4.1.1 速度と実効ステップサイズ

勾配 $\mathbf{g}_t$（ステップ t における厳密勾配またはミニバッチ勾配）が与えられたとき、ヘビーボールモメンタムは速度ベクトル $\mathbf{v}_t$（初期値 $\mathbf{v}_0 = \mathbf{0}$）を保持し、次のように更新する。

$$\begin{aligned} \mathbf{v}_t &= \beta\, \mathbf{v}_{t-1} + \mathbf{g}_t, \\ \mathbf{x}_{t+1} &= \mathbf{x}_t - \eta\, \mathbf{v}_t, \end{aligned} \tag{9.22}$$

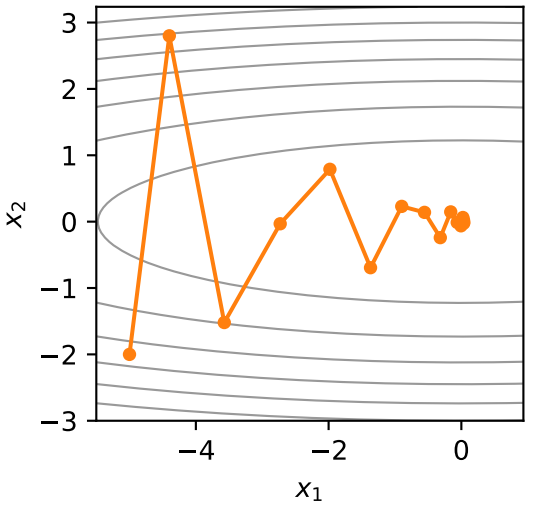
ここで、$\beta \in [0, 1)$ はモメンタム係数である（$\beta = 0$ とすると勾配降下法に戻る）。この漸化式を展開すると、速度は過去のすべての勾配の指数重み付き和として表される。

$$\mathbf{v}_t = \sum_{k=0}^{t-1} \beta^k \mathbf{g}_{t-k} = \mathbf{g}_t + \beta\, \mathbf{g}_{t-1} + \beta^2\, \mathbf{g}_{t-2} + \cdots + \beta^{t-1}\, \mathbf{g}_1.$$

この重み付き和が、谷 $f(\mathbf{x}) = 0.1 x_1^2 + 2 x_2^2$（$\lambda_1 = 0.2, \lambda_2 = 4$）の 2 つの座標に対して $\eta = 0.6$ でどのように働くか考えてみよう。この学習率は通常の勾配降下法の安定限界 $2/\lambda_2 = 0.5$ を超えており、通常の勾配降下法では x_2 方向に沿ってステップごとに $1 - 4(0.6) = -1.4$ 倍となって発散する。ここにモメンタム $\beta = 0.5$ を加えると軌跡が安定化し、両座標とも原点の 0.01 以内に 20 ステップで到達する。

```
h1, h2 = d2l.VALLEY_CURVATURES
f_valley, valley_grad = d2l.f_valley, d2l.valley_grad

def momentum_2d(x1, x2, v1, v2, f_grad, eta, beta):
    g1, g2 = f_grad(x1, x2)
    v1, v2 = beta * v1 + g1, beta * v2 + g2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

d2l.show_trace_2d(f_valley, d2l.train_2d(momentum_2d, valley_grad, eta=0.6,
                                         beta=0.5))
```

```
step 20, x1: 0.007188, x2: 0.002553
```



モメンタムが平坦な座標を加速しつつ同時に急峻な座標を安定化できる理由は、次の 2 つの極限的なケースから説明できる。

- **持続する勾配（平坦な谷底）**：連続する勾配がほぼ一定（$\mathbf{g}_{t-k} \approx \mathbf{g}$）にとどまるとき、速度は $\mathbf{v}_t \approx (1+\beta+\beta^2+\cdots)\mathbf{g} = \frac{1}{1-\beta}\mathbf{g}$ まで蓄積し、次の**実効ステップサイズ**をもたらす。

$$\eta_{\text{eff}} = \frac{\eta}{1-\beta}. \tag{9.23}$$

- **符号が反転する勾配（急峻な壁）**：連続する勾配の符号が交互に反転する（$\mathbf{g}_{t-k} \approx (-1)^k\mathbf{g}$）とき、幾何級数の和は $1-\beta+\beta^2-\cdots = \frac{1}{1+\beta}$ となり、実効ステップサイズは $\frac{\eta}{1+\beta}$ へと減衰する。

Figure 9.5 に、単位サイズの勾配に対する両ケースを示す。

$\eta = 0.6$、$\beta = 0.5$ のとき、モメンタムは谷底に沿った前進を $\eta_{\text{eff}} = 0.6/0.5 = 1.2$ へと増幅する一方、急峻な壁を横切る振動を $0.6/1.5 = 0.4 < 0.5$ へと減衰させる。その結果、曲率 λ に対する安定性の上限は $\eta\lambda < 2$ から $\eta\lambda < 2(1+\beta)$ へと引き上げられる（演習問題 3）。

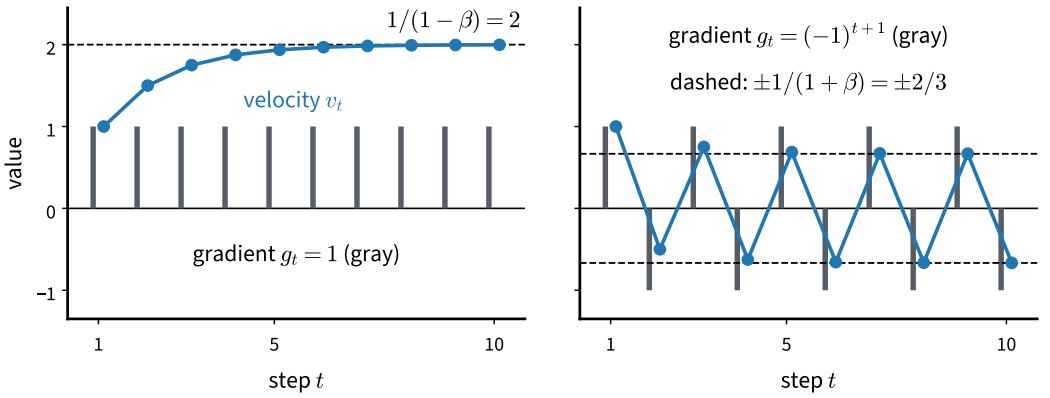


Figure 9.5: 速度 $v_t = \beta v_{t-1} + g_t$（青）の推移（単位勾配 g_t は灰色の棒、$\beta = 0.5$）。左：同符号の勾配が足し合わさり、v_t は $1/(1-\beta) = 2$ に近づく。右：符号が交互に反転する勾配は大半が打ち消し合い、v_t は振幅 $1/(1+\beta) = 2/3$ の振動に落ち着く。

9.4.1.2 β の時間スケールとミニバッチノイズ

重み $1, \beta, \beta^2, \ldots$ の和は $1/(1-\beta)$ となるため、速度は $\tau = 1/(1-\beta)$ ステップの実効記憶ウィンドウと $\log(1/2)/\log\beta$ ステップの半減期を持つ。

```
print('beta   window   half-life   weight inside the window')
for beta in [0.5, 0.9, 0.99]:
    window = 1 / (1 - beta)
    half_life = np.log(0.5) / np.log(beta)
    inside = 1 - beta ** window
    print(f'{beta:4}   {window:6.0f}   {half_life:9.1f}   {inside:24.2f}')
```

```
beta   window   half-life   weight inside the window
 0.5        2         1.0                      0.75
 0.9       10         6.6                      0.65
0.99      100        69.0                      0.63
```

β を 0.9 から 0.99 に増やすと、記憶ホライズンと実効ステップサイズ $\eta_{\text{eff}} = \eta/(1-\beta)$ の両方が 10 倍（10 ステップから 100 ステップへ）に拡大する。

確率的勾配 $\mathbf{g}_t = \nabla f(\mathbf{x}_t) + \xi_t$ のもとで、$1/(1-\beta)$ ステップにわたる平均化は 9.2 節の定常ノイズ球（noise ball）を縮小させるだろうか。答えは否である。各ノイズ摂動 ξ_t は $\mathbf{v}_t$ に重み 1 で加わり、その後のステップで β^k として減衰するため、累積変位として $\eta(1+\beta+\beta^2+\cdots)\xi_t = \eta_{\text{eff}}\xi_t$ だけ寄与する。したがって、実効ステップサイズ $\eta_{\text{eff}} = \eta/(1-\beta)$ を揃えた場合、モメンタムありの SGD とモメンタムなしの SGD はほぼ同一の分散を持つ定常ノイズ球に落ち着く（演習問題 5）。ゆえに、ノイズ律速（noise-limited）の訓練で β を大きくする際には、η を $1-\beta$ に比例してスケーリングし、η_{eff} を一定に保つべきである。

9.4.2 加速と臨界減衰

組 (η, β) としてどのような選択が、固有値 $\lambda_{\min} \le \cdots \le \lambda_{\max}$ と条件数 $\kappa = \lambda_{\max}/\lambda_{\min}$ を持つ二次形式において最適となるだろうか。Polyak (1964) は、次のように設定すればよいことを証明した。

$$\eta^\star = \frac{4}{\left(\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}\right)^2}, \qquad \beta^\star = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^2 \tag{9.24}$$

このとき、最悪ケースにおけるステップごとの縮小率（contraction factor）は次の値に達する。

$$\rho^\star = \sqrt{\beta^\star} = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} \approx 1 - \frac{2}{\sqrt{\kappa}},$$

これにより、相対精度 ϵ に到達するために必要なステップ数は $\frac{\kappa}{2}\log(1/\epsilon)$ から $\frac{\sqrt{\kappa}}{2}\log(1/\epsilon)$ へと削減される（35.1.3 節）。

縮小率が $\sqrt{\beta^\star}$ に等しくなる理由を見るため、$\eta v_t = x_t - x_{t+1}$ を曲率 λ の固有ベクトルに沿った 1 次元の速度更新式に代入すると、次の 2 階線形漸化式が得られる。

$$x_{t+1} = (1 + \beta - \eta\lambda)\, x_t - \beta\, x_{t-1}, \tag{9.25}$$

この漸化式の特性多項式は $z^2 - (1+\beta-\eta\lambda)z + \beta = 0$ である。解と係数の関係により、2 つの根の積は $z_1 z_2 = \beta$ となる。判別式 $\Delta = (1+\beta-\eta\lambda)^2 - 4\beta$ が非正であるとき（これはすべての $\beta \geq \beta_{\mathrm{c}}(\lambda) = (1-\sqrt{\eta\lambda})^2$ で生じる）、根は同じ絶対値 $|z_1| = |z_2| = \sqrt{\beta}$ を持つ複素共役対をなし、その大きさは曲率 λ に依存しない（Figure 9.6）。

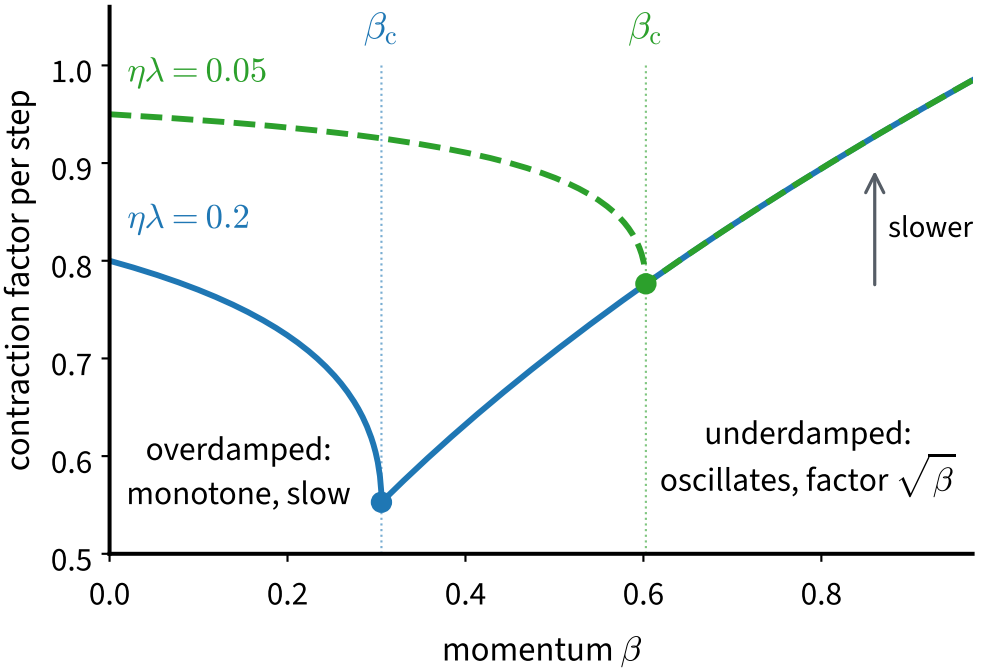


Figure 9.6: 曲率 λ に沿ったヘビーボールモメンタムのステップごとの縮小率（β の関数）。収束は臨界減衰のしきい値 $\beta_{\mathrm{c}} = (1-\sqrt{\eta\lambda})^2$ で最速となり、β_{c} 未満では過減衰、β_{c} を超えると絶対値 $\sqrt{\beta}$ で振動する。

Equation 9.24 の最適パラメータは、最も平坦な固有値と最も急峻な固有値の臨界減衰しきい値を $(1-\sqrt{\eta^\star\lambda_{\min}})^2 = (\sqrt{\eta^\star\lambda_{\max}}-1)^2 = \beta^\star$ として等しくする。これにより両端のモードを臨界減衰させるとともに、すべての中間固有値 $\lambda \in [\lambda_{\min}, \lambda_{\max}]$ が不足減衰（underdamped）の複素根領域に入り、縮小率 $\sqrt{\beta^\star}$ で縮小することを保証する。

先ほどの谷（$\kappa = 20$）では、Equation 9.24 から $\eta^\star \approx 0.668, \beta^\star \approx 0.403$ が得られる。これにより漸近縮小率は 0.905 から $\sqrt{\beta^\star} \approx 0.635$ に低下し、相対誤差 10^{-6} に達するまでのステップ数は 139 から 40 へと減少する。

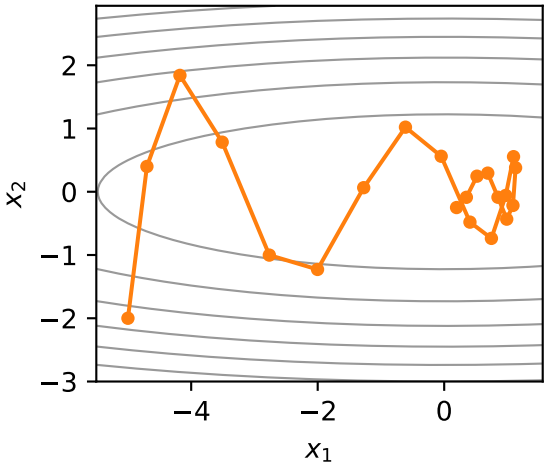
```
def steps_to(tol, eta, beta, steps=400):
    """Steps after which |x1|, |x2| stay below tol times their start."""
    x1, x2, v1, v2, last = -5.0, -2.0, 0.0, 0.0, 0
    for t in range(1, steps + 1):
        x1, x2, v1, v2 = momentum_2d(x1, x2, v1, v2, valley_grad, eta, beta)
        if abs(x1) > 5 * tol or abs(x2) > 2 * tol:
            last = t
    return last + 1

def contraction(eta, beta):
    return max(abs(np.roots([1, -(1 + beta - eta * lam), beta])).max()
               for lam in (h1, h2))

eta_opt = 4 / (np.sqrt(h2) + np.sqrt(h1)) ** 2
beta_opt = ((np.sqrt(h2 / h1) - 1) / (np.sqrt(h2 / h1) + 1)) ** 2
print('  eta   beta   factor   steps to 1%   steps to 1e-6')
for eta, beta in [(2 / (h1 + h2), 0), (0.6, 0.5), (eta_opt, beta_opt),
                  (0.3, 0.8)]:
    print(f'{eta:5.3f}  {beta:5.3f}  {contraction(eta, beta):7.3f}  '
          f'{steps_to(1e-2, eta, beta):12d}  {steps_to(1e-6, eta, beta):14d}')
```

```
  eta   beta   factor   steps to 1%   steps to 1e-6
0.476  0.000    0.905            47             139
0.600  0.500    0.707            16              42
0.668  0.403    0.635            18              40
0.300  0.800    0.894            42             126
```

4 行目（$\eta = 0.3, \beta = 0.8$）に注目しよう。β を臨界しきい値 β_{c} より大幅に大きく選ぶと、両座標とも絶対値 $\sqrt{0.8} \approx 0.894$ の不足減衰となり、軌跡はオーバーシュートして原点周りで減衰振動（リンギング）を起こす。

```
d2l.show_trace_2d(f_valley, d2l.train_2d(momentum_2d, valley_grad, eta=0.3,
                                         beta=0.8))
```

```
step 20, x1: 0.195696, x2: -0.249649
```



9.4.3 データ上でのモメンタム

これまでの解析から、データ上で訓練する際には次の 2 つの異なるレジームが予測される。

1. **曲率律速レジーム（高精度な勾配、大きな κ）：**モメンタムによってより大きな実効ステップサイズ η_{eff} が可能になり、必要な反復回数が大幅に削減される。
2. **ノイズ律速レジーム（小さなミニバッチ）：**定常ノイズフロアは $\eta_{\text{eff}} = \eta/(1-\beta)$ に支配されるため、$\beta = 0.9$ を加えると最適学習率は $1-\beta = 0.1$ 倍に下がる一方、収束速度は同程度にとどまる。

Fashion-MNIST 上で $\beta = 0$ と $\beta = 0.9$ を比較し、両レジームを検証しよう。対象は、(1) フルバッチのソフトマックス回帰（$b = 60{,}000$、厳密な勾配、非負のピクセル強度により悪条件）と、(2) `FashionCNN`（9.3.2 節）の小ミニバッチ訓練（$b = 32$）である。まず、正規化した Fashion-MNIST テンソルをメモリ内ローダ `Batches` としてアクセラレータ上に事前読み込みする。

```
fashion = d2l.FashionMNIST()
X = fashion.train.data[:, None].to(d2l.try_gpu()) / 255  # NCHW
y = fashion.train.targets.to(d2l.try_gpu())

class Batches:
    """The examples X, y on the GPU, in a new random order on every pass."""
    def __init__(self, X, y, batch_size):
        self.X, self.y, self.batch_size = X, y, batch_size

    def train_dataloader(self):
        order = torch.randperm(len(self.y), device=self.y.device)
        return ((self.X[i], self.y[i])
                for i in order.view(-1, self.batch_size))
```

次に、ゼロ初期化した線形モデル `softmax_regression` と、モメンタムを引数にとる SGD コンストラクタ `make_sgd(beta)` を定義する。

```
def softmax_regression():
    """A linear layer from the pixels to the logits, starting at zero."""
```

```
    net = nn.Sequential(nn.Flatten(), nn.Linear(28 * 28, 10))
    nn.init.zeros_(net[1].weight)
    nn.init.zeros_(net[1].bias)
    return net

make_sgd = lambda beta: lambda model, lr: torch.optim.SGD(
    model.parameters(), lr, momentum=beta)
new_seed = lambda: random.getrandbits(31)
```

ここで d2l.lr_sweep を用いて grids にわたる学習率スイープを両モデルと $\beta \in \{0.0, 0.9\}$ に対して実行し、各設定における最良の学習率、目標損失への到達ステップ数、および最終損失を記録する。

```
# Model, batch size, steps, target loss
setups = {'softmax regression': (softmax_regression, 60000, 1000, 0.5),
          'FashionCNN': (d2l.FashionCNN, 32, 3000, 0.3)}
grids = {('softmax regression', 0.0): [0.0625, 0.125, 0.25, 0.5, 1],
         ('softmax regression', 0.9): [0.1, 0.2, 0.4, 0.8],
         ('FashionCNN', 0.0): [0.05, 0.1, 0.2, 0.4],
         ('FashionCNN', 0.9): [0.005, 0.01, 0.02, 0.04]}
best = {}
for (name, beta), lrs in grids.items():
    make_model, b, num_steps, target = setups[name]
    curves, _ = d2l.lr_sweep(make_model, make_sgd(beta), Batches(X, y, b),
                             lrs, new_seed(), num_steps, verbose=False)
    best[name, beta] = min(lrs, key=lambda lr: d2l.final_loss(curves[lr]))
    c = curves[best[name, beta]]
    print(f'{name:<18}  beta={beta}  best lr {best[name, beta]:<5g}  steps to '
          f'{target}: {d2l.steps_to_reach(c, target)}  final loss '
          f'{d2l.final_loss(c):.3f}')
```

```
softmax regression  beta=0.0  best lr 0.125  steps to 0.5: 625  final loss 0.472
softmax regression  beta=0.9  best lr 0.4    steps to 0.5: 100  final loss 0.365
FashionCNN          beta=0.0  best lr 0.2    steps to 0.3: 1550  final loss 0.293
FashionCNN          beta=0.9  best lr 0.02   steps to 0.3: 1425  final loss 0.280
```

いずれの予測も定量的に成り立つことがわかる。

- フルバッチの**ソフトマックス回帰**では、モメンタム $\beta = 0.9$ を加えることで最適学習率が $\eta = 0.125$ から $\eta = 0.4$ に上昇し（$\eta_{\text{eff}} = 4.0$ と実効ステップサイズが $32\times$ に拡大）、損失 0.5 に到達するまでのステップ数は $6\times$ 以上削減され（$625 \to 100$ ステップ）、最終損失も 0.472 から 0.365 へと低下する。
- 小バッチの **FashionCNN**（$b = 32$）では、訓練はノイズ律速となる。最適学習率は $\eta \approx 0.1$〜0.2（$\beta = 0$ 時）から $\eta \approx 0.02$〜0.04（$\beta = 0.9$ 時）へと低下し（比率にして 5〜10 倍の変化であり、実効ステップサイズ η_{eff} は $\beta = 0$ での値から公比 2 のグリッド上で 1 段階以内に保たれる）、いずれの設定でも損失 0.3 に到達し、最終損失も 0.26〜0.30 と 9.3 節の単一シード変動の範囲内に収まる。

9.4.4 ネステロフのモメンタム

不足減衰の実行例（$\eta = 0.3, \beta = 0.8$）においてヘビーボールモメンタムが最小点をオーバーシュートしたのは、速度によってまさに到達しようとしている地点ではなく、現在位置 $\mathbf{x}_t$ で勾配を

評価していたためである。Nesterov (1983) は、**先読み点**（look-ahead point）$\mathbf{x}_t - \eta\beta\mathbf{v}_{t-1}$ で勾配を評価する手法を提案した（Figure 9.7）。

$$\begin{aligned}\mathbf{v}_t &= \beta\,\mathbf{v}_{t-1} + \nabla f(\mathbf{x}_t - \eta\beta\,\mathbf{v}_{t-1}),\\ \mathbf{x}_{t+1} &= \mathbf{x}_t - \eta\,\mathbf{v}_t.\end{aligned} \tag{9.26}$$

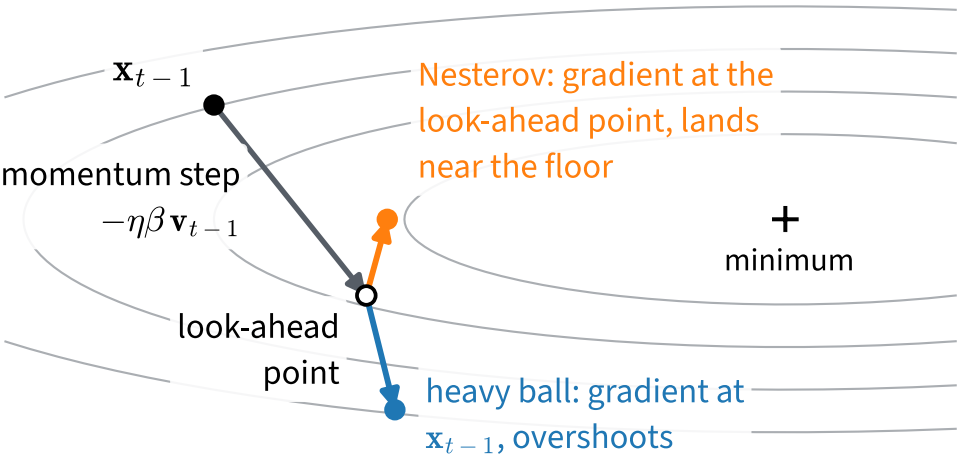
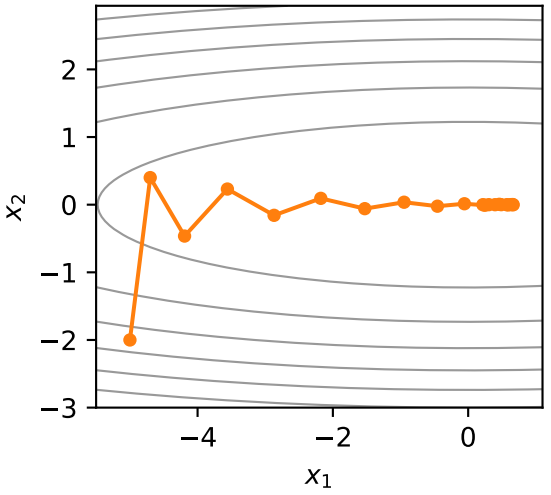


Figure 9.7: 同じ反復点と速度からのヘビーボールモメンタムとネステロフのモメンタムの 1 ステップの比較。ヘビーボールモメンタムは $\mathbf{x}_t$ で勾配を評価してオーバーシュートするが、ネステロフのモメンタムは先読み点 $\mathbf{x}_t - \eta\beta\mathbf{v}_{t-1}$ で勾配を評価し、即座に引き戻す補正を加える。

先読み点で勾配を評価すると、急峻な方向の振動を抑える先回り減衰（anticipatory damping）が導入される。さらに、理論的保証もヘビーボールモメンタムより広い。定数 $\beta = (\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$ を用いれば、ネステロフのモメンタムは二次関数だけでなく、あらゆる滑らかな強凸関数で $\sqrt{\kappa}$ のレートを達成する。また、増加スケジュール $\beta_t = (t-1)/(t+2)$ を用いれば、一般の滑らかな凸目的関数に対して最適な $O(1/t^2)$ の収束率を達成する (Nesterov 2018)（35.1.3 節）。ここでは定数の β を用いる。

```
def nesterov_2d(x1, x2, v1, v2, f_grad, eta, beta):
    g1, g2 = f_grad(x1 - eta * beta * v1,  # Gradient at the look-ahead
                    x2 - eta * beta * v2)  # point, not at (x1, x2)
    v1, v2 = beta * v1 + g1, beta * v2 + g2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

d2l.show_trace_2d(f_valley, d2l.train_2d(nesterov_2d, valley_grad, eta=0.3,
                                         beta=0.8))
```

```
step 20, x1: 0.217791, x2: -0.000070
```



実務上は、ほとんどのアーキテクチャにおいて $\beta = 0.9$ が頑健なデフォルト値となり、学習率は実効ステップサイズ $\eta_{\text{eff}} = \eta/(1-\beta)$ を通じて調整される。しかし、モメンタムは依然としてすべての座標に単一のスカラーのステップサイズを適用する。Transformer のように層によって勾配のスケールが何桁も異なる場合には、座標ごとの適応的なスケーリングが必要となる（9.5 節）。

9.4.5 まとめ

ヘビーボールモメンタムは、指数減衰する速度 $\mathbf{v}_t = \beta\mathbf{v}_{t-1} + \mathbf{g}_t$（記憶ホライズン $1/(1-\beta)$）に沿ってパラメータを更新し、持続的な勾配方向を実効ステップサイズ $\eta_{\text{eff}} = \eta/(1-\beta)$ へと増幅する一方、符号が交互に反転する方向を $\eta/(1+\beta)$ へと減衰させ、安定性の上限を $\eta\lambda < 2(1+\beta)$ に引き上げる。条件数 κ の二次ボウル上では、$(\eta^\star, \beta^\star)$ によって両端の固有値を臨界減衰させることで、反復計算量は $O(\kappa)$ から $O(\sqrt{\kappa})$ へと低下する。ネステロフのモメンタムは先読み点 $\mathbf{x}_t - \eta\beta\mathbf{v}_{t-1}$ で勾配を評価することで先回り減衰をもたらし、不足減衰によるリンギングを抑えるとともに、加速効果を一般の滑らかな凸関数へと拡張する。ミニバッチノイズのもとでは、実効ステップサイズ $\eta_{\text{eff}} = \eta/(1-\beta)$ を揃えれば定常ノイズ球が保たれるため、モメンタムは曲率律速（curvature-limited）レジームで大幅な高速化をもたらす一方、ノイズ律速レジームでは $\eta \propto 1-\beta$ のスケーリングを必要とする。

9.4.6 演習問題

1. [code] **モメンタムと学習率のグリッド**。`momentum_2d` を `f_valley` 上で $\beta \in \{0, 0.25, 0.5, 0.8\}$ および $\eta \in \{0.1, 0.3, 0.6, 1.0\}$ にわたり最大 1,000 ステップ実行せよ。$\|\mathbf{x}_t\| \le 10^{-3}$ に到達するまでのステップ数（または発散）を表にまとめ、各 η における経験的な最適値を $\beta^\star$（Equation 9.24）および $\beta_{\text{c}}(\lambda) = (1-\sqrt{\eta\lambda})^2$ と比較せよ。

2. **速度のノイズ対信号比**。$\mathbf{g}_t = \mathbf{g} + \xi_t$ において、座標ごとに独立同分布で平均ゼロ・分散 σ^2 のノイズが加わるとき、$\mathbf{v}_t$（Equation 9.22）の定常平均が $\mathbf{g}/(1-\beta)$、定常分散が $\sigma^2/(1-\beta^2)$ となり、速度のノイズ対信号比が $\sqrt{(1-\beta)/(1+\beta)}$ 倍に縮小することを示せ。

3. **ネステロフのモメンタムの同値な形式と安定性**。

    1. ライブラリで用いられる形式 $\mathbf{v}_t = \beta\mathbf{v}_{t-1} + \nabla f(\mathbf{y}_t)$、$\mathbf{y}_{t+1} = \mathbf{y}_t - \eta(\nabla f(\mathbf{y}_t) + \beta\mathbf{v}_t)$ が、Equation 9.26 と $\mathbf{y}_t = \mathbf{x}_t - \eta\beta\mathbf{v}_{t-1}$ のもとでまったく同じ速度を生成すること

を数学的帰納法により示せ。

2. Equation 9.25 から、$z^2 - (1 + \beta - \eta\lambda)z + \beta = 0$ の両根が $0 \le \beta < 1$ において単位円の真に内部に存在するのは、$0 < \eta\lambda < 2(1 + \beta)$ のとき、かつそのときに限ることを証明せよ。

4. **指数平均形式のモメンタム。**@eq-momentum を、正規化された EMA 形式 $\mathbf{m}_t = \beta\mathbf{m}_{t-1} + (1 - \beta)\mathbf{g}_t$、$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha\mathbf{m}_t$（Adam で使用、9.5 節）と比較せよ。$\mathbf{m}_t = (1 - \beta)\mathbf{v}_t$ となり、したがって $\alpha = \eta/(1 - \beta) = \eta_{\text{eff}}$ のとき両者の軌跡が一致することを示せ。

5. [code] **バッチサイズにわたるミニバッチノイズ下のモメンタム。●**

   1. $f(x) = \frac{\lambda}{2}x^2$ 上でノイズ付き勾配 $g_t = \lambda x_t + \xi_t$ を用いるとき、ステップサイズ α の通常の SGD と、モメンタム（$\beta = 0.9$）で $\eta = (1 - \beta)\alpha$ とした場合の反復点の定常分散を $\alpha\lambda \in \{0.05, 0.2\}$ について比較せよ。
   2. `d2l.FashionCNN` を $1{,}000$ ステップにわたり、バッチサイズ $b \in \{32, 500, 6000\}$、$\beta \in \{0, 0.9\}$、および揃えた実効ステップサイズ $\eta/(1 - \beta) = 0.2$ で訓練せよ。続いて η を $b = 6000$ においてスイープし、バッチサイズが曲率律速レジームに入るほど大きくなるとモメンタムが訓練を加速することを検証せよ。

9.5 Adam と AdamW

9.3 節では、単純なミニバッチ SGD は `TinyLM` において学習率 $\eta = 0.5$ を用いてようやく約 2.1 nats/byte の損失に到達した。これに対し、100 分の 1 未満の学習率である $\eta = 3 \times 10^{-3}$ を用いた Adam は、以下で測定するように 1.59 nats/byte という損失に同じ $2{,}000$ ステップで到達する。公称学習率が 100 分の 1 ほど小さいオプティマイザがなぜより速く訓練を進められるのか、そしてなぜ Transformer における普遍的なデフォルトとなっているのだろうか。

モメンタム（9.4 節）は勾配を時間方向に平均化するが、それでもすべてのパラメータに単一のスカラーのステップサイズを適用する。しかし深層ネットワーク、とりわけ Transformer では、埋め込み、アテンション射影、MLP 層、正規化パラメータの間で勾配の大きさが何桁も異なる。**Adam**（**Adaptive Moment Estimation**、Kingma and Ba (2015)）は、モメンタムと、**AdaGrad** (Duchi et al. 2011) および **RMSProp** (Tieleman and Hinton 2012) を経て発展した座標ごとの二乗平均平方根（root mean square、RMS）勾配の前処理行列（preconditioner）を組み合わせる。各座標の更新量をその座標自身の過去の勾配スケールで割ることにより、学習率 η は生の勾配に対する乗数ではなく、パラメータ空間における直接的なステップ長となる。

しかし、まさにこの座標ごとの除算が標準的な ℓ_2 正則化（regularization）を破綻させる。損失に $\frac{\lambda}{2}\|\mathbf{x}\|^2$ を加えると、各重みの減衰率がその勾配スケールの逆数で再スケーリングされてしまうからである。Loshchilov and Hutter (2019) は重み減衰（weight decay）を適応的な前処理行列から分離することでこの問題を解決し、**AdamW** を生み出した。本節では、AdaGrad と RMSProp から Adam への発展をたどり、`TinyLM` 上で座標ごとの勾配スケールを測定してモメンタム付き SGD と Adam を比較し、さらに AdamW とその重み減衰のタイムスケール $\tau = 1/(\eta\lambda)$ および平衡ノルムを導出したうえで、どのパラメータグループを重み減衰の対象外とすべきかを詳しく述べる。

```
import copy
import random
```

9.5.1 座標ごとの適応的学習率

9.5.1.1 AdaGrad と RMSProp

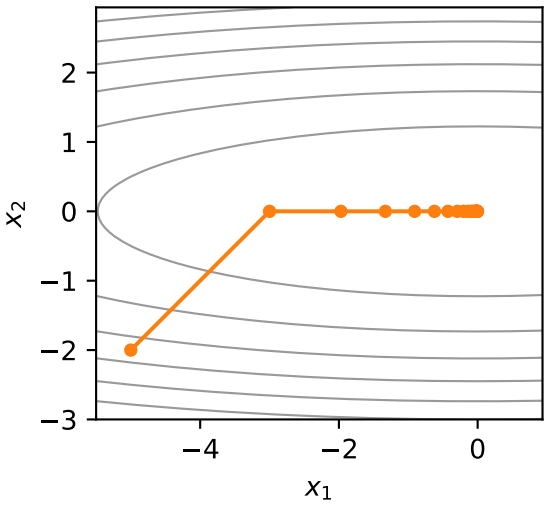
疎または不均一な特徴量を持つモデル（たとえば、頻出トークンは毎回のミニバッチに現れる一方、希少なトークンはごくまれにしか現れないトークン埋め込みなど）では、単一の大域的な学習率ですべての座標に適切に対応することはできない。**AdaGrad** (Duchi et al. 2011) は二乗勾配の累積和 $\mathbf{s}_t$ を要素ごとに蓄え、各座標のステップを $\sqrt{\mathbf{s}_t + \epsilon}$ に反比例させてスケーリングする。

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{9.27}$$

ここで、$\mathbf{s}_0 = \mathbf{0}$ であり、$\epsilon > 0$ はゼロ除算を防ぐための定数である。座標軸に沿った谷においては、各座標をそれ自身の勾配の大きさでスケーリングすることで方向ごとの曲率（curvature）の差が自動的に打ち消され、9.1.5 節の対角前処理行列に対する勾配のみを用いた近似として機能する（35.5.1 節）。

```
def adagrad_2d(x1, x2, s1, s2, f_grad, eta=2.0, eps=1e-6):
    g1, g2 = f_grad(x1, x2)
    s1, s2 = s1 + g1 ** 2, s2 + g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

d2l.show_trace_2d(d2l.f_valley, d2l.train_2d(adagrad_2d, d2l.valley_grad))
```

```
step 20, x1: -0.002295, x2: -0.000000
```



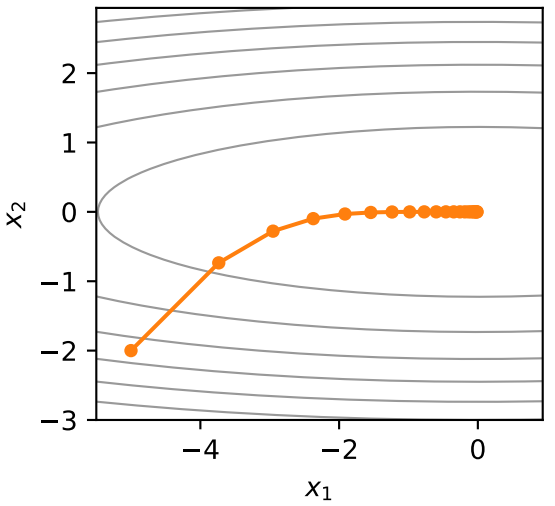
$\mathbf{s}_t$ は二乗勾配を上界なく単調に蓄積するため、持続的な確率的勾配のもとでは AdaGrad の実効ステップサイズが $O(t^{-1/2})$ で減衰し、非凸な深層ネットワークではやがて学習の進行が止まってしまう。**RMSProp** (Tieleman and Hinton 2012) はこの累積和を、二乗勾配の指数移動平均（exponential moving average、EMA）$\mathbf{v}_t$（ウィンドウ幅は $\sim 1/(1-\beta_2)$ ステップ）に置き換える。

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t. \tag{9.28}$$

（補足：標準的な最適化の記法に従い、ここでの $\mathbf{v}_t$ は 9.4 節の速度ベクトルではなく、二次モーメントである二乗勾配の平均を表す。）古い勾配を忘却することで、RMSProp は中程度の学習率 $\eta = 0.4$ からでも消失しないステップサイズを維持する。

```
def rmsprop_2d(x1, x2, v1, v2, f_grad, eta=0.4, beta2=0.9, eps=1e-6):
    g1, g2 = f_grad(x1, x2)
    v1 = beta2 * v1 + (1 - beta2) * g1 ** 2
    v2 = beta2 * v2 + (1 - beta2) * g2 ** 2
    x1 -= eta / math.sqrt(v1 + eps) * g1
    x2 -= eta / math.sqrt(v2 + eps) * g2
    return x1, x2, v1, v2

d2l.show_trace_2d(d2l.f_valley, d2l.train_2d(rmsprop_2d, d2l.valley_grad))
```

```
step 20, x1: -0.010599, x2: 0.000000
```



Equation 9.28 の 2 つの性質に注目しよう。

1. **バイアス補正なしでの初期ステップの増幅：**$\mathbf{v}_0 = \mathbf{0}$ であるため、最初のステップでは $\mathbf{v}_1 = (1-\beta_2)\mathbf{g}_1^2$ となり、初回のジャンプが大きさ $\eta/\sqrt{1-\beta_2} \approx 3.16\,\eta$（$\beta_2 = 0.9$ のとき）、さらには $31.6\,\eta$（$\beta_2 = 0.999$ のとき）と過大になる。
2. **正規化されたステップ長：**ある座標の勾配の大きさがほぼ定常であるときは常に $\sqrt{v_t} \approx |g_t|$ となり、更新量の大きさは $\approx \eta$ となって $|g_t|$ がどれほど小さくなっても変わらない。したがって極小値の近傍では、定学習率の適応的手法は $O(\eta)$ の近傍内で振動し続け、スケジュールによって η を減衰させるまで収まらない（9.7 節）。

9.5.1.2 Adam とバイアス補正

Adam (Kingma and Ba 2015) は、一次モーメント（first moment）の EMA（モメンタム、$\mathbf{m}_t$）と非中心二次モーメント（second moment）の EMA（$\mathbf{v}_t$）を組み合わせ、両方の推定量に厳密

な**バイアス補正**（bias correction）を適用する。$\mathbf{v}_t = (1-\beta_2)\sum_{s=1}^{t}\beta_2^{t-s}\mathbf{g}_s^2$ を初期値 $\mathbf{v}_0 = \mathbf{0}$ から展開すると、重みの総和は $1-\beta_2^t$ となり、1 にはならないことがわかる。そこで $1-\beta_1^t$ と $1-\beta_2^t$ で割ることにより、不偏モーメント推定量 $\hat{\mathbf{m}}_t$ と $\hat{\mathbf{v}}_t$ が得られる。

$$\begin{aligned}\mathbf{m}_t &= \beta_1\,\mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t, & \hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t},\\ \mathbf{v}_t &= \beta_2\,\mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, & \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_2^t}.\end{aligned} \tag{9.29}$$

パラメータ更新では、バイアス補正された一次モーメントを座標ごとの RMS スケール $\sqrt{\hat{\mathbf{v}}_t}$ で割る。

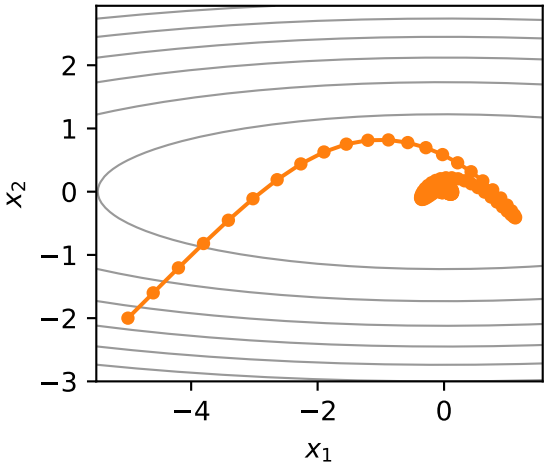
$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}. \tag{9.30}$$

標準的なデフォルト値は $\beta_1 = 0.9$、$\beta_2 = 0.999$（大規模言語モデルの訓練では $\beta_2 = 0.95$ も用いられる (Brown et al. 2020)）、$\epsilon = 10^{-8}$ である。$t=1$ では $\hat{\mathbf{m}}_1 = \mathbf{g}_1$ かつ $\hat{\mathbf{v}}_1 = \mathbf{g}_1^2$ となるため、初回のステップは厳密に $-\eta\ \mathrm{sign}(\mathbf{g}_1)$ となる。

より一般に、目的関数の任意の座標を正の定数 $c>0$ で再スケーリングすると、$\hat{m}_{t,i}$ と $\sqrt{\hat{v}_{t,i}}$ の両方が c 倍されるため、それらの比は（$\epsilon \to 0$ において）不変に保たれる。したがって、**Adam のもとでは、η はパラメータ空間における座標ごとの直接的なステップ長を指定する**ことになる。すなわち、勾配の符号が一貫している座標はステップごとに $\approx \eta$ だけ移動する一方、平均ゼロのノイズが支配的な座標は $\approx \eta\sqrt{(1-\beta_1)/(1+\beta_1)} \approx 0.23\,\eta$ だけ移動する。

```
def adam_2d(x1, x2, s1, s2, f_grad, eta=0.4, beta1=0.9, beta2=0.999,
            eps=1e-8):
    def update(x, g, state):  # state: (m, v, t), 0 before the first step
        m, v, t = state if state else (0, 0, 0)
        m, v = beta1 * m + (1 - beta1) * g, beta2 * v + (1 - beta2) * g ** 2
        m_hat, v_hat = m / (1 - beta1 ** (t + 1)), v / (1 - beta2 ** (t + 1))
        return x - eta * m_hat / (math.sqrt(v_hat) + eps), (m, v, t + 1)
    g1, g2 = f_grad(x1, x2)
    (x1, s1), (x2, s2) = update(x1, g1, s1), update(x2, g2, s2)
    return x1, x2, s1, s2

path = d2l.train_2d(adam_2d, d2l.valley_grad, steps=200, beta2=0.999)
d2l.show_trace_2d(d2l.f_valley, path)
```

```
step 200, x1: 0.000022, x2: 0.000042
```



Adam をゼロから実装するには、各パラメータテンソルに対してステップカウンタ t と 2 つのバッファ $(\mathbf{m}, \mathbf{v})$ を保持する必要がある。

```
class Adam(torch.optim.Optimizer):
    """Adam from scratch: steps scaled by two debiased moving averages."""
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8):
        super().__init__(params, dict(lr=lr, betas=betas, eps=eps))

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            (beta1, beta2), lr, eps = group['betas'], group['lr'], group['eps']
            for p in group['params']:
                if p.grad is None:  # A parameter that received no gradient
                    continue
                state = self.state[p]
                if not state:  # First step: t = 0 and m = v = 0
                    state.update(t=0, m=torch.zeros_like(p),
                                 v=torch.zeros_like(p))
                state['t'] += 1
                t, m, v = state['t'], state['m'], state['v']
                m.mul_(beta1).add_(p.grad, alpha=1 - beta1)
                v.mul_(beta2).add_(p.grad ** 2, alpha=1 - beta2)
                m_hat, v_hat = m / (1 - beta1 ** t), v / (1 - beta2 ** t)
                p.sub_(lr * m_hat / (v_hat.sqrt() + eps))
```

9.5.2 言語モデルにおける Adam

9.5.2.1 TinyLM における勾配スケールと SGD との比較

なぜ Transformer にとって座標ごとのスケーリングがこれほど不可欠なのだろうか。TinyLM（9.3.1 節）全体の勾配スケールを測定するため、まず WikiText103 データセット（$b = 64$、ウィンドウ長 64）を読み込む。

```
data = d2l.WikiText103(batch_size=64, num_steps=64)
```

次に、各パラメータテンソルの二乗平均平方根（RMS）勾配を 10 ミニバッチにわたって計算する `gradients(model, data)`、パラメータ値を複製する `snapshot(model)`、および初期状態の `TinyLM` インスタンスと Adam（$\eta = 3 \times 10^{-3}$）を定義する。

```
def gradients(model, data, num_batches=10):
    """RMS over num_batches minibatches of each parameter's gradient."""
    device = d2l.try_gpu()
    sq = {name: 0 for name, _ in model.named_parameters()}
    for _, (X, Y) in zip(range(num_batches), data.train_dataloader()):
        model.zero_grad()
        loss = F.cross_entropy(model(X.to(device)).flatten(0, 1),
                               Y.to(device).flatten())
        loss.backward()
        for name, p in model.named_parameters():
            sq[name] += p.grad ** 2 / num_batches
    return {name: v.sqrt() for name, v in sq.items()}

def snapshot(model):
    return {name: p.detach().clone() for name, p in model.named_parameters()}

model = d2l.TinyLM(len(data.vocab)).to(d2l.try_gpu())
optimizer = torch.optim.Adam(model.parameters(), lr=3e-3)
```

テンソルを 8 種類の機能別パラメータ群に `rms_by_kind` で分類し、ステップ 0 とステップ 500 における RMS 勾配の大きさを、正規化された Adam のステップ長 $\|\Delta \mathbf{x}\|_{\mathrm{RMS}} / \eta$ と並べて比較する。

```
KINDS = ['token_emb', 'pos_emb', 'qkv', 'proj', 'mlp', 'head']

def rms_by_kind(tensors):
    """RMS of the entries of each kind of parameter, from {name: tensor}."""
    sums, counts = {}, {}
    for name, t in tensors.items():
        kind = (('LayerNorm' if 'norm' in name else 'biases')
                if len(t.shape) == 1 else
                next(k for k in KINDS if k in name))
        sums[kind] = sums.get(kind, 0) + float((t ** 2).sum())
        counts[kind] = counts.get(kind, 0) + math.prod(t.shape)
    return {kind: math.sqrt(sums[kind] / counts[kind]) for kind in sums}

grad_init = rms_by_kind(gradients(model, data))
d2l.train_steps(model, data, optimizer, num_steps=500)
grad_500 = rms_by_kind(gradients(model, data))
before = snapshot(model)
d2l.train_steps(model, data, optimizer, num_steps=1)
after = snapshot(model)
step = rms_by_kind({name: (before[name] - after[name]) / 3e-3
                    for name in before})
print(f'{"":<11}{"RMS gradient at step 0":>24}{"at step 500":>13}'
      f'{"Adam step / lr":>16}')
```

```
    for kind in KINDS + ['LayerNorm', 'biases']:
        print(f'{kind:<11}{grad_init[kind]:>24.1e}{grad_500[kind]:>13.1e}'
              f'{step[kind]:>16.2f}')
    ratio = lambda column: max(column.values()) / min(column.values())
    print(f'{"max / min":<11}{ratio(grad_init):>24.0f}{ratio(grad_500):>13.0f}'
          f'{ratio(step):>16.1f}')
```

```
            RMS gradient at step 0  at step 500  Adam step / lr
token_emb                  2.0e-04      9.2e-05           0.21
pos_emb                    1.8e-04      1.6e-04           0.27
qkv                        8.2e-04      3.8e-04           0.23
proj                       1.4e-03      5.7e-04           0.21
mlp                        1.0e-03      2.6e-04           0.24
head                       4.5e-03      1.3e-03           0.12
LayerNorm                  4.8e-03      1.3e-03           0.18
biases                     7.0e-03      6.2e-04           0.15
max / min                       39           15            2.2
```

生の勾配の大きさは、初期化時点で $40\times$ **の範囲**（トークン埋め込みの $\sim 2 \times 10^{-4}$ からバイアスの $\sim 7 \times 10^{-3}$ まで）に及び、$15\times$ **の範囲**がステップ 500 の時点でも残る。SGD のもとでは、単一のスカラー学習率によって勾配が最大のグループ（バイアス、LayerNorm、出力ヘッド）が埋め込みよりも 15–40× 速く動かされる。一方 Adam のもとでは、8 つすべてのパラメータグループにわたる正規化されたステップサイズは狭い $2\times$ の範囲内（0.1η–0.3η であり、純粋なノイズに対する比 $\sqrt{(1-\beta_1)/(1+\beta_1)} \approx 0.23$ に近い）に収まる。

TinyLM の勾配スケールは 10^{-4}–10^{-2} 付近にあり、かつ Adam の EMA には $(1-\beta_1) = 0.1$ が含まれるため、$\eta \sim 3 \times 10^{-3}$ の Adam は、モメンタム 0.9 かつ $\eta \sim 0.3$ の SGD と同程度の大きさのパラメータステップを生み出す。共通の 3 つのシードにわたって両オプティマイザを比較するため、まずモデルとオプティマイザを生成するクロージャ `make_lm`、`make_sgd`、`make_adam` を用意する。

```
    make_lm = lambda: d2l.TinyLM(len(data.vocab))
    make_sgd = lambda model, lr: torch.optim.SGD(model.parameters(), lr,
                                                 momentum=0.9)
    make_adam = lambda model, lr: torch.optim.Adam(model.parameters(), lr)
    seeds = [random.getrandbits(31) for _ in range(3)]
```

続いて、シードごとに同一の初期重みから出発し、モメンタム付き SGD を $\eta \in \{0.1, 0.3, 0.5, 1.0\}$ にわたって、Adam を $\eta \in \{3 \times 10^{-4}, 10^{-3}, 3 \times 10^{-3}, 10^{-2}, 3 \times 10^{-2}\}$ にわたってスイープする。

```
    def sweep(make_optimizer, lrs):
        """Loss curves of one run per learning rate and seed, printing the final
        losses; the runs of one seed start from the same initialization."""
        runs = [d2l.lr_sweep(make_lm, make_optimizer, data, lrs, seed,
                             verbose=False)[0] for seed in seeds]
        for lr in lrs:
            print(f'  lr {lr:<7g}', *(f'{d2l.final_loss(run[lr]):6.3f}'
                                      for run in runs))
        return {lr: [run[lr] for run in runs] for lr in lrs}
```

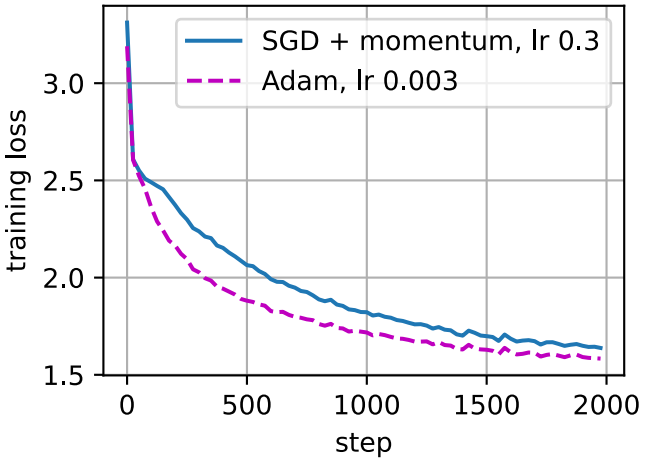
```
    print('final loss of three runs, SGD with momentum')
    sgd_lm = sweep(make_sgd, [0.1, 0.3, 0.5, 1.0])
    print('Adam')
    adam_lm = sweep(make_adam, [3e-4, 1e-3, 3e-3, 1e-2, 3e-2])
```

```
final loss of three runs, SGD with momentum
  lr 0.1      1.835  1.857  1.867
  lr 0.3      1.644  1.900  1.694
  lr 0.5        inf    inf  1.754
  lr 1          inf    inf    inf
Adam
  lr 0.0003   1.930  1.913  1.932
  lr 0.001    1.655  1.662  1.660
  lr 0.003    1.587  1.591  1.587
  lr 0.01     1.642  1.644  1.649
  lr 0.03     2.847  2.781  2.794
```

各オプティマイザを最良の学習率（`best_rate`）で集計し、実用可能な学習率の範囲（最良値から 0.15 nats 以内）、1.8 nats に到達するまでのステップ数、1 ステップでの最大の損失スパイク、および平滑化された損失の軌跡を比較する。

```
    def best_rate(curves):
        """The lr with the lowest mean final loss, and each lr's mean."""
        mean = {lr: sum(map(d2l.final_loss, runs)) / len(runs)
                for lr, runs in curves.items()}
        return min(mean, key=mean.get), mean

    print(f'{"":6}{"best lr":>8}{"final loss":>17}   {"within 0.15 nats":<18}'
          f'{"steps to 1.8":<16}largest one-step rise')
    for name, curves in [('SGD', sgd_lm), ('Adam', adam_lm)]:
        best, mean = best_rate(curves)
        runs = curves[best]
        finals = [d2l.final_loss(run) for run in runs]
        usable = [f'{lr:g}' for lr in curves if mean[lr] <= mean[best] + 0.15]
        steps = ' '.join(str(d2l.steps_to_reach(run, 1.8)) for run in runs)
        rises = ' '.join(f'{max(b - a for a, b in zip(run, run[1:])):.2f}'
                         for run in runs)
        print(f'{name:<6}{best:>8g}{min(finals):>8.3f} to {max(finals):.3f}   '
              f'{" ".join(usable):<18}{steps:<16}{rises}')
    best_sgd, best_adam = best_rate(sgd_lm)[0], best_rate(adam_lm)[0]
    d2l.plot(list(range(0, 2000, 25)),
             [d2l.smooth(sgd_lm[best_sgd][0]),
              d2l.smooth(adam_lm[best_adam][0])],
             'step', 'training loss',
             legend=[f'SGD + momentum, lr {best_sgd:g}',
                     f'Adam, lr {best_adam:g}'])
```

```
       best lr       final loss   within 0.15 nats  steps to 1.8    largest one-step rise
SGD        0.3   1.644 to 1.900   0.1 0.3           1100 None 1250  0.16 1.72 0.39
Adam     0.003   1.587 to 1.591   0.001 0.003 0.01  725 725 700     0.17 0.20 0.16
```



TinyLM において、Adam は目標損失 1.8 nats にモメンタム付き SGD のほぼ半分のステップ数（650–800 対 1,100–1,500）で到達し、より低い最終損失で訓練を終える。さらに $10\times$ の実用可能な学習率ウィンドウ（10^{-3}〜10^{-2}）で動作し、$\eta = 3 \times 10^{-2}$ においてさえ正規化された更新長が η で上から抑えられるため有限に留まる。一方、SGD の最良学習率 $\eta = 0.3$ は発散のわずか 1 グリッド手前に位置しており、1 ステップでの最悪の損失スパイクは Adam の数倍に達し、$\eta = 0.5$ では大半の実行が `inf` へと発散する。サブワード語彙を持つより大規模な Transformer では、ブロック単位のヘッセ行列の不均一性 (Y. Zhang et al. 2024) と裾の重いトークン頻度 (Kunstner et al. 2023, 2024) によって、この差はさらに拡大する。

この実行では、1 回のスパイクによって SGD のあるシードが大きく後退し、最後まで 1.8 に到達しない。

9.5.2.2 ϵ の役割と二次モーメント推定の限界

Equation 9.30 において、ϵ は座標ごとの最大実効学習率を η/ϵ で頭打ちにする。$\epsilon \gg \sqrt{\widehat{v}_{t,i}}$ の極限では、Adam はステップサイズ $(1-\beta_1)\eta/\epsilon$ のモメンタム付き SGD に帰着する。逆に β_2 が小さすぎると、記憶の短い二次モーメント推定（second moment estimate）が、まれに生じる逆向きの大きな勾配よりも、頻繁に現れる小さな勾配に大きな重みを与えてしまうことがある。Reddi et al. (2018) はこの現象を利用して人工的な周期凸反例（$+4, -1, -1$）を構成し、小さな β_2 を持つ Adam が収束しないことを示した（35.5.2.2 節、演習問題 4）。実用上は $\beta_2 \in [0.95, 0.999]$ に保つことで、単調な累積最大値（AMSGrad）を導入しなくてもこの病理を防ぐことができる。

9.5.3 分離型重み減衰（AdamW）

9.5.3.1 結合型 ℓ_2 正則化と分離型重み減衰

単純な SGD（2.7 節）のもとでは、目的関数に ℓ_2 ペナルティ $\frac{\lambda}{2}\|\mathbf{x}\|^2$ を加えることは勾配に $\lambda \mathbf{x}_t$ を加えることになり、これは各ステップの前に重みを $(1-\eta\lambda)$ 倍に縮小（shrinkage）することと数学的に同一である。

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,(\mathbf{g}_t + \lambda \mathbf{x}_t) = (1-\eta\lambda)\,\mathbf{x}_t - \eta\,\mathbf{g}_t. \tag{9.31}$$

しかし、ペナルティ付きの勾配 $\mathbf{g}_t + \lambda \mathbf{x}_t$ を Adam に入力すると、ペナルティ項まで座標ごとの

前処理行列 $\sqrt{\hat{v}_{t,i}} + \epsilon$ で割られてしまう。

$$\underbrace{\frac{\eta\,\lambda}{\sqrt{\hat{v}_{t,i}} + \epsilon}\,x_{t,i}}_{\text{coupled } \ell_2 \text{ penalty in Adam}} \qquad \text{versus} \qquad \underbrace{\eta\,\lambda\,x_{t,i}}_{\text{decoupled weight decay}} \quad . \tag{9.32}$$

結合型 ℓ_2 正則化のもとでは、損失勾配の小さい重みは勾配の大きい重みよりも最大 $10^3\times$ も強く減衰させられる。あるいは、$\lambda|x_{t,i}|$ が損失勾配に対して支配的になると、Adam はペナルティをそれ自身の大きさで正規化し、重みをゼロへ向けてステップあたり速度 η で押し縮め、$\lambda|x| \sim |g_{\text{loss}}|$ になるまでそれが続く。

Loshchilov and Hutter (2019) は、重み減衰を適応的な勾配ステップから分離した **AdamW** を導入した。

$$\mathbf{x}_{t+1} = (1 - \eta\lambda)\,\mathbf{x}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}, \tag{9.33}$$

ここで、$\hat{\mathbf{m}}_t$ と $\hat{\mathbf{v}}_t$ はデータ損失の勾配 $\mathbf{g}_t$ のみから計算される。

```
class AdamW(torch.optim.Adam):
    """Adam with decoupled weight decay: shrink the weights, then step."""
    def __init__(self, params, lr=1e-3, weight_decay=0.01):
        super().__init__(params, lr)  # Adam's own weight_decay stays 0
        self.weight_decay = weight_decay

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is not None:
                    p.mul_(1 - group['lr'] * self.weight_decay)
        super().step()
```

減衰なし、結合型の `Adam + l2`、および分離型の `AdamW` を `TinyLM` 上で比較するため、まず埋め込み以外の 2D 重み行列の要素ごとの RMS を測る `matrix_rms` と、ステップごとの損失および行列の RMS を 250 ステップごとに記録する `train_tracked` を定義する。

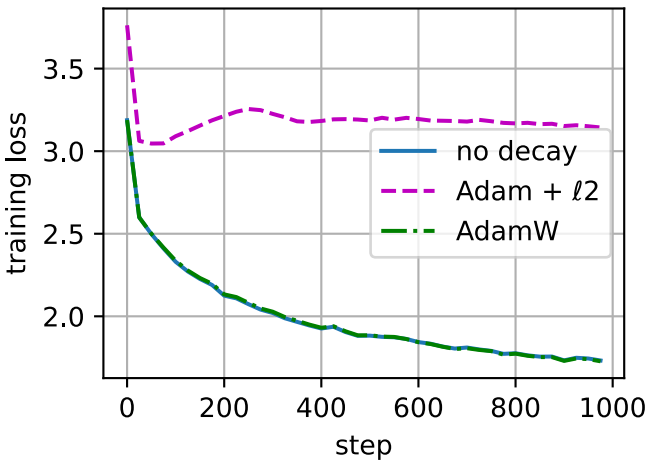
```
def matrix_rms(model):
    """RMS of the entries of the weight matrices, embeddings excluded."""
    W = torch.cat([p.detach().flatten() for name, p in model.named_parameters()
                   if p.ndim == 2 and 'emb' not in name])
    return float(W.square().mean().sqrt())

def train_tracked(make_optimizer, data, seed, num_steps, every=250):
    """Train TinyLM from seed with make_optimizer(params); return the losses,
    the matrix RMS at the start and every `every` steps, and the model."""
    torch.manual_seed(seed)
    model = d2l.TinyLM(len(data.vocab))
    optimizer = make_optimizer(model.parameters())
```

```
    losses, rms = [], [matrix_rms(model)]
    for _ in range(num_steps // every):
        losses += d2l.train_steps(model, data, optimizer, every)
        rms.append(matrix_rms(model))
    return losses, rms, model
```

共通のシードから 3 つの設定すべてを 1,000 ステップ、$\eta = 3 \times 10^{-3}$ および $\lambda = 0.1$ で実行すると、結合型 ℓ_2 ペナルティと分離型重み減衰（decoupled weight decay）の劇的な違いが現れる。

```
seed = random.getrandbits(31)
curves = {}
for name, make_optimizer in [
        ('no decay', lambda params: torch.optim.Adam(params, 3e-3)),
        ('Adam + ℓ2', lambda params: torch.optim.Adam(params, 3e-3,
                                                      weight_decay=0.1)),
        ('AdamW', lambda params: torch.optim.AdamW(params, 3e-3,
                                                   weight_decay=0.1))]:
    curves[name], rms, _ = train_tracked(make_optimizer, data, seed, 1000)
    print(f'{name:<10} final loss {d2l.final_loss(curves[name]):.3f}, RMS of '
          f'the weight matrices {rms[0]:.4f} -> {rms[-1]:.4f}')
d2l.plot(list(range(0, 1000, 25)), [d2l.smooth(c) for c in curves.values()],
         'step', 'training loss', legend=list(curves))
```

```
no decay   final loss 1.740, RMS of the weight matrices 0.0447 -> 0.1078
Adam + ℓ2  final loss 3.151, RMS of the weight matrices 0.0447 -> 0.0028
AdamW      final loss 1.736, RMS of the weight matrices 0.0447 -> 0.0969
```



結合型の `Adam + l2`（$\lambda = 0.1$）は隠れ層の重み行列を初期 RMS の 0.045 から < 0.003 まで崩壊させ、ネットワークの表現を破壊してユニグラムのバイト頻度エントロピー（~ 3.2 nats/byte）で停滞してしまう。一方、分離型の `AdamW` は正則化なしのベースラインと並んで順調に訓練が進みつつ、重みの増大を抑え込んでいる。

9.5.3.2 減衰のタイムスケールと平衡ノルム

Adam の正規化された更新方向を $\mathbf{u}_t = \hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$ と書き、Equation 9.33 を t ステップにわたって展開すると次式が得られる。

$$\mathbf{x}_t = (1-\eta\lambda)^t\,\mathbf{x}_0 - \eta\sum_{s=0}^{t-1}(1-\eta\lambda)^{t-1-s}\,\mathbf{u}_s. \tag{9.34}$$

したがって、AdamW はパラメータ $\mathbf{x}_t$ を、減衰係数 $1-\eta\lambda \approx e^{-\eta\lambda}$ を持つ過去の更新の指数加重移動平均として表しており、その特性**減衰タイムスケール**（decay timescale）は次式で与えられる。

$$\tau = \frac{1}{\eta\lambda}$$

単位はステップ数である (Wang and Aitchison 2025)。$\eta = 3\times10^{-3}$、$\lambda = 0.1$ のとき、減衰タイムスケールは $\tau = 1/0.0003 \approx 3{,}333$ ステップとなる。T ステップの訓練全体では、重み減衰は $T/\tau = T\eta\lambda$ タイムスケールにわたって作用する。

$\tau = 1/(\eta\lambda)$ より長い訓練では、初期重みは減衰して消え去り、各座標 $x_{t,i}$ は確率的な**平衡ノルム**（equilibrium norm）に到達する。そこではステップごとの縮小量 $\eta\lambda x_{t,i}$ と正規化された更新 $-\eta n_{t,i}$（ただし $n_{t,i} = g_{t,i}/\sqrt{\hat{v}_{t,i}}$ は二乗平均が単位大きさ $\sigma^2 \approx 1$）の注入とが釣り合う。SGD のノイズ球（noise ball、9.2 節）において曲率を λ_{decay} に置き換えたものとまったく同じ分散の漸化式により、次式が成り立つ。

$$\begin{aligned} \mathbb{E}[x_{t+1}^2] &= (1-\eta\lambda)^2\,\mathbb{E}[x_t^2] + \eta^2\sigma^2 \\ \Longrightarrow\quad \mathbb{E}[x_\infty^2] &\approx \frac{\eta\,\sigma^2}{2\lambda} \approx \frac{\eta}{2\lambda}. \end{aligned} \tag{9.35}$$

重みの二乗平均平方根の大きさが $\mathrm{RMS}(\mathbf{x}_\infty) \propto \sqrt{\eta/(2\lambda)}$ に安定することで、正規化された重み行列の**相対角度更新率**（relative angular update rate）も $\eta/\mathrm{RMS}(\mathbf{x}_\infty) \propto \sqrt{2\eta\lambda}$ に安定する（**回転平衡**、Kosson et al. (2024)）。これにより、正則化なしの重みが制御不能な $\sqrt{t}$ のランダムウォーク成長を起こし、長期の事前学習において実効的な角度ステップサイズが $1/\sqrt{t}$ で減衰してしまう事態が防がれる (D'Angelo et al. 2024)。η をスイープする際には、ステップごとの縮小率 $s = \eta\lambda$（したがってタイムスケール $\tau = 1/s$）を固定しておけば、学習率の探索を減衰タイムスケールから切り離すことができる (Loshchilov and Hutter 2019; Wang and Aitchison 2025; Bergsma et al. 2025)。

9.5.3.3 重み減衰の対象外となるパラメータとオプティマイザ状態のメモリ

標準的な Transformer の訓練レシピでは、重み行列に分離型重み減衰を適用し、バイアスと正規化パラメータは対象外とする。OLMo 2 はさらに埋め込みテーブルも対象外としており、本書でもその選択を採用する。その結果、重み減衰は **2D の内部射影行列のみ**（`TinyLM` の 90%）に適用され、次の 2 つのパラメータグループは対象外（$\lambda = 0$）となる。

1. **1D のバイアスおよび LayerNorm・RMSNorm のゲイン・シフトベクトル：**正規化のスケールは 1 付近から始まり、特徴量の分散を直接制御する。これらをゼロへ向けて減衰させると、角度更新速度を調整するどころか層の活性化値を歪めてしまう。

2. **トークン埋め込みテーブルと位置埋め込みテーブル（OLMo 2 での除外）：**トークン埋め込みの各行は、そのトークンがミニバッチに現れたときにしか勾配を受け取らないのに対し、分離型重み減衰は毎ステップすべての行を縮小するため、希少なトークンの表現を崩壊させ、下流の LayerNorm の勾配を膨張させてしまう (Team OLMo et al. 2025)。

```
model = d2l.TinyLM(len(data.vocab))
decay = [p for name, p in model.named_parameters()
         if p.ndim == 2 and 'emb' not in name]
no_decay = [p for name, p in model.named_parameters()
            if p.ndim != 2 or 'emb' in name]
optimizer = torch.optim.AdamW([
    {'params': decay, 'weight_decay': 0.1},
    {'params': no_decay, 'weight_decay': 0.0}], lr=3e-3)
n_decay = sum(p.numel() for p in decay)
n = n_decay + sum(p.numel() for p in no_decay)
print(f'decayed: {n_decay} of {n} parameters')
```

```
decayed: 425984 of 470784 parameters
```

9.5.3.4 オプティマイザ状態のメモリ

最後に、パラメータごとに 2 つのバッファ $(\mathbf{m}_t, \mathbf{v}_t)$ を保持することで、静的なパラメータのメモリ使用量は 2 倍になる。32 ビット浮動小数点（スカラーあたり 4 バイト）では、訓練可能な各パラメータに 16 **バイト**のデバイスメモリが必要になる。内訳は、4 バイトがパラメータ $\mathbf{x}$、4 バイトが勾配 $\mathbf{g}$、そして 8 バイトが AdamW の状態 $(\mathbf{m}, \mathbf{v})$ である（13.4 節）。パラメータ数が 10^9 のモデルでは、活性化値を一切保存する前の段階で、オプティマイザ状態だけで 8 GB を消費する。パラメータを凍結する（オプティマイザから除外する）か、$(\mathbf{m}, \mathbf{v})$ を量子化または因子分解することで、このパラメータあたり 8 バイトのオーバーヘッドを直接解消または圧縮できる。

AdamW は重みの各要素を独立に正規化するが、2D 重み行列は活性化値のベクトルに作用する線形作用素として働く。9.6 節では、各 2D 重み行列の更新を要素ごとではなく作用素全体として正規化することで、さらに優れた結果が得られるかを問う。

9.5.4 まとめ

AdaGrad は各座標を累積二乗勾配の平方根に反比例させてスケーリングし、**RMSProp** はその上界のない和を忘却を伴う指数移動平均 $\mathbf{v}_t$ に置き換え、**Adam** はさらに一次モーメントのモメンタム $\mathbf{m}_t$ とバイアス補正 $1-\beta_1^t$、$1-\beta_2^t$ を加える。$\hat{\mathbf{m}}_t$ を $\sqrt{\hat{\mathbf{v}}_t}$ で割ることで、Adam は勾配の対角再スケーリングに対して不変となり、η をパラメータ空間における座標ごとの直接的なステップ長へと変え、生の勾配が 15–40× も異なる Transformer の各パラメータグループ間で更新量の大きさを均一化する。**AdamW** は重み減衰 $\mathbf{x}_{t+1} = (1-\eta\lambda)\mathbf{x}_t - \eta\hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t}+\epsilon)$ を適応的な前処理行列から分離し、一様な減衰タイムスケール $\tau = 1/(\eta\lambda)$、$\sqrt{\eta/(2\lambda)}$ に比例する平衡時の重み RMS、および $\sqrt{2\eta\lambda}$ に比例する定常的な相対角度ステップを確立する。標準的なレシピでは 2D 重み行列のみを減衰させて 1D のバイアスと正規化パラメータ（さらに OLMo 2 に従えば埋め込みテーブル）を対象外とし、$(\mathbf{m}_t, \mathbf{v}_t)$ は `float32` における訓練可能なパラメータあたり 8 バイト（全 16 バイト中）を占める。

9.5.5 演習問題

1. [code] **座標回転と対角スケール不変性**。

1. `adam_2d` を `d2l.f_valley` 上で実行し、目的関数を $c > 0$（ただし $c \gg \epsilon$）で再スケーリングしても、また λ_2 を 4 から 400 に変更しても、Adam が不変であることを検証せよ。
2. 谷を 45° 回転させて $f(\mathbf{x}) = 0.05(x_1 + x_2)^2 + (x_1 - x_2)^2$ とせよ。`adagrad_2d`、`rmsprop_2d`、`adam_2d` を $\eta = 0.4$ で 2,000 ステップにわたり回転前と回転後の谷（および $\lambda_2 = 400$ の場合）で実行し、固有基底を座標軸から回転させると対角前処理行列に対して悪条件性が復活する理由を説明せよ。

2. [code] **定学習率のリミットサイクルと AMSGrad**。●

1. Adam のもとで 1D 座標が $x_t = (-1)^t a$ のように $f(x) = \frac{\lambda}{2}x^2$ 上で交互に振動するとき、その振幅が $a = \frac{1-\beta_1}{1+\beta_1}\frac{\eta}{2}$ となり λ に依存しないことを示せ。
2. `adam_2d` を `f_valley` 上で 20,000 ステップ実行し、反復点が原点の 10^{-40} 以内まで近づく（$\hat{v}_t$ が初期の勾配を記憶している間）にもかかわらず、$\hat{v}_t$ がモメンタムの安定性しきい値を下回って減衰すると弾き出される理由を説明せよ。また、$\hat{v}_t$ を $\max_{s \le t} \hat{v}_s$（AMSGrad）に置き換えるとこの弾き出しが防げることを検証せよ。

3. [extended] **頻出トークン・希少トークンとグループ別学習率**。

1. Kunstner et al. (2024) に従い、256 個のバイト値を `WikiText103` における出現頻度に基づいて頻出トークン（$\geq 0.1\%$）と希少トークン（$< 0.1\%$）に分割し、チューニングされたモメンタム付き SGD、Adam、およびモメンタム付き符号降下法（η sign$(\mathbf{m}_t)$）のもとで各グループの検証損失を比較せよ。
2. モメンタム付き SGD のもとで `TinyLM` の 3 つのパラメータグループ（埋め込み、2D 行列、1D ベクトル）に個別の学習率を割り当て、SGD と Adam の差のうちどれだけがグループ単位のスケーリングと座標単位のスケーリングのそれぞれによって縮まるかを測定せよ。

4. [code] **敵対的な周期勾配と二次モーメントの修正手法**。周期 3 の系列として、$f_t(x) = 4x$（$t \equiv 1 \pmod 3$ のとき）およびそれ以外では $f_t(x) = -x$ と定義された $x \in [-1, 1]$ 上の関数列において (Reddi et al. 2018)、$\beta_1 = 0, \beta_2 = 1/17, \eta_t = 0.5/\sqrt{t}$ で Adam を実行し、AMSGrad および Yogi (Zaheer et al. 2018) と比較せよ。

5. **AdamW の 2 つの極限と回転平衡**。

1. $x_{t+1} = (1 - \eta\lambda)x_t - \eta n_t$ について、決定論的な平衡 $|x_\infty| = 1/\lambda$（$n_t = -1$ が定数のとき）、平均ゼロのノイズ平衡 $\mathbb{E}[x_\infty^2] \approx \eta/(2\lambda)$（$n_t$ が独立同分布で $\mathbb{E}[n_t] = 0, \mathbb{E}[n_t^2] = 1$ のとき）、および混合平衡 $\mathbb{E}[x_\infty^2] \approx \bar{n}^2/\lambda^2 + \eta/(2\lambda)$（$n_t = \bar{n} + \xi_t$ のとき）を導出せよ。
2. $\mathbf{x}_{t+1} = (1 - \eta\lambda)\mathbf{x}_t - \eta\mathbf{u}_t$ を $\mathbb{R}^{1000}$ 上で $\mathbf{u}_t \perp \mathbf{x}_t$ の条件のもとでシミュレーションし (Kosson et al. 2024)、平衡時の RMS が $\sqrt{\eta/(2\lambda)}$ に一致し、ステップごとの回転角が $\sqrt{2\eta\lambda}$ に一致することを検証せよ。

6. [code] (η, λ) **の同時探索とパラメータグループの除外**。●

1. `TinyLM` を 8,000 個の重複ウィンドウ（`d2l.TimeMachine`）上で 800 ステップにわたり、$\eta \in \{10^{-3}, 3 \times 10^{-3}, 10^{-2}\}$ と λ のグリッド全体で訓練し、λ によるパラメータ化とステップごとの縮小率 $s = \eta\lambda$ によるパラメータ化を比較せよ。
2. `WikiText103` 上の検証パープレキシティを $\lambda \in \{0.5, 1.0\}$ において、すべてのパラメータを減衰させる場合と 2D 重み行列のみを減衰させる場合（9.5.3.3 節）とで比較し、改善の大部分をもたらしているのが埋め込みの除外なのか 1D ベクトルの除外なのかを特定せよ。

9.6 Muon

これまでに見てきたすべてのオプティマイザ（SGD、モメンタム、AdamW）は、ニューラルネットワークのパラメータを構造のない平坦なベクトルとして扱っている。AdamW でさえ、2 つのスカラー平均 $(m_{t,ij}, v_{t,ij})$ を各要素 W_{ij}（重み行列 $\mathbf{W}$ の成分）に対して独立に保持しているにすぎない。仮に $\mathbf{W}$ の要素を任意に並べ替えたとしても、AdamW はまったく同じ更新ステップを踏むことになる。

しかし、Transformer のパラメータの 80% 以上は 2 次元の隠れ射影行列 $\mathbf{W} \in \mathbb{R}^{m\times n}$ にあり、これらの行列は層の活性化値 $\mathbf{x} \in \mathbb{R}^n$ を $\mathbf{y} = \mathbf{W}\mathbf{x} \in \mathbb{R}^m$ へ写す線形作用素として働く。重み行列を**線形作用素とみなしたとき**、その最急降下（steepest descent）の方向はどうなるだろうか。

Bernstein and Newhouse (2024) が強調するように、最急降下は更新ステップを抑えるために選ぶノルムに対して相対的に定義される。ステップをユークリッド（ℓ_2）ノルムで測れば SGD の生の勾配方向が得られ、座標ごとの ℓ_∞ ボックスノルムで測れば Adam の記憶なし極限である符号降下法（sign descent）が得られる。その代わりに、更新 $\Delta\mathbf{W}$ が層の出力にもたらし得る最大変化量 $\|\Delta\mathbf{W}\mathbf{x}\|_2/\|\mathbf{x}\|_2$、すなわち**スペクトルノルム** $\|\Delta\mathbf{W}\|_2$ を抑えることにすると、最急降下方向は勾配行列のすべての非ゼロ特異値を 1 に置き換えたものとなり、**直交化勾配**（orthogonalized gradient）$\mathbf{U}\mathbf{V}^\top$ が得られる。

各ステップで厳密な特異値分解（singular value decomposition、SVD）を計算するのは GPU にとって遅すぎるが、5 ステップの奇の行列多項式（**ニュートン・シュルツ反復**、Newton–Schulz iteration）を用いれば、行列積のみを用いて $\mathbf{U}\mathbf{V}^\top$ を近似できる。これを隠れ層の 2 次元重み行列に対するネステロフのモメンタムバッファに適用し（埋め込み、1 次元パラメータ、出力ヘッドは AdamW に任せる）、得られるのが **Muon**（**Momentum Orthogonalized by Newton–Schulz**、Jordan et al. (2024)）である。本節では、ユークリッドノルム球、ℓ_∞ ノルム球、スペクトルノルム球における最急降下を導出し、調整済みの 5 次ニュートン・シュルツ反復を解析したうえで、ハイブリッドな Muon+AdamW オプティマイザをゼロから実装し、`TinyLM` 上で AdamW とベンチマーク比較する。

```
import random
```

9.6.1 ノルムと最急降下

9.6.1.1 ノルム球における最急降下

1 次テイラー展開 $f(\mathbf{x}+\mathbf{d}) \approx f(\mathbf{x}) + \langle \mathbf{g}, \mathbf{d}\rangle$（ただし勾配は $\mathbf{g} = \nabla f(\mathbf{x})$）が与えられたとき、半径 $\eta > 0$、ノルム $\|\cdot\|$ のもとでの最急降下ステップは次の最適化問題を解くことで得られる。

$$\mathbf{d}^\star = \operatorname*{argmin}_{\|\mathbf{d}\| \le \eta} \langle \mathbf{g}, \mathbf{d}\rangle. \tag{9.36}$$

ノルムの幾何構造が異なれば、まったく異なる更新方向が導かれる (Bernstein and Newhouse 2024)（Figure 9.8 および Table 9.3）。

1. **ユークリッド球**（$\|\mathbf{d}\|_2 \le \eta$）：コーシー・シュワルツの不等式により、最小点は負の勾配と同じ方向を向き、$\mathbf{d}^\star = -\eta\,\mathbf{g}/\|\mathbf{g}\|_2$ となって線形化された目的関数を $\eta\|\mathbf{g}\|_2$ だけ減少させる（SGD の方向）。
2. **座標ボックス**（$\|\mathbf{d}\|_\infty = \max_i |d_i| \le \eta$）：$\sum_i g_i d_i$ を制約 $|d_i| \le \eta$ のもとで最小化すると、すべての座標がその境界まで押し出されて $\mathbf{d}^\star = -\eta\ \operatorname{sign}(\mathbf{g})$ となり、線形化された目

的関数は双対な ℓ_1 ノルム $\eta\|\mathbf{g}\|_1$ だけ減少する。これは Adam の記憶なし（$\beta_1 = \beta_2 = 0$）極限である符号降下法にほかならない（9.5 節）。

9.6.1.2 行列とスペクトルノルム

隠れ重み行列 $\mathbf{W} \in \mathbb{R}^{m\times n}$（写像 $\mathbf{x} \mapsto \mathbf{W}\mathbf{x}$）に対し、摂動 $\Delta\mathbf{W}$ を加えると出力の変化は $\|\Delta\mathbf{W}\mathbf{x}\|_2 \le \|\Delta\mathbf{W}\|_2\|\mathbf{x}\|_2$ で抑えられる。ここで、$\|\Delta\mathbf{W}\|_2 = \sigma_1(\Delta\mathbf{W})$ は**スペクトルノルム**（作用素ノルム、1.3.5 節）である。勾配行列 $\mathbf{G} \in \mathbb{R}^{m\times n}$ の簡約特異値分解（33.5 節）を次式で表す。

$$\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top = \sum_{i=1}^{r} \sigma_i\, \mathbf{u}_i\mathbf{v}_i^\top,$$

ここで、$\mathbf{U} \in \mathbb{R}^{m\times r}$ と $\mathbf{V} \in \mathbb{R}^{n\times r}$ は正規直交列をもち、$\sigma_1 \ge \cdots \ge \sigma_r > 0$ である。なお、フロベニウスノルム $\|\mathbf{G}\|_F = \sqrt{\sum_i \sigma_i^2}$（平坦化した行列のユークリッドノルム）は、特異値の大きさが同程度であるとき、スペクトルノルム σ_1 を最大で $\sqrt{\min(m,n)}$ 倍上回り得る。

スペクトルノルム制約 $\|\Delta\mathbf{W}\|_2 \le \eta$ のもとで Equation 9.36 を解くと、次の**直交化勾配**が得られる。

$$\Delta\mathbf{W}^\star = \operatorname*{argmin}_{\|\Delta\mathbf{W}\|_2\le\eta} \langle \mathbf{G}, \Delta\mathbf{W}\rangle = -\eta\,\mathbf{U}\mathbf{V}^\top, \tag{9.37}$$

これは $\mathbf{G}$ のすべての非ゼロ特異値を 1 に設定する操作である。証明は直ちに得られる。任意の行列 $\mathbf{A}$（ただし $\|\mathbf{A}\|_2 \le 1$）に対し、トレースの巡回不変性より次式が成り立つ。

$$\langle \mathbf{G}, \mathbf{A}\rangle = \operatorname{tr}(\mathbf{G}^\top\mathbf{A}) = \operatorname{tr}(\Sigma\,\mathbf{U}^\top\mathbf{A}\mathbf{V}) = \sum_{i=1}^{r} \sigma_i\,(\mathbf{U}^\top\mathbf{A}\mathbf{V})_{ii} \le \sum_{i=1}^{r}\sigma_i,$$

これは、$\|(\mathbf{U}^\top\mathbf{A}\mathbf{V})\|_2 \le \|\mathbf{A}\|_2 \le 1$ から $|(\mathbf{U}^\top\mathbf{A}\mathbf{V})_{ii}| \le 1$ が導かれるためである（演習問題 1）。ここで $\mathbf{A} = -\mathbf{U}\mathbf{V}^\top$ とおけば $\mathbf{U}^\top\mathbf{A}\mathbf{V} = -\mathbf{I}_r$ となり、最小値 $-\eta\sum_i\sigma_i = -\eta\|\mathbf{G}\|_*$（双対ノルムである**核ノルム**、nuclear norm）が達成される。

Adam は**行列の個々の要素**にわたって更新の大きさを揃える（それでもなおランク 1 の方向へ潰れることがあり、たとえば $\operatorname{sign}\begin{bmatrix}3 & 1\\ 1 & 1\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}$ の特異値は 2 と 0 になる）。これに対し、スペクトル最急降下 $-\eta\mathbf{U}\mathbf{V}^\top$ は、等しいステップサイズ η で勾配のすべての有効な特異方向 $(\mathbf{u}_i, \mathbf{v}_i)$ に沿って進む。Figure 9.9 はこれがスペクトルに与える作用を示している。少数の方向に支配された勾配が、すべての有効な方向を等しく重み付けする更新へと変換されるのである。

Table 9.3: 3 つのノルム幾何のもとでの最急降下。

更新に対するノルム球	最急降下方向	線形化減少量（双対ノルム）	適したパラメータ群
ユークリッド球 $(\|\mathbf{d}\|_2 \le \eta)$	$-\eta\,\mathbf{g}/\|\mathbf{g}\|_2$	$\eta\,\|\mathbf{g}\|_2$	非構造化ベクトル（SGD）
座標ボックス $(\|\mathbf{d}\|_\infty \le \eta)$	$-\eta\,\operatorname{sign}(\mathbf{g})$	$\eta\,\|\mathbf{g}\|_1$	非一様な座標・埋め込み（AdamW）

更新に対するノルム球	最急降下方向	線形化減少量（双対ノルム）	適したパラメータ群
スペクトルノルム（$\|\Delta\mathbf{W}\|_2 \le \eta$）	$-\eta\,\mathbf{U}\mathbf{V}^\top$	$\eta\|\mathbf{G}\|_* = \eta\sum_i \sigma_i$	密な 2 次元隠れ重み行列（Muon）

9.6.1.3 行列型の前処理行列（Shampoo）との関連とパラメータの分割

直交化勾配 $\mathbf{U}\mathbf{V}^\top$ は、クロネッカー分解型の行列前処理における記憶なし極限としても現れる。**Shampoo** (Gupta et al. 2018) は左右のグラム行列 $\mathbf{L}_t = \sum_{s\le t}\mathbf{G}_s\mathbf{G}_s^\top$ と $\mathbf{R}_t = \sum_{s\le t}\mathbf{G}_s^\top\mathbf{G}_s$ を蓄積し、$\mathbf{L}_t^{-1/4}\mathbf{G}_t\mathbf{R}_t^{-1/4}$ によって勾配に前処理を施す。これを過去の履歴を蓄積せず単一の勾配 $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$ に対して評価すると、特異値は恒等的に相殺する。

$$(\mathbf{G}\mathbf{G}^\top)^{-1/4}\,\mathbf{G}\,(\mathbf{G}^\top\mathbf{G})^{-1/4} = \mathbf{U}\Sigma^{-1/2}\mathbf{U}^\top\cdot\mathbf{U}\Sigma\mathbf{V}^\top\cdot\mathbf{V}\Sigma^{-1/2}\mathbf{V}^\top = \mathbf{U}\mathbf{V}^\top. \tag{9.38}$$

$\mathbf{L}_t$ と $\mathbf{R}_t$ を保持して逆行列べき乗を計算する代わりに、標準的な一次モーメントのモメンタムバッファに直交化を適用することで、Muon は行列あたりわずか 1 つのモメンタムバッファで全行列前処理の効果を取り込む（35.5.5 節）。

スペクトルノルムの幾何構造は活性化値空間のあいだの密な線形変換に固有のものであるため、Muon は**隠れ層の 2 次元重み行列にのみ**（アテンションの `qkv` や `proj`、および MLP の重み）適用される。ワンホットな行参照を行う埋め込みテーブル、1 次元のバイアスや LayerNorm のスケール、および最終層の語彙読み出しヘッドは AdamW で最適化される (Jordan et al. 2024)。

9.6.2 ニュートン・シュルツ反復による直交化

訓練中に明示的な SVD によって $\mathbf{U}\mathbf{V}^\top$ を計算するのは時間がかかりすぎて実用的ではない。そこで、任意の**奇の行列多項式**（odd matrix polynomial）が $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ の特異ベクトルを保ったまま各特異値に独立に作用することに着目する。

$$p(\mathbf{X}) = a\mathbf{X} + b\,(\mathbf{X}\mathbf{X}^\top)\mathbf{X} + c\,(\mathbf{X}\mathbf{X}^\top)^2\mathbf{X} = \mathbf{U}\,p(\Sigma)\,\mathbf{V}^\top, \tag{9.39}$$

これは $\mathbf{X}\mathbf{X}^\top = \mathbf{U}\Sigma^2\mathbf{U}^\top$ が成り立つためである。まず $\mathbf{X} \leftarrow \mathbf{X}/\|\mathbf{X}\|_F$ と正規化すれば、すべての特異値が $[0,1]$ に収まる。したがって、奇のスカラー多項式 $p(x)$ として $(0,1]$ を 1 に近づけるものを反復適用すれば（**ニュートン・シュルツ反復**、33.5.4.2 節）、高速な行列積のみを用いて $\mathbf{X}$ を直交化できる。

古典的な 3 次多項式 $p(x) = \frac{3}{2}x - \frac{1}{2}x^3$（原点での傾きが $p'(0) = 1.5$ と小さいため、5 ステップでは小さな特異値の持ち上げが遅すぎる）の代わりに、Jordan et al. (2024) は急な初期の傾き $p'(0) = 3.4445$ をもつ調整済みの 5 次多項式を用いている。

$$p(x) = 3.4445\,x - 4.7750\,x^3 + 2.0315\,x^5. \tag{9.40}$$

Figure 9.10 に示すように、Equation 9.40 を 5 回合成すると、$[0.01, 1]$ 内のすべての特異値が狭い帯域 $[0.68, 1.13]$ に持ち上げられる（さらに 0.0015 より大きいすべての特異値も少なくとも 0.68 に達する）。これは 1 への厳密な漸近収束と引き換えに、小さな特異方向を 5 ステップで急速に増幅させる設計である。

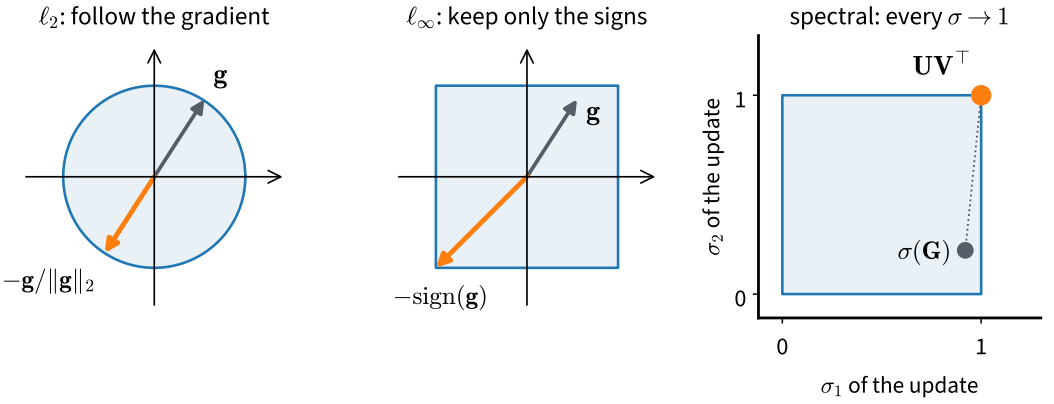


Figure 9.8: 3 つのノルム球のもとでの最急降下方向。ユークリッド ℓ_2 球（勾配方向）、座標ごとの ℓ_∞ ボックス（符号降下法）、およびスペクトルノルム $\|\cdot\|_2$（直交化勾配 $\mathbf{UV}^\top$ であり、有効なすべての特異値が 1 に等しい）。

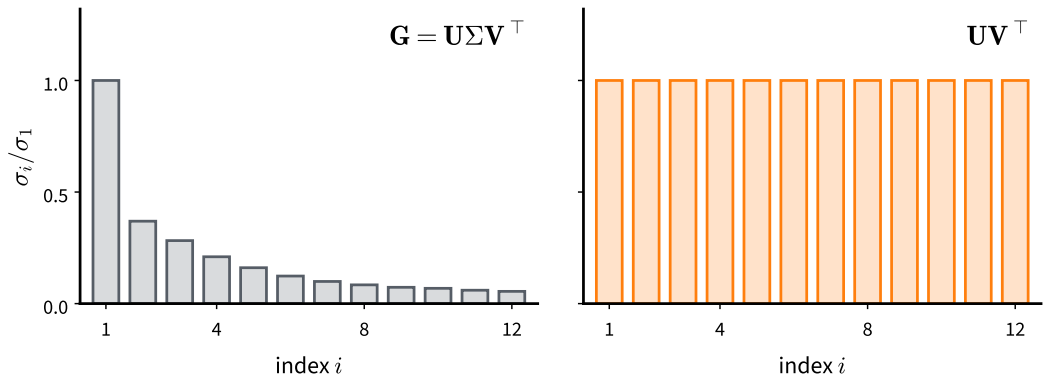


Figure 9.9: 勾配行列 $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$ の最大特異値に対する相対特異値 σ_i/σ_1（左）と、その直交化 $\mathbf{UV}^\top$ の特異値（右）。直交化は特異ベクトルを保ったまますべての非ゼロ特異値を 1 にするため、更新は各特異方向に沿って η ずつ進む。

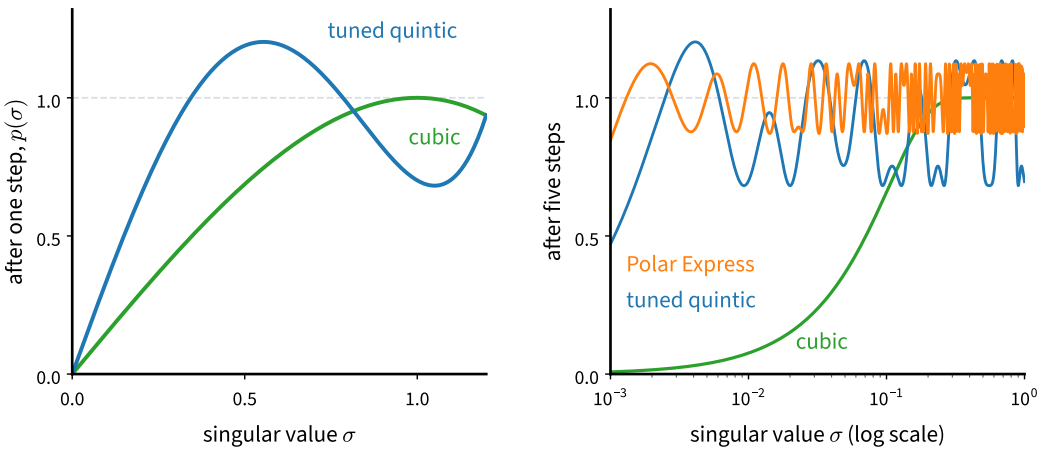


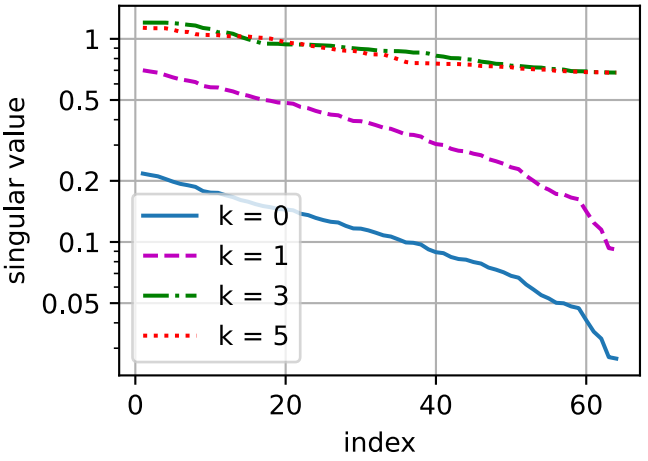
Figure 9.10: 特異値に作用するニュートン・シュルツ多項式。左：古典的な 3 次多項式と調整済み 5 次多項式（Equation 9.40）の 1 回の適用。右：$[10^{-3}, 1]$ 上での 5 回の合成反復（多段階多項式を用いる Polar Express 法 (Amsel et al. 2026) も併記）。

```
def newton_schulz(M, num_iters=5, a=3.4445, b=-4.7750, c=2.0315, eps=1e-7):
    """Approximate the orthogonalization U V^T of M by num_iters steps of
    the odd polynomial a x + b x^3 + c x^5 (Jordan et al.'s tuned quintic)."""
    tall = M.shape[0] > M.shape[1]
    X = M.T if tall else M  # Keep the Gram factor X @ X.T small
    X = X / (X.norm() + eps)
    for _ in range(num_iters):
        A = X @ X.T
        X = a * X + (b * A + c * A @ A) @ X
    return X.T if tall else X
```

$m \times n$ 行列（$m \leq n$）とトークンバッチサイズ B に対し、5 回のニュートン・シュルツ反復にかかる計算コストは $5(4m^2n + 2m^3)$ FLOPs であるのに対し、順伝播と逆伝播は $6mnB$ FLOPs である。すなわちオーバーヘッドは $\sim 5m/B$ であり、$< 1\%$（実用規模のバッチサイズ $B \sim 5 \times 10^5$ のとき）、TinyLM 上でも $\sim 12\%$（$B = 4{,}096$ のとき）にとどまる。newton_schulz を 96×64 のガウス行列に適用すると、反復によって特異値スペクトルが $[0.68, 1.20]$（$k = 3$ の後）、さらに $[0.68, 1.13]$（$k = 5$ の後）へと急速に圧縮される様子が確認できる。

```
G = torch.randn(96, 64)
sigmas = [torch.linalg.svdvals(newton_schulz(G, num_iters=k))
          for k in (0, 1, 3, 5)]
d2l.plot(torch.arange(1, 65), sigmas, 'index', 'singular value',
         legend=[f'k = {k}' for k in (0, 1, 3, 5)], yscale='log')
for k, sigma in zip((3, 5), sigmas[2:]):
    print(f'after {k} iterations: [{sigma.min():.2f}, {sigma.max():.2f}]')
```

```
after 3 iterations: [0.68, 1.20]
after 5 iterations: [0.68, 1.13]
```



実際のネットワークの勾配では、2 つの点が Muon に有利に働く。第 1 に、Muon はノイズの多い単一バッチの勾配ではなく、ネステロフのモメンタムバッファ $\mathbf{G}_t + \beta\mathbf{M}_t$（$\sim 1/(1-\beta) = 20$ ステップにわたる平均、ただし $\beta = 0.95$）を直交化する。第 2 に、フロベニウスノルムの 0.15% 未満の特異値（$3.4445^5 \approx 485$ 倍の増幅では 0.68 に届かない領域）は 1 まで押し上げられるのではなく線形に増幅されるにとどまるため、スペクトル末尾の純粋なノイズ方向が自然に減衰される。

9.6.3 ゼロからの Muon の実装

9.6.3.1 更新式と RMS アライメント

$m \times n$ の隠れ重み行列 $\mathbf{W}$ とその勾配 $\mathbf{G}_t$ に対し、Muon はモメンタムバッファ $\mathbf{M}_t$（$\beta = 0.95$）を更新し、5 回のニュートン・シュルツ反復 NS_5 をネステロフの先読み勾配 $\mathbf{G}_t + \beta\mathbf{M}_t$ に適用するとともに、分離型重み減衰（decoupled weight decay）を組み合わせる (Jingyuan Liu et al. 2025)。

$$\begin{aligned} \mathbf{M}_t &= \beta\,\mathbf{M}_{t-1} + \mathbf{G}_t, \\ \mathbf{W}_{t+1} &= (1 - \eta\lambda)\,\mathbf{W}_t - \eta \cdot 0.2\sqrt{\max(m,n)}\;\mathrm{NS}_5(\mathbf{G}_t + \beta\,\mathbf{M}_t). \end{aligned} \tag{9.41}$$

なぜ $0.2\sqrt{\max(m,n)}$ を掛けるのだろうか。$m \times n$ の直交行列 $\mathbf{UV}^\top$ は $\min(m,n)$ 個の単位特異値をもち、フロベニウスノルムは $\sqrt{\min(m,n)}$、要素ごとの RMS は $\sqrt{\min(m,n)/(mn)} = 1/\sqrt{\max(m,n)}$ となる。したがって $0.2\sqrt{\max(m,n)}$ を掛けることで、行列の次元にかかわらず更新の要素ごとの RMS は $0.2\,\eta$ になる。これは 9.5 節で計測した AdamW の経験的な要素ごとの RMS ステップ幅 $\sim 0.2\,\eta$ と一致するため、両方のオプティマイザで同じ学習率 η と重み減衰 λ を共有できる (Jingyuan Liu et al. 2025)。

```
class Muon(torch.optim.Optimizer):
    """Steepest descent under the spectral norm: orthogonalized momentum."""
    def __init__(self, params, lr, momentum=0.95, weight_decay=0.0):
        super().__init__(params, dict(lr=lr, momentum=momentum,
                                      weight_decay=weight_decay))
```

```
    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:  # A parameter that received no gradient
                    continue
                state = self.state[p]
                if not state:  # First step: allocate the momentum buffer
                    state['buf'] = torch.zeros_like(p)
                buf = state['buf'].mul_(group['momentum']).add_(p.grad)
                G = p.grad + group['momentum'] * buf  # Nesterov momentum
                M = G.reshape(len(G), -1)             # Flattens conv kernels
                O = newton_schulz(M).reshape(p.shape)
                rms_scale = 0.2 * math.sqrt(max(M.shape))  # AdamW's RMS
                p.mul_(1 - group['lr'] * group['weight_decay'])  # Decoupled
                p.add_(O, alpha=-group['lr'] * rms_scale)
```

各処理を確認するために Muon を自ら実装したが、フレームワークにも標準実装が用意されている。その設定は Equation 9.41 の係数 $0.2\sqrt{\max(m,n)}$ とどのように対応しているのだろうか。これを確かめるため、ゼロ初期化した重みと 96×64 のガウス勾配に対して自前の実装とライブラリの実装でそれぞれ 1 ステップを実行し、2 つの更新を比較してみよう。

```
grad, steps = torch.randn(96, 64), []
for make in (lambda p: Muon([p], lr=0.01),
             lambda p: torch.optim.Muon([p], lr=0.01,
                                        adjust_lr_fn='match_rms_adamw')):
    p = nn.Parameter(torch.zeros(96, 64))
    p.grad = grad.clone()
    make(p).step()
    steps.append(p.detach())  # The update, since p started at zero
for name, step in zip(('ours', 'torch.optim.Muon'), steps):
    rms = step.pow(2).mean().sqrt()
    print(f'{name:16s} update RMS / lr = {rms / 0.01:.3f}')
gap = (steps[0] - steps[1]).norm() / steps[0].norm()
print(f'relative difference: {gap:.1e}')
```

```
ours             update RMS / lr = 0.176
torch.optim.Muon update RMS / lr = 0.176
relative difference: 1.3e-02
```

2 つの更新の差は約 1% 以内に収まっており、要素ごとの RMS はいずれも約 $0.18\,\eta$ である。これが公称値の $0.2\,\eta$ をわずかに下回るのは、5 回のニュートン・シュルツ反復後の特異値が 0.68 から 1.13 のあいだに広がっており、厳密な 1 にはならないためである。以下の実験では、本節で作成した自前の実装を引き続き使用する。

`torch.optim.Muon` は 2 次元パラメータのみを受け付けるため、バイアス、正規化パラメータ、埋め込みには依然として別のオプティマイザが必要である。設定 `adjust_lr_fn='match_rms_adamw'` が係数 $0.2\sqrt{\max(m,n)}$ に対応し、デフォルトの `'original'` では代わりに $\sqrt{\max(1, m/n)}$ を掛ける。また、デフォルトの重み減衰は 0.1 である。ライブラリ実装はニュートン・シュルツ反復を bfloat16 で実行するため、2 つの更新のあいだに約 1% の差が生じる。

9.6.3.2 パラメータ群とハイブリッドオプティマイザ

TinyLM において、内部ブロックの 8 個の行列（qkv、proj、mlp1、mlp2 が 2 ブロック分）を Muon に振り分け、残る 22 個のテンソル（埋め込み、LayerNorm ベクトル、バイアス、出力ヘッド）を AdamW に振り分けると、393,216 個のパラメータ（全 470,784 個のうち 84%）が Muon に割り当てられる。Muon はパラメータあたり AdamW の 2 つのモーメントではなく 1 つのモメンタムバッファしか保持しないため、このハイブリッドオプティマイザはオプティマイザ状態の総メモリ使用量を 42% 削減する。

```
data = d2l.WikiText103(batch_size=64, num_steps=64)

def split_lm(model):
    hidden, rest = [], []
    for name, p in model.named_parameters():
        is_hidden = p.ndim == 2 and 'emb' not in name and 'head' not in name
        (hidden if is_hidden else rest).append(p)
    return hidden, rest

hidden, rest = split_lm(d2l.TinyLM(len(data.vocab)))
print(f'Muon:  {len(hidden):>2} tensors, '
      f'{sum(p.numel() for p in hidden):>7} parameters')
print(f'AdamW: {len(rest):>2} tensors, '
      f'{sum(p.numel() for p in rest):>7} parameters')
```

```
Muon:   8 tensors,  393216 parameters
AdamW: 22 tensors,   77568 parameters
```

そこで、隠れ層の 2 次元行列に分離型重み減衰を適用する Muon と、残りのパラメータに重み減衰を適用しない AdamW を、muon_adamw によって単一のハイブリッドオプティマイザにまとめる。

```
class MultiOptimizer:
    """Apply independent optimizers to disjoint parameter groups."""
    def __init__(self, *optimizers):
        self.optimizers = optimizers

    def step(self):
        for opt in self.optimizers:
            opt.step()

    def zero_grad(self):
        for opt in self.optimizers:
            opt.zero_grad()

def muon_adamw(hidden, rest, lr, weight_decay):
    """Muon with decoupled weight decay on hidden, AdamW without it on rest."""
    return MultiOptimizer(Muon(hidden, lr, weight_decay=weight_decay),
                          torch.optim.AdamW(rest, lr, weight_decay=0.0))
```

9.6.3.3 AdamW との比較

共通の初期重みから出発し、$\eta \in \{10^{-3}, 3\times10^{-3}, 10^{-2}, 3\times10^{-2}, 10^{-1}\}$ の範囲および隠れ行列に対する $\lambda = 0.1$ のもとで 2,000 ステップ訓練を行い、250 ステップごとに検証損失を評価し

て Muon+AdamW ハイブリッドを標準の AdamW とベンチマーク比較する。まず、ベースラインとなる adamw のスイープを run_lm で実行する。

```
def val_loss(model):
    """Mean cross-entropy per byte over the validation split."""
    return math.log(d2l.lm_perplexity(model, data.val_dataloader()))

def run_lm(make_optimizer, lrs, seed):
    """d2l.lr_sweep of TinyLM, validated every 250 steps."""
    return d2l.lr_sweep(lambda: d2l.TinyLM(len(data.vocab)), make_optimizer,
                        data, lrs, seed, evaluate=val_loss,
                        metric='validation loss', every=250)

lrs, wd = [1e-3, 3e-3, 1e-2, 3e-2, 1e-1], 0.1

def adamw(model, lr):  # Decay on the hidden matrices only, as for Muon
    hidden, rest = split_lm(model)
    return torch.optim.AdamW([{'params': hidden, 'weight_decay': wd},
                              {'params': rest, 'weight_decay': 0.0}], lr=lr)

seed = random.getrandbits(32)  # One draw, shared by every run below
adamw_lm, adamw_val = run_lm(adamw, lrs, seed)
```

```
lr 0.001: final loss 1.685, validation loss 1.660
lr 0.003: final loss 1.610, validation loss 1.584
lr 0.01: final loss 1.667, validation loss 1.646
lr 0.03: final loss 2.745, validation loss 2.735
lr 0.1: final loss 2.967, validation loss 2.974
```

次に、muon_adamw ハイブリッドについて、まったく同じ学習率グリッド lrs と初期シード seed でスイープを実行する。

```
muon_lm, muon_val = run_lm(
    lambda model, lr: muon_adamw(*split_lm(model), lr, wd), lrs, seed)
```

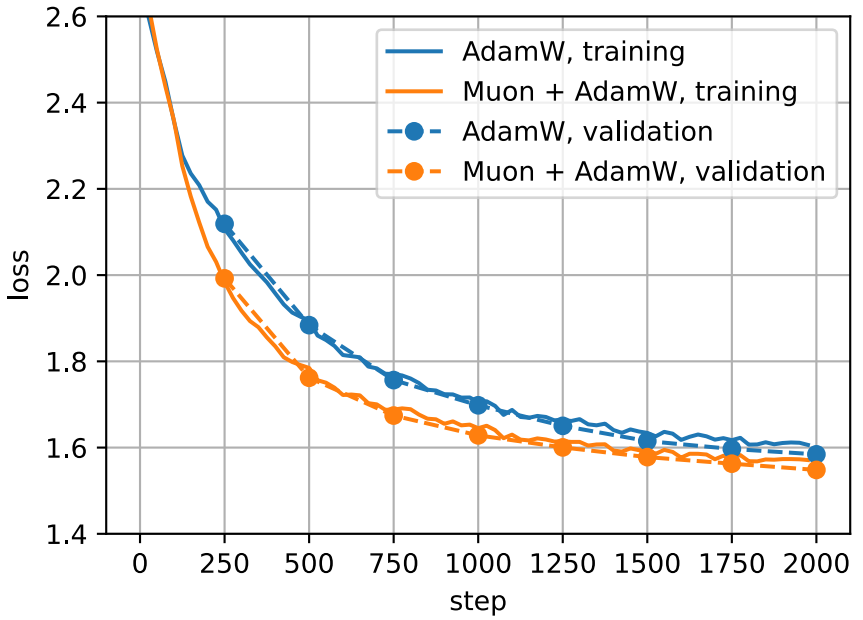
```
lr 0.001: final loss 1.624, validation loss 1.598
lr 0.003: final loss 1.573, validation loss 1.548
lr 0.01: final loss 1.602, validation loss 1.578
lr 0.03: final loss 1.693, validation loss 1.673
lr 0.1: final loss 2.063, validation loss 2.010
```

各オプティマイザを検証損失が最良となった学習率（best_adamw と best_muon）で比較し、平滑化した訓練曲線と検証曲線をプロットするとともに、検証損失が 1.6 nat を下回るステップ数および実用可能な学習率の範囲を表にまとめる。

```
best_adamw = min(adamw_val, key=lambda lr: adamw_val[lr][-1])
best_muon = min(muon_val, key=lambda lr: muon_val[lr][-1])
d2l.plot([list(range(0, 2000, 25))] * 2 + [list(range(250, 2001, 250))] * 2,
         [d2l.smooth(adamw_lm[best_adamw]), d2l.smooth(muon_lm[best_muon]),
          adamw_val[best_adamw], muon_val[best_muon]], 'step', 'loss',
         legend=['AdamW, training', 'Muon + AdamW, training',
                 'AdamW, validation', 'Muon + AdamW, validation'],
```

```
            fmts=('C0-', 'C1-', 'C0o--', 'C1o--'), ylim=(1.4, 2.6),
            figsize=(5, 3.5))
    print('validation loss  lr     step 500  step 2000  below 1.6  within 0.15')
    for name, lr, vals in [('AdamW', best_adamw, adamw_val),
                           ('Muon+AdamW', best_muon, muon_val)]:
        below = d2l.steps_to_reach(vals[lr], 1.6, k=1)  # Checkpoint 1, 2, ...
        near = [r for r in lrs if vals[r][-1] < vals[lr][-1] + 0.15]
        print(f'{name:<16} {lr:<6g} {vals[lr][1]:<9.3f} {vals[lr][-1]:<10.3f} '
              f'{250 * below if below else "never":<10} '
              f'{near[0]:g} to {near[-1]:g}')
```

```
validation loss  lr     step 500  step 2000  below 1.6  within 0.15
AdamW            0.003  1.884     1.584      1750       0.001 to 0.01
Muon+AdamW       0.003  1.762     1.548      1500       0.001 to 0.03
```



TinyLM 上では、Muon+AdamW は検証損失で > 0.1 nat 以上のリードをステップ 500 の時点で築き、1.6 nat のしきい値を AdamW より少なくとも 1 チェックポイント（250 ステップ）早く通過する。さらに、実用可能な学習率の上限を $\eta = 3 \times 10^{-2}$ まで広げる（この学習率では AdamW は 2 nat を大きく上回って終了するが、これは要素ごとの正規化のもとで初期のランク 1 的な勾配が、直交化された更新より最大 $\sqrt{\min(m, n)}$ 倍大きいスペクトルノルムをもち得るためである）。

より大規模な領域では、この優位性は小さくなり、検証事例もまだ限られている。完全な学習率減衰を用いた実験において、Wen et al. (2025) は入念に調整した AdamW に対するステップ数の削減率を計測しており、モデルが大きくなるにつれて約 1.4×（パラメータ数 0.1B）から約 1.1×（1.2B）へと縮小することを報告している。一方、Jingyuan Liu et al. (2025) は独自のスケーリング実験において AdamW の約 2 倍の計算効率を報告している。1 兆パラメータの Kimi K2 は Muon とアテンションロジットのクリッピングを用いて訓練されたが (Kimi Team 2025a)、その規模で比較対象となる AdamW の訓練実験は存在しない。ただし、オプティマイザ状態をほぼ半減できる利点は規模によらず成り立つ。

9.6.4 まとめ

最急降下は、更新を抑えるノルム球に対して相対的に定義される。ユークリッド ℓ_2 球からは SGD が、座標ごとの ℓ_∞ ボックスからは符号降下法（平滑化すると Adam）が、そしてスペクトルノルム球 $\|\Delta\mathbf{W}\|_2 \le \eta$ からは行列作用素のすべての有効な特異方向に沿って等しく進む直交化勾配 $-\eta\mathbf{U}\mathbf{V}^\top$ が得られる。**Muon** は、各隠れ 2 次元重み行列のネステロフのモメンタムバッファ上で、調整済み 5 次ニュートン・シュルツ多項式（Equation 9.40）の 5 回の反復によって $\mathbf{U}\mathbf{V}^\top$ を近似し、更新に $0.2\sqrt{\max(m,n)}$ を掛けて AdamW の要素ごとの RMS ステップサイズに合わせるとともに分離型重み減衰を適用し、埋め込み、1 次元ベクトル、読み出しヘッドは AdamW に任せる。`TinyLM` において、この Muon+AdamW ハイブリッドは初期の収束を加速し、AdamW より $3\times$ 高い学習率でも安定性を保ち、オプティマイザ状態のメモリを 42% 削減する。

9.6.5 演習問題

1. **対角成分の上界とスペクトルステップの一意性**。$\mathbf{U} \in \mathbb{R}^{m\times r}$ と $\mathbf{V} \in \mathbb{R}^{n\times r}$ が正規直交列をもつとする。$\|\mathbf{U}^\top\mathbf{A}\mathbf{V}\|_2 \le \|\mathbf{A}\|_2$ および $|(\mathbf{U}^\top\mathbf{A}\mathbf{V})_{ii}| \le \|\mathbf{A}\|_2$ が成り立つことを示し、さらに Equation 9.37 における最小点 $\Delta\mathbf{W}^\star = -\eta\mathbf{U}\mathbf{V}^\top$ が一意となる条件を求めよ。

2. [code] **経験的な更新 RMS と特異値スペクトル**。`Muon` に計測処理を追加して `TinyLM` 上で実行し、$\Delta\mathbf{W}/\eta$ の要素ごとの RMS を 2,000 ステップにわたって追跡するとともに、単一のミニバッチ勾配 $\mathbf{G}_t$ とモメンタムバッファ $\mathbf{M}_t$ の特異値スペクトルをステップ 0 およびステップ 500 で比較せよ。

3. [code] **ニュートン・シュルツの反復回数と多項式の次数**。調整済み 5 次多項式（Equation 9.40）で `num_iters` $\in \{1,3,5,10\}$ とした場合と、5 回の反復による古典的な 3 次多項式 $p(x) = 1.5x - 0.5x^3$ を `TinyLM` 上で比較し、検証損失と 1 ステップあたりの実経過時間を報告せよ。

4. [code] **埋め込みを AdamW に残す理由**。
   1. $m \times n$ の勾配行列 $\mathbf{M}$ が $r \le n$ 個の非ゼロかつ線形独立な行のみをもつ（ミニバッチ内の r 個の異なるトークンに対応する）とする。このとき、$\mathbf{U}\mathbf{V}^\top$ のすべての有効な行が単位 ℓ_2 ノルムをもつことが、該当トークンの出現回数が 1 回であるか 500 回であるかによらず成り立つことを示せ。
   2. `token_emb` を `TinyLM` の Muon パラメータ群に移し、`token_emb` を AdamW 群に残した場合と比べて検証パープレキシティが悪化することを実験的に確かめよ。

9.7 スケジュール

これまでの節では、訓練全体を通じて学習率 η を一定に保ち、固定されたステップサイズが最適ではない理由を 2 度見てきた。ミニバッチ SGD（9.2 節）では、一定の学習率は反復点を分散 $O(\eta\sigma^2/\lambda)$ の定常なノイズ球（noise ball）に閉じ込める。η を半分にするとノイズフロアも半分になるが、そこに到達するまでに必要な過渡ステップ数は 2 倍になる。AdamW（9.5 節）や Muon（9.6 節）のような正規化オプティマイザでは、勾配が 0 に近づいても更新の大きさは $O(\eta)$ のままであり、η を小さくするまで反復点は最小点に落ち着くことができない。

現代の訓練レシピでは、**学習率スケジュール**（learning rate schedule）η_t をステップ $t = 0, \dots, T-1$ にわたって変化させることで、このトレードオフを解決する。現在のほぼすべてのスケジュールは、0 からピーク学習率 η_0 まで立ち上げる初期の**ウォームアップ**と、それに続く 0 に向けた**減衰**（decay）という 2 つのフェーズを組み合わせている。本節ではまず、ノイズのある二次関数の谷における学習率減衰を、**累積学習率予算**（cumulative learning-rate budget）$B_T = \sum_{t=0}^{T-1} \eta_t$ と**終端学習率**（terminal learning rate）η_T という 2 つの量の観点から解析する。

次に、TinyLM 上で定数、コサイン、およびウォームアップ・安定・減衰（warmup–stable–decay、WSD）の各スケジュールを比較し、高いピーク学習率において線形ウォームアップが初期ステップの不安定性を防ぐ理由を実証する。さらに、一定の学習率を保つプラトーから短い減衰フェーズを分岐させることで、複数の訓練ホライズンにわたって競争力のあるチェックポイントが得られることを示す。

```
import copy
import random
```

9.7.1 ノイズのある谷における学習率減衰

曲率 $\lambda > 0$ を持つ二次関数の谷の単一座標において、平均 0 の勾配ノイズ ξ_t（分散 σ^2）と時間変化するステップサイズ η_t の下での更新を考えよう。

$$x_{t+1} = (1 - \eta_t \lambda)\, x_t - \eta_t\, \xi_t. \tag{9.42}$$

t ステップにわたって、決定論的な平均は $\prod_{s<t}(1-\eta_s\lambda) \approx e^{-\lambda B_t}$ だけ収縮する。ただし、その指数に現れる量は次式で与えられる。

$$B_t = \sum_{s=0}^{t-1} \eta_s \tag{9.43}$$

これが**累積学習率予算**（ステップ t までの学習率曲線の下側の面積）である。η_t が局所的な緩和窓 $1/(\eta_t\lambda)$ にわたって緩やかに変化する限り、分散は瞬間的なノイズ球 $\eta_t\sigma^2/(2\lambda)$ に追随し、次の平均二乗誤差の分解が得られる。

$$\mathbb{E}\big[x_t^2\big] \approx e^{-2\lambda B_t}\, x_0^2 + \frac{\eta_t\, \sigma^2}{2\lambda}. \tag{9.44}$$

Equation 9.44 から、T ステップにわたる任意の有限ホライズンのスケジュールに対する 2 つの要件が明らかになる。

1. **十分な累積予算 B_T**：総予算 $B_T = \sum_{t=0}^{T-1} \eta_t$ は、最も平坦な方向に沿った初期バイアス $e^{-2\lambda_{\min} B_T} x_0^2$ が消失するほど十分に大きくなければならない。
2. **十分な残余予算を伴う小さな終端学習率 η_T**：終端学習率 η_T は、中間ステップ s で注入されたノイズが残余予算 $R_s = \sum_{u=s}^{T-1} \eta_u \gtrsim 1/\lambda_{\min}$ によって収縮されるよう、十分に緩やかに 0 へと減衰しなければならない。

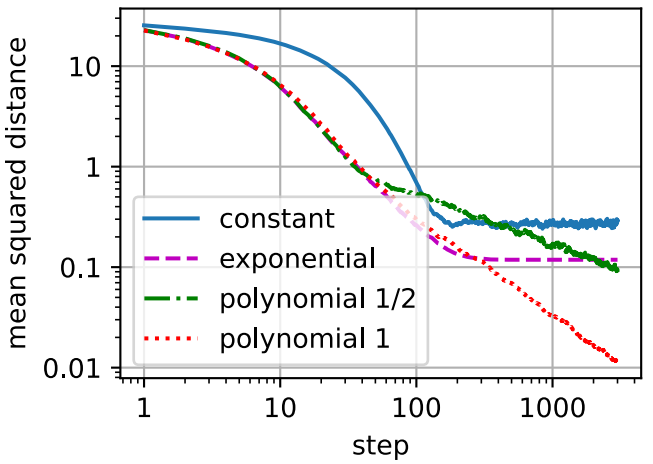
古典的なパラメトリックなスケジュールの 3 つの族として、区分定数減衰（piecewise-constant decay、ステップ減衰）、指数減衰（exponential decay）、多項式減衰（polynomial decay）がある。

$$\begin{aligned} \eta_t &= \eta_i \quad \text{for } t_i \le t < t_{i+1} && \text{(piecewise constant)},\\ \eta_t &= \eta_0\, \gamma^t && \text{(exponential)},\\ \eta_t &= \eta_0\, (1 + t/\tau)^{-\alpha} && \text{(polynomial)}, \end{aligned} \tag{9.45}$$

ここで、$\gamma \in (0, 1)$ は減衰率、$\tau > 0$ は時間スケール、$\alpha > 0$ は指数である。独立した 1,000 回の SGD 実行を谷 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 上で $T = 3{,}000$ ステップにわたりシミュレーションすると、累積予算 B_T と終端学習率 η_T がいかに収束を支配するか、そして急激すぎる指数減衰がなぜノイズを凍結させるかがわかる。

```
def sgd_valley(schedule, num_steps=3000, num_runs=1000):
    """Mean squared distance to the minimum of SGD runs on the valley."""
    rng = np.random.default_rng()
    h = np.array(d2l.VALLEY_CURVATURES)  # Curvatures along x1 and x2
    x = np.tile([-5.0, -2.0], (num_runs, 1))
    dist = []
    for t in range(num_steps):
        x = x - schedule(t) * (h * x + rng.normal(size=x.shape))
        dist.append((x ** 2).sum(axis=1).mean())
    return dist

valley_schedules = {
    'constant': lambda t: 0.1,
    'exponential': lambda t: 0.4 * 0.98 ** t,
    'polynomial 1/2': lambda t: 0.4 / (1 + t / 25) ** 0.5,
    'polynomial 1': lambda t: 0.4 / (1 + t / 25)}
dists = {}
for name, schedule in valley_schedules.items():
    dists[name] = sgd_valley(schedule)
    budget = sum(schedule(t) for t in range(3000))
    print(f'{name:15s} budget {budget:6.1f}, final rate {schedule(2999):.2g}, '
          f'mean squared distance {dists[name][-1]:.3f}')
d2l.plot(np.arange(1, 3001), list(dists.values()), 'step',
         'mean squared distance', legend=list(dists), xscale='log',
         yscale='log')
```

```
constant        budget  300.0, final rate 0.1, mean squared distance 0.286
exponential     budget   20.0, final rate 1.9e-27, mean squared distance 0.118
polynomial 1/2  budget  200.2, final rate 0.036, mean squared distance 0.089
polynomial 1    budget   48.2, final rate 0.0033, mean squared distance 0.012
```



一定の学習率（$\eta = 0.1$）は、定常ノイズ球（$0.25 + 0.016 \approx 0.27$）で頭打ち（プラトー）になる。$\alpha = 1/2$（$\eta_T \approx 0.036$）および $\alpha = 1$（$\eta_T \approx 0.0033$）の多項式減衰は Equation 9.44 に追随し、それぞれ 0.09–0.10 および 0.011–0.012 まで低下する。対照的に、指数減衰（$\gamma = 0.98$）は終端学習率が 10^{-27} に達するにもかかわらず、平均二乗誤差 0.12–0.13 で凍結してしまう。ステップ s 以降の残余予算が $R_s \approx \eta_s/(1-\gamma) = 50\eta_s$ しかないため、$\eta_s \lesssim 0.1$（$R_s\lambda_1 \lesssim 1$）となった後に注入されたノイズは決して収縮して消えることがなく、$(1-\gamma)\sigma^2/(4\lambda_1^2) = 0.125$ の凍結分散が残るからである（演習問題 1）。

9.7.2 言語モデルにおける減衰形状

固定されたステップ予算 T のもとでは、現代の深層学習において次の 2 つの減衰プロファイルが主流となっている。

- **コサイン減衰**（cosine decay）(Loshchilov and Hutter 2016) は、ピーク学習率 η_0（$t = 0$）から終端学習率 η_T（$t = T$）まで半コサイン波をたどる。

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2}\left(1 + \cos\frac{\pi t}{T}\right). \tag{9.46}$$

  $\eta_T = 0$ のとき、コサイン減衰の累積予算は $B_T = \frac{1}{2}\eta_0 T$（一定の学習率 η_0 の予算の半分）となる。
- **安定後減衰（終盤線形減衰）**（stable–then–decay）は、ピーク学習率 η_0 をステップ t_{decay}（ここでは $0.8\,T = 1{,}600$）まで一定に保ち、その後 0 まで訓練の最後の 20% をかけて線形に減衰させる。これにより、一定の学習率の予算の 90%（$B_T = 0.9\,\eta_0 T$）を維持しつつ、$\eta_T = 0$ で訓練を終えることができる。

定数、コサイン、安定後減衰の各スケジュール（$\eta_0 = 3 \times 10^{-3}, T = 2{,}000$）を AdamW（$\lambda = 0$）を用いて `TinyLM` 上で比較するため、まず `WikiText103` を読み込み、モデルのコンストラクタ `tiny_lm(seed)` とスケジュール付きオプティマイザのラッパ関数 `adamw(model, schedule)` を定義する。

```
data = d2l.WikiText103(batch_size=64, num_steps=64)
new_seed = lambda: random.getrandbits(31)

def tiny_lm(seed):
    torch.manual_seed(seed)  # The initialization and the minibatch order
    return d2l.TinyLM(len(data.vocab)).to(d2l.try_gpu())

class Scheduled:
    """An optimizer whose learning rate at step t is schedule(t)."""
    def __init__(self, optimizer, schedule, start=0):
        self.optimizer = optimizer  # Its rate is 1, which LambdaLR multiplies
        self.scheduler = torch.optim.lr_scheduler.LambdaLR(
            optimizer, schedule, last_epoch=start - 1)

    def step(self):
        self.optimizer.step()
        self.scheduler.step()

    def zero_grad(self):
        self.optimizer.zero_grad()

def adamw(model, schedule):
```

```
    """AdamW without weight decay at the learning rate schedule(t)."""
    return Scheduled(torch.optim.AdamW(model.parameters(), lr=1.0,
                                       weight_decay=0), schedule)
```

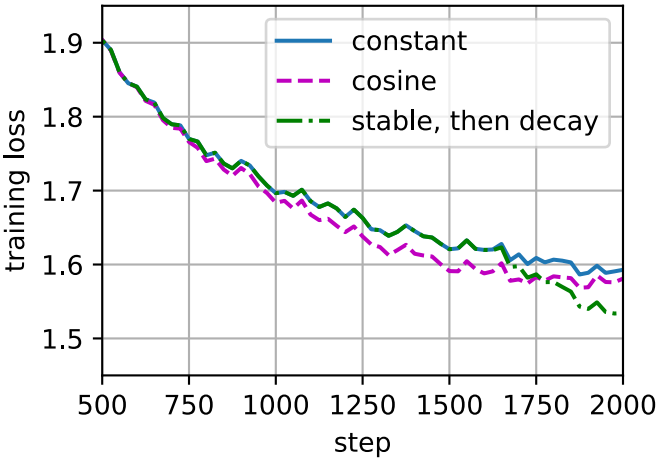
次に、スケジュール関数 constant、cosine、stable_decay（η_0 を decay_start=1600 まで保った後、$T = 2{,}000$ で 0 になるよう線形に減衰させる）と、線形ランプのラッパ関数 warmup を定義する。

```
def constant(t, eta=3e-3):
    return eta

def cosine(t, eta=3e-3, T=2000):
    return eta * (1 + math.cos(math.pi * t / T)) / 2

def stable_decay(t, eta=3e-3, decay_start=1600, T=2000):
    """eta until decay_start, then a linear decay to zero at step T."""
    return eta * min(1, (T - t) / (T - decay_start))

def warmup(schedule, num_steps):
    """schedule, ramped up linearly over its first num_steps steps."""
    return lambda t: min(1, (t + 1) / num_steps) * schedule(t)
```

それでは、2 つのシードにわたって 3 つの減衰プロファイルそれぞれのもとで TinyLM を訓練し、累積予算 B_T と、ステップ 1,500（終盤減衰が始まる前）およびステップ 2,000 における検証損失を記録しよう。

```
schedules = {'constant': constant, 'cosine': cosine,
             'stable, then decay': stable_decay}
val_loss = lambda model: math.log(d2l.lm_perplexity(model,
                                                    data.val_dataloader()))
seeds, curves = [new_seed(), new_seed()], {}
print(f'{"schedule":<20}{"budget":>7}  validation loss at step 1500, 2000')
for name, schedule in schedules.items():
    at_1500, at_2000 = [], []
    for seed in seeds:
        model = tiny_lm(seed)
        optimizer = adamw(model, schedule)
        losses = d2l.train_steps(model, data, optimizer, 1500)
        at_1500.append(val_loss(model))
        losses += d2l.train_steps(model, data, optimizer, 500)
        at_2000.append(val_loss(model))
    curves[name] = d2l.smooth(losses)
    budget = sum(float(schedule(t)) for t in range(2000))
    print(f'{name:<20}{budget:7.2f}  ', ' '.join(f'{v:.3f}' for v in at_1500),
          '  ', ' '.join(f'{v:.3f}' for v in at_2000))
d2l.plot(list(range(25, 2001, 25)), list(curves.values()), 'step',
         'training loss', legend=list(curves), xlim=[500, 2000],
         ylim=[1.45, 1.95])
```

```
schedule             budget  validation loss at step 1500, 2000
constant               6.00   1.609 1.609    1.568 1.564
cosine                 3.00   1.584 1.578    1.558 1.554
stable, then decay     5.40   1.609 1.609    1.507 1.507
```



ステップ 1,500 の時点では、コサイン減衰の学習率がすでにピークの 15% まで低下して高周波の振動を抑えているため、コサイン減衰が優位に立つ。しかし、ステップ 1,600 から 2,000 の間に `stable_decay` が $\eta_t \to 0$ へと線形に減衰させると、その損失は ~ 0.10 nat 急落し、定数スケジュールより 0.05–0.06 nat 低く、累積移動予算が半分しかなかったコサイン減衰よりも ~ 0.05 nat 低い損失で終了する（$B_T = 3.0$ 対 5.4）。なお、学習率減衰が完了する前にチェックポイントを比較すると、訓練終了時の真のランキングが逆転し得る (Wen et al. 2025)。

9.7.3 学習率ウォームアップ

減衰が訓練終盤で反復点を落ち着かせるのに対し、**学習率ウォームアップ**（learning-rate warmup）は η_t を 0 から η_0 まで W ステップ（通常は 1%–5% 程度の割合、総ステップ数は T）かけて線形に立ち上げることで、初期ステップを安定化させる。

なぜウォームアップが必要なのだろうか。第 1 に、Adam（9.5 節）のもとでは、ステップ 1 において単一のミニバッチに基づきすべての座標が $\pm\eta_0$ だけ移動してしまうが、この時点ではモメンタム平均 $\mathbf{m}_t$ と二次モーメント平均 $\mathbf{v}_t$ に複数バッチの統計量がまだ蓄積されていない。初期の線形重み行列が $\mathrm{RMS}(\mathbf{W}_0) \approx 0.045$ である `TinyLM` では、$\eta_0 = 0.03$ でのコールドスタートはすべての重みを初期の大きさの $\sim 67\%$ だけステップ 1 で変化させてしまい、初期化を破壊して訓練を 2.8 nat 付近で停滞させる。第 2 に、9.1.4 節で示したように、η_t を徐々に引き上げると鋭さの上限（勾配降下法では $2/\eta_t$、Adam の前処理後の鋭さでは約 $38/\eta_t$）が段階的に下がり、ピーク学習率に耐えられる平坦な領域へとネットワークを誘導する (Goyal et al. 2017; Kalra and Barkeshli 2024)。

20 ステップの線形ランプ（`warmup(peak, 20)`、全体のわずか 1%、総数 2,000 ステップの実行中）を加えると、初期の相対的な重み変化が約 10% に抑えられて $\eta_0 = 0.03$ での訓練が救われ、最終的な検証損失は約 1 nat 低下する。ステップごとの重み変化を測るため、`linear_weights` で `TinyLM` の埋め込み以外の 2 次元重み行列を抽出する。

```
def linear_weights(model):
    return [p.detach().clone() for name, p in model.named_parameters()
            if p.ndim == 2 and 'emb' not in name]
```

`relative_change` を用いて $\|\Delta\mathbf{W}\|_{\mathrm{RMS}}/\|\mathbf{W}\|_{\mathrm{RMS}}$ をステップ $1, 5, 20,$ および 100 で計算し、$\eta_0 = 0.03$ でのコールドスタートと 20 ステップの線形ウォームアップを 2,000 ステップにわたって比較する。

```
def relative_change(old, new):
    """RMS change of a list of weight matrices relative to their RMS."""
    return math.sqrt(float(sum(((n - o) ** 2).sum() for o, n in zip(old, new))
                           / sum((o ** 2).sum() for o in old)))

peak = lambda t: constant(t, eta=3e-2)
seed = new_seed()
print('step size / weight size at steps 1, 5, 20, 100; final loss')
for name, schedule in [('cold start', peak),
                       ('20-step ramp', warmup(peak, 20))]:
    model = tiny_lm(seed)
    optimizer = adamw(model, schedule)
    losses, sizes = [], []
    for step in range(1, 101):
        before = linear_weights(model)
        losses += d2l.train_steps(model, data, optimizer, 1)
        if step in (1, 5, 20, 100):
            sizes.append(relative_change(before, linear_weights(model)))
    losses += d2l.train_steps(model, data, optimizer, 1900)
    print(f'{name:<14}', ' '.join(f'{s:.2f}' for s in sizes),
          f'  training {d2l.final_loss(losses):.3f},',
          f'validation {val_loss(model):.3f}')
```

```
step size / weight size at steps 1, 5, 20, 100; final loss
cold start     0.67 0.15 0.02 0.01   training 2.862, validation 2.865
20-step ramp   0.03 0.10 0.05 0.01   training 1.809, validation 1.769
```

9.7.4 ウォームアップ・安定・減衰（WSD）とチェックポイント分岐

コサイン減衰 Equation 9.46 では、総ステップホライズン T をステップ 1 の前に固定する必要がある。途中のチェックポイントは依然として高い学習率にあり、T を超えて訓練を延長するにはスケジュールの再開や継ぎ足しが必要になる。短いウォームアップ、ピーク学習率を一定に保つプラトー、および短い最終減衰を組み合わせることで、**ウォームアップ・安定・減衰（WSD）**スケジュールが得られる (Hu et al. 2024; Hägele et al. 2024)。

幾何学的には、Wen et al. (2024) は WSD の軌跡を**河谷**（river valley）のランドスケープ（Figure 9.11）によってモデル化している。高学習率のプラトーの間、反復点は急峻な横断方向の谷壁の間で振動しつつ、平坦な縦断方向の河床に沿って下流へ急速に前進する。そして短い最終減衰が横断方向の振動（曲率 λ_{steep} が大きいため、わずかな残余予算 $R \gtrsim 1/\lambda_{\text{steep}}$ しか必要としない）を減衰させ、反復点を直接谷底へと落ち着かせる。

9.7.4.1 プラトーからの分岐

プラトーフェーズは目標ホライズン T に依存しないため、単一の継続的なプラトー実行を維持したまま、任意のチェックポイントから短い線形クールダウン（250 ステップ）を**分岐**（branch）させることができる。まず、`branch(model, optimizer, schedule)` はモデルの重みと AdamW のモーメントバッファ $(\mathbf{m}_t, \mathbf{v}_t)$ の両方を複製し、クールダウン用の `schedule` のもとで訓練を継続できるようにする。

```
def branch(model, optimizer, schedule):
    """Copies of model and optimizer that continue under schedule."""
```

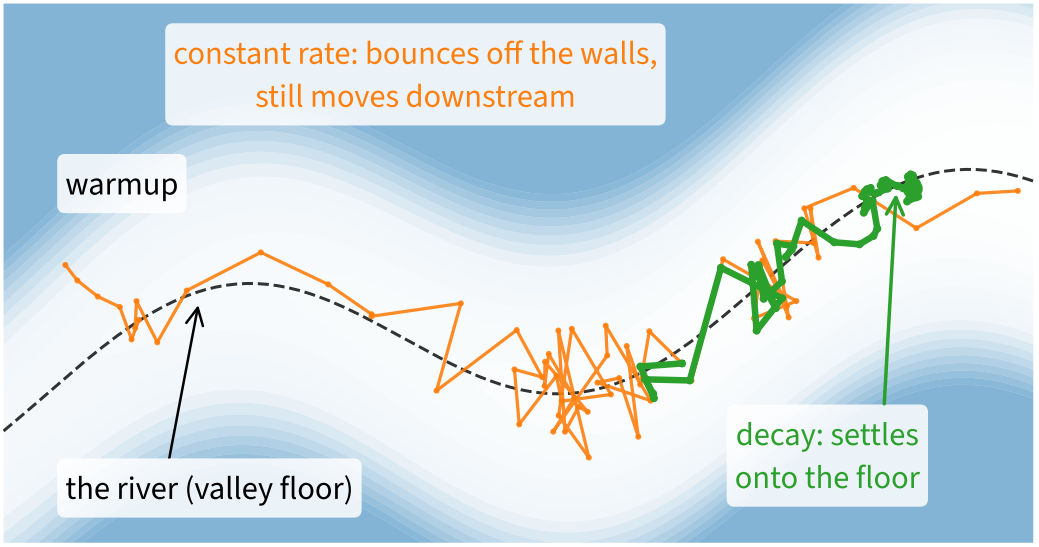


Figure 9.11: ウォームアップ・安定・減衰（WSD）の河谷の幾何学。高い定数プラトーは急峻な横断方向の谷壁の間で振動しながら下流へと急速に前進し、短い減衰ブランチは速やかに谷底へと落ち着く。

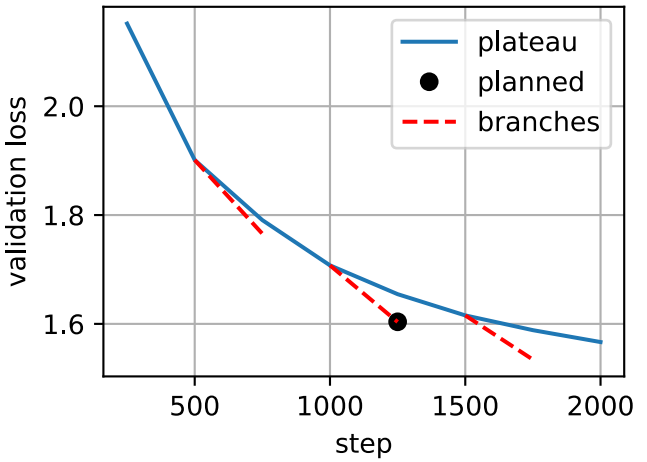
```
    model, adam = copy.deepcopy((model, optimizer.optimizer))  # With moments
    return model, Scheduled(adam, schedule, optimizer.scheduler.last_epoch)
```

それでは、TinyLM 上で 2,000 ステップの定数学習率プラトーを実行し、250 ステップの線形減衰ブランチをステップ 500, 1,000, および 1,500 で分岐させて、ステップ 1,000 → 1,250 のブランチをゼロから計画された専用の 1,250 ステップ実行と比較しよう。

```
seed = new_seed()  # The planned run below starts from the same weights
model = tiny_lm(seed)
optimizer = adamw(model, constant)
plateau, branched = {}, {}
for step in range(250, 2001, 250):
    d2l.train_steps(model, data, optimizer, 250)
    plateau[step] = val_loss(model)
    if step in (500, 1000, 1500):  # A 250-step decay on a copy
        decay = lambda t, s=step: stable_decay(t, decay_start=s, T=s + 250)
        twin, twin_optimizer = branch(model, optimizer, decay)
        d2l.train_steps(twin, data, twin_optimizer, 250)
        branched[step + 250] = val_loss(twin)
planned = tiny_lm(seed)  # Decays from step 1000 to 1250 by design
d2l.train_steps(planned, data, adamw(planned, lambda t: stable_decay(
    t, decay_start=1000, T=1250)), 1250)
print('step  plateau  branch')
for step, v in branched.items():
    print(f'{step:>4} {plateau[step]:8.3f} {v:7.3f}')
planned_loss = val_loss(planned)
```

```
print(f'planned run of 1250 steps: {planned_loss:.3f}')
d2l.plot([list(plateau), [1250]] + [[s - 250, s] for s in branched],
         [list(plateau.values()), [planned_loss]] +
         [[plateau[s - 250], v] for s, v in branched.items()],
         'step', 'validation loss', legend=['plateau', 'planned', 'branches'],
         fmts=('-', 'ko', 'r--', 'r--', 'r--'))
```

```
step  plateau  branch
 750    1.791   1.766
1250    1.655   1.604
1750    1.588   1.534
planned run of 1250 steps: 1.604
```



分岐させたすべてのクールダウンにおいて、検証損失は継続するプラトーよりも急激に低下し（0.05–0.06 nat の低下、ステップ 1,250 および 1,750 時点）、ステップ 1,000 → 1,250 のブランチはゼロから計画された専用の 1,250 ステップ実行に匹敵する。これにより、3 つの訓練ホライズンにわたる完成モデルを、合計 2,250 ステップ（個別実行なら 3,750 ステップ）で得ることができる。

9.7.5 まとめ

学習率スケジュール η_t は、2 つの量によって支配される。1 つは平坦な方向に沿って反復点がどれだけ進むかを決める**累積学習率予算** $B_T = \sum_{t=0}^{T-1} \eta_t$ であり、もう 1 つは最終的なノイズフロアを決定する**終端学習率** η_T である（ただし、以前のノイズを減衰させるために残余予算 $R_s \gtrsim 1/\lambda$ が十分に緩やかに減少する必要がある）。`TinyLM` では、訓練終盤に学習率を 0 まで減衰させることで、定数学習率の AdamW における $O(\eta)$ のステップ揺らぎが除去され、検証損失が 0.05–0.06 nat 低下する。最初の 1%–5% のステップにわたる**線形ウォームアップ**は、$(\mathbf{m}_t, \mathbf{v}_t)$ が安定する前の Adam の過大な初期ジャンプを防ぎ、軌跡を鋭さの上限より下へと導く。最後に、**ウォームアップ・安定・減衰（WSD）**はウォームアップ後に η_0 を一定に保ち、最後の 10%–20% のステップでのみ減衰させることで、移動予算の約 90% を維持しつつ、任意の途中チェックポイントから短いクールダウンブランチを分岐させることを可能にする。

9.7.6 演習問題

1. [code] **指数減衰における凍結ノイズ**。$\eta_s = \eta_0\gamma^s$ および残余予算 $R_s \approx \eta_s/(1-\gamma)$ のもとで $\sum_s \eta_s^2\sigma^2 e^{-2\lambda R_s}$ を積分により評価し、曲率 λ に沿った漸近的な凍結分散が $(1-\gamma)\sigma^2/(4\lambda^2)$ となること（ただし $2\lambda\eta_0 \gg 1-\gamma$ とする）を示せ。また、この予測を `sgd_valley` を用いて $\gamma \in \{0.98, 0.99, 0.995\}$ について検証せよ。

2. [code] **区分定数のステップ減衰**。@eq-scheduler-bias-noise を用いて、3 段階の区分定数スケジュール（3,000 ステップ、`sgd_valley` 上）を設計し、その最終的な平均二乗距離を $\alpha = 1$ の多項式減衰（約 0.01）と比較せよ。

3. [code] **同一の累積予算 B_T と終端学習率 η_T を持つスケジュール**。●全ホライズンの線形減衰 $\eta_t = \eta_0(1-t/T)$ とコサイン減衰 Equation 9.46（$\eta_T = 0$ としたもの。いずれも $B_T = \eta_0 T/2$ かつ $\eta_T = 0$ を満たす）を `sgd_valley`（$\eta_0 = 0.4, T = 3{,}000$）上で比較し、残余予算 R_t を $t \approx T$ の近傍で比較することで、なぜコサイン減衰の方が小さい終端距離で終了するのかを説明せよ。

4. [code] **ウォームアップ期間と許容可能な最大ピーク学習率**。●`TinyLM` において線形ウォームアップの長さを $W \in \{2, 5, 10, 20, 50\}$ ステップにわたって変化させ（$\eta_0 = 0.03$）、安定する最小のウォームアップ長を Adam の一次モーメントの窓幅 $1/(1-\beta_1) = 10$ と比較せよ。さらに、20 ステップのウォームアップによって訓練可能な最大ピーク学習率 η_0 がどれだけ大きくなるかを求めよ。

9.8 バッチサイズ

実際の現場では、ミニバッチサイズ b はアクセラレータのメモリを埋めるように選ばれ、そのまま変更されないことが多い。しかし、訓練を複数の GPU にスケールさせたり、訓練の実経過時間と総計算量（FLOPs）をトレードオフしたりする場面では、根本的な問いに直面する。ミニバッチサイズ b を 2 倍にして（ステップあたりの並列計算量を 2 倍にして）訓練したとき、半分のステップ数で目標の損失に到達するのだろうか。

9.2 節で見たように、b 個の例からなるミニバッチを用いると、勾配のサンプリング分散は $\mathrm{tr}(\Sigma)/b$ に減少する。ここから、次の 2 つの異なる領域が導かれる。

1. **小バッチ（ノイズ支配）領域：**サンプリングノイズが真の勾配より大きいとき、b を 2 倍にするとノイズ分散が半分になり、より大きな学習率を使えるようになる。その結果、必要なステップ数 $S(b)$ は半分に減り、処理される総データ例数 $E(b) = b\,S(b)$ は一定に保たれる（**完全な線形スケーリング**）。
2. **大バッチ（曲率支配）領域：**b が十分に大きくなり、ミニバッチ勾配が真のフルバッチ勾配を精度よく近似するようになると、さらに b を増やしても $S(b)$ の減少幅は逓減する一方、データと計算のコスト $E(b)$ は線形に増大する。

McCandlish et al. (2018) は、これら 2 つの領域の遷移が**臨界バッチサイズ**（critical batch size）b_{crit} で生じること、そしてこの値が**勾配ノイズスケール**（gradient-noise scale）$b_{\mathrm{noise}} = \mathrm{tr}(\Sigma)/\|\nabla f\|^2$ によって予測できることを示した。本節では、b_{noise} の 2 バッチ推定量を構築し、ステップ数とデータ量のトレードオフのフロンティア、ならびに SGD（$\eta \propto b$）と Adam（$\eta \propto \sqrt{b}$）における学習率のスケーリング則を導出し、その両方の予測を `TinyLM` と `FashionCNN` で検証する。

```
import random
```

9.8.1 勾配ノイズスケール

9.8.1.1 信号、ノイズ、2 バッチ推定量

データセット全体の厳密な勾配を $\nabla f(\mathbf{x})$ とし、ミニバッチ勾配 $\hat{\mathbf{g}}_b$ を独立な b 個の例にわたる平均（1 例あたりの勾配の共分散は Σ）とする。$\mathbb{E}[\hat{\mathbf{g}}_b] = \nabla f$ かつ $\mathbb{E}[\|\hat{\mathbf{g}}_b - \nabla f\|^2] = \operatorname{tr}(\Sigma)/b$ であるため、ミニバッチ勾配の二乗ノルムの期待値は、信号パワー $\|\nabla f\|^2$ とノイズパワー $\operatorname{tr}(\Sigma)/b$ の和に分解される。

$$\mathbb{E}\big[\|\hat{\mathbf{g}}_b\|^2\big] = \|\nabla f\|^2 + \frac{\operatorname{tr}\Sigma}{b}. \tag{9.47}$$

信号パワーとノイズパワーは、次の**単純勾配ノイズスケール**（simple gradient-noise scale）において等しくなる (McCandlish et al. 2018)。

$$b_{\text{noise}} = \frac{\operatorname{tr}\Sigma}{\|\nabla f\|^2}. \tag{9.48}$$

$\operatorname{tr}\Sigma$ と $\|\nabla f\|^2$ はいずれも、データセット全体を走査しなくても推定できる。具体的には、二乗勾配ノルムの平均 $N_b = \mathbb{E}[\|\hat{\mathbf{g}}_b\|^2]$ を 2 つのバッチサイズ $b_s < b_l$ で測定し、2 元連立一次方程式 Equation 9.47 を解けばよい。

$$\operatorname{tr}\Sigma \approx \frac{N_{b_s} - N_{b_l}}{1/b_s - 1/b_l}, \qquad \|\nabla f\|^2 \approx \frac{b_l N_{b_l} - b_s N_{b_s}}{b_l - b_s}.$$

まず WikiText-103 と Fashion-MNIST をアクセラレータに読み込み、ランダムなミニバッチサンプラ `sample_fashion(b)` と `sample_lm(b)` を定義する。

```
device = d2l.try_gpu()
data = d2l.WikiText103(batch_size=64, num_steps=64)
fashion = d2l.FashionMNIST(batch_size=256)
X_f = (fashion.train.data.float() / 255).unsqueeze(1).to(device)
y_f = fashion.train.targets.to(device)

def sample_fashion(b):  # b random training images
    i = torch.randint(len(y_f), (b,), device=device)
    return X_f[i], y_f[i]

def sample_lm(b):  # b windows at random offsets in the training text
    starts = torch.randint(len(data.train) - data.num_steps, (b,))
    X, Y = data.windows(data.train, starts.numpy())
    return X.to(device), Y.to(device)
```

```
Downloading ../data/wikitext-103/train-00000-of-00002.parquet from https://huggingface.co/datasets
↪ /Salesforce/wikitext/resolve/main/wikitext-103-raw-v1/train-00000-of-00002.parquet...
Downloading ../data/wikitext-103/validation-00000-of-00001.parquet from
↪ https://huggingface.co/datasets/Salesforce/wikitext/resolve/main/wikitext-103-raw-v1/validati
↪ on-00000-of-00001.parquet...
```

これらのサンプラを用いて、交差エントロピーの補助関数 `ce_loss`、全パラメータにわたる二乗勾配ノルム `grad_sq_norm`、および 2 バッチ推定量 `noise_scale` を `d2l` に保存する。

```
def ce_loss(model, X, Y):
    """Mean cross-entropy of the logits model(X) for the labels Y."""
    logits = model(X)
    return F.cross_entropy(logits.reshape(-1, logits.shape[-1]),
                           Y.reshape(-1))

def grad_sq_norm(model, X, Y):
    """Squared norm of the gradient of ce_loss over all parameters."""
    grads = torch.autograd.grad(ce_loss(model, X, Y),
                                list(model.parameters()))
    return sum((g ** 2).sum() for g in grads)

def noise_scale(model, sample, b_small=16, b_big=2048, m=400):
    """Gradient-noise scale from the mean squared gradient norms of m
    minibatches of size b_small and m // 8 of size b_big from sample(b)."""
    def mean_sq_norm(b, m):  # N_b, averaged over m random minibatches
        return torch.stack([grad_sq_norm(model, *sample(b))
                            for _ in range(m)]).mean()
    n_small, n_big = mean_sq_norm(b_small, m), mean_sq_norm(b_big, m // 8)
    tr_sigma = (n_small - n_big) / (1 / b_small - 1 / b_big)
    sq_norm = (b_big * n_big - b_small * n_small) / (b_big - b_small)
    return (tr_sigma / sq_norm).item()
```

それでは、`noise_scale` を `TinyLM`（$b_s = 8, b_l = 256$）に対して、初期化時点と 500 ステップの Adam（$\eta = 3 \times 10^{-3}$）を実行した後の時点で評価してみよう。

```
lm = d2l.TinyLM(len(data.vocab)).to(device)
print(f'noise scale at initialization: '
      f'{noise_scale(lm, sample_lm, b_small=8, b_big=256):.1f}')
optimizer = torch.optim.Adam(lm.parameters(), lr=0.003)
d2l.train_steps(lm, data, optimizer, num_steps=500)
print(f'after 500 steps of Adam: '
      f'{noise_scale(lm, sample_lm, b_small=8, b_big=256):.0f}')
```

```
noise scale at initialization: 2.2
after 500 steps of Adam: 93
```

画像に対する畳み込みネットワークのノイズスケールも同様に増大するのだろうか。デフォルトの $b_s = 16$ 枚および $b_l = 2{,}048$ 枚の画像を用いて、`FashionCNN` でも初期化時点と 200 ステップの Adam（$\eta = 0.004$、後述のスイープで $b = 256$ のときに用いる学習率）を実行した後の時点で同じ測定を繰り返す。

```
cnn = d2l.FashionCNN().to(device)
print(f'noise scale at initialization: {noise_scale(cnn, sample_fashion):.0f}')
optimizer = torch.optim.Adam(cnn.parameters(), lr=0.004)
d2l.train_steps(cnn, fashion, optimizer, num_steps=200)
print(f'after 200 steps of Adam: {noise_scale(cnn, sample_fashion):.0f}')
```

```
noise scale at initialization: 64
after 200 steps of Adam: 120
```

初期化時点の `TinyLM` では、どのウィンドウも大まかなユニグラム分布への同じ調整を要求するた

め、信号 $\|\nabla f\|^2$ が大きく、$b_{\text{noise}} \approx 2$ 系列にとどまる。頻出バイトの統計量が学習されると（ステップ 500 までには）、$\|\nabla f\|^2$ が低下する一方で系列間の分散 $\operatorname{tr}\Sigma$ は残るため、b_{noise} は 10 倍以上に増大して ~ 50–150 系列となる。

FashionCNN はこれとは異なる挙動を示す。10 個のクラスが均衡しているため、未訓練のネットワークにはすべての画像で一致する調整方向が存在せず、ノイズスケールは最初のステップから 100 枚のオーダーになる。200 ステップの間、推定値の変動は実行ごとに増減いずれの方向にも 2 倍未満にとどまり、100 枚のオーダーのままである。

9.8.2 バッチサイズ、ステップ効率、学習率のスケーリング

9.8.2.1 最適ステップサイズと効率フロンティア

L 滑らかな降下上界 $f(\mathbf{x}-\eta\hat{\mathbf{g}}_b)-f(\mathbf{x}) \le -\eta\nabla f^\top\hat{\mathbf{g}}_b+\frac{L}{2}\eta^2\|\hat{\mathbf{g}}_b\|^2$ の期待値をとり、Equation 9.47 を代入すると次式が得られる。

$$\mathbb{E}\big[f(\mathbf{x}-\eta\,\hat{\mathbf{g}}_b)\big]-f(\mathbf{x}) \le -\eta\,\|\nabla f\|^2+\frac{L}{2}\,\eta^2\Big(\|\nabla f\|^2+\frac{\operatorname{tr}\Sigma}{b}\Big). \tag{9.49}$$

右辺を η について最小化すると、最適ステップサイズ $\eta_{\text{opt}}(b)$ と 1 ステップあたりの保証減少量が得られる。

$$\begin{aligned}\eta_{\text{opt}}(b) &= \frac{1}{L}\cdot\frac{1}{1+b_{\text{noise}}/b},\\ f(\mathbf{x})-\mathbb{E}\big[f(\mathbf{x}-\eta_{\text{opt}}\,\hat{\mathbf{g}}_b)\big] &\ge \frac{\|\nabla f\|^2}{2L}\cdot\frac{1}{1+b_{\text{noise}}/b}.\end{aligned} \tag{9.50}$$

（スカラーの滑らかさの上界 $L\mathbf{I}$ を局所ヘッセ行列 $\mathbf{H}$ に置き換えても、曲率重み付きノイズスケール $\operatorname{tr}(\mathbf{H}\Sigma)/(\nabla f^\top\mathbf{H}\nabla f)$ を用いたまったく同じ形の式が得られ (McCandlish et al. 2018)、これに対して Equation 9.48 の b_{noise} はヘッセ行列不要の代理指標として機能する。）

バッチサイズ b における各ステップは、厳密なフルバッチステップに対して $(1+b_{\text{noise}}/b)^{-1}$ の割合だけ前進するため、フルバッチで $S_{\min}$ ステップを要する目標損失に到達するには、ステップ数 $S(b)$ と総訓練例数 $E(b)=b\,S(b)$ が次式のように必要となる。

$$\frac{S(b)}{S_{\min}} = 1+\frac{b_{\text{noise}}}{b}, \qquad \frac{E(b)}{E_{\min}} = 1+\frac{b}{b_{\text{noise}}}, \tag{9.51}$$

ここで、$E_{\min}=b_{\text{noise}}S_{\min}$ は小バッチ極限 $b \ll b_{\text{noise}}$ で達成される最小データ予算である。**臨界バッチサイズ** $b_{\text{crit}}=b_{\text{noise}}$ においては、ステップ数とデータ例数の双方が理論的最小値の 2 倍 $(S=2S_{\min}, E=2E_{\min})$ となり、パレートフロンティア $((E/E_{\min}-1)(S/S_{\min}-1)=1)$ の屈曲点（knee）をなす。

9.8.2.2 学習率の線形スケーリングと平方根スケーリング

バッチサイズを b_0 から b へ増やすとき、ノイズ支配領域（$b_0, b \ll b_{\text{noise}}$）では学習率 η をどのようにスケーリングすべきだろうか。

- **SGD とモメンタム（線形スケーリング）**：先の Equation 9.50 より、$b \ll b_{\text{noise}}$ のとき $\eta_{\text{opt}}(b) \approx \frac{b}{Lb_{\text{noise}}} \propto b$ となる。言い換えれば、k 回のステップ（サイズ η_0、バッチサイズ

b_0）は、サイズ $k\eta_0$、バッチサイズ kb_0 による 1 回のステップと一致し、同時に定常ノイズフロア $\propto \eta/b$ も保たれる (Goyal et al. 2017; Smith et al. 2018)。

- **Adam と AdamW（平方根スケーリング）**：ある座標の勾配が平均 $\bar{g}$ と標準偏差 $\sigma/\sqrt{b}$ を持つノイズ支配領域では、Adam は勾配をその RMS スケール $\sqrt{\hat{v}_t} \approx \sigma/\sqrt{b}$ で正規化するため、1 ステップあたりのドリフト $-\eta(\sqrt{b}/\sigma)\bar{g}$ とステップノイズ $-\eta z$（$z \sim \mathcal{N}(0,1)$）が生じる。分母の $\sigma/\sqrt{b}$ がすでに $\sqrt{b/b_0}$ 倍の因子を与えるため、$k = b/b_0$ 回の小ステップのドリフトと累積分散に 1 回の大きなステップで一致させるには、η を $\sqrt{b/b_0}$ 倍にスケーリングする必要がある (Malladi et al. 2022)。

$$\begin{aligned} \eta(b) &= \frac{b}{b_0}\,\eta_0 \quad \text{(SGD)}, \\ \eta(b) &= \sqrt{\frac{b}{b_0}}\,\eta_0 \quad \text{(Adam / AdamW)}. \end{aligned} \tag{9.52}$$

b が b_{crit} を超えると、勾配ノイズはもはや分母で支配的ではなくなり、最適学習率は b とともに増大し続けるのではなく、決定論的な曲率による上限で飽和する (Shallue et al. 2019)。

9.8.3 実験的検証：目標到達ステップ数とスケーリング則

固定したホールドアウトの目標損失に到達するまでのステップ数 $S(b)$ を測定するため、まず `eval_loss` を定義し、検証テンソル上の `ce_loss` を 4,096 例ごとのチャンクに分けて評価できるようにする。

```
def eval_loss(model, X, Y, chunk=4096):
    with torch.inference_mode():
        return torch.stack([ce_loss(model, X[i:i + chunk], Y[i:i + chunk])
                            for i in range(0, len(Y), chunk)]).mean().item()
```

停止ハーネス `train_to_target` は、`eval_loss` を `eval_every` ステップごとに `d2l.train_steps` 内で確認し、`target` に到達したステップ数を返す（`max_steps` を使い切った場合は `inf` を返す）。

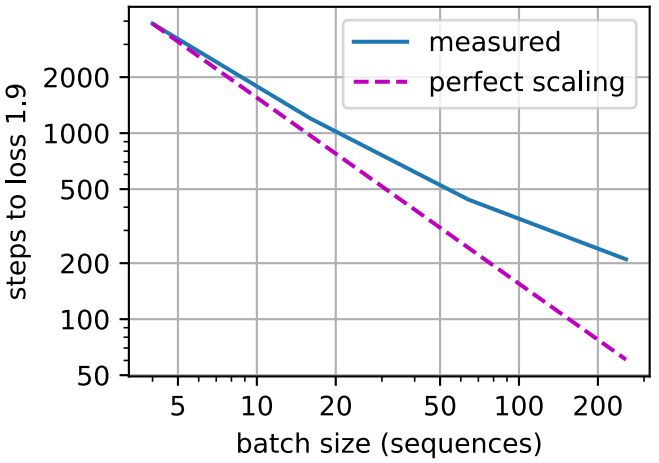
```
def train_to_target(model, data, optimizer, X_eval, Y_eval, target,
                    max_steps, eval_every=10):
    """Return the number of steps until the eval loss reaches target."""
    def reached(step):
        return (step % eval_every == 0
                and eval_loss(model, X_eval, Y_eval) <= target)
    steps = len(d2l.train_steps(model, data, optimizer, max_steps,
                                stop=reached))
    return steps if reached(steps) else float('inf')
```

まず共通の初期化から、`train_to_target` を `TinyLM`（目標検証損失 1.9 nats/byte）に対して $b \in \{4, 16, 64, 256\}$ 系列で実行し、Adam の学習率を $\eta = 0.003\sqrt{b/64}$ に従ってスケーリングする。

```
starts = np.arange(0, len(data.val) - data.num_steps, data.num_steps)
X_val, Y_val = (t.to(device) for t in data.windows(data.val,
```

```
                                                starts[::8][:2048]))
lm_bs, lm_steps = [4, 16, 64, 256], []
seed = random.getrandbits(32)  # One initialization for all batch sizes
for b in lm_bs:
    torch.manual_seed(seed)
    data.batch_size = b
    model = d2l.TinyLM(len(data.vocab)).to(device)
    optimizer = torch.optim.Adam(model.parameters(),
                                 lr=0.003 * math.sqrt(b / 64))
    lm_steps.append(train_to_target(model, data, optimizer, X_val, Y_val,
                                    target=1.9, max_steps=8000))
    print(f'b={b:4d}  steps={lm_steps[-1]:6.0f}  '
          f'examples={b * lm_steps[-1]:8.0f}')
d2l.plot(lm_bs, [lm_steps, [lm_steps[0] * lm_bs[0] / b for b in lm_bs]],
         'batch size (sequences)', 'steps to loss 1.9', xscale='log',
         yscale='log', legend=['measured', 'perfect scaling'])
```

```
b=   4  steps=  3880  examples=   15520
b=  16  steps=  1200  examples=   19200
b=  64  steps=   440  examples=   28160
b= 256  steps=   210  examples=   53760
```

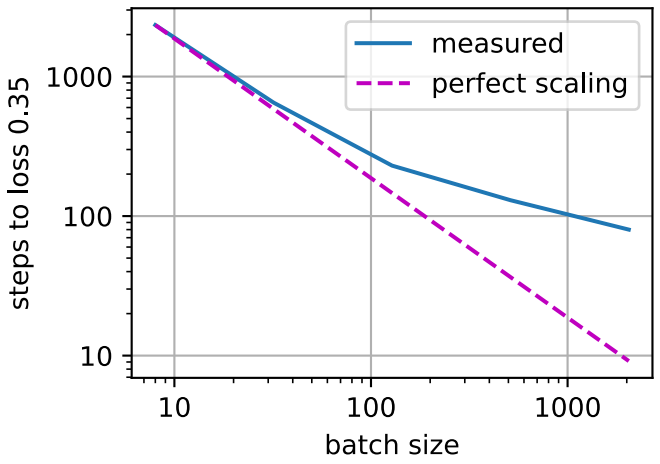


同じ目標到達ステップ数のスイープを FashionCNN（目標損失 0.35）に対しても $b \in \{8, 32, 128, 512, 2048\}$ 枚の画像で繰り返し、学習率を $\eta = 0.004\sqrt{\min(b, 512)/256}$ として $b = 512$ までスケーリングする。

```
cnn_bs, cnn_steps = [8, 32, 128, 512, 2048], []
seed = random.getrandbits(32)  # One initialization for all batch sizes
for b in cnn_bs:
    torch.manual_seed(seed)
    fashion_b = d2l.FashionMNIST(batch_size=b)
    model = d2l.FashionCNN().to(device)
    lr = 0.004 * math.sqrt(min(b, 512) / 256)  # Held fixed above 512
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    cnn_steps.append(train_to_target(model, fashion_b, optimizer,
```

```
                                    X_f[:8192], y_f[:8192], target=0.35,
                                    max_steps=8000))
    print(f'b={b:4d}  steps={cnn_steps[-1]:6.0f}  '
          f'examples={b * cnn_steps[-1]:8.0f}')
d2l.plot(cnn_bs, [cnn_steps,
                  [cnn_steps[0] * cnn_bs[0] / b for b in cnn_bs]],
         'batch size', 'steps to loss 0.35', xscale='log', yscale='log',
         legend=['measured', 'perfect scaling'])
```

```
b=   8  steps=  2340  examples=   18720
b=  32  steps=   650  examples=   20800
b= 128  steps=   230  examples=   29440
b= 512  steps=   130  examples=   66560
b=2048  steps=    80  examples=  163840
```



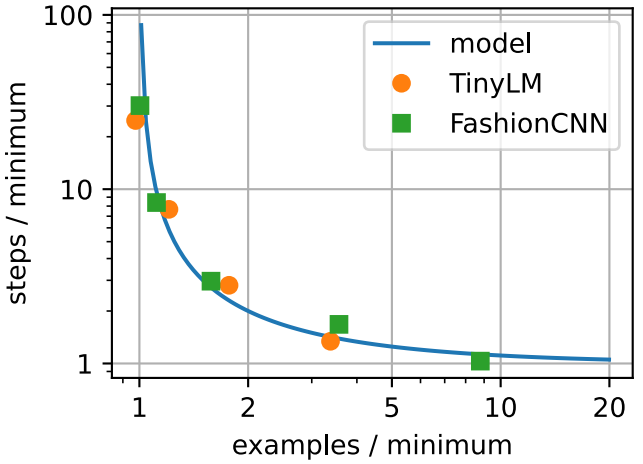
$S(b) = S_{\min}(1 + b_{\text{crit}}/b)$ は $1/b$ に関して線形であるため、`fit_steps` は重み付き最小二乗法によって両モデルの漸近的な最小ステップ数 $S_{\min}$ と実験的な臨界バッチサイズ b_{crit} を抽出する。

```
def fit_steps(bs, steps):
    """Fit S(b) = S_min (1 + b_crit / b) by least squares in 1/b."""
    b, S = np.array(bs, dtype=float), np.array(steps, dtype=float)
    ok = np.isfinite(S)  # Leave out runs that never reached the target
    slope, S_min = np.polyfit(1 / b[ok], S[ok], 1, w=1 / S[ok])
    assert S_min > 0, 'step counts never level off: no floor to fit'
    return S_min, slope / S_min

fits = {'TinyLM': fit_steps(lm_bs, lm_steps),
        'FashionCNN': fit_steps(cnn_bs, cnn_steps)}
for name, (S_min, b_crit) in fits.items():
    print(f'{name:10s}  S_min = {S_min:4.0f} steps  b_crit = {b_crit:4.0f}')
```

```
TinyLM      S_min =  157 steps  b_crit =  102
FashionCNN  S_min =   78 steps  b_crit =  241
```

各実行のデータ例数 $E(b) = b\,S(b)$ を $E_{\min} = b_{\text{crit}} S_{\min}$ で正規化し、ステップ数を $S_{\min}$ で正規化することで、両モデルを普遍的なパレートフロンティア Equation 9.51 と重ねてプロットする。

```
x = np.logspace(0.005, 1.3, 100)  # E / E_min along the model curve
xs, ys = [x], [1 + 1 / (x - 1)]
for (S_min, b_crit), bs, steps in zip(fits.values(), (lm_bs, cnn_bs),
                                      (lm_steps, cnn_steps)):
    xs.append([b * S / (b_crit * S_min) for b, S in zip(bs, steps)])
    ys.append([S / S_min for S in steps])
d2l.plot(xs, ys, 'examples / minimum', 'steps / minimum', xscale='log',
         yscale='log', fmts=('-', 'o', 's'),
         legend=['model', 'TinyLM', 'FashionCNN'])
```



いずれのアーキテクチャも普遍曲線 $(E/E_{\min}-1)(S/S_{\min}-1)=1$ の近くに並ぶ（$E=E_{\min}$ に近い点は曲線の急峻な壁の上に位置するため、測定されたステップ数のわずかな誤差によって上下に大きく動く）。当てはめられた臨界バッチサイズは、TinyLM では $b_{\text{crit}}\sim 100$ 系列、FashionCNN では $b_{\text{crit}}\sim 200$–400 枚である。TinyLM では、この当てはめ結果は先ほど測定した b_{noise} の 2 バッチ推定値（100 系列をやや下回る値）とよく一致する。一方、FashionCNN では約 100 枚という推定値の 2〜4 倍となっている。単純ノイズスケールは曲率を無視するため、b_{crit} の正確な値ではなくオーダー（桁）を与えるのである。

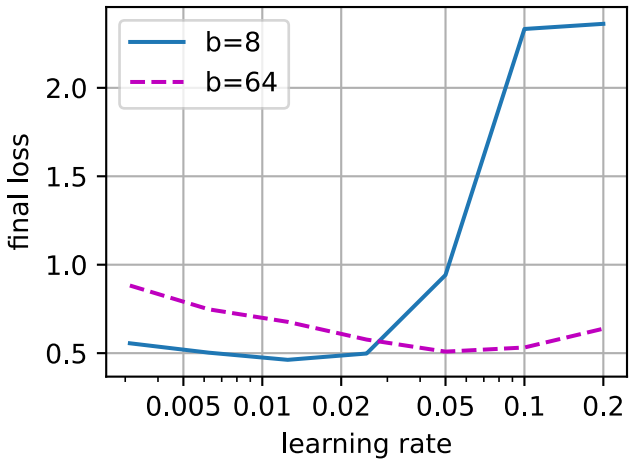
最後に、線形 $(\eta\propto b)$ と平方根 $(\eta\propto\sqrt{b})$ の学習率スケーリング則 Equation 9.52 を、FashionCNN 上で 12,800 枚（1/5 エポック）の固定データ予算のもと直接検証する。補助関数 `final_loss_at` は、`d2l.lr_sweep` を `num_examples // b` ステップだけバッチサイズ `b` で実行し、各実行の最後の 1/4 区間にわたって平均した最終損失を返す。

```
def final_loss_at(make_optimizer, lrs, b, seed, num_examples=12800):
    """Train the CNN once per learning rate; return the final losses."""
    curves, _ = d2l.lr_sweep(d2l.FashionCNN, make_optimizer,
                             d2l.FashionMNIST(batch_size=b), lrs, seed,
                             num_examples // b, verbose=False)
    return [d2l.final_loss(c, k=len(c) // 4) for c in curves.values()]
```

モメンタム付き SGD（$\beta=0.9$）では、バッチサイズを 8×（$b=8$ から $b=64$ へ）増やして `sgd_lrs` にわたり探索すると、最適学習率は右へ 4–8× シフトする（$b=64$ では最終損失が $\eta=0.05$ から 0.1 の間でほぼ平坦になる）。

```
sgd_lrs = [0.003125, 0.00625, 0.0125, 0.025, 0.05, 0.1, 0.2]
make_sgd = lambda model, lr: torch.optim.SGD(model.parameters(), lr,
                                             momentum=0.9)
seed = random.getrandbits(32)
sgd_small = final_loss_at(make_sgd, sgd_lrs, b=8, seed=seed)
sgd_big = final_loss_at(make_sgd, sgd_lrs, b=64, seed=seed)
for b, loss in ((8, sgd_small), (64, sgd_big)):
    print(f'b={b:2d}: best lr {sgd_lrs[np.nanargmin(loss)]:g}')
d2l.plot(sgd_lrs, [sgd_small, sgd_big], 'learning rate', 'final loss',
         xscale='log', legend=['b=8', 'b=64'])
```

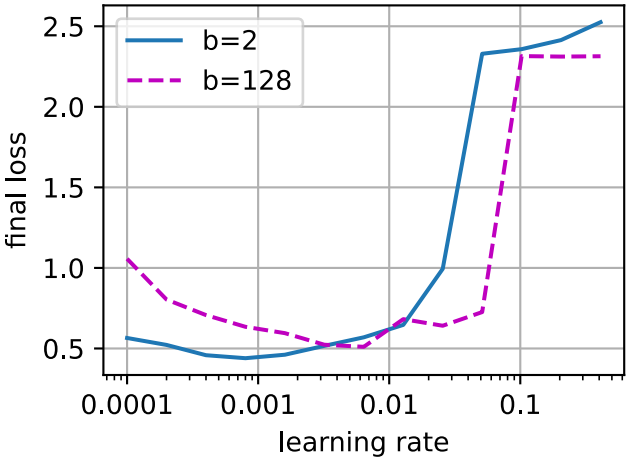
```
b= 8: best lr 0.0125
b=64: best lr 0.05
```



対照的に、Adam のもとでバッチサイズを 64×（$b = 2$ から $b = 128$ へ）増やして `adam_lrs` にわたり探索すると、最適学習率のシフトはわずか 4–8×（2 から 3 段階分の 2× グリッドステップ）にとどまり、$\sqrt{64} = 8\times$ と一致して 64× にはならない。

```
adam_lrs = [0.0001 * 2 ** k for k in range(13)]  # 0.0001 to 0.4096
make_adam = lambda model, lr: torch.optim.Adam(model.parameters(), lr)
seed = random.getrandbits(32)
adam_small = final_loss_at(make_adam, adam_lrs, b=2, seed=seed)
adam_big = final_loss_at(make_adam, adam_lrs, b=128, seed=seed)
i, j = int(np.nanargmin(adam_small)), int(np.nanargmin(adam_big))
print(f'best lr: {adam_lrs[i]:g} at b=2; {adam_lrs[j]:g} at b=128, '
      f'final loss {adam_big[j]:.3f}')
for rule, k in (('no scaling', 0), ('square root', 3), ('linear', 6)):
    if i + k < len(adam_lrs):
        print(f'{rule:11s}: lr {adam_lrs[i + k]:g} at b=128, '
              f'final loss {adam_big[i + k]:.3f}')
    else:
        print(f'{rule:11s}: lr {adam_lrs[i] * 2 ** k:g}, beyond the grid')
d2l.plot(adam_lrs, [adam_small, adam_big], 'learning rate', 'final loss',
         xscale='log', legend=['b=2', 'b=128'])
```

```
best lr: 0.0008 at b=2; 0.0064 at b=128, final loss 0.510
no scaling : lr 0.0008 at b=128, final loss 0.635
square root: lr 0.0064 at b=128, final loss 0.510
linear     : lr 0.0512 at b=128, final loss 0.726
```



9.8.3.1 バッチサイズを選択する実践的手順

1. **まずハードウェアの演算器を飽和させる：**1 ステップあたりの実行時間が b に対して線形に増加し始めるまで、少なくとも b を大きくする（9.2 節）。
2. **ウォームアップ後に b_{noise} を推定する：**初期の損失低下の後（たとえば訓練開始から数百ステップ後）に `noise_scale` を測定する。`TinyLM` で見られたように、b_{noise} は初期化時点ではきわめて小さく、訓練中に 1–2 桁増大することがあるためである（これが GPT-3 などの大規模な事前学習でバッチサイズのウォームアップランプが用いられる理由である (Brown et al. 2020)）。
3. **b を $b_{\text{crit}} \approx b_{\text{noise}}$ に照らして設定する：**計算量やデータがボトルネックである場合は $b \lesssim 0.3\, b_{\text{crit}}$ を選ぶ。並列アクセラレータを用いた実経過時間がボトルネックである場合は $b \approx b_{\text{crit}}$（このとき $S = 2S_{\min}$ かつ $E = 2E_{\min}$）を選び、$b \gg b_{\text{crit}}$ は避ける。
4. **η を Equation 9.52 に従ってスケーリングする：**SGD・モメンタムでは $\eta \propto b$、AdamW では $\eta \propto \sqrt{b}$ に従って b_{crit} までスケーリングし、線形ウォームアップと組み合わせる（9.7.3 節）。

9.8.4 まとめ

勾配ノイズスケール $b_{\text{noise}} = \operatorname{tr}(\Sigma)/\|\nabla f\|^2$ は、サンプリングのノイズパワーと信号パワーが等しくなるバッチサイズを表し、データセット全体の走査を行わずに 2 つのバッチサイズ $b_s < b_l$ における二乗勾配ノルムからオンラインで推定できる（Equation 9.47）。適切に調整された学習率のもとでは、目標損失への到達に必要なステップ数 $S(b)$ とデータ例数 $E(b) = b\,S(b)$ は、双曲線型のトレードオフ $S(b)/S_{\min} = 1 + b_{\text{crit}}/b$ および $E(b)/E_{\min} = 1 + b/b_{\text{crit}}$ に従う。この関係は屈曲点 $b_{\text{crit}} \approx b_{\text{noise}}$ を境として、データ効率の高い小バッチ領域（$b \ll b_{\text{crit}}$）とステップ数が飽和する大バッチ領域（$b \gg b_{\text{crit}}$）とを分ける。b_{crit} 未満では、最適学習率は SGD とモメンタムでは線形（$\eta \propto b$）に、Adam と AdamW では平方根（$\eta \propto \sqrt{b}$）に従ってスケーリングし、b が b_{crit} を超えると頭打ちになる。

9.8.5 演習問題

1. **学習率を固定したままのバッチサイズ拡大**。本節の Equation 9.49 と Equation 9.50 を用いて、η を $\eta_0 = \eta_{\text{opt}}(b_0)$ に固定したままバッチサイズを $b > b_0$ に増やすと、1 ステップあたりの保証減少量は $2 - (1 + b_{\text{noise}}/b)/(1 + b_{\text{noise}}/b_0) < 2$ 倍に改善されるものの、$b \to \infty$ としても決して 2 倍を超えないことを示せ。

2. **時間最適バッチサイズと計算量最適バッチサイズ**。バッチサイズ b における 1 ステップの実経過時間が $t(b) = t_0(1 + b/b_{\text{sat}})$ であるとき、Equation 9.51 を用いて最適なバッチサイズ $b^\star$（総実経過時間 $T(b) = t(b)S(b)$ を最小化する値）を導出し、そのデータオーバーヘッド $E(b^\star)/E_{\min}$ を求めよ。

3. [code] **訓練に伴う信号とノイズの時間変化**。関数 `noise_scale` を変更して $\operatorname{tr}\Sigma$、$\|\nabla f\|^2$、およびそれらの比 b_{noise} を個別に返すようにし、`TinyLM` 上でステップ $0, 25, 50, 100, 200, 500, 1000, 2000, 3000$ においてこれら 3 つすべてを追跡せよ。そして、$100\times$ におよぶ b_{noise} の増大が、主に信号 $\|\nabla f\|^2$ の低下（$\operatorname{tr}\Sigma$ の増加ではなく）によって生じていることを確認せよ。

4. [code] **臨界バッチサイズを超えたスケーリング**。本節の `TinyLM` に対する目標到達ステップ数のスイープ（9.8.3 節）を $b = 1{,}024$ まで拡張し、平方根則 $\eta \propto \sqrt{b}$ に従う場合と η を $b = 256$ での値に固定する場合とを比較せよ。さらに、η を b_{crit} を超えてスケーリングし続けると収束が悪化する理由を説明せよ。

9.9 スケールアップ

ニューラルネットワークは小規模な設定で開発され、大規模な設定で運用される。数十億パラメータ規模の事前学習（pretraining）では学習率をグリッド探索（grid search）する余裕がないため、実務では同じアーキテクチャ族に属する小規模な**プロキシモデル**でハイパーパラメータを調整し、それを目的のスケールへと転移する。

深層学習フレームワークでデフォルトとして用いられる**標準パラメータ化**（standard parametrization、SP）では、各層の重みを分散 $1/\text{fan-in}$ で初期化し、すべての層で単一の大域的な学習率 η を共有するが、この設定では最適な学習率が**モデル幅をまたいで転移しない**。層の幅 n が増すにつれて、Adam の 1 ステップは隠れ層の活性化前の値を $O(\eta n)$ 変化させる。なぜなら、n 個の項からなる行列ベクトル積 $\Delta \mathbf{W}\,\mathbf{h}$ において、各項の符号が入力の活性化値 $\mathbf{h}$ と揃うからである。その結果、モデルが広くなるにつれて最適学習率は下方へドリフトする。

最大更新パラメータ化（maximal update parametrization、μP、Yang and Hu (2021); Yang et al. (2022)）は、各層のファンインとファンアウトの増加に応じて学習率と出力乗数をスケーリングすることでこのドリフトを解消し、すべての隠れ層が $\Theta(1)$ の特徴更新を受けることを幅 $n \to \infty$ において保証する。本節では、$64\times$ の範囲にわたる MLP の幅（$n = 128$〜$8{,}192$）で SP における学習率のドリフトを測定し、第 1 ステップのスケーリング則を導出する。続いて μP を実装して 1 ステップの**座標チェック**（coordinate check）により検証し (Yang et al. 2022; Dey et al. 2024)、幅をまたいだゼロショット学習率転移を実証したうえで、この規則を Transformer、スペクトルノルムオプティマイザ、および深さへと拡張する。

```
import random
```

9.9.1 標準パラメータ化における学習率と幅

9.9.1.1 幅の系列と学習率のドリフト

Fashion-MNIST 上の隠れ層 3 層からなる ReLU MLP を考え、その隠れ層の幅 n を $\{128, 512, 2048, 8192\}$（パラメータ数は 1.35×10^5〜1.41×10^8 個）に変化させる。各重み行列が n に伴ってどのようにスケーリングするかに注目しよう。

- **入力層（`fc_in`、$784 \to n$）**：ファンインは固定（784）であり、ファンアウトが n に伴って増加する。
- **隠れ層（`fc_h1`、`fc_h2`、$n \to n$）**：ファンインとファンアウトの両方が n に伴って増加する。
- **出力読み出し（`fc_out`、$n \to 10$）**：ファンインが n に伴って増加し、ファンアウトは固定（10）である。

標準パラメータ化（SP）では、すべての重み行列を $\mathcal{N}(0, 1/\text{fan-in})$ から初期化し、バイアスを 0 に初期化して、全パラメータで単一の Adam 学習率 η を共有する。まず 60,000 枚の Fashion-MNIST 訓練画像を平坦化して正規化し、アクセラレータ上に配置する。

```
fashion = d2l.FashionMNIST()
device = d2l.try_gpu()
X = (fashion.train.data.float().reshape(-1, 784) / 255).to(device)
Y = fashion.train.targets.to(device)
```

次に `MLP(width)` を定義する。各層 $784 \to n \to n \to n \to 10$ は標準偏差 $\text{fan-in}^{-1/2}$ で初期化し、全パラメータで単一の Adam 学習率を共有する。

```
class MLP(nn.Module):
    """A three-hidden-layer ReLU network under standard parametrization."""
    def __init__(self, width):
        super().__init__()
        self.fc_in = nn.Linear(784, width)
        self.fc_h1 = nn.Linear(width, width)
        self.fc_h2 = nn.Linear(width, width)
        self.fc_out = nn.Linear(width, 10)
        for lin in (self.fc_in, self.fc_h1, self.fc_h2, self.fc_out):
            nn.init.normal_(lin.weight, std=lin.in_features ** -0.5)
            nn.init.zeros_(lin.bias)

    def features(self, X):
        h = F.relu(self.fc_h1(F.relu(self.fc_in(X))))
        return F.relu(self.fc_h2(h))

    def forward(self, X):
        return self.fc_out(self.features(X))

    def configure_adam(self, lr):
        return torch.optim.Adam(self.parameters(), lr)
```

最適学習率が幅 n に応じてどのように変化するかを測定するため、`train_mlp` は `arch(width)` を Adam で 400 ステップ（$b = 512$）訓練し、最後の 50 ステップにわたる平均損失を返す（発散した実行については上限を 2.5 とする）。

```
def train_step(model, optimizer, Xb, Yb):
    loss = F.cross_entropy(model(Xb), Yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.detach()

def train_mlp(arch, width, lr, seed, num_steps=400, batch_size=512):
    torch.manual_seed(seed)
    model = arch(width).to(device)
    optimizer = model.configure_adam(lr)
    idx = torch.as_tensor(np.random.default_rng(seed).integers(
        0, X.shape[0], (num_steps, batch_size)), device=device)
    losses = torch.stack([train_step(model, optimizer, X[i], Y[i])
                          for i in idx])
    v = float(losses[-50:].mean())  # Mean minibatch loss of the last 50 steps
    return min(v, 2.5) if math.isfinite(v) else 2.5
```

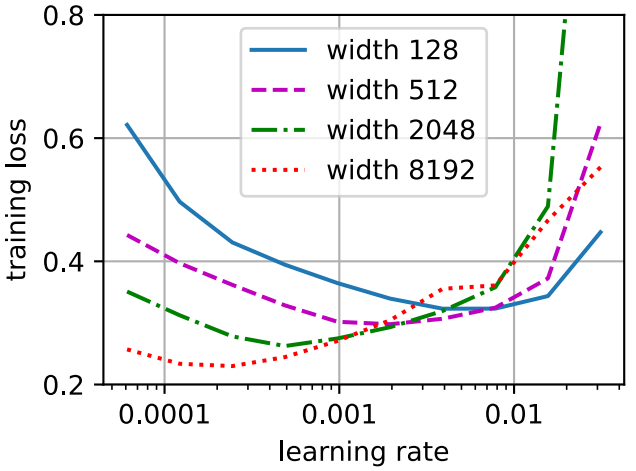
補助関数 sweep、print_optimum、plot_sweep は、幅・学習率・シードのグリッドにわたって train_mlp を評価し、最適学習率 $\eta^\star(n)$ の n に対する両対数傾きを当てはめ、シード平均した損失曲線をプロットする。

```
def sweep(arch, widths, lrs, seeds):
    """Score of every width and learning rate, one column per seed."""
    return {w: np.array([[train_mlp(arch, w, lr, s) for s in seeds]
                         for lr in lrs]) for w in widths}

def print_optimum(loss, lrs):
    """Best rate of the seed-averaged loss per width; slope in log-log."""
    widths = list(loss)
    best = [lrs[loss[w].mean(1).argmin()] for w in widths]
    slope = np.polyfit(np.log(widths), np.log(best), 1)[0]
    print('best lr:', '  '.join(f'{lr:.1e}' for lr in best),
          f'   slope {slope:+.2f}')

def plot_sweep(loss, lrs):
    d2l.plot(lrs, [l.mean(1) for l in loss.values()], 'learning rate',
             'training loss', xscale='log', ylim=[0.2, 0.8],
             legend=[f'width {w}' for w in loss])
```

それでは、sweep を MLP に対して実行し、幅 $n \in \{128, 512, 2048, 8192\}$ および対数等間隔の 10 個の学習率 $\eta \in \{2^{-14}, \ldots, 2^{-5}\}$ を 2 つの共有シードにわたって走査しよう。

```
widths = [128, 512, 2048, 8192]
lrs = [2.0 ** k for k in range(-14, -4)]
seeds = [random.getrandbits(32) for _ in range(2)]  # Shared by every sweep
sp_loss = sweep(MLP, widths, lrs, seeds)
print_optimum(sp_loss, lrs)
plot_sweep(sp_loss, lrs)
```

```
best lr: 3.9e-03  2.0e-03  4.9e-04  2.4e-04    slope -0.70
```



標準パラメータ化では、幅が大きくなるにつれて U 字型の損失曲線が一貫して左へ移動し、最適学習率は 16〜32× 低下する（$\sim 3.9 \times 10^{-3}$（幅 128）から ~ 1〜2×10^{-4}（幅 8,192）への変化であり、両対数傾きは -0.7〜-0.8 となる）。幅 128 での最適学習率を幅 8,192 のモデルにそのまま適用すると、曲線の不安定な右側へと大きく押し上げられ、1,000× 大きいパラメータ数の恩恵が帳消しになってしまう。

9.9.1.2 Adam の第 1 ステップにおける最適学習率のドリフト

なぜ $\eta^\star$ が幅 n に伴って低下しなければならないのかを見るために、正方の隠れ重み行列 $\mathbf{W} \in \mathbb{R}^{n \times n}$ を考え、これが活性化値ベクトル $\mathbf{h} \in \mathbb{R}^n$ を $\mathbf{z} = \mathbf{W}\mathbf{h}$ に写すとする。初期化時には $W_{ij} \sim \mathcal{N}(0, 1/n)$ が $\mathbf{h}$ と独立であるため、各座標 z_i は分散 $\Theta(1)$ をもち、これは n に依存しない。

逆伝播された出力勾配 $\delta = \partial \ell / \partial \mathbf{z}$ をもつ単一の例に対して、重み勾配は外積（outer product）$\mathbf{G} = \delta \mathbf{h}^\top$ となる。ステップ 1 において、Adam のバイアス補正された更新は $\Delta \mathbf{W} = -\eta \ \mathrm{sign}(\mathbf{G}) = -\eta \ \mathrm{sign}(\delta)\, \mathrm{sign}(\mathbf{h})^\top$ である。同じ活性化値ベクトル $\mathbf{h}$ に対して評価すると、活性化前の値 z_i に生じる変化は次のようになる。

$$
\begin{aligned}
(\Delta \mathbf{W}\, \mathbf{h})_i &= -\eta \sum_{j=1}^{n} \mathrm{sign}(\delta_i h_j)\, h_j = -\eta \ \mathrm{sign}(\delta_i) \sum_{j=1}^{n} |h_j| \\
&= -\eta \ \mathrm{sign}(\delta_i)\, n\, \overline{|h|},
\end{aligned}
\tag{9.53}
$$

ここで、$\overline{|h|} = \frac{1}{n} \sum_{j=1}^{n} |h_j| = \Theta(1)$ である。ΔW_{ij} は h_j と相関しているため、和に含まれる n 個の項はすべて同じ符号を共有してコヒーレントに足し合わされ（$O(n)$）、ランダムな符号をもつ初期化時の項のように打ち消し合うことはない（$O(\sqrt{n})$）。特徴更新 $(\Delta \mathbf{W}\, \mathbf{h})_i = \Theta(1)$ を幅 n に対して不変に保つには、ファンイン n をもつ隠れ行列の学習率を $\eta \propto 1/n$ にスケーリングしなければならない（両対数傾きは -1 となる。MLP 全体での実験的な傾きが -0.7〜-0.8 であったのは、$n \times n$ の隠れ行列と、ファンインが固定された入力層やバイアスとの折衷を反映している）。

9.9.2 最大更新パラメータ化（μP）

9.9.2.1 μP のスケーリング則

ファンインのスケーリングが異なる層に単一の大域的学習率を強制するのではなく、**最大更新パラメータ化**（μP、Yang and Hu (2021); Yang et al. (2022)）は参照用の**基準幅**（base width）n_{base} に対する**幅乗数**（width multiplier）$m = n/n_{\text{base}}$ を用いて各層をスケーリングする（Table 9.4）。

1. **入力重みおよび 1D バイアス・ノルム（ファンイン固定）**：初期分散 $1/\text{fan-in}$ と学習率 η をそのまま維持する。
2. **隠れ行列（ファンイン・ファンアウトともに $\propto n$）**：初期分散 $1/\text{fan-in} = 1/n$ を維持し、Adam の学習率を η/m にスケーリングすることで、あらゆる幅において $(\eta/m)\, n\, \overline{|h|} = \eta\, n_{\text{base}}\, \overline{|h|} = \Theta(1)$ となるようにする。
3. **出力読み出し行列（ファンイン $\propto n$、ファンアウト固定）**：$\Delta\mathbf{W}_{\text{out}}\,\mathbf{h}$ と $\mathbf{W}_{\text{out}}\,\Delta\mathbf{h}$ はいずれも相関した n 個の項を出力ロジットへと足し合わせるため、$\mathbf{W}_{\text{out}}$ の各成分の大きさは $O(1/n)$ でなければならない（$O(1/\sqrt{n})$ ではない）。スケーリング前の重みの初期化分散を $1/n_{\text{base}}$ に固定し、順伝播で出力ロジットに $1/m$ を掛け、Adam の学習率を η に保つことで、実効的な出力重みとその更新の双方が $O(1/n)$ でスケーリングされ、あらゆる幅で $\Theta(1)$ のロジット更新が得られる。

Table 9.4: Adam に対する最大更新パラメータ化(μP)の規則(基準幅 n_{base}、幅乗数 $m = n/n_{\text{base}}$)。

パラメータグループ	初期分散	Adam 学習率	順伝播乗数
入力重みと全バイアス	$1/\text{fan-in}$（バイアスは 0）	η	1
隠れ行列（$n \times n$）	$1/\text{fan-in} = 1/n$	η/m	1
出力読み出し（$n \times d_{\text{out}}$）	$1/n_{\text{base}}$	η	$1/m$

```python
class MuMLP(MLP):
    """The same network under muP, relative to a width-8,192 base."""
    def __init__(self, width, base_width=8192):
        super().__init__(width)
        self.m = width / base_width
        # Rule 3: output init variance 1/base_width at every width
        with torch.no_grad():
            self.fc_out.weight.mul_(self.m ** 0.5)

    def forward(self, X):
        # Rule 3: scale the output matrix's logits; the bias is untouched
        return self.fc_out(self.features(X) / self.m)

    def configure_adam(self, lr):
        hidden = [self.fc_h1.weight, self.fc_h2.weight]
        rest = [p for p in self.parameters()
                if not any(p is q for q in hidden)]
        return torch.optim.Adam([
            {'params': rest, 'lr': lr},
            {'params': hidden, 'lr': lr / self.m}])  # Rule 2: hidden LR / m
```

（**`base_width=8192` に設定すると、最大の幅 $n = 8{,}192$ において `MuMLP` が `MLP` と一致するため、上記の SP のスイープと直接比較できる。実運用では、`base_width` を小規模なプロキシの幅に設定し、$m > 1$ によって目的のモデルへとスケールアップする。**）

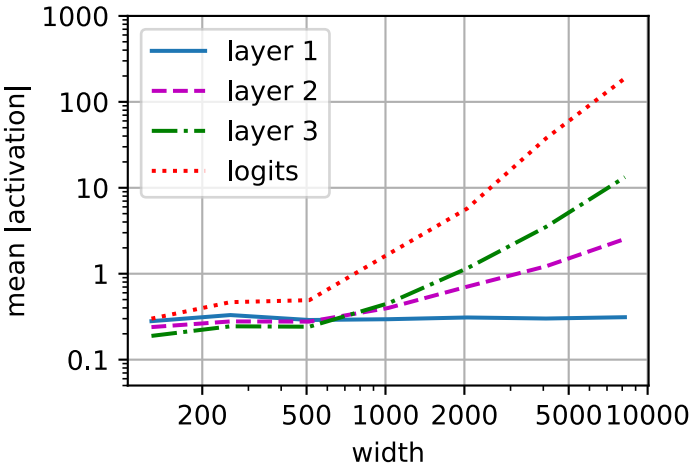
9.9.2.2 座標チェックと学習率転移

スイープを開始する前に μP の実装を検証する手法が**座標チェック**である (Yang et al. 2022; Dey et al. 2024)。ここでは幅 $n \in \{128, 256, \dots, 8192\}$ にわたって Adam を 1 ステップ実行し、各層の活性化値の平均絶対値 $\overline{|h_\ell|}$ を `activations` と `coord_check` で測定する。

```
def activations(model, Xb):
    with torch.inference_mode():
        h1 = F.relu(model.fc_in(Xb))
        h2 = F.relu(model.fc_h1(h1))
        h3 = F.relu(model.fc_h2(h2))
        return [float(h.abs().mean()) for h in (h1, h2, h3, model(Xb))]

def coord_check(arch, widths, lr, seed, num_steps=1):
    Xb, Yb = X[:256], Y[:256]
    sizes = []
    for width in widths:
        torch.manual_seed(seed)
        model = arch(width).to(device)
        optimizer = model.configure_adam(lr)
        for _ in range(num_steps):
            train_step(model, optimizer, Xb, Yb)
        sizes.append(activations(model, Xb))
    return list(zip(*sizes))
```

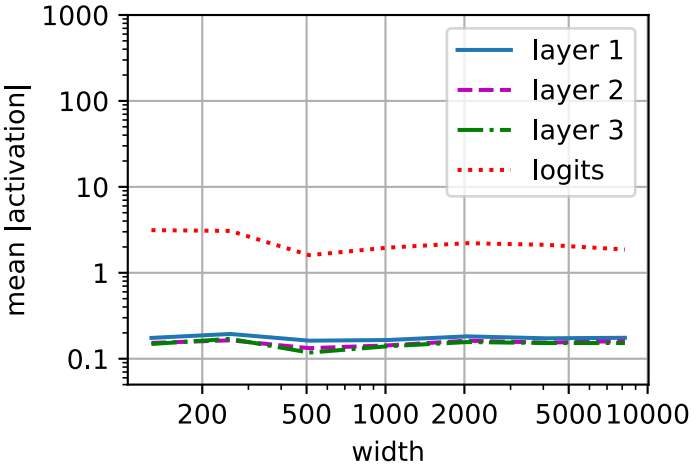
`coord_check` を標準パラメータ化の `MLP` に対して、7 つの幅 $n \in \{128, 256, \dots, 8192\}$ にわたり幅 128 での最適学習率 `lr_128` で実行すると、幅の拡大に伴う深刻な活性化値の発散が明らかになる。

```
check_widths = [128 * 2 ** k for k in range(7)]  # 128 to 8,192
lr_128 = lrs[sp_loss[128].mean(1).argmin()]  # SP's best rate at width 128
sp_acts = coord_check(MLP, check_widths, lr_128, seeds[0])
d2l.plot(check_widths, list(sp_acts), 'width', 'mean |activation|',
         xscale='log', yscale='log', ylim=[0.05, 1000],
         legend=['layer 1', 'layer 2', 'layer 3', 'logits'])
```



まったく同じ 1 ステップの `coord_check` を `MuMLP` に対して基準幅の学習率 `lr_base` で実行すると、μP によって 3 つの隠れ層すべてと出力ロジットが幅に対して不変に保たれることが確認できる。

```
lr_base = lrs[sp_loss[8192].mean(1).argmin()]  # Best at the base width
mup_acts = coord_check(MuMLP, check_widths, lr_base, seeds[0])
d2l.plot(check_widths, list(mup_acts), 'width', 'mean |activation|',
         xscale='log', yscale='log', ylim=[0.05, 1000],
         legend=['layer 1', 'layer 2', 'layer 3', 'logits'])
```

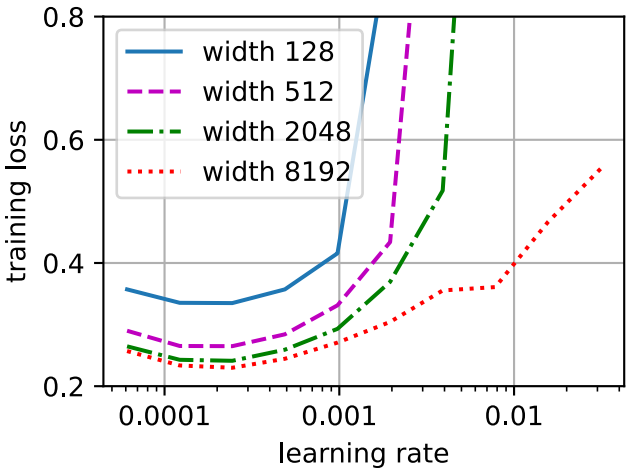


SP（`MLP`）では、最大の幅付近において第 2 層、第 3 層、および出力ロジットがそれぞれ $O(n)$、$O(n^2)$、$O(n^3)$ の増加に近づく（1 ステップ後にはロジットが約 1,000× 上昇しており、これは幅 128 から幅 8,192 への変化に相当する）。一方、μP（`MuMLP`）では、すべての層の活性化値とロジットのスケールが 7 つの幅すべてにわたって平坦なままである。

400 ステップの学習率スイープを μP のもとで繰り返すと、64× の幅の範囲全体にわたって最適学習率が揃うことが確認できる。

```
mup_loss = sweep(MuMLP, widths[:-1], lrs, seeds)
mup_loss[widths[-1]] = sp_loss[widths[-1]]  # At the base width muP is SP
print_optimum(mup_loss, lrs)
plot_sweep(mup_loss, lrs)
```

```
best lr: 2.4e-04  2.4e-04  2.4e-04  2.4e-04    slope +0.00
```



最後に、幅 8,192 で達成される訓練損失について、狭い幅 128 のプロキシで調整した学習率を転移した場合と、幅 8,192 で直接調整した場合とを、SP と μP の双方で比較する。

```
for name, loss in (('SP', sp_loss), ('muP', mup_loss)):
    mean = {w: l.mean(1) for w, l in loss.items()}
    print(f'{name:3s} best loss per width:',
          '  '.join(f'{m.min():.3f}' for m in mean.values()))
    small, wide = widths[0], widths[-1]
    tuned, own = mean[small].argmin(), mean[wide].argmin()
    for label, i in ((f'the width-{small} lr', tuned), ('its own lr', own)):
        runs = '  '.join(f'{v:.3f}' for v in loss[wide][i])
        print(f'    width {wide} at {label:16s} {mean[wide][i]:.3f}'
              f'  (seeds {runs})')
```

```
SP  best loss per width: 0.323  0.298  0.262  0.230
    width 8192 at the width-128 lr 0.355  (seeds 0.382  0.329)
    width 8192 at its own lr       0.230  (seeds 0.232  0.228)
muP best loss per width: 0.335  0.265  0.241  0.230
    width 8192 at the width-128 lr 0.230  (seeds 0.232  0.228)
    width 8192 at its own lr       0.230  (seeds 0.232  0.228)
```

μP では、$\eta^\star(n)$ の両対数傾きは SP における -0.7 以下の急な傾きからほぼ 0 へと縮小する（最適値の移動は $64\times$ の幅の範囲全体で高々 1 グリッドステップに収まる）。幅 128 で調整した学習率を幅 8,192 へそのまま転移した際の準最適性は ≤ 0.003 nat（シードによるノイズを十分に下回る）にとどまり、SP における 0.10–0.15 nat の損失悪化とは対照的である。

9.9.3 Transformer とスペクトルオプティマイザへの拡張

9.9.3.1 Transformer に対する μP の規則

μP を `TinyLM`（9.3.1 節）のようなモデル次元 d、幅乗数 $m = d/d_{\text{base}}$ の Transformer に適用する場合も、ファンイン・ファンアウトの分類はまったく同じであり、アテンションロジットに関する規則が 1 つ加わるだけである（Table 9.5）。訓練中に学習されたクエリとキーのベクトルが整列するため、両者のドット積 $\mathbf{q}^\top \mathbf{k}$ はヘッド次元（head dimension）$d_k \propto d$ にわたって $O(d_k)$ で増大する。したがって、アテンションには $1/d_k$ によるスケーリングが必要となる（追加係数 $1/\sqrt{m}$ を標準的な $1/\sqrt{d_k}$ スケーリングに掛けることに相当する (Yang et al. 2022)）。

Table 9.5: Adam における Transformer の最大更新パラメータ化（μP）の規則（$m = d/d_{\text{base}}$）。

構成要素	スケーリングの理由	μP の規則
埋め込み、1D バイアス、LayerNorm のスケール・シフト	ファンイン固定（参照または要素ごと）	学習率 η
ブロック内部の行列（`qkv`、`proj`、`mlp1`、`mlp2`）	ファンイン・ファンアウトともに $\propto d$	学習率 η/m、初期分散 $1/\text{fan-in}$
出力読み出し `head`	ファンイン $\propto d$、語彙サイズ V は固定	初期分散 $1/d_{\text{base}}$、ロジット $\times 1/m$、学習率 η
アテンションのドット積スコア	整列した d_k 次元ベクトルの内積	スコアを $1/d_k$ でスケーリング（追加係数 $1/\sqrt{m}$）
残差ブランチ（深さ L も増加する場合）	L 個のブランチ出力が残差ストリームに加算	各ブランチに L_{base}/L を乗算（9.9.3.3 節）

9.9.3.2 スペクトルスケーリング条件と Muon

Yang et al. (2023) が示すように、μP のすべての規則は単一の作用素ノルム（operator norm）の原理のもとに統一できる。すなわち、$\mathbf{h} \in \mathbb{R}^{d_{\text{in}}}$（ただし $\|\mathbf{h}\|_2 = \Theta(\sqrt{d_{\text{in}}})$）を $\mathbf{z} \in \mathbb{R}^{d_{\text{out}}}$（ただし $\|\mathbf{z}\|_2 = \Theta(\sqrt{d_{\text{out}}})$）に写す層において、成分ごとの活性化値と更新を $\Theta(1)$ に保つには、重み行列 $\mathbf{W}$ とその更新 $\Delta\mathbf{W}$ の双方が次のスペクトルノルム（spectral norm）をもつ必要がある。

$$\|\Delta\mathbf{W}\|_2 = \Theta\left(\sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}}\right). \tag{9.54}$$

正方の $n \times n$ 隠れ行列に対して、Equation 9.54 は $\|\Delta\mathbf{W}\|_2 = \Theta(1)$ を要求する。Adam では、ランク 1 の符号更新が $\|\Delta\mathbf{W}\|_2 = \Theta(\eta n)$ となるため、$\eta \propto 1/n$ が必要になる。Muon（9.6 節）では、直交化された更新 $\text{NS}_5(\cdot)$ は構成上、単位スペクトルノルムをもつ。これを $\sqrt{d_{\text{out}}/d_{\text{in}}}$ でスケーリングすれば (Bernstein 2025)、学習率は $1/m$ の補正なしで幅をまたいでそのまま転移する（一方、AdamW に合わせた係数 $0.2\sqrt{\max(d_{\text{out}}, d_{\text{in}})}$（9.6 節）はスペクトルノルムが $\Theta(\eta\sqrt{n})$ となるため、$\eta \propto 1/\sqrt{n}$ のスケーリングとなる。演習問題 3 を参照）。

9.9.3.3 深さ・重み減衰・有限サイズ

ここまでは幅のみを変化させてきた。しかし実際のスケールアップでは層も増やすことになり、さらに AdamW のもとでは重み減衰（weight decay）も学習率への変更に追随させなければならない。いずれにも固有の規則が必要である。

まず深さを考えよう。L 個の残差ブロック（residual block）からなるネットワークでは、残差ストリーム（residual stream）は L 個のブランチ出力の和となる。各ブランチの寄与が $\Theta(1)$

であれば、その和は L に伴って増大し、すべてのブロックが出力を動かすためオプティマイザの 1 ステップの影響も同様に増大する。対策はロジット乗数と同じ形をとり、各残差ブランチに $(L_{\text{base}}/L)^{\alpha}$ を掛ける。これには 2 つの指数が提案されている。$\alpha = 1/2$ とすると、ブランチ出力は独立なランダム項として足し合わされることになり、これは初期化時の状態に対応する (Bordelon et al. 2024; G. Yang et al. 2024)。一方、$\alpha = 1$ とすると整列した項として足し合わされることになり、これは訓練によって各項が相関した後の状態に対応する。これは読み出し層に $1/n$（$1/\sqrt{n}$ ではなく）を与えたのと同じ議論である。Dey et al. (2025) は、Transformer において $\alpha = 1$ のみが L の増加に伴ってもすべてのブロックに特徴を学習させ続け、その結果、学習率が幅の場合と同様に深さをまたいでも転移することを示している。

次に重み減衰を考える。AdamW では、減衰の時間スケール $\tau = 1/(\eta\lambda)$ が、重みが平衡ノルムへ近づく速さを決める（9.5 節）。ある層の学習率を m で割りながら λ をそのままにする規則では、その層の τ が m 倍に伸びてしまい、広いモデルには調整元の小規模モデルとは異なる正則化がかかることになる。一貫した選択肢の 1 つは τ を一定に保つことであり、具体的には λ に m を掛ける変更を、η を m で割るすべての箇所に適用する (Wang and Aitchison 2025)。

最後に、これらの規則はいずれも無限大サイズの極限でのみ厳密に成り立つ。実用上の幅では最適値は依然としてわずかにドリフトし、本節のスイープでも 1 グリッドステップ分移動した。したがって、転移した学習率はターゲットサイズでのスイープの代わりとするのではなく、狭い範囲のスイープの中心として用いるのが最善である。

9.9.4 まとめ

標準パラメータ化（SP）では、Adam の 1 ステップがファンイン n の隠れ層の出力を $\Theta(\eta n)$ 変化させるため、ネットワークが広くなるにつれて最適学習率 $\eta^{\star}(n)$ は下方へドリフトする。**最大更新パラメータ化（μP）**は、隠れ行列の学習率を $1/m$（ただし $m = n/n_{\text{base}}$）でスケーリングし、出力層の初期化分散を $1/n_{\text{base}}$ に固定しつつロジットを $1/m$ でスケーリングし、さらに Transformer のアテンションスコアを $1/d_k$ でスケーリングすることで、$\Theta(1)$ の特徴更新と幅をまたいだゼロショット学習率転移を実現する。1 ステップの**座標チェック**は、等比数列をなす複数の幅にわたって、オプティマイザの 1 ステップ後も各層の活性化値の大きさが平坦に保たれることを検証する。ハイパーパラメータ転移を本格的なスケールアップへと拡張する際には、さらに深さに応じて残差ブランチを L_{base}/L でスケーリングし、層ごとの学習率の変化に合わせて AdamW の減衰の時間スケール $\tau = 1/(\eta\lambda)$ を一定に保つことで、ターゲットスケールにおける狭いスイープの中心となる漸近的な基準点が得られる。

9.9.5 演習問題

1. [code] **座標チェックによる誤った μP 規則の診断**。`base_width=128` を `MuMLP` に設定し、`coord_check` を `lr_128` で用いて、次の 3 つの意図的なバグを 1 つずつテストせよ。(a) ロジット乗数 $1/m$ の省略、(b) 隠れ層学習率のスケーリング η/m の省略、(c) 隠れ層学習率の η/m^2 によるスケーリング。どの誤りを `coord_check` が即座に検出できるか、また更新を過剰に縮小する誤り（η/m^2）の検出になぜ複数ステップの損失スイープが必要となるのかを明らかにせよ。

2. [code] **参照用の基準幅の影響**。μP のスイープを `base_width=128` として幅 $\{128, 512, 2048, 8192\}$ にわたり再実行し、$\eta^{\star}(n)$ の両対数傾きがほぼ 0 のままであることを確認せよ。また、どの定数比（入力層と隠れ層の学習率比、あるいは初期ロジットスケール）が幅 8,192 における損失の差を生み出しているかを特定せよ。

3. **層およびオプティマイザにわたるスペクトルノルム条件**。●Equation 9.54 を用いて、Table 9.4 の μP 規則が入力層・隠れ層・出力層にスペクトルノルム $\Theta(\sqrt{d_{\text{out}}/d_{\text{in}}})$ の更新

を与えること（ランク 1 の Adam ステップのもとで）を確認し、さらに Muon（9.6 節）において TinyLM の 4 つのブロック行列に必要となる η の幅スケーリングを導出せよ。

4. [code] **TinyLM における μP と座標チェックの実装**。●Table 9.5 を `d2l.TinyLM` に対して `d_model` $\in \{64, 128, 256, 512, 1024\}$（`num_heads=2`、`base_width=64`）にわたり適用し、$\eta = 3 \times 10^{-3}$ での 1 ステップ座標チェックによって、すべてのサブ層出力およびロジットの大きさが幅に対して平坦に保たれることを確認せよ。

9.10 実践

これまでの小規模な実験では、1 回の実行あたりの訓練時間はわずか数秒であったため、学習率をグリッド状に探索したり、更新が発散するたびに実行をやり直したりできた。しかし、本番の大規模な訓練は事情が異なる。1 回の訓練に数日から数週間を要する場合、大半の設定を事前に固定しなければならず、無防備なまま 1 度の勾配スパイクが生じるだけで何時間もの進捗が損なわれかねない。さらに、最終チェックポイントの品質が最後のミニバッチ抽出の運に左右されてはならない。

極端な大規模環境（たとえば、パラメータ数 540B の PaLM モデルを 6,144 個の TPU チップで訓練した際には、勾配クリッピングを有効にしていたにもかかわらず約 20 回の突発的な損失スパイクが発生した (Chowdhery et al. 2022)）であっても、信頼性の高い訓練は 3 つの運用上の柱に支えられている。

1. **一過性の不安定性に対するガード：**大域ノルムによる**勾配クリッピング**（gradient clipping、8.4.2 節）は、SGD における異常なステップに上限を設けるとともに、AdamW の二次モーメント蓄積変数 $\mathbf{v}_t$ を一過性の勾配スパイクから保護し、チェックポイント保存と巻き戻しによる回復と組み合わせて用いられる。
2. **反復点の平滑化：**モデルの重みの**指数移動平均**（exponential moving average、EMA）を保持することで、学習率をゼロまで完全に減衰させなくとも、ノイズ球内の高周波な変動を除去できる。
3. **短い代理実行による体系的な較正：**デバイスのスループットと b_{noise}（9.8 節）からバッチサイズを選択し、実績のあるオプティマイザのデフォルト設定を採用したうえで、短い実行（ホライズンの 10%）でピーク学習率を較正してから、ウォームアップと減衰の全スケジュール（9.7 節）を実行する。

本節では、これら 3 つの柱をそれぞれ詳しく検討し、未調整のパラメータ数 3.3M、4 ブロックの TinyLM に対する完全なエンドツーエンド設定へと組み立てる。

```
import random
```

9.10.1 勾配クリッピングとスパイクからの回復

8.4.2 節でリカレントネットワーク向けに導入したように、大域ノルムによる**勾配クリッピング**は全パラメータにわたって連結した勾配を単一のベクトル $\mathbf{g}$ として扱い、そのユークリッドノルム $\|\mathbf{g}\|_2$ がしきい値 θ を超えるたびに再スケーリングする。

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|_2}\right)\mathbf{g}. \tag{9.55}$$

スケーリング係数 $\theta/\|\mathbf{g}\|_2$ は大域的であるため、クリッピングはすべての層にわたる相対的な方向を保ったまま、オプティマイザに入力される勾配のユークリッドノルムを θ 以下に抑える。

```
class Clipped:
    """Clip the global gradient norm before every optimizer step."""
    def __init__(self, optimizer, params, max_norm=1.0):
        self.optimizer, self.params = optimizer, list(params)
        self.max_norm, self.norms = max_norm, []

    def step(self):
        self.norms.append(nn.utils.clip_grad_norm_(self.params, self.max_norm))
        self.optimizer.step()

    def zero_grad(self):
        self.optimizer.zero_grad()
```

9.10.1.1 ガードとしてのクリッピングと常時ブレーキ

9.5 節では、モメンタム 0.9 の SGD を `TinyLM` に適用したところ、`inf` への発散が $\eta = 1.0$ では最初の 20 ステップ以内に生じた。$\eta = 1.0$ は損失ランドスケープ全体に対して大きすぎるのだろうか、それとも初期化直後の重みが大きな勾配を生む冒頭の一過性のステップでのみ大きすぎるのだろうか。まず、`WikiText103` ストリーム（$b = 64$、ウィンドウ長 64）を読み込む。

```
data = d2l.WikiText103(batch_size=64, num_steps=64)
```

次に、`run(seed, lr, max_norm)` は `TinyLM` を $2{,}000$ ステップにわたり、モメンタム 0.9 かつ `max_norm` でクリッピングした SGD で訓練し、各ステップの損失とクリッピング前の勾配ノルムの両方を返す。

```
new_seed = lambda: random.getrandbits(31)

def run(seed, lr, max_norm, num_steps=2000):
    """SGD with momentum 0.9 on TinyLM, clipped at max_norm (math.inf: not
    at all); return the losses and the gradient norms before clipping."""
    torch.manual_seed(seed)  # The initialization and the minibatch order
    model = d2l.TinyLM(len(data.vocab))
    optimizer = Clipped(torch.optim.SGD(model.parameters(), lr, momentum=0.9),
                        model.parameters(), max_norm)
    losses = d2l.train_steps(model, data, optimizer, num_steps)
    return losses, torch.stack(optimizer.norms).cpu().numpy()
```

ここで、$(\eta, \theta) \in \{(1.0, \infty), (1.0, 1.0), (0.3, \infty), (3.0, 1.0)\}$ を 2 つのシードにわたって検証し、クリッピングが作動したステップ数（$\|\mathbf{g}_t\|_2 > \theta$）、最後にクリッピングが作動したステップ、および勾配ノルムの中央値を記録する。

```
settings = [(1.0, math.inf), (1.0, 1.0), (0.3, math.inf), (3.0, 1.0)]
seeds, runs = [new_seed() for _ in range(2)], {}
print(f'{"lr":>4}{"clip at":>9}{"final loss":>12}{"steps clipped":>15}'
      f'{"last":>6}{"median norm":>13}{"largest / median":>18}')
for seed in seeds:
    for lr, max_norm in settings:
        losses, norms = runs[seed, lr, max_norm] = run(seed, lr, max_norm)
```

```
            line = f'{lr:4g}{max_norm:9g}{d2l.final_loss(losses):12.3f}'
            if max_norm < math.inf:
                fired = np.nonzero(norms > max_norm)[0]  # Steps clipping changed
                median = np.median(norms)
                line += (f'{len(fired):15d}{fired.max(initial=0):6d}{median:13.2f}'
                         f'{norms[100:].max() / median:18.1f}')  # After step 100
            print(line)
```

```
 lr  clip at  final loss  steps clipped  last  median norm  largest / median
  1      inf         inf
  1        1       1.626             17    28         0.25               1.8
0.3      inf       1.671
  3        1         inf           1997  1996          nan               nan
  1      inf         inf
  1        1       1.619             17    16         0.26               1.8
0.3      inf       1.652
  3        1         inf           1636  1635          nan               nan
```

$\eta = 1.0$ において $\theta = 1.0$ でクリッピングすると、最初の数十ステップ（すべてステップ 50 より前）でのみ作動し、残りの 1,950 ステップ（勾配ノルムの中央値は ~ 0.25）では二度と作動せず、`inf` への発散を防いで損失を ~ 1.61–1.64 nat まで引き下げる。ここでクリッピングは**ガード**として機能している。すなわち、$< 2\%$ のステップにのみ介入して初期の一過性の発散を防いでいる（同時に裾の重い勾配ノイズにも対処する (Zhang, Karimireddy, et al. 2020)）。対照的に、$\eta = 3.0$ では勾配ノルムが有限である限り毎ステップでクリッピングが作動し、それでも訓練は失敗する（損失が 100–200 nat で停滞するか、`inf` にオーバーフローする）。クリッピングが**ブレーキ**として常時作動している場合、単に学習率 η が大きすぎるのである (Godbole et al. 2023)。実務上は、定常状態における勾配ノルムの中央値より θ をおよそ 3–5× 高い値（典型的には $\theta = 1.0$）に設定し、クリッピングの作動率を監視するとよい。

9.10.1.2 AdamW におけるクリッピングと運用上のスパイク回復

AdamW（9.5 節）のもとでは、単一の異常な勾配 $g \gg \sqrt{v_{t-1}}$ が生じても、その直後のステップはそれ自体で高々 $\eta(1-\beta_1)/\sqrt{1-\beta_2}$ 以下に抑えられる（具体的には $\approx 3.2\,\eta$ が $\beta_2 = 0.999$ の値、$0.45\,\eta$ が $\beta_2 = 0.95$ の値である）。直後のステップを抑える代わりに、**AdamW におけるクリッピングは二次モーメント状態 $\mathbf{v}_t$ を保護する**。クリッピングなしの 100× の勾配スパイクは、通常の二乗勾配の $(1-\beta_2)\times 10^4$ 倍を v_t に注入して分母 $\sqrt{\hat{v}_t}$ を肥大化させ、続く $\sim 1/(1-\beta_2)$ ステップ（1,000 ステップが $\beta_2 = 0.999$ の場合、20 ステップが $\beta_2 = 0.95$ の場合）にわたって学習を麻痺させるからである。

勾配クリッピング（および PaLM の z 損失 $10^{-4}\log^2(\sum_j e^{o_j})$ のような補助的なロジット正則化項。演習問題 2 を参照）にもかかわらず突発的な損失スパイクが生じた場合、標準的な運用上の対処法 (Chowdhery et al. 2022; Zhang et al. 2022) は、**スパイクの ~ 100 ステップ前に保存したチェックポイントまで巻き戻し、発生箇所の前後 200–500 個のミニバッチをスキップすること**である。これには、モデルの重み、オプティマイザのモーメント $(\mathbf{m}_t, \mathbf{v}_t)$、スケジューラのステップカウンタ、およびデータストリームのカーソルを一定間隔でチェックポイント保存しておく必要がある。

9.10.2 重み平均化（EMA）

定数または非ゼロの学習率では、確率的な反復点はノイズ球（9.2 節）の内部で最適解の周囲を絶えずゆらぎ続け、曲率 λ_{curv} のモードは $\sim 1/(\eta\lambda_{\text{curv}})$ ステップの自己相関時間を持つ。そこで、次

のように**指数移動平均（EMA）** $\bar{\mathbf{x}}_t$ をライブパラメータ $\mathbf{x}_t$ に対して減衰係数 $\alpha \in [0.99, 0.9999]$ で保持する (Polyak and Juditsky 1992)。

$$\bar{\mathbf{x}}_t = \alpha\, \bar{\mathbf{x}}_{t-1} + (1-\alpha)\, \mathbf{x}_t, \tag{9.56}$$

この移動平均は直近 $1/(1-\alpha)$ ステップのウィンドウにわたって平均をとることで、ライブの訓練軌跡を変えることなく評価時にノイズ球の変動を打ち消す。

定数学習率の SGD（$\eta = 0.3$）における重み EMA あり・なし（$\alpha = 0.999$、ウィンドウ幅 1,000 ステップ ≈ 4.2 エポック、初期の遅れを避けるためエポック 5 の終了後に開始）と、15 エポック（3,525 ステップ）にわたるコサイン学習率減衰とを `FashionCNN` 上で比較する。まず `FashionMNIST`（$b = 256$）を読み込み、検証正解率の補助関数 `test_acc` を定義する。

```
fashion = d2l.FashionMNIST(batch_size=256)
num_epochs, num_batches = 15, len(fashion.train_dataloader())
test_acc = lambda model: d2l.evaluate_accuracy(model, fashion.val_dataloader())
```

次に、スケジュールの補助関数 `warmup_cosine` と重み EMA を定義する。

PyTorch の `torch.optim.swa_utils.AveragedModel` が EMA を保持するため、ここで定義する必要があるのはスケジュールのみである。

```
def warmup_cosine(peak, warmup, num_steps):
    """A linear warmup to peak over warmup steps, then a cosine to zero."""
    def schedule(t):
        if t < warmup:
            return peak * t / warmup
        return peak * (1 + math.cos(math.pi * (t - warmup)
                                    / (num_steps - warmup))) / 2
    return schedule
```

訓練ループ `train_fashion` は、`schedule(t)` のもとで通常の SGD を `num_epochs=15` エポック実行し、各エポック終了時にライブ重みとその EMA（エポック `start=5` から開始）の両方についてホールドアウトのテスト正解率を記録する。

```
def train_fashion(seed, schedule, alpha=0.999, start=5):
    """Plain SGD on FashionCNN at the rate schedule(t); the test accuracy
    after every epoch of the weights and of their EMA from epoch start on."""
    torch.manual_seed(seed)  # The initialization and the minibatch order
    model = d2l.FashionCNN().to(d2l.try_gpu())
    optimizer = torch.optim.SGD(model.parameters(), lr=1.0)  # Times schedule
    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, schedule)
    optimizer.register_step_post_hook(lambda *_: scheduler.step())
    ema = torch.optim.swa_utils.AveragedModel(
        model, multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(alpha))
    live, avg = [], []
    for epoch in range(num_epochs):
        if epoch == start:  # Its first update copies the current weights
            optimizer.register_step_post_hook(
                lambda *_: ema.update_parameters(model))
        d2l.train_steps(model, fashion, optimizer, num_batches)
```

```
            live.append(test_acc(model))
            avg.append(test_acc(ema) if epoch >= start else math.nan)
        return live, avg
```

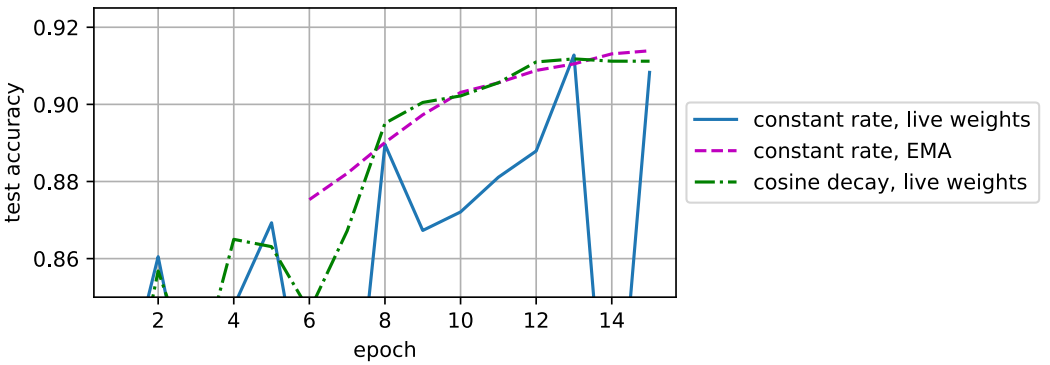
2 つのシードにわたり train_fashion を定数学習率（$\eta = 0.3$）とコサイン減衰（$\eta : 0.3 \to 0$）の両方で実行し、最後の 5 エポックにおけるライブ重みの正解率の範囲を最終的な EMA の正解率と比較する。

```
schedules = {'constant rate': lambda t: 0.3,
             'cosine decay': warmup_cosine(0.3, 0, num_epochs * num_batches)}
cnn_seeds, curves = [new_seed() for _ in range(2)], {}
print(f'{"test accuracy":<15}{"live, last five epochs":>24}{"live, final":>13}'
      f'{"EMA, final":>12}')
for seed in cnn_seeds:
    for name, schedule in schedules.items():
        live, avg = curves[seed, name] = train_fashion(seed, schedule)
        print(f'{name:<15}{min(live[-5:]):15.3f} to {max(live[-5:]):.3f}'
              f'{live[-1]:13.3f}{avg[-1]:12.3f}')
```

```
test accuracy    live, last five epochs  live, final  EMA, final
constant rate             0.792 to 0.913        0.908       0.914
cosine decay              0.906 to 0.912        0.911       0.909
constant rate             0.870 to 0.902        0.902       0.910
cosine decay              0.899 to 0.908        0.908       0.906
```

1 つ目のシードについてエポックごとのテスト正解率の軌跡をプロットすると、定数学習率の EMA がライブ重みのノイズ球によるジッタをいかに除去するかが視覚的にわかる。

```
(live, avg), (decayed, _) = (curves[cnn_seeds[0], name] for name in schedules)
d2l.plot(list(range(1, num_epochs + 1)), [live, avg, decayed], 'epoch',
         'test accuracy', ylim=[0.85, 0.925], figsize=(5, 2.5),
         legend=['constant rate, live weights', 'constant rate, EMA',
                 'cosine decay, live weights'], legend_loc='outside')
```



定数学習率では、最後の 5 エポックにわたってライブ重みのテスト正解率が数パーセントポイントの幅で小刻みに揺れ動き、1 エポックの調子が悪いときにはさらに大きく落ち込むこともある。これに対し、その EMA は帯の上端に沿って滑らかに上昇して約 91% に達し、学習率を一度も下げることなくコサイン減衰させた実行の最終正解率に匹敵する。したがって、WSD スケジュール

の定数プラトー期間中において、重み EMA はクールダウン分岐（9.7.4.1 節）の低コストな近似となる。追加のステップを一切必要としないからである。ただし、完全な代替にはならない。言語モデルでは、プラトーに沿った平均化によってクールダウンによる改善の一部は回収できるものの、すべては回収できない (Hägele et al. 2024)。

9.10.3 エンドツーエンド設定の実例

ここで本章全体を総合し、3,000 ステップの訓練を、未調整の 4 ブロック、幅 256、4 ヘッドの `TinyLM`（パラメータ数 3.3M）に対して構成する。Table 9.6 に 4 つの最先端の事前学習で公開されているオプティマイザ設定をまとめ、Table 9.7 に訓練ログの症状と対処法を対応付けた診断早見表を示す。

Table 9.6: 4 つの大規模言語モデルの事前学習における公開オプティマイザ設定。

モデル	オプティマイザ	β_1, β_2	ピーク学習率・スケジュール	ウォームアップ	クリップ θ	重み減衰 λ	バッチサイズ
Llama 3 405B (Grattafiori et al. 2024)	AdamW	—	8×10^{-5}、8×10^{-7} へコサイン減衰	8k ステップ	—	—	4M → 16M トークンへ段階的増加
DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024)	AdamW	0.9, 0.95	2.2×10^{-4}、2.2×10^{-5} へ WSD コサイン減衰	2k ステップ	1.0	0.1	3,072 → 15,360 系列へ段階的増加
OLMo 2 7B (Team OLMo et al. 2025)	AdamW	0.9, 0.95	3×10^{-4}、3×10^{-5} へコサイン減衰	2k ステップ	1.0	0.1（埋め込み除外）	1,024 系列
Kimi K2 (Kimi Team 2025a)	MuonClip	—	2×10^{-4}、2×10^{-5} へ WSD コサイン減衰	500 ステップ	—	0.1	67M トークン

Table 9.7: 記録された指標（損失、クリッピング前の勾配ノルム、クリッピング率、相対ステップサイズ $\|\Delta\mathbf{W}\|_{\text{RMS}}/\|\mathbf{W}\|_{\text{RMS}}$）からの最適化失敗の診断。

ログ上の症状	診断	主な対処法
勾配ノルムの爆発（勾配爆発）を伴い最初の数ステップで損失が発散	ステップサイズが初期の局所的な鋭さ $2/L$ を超過	線形ウォームアップまたは大域ノルムクリッピングを追加
Adam で損失が高止まりし、初期の相対重み更新が $> 10\%$	コールドスタートのステップが初期化を破壊	線形ウォームアップ（全ステップの 1%–5%）を追加
ウォームアップを入れても Adam で損失が高止まり	ピーク学習率 η_0 が過大	η_0 を 2–3× 引き下げ
訓練を通じて $> 20\%$–50% のステップでクリッピングが作動	クリッピングが常時ブレーキとして作用	まず η_0 を引き下げ（θ の調整より優先）
ステップ間の大きな分散を伴い損失が早期にプラトーへ到達	現在の η/b における定常的なノイズ球	$\eta_t \to 0$ へ減衰、重み EMA で評価、または b を拡大

ログ上の症状	診断	主な対処法
最終ステップまで一定の傾きで損失が滑らかに減少	累積移動量 B_T の予算不足	ピーク η_0 または総ステップ数 T を増加

Table 9.6 に倣い、パラメータ数 3.3M の `TinyLM` を以下のとおり設定する。

1. **バッチサイズ：**$b = 256$ 系列とする（ハードウェアのスループットが飽和し、かつウォームアップ後の勾配ノイズスケール $b_{\text{noise}} \sim 100$–$200$ と同程度になる領域である）。
2. **オプティマイザとガード：**$(\beta_1, \beta_2) = (0.9, 0.95)$ の AdamW、2D の隠れ層行列のみに適用する分離型重み減衰（decoupled weight decay）$\lambda = 0.1$（9.5.3.3 節）、$\theta = 1.0$ での大域ノルム勾配クリッピング、および短いウィンドウの重み EMA（$\alpha = 0.995$）を用いる。
3. **ピーク学習率の較正：**3 回の 300 ステップ（ホライズンの 10%）の試行を $\eta \in \{10^{-3}, 3 \times 10^{-3}, 10^{-2}\}$ にわたって行い、$\eta_0 = 3 \times 10^{-3}$ が最適であることを特定する（クリッピングされるステップは $< 5\%$ であり、$\theta = 1.0$ が純粋にガードとして機能していることが確認できる）。
4. **スケジュール：**60 ステップ（全体の 2%、$T = 3{,}000$）の線形ウォームアップに続けてゼロまでのコサイン減衰を行い、検証損失を定数学習率のベースラインより 0.05–0.06 nat 低減する（シード間のばらつきは < 0.01 nat）。

まず `tiny_lm(seed)` を定義して 4 ブロック、幅 256、4 ヘッドの `TinyLM` を生成し、ノイズスケール推定用にランダムな訓練ウィンドウを抽出する `sample_lm(b)` を定義する。

```
def tiny_lm(seed):
    """A TinyLM of four blocks of width 256 on the GPU."""
    torch.manual_seed(seed)  # The initialization and the minibatch order
    return d2l.TinyLM(len(data.vocab), d_model=256, num_heads=4,
                      num_blks=4).to(d2l.try_gpu())

def sample_lm(b):  # b windows at random offsets in the training text
    starts = torch.randint(len(data.train) - data.num_steps, (b,))
    X, Y = data.windows(data.train, starts.numpy())
    return X.to(d2l.try_gpu()), Y.to(d2l.try_gpu())
```

訓練ハーネス `train(seed, schedule, num_steps)` は、$(\beta_1, \beta_2) = (0.9, 0.95)$ かつ重み減衰 0.1（埋め込み以外の 2D 行列に限定）の AdamW を設定し、大域ノルムクリッピング（`max_norm=1.0`）と重み EMA（`alpha=0.995`）でラップしたうえで、損失、ライブモデル、EMA モデル、およびクリッピング率を返す。

```
def train(seed, schedule, num_steps, max_norm=1.0, alpha=0.995):
    """AdamW with betas (0.9, 0.95) and weight decay 0.1 on the matrices at
    the rate schedule(t), clipped at max_norm, with an EMA of the weights;
    return the losses, the model, the EMA and the fraction of clipped steps."""
    model = tiny_lm(seed)
    matrices = [p for name, p in model.named_parameters()
                if p.ndim == 2 and 'emb' not in name]
    others = [p for name, p in model.named_parameters()
              if p.ndim != 2 or 'emb' in name]
    adamw = torch.optim.AdamW([{'params': matrices, 'weight_decay': 0.1},
                               {'params': others, 'weight_decay': 0.0}],
                              lr=1.0, betas=(0.9, 0.95))  # Times schedule
```

```
    scheduler = torch.optim.lr_scheduler.LambdaLR(adamw, schedule)
    ema = torch.optim.swa_utils.AveragedModel(
        model, multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(alpha))
    adamw.register_step_post_hook(lambda *_: scheduler.step())
    adamw.register_step_post_hook(lambda *_: ema.update_parameters(model))
    optimizer = Clipped(adamw, model.parameters(), max_norm)
    losses = d2l.train_steps(model, data, optimizer, num_steps)
    clip_rate = (torch.stack(optimizer.norms) > max_norm).float().mean()
    return losses, model, ema, float(clip_rate)
```

$b = 256$ において、初期化時点での `d2l.noise_scale` を計測し、3 回の 300 ステップ（ホライズンの 10%）の代理試行を $\eta \in \{10^{-3}, 3 \times 10^{-3}, 10^{-2}\}$ にわたって実行してピーク学習率 `peak` を選択し、さらに `noise_scale` を 300 ステップ後に再推定する。

```
data.batch_size = 256  # Where the throughput levels off
seed, short = new_seed(), {}
print('noise scale at initialization: '
      f'{d2l.noise_scale(tiny_lm(seed), sample_lm, b_small=8, b_big=256):.0f}')
for lr in [1e-3, 3e-3, 1e-2]:
    losses, model, _, clip_rate = train(seed, lambda t: lr, num_steps=300)
    short[lr] = (d2l.final_loss(losses), model)
    print(f'lr {lr:g}: loss {short[lr][0]:.3f} after 300 steps, '
          f'{clip_rate:.0%} of the steps clipped')
peak = min(short, key=lambda lr: short[lr][0])
print(f'peak rate {peak:g}; noise scale after its 300 steps: '
      f'{d2l.noise_scale(short[peak][1], sample_lm, b_small=8, b_big=256):.0f}')
```

```
noise scale at initialization: 2
lr 0.001: loss 1.881 after 300 steps, 3% of the steps clipped
lr 0.003: loss 1.740 after 300 steps, 3% of the steps clipped
lr 0.01: loss 2.498 after 300 steps, 6% of the steps clipped
peak rate 0.003; noise scale after its 300 steps: 153
```

最後に、2 つのシードにわたり 3,000 ステップの本番実行を行い、構築したスケジュール（`warmup_cosine(peak, 60, 3000)`）とその重み EMA を、`peak` での定数学習率ベースラインと比較する。

```
num_steps = 3000
val_loss = lambda model: math.log(d2l.lm_perplexity(model, data.val_dataloader()))
print(f'{"validation loss":<16}{"configured":>11}{"its EMA":>9}'
      f'{"clipped":>9}{"baseline":>10}{"gap":>7}')
results = []
for seed in [new_seed() for _ in range(2)]:
    _, model, ema, clip_rate = train(
        seed, warmup_cosine(peak, num_steps // 50, num_steps), num_steps)
    _, baseline, _, _ = train(seed, lambda t: peak, num_steps)
    results.append((val_loss(model), val_loss(baseline)))
    print(f'{"seed " + str(len(results)):<16}{results[-1][0]:11.3f}'
          f'{val_loss(ema):9.3f}{clip_rate:9.1%}{results[-1][1]:10.3f}'
          f'{results[-1][1] - results[-1][0]:7.3f}')
(c1, b1), (c2, b2) = results
```

```
    print(f'spread between the seeds: configured {abs(c1 - c2):.3f}, '
          f'baseline {abs(b1 - b2):.3f}')
```

```
validation loss  configured  its EMA  clipped  baseline    gap
seed 1                1.245    1.245     0.5%     1.303  0.058
seed 2                1.248    1.249     0.5%     1.300  0.052
spread between the seeds: configured 0.004, baseline 0.003
```

9.10.4 まとめ

大域ノルム勾配クリッピング（$\mathbf{g} \leftarrow \min(1, \theta/\|\mathbf{g}\|_2)\mathbf{g}$）は、初期ステップや異常ステップのごく一部（$< 5\%$）でのみ作動するときには**ガード**として機能し、SGD における初期の発散を防ぎ、AdamW の二次モーメントバッファ $\mathbf{v}_t$ を勾配スパイクから保護する。一方で、絶えず作動し続ける場合は η が大きすぎてクリッピングが**ブレーキ**と化していることを示す。モデルの重みの**指数移動平均（EMA）**（$\bar{\mathbf{x}}_t = \alpha \bar{\mathbf{x}}_{t-1} + (1-\alpha)\mathbf{x}_t$）は、ライブの軌跡を変更することなく定常的なノイズ球の変動を平滑化し、定数学習率の実行において完全なコサイン減衰と同等の正解率を達成する。信頼性の高い大規模訓練では、ハードウェアを飽和させつつ b_{noise} 付近に設定したバッチサイズ、$(\beta_1, \beta_2) = (0.9, 0.95)$ かつ 2D 重み行列に限定した分離型重み減衰 $\lambda = 0.1$ を持つ AdamW、$\theta = 1.0$ での大域ノルムクリッピング、短ホライズンでのピーク学習率の較正、そして線形ウォームアップに続くコサイン減衰または WSD クールダウンを組み合わせる。

9.10.5 演習問題

1. [code] **クリッピングの領域：ガードとブレーキ**。$\theta \in \{0.01, 0.1, 0.5, 1.0, 4.0, \infty\}$ を $\eta = 1.0$ において `run` を用いて `TinyLM` 上で変化させて最終損失とクリッピング率を記録し、クリッピングを 20 ステップまたは 100 ステップの線形ウォームアップに置き換えた場合と比較せよ。

2. **ソフトマックスのロジットドリフトと PaLM の z 損失**。出力ロジット $\mathbf{o} \in \mathbb{R}^V$ と分配関数 $Z = \sum_{j=1}^{V} e^{o_j}$ に対し、交差エントロピー勾配の総和がゼロ（$\mathbf{1}^\top \nabla_{\mathbf{o}} \ell_{\text{CE}} = 0$）である一方、補助ペナルティ $10^{-4} \log^2 Z$ (Chowdhery et al. 2022) を加えると復元勾配 $2 \times 10^{-4} (\log Z)\, \mathbf{p}$ が生じ、$\log Z$ を 0 の方向へ引き戻しつつ $\mathbf{p} = \text{softmax}(\mathbf{o})$ の一様シフト不変性を損なわないことを示せ。

3. [code] **重み EMA のウィンドウ幅と開始時期**。EMA の減衰率 $\alpha \in \{0.9, 0.99, 0.999, 0.9999\}$ と開始エポック `start` $\in \{0, 5, 10\}$ を `train_fashion` 上で定数学習率 $\eta = 0.3$ のもとで変化させて実験し、正解率の軌跡をエポック単位の実効ウィンドウ幅 $1/(1-\alpha)$ と関連付けて考察せよ。

4. [code] **設定実例のアブレーション**。●4 ブロック、幅 256 の `TinyLM`（`train` で定義）において、2 シードのアブレーションを実行し、以下を検証せよ。(a) WSD の終盤線形減衰とコサイン減衰の比較、(b) ウォームアップの除去または勾配クリッピングの除去、(c) ピーク学習率の 10^{-3} または 10^{-2} への変更、(d) 定数学習率ベースラインに対する $\alpha \in \{0.995, 0.999\}$ での重み EMA の評価。

9.11 関連文献

本節では、本章の手法を発展させた基礎的な教科書、サーベイ論文、および近年の研究をまとめる。本文中で引用した収束率（convergence rate）の数学的証明は第 35 章で与えられており（理

論に関する追加の文献は 35.9 節を参照)、自動ハイパーパラメータ最適化は 第 41 章（英語版）で、複数デバイスによる分散訓練システムは第 13 章で扱っている。

9.11.1 基礎的な教科書、サーベイ、プレイブック

- **凸最適化と数値最適化：**@Boyd.Vandenberghe.2004（『Convex Optimization』）は、条件付け（conditioning）、勾配降下法、ニュートン法（9.1 節）、および一般のノルムのもとでの最急降下法（steepest descent、9.6 節）に関する標準的な参考書である。Nocedal and Wright (2006)（『Numerical Optimization』）は、ニュートン法、準ニュートン法、直線探索、信頼領域法を包括的に扱っている。
- **確率的最適化：**@Bottou.Curtis.Nocedal.2018（「Optimization Methods for Large-Scale Machine Learning」）は、確率的勾配のノイズ球、ステップサイズの減衰（9.2 節）、および分散低減（variance reduction）を概説している。
- **対話型ガイドと実務者向けプレイブック：**Gabriel Goh による「Why Momentum Really Works」（Distill、2017 年）は、臨界減衰と $\sqrt{\kappa}$ 加速則（9.4 節）の対話的な可視化を提供している。Jeremy Bernstein による「Deriving Muon」および Bernstein and Newhouse (2024) は、ノルム球にわたる最急降下法を統一的に導出している（9.6 節）。EleutherAI の「Practitioner's Guide to Maximal Update Parameterization」は μP の座標チェック（9.9 節）を詳述しており、Google Research の『Deep Learning Tuning Playbook』(Godbole et al. 2023) は、9.3 節および 9.10 節全体で用いられる科学的（scientific）・局外（nuisance）・固定（fixed）ハイパーパラメータの区別を定式化している。

9.11.2 損失曲面

9.11.2.1 高次元損失ランドスケープの可視化とモード連結性

ニューラルネットワークの損失曲面の低次元断面は、正の斉次性によるスケール不変性（$c\,\mathbf{W}_1$ と $c^{-1}\mathbf{W}_2$ は同じ ReLU ネットワークを計算する）を取り除くために摂動方向をフィルタごとに正規化しなければ、誤解を招くものになりかねない。フィルタ正規化したスライスを用いることで、Li et al. (2018) は残差接続（7.4 節）がぎざぎざした非凸な曲面を滑らかな盆地へと変えることを示した。

独立に訓練された 2 つのネットワーク間を線形補間すると高い損失の障壁を横切ることになるが、Garipov et al. (2018) および Draxler et al. (2018) は**モード連結性**（mode connectivity）を発見した。すなわち、1 回だけ折れ曲がる単純な区分線形曲線によって、訓練損失・テスト損失ともに一様に低い経路に沿って独立な極小点どうしを結ぶことができる。さらに Entezari et al. (2022) と Ainsworth et al. (2023) は、置換対称性（9.1 節）を用いて隠れユニットを揃えると、幅の広いネットワーク間の線形の障壁がほぼ消失することを実証した。

9.11.2.2 平坦な極小点と Sharpness-Aware Minimization (SAM)

より平坦な極小点ほど汎化性能が高いという仮説に基づき、**Sharpness-Aware Minimization (SAM)** (Foret et al. 2021) は、降下方向の勾配を評価する前に勾配上昇方向の摂動ステップを踏むことで、現在の重みの周囲の ℓ_2 球内における最悪ケースの損失を最小化する（1 回の更新につき 2× の勾配コストがかかる）。ただし、そのままのヘッセ行列の鋭さ（sharpness）は層ごとの再パラメータ化に対して不変ではないことに注意が必要である (Dinh, Pascanu, et al. 2017)。

9.11.3 古典的な 2 次手法と分散低減手法

9.11.3.1 準ニュートン法、直線探索、信頼領域法

準ニュートン法（quasi-Newton method）は、ヘッセ行列の逆行列 $\mathbf{H}^{-1}$ を、1 次の勾配のみを用いた連続するセカント対 $(\mathbf{x}_{t+1} - \mathbf{x}_t, \mathbf{g}_{t+1} - \mathbf{g}_t)$ から近似する。記憶制限 BFGS（**L-BFGS**、Liu and Nocedal (1989)）は直近 m 個の対を $O(md)$ のメモリに保持し、決定論的で滑らかな問題に対する定番の手法であるが（35.2.2.2 節）、ミニバッチ勾配の差分に対しては敏感である。**直線探索**（line search、Armijo の十分減少条件の判定により η_t を動的に選ぶ手法）および**信頼領域法**（trust-region method、35.2.3 節、Muon の信頼領域解析については Kovalev (2025) を参照）も同様に、決定論的最適化において威力を発揮する。

9.11.3.2 ヘッセ行列フリー法と鞍点フリーニュートン法

$d \times d$ のヘッセ行列を陽に作らずに 2 次手法を深層ネットワークへ適用するため、**ヘッセ行列フリー最適化**（Hessian-free optimization）(Martens 2010) は、半正定値であるガウス・ニュートン行列上のヘッセ行列ベクトル積（34.5.3.2 節）を用いた共役勾配法によって $\mathbf{Hd} = -\mathbf{g}$ を近似的に解く。一方、**鞍点フリーニュートン法**（Saddle-Free Newton）(Dauphin et al. 2014) は、負の曲率方向が反復点を引き寄せるのではなく反発させるよう、低次元のクリロフ部分空間上で $|\mathbf{H}|^{-1}$ による前処理を施す。

9.11.3.3 確率的収束率と分散低減

一般の凸目的関数に対して、ミニバッチ SGD はミニマックス最適な $O(1/\sqrt{T})$ の収束率を達成し (Nemirovski and Yudin 1983)（35.4.3 節）、非凸で滑らかな目的関数に対しては $\min_{t \le T} \mathbb{E}[\|\nabla f(\mathbf{x}_t)\|^2] = O(1/\sqrt{T})$ を保証する (Ghadimi and Lan 2013)（35.4.3.2 節）。Polyak–Ruppert の反復平均化 (Polyak and Juditsky 1992) は、最適な漸近共分散を達成する（9.10.2 節）。有限のデータセット上では、**SVRG** (Johnson and Zhang 2013)（35.4.4 節）がデータセット全体にわたる周期的な制御変量（control variate）によって定常的なノイズ球を解消するが、深層学習においてはデータ拡張や急速に変化する特徴がその有効性を制限する。

9.11.4 行列オプティマイザと構造的オプティマイザ

- **自然勾配、K-FAC、SOAP：**出力分布の空間でステップ長を測る**自然勾配**（natural gradient）(Amari 1998) は、フィッシャー情報行列の逆行列によって更新を前処理する。**K-FAC** (Martens and Grosse 2015) は、層ごとのフィッシャー行列ブロックを入力の共分散と出力勾配の共分散のクロネッカー積として近似する（35.5.5 節）。一方、**SOAP** (Vyas et al. 2024) は、Shampoo の左右の前処理行列 (Gupta et al. 2018) がなす固有基底の上で Adam を実行する。
- **Muon の理解とスケーリング：**@Wang.Zhang.Li.ea.2025 は、AdamW に対する Muon の高速化の要因が、裾の特徴（tail feature）における連想メモリ射影（アテンションの値・出力行列および MLP 行列）にあることを突き止めた。これは希少トークンにおける SGD に対する Adam の優位性 (Kunstner et al. 2023, 2024) と対をなす結果である。また Shah et al. (2025) は、Muon が AdamW よりも大きなバッチサイズまでデータ効率を維持することを明らかにしている。**Scion** (Pethick et al. 2025) は、AdamW にフォールバックすることなく、作用素ノルムによる最急降下法をすべての層（埋め込みや読み出しヘッドを含む）へと拡張する。
- **効率的・分散的な直交化：**まず **Polar Express** (Amsel et al. 2026) と Cesista et al. (2025) は、ステップごとのニュートン・シュルツ多項式の係数（Figure 9.10）を最適化する。また Kim and Oh (2026) は非厳密な q ステップ直交化のもとでの収束を解析し、**Dion** (Ahn

et al. 2025) はシャーディングされた訓練におけるデバイス間通信を削減するため、低ランクのモメンタム近似を直交化する。

9.11.5 スケジュール、平均化、大規模ベンチマーク

- **ウォームリスタート、スケジュールフリー最適化、モデルスープ：**コサインアニーリングはもともと周期的なウォームリスタートを伴う手法（**SGDR**、Loshchilov and Hutter (2016)）として導入され、サイクル間で予測を平均する**スナップショットアンサンブル**（snapshot ensemble）(Huang, Li, et al. 2017) を可能にした。これに対し、**モデルスープ**（model soup）(Wortsman et al. 2022) は共通の初期化からファインチューニングされた複数のモデルの重みを平均する。**Schedule-Free SGD および AdamW** (Defazio et al. 2024) は、現在の反復点と反復点の移動平均との補間点で勾配を評価することにより、あらかじめホライズン T を定めるスケジュールを不要にする。
- **バッチサイズランプとスケーリング則：**@Malladi.Lyu.Panigrahi.ea.2022 は、バッチサイズの増大に伴う Adam のモーメント $(\beta_1, \beta_2, \epsilon)$ に対する完全な確率微分方程式（SDE）スケーリング則を導出している。H. Zhang et al. (2024) および Merrill et al. (2025) は、臨界バッチサイズがモデルサイズではなく主に処理済みトークン数に応じて増大することを示しており、Dey et al. (2023)、DeepSeek-AI, Bi, et al. (2024)、および Meta AI (2025) は、実用言語モデルにおけるスケール間のハイパーパラメータ転移を報告している。
- **オプティマイザのベンチマークと評価プロトコル：**新しいオプティマイザ（LARS/LAMB (You et al. 2017) や NorMuon (Li et al. 2026) など）による見かけ上の改善は、ベースライン側の局外ハイパーパラメータを同等の労力で調整すると、縮小または消失することが少なくない (Nado et al. 2021; Schmidt et al. 2021; Wen et al. 2025)。**AlgoPerf** (Dahl et al. 2023; Kasimbeg et al. 2025)、言語モデルのベンチマークスイート (Semenov et al. 2025)、および **NanoGPT speedrun** の最適化トラック (Jordan and contributors 2024; Jordan 2026) といった標準化されたベンチマークでは、明示的な調整予算と統計的有意性のしきい値が課されている。最後に、公開されている **OPT-175B 訓練ログブック** (Zhang et al. 2022) には、大規模環境における訓練途中の損失スパイク、チェックポイントの巻き戻し、ハードウェア復旧のありのままの日次記録が残されている。

Chapter 10

アテンション

第 8 章のリカレントモデルは、系列を 1 トークンずつ読み込み、それまでに見たすべてを固定サイズの隠れ状態に圧縮する。長い文書の末尾付近での予測が最初の段落の細部に依存する場合、リカレントネットワークはその細部が必要になると知るはるか前に、隠れ状態へ書き込んでおかなければならない。履歴を単一のベクトルに圧縮するのではなく、ネットワークがすべての位置に表現を保持し、現在の予測に必要なものだけを取り出すことはできないだろうか。

アテンションはまさにこれを実現する。各ルックアップでは、現在の位置が求めるものを表す**クエリ**ベクトルに対し、すべての候補位置にある**キー**ベクトルとの間でスコアを計算する。ソフトマックスがそれらのスコアを和が 1 になる非負の重みへと変換し、層はそれらの位置に格納された**値**（value）ベクトルの重み付き平均を返す。重み付き平均は滑らかであるため、勾配降下法によってクエリ、キー、値を生成する射影をエンドツーエンドで訓練できる。このようなルックアップを複数並列に実行すると、**マルチヘッドアテンション**が得られる。アテンションは、ニューラル機械翻訳でデコーダが原言語のすべての単語を直接参照できるように導入され (Bahdanau et al. 2015)、Transformer における唯一の位置混合演算となった (Vaswani et al. 2017)。

再帰構造を全ペア間のルックアップに置き換えると、新たに 3 つの課題が生じる。

1. **順序への不感性：**重み付き和は入力の置換に対して不変であるため、位置情報を明示的に注入しなければならない。
2. **二次スケーリング：**n 個のトークンの全ペアに対してスコアを計算するには $O(n^2)$ の演算とメモリが必要であり、これが IO を考慮した厳密カーネルと準二次近似の両方の動機となる。
3. **機構的解釈：**アテンション重みが示すのはヘッドが**どこを**読み取るかであり、層をまたいで**どのような**関数を計算するかではない。

最初の 3 つの節では、微分可能なキー・値ルックアップ（10.1 節）、スケール付きドット積スコアリングとブールマスキング（10.2 節）、そしてマルチヘッド自己アテンション（10.3 節）によってこの層を構築する。続く 10.4 節では、絶対、回転（RoPE）、および相対（ALiBi）の位置エンコーディングによって系列の順序を扱う。次の 2 つの節では計算スケールを取り上げる。10.5 節では $n \times n$ のスコア行列を実体化しないタイル化オンラインソフトマックスカーネル（FlashAttention）を導出し、10.6 節ではスライディングウィンドウアテンションと線形アテンションの変種を検討する。最後に、10.7 節では残差ストリーム（residual stream）、QK/OV 分解、および 2 層の誘導ヘッド（induction head）を通じて訓練済みアテンション回路を解析し、10.8 節では各種の拡張と一次文献を概観する。

第 11 章へ直接進む読者は、最初の 4 つの節と 10.7 節冒頭の残差ストリームの定式化だけを押さ

えればよい。計算コストに関する各節は、11.3 節および第 12 章と併せて読むことができる。

10.1 クエリ、キー、値

リカレントニューラルネットワーク（8.4 節）は、後の予測でどの詳細が必要になるかを知る前に、それまでに読み込んだすべての情報を固定サイズの状態ベクトルへと圧縮する。たとえば、隠れ次元が 768 のリカレントモデルが 1,024 トークンを読み込む場合、1,024 個すべての位置の情報を 768 個の数値に要約しなければならない。

データベースは、レコードをそのまま保存し、クエリが届いた時点で初めて何を取り出すかを決定することで、時期尚早な圧縮を回避する。しかし、厳密なデータベース参照（lookup）は不連続である。たとえば、キーと値のペア $\mathcal{D} = \{(2, 10), (5, 20)\}$ が与えられたとき、完全一致クエリ $q = 5$ は 20 を返すが、$q = 4$ はどのキーにも一致しない。勾配降下法で訓練するため、ハード参照を**ソフト**参照に置き換える。すなわち、負の二乗距離 $a(q, k) = -(q - k)^2$ などですべてのキーにスコアを付け、ソフトマックスによってそのスコアを凸重みへと変換する。$q = 4$ のとき、キー 2 と 5 のスコアはそれぞれ -4 と -1 であり、重みは $e^{-4}/(e^{-4} + e^{-1}) \approx 0.05$ と 0.95、重み付き値は $0.05 \cdot 10 + 0.95 \cdot 20 \approx 19.5$ となる。この出力はクエリに対して滑らかに変化し、保存されたレコード数がいくつであっても対応でき、クエリを実行する前に固定サイズのボトルネックを設ける必要もない。

この微分可能なキー・値参照が**アテンション機構**（attention mechanism）である (Bahdanau et al. 2015)。本節ではまず、ベクトルのクエリ・キー・値に対するアテンションプーリングを定式化し、続いてその古典的なノンパラメトリック手法の先駆けである Nadaraya–Watson カーネル回帰を検討する。そこでは、単一のバンド幅（bandwidth）パラメータが参照の鋭さを制御する。

10.1.1 微分可能な参照としてのアテンション

$\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)\}$ を、m 個のキー・値ペアからなるデータベースとし、キーを $\mathbf{k}_i \in \mathbb{R}^d$、値を $\mathbf{v}_i \in \mathbb{R}^{d_v}$ とする。クエリ $\mathbf{q} \in \mathbb{R}^d$ が与えられたとき、**アテンションプーリング**は次の値の重み付き和を返す。

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i, \tag{10.1}$$

ここで、$\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ かつ $\sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i) = 1$ である。したがって出力は値の凸結合（convex combination）となり、完全一致参照（ワンホット重みベクトル）と平均プーリング（一様な重み $1/m$）の間を補間する。

任意の実数値をとる適合度スコア（compatibility score）$a(\mathbf{q}, \mathbf{k}_i)$ を凸重みへと写すには、ソフトマックス関数（3.1 節）を適用する。

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{10.2}$$

$\alpha_i = 1/\big(1 + \sum_{j \neq i} e^{a_j - a_i}\big)$ であるため、重みはスコアの差 $a_j - a_i$ のみに依存し、すべてのキーにわたって定数を加えても不変である。ソフトマックスも線形結合も、$\mathbf{q}$、$\mathbf{k}_i$、$\mathbf{v}_i$ に関して微分可能である。

具体的なベクトルの例として、$a(\mathbf{q},\mathbf{k}) = -\|\mathbf{q}-\mathbf{k}\|^2$ とし、クエリを $\mathbf{q} = (1,1)$、キーを $\mathbf{k}_1 = (1,1)$、$\mathbf{k}_2 = (2,1)$、$\mathbf{k}_3 = (0,0)$、値を $\mathbf{v}_1 = (1,3)$、$\mathbf{v}_2 = (0,0)$、$\mathbf{v}_3 = (4,1)$ とする。スコアは 0、-1、-2 となり、ソフトマックス重み $(1, e^{-1}, e^{-2})/(1+e^{-1}+e^{-2}) \approx (0.67, 0.24, 0.09)$ と次の出力が得られる。

$$\mathbf{o} = 0.67\,(1,3) + 0.24\,(0,0) + 0.09\,(4,1) \approx (1.0, 2.1). \tag{10.3}$$

Figure 10.1 に示されている 2 つの性質に注目しよう。第 1 に、$\mathbf{k}_1$ は $\mathbf{q}$ と完全に一致しているにもかかわらず、重みの 2/3 しか受け取らない。スコアを 10 倍にスケーリングすると（$0, -10, -20$ になり）、重みの 99.995% が $\mathbf{k}_1$ に集中する。このように、スコアの大きさはソフトマックスがどれだけ鋭く 1 つの候補に絞り込むかを支配する。第 2 に、$\mathbf{v}_2 = (0,0)$ は和への寄与がゼロであるにもかかわらず、その 0.24 という重みは他の値を希釈することで出力を原点の方へと引き寄せる。

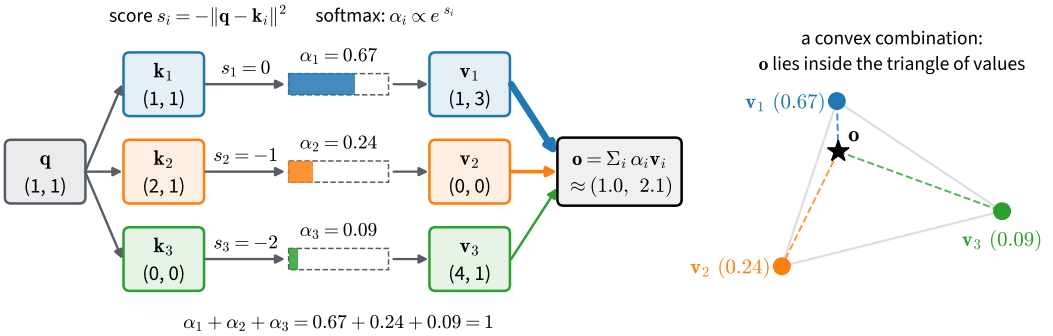


Figure 10.1: クエリ $\mathbf{q} = (1,1)$ の 3 つのキー・値ペア（キーは $(1,1)$、$(2,1)$、$(0,0)$、値は $(1,3)$、$(0,0)$、$(4,1)$）に対するソフト参照。スコア $s_i = -\|\mathbf{q}-\mathbf{k}_i\|^2$ は総和が 1 となるソフトマックス重み α_i に変換され（左）、出力 $\mathbf{o} \approx (1.0, 2.1)$ は値の重み付き和として、各値を頂点とする三角形の内部の点になる（右）。

アテンションの規則 Equation 10.1 は任意のレコード数 m に対してそのまま適用できるため、単一の層で任意の長さの系列を固定サイズの状態に圧縮することなく処理できる。その代わり、コストはメモリへと移る。幅 768 のリカレント層が保持する数値は 768 個であるのに対し、GPT-2 small の 12 ヘッドアテンション層（ヘッドあたり $d = 64$）が 1,024 トークンを処理する場合、ヘッドあたり $2 \times 1{,}024 \times 64 = 131{,}072$ 個のキーと値の要素、すなわち全 12 ヘッドでは約 1.6×10^6 個の数値を保持する。

10.1.2 固定カーネルによるアテンションプーリング

深層学習が登場するずっと前から、**Nadaraya–Watson カーネル回帰**（Nadaraya–Watson kernel regression）(Nadaraya 1964; Watson 1964) はノンパラメトリック回帰に Equation 10.1 を用いていた。そこでは訓練入力をキー、訓練ラベルを値とし、固定の類似度カーネルをスコア付けの規則として扱う。

10.1.2.1 カーネルと Nadaraya–Watson 推定量

非負の類似度カーネル $K(\mathbf{q},\mathbf{k})$、たとえば**バンド幅** $\sigma > 0$ をもつ次のガウスカーネルが与えられたとする。

$$K(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{\|\mathbf{q} - \mathbf{k}\|^2}{2\sigma^2}\right), \tag{10.4}$$

これをキーにわたって正規化すると、**Nadaraya–Watson 推定量**（Nadaraya–Watson estimator）が得られる。

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{K(\mathbf{q}, \mathbf{k}_i)}{\sum_j K(\mathbf{q}, \mathbf{k}_j)}. \tag{10.5}$$

つねに正の値をとる任意のカーネルに対して、Equation 10.5 はスコアを $a(\mathbf{q}, \mathbf{k}) = \log K(\mathbf{q}, \mathbf{k})$ としたソフトマックスアテンション Equation 10.2 に一致する。ガウスカーネルでは $a(\mathbf{q}, \mathbf{k}) = -\|\mathbf{q} - \mathbf{k}\|^2/(2\sigma^2)$ となり、$2\sigma^2 = 1$ のときに冒頭の例が復元される。密度関数と目標関数に関する正則性条件のもとで、$m \to \infty$ かつ $\sigma \to 0$ を適切な速度で近づけると、$f(\mathbf{q})$ は条件付き期待値 $\mathbb{E}[y \mid \mathbf{q}]$ に収束する (Mack and Silverman 1982)。

10.1.2.2 バンド幅

バンド幅 σ が推定量をどのように制御するかを見るため、$n = 40$ 個のノイズを含む訓練観測値を $y = 2\sin(x) + x + \epsilon$（ただし $\epsilon \sim \mathcal{N}(0, 1)$、定義域は $[0, 5)$）から生成し、密なグリッド上で予測を評価する。

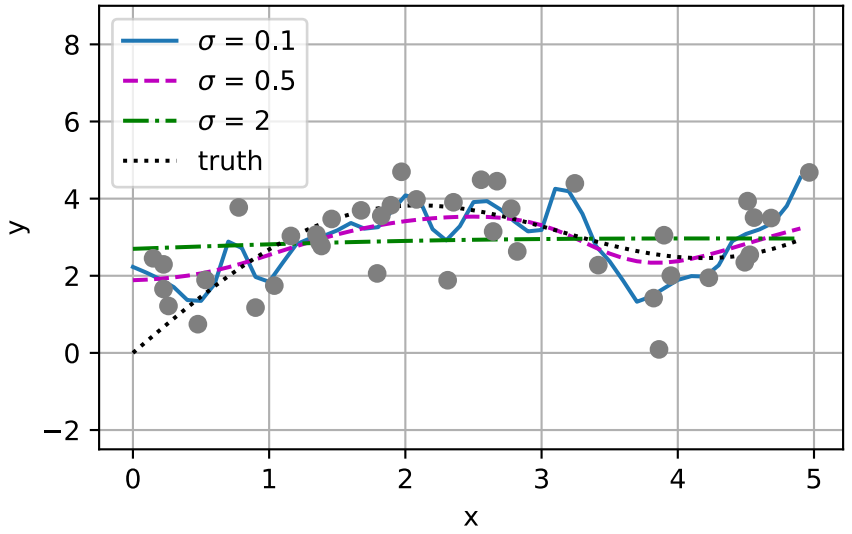
```
n = 40
x_train, _ = torch.sort(torch.rand(n) * 5)
y_train = 2 * torch.sin(x_train) + x_train + torch.randn(n)
x_val = torch.arange(0, 5, 0.1)
y_val = 2 * torch.sin(x_val) + x_val
```

実装では、クエリとキーのすべてのペア間の距離を計算し、ガウスカーネルを評価してクエリごとの各行を正規化したうえで、`y_train` を掛ける。

```
def nadaraya_watson(x_train, y_train, x_val, sigma):
    dists = x_val[:, None] - x_train[None, :]  # (queries, keys)
    k = d2l.exp(-dists**2 / (2 * sigma**2))
    attention_w = k / k.sum(1, keepdims=True)  # Normalize over the keys
    return attention_w @ y_train, attention_w
```

$\sigma \to 0$ の極限では、すべての重みが最も近い訓練点に集中し（1 最近傍補間）、$\sigma \to \infty$ の極限では、すべての重みが $1/n$（全体の標本平均）に近づく。以下に $\sigma \in \{0.1, 0.5, 2.0\}$ の場合をプロットする。

```
sigmas = (0.1, 0.5, 2.0)
estimates = [nadaraya_watson(x_train, y_train, x_val, s)[0] for s in sigmas]
d2l.plot([x_val] * 4 + [x_train], estimates + [y_val, y_train], 'x', 'y',
         legend=[rf'$\sigma$ = {s:g}' for s in sigmas] + ['truth'],
         fmts=('-', 'm--', 'g-.', 'k:', 'C7o'), ylim=(-2.5, 9),
         figsize=(5, 3), legend_loc='upper left')
```



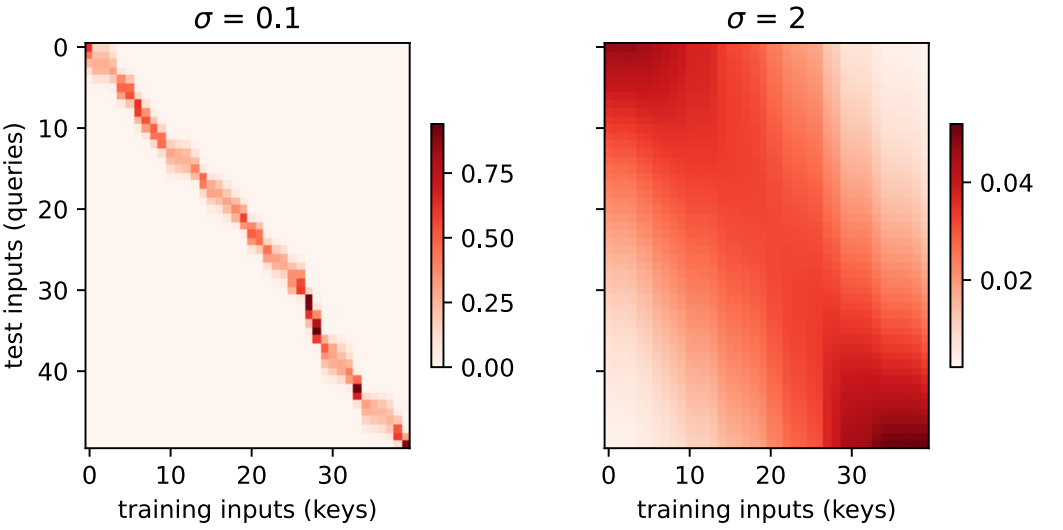
狭いバンド幅（$\sigma = 0.1$）はノイズを含む個々のラベルに過学習する一方、広いバンド幅（$\sigma = 2.0$）は過剰に平滑化して水平線へと近づく。中間の値である $\sigma = 0.5$ は、近傍点がすべて片側に偏る境界付近を除いて真の曲線によく追従する。実用上、σ は 1 つ抜き交差検証（leave-one-out cross-validation）によって選択でき、それには重み行列の対角成分を 0 にして各行を再正規化するだけでよい（演習問題 5）。

より広く見れば、$1/\sigma^2$ はソフトマックスの前に生のスコア $-\frac{1}{2}(q-k)^2$ に掛かる逆温度（inverse temperature）として働く。スコアを拡大するとペア間の差が引き伸ばされ、ソフトマックスはワンホット参照へと近づく。逆にスコアを縮小すると差が圧縮され、ソフトマックスは一様平均へと近づく。これと同じ温度スケーリングは、ドット積アテンション（10.2 節）でも再び登場する。

10.1.3 診断手段としてのアテンション重み

アテンション行列 **A** の各行はキー上の確率分布であるため、**A** をヒートマップとしてプロットすれば、各クエリがどのキーを参照しているかがわかる。

```
w_narrow, w_wide = (nadaraya_watson(x_train, y_train, x_val, s)[1]
                    for s in (0.1, 2.0))
# show_heatmaps takes (rows of panels, columns of panels, queries, keys)
d2l.show_heatmaps(d2l.stack([w_narrow, w_wide])[None],
                  xlabel='training inputs (keys)',
                  ylabel='test inputs (queries)',
                  titles=[r'$\sigma$ = 0.1', r'$\sigma$ = 2'],
                  figsize=(7, 3), cbar='each')
```



$\sigma = 0.1$ では、重みは最も近い訓練入力の周囲の狭い対角帯に集中する。一方、$\sigma = 2.0$ では重みはデータセット全体に広く分散し、どの重みも一様な値 $1/40 = 0.025$ の約 2 倍を超えない。最大となる約 0.05 の重みは端にあるクエリに属するもので、そこでも遠いキーが受け取る重みはわずか約 0.002 にとどまる。

深いネットワークでアテンションのヒートマップを観察する際には、2 つの注意点がある。

1. **重みは層がどこを見ているかを示すのであって、何を伝達するかを示すのではない：**あるキーの値ベクトルがゼロであっても（先のベクトルの例における $\mathbf{v}_2 = (0, 0)$ のように）、あるいは下流の射影の零空間（null space）に含まれていても、そのキーは大きなアテンション重みを受け取り得る。ヘッドの因果的な役割を検証するには、介入実験（interventional experiments）が必要である（10.7 節）。
2. **固定カーネルは静的な類似度計量を強いる：**すべてのクエリは Equation 10.5 において入力空間の同じユークリッド距離を用いる。しかし系列モデリングでは、関連性はコンテキストやタスクに依存する。たとえば、動詞は数トークン離れた構文上の主語を必要とするかもしれない。そのため、学習されるアテンション層では、スコアを計算する前に入力を訓練可能なクエリ・キー・値の空間へと射影する。

10.1.4 まとめ

アテンションプーリングは値ベクトルの凸結合 $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ を計算し、そのソフトマックス重み $\alpha(\mathbf{q}, \mathbf{k}_i)$ は厳密な辞書参照と一様な平均プーリングの間を滑らかに補間する。固定サイズの再帰状態とは異なり、アテンションはペア数 m に依存しない規則によって保存されたすべてのキー・値ペアに直接アクセスするが、作業メモリは m に対して線形に増大する。Nadaraya–Watson カーネル回帰は固定のユークリッド類似度カーネルを用いたアテンションプーリングに対応し、そのバンド幅 σ は統計的バイアスと分散のトレードオフを支配するソフトマックス温度として働く。アテンションのヒートマップはクエリとキーのアライメントを可視化するものの、固定された入力空間の距離を超えるには、訓練可能な射影と学習されるスコア関数が必要である。

10.1.5 演習問題

1. **2 つの値の平均。**キー $\mathbf{k}_1, \ldots, \mathbf{k}_m$ を正規直交とし、スコアを $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}$、重みを Equation 10.2 によるものとする。

1. クエリ $\mathbf{q} = c\,(\mathbf{k}_a + \mathbf{k}_b)$（$c > 0$）に対して m 個すべての重みを計算し、$c \to \infty$ のとき出力が $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$ に収束することを示せ。
2. 選択された 2 つのキーが重み全体の少なくとも 99% を受け取るために必要な c の大きさは、$m = 10$ のときどれだけか。
3. 次に、キーが互いに直交し $\|\mathbf{k}_b\| = 2\,\|\mathbf{k}_a\|$ を満たすとする。再び $\mathbf{v}_a$ と $\mathbf{v}_b$ の平均をとるクエリを求め、さらに $\mathbf{q} = c\,(\mathbf{k}_a + \mathbf{k}_b)$ が $c \to \infty$ のとき代わりに何を返すか述べよ。

Stanford CS224N、課題 4 の問題 1(b) および 1(c) を改題。

2. [code] **有効キー数。** 1 つのクエリに対する重み $\alpha_i(q)$ について、有効キー数を $n_{\text{eff}}(q) = 1/\sum_i \alpha_i(q)^2$ と定義する。

   1. 値を $y_i = g(x_i) + \epsilon_i$（ノイズは互いに独立で分散 s^2）とし、キーを固定する。推定量 Equation 10.5 の q における分散が $s^2/n_{\text{eff}}(q)$ であることを示せ。
   2. n_{eff} を `x_val` 内のすべてのクエリにおいて、`nadaraya_watson` が返す重みから `sigmas` 内の各バンド幅について計算し、3 本の曲線をプロットせよ。次に、`y_train` のノイズを 200 回引き直し（`x_train` は固定）、各クエリにおける推定値の標準偏差を $1/\sqrt{n_{\text{eff}}(q)}$ と比較せよ。

3. [code] **アテンションの勾配。** ●$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i$ であり、かつ $i = 1, \dots, m$ に対して $\mathbf{k}_i = \mathbf{v}_i$ であるとする。ソフトマックス正規化 Equation 10.2 が定めるキー上の分布を $p(\mathbf{k}_i; \mathbf{q})$ と表す。

   1. 次式を証明せよ。
   $$\nabla_{\mathbf{q}} \operatorname{Attention}(\mathbf{q}, \mathcal{D}) = \operatorname{Cov}_{p(\mathbf{k}_i;\mathbf{q})}[\mathbf{k}_i]. \tag{10.6}$$
   2. このヤコビ行列が半正定値であることを結論づけ、どのようなデータベースに対して零行列になるかを求めよ。
   3. $\mathbf{q} = c\,\mathbf{u}$ とおく。ここで固定された方向 $\mathbf{u}$ に対し、ちょうど 1 つのキーが最大のスコア $\mathbf{u}^\top \mathbf{k}_i$ を達成するとする。$c \to \infty$ としたときのヤコビ行列の極限を求め、それが勾配降下法によるクエリの訓練に対して何を意味するか述べよ。
   4. $m = 8$、$d = 4$ のランダムなデータベース上で、自動微分を用いてこの恒等式と極限を検証せよ。

4. [code] **検索としてのソフト参照。** ●m 個のキーを $\mathbb{R}^{16}$ の単位球面上から一様に引いてデータベースを構築し、値 $\mathbf{v}_i$ をインデックス i のワンホット指示ベクトルとする。これらをバンド幅 σ のガウスカーネルで重み付けし、Equation 10.5 と同様に正規化して、出力が重みベクトルになるようにする。保存されたキーにノイズを加えたコピー $\mathbf{q} = \mathbf{k}_i + \epsilon$（ただし $\epsilon \sim \mathcal{N}(\mathbf{0}, \tau^2 \mathbf{I})$）をクエリとする。

   1. すべての $\sigma > 0$ に対して最大の重みはクエリに最も近いキーに割り当てられ、したがってバンド幅はどのインデックスが勝者になるかには影響しないことを示せ。ただし、バンド幅は勝者が重みのどれだけの割合を占めるかには影響する。
   2. ノイズがない場合、正解キーに対する重みは次式となる。
   $$\alpha_i = \frac{1}{1 + \sum_{j \neq i} \exp\big(-\|\mathbf{k}_i - \mathbf{k}_j\|^2 / 2\sigma^2\big)}. \tag{10.7}$$
   ランダムな単位ベクトルに対して $\mathbf{k}_i^\top \mathbf{k}_j \approx 0$、すなわち $\|\mathbf{k}_i - \mathbf{k}_j\|^2 \approx 2$ と近似し、α_i が 0.9 まで下がるバンド幅を $m = 20$ および $m = 200$ について導出せよ。σ は m が大きくなるにつれてどのように縮小しなければならないか。
   3. 次の層が出力を利用できるような参照では、単に最大の割合を得るだけでなく、重みの大半が 1 つのキーに集中する必要がある。$\alpha_i \geq 0.5$ のときクエリの検索に成功し

たとよぶことにする。τ を 0 から 1 まで動かして、$m = 20$ と $m = 200$ のそれぞれで各 m に対する (b) のバンド幅を用い、検索正解率を τ に対してプロットせよ。重みはスコア $-\|\mathbf{q} - \mathbf{k}_j\|^2/2\sigma^2$ のソフトマックス Equation 10.2 として計算せよ。クエリから正解キーまでの距離が変わらないにもかかわらず、m を大きくすると固定した τ での正解率が低下する理由を説明せよ。

5. [extended] **学習される計量。**●学習された距離を用いるカーネル回帰 (Weinberger and Tesauro 2007) は、Equation 10.5 の固定類似度を学習された類似度に置き換える。$n = 200$ 個の入力 $\mathbf{x}_i$ を $[0, 5]^5$ から一様に引き、ラベルを標準ガウスノイズ付きの $y_i = 2\sin(x_{i1}) + x_{i1} + \epsilon_i$ とすることで、第 1 座標のみが意味をもつようにする。クエリとキーのスコアを

$$a(\mathbf{q}, \mathbf{k}) = -\tfrac{1}{2}\|\mathbf{W}(\mathbf{q} - \mathbf{k})\|^2, \qquad \mathbf{W} = \mathrm{diag}(w_1, \dots, w_5), \tag{10.8}$$

で与え、Equation 10.2 と同様に正規化し、$\mathbf{W}$ を 1 つ抜き誤差（$\mathbf{x}_i$ における推定値からペア $(\mathbf{x}_i, y_i)$ を除外した誤差）に当てはめよ。新たに生成した 1000 個の入力上で、ノイズのない関数に対するテスト誤差を測定せよ。

   1. 等方的なカーネル $w_1 = \cdots = w_5 = 1/\sigma$ を、σ に関するグリッド探索によって当てはめよ。
   2. 等方的な解を初期値として、5 つのスケールを勾配降下法で当てはめよ。学習されたスケールを各座標の関連度と比較し、さらにテスト誤差を等方的カーネルおよび第 1 座標のみを最良のバンド幅で用いるオラクルのテスト誤差と比較せよ。
   3. 学習された計量とオラクル計量の 1 つ抜き誤差をそれぞれのテスト誤差と比較し、同じ比較を $n = 1000$ でも繰り返せ。有限サンプルから類似度を学習することについて、この結果は何を示しているか。
   4. 一般の行列 $\mathbf{W} \in \mathbb{R}^{5\times 5}$ はクエリとキーを同一の線形変換で写すのに対し、学習されるアテンションは別々の行列 $\mathbf{W}_q$ と $\mathbf{W}_k$ でそれらを写す。共有の変換が保証し、別々の変換では失われるスコア a の性質を 2 つ挙げよ。

10.2 アテンションのスコアリングとマスキング

10.1.2 節では、アテンション重みは入力空間におけるユークリッド距離上の固定されたガウスカーネルによって決定されていた。しかし系列モデルでは、類似度は学習されなければならない。たとえば主語を探す動詞が必要とするのは、その統語的役割を果たす単語であって、生の埋め込み空間で最も近い単語ではないからである。ソフトな検索を訓練可能なネットワーク層へと変えるにあたっては、2 つの疑問が生じる。すなわち、その層はどのアテンションスコア関数 $a(\mathbf{q}, \mathbf{k})$ を計算すべきか、そしてどのクエリ・キーのペアを検索から除外すべきかである。

$n = 1{,}024$ トークンを読み取るヘッドは 1 系列あたり $n^2 = 1{,}048{,}576$ 個のクエリ・キーのペアを評価するため、スコアは低コストで計算でき、行列積のハードウェアへ無理なくマッピングできるものでなければならない。さらに、可変長のミニバッチに含まれるパディングトークンや、自己回帰言語モデルにおける未来のトークンのような無効なペアは、正規化の前に重みがゼロになるよう除外しなければならない。

本節では、**スケール化ドット積アテンション**（scaled dot-product attention、$\mathbf{q}^\top\mathbf{k}/\sqrt{d}$）を導出し、$1/\sqrt{d}$ 係数が初期化時におけるソフトマックスの飽和をどのように防ぐかを説明する。続いてブールマスキングとマスクの合成を実装し、学習されたアテンションが出力の教師信号のみから系列のアライメントを復元する様子を示す。

10.2.1 ドット積アテンション

10.1.2 節の単位帯域幅を持つガウスカーネルについて負の二乗距離を展開すると、内積と 2 つの二乗ノルムの和が現れる。

$$a(\mathbf{q}, \mathbf{k}_i) = -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 = \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \tag{10.9}$$

$-\frac{1}{2}\|\mathbf{q}\|^2$ はすべてのキー i にわたって定数であるため、ソフトマックス正規化によって相殺される。すべてのキーが等しいノルムを持つならば、$-\frac{1}{2}\|\mathbf{k}_i\|^2$ も同様に相殺される。キーのノルムが異なる場合であってもこれら両方のノルム項を落とすと、**ドット積スコア**（dot-product score）$\mathbf{q}^\top \mathbf{k}_i$ が得られる。大きな $\|\mathbf{k}_i\|$ にペナルティを課すガウスカーネルとは異なり、ドット積ではキーのノルムが乗法的な事前分布として働くため、長いキーほど同じ方向を向くあらゆるクエリからより大きな重みを引き寄せる（演習問題 1c）。

これに対し、初期のニューラル機械翻訳モデルでは、**加法アテンション**（additive attention）として知られる隠れ層 1 層の MLP が用いられた (Bahdanau et al. 2015)。

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \tag{10.10}$$

ここで、$\mathbf{W}_q \in \mathbb{R}^{r \times d_q}$ と $\mathbf{W}_k \in \mathbb{R}^{r \times d_k}$ は学習される行列、$\mathbf{w}_v \in \mathbb{R}^r$ は重みベクトルである。加法スコアリングとドット積スコアリングの漸近的な演算回数は同程度であるが（$O(nmr)$ 対 $O(nmd)$。ただし n はクエリ数、m はキー数）、加法アテンションでは隠れ層の活性化値からなる $n \times m \times r$ のテンソルをメモリ上に実体化し、要素ごとに tanh を適用しなければならない。これに対し、ドット積アテンションは $n \times m$ 個のスコアすべてを 1 回の行列積 $\mathbf{Q}\mathbf{K}^\top$ で計算する。双線形スコアは MLP より一般性では劣るものの、クエリとキー自体が学習された射影 $(\mathbf{W}_q\mathbf{q})^\top(\mathbf{W}_k\mathbf{k})$ によって生成される場合には、その一般性の低下は実用上何ら問題にならない（演習問題 6 も参照）。

もう 1 つ重要なスケーリング係数が残っている。$\mathbf{q}, \mathbf{k} \in \mathbb{R}^d$ の各成分が、平均 0、分散 1 の独立な確率変数であると仮定しよう。このとき各項 $q_j k_j$ は平均 0、分散 1 となるため、その和 $\mathbf{q}^\top \mathbf{k}$ は平均 0、分散 d となる（標準偏差は 8 となる。ただし $d = 64$ の場合）。大きなドット積によってソフトマックスが勾配消失（vanishing gradients）の領域へ押し込まれるのを大きな d において防ぐため、Vaswani et al. (2017) は $\sqrt{d}$ で割って単位分散を回復させ、次の**スケール化ドット積**スコアを与えた。

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}. \tag{10.11}$$

ここで $1/\sqrt{d}$ は、10.1.2 節における逆温度 $1/\sigma^2$ の役割を果たしており、ベクトルの幅 d に応じて自動的に較正される。続いてソフトマックス正規化により、アテンション重みは次式で与えられる。

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1}^{m} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}, \tag{10.12}$$

これにより、出力 $\sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\, \mathbf{v}_i$ が得られる。

10.2.1.1 計算例

$d = 4$（したがって $1/\sqrt{d} = 1/2$）とし、2 つのクエリと 3 組のキー・値のペアを次のように定める。

$$\begin{aligned} \mathbf{q}_1 &= (2,2,0,0), & \mathbf{k}_1 &= (1,1,0,0), & \mathbf{v}_1 &= (1,0), \\ \mathbf{q}_2 &= (0,0,2,0), & \mathbf{k}_2 &= (0,0,1,1), & \mathbf{v}_2 &= (0,1), \\ & & \mathbf{k}_3 &= (0,0,1,-1), & \mathbf{v}_3 &= (1,1). \end{aligned}$$

1. **スコア：** $\mathbf{q}_1$ と 3 つのキーとの生のドット積は $(4,0,0)$ であり、$\mathbf{q}_2$ では $(0,2,2)$ となる。
2. **スケーリング：** $\sqrt{4} = 2$ で割ると、$(2,0,0)$ と $(0,1,1)$ が得られる。
3. **ソフトマックス：** $\mathbf{q}_1$ は重み $(e^2/(e^2+2), 1/(e^2+2), 1/(e^2+2)) \approx (0.787, 0.107, 0.107)$ を割り当て、$\mathbf{q}_2$ は $(1/(1+2e), e/(1+2e), e/(1+2e)) \approx (0.155, 0.422, 0.422)$ を割り当てる。
4. **重み付き和：** $\mathbf{o}_1 = 0.787\,\mathbf{v}_1 + 0.107\,\mathbf{v}_2 + 0.107\,\mathbf{v}_3 \approx (0.893, 0.213)$ および $\mathbf{o}_2 \approx (0.578, 0.845)$ となる。

直交するキーであっても非ゼロのソフトマックス重み（0.107）を受け取り、クエリの非ゼロ成分上で同点となるキー（$\mathbf{k}_2$ と $\mathbf{k}_3$、$\mathbf{q}_2$ のもとで）は残りの重みを等分する。もしスケーリング係数 $1/2$ を省くと、$\mathbf{q}_1$ が $\mathbf{k}_1$ に与える重みは 0.787 から $e^4/(e^4+2) \approx 0.96$ へと鋭くなる。

10.2.1.2 スコアの分散と $1/\sqrt{d}$ 係数

1 つのスコアが他のすべてのスコアを大きなマージンで上回るとき、重みベクトル α はワンホットベクトルに近づき、ソフトマックスは**飽和**（saturate）する。$\alpha = \mathrm{softmax}(\mathbf{a})$ のヤコビ行列（Jacobian matrix）は次式で与えられる。

$$\frac{\partial \alpha}{\partial \mathbf{a}} = \mathrm{diag}(\alpha) - \alpha\alpha^{\top}, \tag{10.13}$$

この行列は、α がワンホットになるにつれてゼロ行列へと収束する（演習問題 2）。飽和した行では、スコアに関して逆伝播される勾配が消失するため、選択されなかったキーが訓練中に重みを獲得することは困難になる。

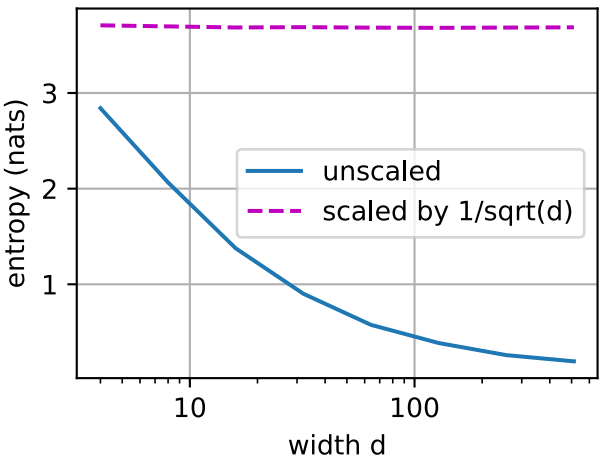
標準正規分布からランダムに生成したクエリ（2,048 個のクエリがそれぞれ 64 個のランダムなキーにアテンションを向ける設定）を用いて、次元 $d \in \{4, \ldots, 512\}$ にわたり、スケーリングなしのスコアと $1/\sqrt{d}$ でスケーリングしたスコアを比較することでこの効果を測定できる。64 個のキーに対する一様な重みのエントロピーは $\log 64 \approx 4.16$ nat であるのに対し、ワンホット分布のエントロピーは 0 である。

```
def softmax_weights(d, scale, num_keys=64, num_queries=2048):
    """Log-weights of random queries over random keys, scores scaled."""
    q = torch.randn(num_queries, d)
    keys = torch.randn(num_queries, num_keys, d)
    scores = (keys @ q[..., None]).squeeze(-1) * scale
    return F.log_softmax(scores, dim=-1)

def entropy(log_alpha):
    return -(log_alpha.exp() * log_alpha).sum(-1).mean().item()
```

```
def largest_weight(log_alpha):
    return log_alpha.exp().max(-1).values.mean().item()

def jacobian_norm(log_alpha):
    """Mean Frobenius norm of the Jacobian diag(alpha) - alpha alpha^T."""
    alpha = log_alpha.exp()
    J = torch.diag_embed(alpha) - alpha[..., :, None] * alpha[..., None, :]
    return torch.linalg.matrix_norm(J).mean().item()
```

ヘッド幅 d を対数軸上で 4 から 512 まで変化させると、スケーリングなしのドット積アテンションでは d の増大に伴って急激なエントロピー崩壊が生じる一方、$1/\sqrt{d}$ でスケーリングするとすべての幅にわたってアテンションのエントロピーが平坦に保たれることがわかる。

```
ds = [2**k for k in range(2, 10)]
entropies = [[entropy(softmax_weights(d, 1.0)) for d in ds],
             [entropy(softmax_weights(d, 1 / math.sqrt(d))) for d in ds]]
d2l.plot(ds, entropies, 'width d', 'entropy (nats)',
         legend=['unscaled', 'scaled by 1/sqrt(d)'], xscale='log')
```



下の表は、5 通りの幅におけるエントロピー、最大重みの平均、およびソフトマックスのヤコビ行列のフロベニウスノルム（Frobenius norm）の平均を示したものである。

```
print('      entropy (nats)    largest weight     Jacobian norm')
print('  d' + '   unscaled scaled' * 3)
for d in (4, 16, 64, 128, 512):
    stats = [(entropy(la), largest_weight(la), jacobian_norm(la))
             for la in (softmax_weights(d, 1.0),
                        softmax_weights(d, 1 / math.sqrt(d)))]
    print(f'{d:3d}' + ''.join(f'   {u:8.3f} {s:6.3f}'
                              for u, s in zip(*stats)))
```

```
    entropy (nats)    largest weight     Jacobian norm
d   unscaled scaled   unscaled scaled   unscaled scaled
4       2.782  3.701      0.285  0.105      0.256  0.180
```

```
 16      1.388  3.691      0.578  0.105      0.295  0.182
 64      0.584  3.682      0.791  0.108      0.209  0.184
128      0.372  3.683      0.862  0.107      0.158  0.184
512      0.182  3.681      0.926  0.108      0.094  0.184
```

$1/\sqrt{d}$ によるスケーリングを行うと、エントロピーはすべての幅にわたって $\log 64 - \frac{1}{2} \approx 3.66$ nat 付近に保たれ（演習問題 5）、最大重みは ≈ 0.11、ヤコビ行列のノルムは ≈ 0.18 となる。スケーリングがない場合、スコアの標準偏差は $\sqrt{d}$ に比例して増大する。$d = 512$ に達する頃にはエントロピーは ≈ 0.2 nat まで崩壊し、ランダム初期化の時点で最大のキーが重みの 90% 超を占め、ヤコビ行列のノルムは半減する。$1/\sqrt{d}$ によるスケーリングは、初期状態のソフトマックスの鋭さをヘッド幅から切り離しつつ、訓練中には学習された射影が重みを自由に鋭くできるようにする（クエリとキーが整列するようになるとドット積は $O(d)$ で増大するため、最大更新パラメータ化（maximal update parametrization）における $1/d$ スケーリングの動機となる。9.9 節を参照）。近年の多くのモデルでは、温度の調整を射影に一切委ねず、クエリとキーを単位長に正規化したうえでスコアのスケールを独立したパラメータとして学習する。これは **QK-norm** と呼ばれる設計であり、11.1 節で改めて扱う。

このスケーリングは幅 d に対しては鋭さを一定に保つが、キーの数 m に対しては一定に保たない。スコアが単位分散を持つとき、エントロピーは約 $\log m - \frac{1}{2}$ となるため、重みはおよそ $0.6\,m$ 個のキーに分散する。つまりコンテキストがどれほど長くなっても、その一定の割合に広がる。検索も同様に希釈される。一致するキーが他の $m-1$ 個の各キーに対してスコア差 g だけリードしている場合、そのキーが受け取る重みは $1/(1+(m-1)e^{-g})$ である。$g = 9$ のスコア差は、約 0.9 の重み（キー数が 1,000 個のとき）を与えるが、0.1 未満（キー数が 100,000 個のとき）にまで落ち込む。鋭さを維持するには、スコア差が $\log m$ のオーダーで増大しなければならない。したがって、あるコンテキスト長で訓練されたモデルは、より長いコンテキスト長ではより拡散したアテンションを向けることになり、これがコンテキスト長に応じてスコアのスケールを引き上げるモデルが存在する理由である（10.8 節）。

10.2.2 マスキング

特定のクエリ・キーのペアをアテンションから除外する代表的な条件が 2 つある。

1. **パディング：**長さの異なる系列を `<blank>` トークンでパディングしてミニバッチを構成するとき、どのクエリもパディング位置にアテンションを向けてはならない。
2. **因果性：**自己回帰型の言語モデリングでは、位置 t のクエリは位置 $j \le t$ のキーにのみアテンションを向けることができる。

これら両方の制約は、ブール型の**マスク**テンソル（`True` の位置でクエリ i がキー j にアテンションを向けることを許可する）によって表される。

10.2.2.1 マスク付きソフトマックス演算

密なテンソルの形状を保つため、**マスク付きソフトマックス**（masked softmax）は除外されたスコア（`~mask`）を大きな負の数に置き換えてから `softmax` を呼び出す。この埋め値の選択には、浮動小数点形式（floating-point format）に応じた注意が必要である。

- -10^6 のようなハードコードされた定数は、float16（表現可能な最大の有限絶対値は $65{,}504$）でオーバーフローする。
- リテラルの $-\infty$ は、クエリ行の全体がマスクされると必ず `NaN` を生じる。数値的に安定なソフトマックスは行の最大値 $-\infty$ を引くため、$-\infty - (-\infty) = \text{NaN}$ となるからである。
- テンソルのデータ型における最小の**有限**値（`finfo(dtype).min`）を用いれば、float16、bfloat16、float32 のいずれにおいても `NaN` を生じることなく、除外された指数項を厳密

なゼロへとアンダーフローさせることができる。すべてのキーがマスクされたクエリ行（`~mask.any(-1)`）に対しては、`masked_softmax` が結果の重みを明示的にゼロにする。

```
def masked_softmax(X, mask=None):
    """Softmax over the last axis with zero weight where mask is False."""
    if mask is None:
        return F.softmax(X, dim=-1)
    # Most negative finite value: zero weight at any precision, no NaN
    X = X.masked_fill(~mask, torch.finfo(X.dtype).min)
    weights = F.softmax(X, dim=-1)
    # A query without any valid key gets zero weights, not a uniform row
    return torch.where(mask.any(-1, keepdim=True), weights, 0.0)
```

`mask` は `X` に対してブロードキャスト可能でありさえすればよいため、系列内のすべてのクエリで共有されるパディングマスクは形状 `(batch, 1, keys)` で保持できる。

```
def padding_mask(valid_lens, num_keys):
    """Boolean mask that is True for the first valid_lens keys."""
    # valid_lens of shape (batch,) gives (batch, 1, keys); per-query
    # lengths of shape (batch, queries) give (batch, queries, keys)
    if valid_lens.dim() == 1:
        valid_lens = valid_lens[:, None]
    keys = torch.arange(num_keys, device=valid_lens.device)
    return keys < valid_lens[..., None]
```

長さ 4 で有効長が 2 と 3 であるスコアゼロの 2 系列からなるバッチに対し、`masked_softmax` は各行の有効なプレフィックス上に一様に重みを分配する。

```
masked_softmax(torch.zeros(2, 2, 4), padding_mask(torch.tensor([2, 3]), 4))
```

```
tensor([[[0.5000, 0.5000, 0.0000, 0.0000],
         [0.5000, 0.5000, 0.0000, 0.0000]],

        [[0.3333, 0.3333, 0.3333, 0.0000],
         [0.3333, 0.3333, 0.3333, 0.0000]]])
```

10.2.2.2 因果マスキング

因果マスク（causal mask）は、クエリ i がキー j にアテンションを向けることを $j \leq i$ の場合に限り許可し、$n(n-1)/2$ 個を全 n^2 ペアのうちから除外する下三角行列を形成する（除外数は $523{,}776$ ペアとなる、ただし $n = 1{,}024$ の場合）。`num_queries < num_keys` のとき（キャッシュを用いた生成時など）には、`causal_mask` はクエリを系列の末尾の位置に揃える。

```
def causal_mask(num_queries, num_keys=None, device=None):
    """Boolean (queries, keys) mask, True where a query may see a key."""
    # Queries are the last positions, as when new tokens attend to a
    # cache: query i sees key j if j <= i + num_keys - num_queries
    num_keys = num_queries if num_keys is None else num_keys
    i = torch.arange(num_queries, device=device)[:, None]
    j = torch.arange(num_keys, device=device)[None, :]
```

```
    return j <= i + (num_keys - num_queries)
```

10.2.2.3 マスクの合成

複数の制約が同時に適用されるとき、あるペアが有効となるのはすべての制約がそのペアを許可する場合に限られるため、ブールマスクはブロードキャストのもとでの論理積（logical AND、&）によって合成される（Figure 10.2）。

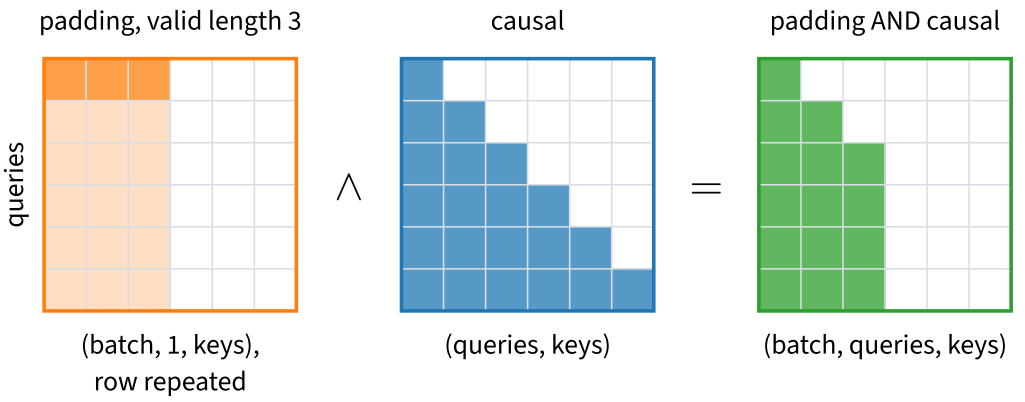
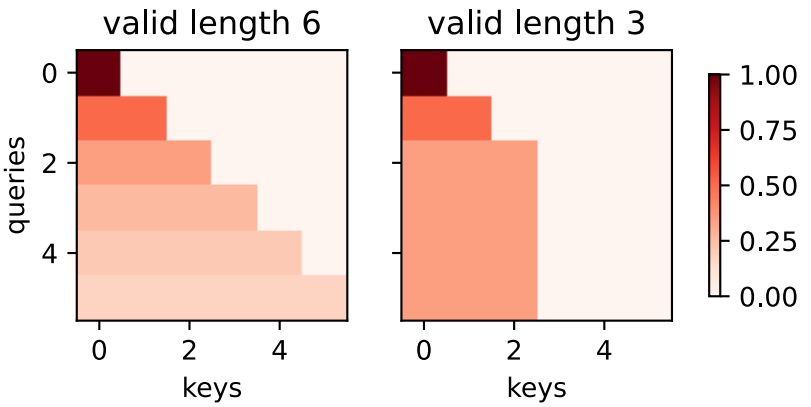


Figure 10.2: マスクは論理積で合成される。長さ 6、有効長 3 の系列に対し、パディングマスクはキーの 1 行分を保持し、ブロードキャストによってすべてのクエリに繰り返される（薄色）。因果マスクは下三角であり、両者の論理積は双方が許可するペアのみを許可する。行はクエリ、列はキーであり、塗りつぶされた要素が許可されたペアである。

等しいスコアに対してパディングマスク（有効長 6 と 3）と因果マスクを組み合わせると、1 番目の系列では完全な因果下三角が得られ、2 番目の系列ではすべての行がキー 3 で打ち切られる。

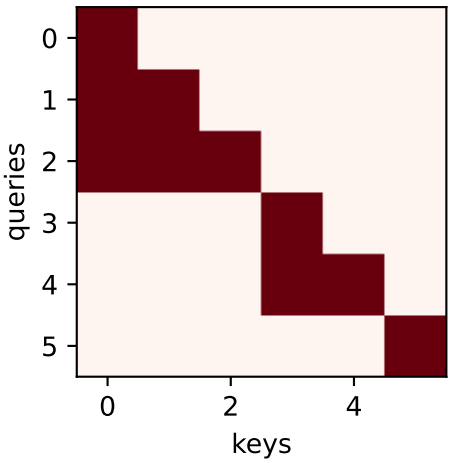
```
n = 6
padding = padding_mask(torch.tensor([6, 3]), n)  # (batch, 1, key)
causal = causal_mask(n)                          # (query, key)
mask = padding & causal                          # (batch, query, key)
weights = masked_softmax(torch.zeros(2, n, n), mask)  # Equal scores
d2l.show_heatmaps(weights[None], xlabel='keys', ylabel='queries',
                  titles=['valid length 6', 'valid length 3'],
                  figsize=(5, 2.5))
```



短い方の系列では、クエリ 4 から 6 自体がパディングであるが、因果下三角は常にキー 1 まで遡るため、いずれのクエリも依然として有効なキーを持つ。これは論理積の性質ではなく、マスクの形状による性質である。もしパディングマスクを、各クエリが直近の少数のキーしか参照できない**スライディングウィンドウ**と組み合わせたならば、ウィンドウ全体がパディング領域内に収まるクエリは有効なキーを 1 つも持たなくなるだろう。単独では無害なマスクであっても、組み合わせると行が空になることがあり、`masked_softmax` がゼロにするのはまさにこのケースである。

同じブール代数は、パディングのオーバーヘッドを避けるために複数の短い文書を 1 行に連結する**パック化系列**（packed sequences）も扱える。同一文書を表す指示行列 `doc[:, None] == doc[None, :]` と `causal_mask` の論理積をとると、ブロック対角の因果マスクが得られる。

```
doc = torch.tensor([0, 0, 0, 1, 1, 2])  # Document of each position
same_doc = doc[:, None] == doc[None, :]
packed = same_doc & causal_mask(6)
d2l.show_heatmaps(packed.float()[None, None], xlabel='keys',
                  ylabel='queries', cbar=False)
```



10.2.3 ミニバッチに対するアテンション

n 個のクエリを $\mathbf{Q} \in \mathbb{R}^{n \times d}$ に、m 個のキーを $\mathbf{K} \in \mathbb{R}^{m \times d}$ に、そして m 個の値を $\mathbf{V} \in \mathbb{R}^{m \times d_v}$ に積み重ねると、スケール化ドット積アテンションは 2 回の行列積と行ごとのソフトマックスとして表される（Figure 10.3）。

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times d_v}. \tag{10.14}$$

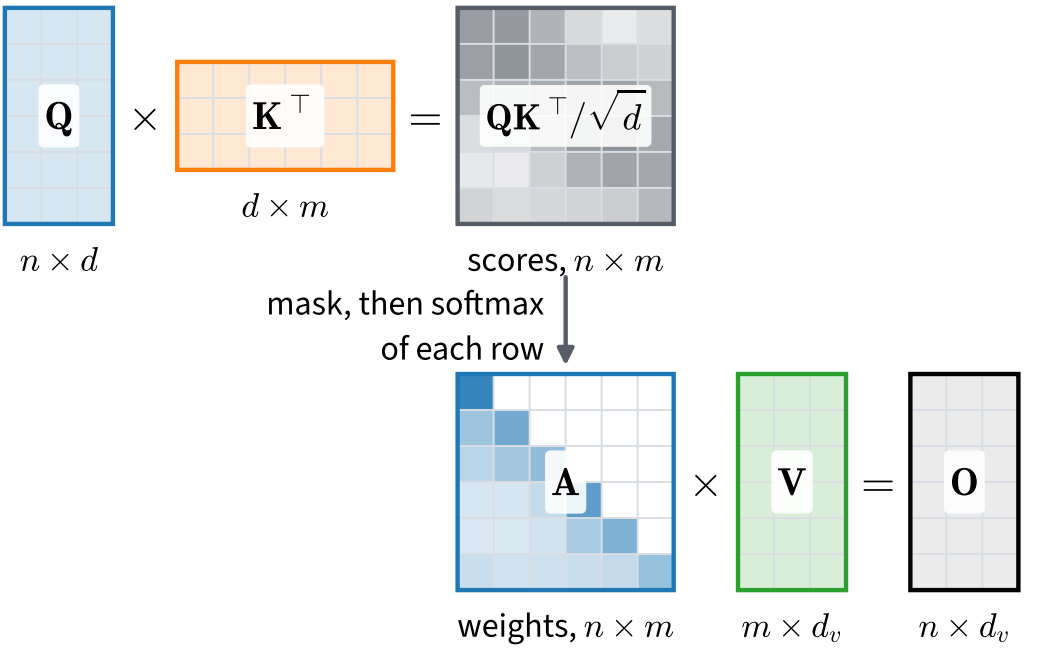


Figure 10.3: スケール化ドット積アテンションにおける形状。$\mathbf{Q}\mathbf{K}^\top/\sqrt{d}$ は $n \times m$ 個すべてのクエリ・キーのペアのスコアを保持する。マスキングと各行にわたるソフトマックスによってこれらが重み $\mathbf{A}$ へと変換され（ここでは上三角を空にする因果マスクを使用）、さらに $\mathbf{V}$ を掛けることでクエリごとに 1 つの値の重み付き和が得られる。

先頭にバッチ軸（およびヘッド軸）を持つバッチ入力に対しては、バッチ行列積 `@` が最後の 2 次元に対して演算し、先頭の各軸を自動的にブロードキャストする。GPT-2 small の 1 つのヘッド（$n = m = 1{,}024$、$d = d_v = 64$）では、スコア行列は 1,048,576 個の浮動小数点数（fp32 で 4 MiB）を保持し、2 回の積には $4n^2 d \approx 0.27$ GFLOPs を要する。いずれも n に関して二次のオーダーである。

10.2.3.1 DotProductAttention クラス

`DotProductAttention` は、任意のマスキングとアテンション重みへのドロップアウト（デフォルトは 0.0）を備えて Equation 10.14 を実装し、確認用に直近のアテンション重みを `attention_weights` に保存する。

```
class DotProductAttention(nn.Module):
    """Scaled dot-product attention over the last two axes."""
    def __init__(self, dropout=0.0):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.attention_weights = None

    # Leading axes (batch, and heads if present) are carried along.
    # Shape of queries: (..., no. of queries, d)
    # Shape of keys: (..., no. of key-value pairs, d)
    # Shape of values: (..., no. of key-value pairs, value dimension)
    # mask: boolean, broadcastable to (..., no. of queries, no. of keys),
    # True where a query may attend to a key
    def forward(self, queries, keys, values, mask=None):
        d = queries.shape[-1]
        scores = queries @ keys.transpose(-2, -1) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, mask)
        return self.dropout(self.attention_weights) @ values
```

DotProductAttention を 10.2.1.1 節のベクトルに対して評価すると、先ほどの手計算が再現される。

```
queries = d2l.tensor([[[2.0, 2, 0, 0], [0, 0, 2, 0]]])           # (1, 2, 4)
keys = d2l.tensor([[[1.0, 1, 0, 0], [0, 0, 1, 1], [0, 0, 1, -1]]])  # (1, 3, 4)
values = d2l.tensor([[[1.0, 0], [0, 1], [1, 1]]])                  # (1, 3, 2)
attention = DotProductAttention()
output = attention(queries, keys, values)
print('weights:', np.asarray(attention.attention_weights[0]).round(3))
print('output: ', np.asarray(output[0]).round(3))
```

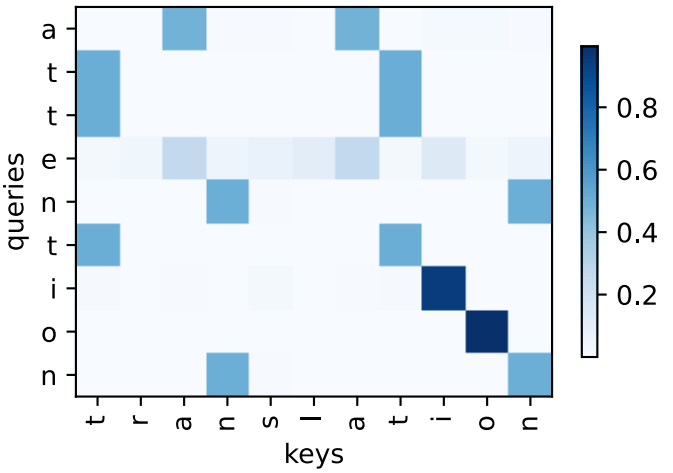
```
weights: [[0.787 0.107 0.107]
 [0.155 0.422 0.422]]
output:  [[0.893 0.213]
 [0.578 0.845]]
```

10.2.3.2 2つの単語の文字マッチング

射影を伴わない内容マッチングが何を識別でき、何を識別できないかを確かめるため、"attention" と "translation" に現れる異なる各文字に標準正規分布からランダムに生成した埋め込み $\mathbf{e} \in \mathbb{R}^{32}$ を割り当て、"attention" の各文字をクエリとして "translation" の各文字を検索させてみよう。同一の文字どうしではスケール化スコアの期待値が $\mathbb{E}[\|\mathbf{e}\|^2]/\sqrt{32} = \sqrt{32} \approx 5.7$ となる一方、異なる文字どうしではスコアの期待値が 0、分散が単位分散となる。

```
words = 'attention', 'translation'
letters = sorted(set(''.join(words)))
emb = torch.randn(len(letters), 32)
queries, keys = (emb[[letters.index(c) for c in w]][None] for w in words)
attention = DotProductAttention()
attention(queries, keys, keys)
d2l.show_heatmaps(attention.attention_weights[None], xlabel='keys',
                  ylabel='queries', figsize=(4, 3.5), cmap='Blues',
```

```
                xticklabels=words[1], yticklabels=words[0])
```



ここから 3 つの挙動が際立って見て取れる。

1. 一意に一致する文字（i、o）は、ほぼすべての重みを対応する相手の文字に集中させる。
2. 複数回一致する文字（a、t、n）は、"translation" 内に現れるその文字のすべての出現箇所に重みを均等に分割する。内容のみからなるキーは位置の識別情報を持たないためである。
3. 一致する相手のないクエリ（e）は、ランダムなキー全体に確率質量を分散させる。関連するキーが存在しなくてもソフトマックスの重みの総和は 1 にならなければならないからである（訓練済みの言語モデルでは、先頭トークンがこのような行き場のない質量を**アテンションシンク**として吸収することが多い (Xiao et al. 2024)）。

10.2.4 学習されたアライメントとしてのアテンション

学習可能なアテンションは、ニューラル機械翻訳 (Bahdanau et al. 2015) において、系列対系列 RNN（12.2 節）が抱えていた固定長ベクトルのボトルネックを解消するために考案された。各デコーディングステップでデコーダがすべてのエンコーダ状態にクエリを投げると、学習されたアテンション重みは原言語と目的言語の間の単語アライメントを自発的に復元した。それは単調な対角線を基本としつつ、語順が異なる箇所（たとえば英語の「**black cat**」に対するフランス語の「**chat noir**」）では非対角の交差を伴うものであった。

最小限のモデル `ToyAligner` を用いて、アライメントが創発する様子を観察できる。入力系列は 8 個のトークン（サイズ 16 の語彙から抽出）からなり、目標系列は固定された置換 π（位置 3 と 4 を入れ替える）を適用して得られる。各出力位置 j は学習されるクエリベクトルを持ち、入力側の各キーは連結されたワンホットトークンとワンホット入力位置の線形射影であり、値は入力トークンのワンホットベクトルである。

```
class ToyAligner(nn.Module):
    """One learned query per output position; keys from token and position."""
    def __init__(self, seq_len=8, vocab_size=16, d=16):
        super().__init__()
        self.vocab_size, n_in = vocab_size, vocab_size + seq_len
        self.queries = nn.Parameter(torch.randn(seq_len, d))
```

```
        self.W_k = nn.Parameter(torch.randn(n_in, d) / math.sqrt(n_in))
        self.attention = DotProductAttention()

    def forward(self, X):
        # X: (batch, seq_len) token ids; the values are their one-hot vectors
        tokens = F.one_hot(X, self.vocab_size).float()
        eye = torch.eye(X.shape[1], device=X.device)
        positions = eye.expand(len(X), -1, -1)
        keys = torch.cat((tokens, positions), -1) @ self.W_k
        # The queries broadcast over the batch: (batch, seq_len, vocab_size)
        return self.attention(self.queries, keys, tokens)
```

並べ替えられた目標トークンの負の対数尤度（negative log-likelihood）を最小化することで、`ToyAligner` を Adam により 100 ステップ訓練する。

```
perm = torch.tensor([0, 1, 2, 4, 3, 5, 6, 7])  # Output j copies source perm[j]

def align_data(n, seq_len=8, vocab_size=16):
    X = torch.randint(0, vocab_size, (n, seq_len))
    return X, X[:, perm]

def train_aligner(num_steps=100):
    model = ToyAligner()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.05)
    for _ in range(num_steps):
        X, Y = align_data(128)
        loss = -model(X).gather(-1, Y[..., None]).log().mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model
```

未訓練のモデルと 3 回の独立な訓練実行を 1,000 本のテスト系列上で比較すると、勾配降下法は確実に 100% の正解率へと収束し、出力位置にわたる平均で重みの 99% 超を $\pi(j)$ に割り当てることがわかる。

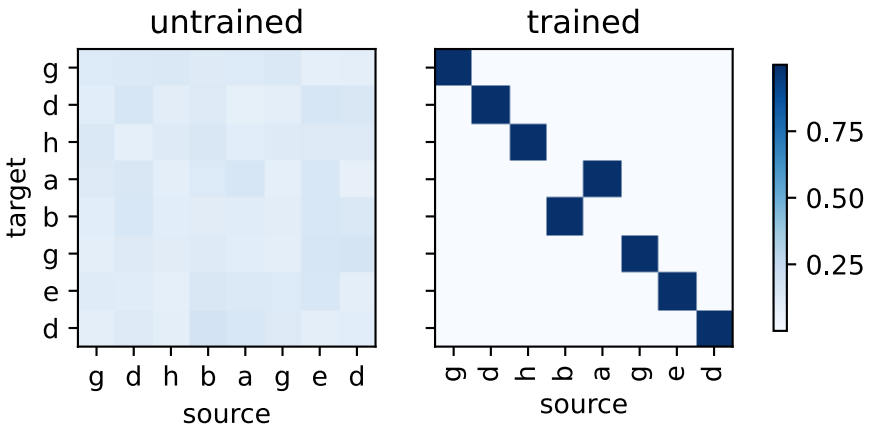
```
X, Y = align_data(1000)

def evaluate_aligner(model, X, Y, name):
    with torch.inference_mode():
        accuracy = (model(X).argmax(-1) == Y).float().mean()
    weights = model.attention.attention_weights  # (1000, 8, 8)
    largest = (weights.argmax(-1) == perm).float().mean()
    on_perm = weights[:, torch.arange(8), perm].mean()
    print(f'{name:>9}: accuracy {accuracy:.3f}, largest weight on pi(j) '
          f'{largest:.3f}, mean weight on pi(j) {on_perm:.3f}')

untrained = ToyAligner()
evaluate_aligner(untrained, X, Y, 'untrained')
for run in range(1, 4):
    model = train_aligner()
```

```
    evaluate_aligner(model, X, Y, f'run {run}')
```

```
untrained: accuracy 0.272, largest weight on pi(j) 0.161, mean weight on pi(j) 0.132
    run 1: accuracy 1.000, largest weight on pi(j) 1.000, mean weight on pi(j) 1.000
    run 2: accuracy 1.000, largest weight on pi(j) 1.000, mean weight on pi(j) 0.999
    run 3: accuracy 1.000, largest weight on pi(j) 1.000, mean weight on pi(j) 1.000
```

最初のテスト系列に対する訓練前後のアテンション重み行列を可視化すると、ToyAligner が構造のない初期分布を π の厳密な置換行列（位置 3 と 4 に非対角の入れ替えを持つ）へと変化させていることが確認できる。

```
letters = 'abcdefghijklmnop'  # One letter per token id
maps = d2l.stack([untrained.attention.attention_weights[0],
                  model.attention.attention_weights[0]])
d2l.show_heatmaps(maps[None], xlabel='source', ylabel='target',
                  titles=['untrained', 'trained'], figsize=(5.5, 3),
                  cmap='Blues', xticklabels=[letters[int(i)] for i in X[0]],
                  yticklabels=[letters[int(i)] for i in Y[0]])
```



キーがトークンの識別情報と並んで位置を符号化しているため、入力系列内で同じトークンが繰り返される場合であっても、訓練後のアテンションヒートマップは π の厳密な置換行列を復元する。しかし一般の系列モデリングでは、クエリは系列自体から導出されなければならず、1 つの位置が複数の異なる値を同時に取り出す必要があるかもしれず（10.3 節）、さらに位置エンコーディング（positional encoding）は異なる系列長にわたって汎化しなければならない（10.4 節）。

10.2.5 まとめ

スケール化ドット積スコアリング $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}/\sqrt{d}$ は、ガウスカーネルにおける二次のノルムペナルティを省き、すべてのペア間のスコアを 1 回の行列積 $\mathbf{Q}\mathbf{K}^\top$ で評価する。$\sqrt{d}$ で割ることにより、単位分散の入力に対してスコアの単位分散が保たれ、ヘッド幅 d が増大してもソフトマックスの飽和やスコア勾配の消失が防がれる。系列のパディングや因果デコーディングといった構造上の制約は、論理積のもとで合成されるブールマスクによって適用され、`finfo(dtype).min` と空行の明示的なゼロ化を併用することで `float16` でのオーバーフローや NaN の伝播を回避する。最後に、内容のみを持つキーは重複するトークン間でアテンションを均等に分割するのに対し、位置情報を備えたキーを用いれば系列のアライメントをエンドツーエンドで学習できる。

10.2.6 演習問題

1. **スコアの不変性**。ガウススコア Equation 10.9 とスケール化ドット積 Equation 10.11 は、いずれもソフトマックス Equation 10.12 を通じてアテンション重みを定義する。
   1. 2 つのスコアのそれぞれについて、次のどの変換が重みを変化させないかを求めよ。すべてのキーに 1 つのベクトル $\mathbf{c}$ を加える変換、クエリとすべてのキーに $\mathbf{c}$ を加える変換、クエリとすべてのキーに 1 つの直交行列を掛ける変換。
   2. クエリとキーは通常、入力からのアフィン写像 $\mathbf{q} = \mathbf{W}_q\mathbf{x} + \mathbf{b}_q$ および $\mathbf{k}_i = \mathbf{W}_k\mathbf{y}_i + \mathbf{b}_k$ によって計算される。$\mathbf{q}^\top \mathbf{k}_i$ を 4 つの項に展開し、重みが $\mathbf{b}_k$ に依存しないことを示せ。$\mathbf{b}_k$ に関する損失の勾配はどうなるか。
   3. $\mathbf{b}_q$ は 4 つの項のうちどれを通じて重みに影響を与えるか。また、バイアスなしのスコアでは表現できない、キーに対するどのような選好をそれによって表現できるか。
2. [code] **ソフトマックスのヤコビ行列**。`jacobian_norm` はヤコビ行列 Equation 10.13 のフロベニウスノルムを測定する。
   1. Equation 10.13 を Equation 10.12 から導出し、有限のスコアに対してその零空間が全成分 1 のベクトルによって張られることを示し、さらにランダムなスコアベクトル上で自動微分と比較してこの式を検証せよ。
   2. k 個のキー（全 m 個のうち）に一様に広がる重みに対してノルムを計算し、それを最大化する k を求め、ランダムな重みベクトルをサンプリングしてこの最大値を超えるものがないことを確認せよ。この結果を用いて、表のスケーリングなしのノルムが $d = 4$ から $d = 16$ にかけて上昇した後に低下する理由を説明せよ。
3. **完全にマスクされた行**。`masked_softmax` は除外されたスコアをそのデータ型（dtype）の最小の有限値で上書きし、続いて有効なキーを持たないクエリの行をゼロにする。
   1. すべてのキーが除外された 4 個のスコアからなる行を作成せよ。それを float16 の -10^6、float32 の $-\infty$、および float32 の `finfo(dtype).min` でそれぞれ埋め、`masked_softmax` の最後のゼロ化を行わずにフレームワークのソフトマックスをそれぞれに適用せよ。得られる結果を報告し、行の最大値を引く計算過程を追って各結果を説明せよ。
   2. クエリ・キー・値のすべてが同一の系列から得られる自己アテンション層において、10.2.2.3 節の合成マスクで起こり得るように、パディング位置にあるクエリが有効なキーを 1 つも持たないとし、除外されたスコアが $-\infty$ で上書きされてその位置の出力が NaN になったとする。その出力自体は破棄され、次のアテンション層ではすべての有効なクエリがそのパディング位置をキーから除外する。それにもかかわらず、次の層でその系列のすべての有効な位置の出力が NaN によって汚染される理由を説明せよ。
4. **合成マスク**。10.2.2.3 節ではマスクを論理積で合成し、単独では無害なマスクでも組み合わせると有効なキーを持たないクエリが生じ得ることを指摘した。幅 w の因果スライディングウィンドウは、位置 t のクエリが位置 $t - w + 1, \ldots, t$ のキーにアテンションを向けることを許可する。
   1. 長さ 6、有効長 3 の系列に対し、パディングマスクと幅 2 の因果スライディングウィンドウを 6×6 のブール行列として書き下せ。それぞれのマスク単独ではすべてのクエリに少なくとも 1 つのキーが残ることを確認し、両者の論理積で行が空になるクエリを求めよ。
   2. すべてのマスクが各クエリに自身の位置へのアテンションを許可するならば、それらのマスクの論理積には空の行が生じないことを示せ。パディングマスク、因果マスク、スライディングウィンドウマスク、パック化系列マスクのうち、この条件を満たすものはどれか。また、その答えは先ほど見つけた空の行について何を示唆するか。
   3. 実装によっては、許可されたペアの要素を 0、除外されたペアの要素を $c < 0$ とするバイアス行列を加算し、バイアス同士の足し算によってマスクを合成するものがある。c として単精度における最小の有限値（約 -3.4×10^{38}）をとる。単精度においてこ

のようなバイアスを 2 つ足し合わせた和を計算し、すべてのキーが両方のマスクで除外されている行と、すべてのキーが除外されているが一部のキーは片方のマスクのみで除外されている行のそれぞれに対して、ソフトマックスが何を返すかを求めよ。

5. **ランダムなスコアのエントロピー。**●上でプロットしたスケーリングありのエントロピー曲線は 3.7 nat 付近にとどまり、一様分布の値 $\log 64 \approx 4.16$ を下回っている。1 つのクエリ $\mathbf{q} \in \mathbb{R}^d$ を固定し、`softmax_weights` と同様に独立な標準正規分布の成分を持つ m 個のキーを抽出せよ。
   1. スケール化スコア $a_i = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}$ が平均 0、分散 $s^2 = \|\mathbf{q}\|^2/d$ の独立な正規確率変数であることを示し、s^2 の期待値を、$\mathbf{q}$ が `softmax_weights` と同様に抽出される場合について計算せよ。
   2. 重みのエントロピーは $H = \log Z - \sum_i \alpha_i a_i$（ただし $Z = \sum_j \exp(a_j)$）である。大きな m に対して Z/m と $\sum_i \exp(a_i)\, a_i/m$ をそれぞれの期待値で置き換え、次式を導出せよ。
   $$H \approx \log m - \frac{s^2}{2}.$$
   この近似式を $m = 64$ および $s = 1$ に対して評価せよ。
   3. この近似が成り立つには、Z がその平均の周りに集中する必要がある。Z の相対標準偏差を s と m の関数として計算し、そこからこの近似が成り立つための s^2 に関する条件を導出せよ。スケーリングなしのスコアでは $s^2 = \|\mathbf{q}\|^2$ であり、その期待値は d である。$m = 64$ に対し $s^2 = 4$ および $s^2 = 8$ における相対標準偏差を評価し、この近似値を上でプロットした $d = 4$ および $d = 8$ のスケーリングなしのエントロピーと比較せよ。
6. [code] **ゼロ近傍での加法スコア。**●加法スコア Equation 10.10 は、射影されたクエリと射影されたキーを tanh の内側で足し合わせる。
   1. tanh を恒等関数に置き換えると、アテンション重みがクエリに依存しなくなることを示せ。
   2. tanh をゼロの周りで 3 次まで展開し、重みがクエリに依存する最低次の項を特定し、さらに $\mathbf{W}_q$ と $\mathbf{W}_k$ の両方に小さな係数 ϵ を掛けたときにこの依存性がどのようにスケールするかを述べよ。
   3. 3 個のクエリと 6 個のキー（いずれも $\mathbb{R}^8$ 上で標準正規分布に従う成分を持つ）および隠れ層の幅 $r = 16$ に対して加法スコアを実装せよ。ここで $\mathbf{W}_q$、$\mathbf{W}_k$、$\mathbf{w}_v$ は、標準正規分布の成分をそれぞれの入力幅の平方根で割って抽出するものとする。$\mathbf{W}_q$ と $\mathbf{W}_k$ に $\epsilon \in \{1, 0.3, 0.1, 0.03\}$ を掛け、3 つのクエリの重み行間の最大差を多数回の抽出にわたって平均して測定し、導出したスケーリング則を検証せよ。これらのスケールのうち、重みがクエリに有意に依存するのはどれか。
7. [code] **アライメントの変種。**●10.2.4 節の `ToyAligner` は、出力位置ごとに学習される 1 つのクエリと、トークンおよび位置を保持するキーから置換 π を学習する。
   1. キーからワンホット位置を取り除き、トークンのみに依存するようにせよ。訓練の前に、出力位置 j が依然として重みを $\pi(j)$ に置けるかどうか、またモデルがおおよそどの程度の正解率に到達できるかを予測し、実際に訓練して比較せよ。その結果を 10.2.3.2 節の文字マッチングと関連づけよ。
   2. スコアを加法規則 Equation 10.10 に置き換え、各モデルが 99% の正解率に達するまでに要するステップ数を比較せよ。訓練の前に、演習問題 6 を用いて、射影を小さく初期化したときに加法モデルがどのような初期挙動を示すかを予測し、その予測を検証せよ。
   3. クエリとキーに標準正規分布による初期化のもとで幅 $d \in \{16, 256, 1024\}$ を与え、係数 $1/\sqrt{d}$ がある場合とない場合で訓練せよ。訓練曲線を初期化時点における重みのエントロピーと関連づけよ。
   4. 先行するトークンが語彙の指定された部分集合に属するときにのみ隣接する 2 要素を

入れ替えるといった、トークンに依存する並べ替えを `ToyAligner` では学習できないことを示せ（ヒント：このモデルのスコアはすべての位置で共有されるトークン項と位置項の和である。隣接する 2 要素のスコアを比較せよ）。これを可能にするには、クエリまたはキーが何に依存していなければならないかを述べよ。

10.3 自己アテンションと複数のヘッド

10.2 節の例では、クエリはある系列から得られ、キーと値は別の系列から得られていた。しかし、デコーダのみの言語モデルが持つ系列は 1 つだけである。各トークンを文脈化するには、クエリ・キー・値のすべてをその系列自体から導出しなければならない。

「**Alice gave Bob the book**」という文を考えよう。**gave** を解釈するには、与え手（**Alice**）、受け手（**Bob**）、対象物（**book**）を特定する必要がある。**自己アテンション**（self-attention）では (Cheng et al. 2016; Lin et al. 2017; Vaswani et al. 2017)、すべてのトークンが自身の表現をクエリ・キー・値へと射影するため、**gave** は文中の他の単語にクエリを送り、それらの値を集約できる。

しかし、1 回のアテンション検索が生成する確率分布はクエリあたり **1 つ**だけであり、返されるのも混合された単一のベクトルにとどまる。トークンが複数の異なる項を 1 つの平均に潰してしまうことなく取り出すには、どうすればよいだろうか。本節ではまず、この 5 語の文に対する自己アテンションの単一ヘッドの動作をたどり、次に 1 つのヘッドでは独立な 2 つの値を復元できない理由を示す情報理論的な下界を証明し、最後に**マルチヘッドアテンション**と**グループ化クエリアテンション**（grouped-query attention、GQA）を構成する。

10.3.1 自己アテンション

自己アテンションでは、各トークン表現 $\mathbf{x}_i \in \mathbb{R}^d$（$\mathbf{X} \in \mathbb{R}^{n\times d}$ の第 i 行）が、学習された線形射影 $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{d\times d_h}$ および $\mathbf{W}_v \in \mathbb{R}^{d\times d_v}$ を介して 3 つの異なる役割を果たす。すなわち、**クエリ** $\mathbf{x}_i\mathbf{W}_q$ はそのトークンが求めるものを指定し、**キー** $\mathbf{x}_i\mathbf{W}_k$ は他のクエリに対して提示するものを指定し、**値** $\mathbf{x}_i\mathbf{W}_v$ は選択されたときに返される内容を指定する。これにスケール化ドット積アテンション（scaled dot-product attention）を適用すると、幅 d_h の 1 つのアテンション**ヘッド**が得られる。

$$\operatorname{softmax}\left(\frac{(\mathbf{X}\mathbf{W}_q)(\mathbf{X}\mathbf{W}_k)^\top}{\sqrt{d_h}}\right)\mathbf{X}\mathbf{W}_v \in \mathbb{R}^{n\times d_v}. \tag{10.15}$$

10.3.1.1 1 つの文に対する単一ヘッド

Equation 10.15 を具体的にたどるため、「**Alice gave Bob the book**」の各単語を、4 次元の one-hot な文法的役割（人物、動作、物、機能語）と 5 次元の one-hot な単語識別子を連結した 9 次元ベクトルで表す。$\mathbf{W}_q$ と $\mathbf{W}_k$ は 4 次元の役割のみを $d_h = 2$ 次元に読み出し、$\mathbf{W}_v$ は 5 次元の単語指示ベクトルを抽出するように設定する。

役割	クエリ	キー
人物（Alice、Bob）	$(1, -1)$	$(1, 1)$
動作（**gave**）	$(1, 1)$	$(1, -1)$
物（book）	$(1, -1)$	$(1, 1)$
機能語（「the」）	$(0, 0)$	$(0, 0)$

```
tokens = ['Alice', 'gave', 'Bob', 'the', 'book']
# Role of each token: person, action, thing, function word
roles = d2l.tensor([[1.0, 0, 0, 0], [0, 1, 0, 0], [1, 0, 0, 0],
                    [0, 0, 0, 1], [0, 0, 1, 0]])
X = d2l.concat((roles, d2l.eye(5)), 1)  # (5, 9): the role, then the word
# Queries and keys read the role part (rows: person, action, thing,
# function word), values the word part
W_q = d2l.concat((d2l.tensor([[1.0, -1], [1, 1], [1, -1], [0, 0]]),
                  d2l.zeros((5, 2))))
W_k = d2l.concat((d2l.tensor([[1.0, 1], [1, -1], [1, 1], [0, 0]]),
                  d2l.zeros((5, 2))))
W_v = d2l.concat((d2l.zeros((4, 5)), d2l.eye(5)))
Q, K, V = X @ W_q, X @ W_k, X @ W_v  # (5, 2), (5, 2), (5, 5)
```

gave について見ると、クエリ $(1, 1)$ はドット積 2（各名詞のキー $(1, 1)$ との間）および 0（**gave** と「**the**」との間）をとる。$\sqrt{2}$ で割ると、スケール化されたスコアは各名詞で 1.41、それ以外で 0 となり、ソフトマックスによって各名詞の重みは $e^{1.41}/(3e^{1.41} + 2) \approx 0.29$、**gave** と「**the**」の重みは 0.07 に変換される（Figure 10.4）。

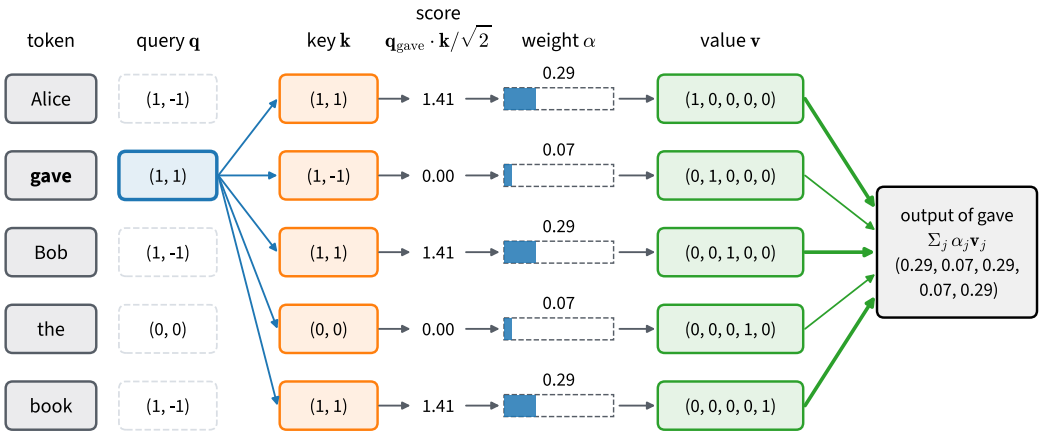
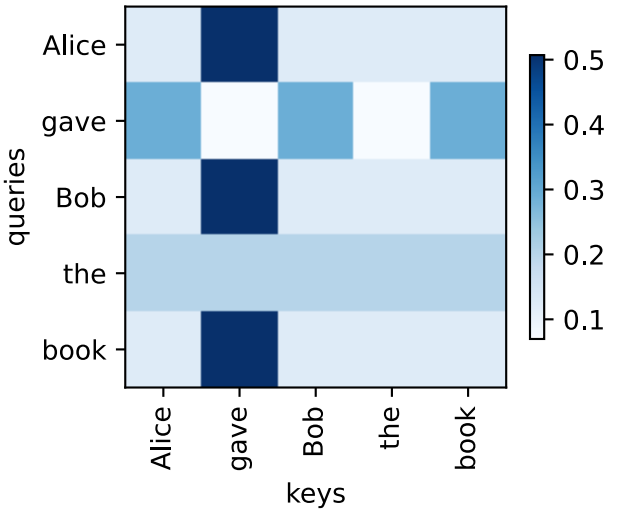


Figure 10.4: 上のコードで手動設定した写像（$d_h = 2$）を用いた、「Alice gave Bob the book」に対する自己アテンションの 1 つのヘッド。すべてのトークンがクエリ・キー・値を出力する。**gave** のクエリ $(1, 1)$ はすべてのキーに対して $\mathbf{q} \cdot \mathbf{k}/\sqrt{2}$ でスコア付けされ、ソフトマックスがスコア 1.41 と 0 を重み 0.29 と 0.07 に変換し、**gave** の出力は 5 つの単語を表す値の重み付き和となる。

```
attention = d2l.DotProductAttention()
Y = attention(Q[None], K[None], V[None])  # A minibatch of one sentence
weights = attention.attention_weights[0]
scores = Q[1] @ K.T / math.sqrt(Q.shape[-1])  # Scaled by sqrt(d_h)
print('scores of gave: ', np.asarray(scores).round(2))
print('weights of gave:', np.asarray(weights[1]).round(2))
print('weights of the: ', np.asarray(weights[3]).round(2))
print('output of gave: ', np.asarray(Y[0, 1]).round(2))
attention(Q[None], K[None], V[None], d2l.causal_mask(5))
```

```
print('causal weights of gave:',
      np.asarray(attention.attention_weights[0, 1]).round(2))
d2l.show_heatmaps(weights[None, None], xlabel='keys', ylabel='queries',
                  figsize=(3.2, 3.2), cmap='Blues',
                  xticklabels=tokens, yticklabels=tokens)
```

```
scores of gave:  [1.41 0.   1.41 0.   1.41]
weights of gave: [0.29 0.07 0.29 0.07 0.29]
weights of the:  [0.2 0.2 0.2 0.2 0.2]
output of gave:  [0.29 0.07 0.29 0.07 0.29]
causal weights of gave: [0.8 0.2 0.  0.  0. ]
```



逆に、各名詞のクエリ $(1, -1)$ は **gave** に対してスコア 1.41（重み 0.51）、それ以外に対して 0 を与え、一方で「**the**」はゼロクエリを持つため一様にアテンションを向ける（5 語すべてに 0.20）。因果マスク（causal mask）を適用すると、**gave** の参照先は **Alice** と自身のみに制限され、それらのスコア $(1.41, 0)$ は再正規化されて $(0.80, 0.20)$ になる。

10.3.1.2 クエリとキーに対する個別の写像

なぜ $\mathbf{W}_q$ と $\mathbf{W}_k$ に別々の行列を使うのだろうか。仮に単一の射影 $\mathbf{u}_i = \mathbf{x}_i\mathbf{W}$ がクエリとキーの両方を兼ねるとすると、スコア行列 $s_{ij} = \mathbf{u}_i^\top \mathbf{u}_j / \sqrt{d_h}$ は対称かつ半正定値（positive semidefinite）となり、コーシー・シュワルツの不等式により $s_{ij} \le \sqrt{s_{ii}\, s_{jj}} \le \max(s_{ii}, s_{jj})$ を満たす。すると、2 つのことが不可能になる。まず、$s_{ij} = s_{ji}$ であるため、動詞が名詞を求める強さを、名詞が動詞を求める強さより大きくすることはできない。また、2 つのトークンが互いに相手を自分自身より高くスコア付けすることも決してできない。動詞が自分自身より名詞を選好できるのは名詞のノルムがより大きい場合に限られ、そのとき名詞は動詞よりも自分自身を選好してしまうからである。クエリ・キー・値の射影を分けることで、**誰が探すか**、**誰が見つけられるか**、そして**どの内容を受け渡すか**が切り離される。

10.3.2 単一ヘッドの限界

先ほどの 5 語の文では、**gave** は **Alice**、**Bob**、**book** を 1 つの重み付き平均へと混合していた。クエリあたり 1 つのアテンション分布しか持たないことによるボトルネックを定量化するため、**両方コピータスク**（copy-both task）を考えよう。2 つのソース位置が、位置のみに依存する固定キーとともに独立な標準ガウス分布に従う値ベクトル $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^{d_v}$ を保持しており、単一のクエリが線形読み出し（linear readout）を介してそれらの連結 $\mathbf{t} = [\mathbf{v}_1; \mathbf{v}_2] \in \mathbb{R}^{2d_v}$ を出力しなければならないとする。

キーは位置のみに依存するため、単一ヘッドは固定のソフトマックス重み (α_1, α_2)（ただし $\alpha_1 + \alpha_2 = 1$）を割り当て、単一の混合ベクトル $\mathbf{m} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 \in \mathbb{R}^{d_v}$ を線形読み出しに渡す。

命題。 位置のみに依存する 2 つの固定キー、独立な標準ガウス分布に従う値 $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^{d_v}$、および線形読み出しを持つ 1 つのアテンションヘッドにおいて、両方コピータスクで達成可能な最小の相対二乗誤差は $1/2$ であり、任意のソフトマックス対 (α_1, α_2) に対してこの値をとる。

証明。 $\mathbf{m} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2$ が与えられたとき、$\mathbf{v}_1$ の最小平均二乗誤差推定量は条件付き期待値（conditional expectation）$\mathbb{E}[\mathbf{v}_1 \mid \mathbf{m}] = \frac{\alpha_1}{\alpha_1^2 + \alpha_2^2}\,\mathbf{m}$ であり、これは $\mathbf{m}$ に関して線形であって、座標あたりの残差分散は $1 - \frac{\alpha_1^2}{\alpha_1^2+\alpha_2^2} = \frac{\alpha_2^2}{\alpha_1^2+\alpha_2^2}$ となる。対称性より、$\mathbf{v}_2$ の推定でも座標あたり $\frac{\alpha_1^2}{\alpha_1^2+\alpha_2^2}$ の残差分散が残る。これらの和は座標あたり 1 であり、ターゲットの全分散である座標あたり 2 に対する比をとると、相対二乗誤差は $1/2$ となる（(α_1, α_2) によらない）。■

両方コピータスクを解くには 2 つの要素が必要である。（1）キーに対する少なくとも 2 つの線形独立な**重み付け**と、（2）取得した全 $2d_v$ 個の座標を保持するのに十分な**出力次元**（容量）である（演習問題 5）。以下では、単一ヘッドに対する二乗平均平方根相対誤差の下界 $\sqrt{1/2} \approx 0.707$ を検証するとともに、異なる重み付け（たとえば $(0.88, 0.12)$ と $(0.12, 0.88)$ のようなソフトな重み付けであっても）を持つ 2 つのヘッドがあれば、線形読み出しによって 2×2 の混合行列を浮動小数点精度まで逆変換できることを示す（Figure 10.5）。

```
N, d_v = 10000, 8
V1, V2 = torch.randn(N, d_v), torch.randn(N, d_v)
target = torch.cat([V1, V2], dim=1)

def readout_error(M, target):
    """Best linear readout of target from M, relative error."""
    W = torch.linalg.lstsq(M, target).solution
    return ((M @ W - target).norm() / target.norm()).item()

M = 0.7 * V1 + 0.3 * V2
print(f'one head, weights (0.70, 0.30): relative error '
      f'{readout_error(M, target):.3f}; sqrt(1/2) = {0.5 ** 0.5:.3f}')
for a in (1.0, 0.88):
    M2 = torch.cat([a * V1 + (1 - a) * V2, (1 - a) * V1 + a * V2], dim=1)
    print(f'two heads, weights ({a:.2f}, {1 - a:.2f}) and '
          f'({1 - a:.2f}, {a:.2f}): '
          f'relative error {readout_error(M2, target):.1e}')
```

```
one head, weights (0.70, 0.30): relative error 0.709; sqrt(1/2) = 0.707
two heads, weights (1.00, 0.00) and (0.00, 1.00): relative error 1.5e-06
two heads, weights (0.88, 0.12) and (0.12, 0.88): relative error 7.0e-07
```

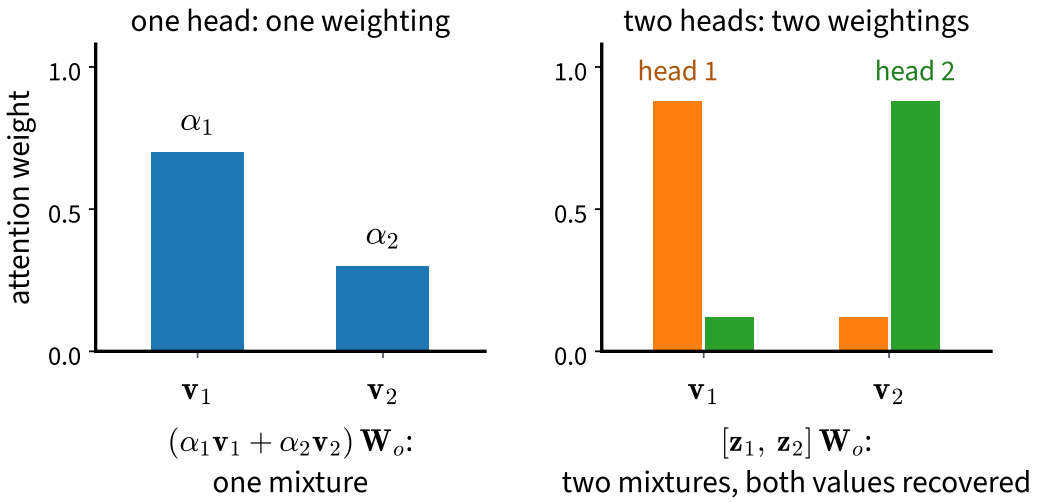


Figure 10.5: クエリあたり 1 つの重み付けと 2 つの重み付けの対比。単一ヘッドは 1 つのソフトマックス重み付けで両方コピータスクを処理するため、その出力では要求された 2 つの値が混ざり合う。異なる重み付けを持つ 2 つのヘッドは 2 つの混合ベクトルを供給し、そこから線形読み出しが両方の値を復元する。

10.3.3 マルチヘッドアテンション

10.3.3.1 射影されたヘッド

マルチヘッドアテンションでは (Vaswani et al. 2017)、H 個の独立なアテンションヘッドを並列に実行する。各ヘッドは独自のクエリ・キー・値の射影 $\mathbf{W}_i^{(q)}, \mathbf{W}_i^{(k)}, \mathbf{W}_i^{(v)} \in \mathbb{R}^{d\times d_h}$ を持ち、次のように計算される。

$$\mathbf{Z}_i = \operatorname{softmax}\left(\frac{(\mathbf{X}\mathbf{W}_i^{(q)})(\mathbf{X}\mathbf{W}_i^{(k)})^\top}{\sqrt{d_h}}\right)\mathbf{X}\mathbf{W}_i^{(v)} \in \mathbb{R}^{n\times d_h}, \qquad i = 1, \dots, H, \tag{10.16}$$

続いて、それらの出力を連結したうえで、最終的な線形射影 $\mathbf{W}_o \in \mathbb{R}^{Hd_h\times d}$ を適用する（Figure 10.6）。

$$[\mathbf{Z}_1, \dots, \mathbf{Z}_H]\,\mathbf{W}_o \in \mathbb{R}^{n\times d}. \tag{10.17}$$

10.3.3.2 パラメータ数と FLOP 数

ヘッド幅を $d_h = d/H$ に設定すると、全ヘッドにわたる合計次元は $Hd_h = d$ に保たれる。横に並べると、H 個のクエリ射影は単一の $d \times d$ 行列をなし、キー・値・出力の各射影も同様である。1 回の積和演算を 2 FLOPs と数えると、n トークンに対するマルチヘッド自己アテンション層が必要とする計算量とパラメータ数は次のようになる。

$$\underbrace{8nd^2}_{\text{projections}} + \underbrace{4n^2d}_{\text{scores and mixing}} \ \text{FLOPs}, \qquad 4d^2 \ \text{parameters}. \tag{10.18}$$

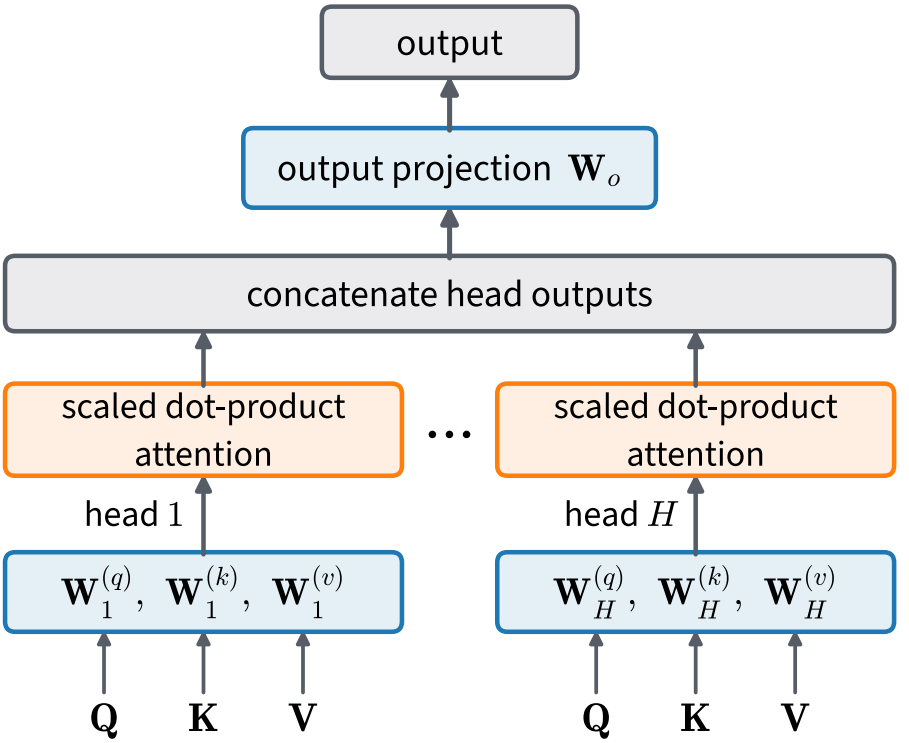


Figure 10.6: マルチヘッドアテンション。各ヘッドが同じクエリ・キー・値を読み込み、それぞれの部分空間へ射影してアテンションを計算する。ヘッドの出力は連結され、1 つの出力射影によって再結合される。

GPT-2 small（$d = 768$、$n = 1{,}024$）の 1 層では、射影に 4.8 GFLOPs、スコア計算と値の混合に 3.2 GFLOPs を要し、パラメータ数は 2.4×10^6 個となる。注目すべきことに、パラメータ数も主要項の FLOP 数もヘッド数 H に依存しない。次元 d を H 個のヘッドに分割すると、各ヘッドの内積コストは $1/H$ に縮む一方で、それを H 個実行するからである（$O(Hn^2)$ のソフトマックスのオーバーヘッドのみが H に比例する）。逆に、d_h をいくらでも小さくできるわけではない。$d_h = 1$ ではスカラースコア qk_j が k_j に関して単調になるため、クエリは最大または最小の端にあるキーにしか集中できなくなる（演習問題 1）。実際には $d_h \in \{64, 128\}$ が標準的である。

10.3.3.3 畳み込み・再帰との比較

Figure 10.7 と Table 10.1 では、1 次元畳み込み（第 6 章）、再帰（8.4 節）、自己アテンションを、n 個のトークン（幅 d）に対する系列対系列の層として比較する。

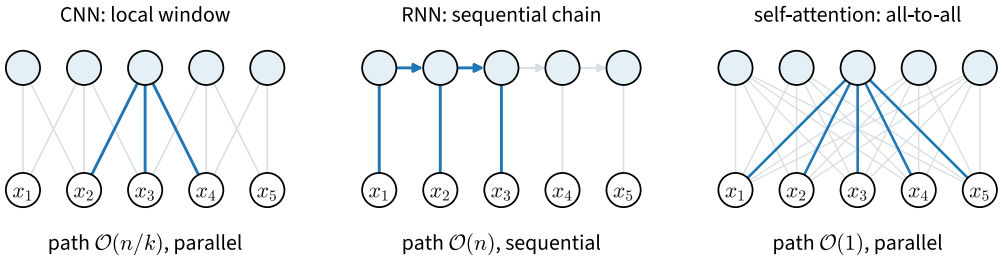


Figure 10.7: 5 トークンに対する系列層の結合構造。強調された辺は、1 層で入力が 3 番目の出力にどう到達するかを示す。カーネルサイズ $k = 3$ の畳み込みは近傍を参照し、再帰は連鎖に沿って 1 ステップずつ情報を伝え、自己アテンションはすべての位置を直接結ぶ。各パネルの下には、離れたトークン間の経路長と、出力が並列に計算されるかどうかを示す。

Table 10.1: n 個の d 次元トークンを n 個の d 次元トークンへ写す層あたりのコスト。

層の種類	計算量	逐次演算数	最大経路長
畳み込み（カーネル k）	$O(knd^2)$	$O(1)$	$O(n/k)$
再帰	$O(nd^2)$	$O(n)$	$O(n)$
自己アテンション	$O(nd^2 + n^2 d)$	$O(1)$	$O(1)$

自己アテンションは $O(1)$ の逐次ステップ数（訓練における完全な並列性）と任意のトークン対間の最大経路長 $O(1)$ を兼ね備えており、$O(n)$ ステップにわたるリカレントな勾配の連鎖を解消する（8.5 節）。その代償はスコア計算と混合にかかる二次の項 $4n^2 d$ であり、射影のコスト $8nd^2$ を $n > 2d$（GPT-2 small では $n > 1{,}536$）の領域で上回る。

10.3.3.4 実装

`MultiHeadAttention` では、単一の線形層が全 H 個のヘッドの射影を一度に計算し、`split_heads` が形状 `(batch, length, H * d_h)` を `(batch, H, length, d_h)` に変形することで、`DotProductAttention` が全ヘッドを 1 回のバッチ演算として実行できるようにする（Figure 10.8）。現代の Transformer の慣例に従い、射影ではデフォルトで加法バイアスを省略する（`bias=False`）。また、引数 `num_kv_heads` によってキー・値ヘッドの共有をサポートする（10.3.3.6 節）。

```python
class MultiHeadAttention(nn.Module):
    """Multi-head attention with heads as an explicit axis; num_kv_heads <
    num_heads shares each key-value head across a group of query heads."""
    def __init__(self, num_hiddens, num_heads, dropout=0.0, bias=False,
                 num_kv_heads=None):
        super().__init__()
        assert num_hiddens % num_heads == 0, 'heads must divide num_hiddens'
        self.num_heads = num_heads
        self.num_kv_heads = num_kv_heads or num_heads
        assert num_heads % self.num_kv_heads == 0
        kv_hiddens = self.num_kv_heads * (num_hiddens // num_heads)
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(num_hiddens, num_hiddens, bias=bias)
        self.W_k = nn.Linear(num_hiddens, kv_hiddens, bias=bias)
        self.W_v = nn.Linear(num_hiddens, kv_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, mask=None):
        # queries: (batch, no. of queries, num_hiddens); keys and values:
        # (batch, no. of key-value pairs, num_hiddens). mask: boolean,
        # broadcastable to (batch, no. of queries, no. of keys), shared by
        # all heads, or to (batch, num_heads, no. of queries, no. of keys)
        q = self.split_heads(self.W_q(queries), self.num_heads)
        k = self.split_heads(self.W_k(keys), self.num_kv_heads)
        v = self.split_heads(self.W_v(values), self.num_kv_heads)
        group = self.num_heads // self.num_kv_heads
        k, v = k.repeat_interleave(group, 1), v.repeat_interleave(group, 1)
        if mask is not None and mask.dim() == 3:
            mask = mask[:, None]           # The same mask for every head
        # Shape of Y: (batch, num_heads, no. of queries, head width)
        Y = self.attention(q, k, v, mask)
        return self.W_o(self.merge_heads(Y))

    @property
    def attention_weights(self):
        """Weights of the last call: (batch, num_heads, queries, keys)."""
        return self.attention.attention_weights
```

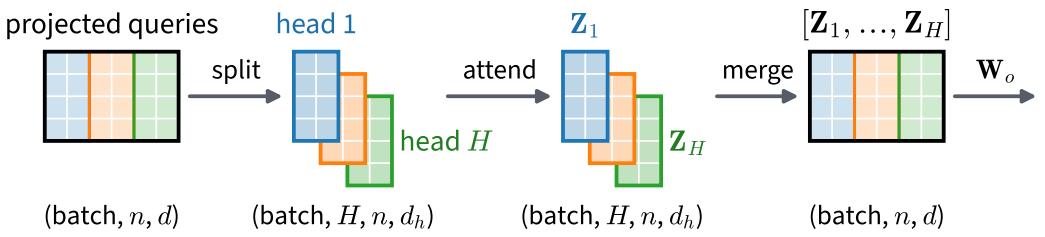


Figure 10.8: 1 つの射影のスライスとしての各ヘッド。n トークンの射影されたクエリを H 個の列スライス（幅 $d_h = d/H$）に分割し、独立したヘッド軸とする。各ヘッドが個別にアテンションを計算した後、出力射影 $\mathbf{W}_o$ の前でヘッド出力を再び横に並べる。キーと値も同様に分割される。

形状変換と転置を担う 2 つの補助メソッド `split_heads` と `merge_heads`（Figure 10.8）は、アテンションの前に最後の次元 Hd_h を明示的なヘッド軸を持つ `(batch, num_heads, length, width)` へと折り畳み、アテンションの後に H 個のヘッド出力を再び連結する。

```
@d2l.add_to_class(MultiHeadAttention)
def split_heads(self, X, num_heads):
    """(batch, length, num_heads * width) -> (batch, num_heads, length,
    width)."""
    return X.reshape(X.shape[0], X.shape[1], num_heads, -1).transpose(1, 2)

@d2l.add_to_class(MultiHeadAttention)
def merge_heads(self, X):
    """Reverse split_heads: (batch, num_heads, length, width) -> (batch,
    length, num_heads * width)."""
    return X.transpose(1, 2).reshape(X.shape[0], X.shape[2], -1)
```

10.3.3.5 両方コピータスクにおける学習されたヘッド

ここで、幅 $d = 16$ の `MultiHeadAttention` を両方コピータスク（$d_v = 8$ をゼロパディングして幅 16 とし、キーを $\mathbf{e}_1, \mathbf{e}_2$、クエリを $\mathbf{e}_3$ とする）で検証しよう。

```
def copy_both_data(n, d=16, d_v=8):
    """One query, two position-only keys, values of width d_v padded to d."""
    V = torch.randn(n, 2, d_v)
    keys = torch.eye(d)[:2].expand(n, 2, d)    # e_1, e_2 mark the positions
    query = torch.eye(d)[2].expand(n, 1, d)    # e_3 asks for both values
    values = F.pad(V, (0, d - d_v))
    return query, keys, values, V.reshape(n, 1, 2 * d_v)

def train_copy_both(num_heads, num_kv_heads=None, num_steps=2000):
    model = MultiHeadAttention(16, num_heads, num_kv_heads=num_kv_heads)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
    for _ in range(num_steps):
        query, keys, values, target = copy_both_data(256)
        loss = ((model(query, keys, values) - target) ** 2).mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model

def evaluate_copy_both(model, n=10000):
    """Relative error on fresh examples; each head's weight on v_1."""
    query, keys, values, target = copy_both_data(n)
    with torch.inference_mode():
        err = (model(query, keys, values) - target).norm() / target.norm()
    return err.item(), model.attention_weights[0, :, 0, 0].tolist()
```

各ヘッド数 $H \in \{1, 2, 4\}$（いずれも合計幅 $d = 16$、パラメータ数 1,024 で共通）について 2 つのシードで訓練し、相対再構成誤差および各ヘッドのアテンション重み α_1（第 1 の値ベクトル $\mathbf{v}_1$ に対する重み）を比較する。

```
print('H  run  parameters  error    weight on v_1, per head')
for num_heads in (1, 2, 4):
    for run in (1, 2):
        model = train_copy_both(num_heads)
        err, alpha_1 = evaluate_copy_both(model)
        print(f'{num_heads}  {run:3d}  {d2l.num_params(model):10d}  '
              f'{err:<#8.2g} ' + ' '.join(f'{a:.2f}' for a in alpha_1))
```

```
H  run  parameters  error    weight on v_1, per head
1    1        1024  0.71     0.51
1    2        1024  0.71     0.52
2    1        1024  1.1e-07  0.98 0.12
2    2        1024  1.1e-07  0.93 0.03
4    1        1024  1.1e-07  0.96 0.94 0.16 0.12
4    2        1024  1.2e-07  0.04 0.26 0.88 0.95
```

いずれの構成もパラメータ数は $4 \times 16^2 = 1{,}024$ である。先の命題が予測したとおり、$H = 1$ は下界である $\sqrt{1/2} \approx 0.71$ で頭打ちになるのに対し、$H \in \{2, 4\}$ はヘッドごとに異なる重み付けを学習し、誤差を $\sim 10^{-7}$ まで縮小する。

10.3.3.6 キーと値の共有

自己回帰生成（autoregressive generation）の際、過去のキーと値はメモリにキャッシュされるため、新しい各トークンは過去の全位置の射影を再計算することなく参照できる（11.3 節）。**グループ化クエリアテンション**（GQA）では (Ainslie et al. 2023)、このキャッシュを H/H_{kv} 分の 1 に削減するため、各キー・値ヘッドを H/H_{kv} 個のクエリヘッドからなるグループで共有する。特に $H_{\text{kv}} = 1$ とすると、**マルチクエリアテンション**（MQA）が得られる (Shazeer 2019)。

両方コピータスクでは、共有された値ヘッドが張る空間は値あたり高々 $H_{\text{kv}} d_h$ 座標であるため、d_v 次元の値を復元するには $H_{\text{kv}} d_h \geq d_v$ が必要となる（演習問題 6）。$d = 16$、$d_v = 8$ のとき、構成 $(H, H_{\text{kv}}) = (2, 1)$ と $(4, 2)$ は $H_{\text{kv}} d_h = 8$ となって fp32 精度でタスクを解く一方、$(4, 1)$ は $H_{\text{kv}} d_h = 4$ しかなく、4 つのクエリヘッドにわたって異なる重み付けを学習しても 0.71 のボトルネックに逆戻りする。

```
print('H  H_kv  run  parameters  error    weight on v_1, per head')
for num_heads, num_kv_heads in ((2, 1), (4, 2), (4, 1)):
    for run in (1, 2):
        model = train_copy_both(num_heads, num_kv_heads)
        err, alpha_1 = evaluate_copy_both(model)
        print(f'{num_heads}  {num_kv_heads:4d}  {run:3d}  '
              f'{d2l.num_params(model):10d}  {err:<#8.2g} '
              + ' '.join(f'{a:.2f}' for a in alpha_1))
```

```
H  H_kv  run  parameters  error    weight on v_1, per head
2     1    1         768  1.1e-07  0.93 0.03
2     1    2         768  1.1e-07  0.03 0.96
4     2    1         768  1.1e-07  0.08 0.95 0.98 0.08
4     2    2         768  1.1e-07  0.03 0.88 0.02 0.88
4     1    1         640  0.71     0.97 0.04 0.13 0.89
4     1    2         640  0.71     0.03 0.04 0.72 0.98
```

10.3.4 クロスアテンション

自己アテンションが同じ系列 **X** をクエリ・キー・値として渡す（`attention(X, X, X)`）のに対し、**クロスアテンション**はターゲット系列 **A** からクエリを、ソース系列 **B** からキーと値を取り出し（`attention(A, B, B)`）、**A** のトークンごとに 1 行の出力を生成する。エンコーダ・デコーダ型の Transformer は翻訳時に符号化されたソース文を読み取るためにクロスアテンションを用い、マルチモーダルモデルはテキストトークンから視覚特徴量へクエリを送るためにこれを用いる (Alayrac et al. 2022)。

しかし、複数のヘッドを用いても、自己アテンションは依然として語順を識別できない。「**Alice gave Bob the book**」において **Alice** と **Bob** を入れ替えても、**gave** における出力は変わらないのである。系列の順序を注入する方法については 10.4 節で扱う。

10.3.5 まとめ

自己アテンションは系列の各トークンをクエリ・キー・値へと射影し、$O(1)$ の逐次ステップ数と $O(1)$ の経路長、および $O(nd^2 + n^2 d)$ FLOPs の計算量ですべての位置対を結ぶ。単一ヘッドはクエリあたり値の凸結合を 1 つしか形成できないため、独立な 2 つの値を同時に取得するタスクでは、いかなる単一ヘッドも少なくとも $1/2$ の相対二乗誤差を被る。マルチヘッドアテンションは、H 個の部分空間ヘッド（幅 $d_h = d/H$）を並列に実行して出力を $\mathbf{W}_o$ で結合することでこのボトルネックを克服し、必要なパラメータ数は $4d^2$、計算量は $8nd^2 + 4n^2 d$ FLOPs といずれも H に依存しない。グループ化クエリアテンションとマルチクエリアテンションは、$H_{\text{kv}} < H$ 個のキー・値ヘッドをクエリグループ間で共有することで生成時のキャッシュを H/H_{kv} 分の 1 に縮小しつつ、共有された値の幅 $H_{\text{kv}} d_h$ が取得対象の特徴量を収容できる限り検索能力を維持する。

10.3.6 演習問題

1. [code] **幅の狭いヘッド**。$d_h = d/H$ とすると、Equation 10.18 の主要項は H に依存しないため、ヘッド数は $H = d$ まで増やしてもコストが増えないように見える。マスクなしでスコア $\mathbf{q}^\top \mathbf{k}_j / \sqrt{d_h}$ をキー $\mathbf{k}_1, \dots, \mathbf{k}_m \in \mathbb{R}^{d_h}$ に対して計算する 1 つのヘッドを考える。クエリを限りなく増大する係数でスケールしたときに重みがキー j（すなわち $\mathbf{k}_j$）へ集中するとき、そのクエリはそのキーに対して鋭い（sharp）と呼ぶことにする。
   1. $d_h = 1$ で互いに異なるスカラーのキーが与えられたとき、任意のクエリの重みが k_j に関して単調になることを示せ。これから、このようなヘッドのクエリが鋭くなりうるのは最大または最小のキーに対してのみであることを導け。
   2. $d_h = 2$ のとき、クエリからキーへのいかなる割り当てに対しても、n 個の各クエリがそれぞれ任意に選んだキーに対して鋭くなるようにキーとクエリを選べることを示せ。
   3. これら 2 つの結果を用いて、H が d に近づくにつれて層が表現できなくなるアテンションパターンを述べよ。
   4. ドロップアウト 0 の `MultiHeadAttention` について、$d = 256$、$n = 512$、ミニバッチサイズ 8 系列の自己アテンションにおける順伝播の実行時間を、$H \in \{1, 2, 4, 8, 16, 32, 64\}$ にわたって計測せよ。最初にウォームアップ呼び出しを 1 回行い、計測を止める前に結果の完了を待つこと。最も遅い構成と最も速い構成の実行時間の比を報告せよ。さらに、H とともに増大し Equation 10.18 では省略されているコストを特定し、それがこの傾向を説明できることを確認せよ。
2. [code] **スライスとしてのヘッド**。`MultiHeadAttention` はクエリ・キー・値のそれぞれに対して幅 d の射影を 1 回ずつ計算し、`split_heads` によって各ヘッドへ分割する。
   1. ヘッド i は `W_q`、`W_k`、`W_v` の出力特徴量のスライスを担当する。`W_o` のどの入力特徴量がそのヘッドの出力を受け取るかを求めよ。さらに、ヘッドについてループを回

し、スライスした射影を適用してヘッドごとに `d2l.DotProductAttention` を 1 回呼び出す順伝播を実装せよ。パディングマスク付きのランダムな入力に対し、これがドロップアウト 0 の `MultiHeadAttention` と一致することを検証し、さらに `num_kv_heads < num_heads` の場合へループを拡張せよ。

2. `split_heads` の本体を `X.reshape(X.shape[0], num_heads, X.shape[1], -1)` に置き換えよ（これは同じ形状のテンソルを返す）。このとき各「ヘッド」がどの数値を保持するかを説明し、ランダムな入力で出力が変化することを示したうえで、形状のチェックだけではこの誤りを検出できない理由を説明せよ。2 つのバージョンの出力が一致するような入力の長さと幅は存在するか。存在するならばそれはどのような場合か。

3. **ソースの置換**。クロスアテンションにおいて、クエリは $\mathbf{A} \in \mathbb{R}^{n\times d}$ から、キーと値は $\mathbf{B} \in \mathbb{R}^{m\times d}$ から得られ、どちらも位置情報を持たないとする。Π を $m \times m$ の置換行列とする。
   1. マスクを適用しないとき、キーと値の両方の引数で $\mathbf{B}$ を $\Pi\mathbf{B}$ に置き換えても `MultiHeadAttention` の出力は変化せず、キーの引数のみを置き換えると一般に出力が変化することを示せ。
   2. 次に、パディングマスクによって先頭 ℓ 行のみが $\mathbf{B}$ の有効な行として指定されているとする。マスクを固定したままにしたとき、出力を変化させない置換はどのようなものか。

4. [code] **ほぼ同一のヘッド**。●重み $(0.5+\epsilon, 0.5-\epsilon)$ と $(0.5-\epsilon, 0.5+\epsilon)$ を持つ 2 つのヘッドは 2 つの異なる混合ベクトルを供給するため、任意の $\epsilon > 0$ に対して線形読み出しはそれらを厳密に逆変換できる。
   1. 2×2 の混合行列を書き下し、その固有値と固有ベクトルを計算して、条件数を ϵ の関数として表せ。
   2. 標準偏差 $\sigma = 0.01$ の独立なガウスノイズを両方の混合ベクトルに加え、`readout_error` を用いて $\epsilon \in \{0.4, 0.1, 0.03, 0.01, 0.003, 0.001\}$ に対する誤差を測定せよ。両対数軸上で ϵ に対して誤差をプロットし、曲線が直線になる領域の傾きを読み取って、(a) の条件数を用いてその傾きを説明せよ。
   3. 初期化直後の層の各ヘッドは、両方コピータスクの 2 つの位置に $(0.5, 0.5)$ 付近の重みを割り当てる。`train_copy_both` を変更して 20 ステップごとに相対誤差と各ヘッドの重みを記録し、$H = 2$ について両方をプロットせよ。最初の数百ステップにおける誤差曲線の形状を、混合行列の条件数（悪条件性）を用いて説明せよ。さらに $H = 8$ でも同様の実験を行い、最終的な各ヘッドを、ほぼ one-hot なもの、ソフトだが他と異なるもの、別のヘッドとほぼ重複しているものに分類せよ。

5. **下界が数えているもの**。●10.3.2 節の命題ではソフトマックス重みを固定したが、その誤差下界はより弱い事実に依拠している。すなわち、読み出しは単一のベクトル $\mathbf{m} \in \mathbb{R}^{d_v}$ を受け取り、ターゲット $\mathbf{t} \in \mathbb{R}^{2d_v}$ を出力しなければならないという点である。
   1. $\mathbf{m} \in \mathbb{R}^{d_v}$ を $\mathbf{v}_1$ と $\mathbf{v}_2$ の任意の関数（重みが値に依存する混合ベクトルを含む）とし、$\mathbf{W}_o \in \mathbb{R}^{d_v\times 2d_v}$ を任意の行列とする。次の不等式が成り立つことを示せ。

$$\mathbb{E}\,\|\mathbf{m}\mathbf{W}_o - \mathbf{t}\|^2 \geq \tfrac{1}{2}\,\mathbb{E}\,\|\mathbf{t}\|^2.$$

   これから、読み出しが線形であり受け取る数値が d_v 個である限り、内容依存のキー、層の追加、あるいは残差接続を導入してもこの下界は変わらないことを導け。
   2. 幅 d の層（幅 $d_h = d/2$ のヘッドを 2 つ持つ）が、値 $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^{d/2}$（ゼロパディングにより幅 d に拡張）を受け取るとする。このタスクを厳密に解くような各ヘッドのアテンション重み、値射影 $\mathbf{W}_i^{(v)}$、および出力射影 $\mathbf{W}_o$ を与えよ。また、その重みを 10.3.3.5 節で出力された重みと比較せよ。
   3. この命題を k 個のソース位置（独立な標準ガウス分布に従う値 $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{R}^{d_v}$ とターゲット $[\mathbf{v}_1; \ldots; \mathbf{v}_k]$ を持つ）へ一般化せよ。H 個の各ヘッドが値に対して固定の重みを適用し、ヘッド i の重みが第 i 行をなす行列を $\mathbf{A} \in \mathbb{R}^{H\times k}$ とし、線形読み出しが全 H 個の混合ベクトルを参照するとする。達成可能な最良の相対二乗誤差が次の式

で与えられることを示せ。

$$1 - \frac{\operatorname{rank}(\mathbf{A})}{k}, \tag{10.19}$$

また、すべての成分が正であるソフトマックス重みによって $\operatorname{rank}(\mathbf{A}) = \min(H, k)$ を達成できることも示せ。

6. [code] **両方コピータスクにおけるキーと値の共有**。●幅 d の層が、H 個のクエリヘッド（幅 $d_h = d/H$）と $H_{\text{kv}} < H$ 個のキー・値ヘッドを持つとする。
   1. 演習問題 5 の次元カウントを用いて、この層がタスクを解けるのは $H_{\text{kv}} d_h \geq d_v$ の場合に限られること、またこの条件で十分であることを示せ。
   2. 2 つのクエリヘッドが 1 つのキー・値ヘッドを共有する `train_copy_both(2, 1)` で層を訓練せよ。2 つのヘッド間で異なるパラメータと共有されているパラメータを特定し、両方コピータスクの性質から、なぜその箇所に違いが生じなければならないのかを説明せよ。

10.4 位置情報

“dog bites man” と **“man bites dog”** という 2 つの文は、同じ 3 つの単語からなっている。これらを 10.3 節のマスクなし自己アテンション層に通すと、順序が入れ替わっただけのまったく同じ出力ベクトルが得られる。アテンション重みがトークンの内容のみに依存するためである。語順を区別するには、表現またはスコア計算規則に位置情報を注入しなければならない。

位置情報の導入手法は大きく 3 つの系統に分けられる。各トークン埋め込みに位置ベクトル $\mathbf{p}_i$ を加算する手法、位置に応じた角度でクエリとキーを回転させる手法（**回転位置埋め込み**（rotary position embeddings、RoPE））、そしてアテンションスコアから距離ペナルティを減算する手法（**ALiBi**）である。これら 3 系統はいずれも訓練コンテキスト長の内側では位置を区別できるが、より長い系列への外挿（extrapolation）を求められると大きく異なる振る舞いを示す。本節ではまず自己アテンションの置換対称性（permutation symmetry）を定式化し、続いて長さ 128 で訓練して最大長 512 まで評価する 2 層のアテンションのみのバイト言語モデル（`TinyCharLM`）上で 5 つの位置表現手法を構築・比較する。

10.4.1 置換同変性

$\mathbf{X} \in \mathbb{R}^{n \times d}$ を n 個のトークンベクトルを並べた行列とし、$f(\mathbf{X}) = \operatorname{softmax}(\mathbf{S})\,\mathbf{X}\mathbf{W}_v$（ただし $\mathbf{S} = \mathbf{X}\mathbf{W}_q(\mathbf{X}\mathbf{W}_k)^\top/\sqrt{d_h}$）をドロップアウトを無効化したマスクなし自己アテンション（10.3 節）とする。n 個のトークンの並び替えは、$\mathbf{X}$ に左から $n \times n$ の置換行列 Π を掛けることに対応する。

命題（置換同変性）。任意の置換行列 Π に対して、$f(\Pi\mathbf{X}) = \Pi f(\mathbf{X})$ が成り立つ。

証明。線形射影は行ごとに作用するため、$(\Pi\mathbf{X})\mathbf{W} = \Pi(\mathbf{X}\mathbf{W})$ が $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ のそれぞれについて成り立つ。したがって、スコア行列は $\Pi\mathbf{S}\Pi^\top$ となる。行ごとのソフトマックスは行および列の置換と可換であるため、$\operatorname{softmax}(\Pi\mathbf{S}\Pi^\top) = \Pi \operatorname{softmax}(\mathbf{S})\,\Pi^\top$ が得られる。これに置換後の値 $\Pi\mathbf{X}\mathbf{W}_v$ を掛け、$\Pi^\top\Pi = \mathbf{I}$ を用いると $\Pi f(\mathbf{X})$ となる。複数ヘッドを連結して行ごとに $\mathbf{W}_o$ を適用しても同変性は保たれる。■

4 ヘッドの `MultiHeadAttention` 層上で、置換同変性（permutation equivariance）を数値的に確認しよう。

```
attention = d2l.MultiHeadAttention(num_hiddens=64, num_heads=4)
X = torch.randn(1, 8, 64)
perm = torch.randperm(8)
```

```
Y = attention(X, X, X)
Y_perm = attention(X[:, perm], X[:, perm], X[:, perm])
print(f'max |Y[perm] - Y_perm|: {(Y[:, perm] - Y_perm).abs().max():.2e}')
```

```
max |Y[perm] - Y_perm|: 1.08e-07
```

同変な層やトークンごとの多層パーセプトロン（MLP）を積み重ねても同変性は保たれる。ただし、**因果マスク**（causal mask）を導入すると位置 i が $i+1$ 個の先行要素を参照するようになるため、置換対称性が破れる。後述する **NoPE** はこの非対称性を利用している。

10.4.2 絶対位置埋め込み

オリジナルの Transformer (Vaswani et al. 2017) は、位置ベクトル $\mathbf{p}_i \in \mathbb{R}^d$（第 i 行にあたる $\mathbf{P}$ の行）を各トークン埋め込み $\mathbf{x}_i$ に加算することで、入力段階で置換対称性を破る。トークンをシャッフルすると $\mathbf{X}+\mathbf{P}$ は $\Pi\mathbf{X}+\mathbf{P} \neq \Pi(\mathbf{X}+\mathbf{P})$ へと写されるからである。

10.4.2.1 学習による位置表現

BERT (Devlin et al. 2018) や GPT-2 (Radford et al. 2019) などのモデルでは、$\mathbf{P} \in \mathbb{R}^{L_{\max} \times d}$ を、固定の最大長 $L_{\max}$ まで位置ごとに 1 つのパラメータ行を持つ訓練可能な埋め込みテーブル（embedding table）として扱う。訓練コンテキスト長を超える行は一度も勾配を受け取らないため、学習型のテーブルは未知の位置へ外挿できない。

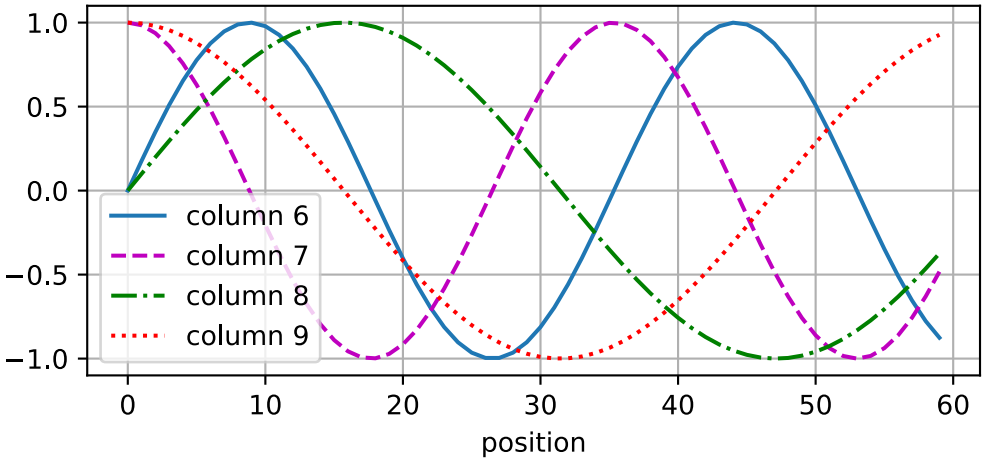
10.4.2.2 正弦波エンコーディング

学習パラメータを用いずに任意の長さに対する位置ベクトルを定義するため、Vaswani et al. (2017) は位置 i に対応する $\mathbf{P}$ の行に、等比的に配置された $d/2$ 個の周波数 $\omega_m = 10000^{-2m/d}$ における正弦と余弦を格納する。

$$p_{i,2m} = \sin\left(\frac{i}{10000^{2m/d}}\right), \qquad p_{i,2m+1} = \cos\left(\frac{i}{10000^{2m/d}}\right). \tag{10.20}$$

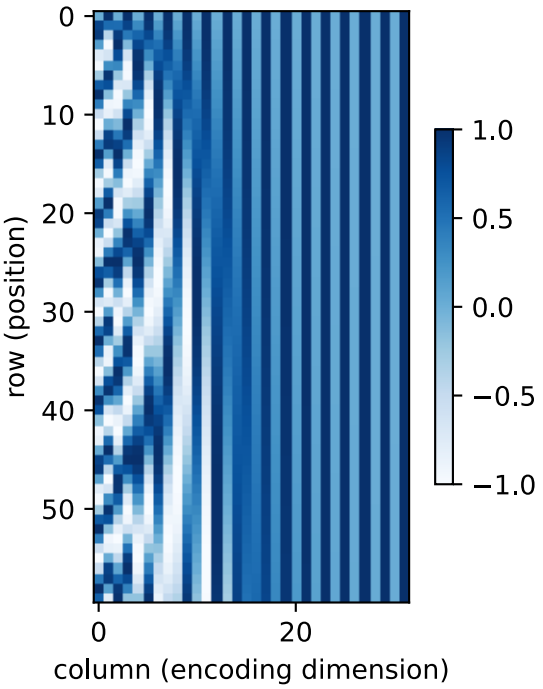
ω_m がトークンあたり 1 ラジアン（$m=0$）から 10^{-4}（$m=d/2-1$）へと減少するにつれ、各列ペアは二進カウンタの連続版として機能する。すなわち、小さいインデックスの列は急速に振動して隣接する位置を区別し、大きいインデックスの列は広い領域にわたって緩やかに変化する。

```
def sinusoidal_encoding(max_len, num_hiddens):
    """The (max_len, num_hiddens) table of sinusoidal position codes."""
    theta = torch.arange(max_len)[:, None] / 10000 ** (
        torch.arange(0, num_hiddens, 2) / num_hiddens)
    return torch.stack([torch.sin(theta), torch.cos(theta)],
                       -1).reshape(max_len, num_hiddens)

P = sinusoidal_encoding(60, 32)
d2l.plot(torch.arange(60), P[:, 6:10].T, xlabel='position',
         figsize=(6, 2.5), legend=[f'column {d}' for d in range(6, 10)])
```



60×32 のエンコーディング行列 P 全体をヒートマップとして表示すると、整数カウンタの各ビットと同様に、左側の急速に振動する高周波ペアから右側の緩やかに変化する低周波ペアへと移り変わる様子がわかる。

```
d2l.show_heatmaps(P[None, None], xlabel='column (encoding dimension)',
                  ylabel='row (position)', figsize=(3.5, 4), cmap='Blues')
```



10.4.2.3 回転としての正弦波エンコーディング

三角関数の加法定理により、位置 i をオフセット δ だけシフトすると、2×2 回転（δ のみに依存）が各周波数ペア $(2m, 2m+1)$ に適用される。

$$\begin{bmatrix} \cos(\delta\omega_m) & \sin(\delta\omega_m) \\ -\sin(\delta\omega_m) & \cos(\delta\omega_m) \end{bmatrix} \begin{bmatrix} p_{i,2m} \\ p_{i,2m+1} \end{bmatrix} = \begin{bmatrix} \sin\left((i+\delta)\omega_m\right) \\ \cos\left((i+\delta)\omega_m\right) \end{bmatrix} = \begin{bmatrix} p_{i+\delta,2m} \\ p_{i+\delta,2m+1} \end{bmatrix}. \tag{10.21}$$

しかし、$\mathbf{p}_i$ は $\mathbf{x}_i$ に**加算**されるため、$\mathbf{A} = \mathbf{W}_q \mathbf{W}_k^\top / \sqrt{d_h}$ を用いた双線形のアテンションスコアは 4 つの項に展開される。

$$\begin{aligned} (\mathbf{x}_i + \mathbf{p}_i)^\top \mathbf{A} \, (\mathbf{x}_j + \mathbf{p}_j) = \; & \underbrace{\mathbf{x}_i^\top \mathbf{A} \, \mathbf{x}_j}_{\text{content–content}} + \underbrace{\mathbf{x}_i^\top \mathbf{A} \, \mathbf{p}_j}_{\text{content–position}} \\ & + \underbrace{\mathbf{p}_i^\top \mathbf{A} \, \mathbf{x}_j}_{\text{position–content}} + \underbrace{\mathbf{p}_i^\top \mathbf{A} \, \mathbf{p}_j}_{\text{position–position}} . \end{aligned} \tag{10.22}$$

2 つの交差項は内容と**絶対**位置（absolute position）を結合しており（i と j が一緒にシフトすると値が変化する）、さらに $\mathbf{p}_i^\top \mathbf{A} \, \mathbf{p}_j$ でさえも純粋に $j - i$ のみに依存するのは、$\mathbf{A}$ が 2×2 のスケール付き回転ブロックからなるブロック対角行列である場合に限られる（演習問題 2）。

10.4.3 回転位置埋め込み

回転位置埋め込み（RoPE）(Su et al. 2024) は、すべてのスコアが相対オフセット $j - i$ のみに依存するよう、クエリとキーに対して位置変換を**乗法的**に適用する。

$$(\mathbf{R}_i \mathbf{q})^\top (\mathbf{R}_j \mathbf{k}) = \mathbf{q}^\top \mathbf{R}_i^\top \mathbf{R}_j \mathbf{k} = \mathbf{q}^\top \mathbf{R}_{j-i} \mathbf{k} \quad \text{for all } i, j. \tag{10.23}$$

$\mathbf{R}_i^\top \mathbf{R}_j = \mathbf{R}_{j-i}$ および $\mathbf{R}_0$ がフルランクであることを要請すると、次が導かれる。(1) $\mathbf{R}_0 = \mathbf{I}$（$\mathbf{R}_0^\top \mathbf{R}_0 = \mathbf{R}_0$ より）。(2) $\mathbf{R}_i^\top \mathbf{R}_i = \mathbf{I}$ であり、したがってすべての $\mathbf{R}_i$ は直交かつノルム保存である。(3) $\mathbf{R}_{i+j} = \mathbf{R}_i \mathbf{R}_j$ であり、したがって $\mathbf{R}_i = \mathbf{R}_1^i$ となる。適切な正規直交基底（orthonormal basis、$\mathbf{W}_q, \mathbf{W}_k$ に吸収可能）のもとで、任意の直交行列 $\mathbf{R}_1$ は 2×2 の平面回転へと分解される。RoPE は、各クエリとキーの d_h 個の座標を $d_h/2$ 組のペアに分割し、ペア m を位置 i において角度 $i\omega_m$（ただし $\omega_m = 10000^{-2m/d_h}$）だけ回転させる（Figure 10.9）。

$$\begin{aligned} \mathbf{R}_i &= \begin{bmatrix} R(i\omega_0) & & \\ & \ddots & \\ & & R(i\omega_{d_h/2-1}) \end{bmatrix}, \\ R(\alpha) &= \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}. \end{aligned} \tag{10.24}$$

値は回転させないため、層の出力は大域的な位置シフトに対して不変に保たれる。RoPE は、現代のオープンウェイトの言語モデルにおける標準的な位置エンコーディング（positional encoding）である。

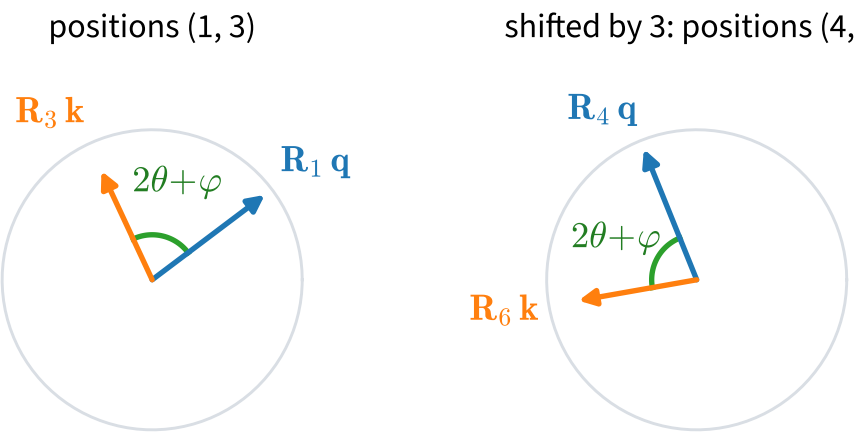


Figure 10.9: 回転埋め込みは、クエリとキーの 2 次元特徴ペアのそれぞれを位置に比例する角度だけ回転させる。両方の位置を同じ量（ここでは 3）だけシフトすると、両ベクトルが一緒に回転するため、両者のなす角（どちらのパネルでも $2\theta + \varphi$）、ひいてはアテンションスコアは変化しない。ここで、$\theta = \omega_m$ はペア m が位置ごとに回転する角度であり、φ は回転前のクエリとキーのなす角である。

```
def rope_tables(n, d, offset=0, base=10000.0, scale=None,
                device=None):
    """Cosines and sines, each (n, d/2), of the angles of the positions
    offset, ..., offset + n - 1."""
    pos = torch.arange(n, dtype=torch.float32, device=device) + offset
    omega = base ** (-torch.arange(0, d, 2, device=device) / d)
    if scale is not None:  # One angle factor per pair, shape (d/2,)
        omega = omega * scale
    theta = pos[:, None] * omega[None, :]
    return torch.cos(theta), torch.sin(theta)

def rope(x, offset=0, base=10000.0, scale=None, tables=None):
    """Rotate feature pairs of x (..., num_steps, d) by position angles;
    tables from rope_tables take the place of offset, base, and scale."""
    if tables is None:
        tables = rope_tables(*x.shape[-2:], offset, base, scale, x.device)
    # Angles in fp32, tables in the dtype of x: bf16 queries stay bf16
    cos, sin = (u.to(x.dtype) for u in tables)
    x1, x2 = x[..., 0::2], x[..., 1::2]
    return torch.stack([x1 * cos - x2 * sin,
                        x1 * sin + x2 * cos], -1).flatten(-2)
```

すべての位置を任意のオフセットだけシフトしても、RoPE のスコアは float32 の丸め誤差の範囲で同一に保たれるのに対し、加算型正弦波のスコアは Equation 10.22 の内容・位置の交差項のために $O(1)$ だけ変化する。

```
T, d = 32, 64
q, k = torch.randn(T, d), torch.randn(T, d)

scores = lambda off: rope(q, off) @ rope(k, off).T / math.sqrt(d)
for off in (1, 17, 480):
    print(f'RoPE, all positions shifted by {off:>3}: max score change '
          f'{(scores(off) - scores(0)).abs().max():.1e}')

P = sinusoidal_encoding(600, d)
added = lambda off: (q + P[off:off+T]) @ (k + P[off:off+T]).T / math.sqrt(d)
print(f'additive sinusoidal, shifted by  17: max score change '
      f'{(added(17) - added(0)).abs().max():.1e}')
```

```
RoPE, all positions shifted by   1: max score change 2.0e-06
RoPE, all positions shifted by  17: max score change 2.1e-06
RoPE, all positions shifted by 480: max score change 1.5e-05
additive sinusoidal, shifted by  17: max score change 2.7e+00
```

10.4.4 距離ペナルティとエンコーディングなし

さらに別の 2 つの手法では、トークン、クエリ、キーの各ベクトルには一切手を加えない。

- **ALiBi**（**Attention with Linear Biases**、(Press et al. 2022)）は、ヘッド $h \in \{1, \dots, H\}$ における因果アテンション（causal attention）のスコアから線形の距離ペナルティを直接減算する。

$$\text{score}_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_h}} - m_h\,(i-j), \qquad m_h = 2^{-8h/H}. \tag{10.25}$$

  等比的な傾き m_h（たとえば $1/4, 1/16, 1/64, 1/256$、$H = 4$ の場合）によってヘッドごとに異なる直近ホライズンが与えられ、固定のテーブルサイズを必要としない。

- **NoPE**（**No Positional Encoding**、(Kazemnejad et al. 2023)）は位置信号を一切加えず、因果マスク（$i + 1$ 個のキーを位置 i で参照可能にする）のみに頼って置換対称性を破る。

10.4.5 訓練長を超える外挿

これら 5 つの手法が分布内（in-distribution）およびより長いコンテキストへ外挿する際にどのように振る舞うかを比較するため、長さ $L = 128$ バイトで訓練し、長さ 128、256、512 でパープレキシティを評価する。

10.4.5.1 言語モデル TinyCharLM

`TinyCharLM` は 256 個のバイト値を幅 $d = 128$ に埋め込み、2 つの因果自己アテンションブロック（$H = 4$ 個のヘッド、各幅 $d_h = 32$ であり、位置情報が pos のみから入るように MLP や正規化層は設けない）を適用したのち、共有された埋め込み重みを用いてバイトロジットへと射影する (Press and Wolf 2017)。

```
class TinyCharLM(nn.Module):
    """Attention-only language model."""
```

```
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4, num_blks=2,
                 pos='rope', max_len=512, bias=False):
        super().__init__()
        assert pos in ('learned', 'sinusoidal', 'rope', 'alibi', 'none')
        self.num_heads, self.pos = num_heads, pos
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        nn.init.normal_(self.token_emb.weight, std=0.02)
        if pos == 'learned':
            self.pos_emb = nn.Embedding(max_len, num_hiddens)
            nn.init.normal_(self.pos_emb.weight, std=0.02)
        if pos == 'sinusoidal':
            self.register_buffer('P', sinusoidal_encoding(max_len,
                                                          num_hiddens))
        if pos == 'rope':  # Base and per-pair angle factors of the rotation
            self.rope_base = 10000.0
            self.register_buffer('rope_scale',
                                 torch.ones(num_hiddens // num_heads // 2))
        self.blks = nn.ModuleList([nn.ModuleDict(dict(
            qkv=nn.Linear(num_hiddens, 3 * num_hiddens, bias=bias),
            proj=nn.Linear(num_hiddens, num_hiddens, bias=bias)))
            for _ in range(num_blks)])

    def stream(self, X, num_blks=None):
        """Residual stream after the first num_blks blocks (default: all),
        and the attention weights of those blocks."""
        H, maps = self.embed(X), []
        for blk in self.blks[:num_blks]:
            out, weights = self.attend(blk, H)
            H, maps = H + out, maps + [weights]
        return H, maps

    def forward(self, X):
        return F.linear(self.stream(X)[0], self.token_emb.weight)  # Tied

    def attention_weights(self, X):
        """Per-block attention maps, each (batch, num_heads, T, T)."""
        return self.stream(X)[1]
```

`embed` では、重み共有された出力ロジットを変えることなく、正弦波テーブルを $1/\sqrt{d}$ 倍してその要素のスケールを $\mathcal{N}(0, 0.02^2)$ のトークン埋め込みに合わせる。

```
@d2l.add_to_class(TinyCharLM)
def embed(self, X):
    """Token embeddings plus the absolute position code, if any."""
    H = self.token_emb(X)
    if self.pos == 'learned':
        H = H + self.pos_emb(torch.arange(X.shape[1], device=X.device))
    if self.pos == 'sinusoidal':  # Table scaled to the token embeddings
        H = H + self.P[:X.shape[1]] / math.sqrt(H.shape[-1])
    return H
```

`attend` では、RoPE はクエリとキーを回転させるのに対し、ALiBi はソフトマックス前のスコア

に距離バイアス $-m_h(i-j)$ を加算する。

```
@d2l.add_to_class(TinyCharLM)
def alibi(self, T, device):
    """ALiBi bias -m_h (i - j) of every head, (num_heads, T, T)."""
    h = torch.arange(1, self.num_heads + 1, device=device)
    slopes = 2.0 ** (-8.0 * h / self.num_heads)
    pos = torch.arange(T, device=device, dtype=torch.float32)
    return slopes[:, None, None] * (pos[None, :] - pos[:, None])

@d2l.add_to_class(TinyCharLM)
def attend(self, blk, H):
    """Output of attention block blk on the stream H, and its weights."""
    B, T, D = H.shape
    q, k, v = blk['qkv'](H).chunk(3, -1)
    q, k, v = (u.reshape(B, T, self.num_heads, -1).transpose(1, 2)
               for u in (q, k, v))
    if self.pos == 'rope':
        q = rope(q, base=self.rope_base, scale=self.rope_scale)
        k = rope(k, base=self.rope_base, scale=self.rope_scale)
    scores = q @ k.transpose(-2, -1) / math.sqrt(q.shape[-1])
    if self.pos == 'alibi':
        scores = scores + self.alibi(T, H.device)
    weights = d2l.masked_softmax(scores, d2l.causal_mask(T, device=H.device))
    out = (weights @ v).transpose(1, 2).reshape(B, T, D)
    return blk['proj'](out), weights
```

10.4.5.2 実験

`d2l.WikiText103`（9.3.1 節）の 128 バイトウィンドウ上で、各手法につき 3 つのシードを 3,000 ステップの AdamW により訓練する。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
schemes = ('learned', 'sinusoidal', 'rope', 'alibi', 'none')

def train_tiny(**kwargs):
    """A TinyCharLM after 3000 steps, and its final training loss."""
    model = TinyCharLM(len(data.vocab), **kwargs)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                                  weight_decay=0.0)
    return model, d2l.final_loss(d2l.train_steps(model, data, optimizer,
                                                 3000))

runs = {}  # Three independently initialized models per scheme
for pos in schemes:
    trained = [train_tiny(pos=pos) for _ in range(3)]
    runs[pos] = [model for model, _ in trained]
    print(f'{pos:>10}: final training loss '
          + ', '.join(f'{loss:.2f}' for _, loss in trained))
models = {pos: ms[0] for pos, ms in runs.items()}  # The first run of each
```

```
   learned: final training loss 2.28, 2.27, 2.26
sinusoidal: final training loss 2.26, 2.28, 2.29
      rope: final training loss 2.02, 2.00, 1.99
     alibi: final training loss 2.31, 2.29, 2.31
      none: final training loss 2.45, 2.46, 2.47
```

閉形式のバイトバイグラムのベースラインとあわせて、同一の検証用バイト列を $n \in \{128, 256, 512\}$ のウィンドウに再編成したデータ上で検証パープレキシティを評価する。

```
train_len, batch = data.num_steps, 16
contexts = (train_len, 2 * train_len, 4 * train_len)
block = batch * max(contexts)  # Whole minibatches at every context
num_bytes = (len(data.val) - 1) // block * block
X, Y = data.windows(data.val, range(0, num_bytes, train_len))
eval_sets = {}
for n in contexts:  # The same bytes, regrouped into windows of n
    Xn, Yn = X.reshape(-1, n), Y.reshape(-1, n)
    eval_sets[n] = [(Xn[i:i + batch], Yn[i:i + batch])
                    for i in range(0, len(Xn), batch)]

def perplexities(model):
    return [d2l.lm_perplexity(model, eval_sets[n]) for n in contexts]

def mean_range(values):
    """Mean and (min-max) of the runs, one column of the table."""
    return f'{np.mean(values):5.1f} ({min(values):.1f}-{max(values):.1f})'

ppls = {pos: [perplexities(model) for model in ms] for pos, ms in runs.items()}
print(f'{num_bytes:,} validation bytes, mean (min-max) of three runs')
print(f'{"context":>10}: ' + ''.join(f'{n:>18}' for n in contexts))
for pos in schemes:  # zip(*...) turns runs x contexts into contexts x runs
    print(f'{pos:>10}: '
          + ''.join(f'{mean_range(col):>18}' for col in zip(*ppls[pos])))
```

```
1,138,688 validation bytes, mean (min-max) of three runs
   context:                128                256                512
   learned:      9.6 (9.5-9.7)  15.3 (14.9-15.8)  19.7 (18.7-21.1)
sinusoidal:      9.7 (9.4-9.9)  12.8 (12.5-13.3)  15.3 (14.7-15.9)
      rope:      7.3 (7.2-7.4)  11.8 (10.5-13.6)  27.0 (18.4-35.5)
     alibi:     9.9 (9.8-10.0)    9.9 (9.8-10.0)     9.9 (9.7-9.9)
      none:   11.6 (11.5-11.7)  11.7 (11.6-11.8)  11.8 (11.8-11.8)
```

これらの数値の基準点を得るため、訓練コーパス全体のラプラス平滑化された経験遷移カウントから、同じ検証ウィンドウ上での厳密なバイト単位バイグラムのパープレキシティを計算する。

```
V = len(data.vocab)
counts = np.ones((V, V))  # Byte pair counts, plus one for unseen pairs
for s in range(0, len(data.train) - 1, 10**7):  # Chunks of 10^7 bytes
    a = data.train[s:s + 10**7 + 1].astype(np.int64)
    counts += np.bincount(a[:-1] * V + a[1:], minlength=V * V).reshape(V, V)
bigram = counts / counts.sum(1, keepdims=True)  # P(next byte | this byte)
x, y = np.asarray(X).reshape(-1), np.asarray(Y).reshape(-1)
print(f'byte bigram: perplexity {np.exp(-np.log(bigram[x, y]).mean()):.1f}')
```

```
byte bigram: perplexity 11.6
```

訓練長（$n = 128$）では、RoPE が最も低いパープレキシティ（7.3）を達成し、学習型（9.6）、正弦波（9.7〜9.8）、ALiBi（9.9〜10.0）を上回る。これらはいずれもバイトバイグラムのベースライン（11.6）より改善しているが、この小さな 2 ブロックモデルでは NoPE はバイグラムの水準にとどまる。

$n = 512$ では、表現が訓練範囲外の値に遭遇する 3 つの手法が急激に劣化する。すなわち、学習型埋め込みは未訓練の行（128〜511）にあたり、正弦波の低速な列は未知の値をとり、RoPE の低周波ペアは訓練中に一度も現れなかった角度へと回転する。一方、ALiBi のパープレキシティは 9.9 のまま平坦に保たれる。その理由を調べるため、ALiBi の 4 つのヘッドが距離 ≥ 128 のキーに割り当てるアテンション重みの割合を計測し、距離 ≥ 128 のすべてのキーをマスクした場合と比較する。

```
alibi_model, T = models['alibi'], max(contexts)
device = next(alibi_model.parameters()).device
X_eval = eval_sets[T][0][0].to(device)  # 16 windows of 512 bytes
i = torch.arange(T, device=device)
far = i[:, None] - i[None, :] >= train_len  # Offsets never trained on

def far_weight(weights):
    """Per head: mean weight on far keys, over queries from train_len on."""
    return (weights * far).sum(-1)[..., train_len:].mean((0, -1)).tolist()

with torch.inference_mode():
    maps = alibi_model.attention_weights(X_eval)
    bias_only = d2l.masked_softmax(alibi_model.alibi(T, device)[None],
                                   d2l.causal_mask(T, device=device))
print(f'{"head":>10}: ' + ''.join(f'{h:>9}' for h in range(1, 5)))
for name, weights in [('bias alone', bias_only)] + [
        (f'block {b}', w) for b, w in enumerate(maps, 1)]:
    print(f'{name:>10}: ' + ''.join(f'{w:9.3f}' for w in far_weight(weights)))
```

```
      head:         1        2        3        4
bias alone:     0.000    0.000    0.115    0.401
   block 1:     0.000    0.000    0.115    0.400
   block 2:     0.000    0.000    0.111    0.395
```

これを `WindowedALiBi` によって直接確認しよう。このクラスは訓練済み ALiBi の重みをコピーし、距離 ≥ 128（`window=train_len`）のすべてのキーをハードマスクして長さ $T = 512$ で評価する。

```
class WindowedALiBi(TinyCharLM):
    """ALiBi whose queries see only the window most recent keys."""
    def __init__(self, *args, window, **kwargs):
        super().__init__(*args, **kwargs)
        self.window = window

    def alibi(self, T, device):
        i = torch.arange(T, device=device)
        return super().alibi(T, device).masked_fill(
            i[:, None] - i[None, :] >= self.window, -math.inf)

full, windowed = [], []
```

```
for model in runs['alibi']:
    model_w = WindowedALiBi(len(data.vocab), pos='alibi',
                            window=train_len).to(device)
    model_w.load_state_dict(model.state_dict())  # Same trained weights
    full.append(d2l.lm_perplexity(model, eval_sets[T]))
    windowed.append(d2l.lm_perplexity(model_w, eval_sets[T]))
for name, ppl in ('all keys', full), (f'keys < {train_len} back', windowed):
    print(f'{name:>16}: perplexity at {T} {mean_range(ppl)}')
```

```
      all keys: perplexity at 512   9.9 (9.7-9.9)
keys < 128 back: perplexity at 512   9.9 (9.7-9.9)
```

距離 ≥ 128 のすべてのキーを取り除いても ALiBi のパープレキシティは変化しない。傾きの急なヘッドはもともと遠方のキーを無視しており、緩やかなヘッドは広い直近重み付き平均を計算していて 128 バイトより先の情報から何も得ていないためである。したがって、平坦な外挿曲線はその手法が遠方のキーで破綻しないことを示しているにすぎず、必ずしもそこから長距離の情報を抽出していることを意味しない。

10.4.5.3 訓練済みモデルのコンテキスト拡張

なぜ RoPE は訓練長 $L = 128$ を超えると破綻するのか、そしてどうすればそれを修復できるだろうか。ヘッド幅 $d_h = 32$ の場合、16 組の座標ペアは周期 $2\pi/\omega_m = 2\pi \cdot 10000^{m/16}$ を持つ。最初の 6 組の**高速**ペアは周期が $\leq 112 < 128$ であり、訓練ウィンドウ内で既に 1 周以上の回転を完了している。一方、残りの 10 組の**低速**ペアは周期が 199 から 35,000 に及び、$n = 512$ では未知の角度領域へと回転してしまう。

位置補間（position interpolation）では (Shouyuan Chen et al. 2023)、すべての角度を $128/512 = 1/4$ 倍にスケーリングして、新しいコンテキスト範囲を訓練済み区間 $[0, 128)$ に写し戻す。しかし、ファインチューニングなしに高速ペアまでスケーリングすると、隣接バイトの区別に使われている局所的な位相差が損なわれる。高速ペアには手を加えず低速ペア**のみ**をスケーリングする手法は、**YaRN** (Peng et al. 2024) や **NTK 考慮スケーリング**（NTK-aware scaling、基数を $b \mapsto b\, s^{d_h/(d_h-2)}$ へと大きくする手法）の中核にある考え方であり、訓練コストゼロで外挿時のパープレキシティの大半を回復できる。

```
rope_model = models['rope']
d_head = 2 * rope_model.rope_scale.numel()
periods = (2 * math.pi
           * rope_model.rope_base ** (torch.arange(0, d_head, 2) / d_head))
slow = periods > train_len
print(f'{int(slow.sum())} of {d_head // 2} pairs have periods above '
      f'{train_len}')
ones = torch.ones(d_head // 2)
shrink = train_len / max(contexts)  # Longest context into the trained one
factors = {'unscaled': ones, 'all pairs': ones * shrink,
           'slow pairs': torch.where(slow, ones * shrink, ones)}
print(f'{"context":>10}: ' + ''.join(f'{n:>18}' for n in contexts))
for name, factor in factors.items():  # Each of the three trained RoPE runs
    ppl = []
    for rope_model in runs['rope']:
        rope_model.rope_scale.copy_(factor)
        ppl.append(perplexities(rope_model))
```

```
            rope_model.rope_scale.copy_(ones)
        print(f'{name:>10}: ' + ''.join(f'{mean_range(col):>18}'
                                        for col in zip(*ppl)))
```

```
10 of 16 pairs have periods above 128
   context:                128               256               512
  unscaled:      7.3 (7.2-7.4)  11.8 (10.5-13.6)  27.0 (18.4-35.5)
 all pairs:   40.5 (37.9-43.4)  40.7 (38.2-43.6)  40.8 (38.3-43.7)
slow pairs:      8.0 (7.8-8.2)     8.5 (8.2-8.7)     8.8 (8.5-9.1)
```

低速ペアのみを再スケーリングすると、重みを 1 つも更新することなく長さ 512 でのパープレキシティが約 27 から約 9 へと低下し、目標長で短時間のファインチューニングを行えば残りの差も解消される（演習問題 6）。

10.4.6 まとめ

マスクなし自己アテンションは置換同変であり、$f(\Pi\mathbf{X}) = \Pi f(\mathbf{X})$ を任意のトークン置換 Π に対して満たす。加算型の位置埋め込み（学習型または正弦波）は入力段階でこの対称性を破るが、絶対インデックスに結びついた内容・位置の交差項を生じさせ、訓練長を超えると劣化する。アテンションスコアが相対オフセット $j-i$ のみに依存することを要請すると、クエリとキーに対する直交平面回転（**RoPE**）が導かれるのに対し、**ALiBi** はヘッドごとの線形距離ペナルティ $-m_h(i-j)$ を減算する。訓練ウィンドウを超えると、RoPE は特に波長が訓練長を上回る低周波の座標ペアにおいて劣化するが、高周波ペアをそのまま残しつつこれらの低速ペアを選択的に補間することで、長文脈での性能が回復する。

10.4.7 演習問題

1. **異なる位置と異なる符号。** $\mathbf{p}_i$ を、第 i 行として正弦波テーブル Equation 10.20（幅 d、周波数 $\omega_m = 1/10000^{2m/d}$）から取り出した行ベクトルとする。
   1. 次式を示し、
   $$\mathbf{p}_i^\top \mathbf{p}_{i+\delta} = \sum_{m=0}^{d/2-1} \cos(\delta\,\omega_m),$$
   これから $\|\mathbf{p}_i - \mathbf{p}_{i+\delta}\|^2$ を導出して、両方の量がオフセット δ のみに依存することを結論づけよ。
   2. $\mathbf{p}_i = \mathbf{p}_{i'}$ が成り立つのは $(i-i')\,\omega_m$ が 2π の倍数である（すべての m について）場合に限られること、したがって `max_len` の値にかかわらず、相異なる 2 つの整数位置が同じ行を共有することはないことを結論づけよ。
   3. 最も速いペアは周期 2π を持つため、2π の倍数に近い整数オフセットでほぼ同じ値を繰り返す。オフセット $\delta < 512$ のうち、その角度が 2π の倍数に最も近づくものを求め、残差（residual）の角度を示せ。$d = 32$ の場合について、このオフセットにおける $\|\mathbf{p}_i - \mathbf{p}_{i+\delta}\|$ を $\delta = 1$ における値と比較し、ほぼ反復する 2 つの位置をどのペアが引き離しているかを述べ、その答えを 10.4.2.2 節の二進カウンタと関連づけよ。
2. **スコアにおける加算型符号。** 10.4.2.2 節の加算型エンコーディングを用いるとき、第 1 層のヘッドは位置 i と j の入力に対し、Equation 10.22 の 4 つの項（ただし $\mathbf{A} = \mathbf{W}_q\mathbf{W}_k^\top/\sqrt{d_h}$）によってスコアを計算する。
   1. $\mathbf{J}$ を平面上の $\pi/2$ 回転とする。Equation 10.21 を用いて、位置・位置の項 $\mathbf{p}_i^\top \mathbf{A}\,\mathbf{p}_j$ は、$j-i$ のみに依存することを、$\mathbf{A}$ が周波数ペアごとに 1 つの 2×2 ブロック $a_m\mathbf{I} + b_m\mathbf{J}$ を持つブロック対角行列である場合について示せ。演習問題 1 は $\mathbf{A} = \mathbf{I}$ の場合を扱っ

ている。次に、異なる 2 つの周波数ペアを結合する $\mathbf{A}$ の非ゼロ要素が 1 つでもあると、この性質が損なわれることを示せ。

2. 内容・位置の項 $\mathbf{x}_i^\top \mathbf{A}\, \mathbf{p}_j$ が j のあらゆるシフトに対して変化しない条件を求めよ。その答えを用いて、10.4.3 節のシフト実験における加算型エンコーディングのスコア変化と、RoPE に内容・位置の項が存在しない理由を説明せよ。

3. [code] **どこで外挿が破綻するか**。10.4.5.2 節におけるコンテキスト 512 でのパープレキシティは、各系列のすべての位置にわたって損失を平均したものである。
   1. 5 つのモデルそれぞれにおいて、長さ 512 の系列の位置 0 から 127 におけるロジットが、先頭の 128 トークンのみから計算したロジットと浮動小数点の丸め誤差の範囲で一致することを数値的に確認せよ。`TinyCharLM` のどの 2 つの性質が、すべての手法に対してこれを保証しているか。
   2. 5 つのモデルについて長さ 512 の系列の各位置における検証損失を計算し、16 位置ごとのウィンドウで平均して位置に対してプロットせよ。各手法について、曲線はどの位置で位置 128 未満で保っていた水準から離れるか。また、損失は位置 128 で不連続に跳ね上がるか、それとも徐々に増大するか。
   3. `models['rope']` のすべてのクエリが自身を含む直近 128 個のキーのみを参照するよう、`attend` をオーバーライドするサブクラスで因果マスクに帯状の制約を追加し、訓練済みの重みをコピーして同じ計測を繰り返せ。位置 128 以降のウィンドウ制限付き曲線と、それより前の制限なし曲線とを比較せよ。127 を超えるオフセットは、コンテキスト 512 における RoPE の劣化を説明するか。

4. **対称性を保つバイアス**。●証明中のソフトマックス前のスコア行列に対し、固定行列 $\mathbf{B} \in \mathbb{R}^{n\times n}$ を $\mathbf{S}$ に加算して、自己アテンション層（10.3 節）が $\mathrm{softmax}(\mathbf{S}+\mathbf{B})\,\mathbf{X}\mathbf{W}_v$ を計算するものとする。
   1. 置換行列 Π が $\Pi^\top \mathbf{B} \Pi = \mathbf{B}$ を満たす限り、命題が依然として成り立つことを示せ。
   2. すべての置換に対してこの条件を満たす行列 $\mathbf{B}$ をすべて求め、そのそれぞれがアテンション重みをどのように変化させるかを述べよ。
   3. 因果マスクがない場合、Equation 10.25 の ALiBi バイアスは対称な形 $B_{ij} = -m_h\,|i-j|$ へと拡張される。この行列を保つ置換をすべて求め、このバイアス以外に一切の位置信号を受け取らないマスクなし層に対してその結果が何を意味するかを述べよ。また、Equation 10.25 の因果的な ALiBi がこの対称性を共有しない理由を説明せよ。

5. **RoPE におけるオフセット選好**。●クエリ $\mathbf{q}$ とキー $\mathbf{k}$ を、$d_h/2$ 組の 2 次元ペア $\mathbf{q}^{(m)}, \mathbf{k}^{(m)} \in \mathbb{R}^2$（Equation 10.24 が作用する単位）に分割し、$\delta = j - i$ をキー位置 j のクエリ位置 i からのオフセットとする。
   1. RoPE のスコア Equation 10.23 が次式に等しいことを示せ。

$$\sum_{m=0}^{d_h/2-1} \|\mathbf{q}^{(m)}\|\,\|\mathbf{k}^{(m)}\| \cos(\delta\,\omega_m - \psi_m), \tag{10.26}$$

   ここで、ψ_m は $\mathbf{k}^{(m)}$ から $\mathbf{q}^{(m)}$ への角度（符号を反転させた φ、Figure 10.9 参照）である。
   2. 角度 ψ_m を、すべてのペアが直前のトークンであるオフセット $\delta = -1$ で最大の寄与を与えるように選べ。すべての積 $\|\mathbf{q}^{(m)}\|\,\|\mathbf{k}^{(m)}\|$ が c に等しいとき、スコアは δ の関数として、c とドット積 $\mathbf{p}_i^\top \mathbf{p}_{i+\delta}$（演習問題 1 における幅 d_h の正弦波テーブルのもの）をシフトしたものの積になることを示せ。
   3. $d_h = 32$ とし、位置 127 にあるクエリ（オフセット $\delta = -127, \dots, 0$ を参照する）を考える。係数 $1/\sqrt{d_h}$ を c に吸収させ、アテンションロジットがこれらのスコアに等しいとする。このとき、c の値として、オフセット -1 がソフトマックス重みの半分を受け取るようなものを求め、それと最も強く競合するオフセットを特定せよ。オフ

セット -1 でのスコアとその隣接オフセットでのスコアとの差を決定づけているのはどのペアか。

6. [extended] **位置補間。**●10.4.5.3 節の最後のコードセルでは、訓練済みの RoPE モデルの角度を一切の追加訓練なしに再スケーリングしている。本問ではその結果の理由を解明し、NTK 考慮スケーリングの基数変更を加え、最後にファインチューニングによって手法を完成させる。
   1. 訓練オフセット 0 から 127 の範囲内で 1 回転を完了するペアは、より長いオフセットが生み出す任意の角度に対し、$\omega_m/2$ 以内の差（2π を法として）で既に出会っていることを示せ。演習問題 5 におけるオフセット -1 のスコアとその隣接オフセットのスコアとの差を、$d_h = 32$ の場合について、再スケーリング前、$128/512$ による一様な再スケーリング後、および周期が 128 位置を超えるペアのみを再スケーリングした後のそれぞれで計算せよ。その結果を、コードセル内のコンテキスト 128 におけるパープレキシティと関連づけよ。
   2. 基数 $b' = b\,s^{d_h/(d_h-2)}$ が ω_0 を不変に保ちつつ $\omega_{d_h/2-1}$ を s で割ることを示せ。属性 `rope_base`（`models['rope']` 内）を $s = 4$ に対するこの基数に設定し、訓練せずに長さ 128、256、512 で評価して、同セルの 2 つの再スケーリング手法と比較せよ。
   3. コンテキスト 512 のもと、`d2l.WikiText103(batch_size=16, num_steps=512)` 上で同セルの 2 つの再スケーリング手法のそれぞれについて（いずれも訓練済み `models['rope']` のコピーを初期値として）数百ステップだけファインチューニングせよ。長さ 128、256、512 で再評価し、スケーリングなしのモデルおよび 10.4.5.2 節の実験における同じ長さでの ALiBi モデルと比較せよ。

10.5 アテンションのコスト

長文脈（long-context）言語モデルは、1 回のパスで数万から数十万個のトークンを処理する。密な自己アテンションはあらゆる位置の対に対してスコアを評価するため、演算量と素朴な実装でのメモリ使用量はいずれも系列長 n の二乗に比例して増大する。$n = 8{,}192$ のとき、単一のヘッドは 6.7×10^7 個のクエリ・キー対（float32 で 256 MiB）を評価することになる。

数年にわたり、$O(n^2)$ のメモリは厳密なアテンションに避けられないコストだと広く考えられていた。その後、Rabe and Staats (2021) は Milakov and Gimelshein (2018) の**オンラインソフトマックス**のアルゴリズムを発展させ、厳密なアテンションにおいて $n \times n$ のスコア行列を実体化（materialize）する必要が一切ないことを示した。さらに **FlashAttention** (Dao et al. 2022) は、タイル化したオンラインソフトマックスを IO を考慮した GPU カーネルへと昇華させ、素朴なアテンションより高速でありながらメモリ使用量が線形にとどまる計算を実現した。

本節では、密なアテンションの FLOPs とメモリ使用量を算出して GPU 上でベンチマークし、ブロック単位のオンラインソフトマックスを導出したうえで、チャンク分割による実装および融合された FlashAttention カーネルを素朴な行列実体化と比較する。

```
from torch.nn.attention import SDPBackend, sdpa_kernel
```

10.5.1 二次の時間とメモリ

10.5.1.1 FLOPs の算出

幅 d、H 個のヘッド（各ヘッドの幅 $d_h = d/H$）、トークン数 n のマルチヘッド自己アテンション層では、マルチヘッドの計算量（10.3 節）に行ごとのソフトマックスを合わせると次式が得られる。

$$\underbrace{8nd^2}_{\text{projections}} + \underbrace{4n^2d}_{\text{scores and mixing}} + O(Hn^2) \quad \text{FLOPs.} \tag{10.27}$$

二次の項 $4n^2d$（$2n^2d_h$ が $\mathbf{QK}^\top$ 分、$2n^2d_h$ が $\mathbf{AV}$ 分で、H 個のヘッドの合計）は、線形な射影の項 $8nd^2$ を $n > 2d$ となった時点で上回る。GPT-2 small（$d = 768$、$n = 1{,}024$）ではまだ射影が支配的である。一方、$d = 4{,}096$ のモデルで $n = 131{,}072$ とすると $n/(2d) = 16$ となり、スコア計算と混合にはアテンションの射影に対して $16\times$ の FLOPs を要する（第 11 章のフィードフォワードネットワークを含めた Transformer ブロック全体に対しても約 $5\times$ であるが、これを含めるとブロック全体での逆転点は $n \approx 6d$ に移動する）。

10.5.1.2 メモリ使用量の算出

長文脈では、演算量よりも先にメモリがボトルネックになることが多い。素朴な実装は $n \times n$ のスコア行列とその $n \times n$ のソフトマックス重み行列の両方を実体化するため、float32 では 1 ヘッド・1 系列あたり $8n^2$ バイトを消費する（これに対し射影は $O(nd)$ である）。幅 $d_h = 64$ の 1 つのヘッドにおいて $n = 8{,}192$ のとき、$\mathbf{Q}, \mathbf{K}, \mathbf{V}$ の合計はわずか 6 MiB にすぎないが、2 つの $n \times n$ バッファは 512 MiB を消費する。さらに $n = 131{,}072$ では、単一の float32 アテンション行列だけで 64 GiB が必要になる。訓練時に GPT-2 small の全 12 層・12 ヘッドにわたってアテンション重みを $n = 8{,}192$ で保存しようとすれば、1 系列あたり 144×256 MiB $= 36$ GiB を要することになる。

$8n^2$ バイトの一時メモリ使用量を、1 つの float32 ヘッド（$d_h = 64$）上で `d2l.peak_memory`（5.7 節）を用いて確認しよう。

```
def attention(Q, K, V):
    """softmax(Q K^T / sqrt(d_h)) V through the full score matrix."""
    scores = Q @ K.transpose(-2, -1) / math.sqrt(Q.shape[-1])
    return torch.softmax(scores, dim=-1) @ V
```

`d2l.peak_memory(attention, Q, Q, Q)` を系列長 $n \in \{2048, 4096, 8192, 16384\}$ にわたって評価すると、n が 2 倍になるごとに一時メモリの確保量が $8n^2$ バイトの理論値どおり 4 倍になることが確認できる。

```
d_h = 64
for n in [2048, 4096, 8192, 16384]:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    mem = d2l.peak_memory(attention, Q, Q, Q)  # Allocator high-water mark
    print(f'n = {n:5d}: measured {mem/2**20:7.1f}'
          f' MiB, predicted 8n^2 B = {8*n*n/2**20:7.1f} MiB')
```

```
n =  2048: measured    40.6 MiB, predicted 8n^2 B =    32.0 MiB
n =  4096: measured   129.0 MiB, predicted 8n^2 B =   128.0 MiB
n =  8192: measured   514.0 MiB, predicted 8n^2 B =   512.0 MiB
n = 16384: measured  2052.0 MiB, predicted 8n^2 B =  2048.0 MiB
```

10.5.1.3 実行時間の計測

同じ順伝播の実行時間を `d2l.Benchmark`（5.7 節）で計測すると、n がカーネル起動のレイテンシを償却できるほど大きくなった領域（$n \gtrsim 8{,}192$）で、実行時間が二乗に比例して増加することが確認できる。

```
for n in [2048, 4096, 8192, 16384]:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    print(d2l.Benchmark(lambda: attention(Q, Q, Q), desc=f'n = {n:5d}'))
```

```
n =  2048: 0.07 ms/call
n =  4096: 0.46 ms/call
n =  8192: 1.84 ms/call
n = 16384: 7.37 ms/call
```

10.5.2 行列を保持しない厳密なアテンション

10.5.2.1 ブロック単位のオンラインソフトマックス

一見すると、ソフトマックスの分母 $\sum_{j'} \exp(a_{j'})$ と数値安定性のための行最大値 $\max_{j'} a_{j'}$（3.3.5.2 節）を求めるには、どの値に重み付けするよりも先に全 n 個のスコアが必要であるように思える。**オンラインソフトマックス** (Milakov and Gimelshein 2018) は、キーをサイズ c のブロック単位で処理し、クエリごとに 3 つの逐次統計量（running statistics）を保持することで、この 2 パスの要件を回避する。その 3 つとは、スコアの逐次最大値 m、分母の逐次和 $s = \sum_j e^{a_j - m}$、および未正規化の出力アキュムレータ $\mathbf{o} = \sum_j e^{a_j - m}\mathbf{v}_j$ である。

スコア $\{a_j\}$ と値 $\{\mathbf{v}_j\}$ の新しいブロックが到着したとき、逐次最大値を $m' = \max(m, \max_j a_j)$ に更新するには、それまでに累積した和 s と出力 $\mathbf{o}$ に $e^{m-m'}$ を掛けて再スケーリングする必要がある。

$$\begin{aligned} m' &= \max\left(m, \max_j a_j\right), \\ s' &= s\, e^{m-m'} + \sum_j e^{a_j - m'}, \\ \mathbf{o}' &= \mathbf{o}\, e^{m-m'} + \sum_j e^{a_j - m'}\, \mathbf{v}_j. \end{aligned} \tag{10.28}$$

たとえば、ブロック 1 のスコアが $(1, 3)$ であれば、$m = 3$ かつ $s = e^{-2} + e^{0} \approx 1.135$ となる。次にブロック 2 のスコアが $(5, 2)$ であれば、新しい最大値は $m' = 5$ となるため、ブロック 1 の和は $e^{3-5} = e^{-2}$ 倍に再スケーリングされたうえで $e^{5-5} + e^{2-5}$ が加算され、$s' = e^{-4} + e^{-2} + e^{0} + e^{-3} \approx 1.203$ が得られる。これは大域的最大値 5 をあらかじめ知った状態で計算する標準的なソフトマックスの分母と一致する。最後のブロックを処理し終えると、$\mathbf{o}/s$ は浮動小数点の丸め誤差の範囲内で $\mathrm{softmax}(\mathbf{a})\mathbf{V}$ と等しくなり、しかも一度にメモリ上に保持するのは 1 つの $n \times c$ スコアストライプだけで済む（Figure 10.10、(Rabe and Staats 2021)）。

10.5.2.2 チャンク分割による実装

行列全体を用いる `causal_attention` と、`chunked_attention` を比較する。後者は幅 `chunk_size=512` のキー・値ブロックを Equation 10.28 に従って順に処理する（ブロック全体が因果マスクされるときに m の初期化と未来の位置のマスクに `finfo(dtype).min` を用いることで、$-\infty - (-\infty) =$ NaN の発生を防いでいる。演習問題 2d を参照）。

```
def causal_attention(Q, K, V):
    """Reference: causal attention with the full score matrix."""
    scores = Q @ K.transpose(-2, -1) / math.sqrt(Q.shape[-1])
```

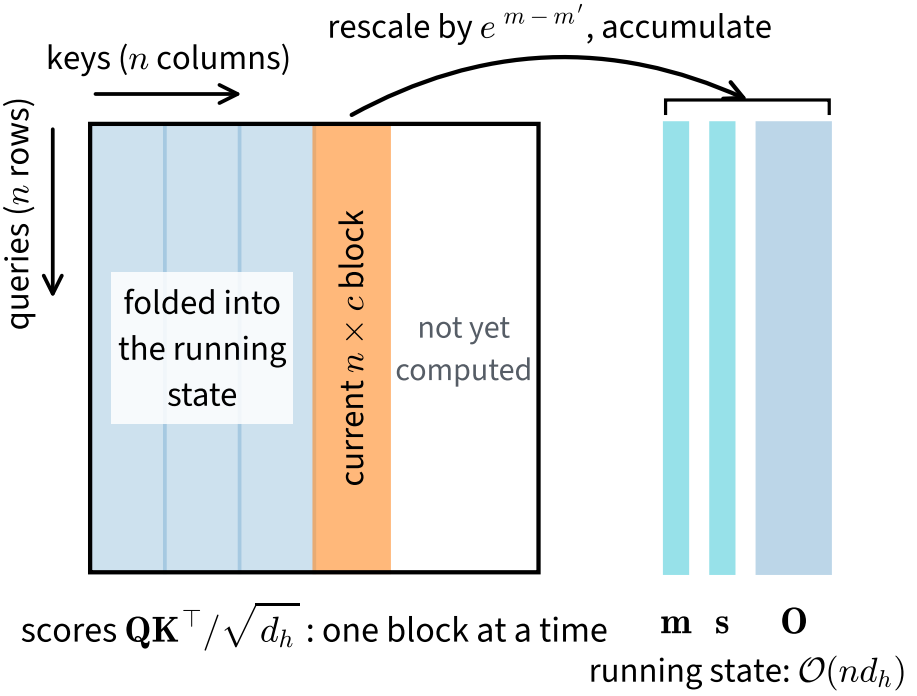


Figure 10.10: オンラインソフトマックスはスコア行列を 1 キーブロックずつ処理する。スコアのうちメモリ上に置かれるのは現在の $n \times c$ ストライプだけであり、その横に全クエリの逐次最大値・逐次和・出力 $\mathbf{m}, \mathbf{s}, \mathbf{O}$ が並ぶ。$n \times n$ の行列全体が保存されることは一切ない。

```
    mask = d2l.causal_mask(Q.shape[-2], device=Q.device)
    scores.masked_fill_(~mask, torch.finfo(Q.dtype).min)
    return torch.softmax(scores, dim=-1) @ V

def chunked_attention(Q, K, V, chunk_size=512):
    """Exact causal attention, one n-by-chunk_size block at a time."""
    n, d_h = Q.shape
    m = torch.full((n, 1), torch.finfo(Q.dtype).min, device=Q.device)
    s = torch.zeros(n, 1, device=Q.device)
    O = torch.zeros(n, V.shape[-1], device=Q.device)
    pos_q = torch.arange(n, device=Q.device)[:, None]
    for start in range(0, n, chunk_size):
        Kc, Vc = K[start:start + chunk_size], V[start:start + chunk_size]
        pos_k = torch.arange(start, start + Kc.shape[0], device=Q.device)
        scores = Q @ Kc.T / math.sqrt(d_h)
        scores.masked_fill_(pos_k[None, :] > pos_q, torch.finfo(Q.dtype).min)
        m_new = torch.maximum(m, scores.max(dim=-1, keepdim=True).values)
        scale = torch.exp(m - m_new)                  # Rescale the past
        P = torch.exp(scores - m_new)
        s = s * scale + P.sum(dim=-1, keepdim=True)
        O = O * scale + P @ Vc
        m = m_new
    return O / s
```

chunked_attention は causal_attention と float32 の精度（$\sim 10^{-6}$）で一致しつつ、ピークメモリを $O(n^2)$ から $O(nc)$ へと削減する。

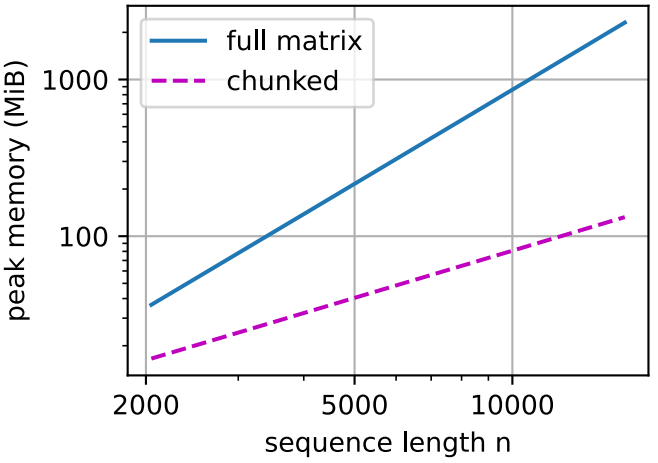
```
n, d_h = 2048, 64
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
err = (chunked_attention(Q, K, V) - causal_attention(Q, K, V)).abs().max()
print(f'maximum deviation: {float(err):.2e}')
```

```
maximum deviation: 2.38e-07
```

d2l.peak_memory を $n \in \{2048, 4096, 8192, 16384\}$ にわたって比較すると、chunked_attention が causal_attention の二次曲線的な $O(n^2)$ のメモリ増加を線形な $O(nc)$ のスケーリングに置き換えていることが確認できる。

```
lengths, mems = [2048, 4096, 8192, 16384], [[], []]
for n in lengths:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    mems[0].append(d2l.peak_memory(causal_attention, Q, Q, Q) / 2**20)
    mems[1].append(d2l.peak_memory(chunked_attention, Q, Q, Q) / 2**20)
print(f'n = {n:,}: full matrix {mems[0][-1]:6.1f} MiB, '
      f'chunked {mems[1][-1]:6.1f} MiB')
d2l.plot(lengths, mems, 'sequence length n', 'peak memory (MiB)',
         legend=['full matrix', 'chunked'], xscale='log', yscale='log')
```

```
n = 16,384: full matrix 2308.0 MiB, chunked  132.4 MiB
```



10.5.2.3 メモリトラフィックと融合カーネル

GPU の演算ユニットは、オフチップのデバイスメモリがデータを転送できる速度よりも何桁も高速に行列演算を実行する（たとえば RTX 4090 では、float16 の ∼ 165 TFLOP/s に対し、メモリ帯域幅は ∼ 1 TB/s である。第 13 章を参照）。データセンター向けカードはこのメモリを広帯域メモリ（high-bandwidth memory、HBM）で構成し、RTX 4090 は GDDR6X で構成する。本節では FlashAttention の論文に倣い、いずれの種類のデバイスメモリも HBM と表記する。素朴なアテンションは**メモリ帯域幅律速**（memory-bandwidth bound）である。すなわち、$n \times n$ のスコア行列を HBM に書き込み、ソフトマックスのためにそれを読み戻し、$n \times n$ の重みを HBM に書き込み、さらに $\mathbf{V}$ を掛けるために再びそれを読み出すため、低い演算密度（arithmetic intensity）のまま $4n^2$ 個の浮動小数点数を HBM 上で移動させることになる（演習問題 4 および 5）。`chunked_attention` のような Python ループも、結局は個別の GPU カーネルを起動してすべての $n \times c$ ストライプを HBM に対して読み書きするため、ピーク時のメモリ確保量は削減できても HBM の総トラフィックは削減できない。

FlashAttention (Dao et al. 2022) はクエリとキーの両方をタイル化し、各スコアブロックとその逐次統計量 $(m, s, \mathbf{o})$ がループの間ずっと高速なオンチップ SRAM 内に留まり、HBM に一切触れないようにする。逆伝播に向けては、順伝播で $n \times n$ のアテンション重みを保存する代わりに、クエリごとにスカラーの対数正規化項（log-normalizer）$\ell = m + \log s$ のみを保存し（$O(n)$ メモリ）、誤差逆伝播法の実行中にオンチップでスコアタイルを再計算する。

フレームワークが提供する融合 FlashAttention カーネルを `causal_attention` と比較し、float16 で $B = 2$ 系列、$H = 8$ ヘッド、$d_h = 64$、$n = 8{,}192$ の条件でベンチマークする。マスクなしの融合カーネルの実行も含めることで、$O(n^2)$ の HBM トラフィック削減による高速化と、上三角の因果タイルをスキップすることによる ∼ 1.5× の高速化とを切り分ける。

融合カーネルは `F.scaled_dot_product_attention` である。

```
def fused_attention(Q, K, V, is_causal=True):
    """Attention in PyTorch's fused FlashAttention kernel."""
    with sdpa_kernel(SDPBackend.FLASH_ATTENTION):
        return F.scaled_dot_product_attention(Q, K, V, is_causal=is_causal)

B, H, n = 2, 8, 8192
```

```
X = torch.randn(B, H, n, d_h, device=d2l.try_gpu(), dtype=torch.float16)
# The unmasked kernel computes every tile: it shows what skipping saves
for name, f in [('naive', causal_attention), ('fused', fused_attention),
                ('fused, no mask',
                 lambda Q, K, V: fused_attention(Q, K, V, is_causal=False))]:
    t = d2l.Benchmark(lambda: f(X, X, X)).time
    mem = d2l.peak_memory(f, X, X, X)
    print(f'{name:>14}: {t*1e3:6.2f} ms, peak memory {mem/2**20:7.1f} MiB')
```

```
         naive:  19.95 ms, peak memory  4176.0 MiB
         fused:   1.08 ms, peak memory    16.5 MiB
fused, no mask:   1.72 ms, peak memory    16.5 MiB
```

融合カーネルは一時メモリを ~ 4 GiB（$16 \cdot 4n^2$ バイト）から 16 MiB の出力の数倍程度まで削減し、GPU のピーク行列スループットに近い速度で数倍高速に動作する。しかし、融合カーネルを用いてもなお、厳密なアテンションは $\Theta(n^2 d)$ 回の算術演算を必要とする。線形時間計算量を達成するには、アテンション機構そのものを変更する必要がある（10.6 節）。

10.5.3 まとめ

密な自己アテンションは 1 層あたり $8nd^2 + 4n^2 d$ FLOPs を要するため、線形の射影よりも二次のスコア計算・混合の演算量が $n > 2d$ となった時点で大きくなる。さらに厄介なことに、$n \times n$ のスコア行列と重み行列を実体化すると float32 で 1 ヘッドあたり $8n^2$ バイトが確保され、現代の GPU では素朴なアテンションがメモリ帯域幅律速に陥る。オンラインソフトマックスは 2 つの $n \times n$ バッファを近似なしで不要にする手法であり、キーをブロック単位で処理して逐次最大値 m、正規化項 s、値アキュムレータ $\mathbf{o}$ を保持し、その部分和を $e^{m-m'}$ 倍に再スケーリングする（m が増加するたびに行う）。FlashAttention はこのタイル化された漸化式を高速なオンチップ SRAM 内で融合し、オフチップメモリを $O(n)$ に削減して実経過時間で数倍の高速化を達成しつつ、厳密なアテンションとその $\Theta(n^2 d)$ の演算量を維持する。

10.5.4 演習問題

1. **手計算によるオンラインソフトマックス**。1 つのクエリが、2 個ずつ 2 つのブロックに分かれた 4 つのキーと照合されるとする。スコアは第 1 ブロックで $(0, 2)$、第 2 ブロックで $(1, 3)$ であり、値は第 1 ブロックで数値 $1, 2$、第 2 ブロックで $3, 4$ である。
   1. Equation 10.28 をブロックごとに適用せよ。初期値は $m = -\infty$、$s = 0$、$o = 0$ とする。各ブロックの処理後の m、s、o を記録し、除算して出力を求めよ。
   2. 4 つのソフトマックス重みを直接計算し、同じ出力が得られることを確かめよ。
   3. 最後のブロックの処理後に $\ell = m + \log s$ を作り、4 つの重みのそれぞれをそのスコアのみから $e^{a_j - \ell}$ として再計算せよ。これらが前問の重みと一致し、かつ和が 1 になることを確かめ、融合カーネルの逆伝播において ℓ のみを保持すればよい（m と s の代わりに）理由を説明せよ。
2. [code] **逐次最大値**。`chunked_attention` から逐次最大値を取り除き、状態が e^{a_j} の単純な和を累積するように変更せよ。**Q**、**K**、**V** および状態 s、**O** を float16 とし、10 倍にスケールアップしたクエリに対してその結果を実行せよ。
   1. 実行する前に、最大の指数 x（float16 で e^x が有限となるもの）を計算し、幅 64 のクエリとキーが独立な標準正規分布に従う成分をもつとき、スケーリング後のスコアがこの値を超える割合を見積もれ。
   2. 変更した関数を実行し、最初の無限大または未定義値がどの演算で現れるかを報告せよ。

   3. m を保持することで、すべての指数が 0 以下に抑えられる理由を説明せよ。正の最小の float16 数より小さい項 $e^{a_j - m'}$ は 0 に丸められる。そのような項の s への寄与の総和を、最大スコアの項に対する相対値として、n 個のキーからなる 1 行について上から評価せよ。
   4. `chunked_attention` は m をその dtype の最小の有限値で初期化し、スコアも同じ値でマスクする。$n = 2c$（キー c 個ずつのブロック）とし、ループが 2 つのキーブロックを逆順に走査するとする。このとき、最初に走査されるブロックのスコアは、第 1 ブロック内のクエリに対してすべてマスクされる。そのクエリに対する m、s、$\mathbf{O}$ を両方の反復にわたってトレースし、最終出力が依然として厳密であることを示せ。
3. **部分状態のマージ**。スコア a_j と値 $\mathbf{v}_j$ をもつ 1 つのクエリを固定する。キーの集合 B に対し、3 つ組 $(m_B, s_B, \mathbf{o}_B)$（Equation 10.28 が B 内のキーのみから生成するもの）を B の部分状態と呼ぶことにすると、$\mathbf{o}_B / s_B$ は B に制限されたアテンション出力となる。
   1. $B \cup B'$ の部分状態を、互いに素な集合 B と B' の 2 つの部分状態から導出し、Equation 10.28 が B' を単一のブロックとした特別な場合であることを示せ。
   2. このマージ操作が結合的かつ可換であることを示せ。このことから、キーブロックは任意の順序で処理でき、あるいは並列に処理して深さ $\lceil \log_2(n/c) \rceil$ の木（ブロックあたり c 個のキー）でマージできると結論づけ、生成時に単一のクエリが長いキャッシュにアテンションを向ける場面でなぜこれが重要になるかを説明せよ。
   3. 対 (m_B, s_B) を単一の数値 $\ell_B = m_B + \log s_B$ に、3 つ組を $(\ell_B, \bar{\mathbf{o}}_B)$（ただし $\bar{\mathbf{o}}_B = \mathbf{o}_B / s_B$）に置き換えよ。マージ後の ℓ と $\bar{\mathbf{o}}$ をこれらの量で表し、$\bar{\mathbf{o}}$ が $\bar{\mathbf{o}}_B$ と $\bar{\mathbf{o}}_{B'}$ の凸結合であり、その重みが ℓ_B と $\ell_{B'}$ のみに依存することを示せ。
4. **アテンションの演算密度**。n 個のクエリとキー（幅 d_h）および値（幅 d_v）をもつ 1 つのヘッドを考える。2 つの行列積である $\mathbf{QK}^\top$ と重みおよび $\mathbf{V}$ の積には、それぞれ $2n^2 d_h$ 回と $2n^2 d_v$ 回の浮動小数点演算を要し、ソフトマックスには 1 スコアあたり数回の演算を要する。
   1. スコア行列がメモリに 1 回書き込まれてソフトマックスで 1 回読み出され、その出力が 1 回書き込まれて値との積で 1 回読み出されると仮定する（すべて単精度とする）。これら 4 回のパスで移動するバイト数と、$\mathbf{Q}$、$\mathbf{K}$、$\mathbf{V}$ の読み出しおよび出力の書き込みで移動するバイト数を数えよ。
   2. ソフトマックスの演算を無視してバイト数に対する演算回数の比を作り、$n \to \infty$ の極限（d_h と d_v は固定）をとれ。その極限が $(d_h + d_v)/8$ になることを示し、ソフトマックスの演算を含めるとこれがどう変わるかを述べよ。
   3. 入力とスコアを 16 ビット形式とした場合について同じ極限を求めよ。H100 または B200 の 16 ビットのスループットとメモリ帯域幅を Table 13.2 から読み取り、1 バイトあたりの演算回数の比を計算して、その GPU 上で $d_h = d_v = 64$ のアテンションが演算とメモリトラフィックのどちらに律速されるかを判定せよ。
5. **タイル化されたスケジュールのトラフィック**。●演習問題 4a より、素朴な層は主メモリとの間で $4n^2$ 個の浮動小数点数を移動させる（n に関して線形な項を除く。スコア行列と重み行列がそれぞれ 1 回書き込まれ 1 回読み出されるためである）。以下では n に関して線形なトラフィックをすべて無視せよ。
   1. `chunked_attention` が一度に 1 つずつ演算を実行し、各演算が主メモリから入力を読み出して出力を書き戻すとする。n、c、d_h の関数として、スコアストライプのために移動する浮動小数点数、$\mathbf{Q}$ とアキュムレータ $\mathbf{O}$ のために移動する浮動小数点数、および起動される演算の回数を数えよ。素朴な層と比べて、チャンク分割はこれらの量のうちどれを削減するか。
   2. 一方、融合カーネルは b 個のクエリからなるタイル、それらの出力アキュムレータ、および b 個のキーと b 個の値からなるタイルを各 1 つずつ、容量 M 個（浮動小数点数換算）の高速メモリ内に保持し、$4bd_h \le M$ を満たすとする。タイルのスコアおよび逐次統計量 m と s も高速メモリ内に留まる。このカーネルは各クエリタイルを 1 回読み込み、すべてのキータイルと値タイルをそこへ順にストリーミングし、各出

力タイルを 1 回書き出す。線形な項を除いたトラフィックが $2n^2 d_h/b$ 個の浮動小数点数になること、そして許容される最大のタイルをとると $\Theta(n^2 d_h^2/M)$（Dao et al. (2022) が FlashAttention に対して証明した上界）が得られることを示せ。

3. $d_h \in \{64, 128, 256\}$、$M = 2^{16}$ に対して素朴な層のトラフィックに対する融合カーネルのトラフィックの比を計算し、ヘッド幅がタイリングの恩恵をどのように制限するかを述べよ。

10.6 ウィンドウアテンションと線形アテンション

融合カーネル（fused kernel、10.5 節）は $O(n^2)$ の中間スコア行列を不要にするが、$\Theta(n^2 d)$ の FLOP 数とキー・値キャッシュはいずれも依然として系列長 n に伴って増大する。しかし、100,000 トークンの文書を処理する場合、大半のトークンは主に局所的な文脈に依存し、長距離の検索が必要になるのは時折生じる参照に限られる。

アテンションの計算コストを $O(n)$ に削減する主要なアーキテクチャ上の戦略は 2 つある。

1. **スライディングウィンドウアテンション** (Child et al. 2019; Beltagy et al. 2020) は、各クエリの参照先を直近 w 個のキーに制限することで、1 層あたりの計算量を $O(nw)$ に削減し、キー・値キャッシュを w トークンに抑える。
2. **線形アテンション**（linear attention、(Katharopoulos et al. 2020)）は、ソフトマックスカーネル $\exp(\mathbf{q}^\top \mathbf{k}/\sqrt{d_h})$ を積に分解されたカーネル $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ に置き換えることで、キーと値を固定サイズ $d_h \times d_v$ の再帰状態行列に累積できるようにする。

いずれの手法も、密なソフトマックスアテンションとは異なる関数を計算する。本節では、スライディングウィンドウが 1 ホップの到達範囲と引き換えにどのようにタイルスパース性を得るかを分析し、続いて線形アテンションの並列形式・再帰形式・チャンク形式を導出したうえで、分解された特徴マップがソフトマックスのような先鋭化の温度をもたない理由を示す。

```
from torch.nn.attention.flex_attention import create_block_mask, flex_attention
```

10.6.1 スライディングウィンドウアテンション

10.6.1.1 帯マスク

幅 w の因果的な**スライディングウィンドウアテンション**では、クエリ i はキー j に対し、$j \le i$ かつ $i - j < w$ の場合に限りアテンションを向ける。因果性と局所性を組み合わせたこの条件により、$n(n+1)/2$ 個の有効ペアからなる下三角領域は、高々 nw 個のペアからなる対角帯に置き換わる。

```
def window_mask(n, w, device=None):
    """Causal band: query i may attend to keys i - w + 1, ..., i."""
    i = torch.arange(n, device=device)
    return d2l.causal_mask(n, device=device) & (i[:, None] - i[None, :] < w)
```

10.6.1.2 深さに伴う受容野の拡大

畳み込みを積み重ねると深さに伴って受容野が拡大するのと同様に（第 6 章）、L 層の幅 w のスライディングウィンドウ層を積み重ねると、情報は 1 層あたり $w-1$ ステップ後方へ伝播でき、理論的受容野（theoretical receptive field）は次の式で与えられる。

$$1 + L(w-1) \tag{10.29}$$

これが到達可能なトークン数である。たとえば、Mistral 7B v0.1 (Jiang et al. 2023) は $w = 4{,}096$ を $L = 32$ 層にわたって用いており、最大 131,041 トークンをカバーする。しかし、理論上の到達範囲と直接的な検索は異なる。完全な自己アテンションが任意の 2 トークンを 1 ホップで結ぶのに対し、ウィンドウの積層では層をまたぐ $\lceil \Delta/(w-1) \rceil$ 回の逐次ホップを経てようやく距離 Δ 先の事実を中継する。さらに、中継される事実はホップのたびに希釈される。最小の例を Figure 10.11 に示す。一様なウィンドウ平均（$w = 4$、$L = 4$）のもとでは、境界 $\Delta = 12$ にあるトークンは単一の 4 ホップ経路を通って重み $(1/4)^4 = 1/256$ で到達するが、単一の大域層ではこれが $1/13$ となる。

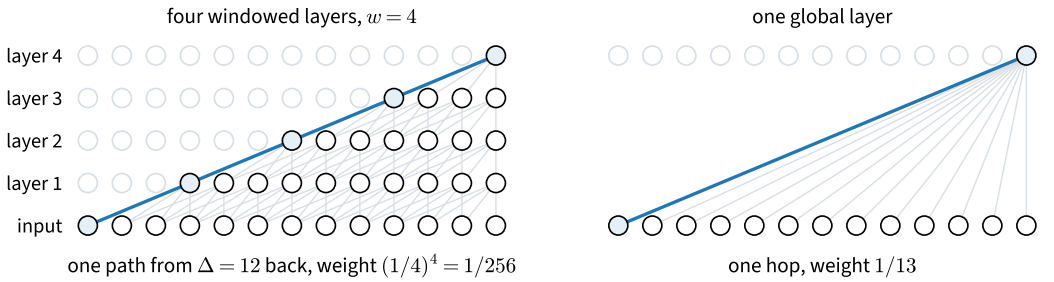


Figure 10.11: 積層ウィンドウと単一の大域層の比較。左：幅 $w = 4$ のウィンドウ層を 4 層重ねた場合、右上の出力は過去の位置からなる錐状の領域に依存し、$\Delta = 12$ 位置前の入力は 4 ホップの単一経路（青）のみを通って到達するため、一様なアテンション重みのもとでの重みは $(1/4)^4 = 1/256$ となる。右：単一の大域層は同じ 2 つの位置を 1 ホップ、重み $1/13$ で結ぶ。

10.6.1.3 大域層

局所ウィンドウによるメモリと計算量の削減効果を大部分維持しつつ、1 ホップの長距離検索を保つため、現代の長文脈（long-context）アーキテクチャは r 個のウィンドウ層と 1 つの完全な**大域**（global）アテンション層を交互に配置する。たとえば Gemma 3 は、大域層 1 つにつき $r = 5$ 個のウィンドウ層（$w = 1{,}024$）を用いている (Gemma Team 2025)。生成時、ウィンドウ層は直近 $\min(n, w)$ 個のキーと値のみをキャッシュするため（11.3 節）、この交互配置により KV キャッシュ全体の大きさは次の比率に削減される。

$$\frac{r\,\min(n, w) + n}{(r+1)\,n} \xrightarrow{\;n \gg w\;} \frac{1}{r+1}$$

これは全層を大域層とした構成に対する比率であり、$\approx 27\%$（$n = 8{,}192$ のとき）、漸近的には $1/6$（$r = 5, w = 1{,}024$ のとき）となる。

10.6.1.4 融合カーネルにおけるタイル

密な $n \times n$ 行列積の内部で `window_mask` を適用しても、依然として n^2 個すべてのスコアが計算されてしまう。$O(nw)$ の時間とメモリを実現するには、有効な対角帯のみを評価する実装が必要である。たとえば、クエリをサイズ w のブロックにまとめ、現在のキーブロックと直前のキーブロックのみにアテンションを向ければよい（合計 $2nw$ 個のスコア）。

```
def windowed_attention(Q, K, V, w):
    """Causal sliding-window attention in O(nw) time and memory."""
    n, d_h = Q.shape
    assert n % w == 0, 'blocks must tile the sequence exactly'
    Qb, Kb, Vb = (X.reshape(n // w, w, -1) for X in (Q, K, V))
    # Each query block attends to its own key block and the one before it
    prev = lambda X: torch.cat([torch.zeros_like(X[:1]), X[:-1]])
    Kb, Vb = torch.cat([prev(Kb), Kb], 1), torch.cat([prev(Vb), Vb], 1)
    i = torch.arange(w, device=Q.device)[:, None]      # Query in its block
    j = torch.arange(-w, w, device=Q.device)[None, :]  # Key offset from it
    first = torch.arange(n // w, device=Q.device)[:, None, None] == 0
    mask = (j <= i) & (i - j < w) & ~(first & (j < 0))  # No keys before 0
    scores = Qb @ Kb.transpose(-1, -2) / math.sqrt(d_h)
    return (d2l.masked_softmax(scores, mask) @ Vb).reshape(n, -1)
```

FlashAttention や FlexAttention (Dong et al. 2025) のようなタイル化 GPU カーネルでは、計算は $b \times b$ のタイル単位（FlexAttention のデフォルトでは $b = 128$）で進み、タイル内の b^2 個の要素が**すべて**マスクされている場合に限りそのタイルがスキップされる。$n = 2{,}048$ かつ $b = w = 128$ のとき、因果マスクは $T(T+1)/2$ 個（全 T^2 タイル中 53.1%、ただし $T = n/b = 16$）を訪問するのに対し、スライディングウィンドウは各タイル行の対角タイルとその左隣のみを訪問する（$2T - 1 = 31$ タイル、すなわち 12.1%、Figure 10.12）。

```
def tiles_visited(mask, b):
    """Fraction of b-by-b tiles of a mask that hold at least one True."""
    T = mask.shape[0] // b
    return mask.reshape(T, b, T, b).any(dim=(1, 3)).float().mean().item()

n, b, w = 2048, 128, 128
print(f'causal:  {tiles_visited(d2l.causal_mask(n), b):.1%} of tiles')
print(f'window:  {tiles_visited(window_mask(n, w), b):.1%} of tiles')
```

```
causal:  53.1% of tiles
window:  12.1% of tiles
```

`windowed_attention` とフレームワークの融合スライディングウィンドウカーネルはいずれも、密行列にマスクを適用する定義と float32 の丸め誤差の範囲内で一致する。

融合カーネルには、`flex_attention` と `create_block_mask` を用いる。

```
n, w, d_h = 2048, 128, 64
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
reference = d2l.masked_softmax(Q @ K.T / math.sqrt(d_h),
                               window_mask(n, w, Q.device)) @ V
# create_block_mask evaluates the band at every position and records which
# tiles are empty, full or cut by it; compiled, the kernel skips the empty ones
band = lambda batch, head, q, k: (k <= q) & (q - k < w)
block_mask = create_block_mask(band, None, None, n, n, device=Q.device)
fused = torch.compile(flex_attention)(Q[None, None], K[None, None],
                                      V[None, None], block_mask=block_mask)
for name, O in (('blocked', windowed_attention(Q, K, V, w)),
                ('fused kernel', fused[0, 0])):
```

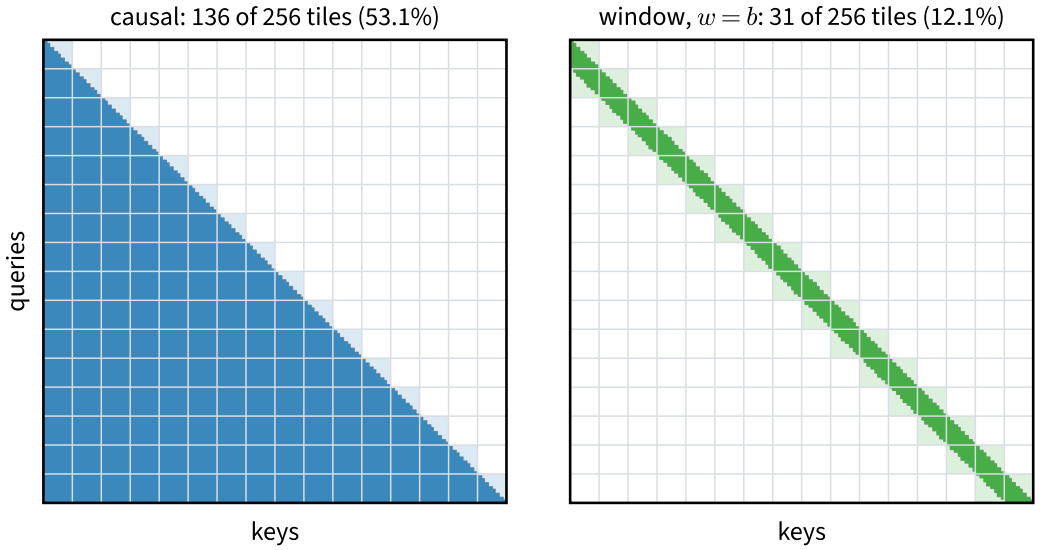


Figure 10.12: 融合カーネルが訪問するタイル（タイル行数 $T = 16$ の場合）。濃い要素は許可されたスコアを表し、正方形内の薄い残余部分はマスク境界で切断されるもののカーネルが訪問せざるを得ないタイルを示す。因果マスク（左）は $T(T+1)/2 = 136$ 個のタイルを訪問し、切断が生じるのは対角上の 16 個のみである。幅 $w = b$ のウィンドウ（右）は各行の対角タイルとその左隣、計 $2T - 1 = 31$ 個のタイルを訪問し、そのすべてがマスク境界で切断される。

```
    print(f'{name:>12}: maximum deviation '
          f'{float((O - reference).abs().max()):.2e}')
```

```
     blocked: maximum deviation 4.17e-07
fused kernel: maximum deviation 6.56e-07
```

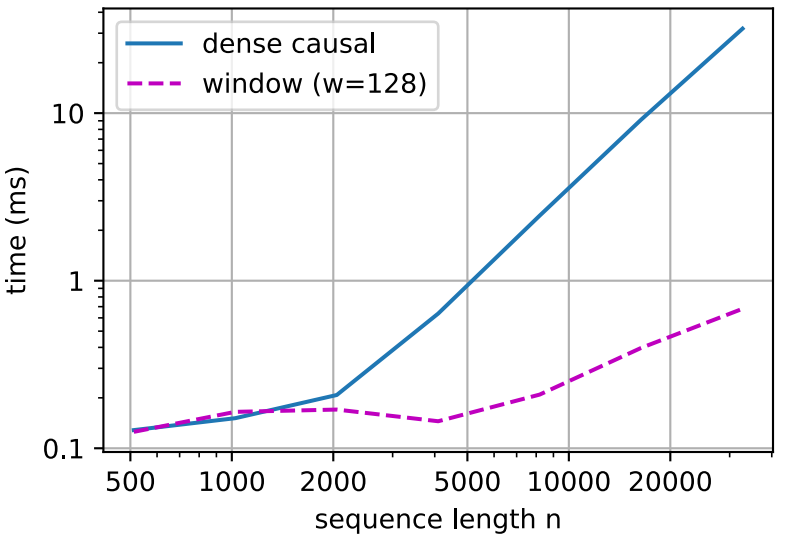
10.6.1.5 ウィンドウカーネルと密カーネルの比較

タイル行数を $T = n/128$ とすると、有効な因果タイル数と有効なウィンドウタイル数（$w = b = 128$）の比は $T(T+1)/(2(2T-1)) \approx T/4$ となり、理論上のタイル削減率は $64\times$（$n = 32{,}768$、すなわち $T = 256$ のとき）と予測される。同一の bfloat16 融合カーネル内で、両方のマスクを $n \in [512, 32{,}768]$ にわたってベンチマークしよう（ヘッド数 $H = 32$、$d_h = 64$）。

```
flex = torch.compile(flex_attention)
w = b = 128  # Window width; tile size of the kernel
masks = {'dense causal': lambda batch, head, q, k: k <= q,
         f'window (w={w})': lambda batch, head, q, k: (k <= q) & (q - k < w)}
H, lengths = 32, [512, 1024, 2048, 4096, 8192, 16384, 32768]
times = {name: [] for name in masks}
for n in lengths:
    X = torch.randn(1, H, n, d_h, device=d2l.try_gpu(), dtype=torch.bfloat16)
    for name, rule in masks.items():
        block_mask = create_block_mask(rule, None, None, n, n, device=X.device)
        times[name].append(d2l.Benchmark(
```

```
            lambda: flex(X, X, X, block_mask=block_mask)).time * 1e3)
    T = n // b  # Rows of b x b tiles; causal T(T+1)/2, window 2T - 1
    dense, window = (t[-1] for t in times.values())
    print(f'n = {n:6,}: window {dense / window:4.1f}x faster, '
          f'tile count predicts {T * (T + 1) / 2 / (2 * T - 1):4.1f}x')
d2l.plot(lengths, list(times.values()), 'sequence length n', 'time (ms)',
         legend=list(times), xscale='log', yscale='log', figsize=(4.5, 3))
```

```
n =    512: window  1.0x faster, tile count predicts  1.4x
n =  1,024: window  0.9x faster, tile count predicts  2.4x
n =  2,048: window  1.2x faster, tile count predicts  4.4x
n =  4,096: window  4.4x faster, tile count predicts  8.4x
n =  8,192: window 11.7x faster, tile count predicts 16.4x
n = 16,384: window 23.0x faster, tile count predicts 32.4x
n = 32,768: window 47.0x faster, tile count predicts 64.4x
```



$n \approx 2{,}048$ を超えると、密な因果アテンションの実行時間が二次関数的に増大するのに対し、ウィンドウアテンションの実行時間は線形に増大し、$> 30\times$ の高速化（$n = 32{,}768$ のとき）に達する。これはタイル数から予測される $64\times$ の 1/2 から 3/4 の間にあたる。タイル数による比較は訪問されるすべてのタイルを同一の計算量とみなすため、得られる高速化の上界ではあっても推定値ではない。

FlexAttention では、幅 b の帯で訪問されるすべてのタイルがマスク境界によって切断されるため（Figure 10.12）、要素ごとのマスク処理のコストが生じるのに対し、因果マスクの下三角内部にあるタイルはマスク処理なしで実行される。

10.6.2 線形アテンション

10.6.2.1 スコアのカーネル化

どのペアのスコアを計算するかを制限する代わりに、**線形アテンション** (Katharopoulos et al. 2020) は指数類似度 $\exp(\mathbf{q}^\top \mathbf{k}/\sqrt{d_h})$ を積に分解されたカーネル $K(\mathbf{q}, \mathbf{k}) = \phi(\mathbf{q})^\top \phi(\mathbf{k})$ に置き換える。ここで $\phi : \mathbb{R}^{d_h} \to \mathbb{R}_+^{d_h}$ は要素ごとの非負の特徴マップである。行列積の結合法則によ

り、次が成り立つ。

$$\left(\phi(\mathbf{Q})\,\phi(\mathbf{K})^\top\right)\mathbf{V} = \phi(\mathbf{Q})\left(\phi(\mathbf{K})^\top\mathbf{V}\right),$$

これにより、$n \times n$ 行列 $\phi(\mathbf{Q})\phi(\mathbf{K})^\top$ の構築を避け、代わりに $d_h \times d_v$ の要約行列を計算すれば済む。因果マスク（$s \le t$）のもとで、分子と分母の和から $\phi(\mathbf{q}_t)^\top$ をくくり出すと次式を得る。

$$\begin{aligned}\mathbf{o}_t^\top &= \frac{\sum_{s\le t}\phi(\mathbf{q}_t)^\top\phi(\mathbf{k}_s)\,\mathbf{v}_s^\top}{\sum_{s\le t}\phi(\mathbf{q}_t)^\top\phi(\mathbf{k}_s)} = \frac{\phi(\mathbf{q}_t)^\top\mathbf{S}_t}{\phi(\mathbf{q}_t)^\top\mathbf{z}_t},\\ \mathbf{S}_t &= \sum_{s\le t}\phi(\mathbf{k}_s)\mathbf{v}_s^\top, \qquad \mathbf{z}_t = \sum_{s\le t}\phi(\mathbf{k}_s).\end{aligned} \tag{10.30}$$

Katharopoulos et al. (2020) に従い、要素ごとに $\phi(x) = \mathrm{elu}(x) + 1 > 0$ と選ぶ（float32 のアンダーフローを防ぐため、分母には小さな $\epsilon = 10^{-6}$ を加える）。プレフィックス和 $(\mathbf{S}_t, \mathbf{z}_t)$ は次の線形漸化式（linear recurrence）を満たす。

$$\begin{aligned}\mathbf{S}_t &= \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\,\mathbf{v}_t^\top,\\ \mathbf{z}_t &= \mathbf{z}_{t-1} + \phi(\mathbf{k}_t),\end{aligned} \tag{10.31}$$

これにより、因果アテンションは、外積（outer product）によって更新される $d_h \times d_v$ の連想メモリ行列を隠れ状態にもつ RNN、すなわち**高速重みプログラマ**（fast weight programmer、(Schlag et al. 2021)）へと変換される。Equation 10.30 は、累積和（cumsum）を用いて並列に評価することも、$O(d_h d_v)$ のメモリで 1 ステップずつ再帰的に評価することもできる。

```
def phi(X):
    return F.elu(X) + 1

def linear_attention_parallel(Q, K, V, eps=1e-6):
    """Training form: all outputs at once via cumulative sums."""
    Qp, Kp = phi(Q), phi(K)
    S = torch.cumsum(Kp[:, :, None] * V[:, None, :], dim=0)  # (n, d_h, d_v)
    z = torch.cumsum(Kp, dim=0)                              # (n, d_h)
    den = (Qp * z).sum(-1, keepdim=True) + eps
    return (Qp[:, :, None] * S).sum(1) / den

def linear_attention_recurrent(Q, K, V, eps=1e-6):
    """Generation form: a (d_h, d_v) state, updated token by token."""
    Qp, Kp = phi(Q), phi(K)
    S = torch.zeros(Q.shape[-1], V.shape[-1], device=Q.device)
    z = torch.zeros(Q.shape[-1], device=Q.device)
    outputs = []
    for t in range(Q.shape[0]):
        S = S + Kp[t][:, None] * V[t][None, :]   # Outer-product write
        z = z + Kp[t]
        outputs.append(Qp[t] @ S / (Qp[t] @ z + eps))  # Query the memory
    return torch.stack(outputs)
```

10.6.2.2 チャンク単位の計算

GPU 上での訓練時、n 個すべての中間行列 $\mathbf{S}_t \in \mathbb{R}^{d_h \times d_v}$ を `linear_attention_parallel` で実体化すると $O(nd_hd_v)$ のメモリ（$n = 16{,}384, d_h = d_v = 64$ のときヘッドあたり 256 MiB）を消費する一方、`linear_attention_recurrent` は n 回の逐次的なベクトル演算ステップを実行するため GPU の行列演算コアを十分に活用できない。系列を長さ c のチャンクに分割する手法 (S. Yang, Wang, Shen, et al. 2024) は両者の利点を兼ね備える。位置 t_0 から始まりチャンク前の状態 $(\mathbf{S}, \mathbf{z})$ をもつチャンクに対しては、次式が成り立つ。

$$\mathbf{o}_t^\top = \frac{\phi(\mathbf{q}_t)^\top \mathbf{S} + \sum_{t_0 \le s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)\, \mathbf{v}_s^\top}{\phi(\mathbf{q}_t)^\top \mathbf{z} + \sum_{t_0 \le s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)}, \tag{10.32}$$

したがって、チャンク内のアテンション（$c \times c$）とチャンク間の状態伝播はいずれも $O(nc(d_h + d_v) + nd_hd_v)$ FLOPs の密行列積を用い、保存するチャンク状態も n/c 個だけで済む。

```
def linear_attention_chunked(Q, K, V, c=64, eps=1e-6):
    """Training form in chunks: quadratic within, recurrent across."""
    n, d_h = Q.shape
    assert n % c == 0, 'chunks must tile the sequence exactly'
    Qp, Kp = phi(Q).reshape(n // c, c, d_h), phi(K).reshape(n // c, c, d_h)
    Vc = V.reshape(n // c, c, -1)
    # The state each chunk starts from: sums over all earlier chunks
    S = torch.cumsum(Kp.transpose(-1, -2) @ Vc, dim=0)   # (n/c, d_h, d_v)
    z = torch.cumsum(Kp.sum(1), dim=0)                   # (n/c, d_h)
    S = torch.cat([torch.zeros_like(S[:1]), S[:-1]])
    z = torch.cat([torch.zeros_like(z[:1]), z[:-1]])
    A = torch.tril(Qp @ Kp.transpose(-1, -2))            # (n/c, c, c)
    num = Qp @ S + A @ Vc                                # Past + this chunk
    den = Qp @ z[..., None] + A.sum(-1, keepdim=True) + eps
    return (num / den).reshape(n, -1)
```

3 つの形式はいずれも float32 の丸め誤差の範囲内でまったく同じ出力を計算し、再帰状態は系列長によらずヘッドあたり一定の 16 KiB（float で $64 \times 64 + 64$ 個）しか占有しない。

```
n = 512
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
recurrent = linear_attention_recurrent(Q, K, V)
for name, f in (('parallel', linear_attention_parallel),
                ('chunked', linear_attention_chunked)):
    err = (f(Q, K, V) - recurrent).abs().max()
    print(f'{name:>8}: maximum deviation {float(err):.2e}')
print(f'recurrent state: {d_h}x{d_h} + {d_h} floats '
      f'= {(d_h * d_h + d_h) * 4 / 1024:.0f} KiB at any sequence length')
```

```
parallel: maximum deviation 4.77e-07
 chunked: maximum deviation 4.77e-07
recurrent state: 64x64 + 64 floats = 16 KiB at any sequence length
```

10.6.2.3 容量と鋭さ

任意個のキー・値ペアを $d_h \times d_v$ の状態行列に圧縮することには、2 つの本質的なコストが生じる。

1. **有限の連想容量：** $\phi(\mathbf{k}_s) \in \mathbb{R}_+^{d_h}$ は非負であるため、クエリがクロストークなしに保存された単一の値を取り出せるのは、キーの特徴ベクトル同士がほぼ直交する（台がほぼ互いに素である）場合に限られ、厳密な検索は高々 d_h 個のキーに制限される。
2. **温度による先鋭化の欠如：** ソフトマックスアテンションでは、クエリを $\beta > 1$ 倍にスケーリングするとスコア差が指数的に増幅され（$e^{\beta\Delta}$）、$\beta \to \infty$ で分布は one-hot 参照へと収束する。一方、$\phi(x) = \mathrm{elu}(x) + 1$ を用いる線形アテンションでは、$\mathbf{q}$ を大きな β でスケーリングすると $\phi(\beta\mathbf{q}) \approx \beta\,\mathrm{relu}(\mathbf{q})$ となるため、スケール因子 β が分子と分母で相殺される（演習問題 2）。

```
Q = torch.randn(1000, d_h, device=d2l.try_gpu())
K = torch.randn(8, d_h, device=d2l.try_gpu())
for beta in (1, 4, 16):  # Scale the queries up; 8 keys: uniform is 0.125
    soft = torch.softmax(beta * Q @ K.T / math.sqrt(d_h), dim=-1)
    lin = phi(beta * Q) @ phi(K).T
    lin = lin / lin.sum(-1, keepdim=True)
    print(f'queries x{beta:<2}: mean largest weight, softmax '
          f'{soft.amax(-1).mean():.2f}, elu+1 {lin.amax(-1).mean():.2f}')
```

```
queries x1 : mean largest weight, softmax 0.36, elu+1 0.15
queries x4 : mean largest weight, softmax 0.76, elu+1 0.15
queries x16: mean largest weight, softmax 0.94, elu+1 0.15
```

10.6.2.4 各機構における想起

これら 2 つの限界とウィンドウの限界は、考えられる最も単純な参照タスクに現れる。N 個の幅 $d_h = 64$ のランダムなキー・値ペアを保存し、保存されたキーの 1 つをクエリとして与え、出力が他のどの保存された値よりも正しい値と大きな内積をもつときにヒットと数える。このクエリに 4 つの機構で応答する。すなわち、密なソフトマックスアテンション、直近 $w = 16$ 個のペアのみを見るウィンドウ、elu+1 特徴マップを用いた線形アテンション、そして恒等特徴マップを用い正規化項をもたない線形アテンションである**外積状態**（outer-product state）$\mathbf{q}^\top \sum_i \mathbf{k}_i \mathbf{v}_i^\top$ である。最後の機構は特徴マップの影響を切り離すことで、$d_h \times d_v$ の状態そのものがもつ容量を浮き彫りにする。

```
def recall(N, w=16, trials=1000):
    """Look up one of N stored pairs by its key; fraction of hits per rule."""
    K, V = (torch.randn(trials, N, d_h, device=d2l.try_gpu())
            for _ in range(2))
    target = torch.randint(0, N, (trials,), device=K.device)
    rows = torch.arange(trials, device=K.device)
    q = K[rows, target][:, None]                        # (trials, 1, d_h)
    scores = q @ K.transpose(1, 2) / math.sqrt(d_h)     # (trials, 1, N)
    recent = torch.arange(N, device=K.device) >= N - w  # The last w pairs
    lin = phi(q) @ phi(K).transpose(1, 2)
    outputs = {'softmax': torch.softmax(scores, -1) @ V,
               f'window (w={w})': d2l.masked_softmax(scores, recent) @ V,
               'elu+1': lin / lin.sum(-1, keepdim=True) @ V,
```

```
                'outer-product state': q @ (K.transpose(1, 2) @ V)}
    # A hit: of the N stored values, the target is nearest to the output
    return {name: float(((O @ V.transpose(1, 2)).argmax(-1)[:, 0] == target)
                        .float().mean()) for name, O in outputs.items()}

for N in (16, 64, 256, 1024):
    print(f'N = {N:4}: ' + ', '.join(f'{name} {hit:.2f}'
                                     for name, hit in recall(N).items()))
```

```
N =   16: softmax 1.00, window (w=16) 1.00, elu+1 0.24, outer-product state 1.00
N =   64: softmax 1.00, window (w=16) 0.24, elu+1 0.04, outer-product state 0.99
N =  256: softmax 1.00, window (w=16) 0.07, elu+1 0.00, outer-product state 0.67
N = 1024: softmax 1.00, window (w=16) 0.02, elu+1 0.00, outer-product state 0.09
```

ソフトマックスアテンションはあらゆる N で目的のペアを見つけ出す。ウィンドウは目的のペアが直近 w 個に含まれる場合にのみ見つけられるため、その想起率は約 w/N となり、$N = 16$ では完全、$N = 64$ では 1/4、$N = 1{,}024$ では数パーセントにとどまる。さらに際立つのが elu+1 線形アテンションであり、$N = 16$、すなわち保存すべきペア数の 4 倍の行数を状態がもつ場合でさえ失敗する。重みを鋭くできないため、出力は保存されたすべての値の平均に近くなり、正しい値がほとんど際立たないのである。

外積状態ははるかに良好に機能し、容量の限界を鮮明に示す。クエリ $\mathbf{q} = \mathbf{k}_j$ に対する出力は、$\|\mathbf{k}_j\|^2 \mathbf{v}_j \approx d_h \mathbf{v}_j$ に他のペアからのクロストーク $\sum_{i \neq j} (\mathbf{k}_j^\top \mathbf{k}_i)\, \mathbf{v}_i$ を加えたものになる。各クロストーク係数の大きさは約 $\sqrt{d_h}$ であるため、信号に対するノイズの比は約 $\sqrt{(N-1)/d_h}$ となる。想起率は $N \approx d_h$ まではほぼ完全であるが、$N = 4d_h$ で約 2/3 に低下し、$N = 16d_h$ では崩壊する。ソフトマックスアテンションは N 個すべてのペアを保持してあらゆる N で想起できるのに対し、固定サイズの状態は N がその幅の付近にとどまるあいだしか想起できない。

10.6.3 機構の選び方

- **厳密な融合アテンション**は短〜中程度の文脈におけるデフォルトであり、$O(n)$ の作業メモリで厳密な 1 ホップ検索を実現する。
- **スライディングウィンドウ層と大域層の交互配置**は長文脈 Transformer（例：Gemma 3）の標準的な構成であり、局所層での KV キャッシュメモリとタイルあたりの FLOP 数を削減しつつ、周期的な大域層で 1 ホップの大域ルーティングを維持する。
- **線形アテンション**で厳密な連想想起（associative recall）を扱うには、データ依存の減衰ゲート（古い項目を忘却できるようにするため。演習問題 4 および第 12 章を参照）と、完全なソフトマックス層とのハイブリッドな交互配置（例：MiniMax-01、(MiniMax 2025)）が必要になる (Arora, Eyuboglu, Zhang, et al. 2024; S. Yang, Wang, Shen, et al. 2024)。

10.6.4 まとめ

スライディングウィンドウアテンションは、各クエリの参照先を直近 w 個のキーに制限することで、演算量を $O(nw)$、キャッシュメモリを $O(w)$ に削減しつつ、有効受容野を $1 + L(w-1)$ トークン（L 層積層時）へと拡大する。現代の長文脈モデルは、1 ホップ検索を維持するためにこれらの局所層と周期的な大域アテンション層を交互に配置する。GPU 上では、タイル化カーネルが完全にマスクされた $b \times b$ のスコアタイルをスキップすることで、境界でのマスク処理のオーバーヘッドを除きこの高速化を実現する。一方、線形アテンションは代数的なアプローチをとり、指数スコアを積に分解された類似度カーネル $\phi(\mathbf{q})^\top \phi(\mathbf{k})$ に置き換えることで、因果アテンションを、並列チャンクで訓練可能な $O(n)$ の漸化式（固定サイズ $d_h \times d_v$ の行列状態をもつ）へと

帰着させる。非負の特徴量をもつ固定サイズの状態では、クエリのスケーリングによって重みを鋭くすることも、クロストークなしに $O(d_h)$ 個を超える直交キーを保持することもできないため、実用的な線形アテンションのアーキテクチャはデータ依存の減衰ゲートとハイブリッドなソフトマックス層を組み込んでいる。

10.6.5 演習問題

1. [code] **融合カーネルのタイル**。融合カーネルは $b \times b$ 個のスコアからなるタイル単位で動作し、マスクがそのタイル全体を除外する場合に限りタイルをスキップする。$T = n/b$ とする。
   1. 因果マスクが $T(T+1)/2$ 個のタイル（全 T^2 タイル中）を残すこと、および幅 $w = kb$（整数 $k < T$）のウィンドウが $T(k+1) - k(k+1)/2$ 個のタイルを残すことを示せ。両方の個数を `tiles_visited` で確認せよ。w が b の倍数でないとき、個数はどのように変化するか。
   2. 複数の文書が長さ $n = 2048$ トークンの 1 つの系列にパックされており、10.2 節のマスクと同様に、マスクが因果性と「同一文書」の条件を組み合わせているとする。$b = 128$ で `tiles_visited` を用い、512 トークンの 4 つの文書を、1,000、600、300、148 トークンの文書と比較せよ。文書の境界がタイルの境界と一致しない場合により多くのタイルを要する理由を説明せよ。
2. [code] **特徴マップ**。`phi` で用いた特徴マップ $\phi(x) = \mathrm{elu}(x) + 1$ は数ある選択肢の 1 つにすぎない。いずれも要素ごとに適用される $\phi(x) = \mathrm{relu}(x)$ および $\phi(x) = \exp(x)$ と、$n = 512$ 個の幅 `d_h` のランダムなクエリとキー上で比較せよ。
   1. 各マップについて、要素が次式で与えられる因果線形アテンションの重み行列を計算せよ。
   $$\frac{\phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)}{\sum_{s' \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_{s'})}, \qquad s \le t,$$
   さらに、因果ソフトマックス重み行列 `d2l.masked_softmax(Q @ K.T / math.sqrt(d_h), d2l.causal_mask(n))` との最大絶対差、および各行のエントロピーの平均を測定せよ。
   2. $\mathbf{q}_t$ とキー $\mathbf{k}_1, \dots, \mathbf{k}_t$ がどのような条件を満たすとき、ReLU マップは未定義の行を生じるか。すべての要素が独立な標準正規分布に従うとき、第 t 行でこの事象が起こる確率を導出し、$t = 1$ について幅 64 と幅 4 のそれぞれで値を求めよ。
   3. 本節ではクエリを $\beta \in \{1, 4, 16\}$ 倍して最大の重みを観察した。ReLU マップと指数マップに対して同様の実験を行い、代わりに各行の平均エントロピーを測定せよ。3 つのマップのそれぞれについて $\beta \to \infty$ としたときの正規化重みの極限を導出し、ソフトマックスが one-hot になる状況でいずれも one-hot にならない理由を説明せよ。
3. **厳密な特徴マップ**。●ソフトマックスカーネル $K(\mathbf{q}, \mathbf{k}) = \exp(\mathbf{q}^\top \mathbf{k}/\sqrt{d_h})$ も積に分解できるが、そのためには無限個の特徴量が必要になる。
   1. K をテイラー級数に展開し、p 次の項が $\phi_p(\mathbf{q})^\top \phi_p(\mathbf{k})$ に等しいことを示せ（ここで ϕ_p は次元 d_h^p の特徴マップであり、引数の p 重テンソル積から構成される）。
   2. 級数を 2 次の項までで打ち切れ。打ち切ったカーネルが任意の $\mathbf{q}$ と $\mathbf{k}$ の組に対して正であり、したがって Equation 10.30 の正規化項が 0 になり得ないことを示したうえで、ϕ_0、ϕ_1、ϕ_2 を連結した特徴マップの次元を求めよ。$d_h = 64$ かつ値の幅が 64 のとき、単精度における状態 $(\mathbf{S}, \mathbf{z})$ のサイズと、ソフトマックスヘッドのキー・値キャッシュが同じサイズに達する系列長 n を計算せよ。この種の打ち切りは Based (Arora, Eyuboglu, Zhang, et al. 2024) の特徴マップとして用いられている。
   3. $n \times n$ 行列（要素 $K(\mathbf{x}_i, \mathbf{x}_j)$）が相異なる $\mathbf{x}_1, \dots, \mathbf{x}_n$ に対して正定値であることは既知としてよい。クエリとキーが一致するとき、次元 $D < n$ のいかなる特徴マップもこれらの点上で K を厳密には再現できないことを示せ。

4. [code] **減衰する状態。**●Equation 10.31 の状態にスカラー減衰 $\gamma \in (0,1)$ を導入する。

$$\mathbf{S}_t = \gamma\,\mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\,\mathbf{v}_t^\top, \qquad \mathbf{z}_t = \gamma\,\mathbf{z}_{t-1} + \phi(\mathbf{k}_t). \tag{10.33}$$

   1. 並列形式において、位置 s が位置 t の出力に与える寄与が γ^{t-s} で重み付けされることを示し、`linear_attention_parallel` の累積和を γ のべき乗でリスケーリングすることでこれを実装せよ。
   2. `linear_attention_recurrent` に減衰を追加し、$n = 512$ かつ $\gamma = 0.99$ で 2 つの形式が一致することを検証せよ。
   3. このリスケーリングは累積和の前に位置 s に γ^{-s} を掛ける。単精度でオーバーフローが生じるのはどのような n のときか。$\gamma = 0.9$ と $\gamma = 0.5$ のそれぞれについて求めよ。
   4. `linear_attention_chunked` に減衰を追加せよ。すなわち、各位置をそのチャンクの先頭を基準としてリスケーリングし、減衰付きの漸化式でチャンクから次のチャンクへと状態を引き継げ。γ^{-c} より大きな因子が現れないことを示したうえで、$\gamma = 0.5$、$n = 512$、$c = 64$ で実行し、再帰形式と比較せよ。
   5. 重み γ^{t-s} が 0.01 を下回るラグ（$\gamma = 0.99$ のとき）を計算せよ。あるタスクでは、2000 位置前にある 1 つのトークンを想起する必要があり、そのあいだのトークンはすべてフィラー（埋め草）であるとする。固定された γ ではそのトークンとフィラーの双方にうまく対処できない理由、および各トークン自身の入力から計算される減衰であれば代わりに何ができるかを説明せよ。

10.7 アテンション回路

10.1 節で述べたように、アテンションのヒートマップは層が何を計算しているかについての仮説であり、証明ではない。ヒートマップが示すのはヘッドがどこを見るかという点であり、それは計算の半分にすぎない。ヘッドが実際に出力へ何を書き込むかは、その値射影と出力射影に依存する。あるトークンに鋭くアテンションを向けるヘッドであっても、そのトークンの同一性をコピーする場合もあれば、部分空間の単一の特徴量を抽出する場合や、無視できるほど小さなベクトルしか寄与しない場合もある。

訓練済みのアテンションヘッドが何を計算するかを理解するため、本節では Elhage et al. (2021) による機構的回路（mechanistic circuit）の定式化を、アテンションのみの言語モデル `TinyCharLM`（10.4 節）に適用する。各ヘッドは、**残差ストリーム**（residual stream）上のランク d_h の 2 つの双線形・線形作用素へと分解される。すなわち、どの位置にアテンションを向けるかを決める **QK 回路**（QK circuit）と、アテンションを向けられたベクトルが出力ロジットをどう変化させるかを決める **OV 回路**（OV circuit）である。この分解を用いて、1 層のモデルがバイグラムやスキップトライグラム（単一トークンのコピーを含む）は表現できてもコンテキスト内のパターン補完（$[\mathrm{A}][\mathrm{B]} \ldots [\mathrm{A}] \to [\mathrm{B}]$）は実行できない理由を示し、対照的に 2 層のモデルでは**直前トークンヘッド**（previous-token head）と**照合コピーヘッド**（match-and-copy head）を合成して**誘導ヘッド**（induction head）を形成できることを示す (Olsson et al. 2022)。最後に、重みの検査と因果的ヘッドアブレーションによってこの回路仮説を検証する。

```
import copy
```

10.7.1 回路と残差ストリーム

10.7.1.1 QK 回路と OV 回路

`TinyCharLM`（10.4.5.1 節）では、トークン埋め込み $\mathbf{e}_{x_i} \in \mathbb{R}^d$（$d = 128$）が N 個の残差ブロックを通過する。各ブロックは $H = 4$ 個の因果アテンションヘッド（幅 $d_h = d/H = 32$）からなり、MLP、正規化、バイアス項を持たず、その後に重み共有された逆埋め込み射影（unembedding projection）$\mathbf{E} \in \mathbb{R}^{|\mathcal{V}|\times d}$ が続く。

トークンベクトルを列ベクトル $\mathbf{h}_j \in \mathbb{R}^d$ と書くと、各ヘッドは $\mathbf{h}_j$ を $\mathbf{W}_v \in \mathbb{R}^{d_h\times d}$ によって射影し、重み付き和 $\sum_{j\le i} \alpha_{ij}\mathbf{W}_v\mathbf{h}_j$ を作る。ブロックの出力行列を H 個の列スライス $\mathbf{W}_o \in \mathbb{R}^{d\times d_h}$（ヘッドごとに 1 つ）に分割すると、連結と射影のステップは各ヘッドの独立な寄与の和として表せる。

$$\mathbf{W}_o \sum_{j\le i} \alpha_{ij}\,\mathbf{W}_v\mathbf{h}_j = \sum_{j\le i} \alpha_{ij}\,\mathbf{W}_o\mathbf{W}_v\,\mathbf{h}_j.$$

同様に、クエリ位置 i とキー位置 j の間のソフトマックス前のスコアは $(\mathbf{W}_q\mathbf{h}_i)^\top(\mathbf{W}_k\mathbf{h}_j)/\sqrt{d_h}$（ただし $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{d_h\times d}$）である。したがって、各ヘッドは 2 つの $d \times d$ 行列（ランクは高々 d_h）によって完全に特徴づけられる (Elhage et al. 2021)。

$$s_{ij} = \frac{\mathbf{h}_i^\top \mathbf{W}_{\mathrm{QK}}\,\mathbf{h}_j}{\sqrt{d_h}}, \qquad \mathbf{W}_{\mathrm{QK}} = \mathbf{W}_q^\top\mathbf{W}_k, \qquad \mathbf{W}_{\mathrm{OV}} = \mathbf{W}_o\mathbf{W}_v. \tag{10.34}$$

- **QK 回路 $\mathbf{W}_{\mathrm{QK}}$**（RoPE のもとでは $\mathbf{W}_q^\top\mathbf{R}_{j-i}\mathbf{W}_k$、10.4.3 節）は、ヘッドが**どこに**アテンションを向けるかを決める。
- **OV 回路 $\mathbf{W}_{\mathrm{OV}}$** は、参照元の位置にアテンションが向けられたとき、ヘッドが参照先の位置**どの**ベクトルを書き込むかを決める。

個々の因子ではなく積 $\mathbf{W}_{\mathrm{QK}}$ と $\mathbf{W}_{\mathrm{OV}}$ を調べることで、ゲージ自由度（gauge freedom）が取り除かれ（$\mathbf{W}_o\mathbf{G}^{-1}$ と $\mathbf{G}\mathbf{W}_v$ からはまったく同じ $\mathbf{W}_{\mathrm{OV}}$ が任意の可逆行列 $\mathbf{G} \in \mathbb{R}^{d_h\times d_h}$ について得られる）、ランク d_h の部分空間ボトルネックが明示的になる。

10.7.1.2 層をまたぐ残差表現

$\mathbf{h}_i^{(0)} = \mathbf{e}_{x_i}$ とおくと、ブロック $\ell \in \{1,\dots,N\}$ は位置 i の表現を加法的に更新する。

$$\mathbf{h}_i^{(\ell)} = \mathbf{h}_i^{(\ell-1)} + \sum_{m=1}^{H}\sum_{j\le i} \alpha_{ij}^{\ell m}\,\mathbf{W}_{\mathrm{OV}}^{\ell m}\,\mathbf{h}_j^{(\ell-1)}, \tag{10.35}$$

そして、重み共有された出力ヘッドがロジット $\mathbf{z}_i = \mathbf{E}\,\mathbf{h}_i^{(N)}$ を出力する。系列 $\mathbf{h}_i^{(0)},\dots,\mathbf{h}_i^{(N)}$ は位置 i における**残差ストリーム**（Figure 10.13）であり、すべてのヘッドが加法的に読み書きする共有通信チャネルとなっている。

10.7.1.3 バイグラム・スキップトライグラムと 1 層の限界

Equation 10.35 を深さ方向に展開すると、N 層のアテンション層がどのような関数を表現できるかが明らかになる。

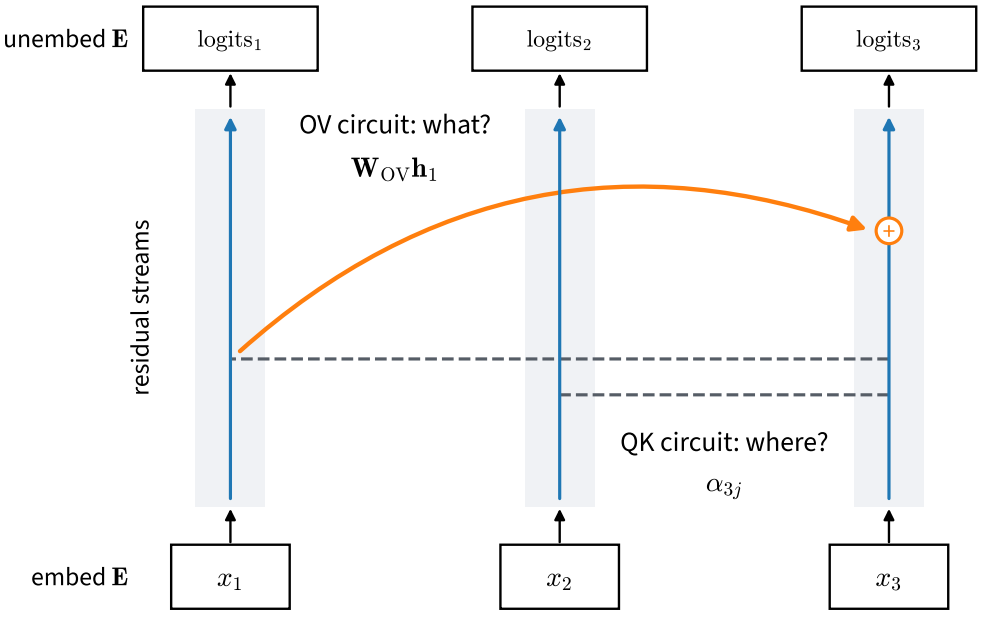


Figure 10.13: 残差ストリーム。各位置は埋め込みから出力ロジットへとベクトルを運ぶ。QK 回路がアテンション重み α_{3j} を決定し、OV 回路がアテンションを向けられた各ベクトルから参照先ストリームへの寄与を決定する。

- **0 ブロック**（$N = 0$）：$\mathbf{z}_i = \mathbf{E}\,\mathbf{e}_{x_i}$ となり、対称かつ半正定値なグラム行列 $\mathbf{E}\mathbf{E}^\top$ を介して、制約付きのトークンバイグラムのロジットしかモデル化できない（演習問題 1）。
- **1 ブロック**（$N = 1$）：

$$\mathbf{z}_i = \mathbf{E}\,\mathbf{e}_{x_i} + \sum_{m=1}^{H} \sum_{j \le i} \alpha_{ij}^{m}\, \mathbf{E}\,\mathbf{W}_{\mathrm{OV}}^{m}\,\mathbf{e}_{x_j}. \tag{10.36}$$

  各ヘッドの項は $x_j \ldots x_i \to x_{\text{next}}$ という形の**スキップトライグラム**規則を実装する（たとえば、**“in”** から先行する **“keep”** にアテンションを向けて **“mind”** のスコアを押し上げたり、x_j 自体を押し上げて**コピー**したりする）。

- **2 ブロック**（$N = 2$）**と誘導ヘッド**：コンテキストに [A][B] ... [A] が含まれており、モデルがコンテキスト内参照によって [B] を予測しなければならないとする。1 ブロックのモデルでは、内容の照合によってこれを行うことはできない。[B] を位置 j からコピーするにはヘッドが位置 j にアテンションを向ける必要があるが、Equation 10.36 では位置 j のキーは $\mathbf{e}_{x_j} = \mathbf{e}_{[\mathrm{B}]}$ のみに依存し、その直前のトークン $x_{j-1} = [\mathrm{A}]$ については何も知らないからである。

  2 ブロックあれば、**K 合成**（K-composition）によってこの問題を解決できる（Figure 10.14、演習問題 2）。ブロック 1 では、**直前トークンヘッド**がオフセット -1 にアテンションを向け、$x_{j-1} = [\mathrm{A}]$ を残差ストリーム $\mathbf{h}_j^{(1)}$（位置 j）に書き込む。続いてブロック 2 では、**誘導ヘッド**が自身のクエリ [A] を $\mathbf{h}_j^{(1)}$ 内のそのキー特徴量と照合し、OV 回路を使って $x_j = [\mathrm{B}]$

を予測ロジットへとコピーする。

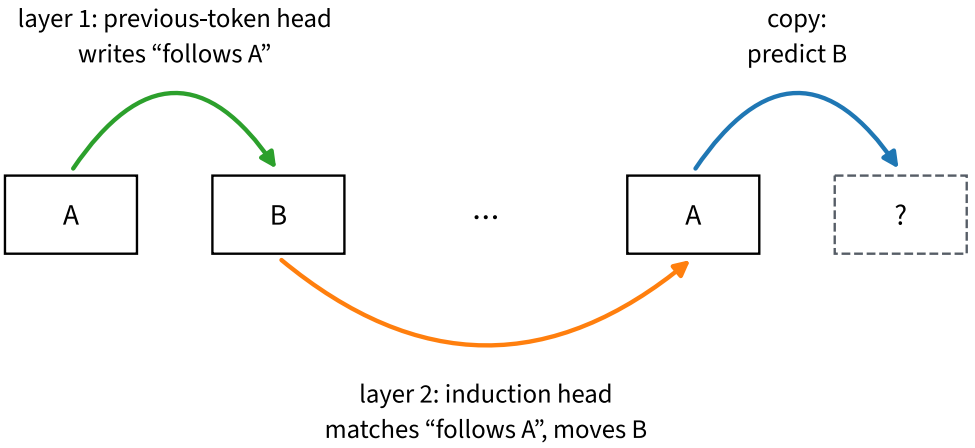


Figure 10.14: 誘導回路は 2 層にまたがる。第 1 層では直前トークンヘッドが各トークンの先行トークンをそのストリームに書き込むため、B の位置のキーは「A に続く」と告知できる。第 2 層では誘導ヘッドの QK 回路が現在の A をその告知と照合し、OV 回路が B を予測へとコピーする。

10.7.2 コピーと誘導

10.7.2.1 反復系列タスク

勾配降下法がこの 2 層の回路を発見するかどうかを検証するため、**反復ランダムトークン**の合成系列でモデルを訓練する。各系列では、長さ $L \in [L_{\min}, L_{\max}]$ のランダムなパターンをサイズ 64 の語彙から一様に抽出し、長さ 64 になるまで並べて繰り返す。パターンは毎回新たに抽出されるため、初回コピー（$t < L$）は予測不可能である（ランダム予測での交差エントロピーは $\log 64 \approx 4.16$ nats）。パターンが再開する位置 $t = L$ のトークンも同様である。そのコンテキスト内ではまだ何も繰り返されておらず、モデルは L を知り得ないからである。したがって、ターゲット $t \leq L$ を初回コピーとして数える。後続のコピー（$t > L$）は、コンテキスト内参照によってコンテキストから予測できる。

```
def repeated_batch(batch_size, num_steps, vocab, Lmin, Lmax, device=None):
    """Random patterns of length Lmin..Lmax, tiled to num_steps tokens."""
    L = torch.randint(Lmin, Lmax + 1, (batch_size,), device=device)
    s = torch.randint(0, vocab, (batch_size, num_steps), device=device)
    pos = torch.arange(num_steps, device=device)
    return torch.gather(s, 1, pos[None, :] % L[:, None]), L

x, L = repeated_batch(batch_size=2, num_steps=16, vocab=10, Lmin=5, Lmax=8)
print(f'pattern lengths: {L.tolist()}')
print(x)
```

```
pattern lengths: [5, 6]
tensor([[3, 6, 6, 6, 2, 3, 6, 6, 6, 2, 3, 6, 6, 6, 2, 3],
        [9, 2, 9, 6, 3, 7, 9, 2, 9, 6, 3, 7, 9, 2, 9, 6]])
```

`eval_copy` は、初回コピー（$t \leq L$）での交差エントロピー、後続のコピー（$t > L$）での交差エントロピー、および後続のコピーでの top-1 正解率を測定する。

```
@torch.inference_mode()
def eval_copy(model, x, L):
    """First-copy loss, later-copy loss, and later-copy accuracy."""
    logits, Y = model(x[:, :-1]), x[:, 1:]
    ce = F.cross_entropy(logits.transpose(1, 2), Y, reduction='none')
    later = torch.arange(1, Y.shape[1] + 1, device=Y.device) > L[:, None]
    acc = (logits.argmax(-1) == Y)[later].float().mean()
    return torch.stack((ce[~later].mean(), ce[later].mean(), acc)).tolist()

class RepeatedTokens(d2l.DataModule):
    """An endless stream of fresh repeated-token minibatches."""
    def __init__(self, Lmin, Lmax, batch_size=128, num_steps=64, vocab=64):
        super().__init__()
        self.save_hyperparameters()

    def train_dataloader(self):
        while True:
            x, _ = repeated_batch(self.batch_size, self.num_steps,
                                  self.vocab, self.Lmin, self.Lmax)
            yield x[:, :-1], x[:, 1:]

def train_repeat(model, num_steps, Lmin, Lmax, eval_every=50):
    """Train with Adam; return (first, later) copy losses per round."""
    data, device = RepeatedTokens(Lmin, Lmax), d2l.try_gpu()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    history = []
    for _ in range(num_steps // eval_every):
        d2l.train_steps(model, data, optimizer, eval_every)
        x, L = repeated_batch(256, 64, 64, Lmin, Lmax, device)
        history.append(eval_copy(model, x, L)[:2])
    return history

def drop_step(hist, eval_every=50):
    """First step at which the later-copy loss is below 1 nat, or None."""
    return next((eval_every * (k + 1) for k, (_, rep) in enumerate(hist)
                 if rep < 1), None)
```

10.7.2.2 固定オフセットによる解

可変周期を試す前に、パターンの周期が $L = 32$ に固定されているとしよう。**1 ブロック**のモデル（`num_blks=1`, `bias=True`）は、容易に 100% の 2 回目のコピーの正解率を達成する。

```
model_shortcut = d2l.TinyCharLM(vocab_size=64, num_blks=1, pos='rope',
                                bias=True)
train_repeat(model_shortcut, 800, Lmin=32, Lmax=32)
first, rep, acc = eval_copy(model_shortcut, *repeated_batch(
    256, 64, 64, 32, 32, d2l.try_gpu()))
print(f'first copy {first:.2f} nats, second copy {rep:.2f} nats, '
      f'second-copy accuracy {acc:.2f}')
```

```
first copy 4.38 nats, second copy 0.14 nats, second-copy accuracy 1.00
```

回路の解析では内容に基づく誘導に 2 層が必要であると論じたのに、なぜ単一の層で後続トークンをコピーできるのだろうか。$L = 32$ が一定のとき、位置 $i + 1$ のターゲットトークンは常に固定オフセット $j = i - 31$ のトークンと一致する。定数バイアスのクエリとキーを持つ単一の RoPE ヘッドは、内容を一切照合せず純粋に相対オフセット -31 だけでアテンションを向けることができる（演習問題 3）。`model_shortcut` を周期 $L = 24$ のパターン（ターゲットはオフセット -23 にある）で評価すると、このショートカットはただちに露呈する。アテンションはオフセット -31 に固定されたままであり、正解率はランダム予測の水準（$1/64 \approx 0.02$）まで崩壊する。

```
x, L = repeated_batch(256, 64, 64, 24, 24, d2l.try_gpu())
first, rep, acc = eval_copy(model_shortcut, x, L)
with torch.inference_mode():
    m = model_shortcut.attention_weights(x[:, :-1])[0]
print(f'patterns of length 24: second-copy accuracy {acc:.2f}')
for offset in (31, 23):
    mass = m.diagonal(-offset, dim1=2, dim2=3).mean((0, 2))
    print(f'mass at offset {offset}, per head: '
          + ' '.join(f'{v:.2f}' for v in mass.tolist()))
```

```
patterns of length 24: second-copy accuracy 0.02
mass at offset 31, per head: 1.00 0.96 1.00 1.00
mass at offset 23, per head: 0.00 0.00 0.00 0.00
```

10.7.2.3 2 ブロックの誘導回路

パターンの周期を $L \in \{16, \ldots, 32\}$ から一様にサンプリングすると、いかなる固定オフセットのショートカットも使えなくなる。ここでは、独立な 3 つの 2 ブロックモデルと対照用の 1 ブロックモデル（すべて `bias=False`）を 5,000 ステップ訓練する。

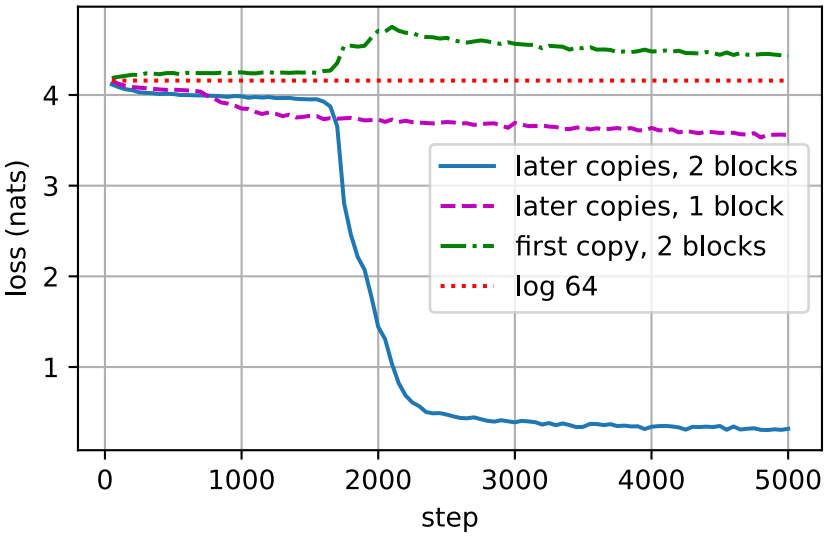
```
runs = []  # (blocks, model, history): three two-block runs, one control
for num_blks in (2, 2, 2, 1):
    model = d2l.TinyCharLM(vocab_size=64, num_blks=num_blks, pos='rope')
    runs.append((num_blks, model, train_repeat(model, 5000, 16, 32)))
models_two = [model for _, model, _ in runs[:3]]
_, model_two, hist_two = runs[0]   # Run 1: maps and curves shown for it
_, model_one, hist_one = runs[3]
x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
print('blocks  first copy  later copies  accuracy  below 1 nat at step')
for num_blks, model, hist in runs:
    first, rep, acc = eval_copy(model, x, L)
    print(f'{num_blks:6d}{first:12.2f}{rep:14.2f}{acc:10.2f}'
          f'{drop_step(hist) or "never":>21}')
```

```
blocks  first copy  later copies  accuracy  below 1 nat at step
     2        4.44          0.30      0.94                 2150
     2        4.31          0.16      0.96                 1550
     2        4.31          0.19      0.95                 2650
     1        4.26          3.55      0.11                never
```

2 ブロックの 3 回の実行はすべて 90% を超える後続コピーの正解率（損失にして約 0.2〜0.4 nats）を達成するのに対し、1 ブロックのモデルは 3 nats を超えたところで停滞する（正解率は $\sim$ 10% であり、コンテキスト内に現れたトークンのユニグラム集合のスコアを押し上げている

にすぎない）。初回コピーの損失は、どの実行でもランダム予測の値である $\log 64$ をわずかに**上回る**結果となる。4 つのモデルはすべてコンテキスト内に既出のトークンを選好するよう学習しており、最初の周期ではこの確信が裏目に出る。すなわち、そこで偶然トークンが重複すると、2 ブロックのモデルはそのトークンの前回の後続トークンを予測してしまうのである。さらに、訓練曲線は初期のプラトーの後に急激な相転移（phase transition）を示す。

```
steps = list(range(50, 5001, 50))
d2l.plot(steps, [[h[1] for h in hist_two], [h[1] for h in hist_one],
                 [h[0] for h in hist_two], [math.log(64)] * len(steps)],
         'step', 'loss (nats)',
         legend=['later copies, 2 blocks', 'later copies, 1 block',
                 'first copy, 2 blocks', 'log 64'], figsize=(5, 3))
```



10.7.3 回路の裏付け

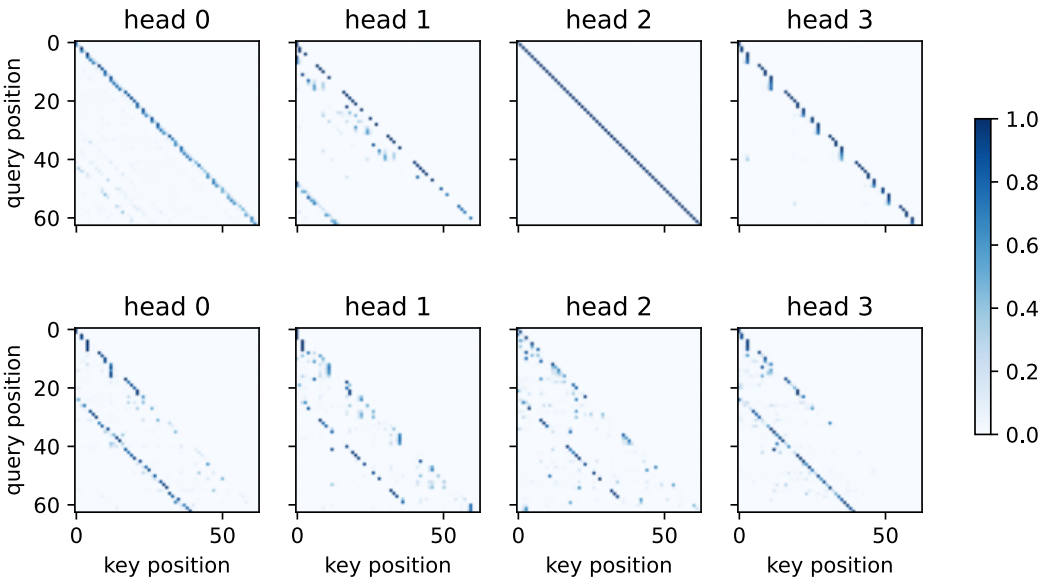
`model_two` が Figure 10.14 の 2 段階の誘導回路（induction circuit）を実際に実装しているかを検証するため、(1) アテンションマップ、(2) OV コピー行列、(3) ヘッドアブレーションを調べる。

10.7.3.1 学習されたヘッドのアテンションパターン

周期 $L = 24$ のテスト系列では、この回路の予測によれば、少なくとも 1 つのブロック 1 のヘッドが 1 位置前（$j = i - 1$）にアテンションを向け、ブロック 2 のヘッドは誘導ターゲット $j = i + 1 - L = i - 23$ に（$i \geq 24$ 以降で）アテンションを向けるはずである。

```
x, L = repeated_batch(1, 64, 64, 24, 24, d2l.try_gpu())
with torch.inference_mode():
    maps = torch.stack(model_two.attention_weights(x[:, :-1]))
d2l.show_heatmaps(maps[:, 0], xlabel='key position',
                  ylabel='query position',
```

```
                titles=[f'head {h}' for h in range(4)],
                figsize=(9, 4.5), cmap='Blues')
```



可変のパターン長 $L \in \{16, \ldots, 32\}$ および 3 回の実行すべてにわたってこれら両方の役割を定量化するため、head_scores は直前トークン（$j = i - 1$）と誘導ターゲット（$j = i + 1 - L$、ただし $i \geq L$）に対する各ヘッドの平均アテンション重みを計算する。

```
@torch.inference_mode()
def head_scores(model, x, L):
    """Mean attention on the previous token and on the induction target:
    shape (blocks, 2, heads)."""
    maps = torch.stack(model.attention_weights(x[:, :-1]))
    i = torch.arange(maps.shape[-1], device=x.device)
    tgt = i + 1 - L[:, None]              # Induction target of query i
    is_tgt = i == tgt[..., None]          # (batch, query, key)
    valid = tgt >= 1                      # Query token occurred before
    prev = maps.diagonal(-1, dim1=3, dim2=4).mean((1, 3))
    ind = (maps * is_tgt[:, None]).sum(-1)  # (blocks, batch, heads, query)
    ind = (ind * valid[:, None]).sum((1, 3)) / valid.sum()
    return torch.stack((prev, ind), 1)

x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
all_scores = [head_scores(model, x, L) for model in models_two]
scores = all_scores[0]  # Run 1, whose maps are shown above
for r, run_scores in enumerate(all_scores):
    for b, rows in enumerate(run_scores.tolist()):
        for name, row in zip(('previous', 'induction'), rows):
            print(f'run {r + 1}, block {b + 1}, {name:<9}: '
                  + ' '.join(f'{v:.2f}' for v in row))
```

```
run 1, block 1, previous : 0.34 0.46 0.00 0.50
```

```
run 1, block 1, induction: 0.00 0.01 0.00 0.00
run 1, block 2, previous : 0.09 0.12 0.07 0.14
run 1, block 2, induction: 0.40 0.38 0.41 0.58
run 2, block 1, previous : 0.02 0.03 0.99 0.00
run 2, block 1, induction: 0.00 0.01 0.00 0.00
run 2, block 2, previous : 0.07 0.06 0.08 0.06
run 2, block 2, induction: 0.45 0.58 0.23 0.61
run 3, block 1, previous : 0.19 0.01 0.39 0.46
run 3, block 1, induction: 0.00 0.00 0.00 0.00
run 3, block 2, previous : 0.08 0.07 0.06 0.07
run 3, block 2, induction: 0.20 0.20 0.47 0.24
```

どの実行でも、ブロック 1 のヘッドのうち 1〜3 個が直前トークンヘッドに特化する一方、ブロック 2 のヘッドはアテンションを誘導ターゲット $j = i + 1 - L$ に集中させる。

10.7.3.2 コピー行列

位置 j にアテンションを向けて x_j を予測できるのは、ブロック 2 の OV 回路がトークン x_j のロジットを押し上げる場合に限られる。ここで、次の量を定義する。

$$C_{ab} = \mathbf{e}_b^\top \mathbf{W}_{\mathrm{OV}} \bar{\mathbf{h}}_a \tag{10.37}$$

これはトークン b に与えられるロジットの押し上げ量であり、あるブロックの総和 OV 回路が残差ストリームの平均ベクトル $\bar{\mathbf{h}}_a$（トークン a を保持する各位置での平均、または生の埋め込み $\mathbf{e}_a$）を読み取ったときの値を表す。コピーを行う OV 回路では、対角成分（$b = a$）がほぼすべての 64 行で行内の最大値となる（ランダムな場合は $1/64 \approx 0.02$ である）。

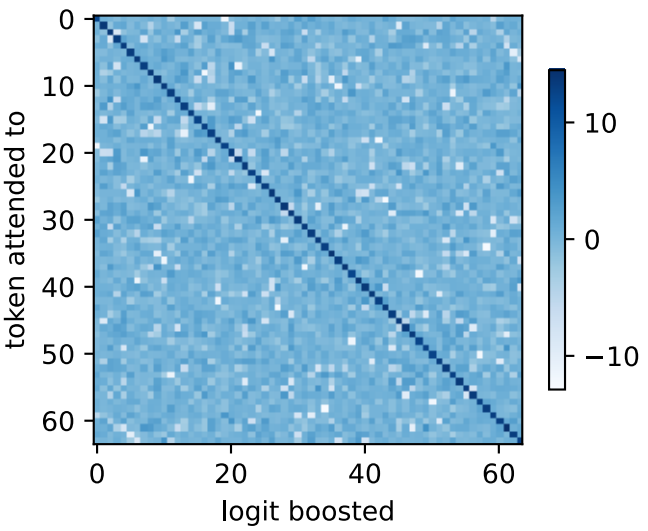
```python
def circuits(model, blk):
    """W_QK and W_OV of every head of block blk, each (heads, D, D)."""
    # nn.Linear stores its weight as (output, input), the orientation of the
    # column-vector equations, so no transpose is needed
    W_qkv = model.blks[blk]['qkv'].weight           # (3D, D): W_q; W_k; W_v
    W_o = model.blks[blk]['proj'].weight            # (D, D): heads in columns
    H, D = model.num_heads, W_o.shape[0]
    W_q, W_k, W_v = W_qkv.reshape(3, H, D // H, D)  # Each (H, d_h, D)
    W_QK = W_q.mT @ W_k
    W_OV = W_o.reshape(D, H, D // H).transpose(0, 1) @ W_v
    return W_QK, W_OV

@torch.inference_mode()
def copy_matrix(model, blk, X=None):
    """C[a, b] = e_b^T W_OV h_a: W_OV summed over the heads of blk, h_a the
    mean stream vector entering blk at the positions of token a in X, or the
    embedding e_a if X is None."""
    E = h = model.token_emb.weight
    if X is not None:
        S = model.stream(X, blk)[0]             # The stream entering blk
        h = torch.zeros_like(E).index_add_(0, X.flatten(), S.flatten(0, 1))
        h = h / torch.bincount(X.flatten(), minlength=len(E))[:, None]
    W_OV = circuits(model, blk)[1].sum(0)
    return h @ W_OV.T @ E.T
```

```
def diagonal_share(C):
    """Fraction of the rows of C whose largest entry is on the diagonal."""
    rows = torch.arange(len(C), device=C.device)
    return float((C.argmax(-1) == rows).float().mean())
```

2 ブロックの 3 回の訓練実行すべて（ブロック 1 の残差ストリームの平均 $\bar{\mathbf{h}}_a$ と生のトークン埋め込み $\mathbf{e}_a$ の両方）および対照用の 1 ブロックモデルにわたって `diagonal_share(copy_matrix(...))` を評価すると、OV 回路がほぼ対角なコピー行列を形成していることがわかる。

```
x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
print('rows whose largest entry is the diagonal (chance 1/64)')
print(f'{"":<32}   run 1   run 2   run 3')
for name, X in (('2 blocks, block 2, mean stream', x),
                ('2 blocks, block 2, embeddings', None)):
    shares = [diagonal_share(copy_matrix(m, 1, X)) for m in models_two]
    print(f'{name:<32}' + ''.join(f'{v:8.2f}' for v in shares))
print(f'{"1 block, mean stream":<32}'
      f'{diagonal_share(copy_matrix(model_one, 0, x)):8.2f}')
d2l.show_heatmaps(copy_matrix(model_two, 1, x)[None, None].cpu(),
                  xlabel='logit boosted', ylabel='token attended to',
                  figsize=(3.6, 3.6), cmap='Blues')
```

```
rows whose largest entry is the diagonal (chance 1/64)
                                 run 1   run 2   run 3
2 blocks, block 2, mean stream    1.00    1.00    0.91
2 blocks, block 2, embeddings     0.61    0.97    0.70
1 block, mean stream              1.00
```



ここで示唆に富む 2 つの機微に注目しよう。(1) 対照用の 1 ブロックモデルも対角な OV コピー行列を学習しており、このことから QK の K 合成を伴わない OV コピーだけではコンテキスト内の後続トークンを検索できないことがわかる（演習問題 6)。(2) ブロック 2 は、ブロック 1 のストリーム $\bar{\mathbf{h}}_a$ からのほうが、実行ごとに対角成分の割合が大きくばらつく生の埋め込み $\mathbf{e}_a$ 単体か

らよりも確実にコピーできる。

10.7.3.3 ヘッドアブレーション

因果的な必要性を検証するため、ablate はブロック内の選択したヘッドの出力射影スライス $\mathbf{W}_o^m$ を 0 にする。ここでは、ブロック 1 の直前トークンヘッド（スコア > 0.2）を除去する場合、ブロック 1 の残りのヘッドを除去する場合、およびブロック 2 の誘導ヘッドを強いものから順に累積的に除去する場合を比較する。

```
def ablate(model, blk, heads):
    """Copy of model with the given heads of block blk removed."""
    model = copy.deepcopy(model)
    W_o = model.blks[blk]['proj'].weight   # (D, D): heads side by side
    with torch.no_grad():
        W_o.view(W_o.shape[0], model.num_heads, -1)[:, heads] = 0
    return model
```

訓練済みの 3 つの 2 ブロックモデルすべてに ablate を適用すると、ブロック 1 の直前トークンヘッドを除去した場合とブロック 2 の上位の誘導ヘッドを除去した場合とで、後続コピーの交差エントロピーおよびコピー正解率がどれだけ悪化するかが定量化される。

```
x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
H, table = model_two.num_heads, {}  # Variant: (loss, accuracy) per run
for r, (model, s) in enumerate(zip(models_two, all_scores)):
    prev = [h for h in range(H) if s[0, 0, h] > 0.2]  # Previous-token heads
    rest = [h for h in range(H) if h not in prev]
    order = s[1, 1].argsort(descending=True).tolist()  # Strongest first
    print(f'run {r + 1}: block 1 previous-token heads {prev}, others {rest}')
    variants = {'none': model,
                'block 1, previous-token heads': ablate(model, 0, prev),
                'block 1, all other heads': ablate(model, 0, rest)}
    for k in range(1, H + 1):
        variants[f'block 2, top {k} induction'] = ablate(model, 1, order[:k])
    for name, variant in variants.items():
        table.setdefault(name, []).append(eval_copy(variant, x, L)[1:])
table['one-block model'] = [eval_copy(model_one, x, L)[1:]]
print(f'{"heads removed":<31}later-copy loss / accuracy: run 1, 2, 3')
for name, cols in table.items():
    print(f'{name:<31}'
          + ''.join(f'{rep:7.2f} /{acc:5.2f}' for rep, acc in cols))
```

```
run 1: block 1 previous-token heads [0, 1, 3], others [2]
run 2: block 1 previous-token heads [2], others [0, 1, 3]
run 3: block 1 previous-token heads [2, 3], others [0, 1]
heads removed                  later-copy loss / accuracy: run 1, 2, 3
none                              0.31 / 0.94   0.15 / 0.96   0.17 / 0.96
block 1, previous-token heads     4.64 / 0.07   3.71 / 0.09   3.80 / 0.17
block 1, all other heads          5.66 / 0.02   3.96 / 0.10   3.85 / 0.13
block 2, top 1 induction          0.63 / 0.92   0.56 / 0.91   1.65 / 0.70
block 2, top 2 induction          1.66 / 0.68   4.49 / 0.28   3.14 / 0.45
block 2, top 3 induction          3.38 / 0.24   5.68 / 0.06   3.88 / 0.23
block 2, top 4 induction          4.25 / 0.02   4.19 / 0.01   4.18 / 0.02
```

```
one-block model                 3.56 / 0.11
```

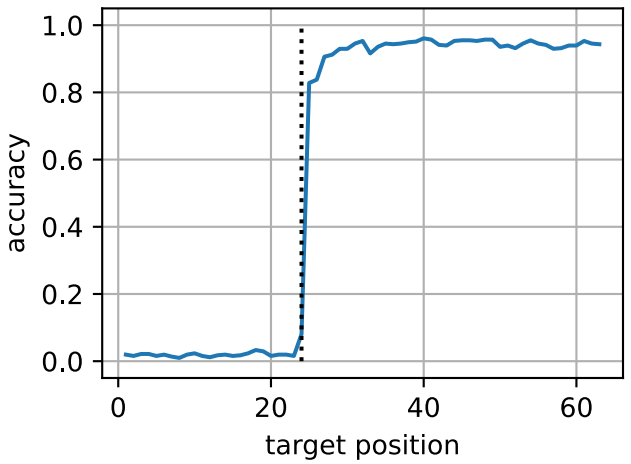
ブロック 1 の直前トークンヘッドをアブレーションすると、後続コピーの正解率は 1 ブロックモデルと同水準の 0.2 未満まで崩壊し、それらの必要性が裏付けられる。興味深いことに、ブロック 1 の**それ以外の**ヘッドをアブレーションしても検索が壊れる。`copy_matrix` がブロック 1 で変換されたストリームに依存していた（さらに正規化を持たないネットワークではゼロアブレーションによってストリームのノルムがずれる。平均アブレーションについては演習問題 5 を参照）ため、ブロック 2 はそれらのヘッドにも依存しているのである。ブロック 2 では、4 つのヘッドが誘導の負荷を冗長に分担しており、正解率がランダム予測水準まで落ちるには複数のヘッドをアブレーションする必要がある。

10.7.4 コンテキスト内学習

10.7.4.1 未知のパターンへの汎化

取り得るパターンは $64^{16} \approx 8 \times 10^{28}$ 通り存在するため、テスト系列はどれも未知であり、訓練分布全体ではすべてのトークン対が等確率で現れる。誘導回路は、コンテキストウィンドウ内で 1 度提示されただけで各トークンとその直後のトークンをその場で結びつける。これは**コンテキスト内学習**（in-context learning）の最小の例である。

```
x, L = repeated_batch(512, 64, 64, 24, 24, d2l.try_gpu())
with torch.inference_mode():
    acc = (model_two(x[:, :-1]).argmax(-1) == x[:, 1:]).float().mean(0)
d2l.plot([torch.arange(1, 64), [24, 24]], [acc.cpu(), [0, 1]],
         'target position', 'accuracy', fmts=('-', 'k:'))
```



正解率は、初回コピーでの $1/64$（内容照合がまだ重複トークンを見ていない再開境界 $t = 24$ でも同じ）から、約 0.8（位置 25）へと跳ね上がり、その数位置後には 0.9 を超える。

再開後に残る誤差は何に起因するのだろうか。あるトークンが $K > 1$ 回、異なる後続トークンを伴って L トークンのパターン内に現れるとき、現在のトークンだけの照合では曖昧さが残る。すなわち、K 個の出現箇所から一様に選ぶと確率 $1/K$ でしか成功しない。$L = 24$、$p = 1/64$ のとき、パターン内でトークンが重複する確率は $1 - (63/64)^{23} \approx 0.30$ であり、一様な単一トークン照合が予測する全体の正解率は次の式で与えられる。

$$\mathbb{E}\left[\frac{1}{K}\right] = \frac{1-(1-p)^L}{Lp} \approx 0.84, \qquad p = \frac{1}{64},$$

また、重複トークンに限ると正解率は ≈ 0.47 となる。テスト位置を $K = 1$（一意）と $K > 1$（重複）に分けると、3 回の実行すべてで一意なトークンでは 0.97 以上、重複トークンでも約 0.7〜0.9 に達し、単一トークン照合のベースラインである 0.47 を大きく上回る。ここでは検証しないが、もっともらしい理由として、モデルがより長い n グラムのコンテキストも照合していることや、特定の RoPE オフセットを選好していることが挙げられる（演習問題 2）。

```
period, num_steps = 24, 64
x, _ = repeated_batch(512, num_steps, 64, period, period, d2l.try_gpu())
pattern = x[:, :period]
K = (pattern[:, :, None] == pattern[:, None, :]).sum(-1)  # Count in pattern
K = K[:, torch.arange(period, num_steps - 1) % period]  # Current token
parts = (K == 1, K > 1, K > 0)           # Unique, repeated, all
rows = {'share of targets': [p.float().mean() for p in parts],
        'single-token match': [(1 / K[p]).mean() for p in parts]}
for r, model in enumerate(models_two):
    with torch.inference_mode():
        hit = (model(x[:, :-1]).argmax(-1) == x[:, 1:])[:, period:]
    rows[f'run {r + 1} accuracy'] = [hit[p].float().mean() for p in parts]
print(f'{"current token":<18}   unique repeated      all')
for name, row in rows.items():
    print(f'{name:<18}' + ''.join(f'{v:9.2f}' for v in row))
```

```
current token        unique repeated      all
share of targets       0.69     0.31     1.00
single-token match     1.00     0.47     0.84
run 1 accuracy         0.99     0.80     0.93
run 2 accuracy         1.00     0.87     0.96
run 3 accuracy         1.00     0.85     0.95
```

10.7.4.2 より大規模なモデルにおける誘導ヘッド

Olsson et al. (2022) は、さまざまなスケールの言語モデルにおいて、誘導ヘッドが急激な相転移の過程で形成され、それが一般的なコンテキスト内学習の創発と時期を同じくすることを実証した（演習問題 8）。コピーやあいまいなパターン補完にとどまらず、アテンション層は回帰タスクにおけるコンテキスト内の勾配ステップを実装することもできるが（演習問題 7、(Oswald et al. 2023)）、事前学習済みの言語モデルが実際にこの方法でコンテキスト内学習をしているかどうかについては議論がある (Shen et al. 2024)。

ここまでの解析は正規化も MLP も持たないモデルに依拠していたが、実際の Transformer ブロック（第 11 章）はその両方を備えている。いずれも基本的な考え方は変えないが、解析手法には修正が必要になる。層正規化（layer normalization）はストリームを自身のノルムで再スケーリングするため、ストリームからロジットへの写像はそのスケールを固定したときにのみ線形になる。そこで回路解析では、スケールを観測された値に固定し、残りを従来どおり扱う。MLP はどのアテンションパターンも経由しない経路を追加し、あるヘッドの出力が後段のヘッドに読み取られる前に MLP によって書き換えられることもある。最後に、訓練済みの言語モデルにおける誘導ヘッドはあいまいな照合を行う。すなわち、以前出現したトークンと同一ではなく単に類似しているだけのトークンに対しても、$[\mathrm{A}^*][\mathrm{B}^*] \ldots [\mathrm{A}] \to [\mathrm{B}]$ を補完する。

10.7.5 説明としてのアテンション重み

ここまでの検証から、機構的な説明としてアテンションのヒートマップだけでは不十分な理由が明らかになる。

- **重要なヘッドでも目立たないマップを持ち得る：**ブロック 1 の直前トークンヘッド以外のヘッドはヒートマップ上では不活性に見えたが、それらをアブレーションするとブロック 2 の検索が壊れた。
- **鋭いマップでも効果が無視できるほど小さい場合がある：**事前学習済みの言語モデルでは、遊休状態のヘッドが重みの大半を先頭トークン（**アテンションシンク**）に置くことが頻繁にあるが、その OV ベクトルはほぼゼロのノルムしか持たない (Xiao et al. 2024; Gu et al. 2025)。

Table 10.2 に、誘導回路を検証するために用いた証拠の階層をまとめる。

Table 10.2: 誘導ヘッドの解釈に対するレベル別の証拠。

本節での検証	証拠のレベル	支持される結論	残る限界
後続コピーの損失	振る舞い	2 ブロックは反復トークンタスクを解けるが、1 ブロックでは解けない	内部の構成要素を特定しない
周期の変更	入力への介入	固定周期モデルは内容ではなく位置を利用する	入力だけでなくタスクも変えてしまう
ヘッドスコア	活性化値	各ヘッドが予測どおりのオフセットにアテンションを向ける	値の変換を考慮していない
コピー行列	重みと平均活性化値	ブロック 2 の OV 回路がトークンをコピーする	検索を行わないモデルにも存在する
ヘッドアブレーション	構成要素への介入	後続コピーの予測には直前トークンヘッドだけでなくブロック 1 全体とブロック 2 の誘導ヘッドが必要	ゼロ化によりストリームが訓練分布から外れる

10.7.6 まとめ

各アテンションヘッドは、高々ランク d_h の 2 つの行列を介して残差ストリームに作用する。すなわち、どこにアテンションを向けるかを決める **QK 回路** $\mathbf{W}_q^\top \mathbf{W}_k$ と、参照先の位置へどのベクトルを書き込むかを決める **OV 回路** $\mathbf{W}_o \mathbf{W}_v$ である。単一のアテンション層はバイグラムやスキップトライグラム（単一トークンのコピーを含む）を表現できる一方、未知のコンテキスト内パターン $[\mathrm{A}][\mathrm{B}] \ldots [\mathrm{A}] \to [\mathrm{B}]$ を補完するには、**直前トークンヘッド**と照合コピーを行う**誘導ヘッド**が残差ストリームを介して合成できるよう少なくとも 2 つの層が必要である。可変周期の反復ランダムパターンで訓練すると、2 ブロックモデルは急激な相転移を経て 90% を超えるコンテキスト内正解率に達するのに対し、固定周期では 1 ブロックモデルが脆い位置ショートカットを利用できてしまう。したがって、ヘッドが実際に何を計算しているかを確立するには、アテンションパターンのプローブ、OV 重みの検査、および因果的ヘッドアブレーションを組み合わせる必要がある。

10.7.7 演習問題

1. **重み共有バイグラム。**0 ブロックのとき、トークン b がトークン a に続くロジットは $\mathbf{e}_b^\top \mathbf{e}_a$ であるため、埋め込みの幾何構造だけでモデルが選好する後続トークンが決まる。b を a の

最上位後続トークンと呼ぶのは $\mathbf{e}_b^\top \mathbf{e}_a$ がその行で最大のロジットであるときとし、同順位はないものとする。

1. すべての埋め込みが同じノルムを持つならば、あらゆるトークンの最上位後続トークンはそのトークン自身になることを示せ。
2. 等ノルムの仮定を外す。トークン $b \neq a$ が a の最上位後続トークンになり得るのは $\|\mathbf{e}_b\| > \|\mathbf{e}_a\|$ の場合に限られることを示し、最上位後続トークンが巡回 $a_1 \to a_2 \to \cdots \to a_k \to a_1$（$k \geq 2$）を形成し得ないことを結論づけよ。
3. あるコーパスにおいて、文字 `t` の最頻の後続文字が `h`、`h` の最頻の後続文字が `e`、`e` の最頻の後続文字が空白、空白の最頻の後続文字が `t` であるとする。どれほど長く訓練しても、0 ブロックのモデルではこれら 4 つすべてを最上位後続トークンとして再現できないことを示せ。
4. 1 ブロックのモデルでは、先頭位置にあるすべてのヘッドはその位置自身にのみアテンションを向ける。そこでのロジットを $\mathbf{EMe}_{x_0}$（$\mathbf{M}$ は OV 回路から構成される行列）と書き表し、前問までの 2 つの制約がいずれも $\mathbf{EME}^\top$ には当てはまらない理由を説明せよ。

2. **2 つのブロックを通る経路**。N 個のブロック（各 H ヘッド）からなるモデルの T トークン系列上でのアテンションパターンを固定し、$\mathbf{A}^{\ell m}$ を $T \times T$ 行列（成分は $\alpha_{ij}^{\ell m}$）とする。
   1. Equation 10.35 のロジットを $N = 2$ に対して経路項の和へと展開せよ。ブロック 1 のヘッド m とブロック 2 のヘッド m' を通る経路が、パターン $\mathbf{A}^{2m'}\mathbf{A}^{1m}$ と OV 回路 $\mathbf{W}_{\text{OV}}^{2m'}\mathbf{W}_{\text{OV}}^{1m}$ を持つ単一のヘッドのように作用することを示し、その回路のランクの上界を求めよ。
   2. N ブロックに対する経路項の個数を数え、`model_two` についてその個数を計算せよ。
   3. 訓練済みモデルではブロック 2 のパターンは固定されておらず、$\mathbf{h}^{(1)}$ にブロック 2 の QK 回路を介して依存する。$\mathbf{h}_i^{(1)}$ と $\mathbf{h}_j^{(1)}$ を Equation 10.34 のスコアに代入し、クエリ側、キー側、またはその両方にブロック 1 のヘッドの出力が含まれるかどうかによって項を分類せよ。これらの項は **Q 合成**および **K 合成**と呼ばれ、最初の小問の経路は **V 合成**と呼ばれる (Elhage et al. 2021)。Figure 10.14 の誘導回路はどの種類の項に依拠しているか。
   4. 2 トークンのプレフィックス (x_{i-1}, x_i) での照合が、ブロック 1 にオフセット -1 と -2 へアテンションを向けるヘッドが含まれるときに 2 つのブロックで実現できることを示し、照合のそれぞれの半分を担う合成項の名称を答えよ。
3. [code] **バイアスなしのショートカット**。10.7.2.2 節の固定オフセットモデルは射影バイアスを持っている。
   1. 回転前のヘッドのクエリとキーが、すべての位置で同じ定数ベクトル $\mathbf{q}$ と $\mathbf{k}$ であるとする。$\mathbf{R}_i^\top \mathbf{R}_j = \mathbf{R}_{j-i}$ を用いてスコアがオフセット $j - i$ のみに依存することを示し、$\mathbf{q}$ と $\mathbf{k}$ をどのように選べばオフセット -31 でスコアが最大になるかを説明せよ。
   2. バイアスがない場合、すべてのクエリとキーはストリームの線形関数となる。ヘッドがほぼ定数のクエリとキーを生成するためには、トークン埋め込みに何が含まれていなければならないか。
   3. 本文の 1 ブロックモデルを `bias=False` で 3 回訓練し、2 回目のコピーの正解率とオフセット 31 における各ヘッドのアテンションを報告せよ。バイアスなしでショートカットはどの程度確実に形成されるか。また、周期 24 のプローブでも維持されるか。
4. **何も変えない重み**。クエリ i におけるヘッドの出力は $\mathbf{W}_{\text{OV}} \sum_j \alpha_{ij} \mathbf{h}_j$（Equation 10.35）であり、ソフトマックスによりマスクされていないキー上のすべての重み α_{ij} は正になる。Jain and Wallace (2019) は、訓練済みのテキスト分類器において大きく異なるアテンション分布がしばしば予測を変化させないことを見いだし、アテンション重みは予測を説明しないと結論づけた。
   1. n 個のキー上のアテンション分布のうち、α_i と同じヘッド出力を与えるものの集合は、α_i の近傍に少なくとも次元 $n - 1 - d_h$ のアフィン部分空間の一部を含むことを

示せ。63 トークンの入力を `model_two` に与えたときの最後のクエリについてこの下界を計算せよ。

2. 最初の小問における代替分布は、Jain and Wallace (2019) のものと同様に、1 回につき 1 つの入力に対して構成されている。Wiegreffe and Pinter (2019) は、代替分布が説明に対する反証となるのは、単一の訓練済みモデルがあらゆる入力に対してそのような代替分布を生成しつつほぼ同じ予測を行う場合に限られると論じた。最初の小問の次元計算からはそのようなモデルが存在するかどうかについて何も言えない理由を説明し、`model_two` についてそれを判定する実験を記述せよ。

5. [code] **ゼロアブレーションと平均アブレーション**。10.7.3.3 節のアブレーションでは、ヘッドの出力をゼロに設定した。
   1. `proj` のあるヘッドに対応する列をゼロにすると、残差ストリームに対するそのヘッドの寄与だけが厳密に取り除かれる理由を説明せよ。ブロック 1 のヘッドを除去したとき、どのアテンションパターンが変化せず、どのパターンが変化し得るか。
   2. 訓練中にヘッドの出力がゼロになることはないため、ゼロ化するとストリームはモデルが未経験の状態へと移動する。**平均アブレーション**は、代わりにその出力 $\mathbf{o}_m$（ヘッド m の出力）を位置と系列にわたる平均 $\bar{\mathbf{o}}_m$ で置き換える。これを実装せよ。`TinyCharLM` を `bias=True` で構築し、`model_two` の重みを（`qkv` のバイアスはゼロとして）コピーし、ヘッド m に対応する `proj` の列をゼロにしたうえで、`proj` のバイアスをゼロ化前のそれらの列と $\bar{\mathbf{o}}_m$ の積に設定せよ。本文のアブレーションを再度実行せよ。ブロック 1 の直前トークンヘッド以外のヘッドを除去しても依然として検索は壊れるか。また、その答えは 10.7.3.3 節の 2 つの解釈のうちどちらを支持するか。
   3. ブロック 2 で 2 番目に強い誘導ヘッドを単独で除去せよ。単一ヘッドをそれぞれ除去したときの後続コピーの損失の増加量の和と、両方のヘッドを同時に除去したときの増加量を比較せよ。この比較から、ヘッドがどのように役割を分担しているかについて何が言えるか。
   4. Table 10.2 に挙げた限界のうち、平均アブレーションによって解消されるものはどれで、残るものはどれか。

6. [code] **重みにおける K 合成**。●誘導ヘッドの照合が機能するのは、ブロック 2 の QK 回路がブロック 1 の OV 回路によってストリームに書き込まれた内容を読み取るからである。
   1. ブロック 1 のヘッド m とブロック 2 のヘッド m'（`model_two`）のすべての対について、64×64 行列
   $$\mathbf{E}\,\mathbf{W}_{\mathrm{QK}}^{2m'}\,\mathbf{W}_{\mathrm{OV}}^{1m}\,\mathbf{E}^{\top}, \tag{10.38}$$
   を計算せよ。その成分 (a, b) は、クエリトークン a を直前のトークンが b であったキーに対してスコア付けする。Equation 10.37 のコピーテストがコピーに対して行ったのと同様に、最大成分が対角線上に位置する行の割合を測定せよ。
   2. 同じブロック 2 のヘッドにおける直接経路 $\mathbf{E}\,\mathbf{W}_{\mathrm{QK}}^{2m'}\,\mathbf{E}^{\top}$ と比較せよ。どのヘッド対が照合を担っているか。また、それらはアテンションに基づく `head_scores` のスコアと一致するか。
   3. 上の計算は、RoPE が QK 回路の内部で適用する回転を無視している。固定オフセットに対する相対回転 $\mathbf{R}_{j-i}$ を含め、オフセット -23 でこれを行うと対のランキングが変化するかどうかを確認せよ。

7. [code] **勾配ステップを実行するヘッド**。●誘導ヘッドはコピーによってコンテキストから学習する。モデルはコンテキスト内のラベル付き例から規則を推論することもできる。系列が対 $(\mathbf{x}_i, y_i)$（$i = 1, \dots, n$）を保持し、ここで標準正規分布に従う $\mathbf{x}_i \in \mathbb{R}^p$ に対して $y_i = \mathbf{w}^{\top}\mathbf{x}_i$（$\mathbf{w}$ は系列ごとに新たに抽出されるベクトル）であり、その後にラベルを予測すべきクエリ $\mathbf{x}_q$ が続くとする。
   1. $\frac{1}{2}\sum_i (y_i - \mathbf{w}^{\top}\mathbf{x}_i)^2$ に対する勾配降下法を $\mathbf{w} = \mathbf{0}$ から学習率 η で 1 ステップ行うと、予測 $\widehat{y}_q = \eta \sum_i y_i\, \mathbf{x}_i^{\top}\mathbf{x}_q$ が得られることを示せ。これをソフトマックスなしのアテンションとして解釈したとき、クエリ、キー、値の役割を果たすのはそれぞれ何か。ま

た、10.6 節の線形アテンション（特徴マップ $\phi(\mathbf{x}) = \mathbf{x}$）と比較せよ。

2. 各例にトークン $\mathbf{z}_i = [\mathbf{x}_i; y_i] \in \mathbb{R}^{p+1}$ を、クエリにトークン $\mathbf{z}_q = [\mathbf{x}_q; 0]$ を与える。クエリトークンにおける出力が $\hat{y}_q$ となるよう、ソフトマックスなしの 1 つのヘッドのクエリ写像、キー写像、値写像を定めよ。クエリトークンが自身にもアテンションを向けることが問題にならないのはなぜか。このヘッドを実装し、勾配ステップと一致することを確認せよ。このヘッドの重みは $\mathbf{w}$ に依存するか。
3. $\eta = 1/n$ のとき、このステップにより $\mathbf{w}_1 = \left(\frac{1}{n}\sum_i \mathbf{x}_i \mathbf{x}_i^\top\right)\mathbf{w}$ および $\mathbb{E}\,\|\mathbf{w}_1 - \mathbf{w}\|^2 = \frac{p+1}{n}\|\mathbf{w}\|^2$ が成り立つことを示せ。ヒント：標準正規分布に従う $\mathbf{x}$ に対して $\mathbb{E}[(\mathbf{x}^\top\mathbf{w})^2\|\mathbf{x}\|^2] = (p+2)\|\mathbf{w}\|^2$ である。これより、$\hat{y}_q$ の期待二乗誤差（y_q の分散に対する相対値）が $(p+1)/n$ となることを結論づけよ。
4. $p = 8$ に対し、500 本の系列にわたる相対誤差を $n = 10$、100、1000 で測定し、理論予測 $9/n$ と比較せよ。$n = 10$ のとき、このヘッドは常にゼロを予測する場合と比べてどうなるか。

8. [extended] **テキストにおける誘導ヘッド。**●●Olsson et al. (2022) は、言語モデルにおける誘導ヘッドの形成を、コンテキストの前半位置から後半位置にかけての損失の低下量として測られるコンテキスト内学習の急激な向上と結びつけている。これを 10.4.5.2 節のバイトレベルモデルで検証せよ。
   1. `d2l.TinyCharLM`（`pos='rope'`）を 10.4.5.2 節の WikiText-103 のバイト列上で、1 ラウンドあたり 100 ステップの `d2l.train_steps` を 30 ラウンド実行して訓練せよ。各ラウンドの後に 2 つの量を記録せよ。1 つ目は、テキスト語彙上のパターン長 16〜32 の反復ランダム系列に対して `head_scores` で計算した、ブロック 2 のヘッドにおける誘導ターゲットスコアの最大値である。2 つ目はコンテキスト内利得、すなわちターゲット位置 0〜15 での検証損失から位置 96〜127 での検証損失を引いた差である。
   2. 両方の量を訓練ステップに対してプロットせよ。誘導スコアとコンテキスト内利得は同じ期間に上昇するか。
   3. 訓練後、誘導スコアが最も高いブロック 2 のヘッド群を `ablate` で除去し、コンテキスト内利得と検証損失を再度測定せよ。これらのヘッドは利得のどれだけの割合を担っているか。
   4. 反復ランダム系列には反復以外の構造がないのに対し、テキストでは単語や句が不規則な間隔で繰り返される。もしテキストに重複する単語が一切含まれなかったら、結果のどれが変化するか。また、それはなぜか。

10.8 関連文献

ここまでの各節では、層の各構成要素について代表的な設計を 1 つずつ展開してきた。すなわち、ソフトマックスルックアップ（10.1 節）、スケール付きドット積によるスコア計算とマスキング（10.2 節）、マルチヘッドおよびグループ化クエリ自己アテンション（10.3 節）、位置エンコーディング（10.4 節）、タイル化オンラインソフトマックス（10.5 節）、スライディングウィンドウアテンションと線形アテンション（10.6 節）、そして機構的回路（10.7 節）である。本節では、これらと密接に関連する派生手法、現代のオープンウェイトモデルにおけるアーキテクチャ上の改良、および一次文献を概観する。

10.8.1 ルックアップとその正規化

10.8.1.1 連想記憶としてのアテンション

Ramsauer et al. (2021) は、ソフトマックスアテンションの 1 ステップが、キーと値の行を記憶パターンとする連続値**現代的ホップフィールドネットワーク**（modern Hopfield network）の 1 回の更新に対応することを示し、ソフトマックスアテンションを古典的な連想記憶（associative

memory）と結びつけた。逆温度（カーネル回帰における $1/\sigma^2$、ドット積アテンションにおける $1/\sqrt{d}$）によって、不動点（fixed point）が個々の記憶パターンになるか（高逆温度）、複数のパターンにわたる準安定な平均になるか（低逆温度）が決まる。この視点から同研究は、BERT などの事前学習済みエンコーダにおいて、浅い層は主として広範な平均化レジームで動作し、深い層は鋭い検索レジームで動作することを観察している。適応的カーネルバンド幅 (Silverman 1986) は、クエリまたはヘッドに依存するソフトマックス温度に対応する。

10.8.1.2 単調アライメント

Bahdanau et al. (2015) が機械翻訳向けに制約のない内容ベースのアライメントを導入する 1 年前、Graves (2013) は手書き文字合成向けに微分可能な**単調**（monotone）位置ベースのアテンション機構を導入し、平均位置がペンストロークごとに単調に前進する混合ガウス窓をパラメータ化した。これは音声や手書き文字には適した帰納バイアス（inductive bias）であるが（(Rabiner and Juang 1993)）、翻訳における語順の逆転に対しては制約が強すぎる（10.2.4 節）。

10.8.1.3 総和が 1 である必要のない重み

ソフトマックスの各行は和が 1 にならなければならないため、関連するキーを持たないクエリは、確率質量を先頭トークンへと集中させることが多い。これが 10.2.3.2 節および 10.7 節で見た**アテンションシンク**である（11.3.3.3 節も参照）。M. Sun et al. (2024) は、アテンションシンクを残差ストリーム内で暗黙的バイアス（implicit bias）項として働く巨大な外れ値の活性化値と関連づけており、一方 Barbero, Arroyo, et al. (2025) は、シンクが無操作（no-op）のターゲットを提供することで、深い長文脈 Transformer における表現崩壊（representation collapse）を防いでいると論じている。

近年のいくつかの設計では、この総和 1 の制約を緩和し、入力トークンのシンクを使わずにヘッドを待機状態にできるようにしている。

- **シグモイドアテンション** (Ramapuram et al. 2025) は、行ごとのソフトマックスを $-\log n$ だけシフトした要素ごとのロジスティックシグモイドに置き換え、重みを $1/(n+1)$ 付近に初期化する。
- **ゲート付きアテンション**（gated attention、(Qiu et al. 2025)）は、Qwen3-Next (Qwen Team 2025) で採用されており、各ヘッドのソフトマックス出力に入力依存のシグモイドゲートを乗じる。
- **学習可能なシンクロジット**（learnable sink logits）は、`gpt-oss` (OpenAI 2025) で採用されており、対応する値ベクトルを持たない訓練可能な仮想キースコアをソフトマックスの分母に追加する。
- **差分アテンション**（differential attention、(T. Ye et al. 2025)）は、ヘッドごとに 2 つのソフトマックス分布の差をとることで、無関係なキーに対するコモンモードの背景重みを相殺する。

10.8.2 自己アテンションとヘッド

10.8.2.1 起源とヘッドの冗長性

自己アテンションは、**系列内アテンション**（intra-attention）を伴う再帰構造 (Cheng et al. 2016)、自然言語推論（natural language inference）向けの再帰を用いない分解可能アテンション (Parikh et al. 2016)、および多様性ペナルティを伴う複数行の文埋め込み (Lin et al. 2017) から発展し、その後に Vaswani et al. (2017) が Transformer において射影付きマルチヘッド自己アテンションとして統合した。

単一混合のボトルネック（10.3.2 節）を克服するには複数のヘッドが不可欠であるが、訓練済みモデルはテスト時に大きなヘッドの冗長性を示すことが多い。実際、Michel et al. (2019) および Voita et al. (2019) は、BERT や NMT モデルにおける大半のヘッドを最小限の性能劣化で事後的に枝刈りでき、位置・構文・低頻度語を担う中核的なヘッドだけが残ることを示した。現代のアーキテクチャでは、グループ化クエリアテンション（grouped-query attention、10.3.3.6 節、11.3.3 節）によって、訓練前の段階からパラメータとキャッシュの効率化が図られている。

10.8.3 位置

10.8.3.1 学習される相対位置

RoPE の登場以前は、オフセットごとに訓練可能なキー埋め込み (Shaw et al. 2018)、Music Transformer におけるスキューインデックス付き相対ロジット (Huang et al. 2018)、あるいは T5 における対数距離バケットごとのスカラーバイアス (Raffel et al. 2020) を通じて相対オフセットが注入されていた。その後、ALiBi は学習されるバケットを固定の線形スロープに置き換え、RoPE はパラメータを持たない座標回転によって、オフセットのみに厳密に依存するスコアを実現した。

10.8.3.2 訓練済みモデルにおける RoPE

オープンウェイトモデルは、より長いコンテキストに対応するため、RoPE の基底周波数 b を段階的に引き上げてきた。具体的には、Llama 2 の $b = 10^4$ から、Llama 2 Long (Xiong et al. 2024) や Llama 3 (Grattafiori et al. 2024) では 5×10^5 へ、さらに Gemma 3 (Gemma Team 2025) のグローバル層では 10^6 へと拡大している。b を大きくすると、波長が訓練ウィンドウを超える低速レジームへとより多くの座標ペアが移行する（10.4.5.3 節）。Barbero, Vitvitskyi, et al. (2025) は、訓練済みモデルが高周波の RoPE ペアを位置ヘッド（直前トークンヘッドなど）に用い、低周波のペアをほぼ位置不変な意味チャネルとして用いていることを示した。これが、極端な距離で意味チャネルがドリフトしないよう最低周波成分を完全にゼロにする **p-RoPE** の動機となっている。YaRN (Peng et al. 2024) を超える事後学習でのコンテキスト拡張として、**LongRoPE** (Ding et al. 2024) は次元ごとの再スケーリング係数に対する進化的探索を用い、コンテキストウィンドウを最大 200 万トークンまで拡張している。

10.8.3.3 長さ汎化と長文脈の活用

T.-C. Chi et al. (2023) は ALiBi の有効受容野を解析し、平坦なパープレキシティ外挿が長距離の検索ではなく局所的なウィンドウ処理の反映であるという 10.4.5.2 節の結果を裏づけている。位置エンコーディングを持たない因果モデル（NoPE、(Haviv et al. 2022)）については、Jie Wang et al. (2024) が、系列長とともにアテンションエントロピーが増大すること、またヘッドごとの温度スケーリングによってこれを打ち消せることを示した。この仕組みは Llama 4 (Meta AI 2025) の NoPE グローバル層において推論時に採用されている。最後に、長文ドキュメント全体でのパープレキシティが低いからといって、ウィンドウ全体にわたる有効な検索が保証されるわけではない。実際、N. F. Liu et al. (2024) は、関連する事実が長いコンテキストの先頭や末尾付近ではなく中央付近に現れると正解率が大幅に低下する「**中間での埋没**」（lost in the middle）現象を実証している（21.5 節（英語版） 参照）。

10.8.4 コスト

10.8.4.1 初期の線形計算量近似

FlashAttention の登場以前は、数多くのアーキテクチャが $n \times n$ のアテンション行列を近似していた。たとえば、**Performer** (Choromanski et al. 2021) は正値直交ランダム特徴量を用いて指

数カーネルを近似し、**Linformer** (Wang et al. 2020) は低ランクの系列射影を適用し、**Reformer** (Kitaev et al. 2020) は局所性鋭敏型ハッシュ（locality-sensitive hashing）を用い、**Longformer** (Beltagy et al. 2020) と **Sparse Transformer** (Child et al. 2019) は局所ウィンドウにストライド付きトークンやグローバルトークンを組み合わせた（Tay et al. (2020) によるサーベイを参照）。

10.8.4.2 公開モデルにおけるウィンドウと内容適応型スパース性

Mistral 7B (Jiang et al. 2023) はすべての層にスライディングウィンドウを適用していたが（後に (Mistral AI 2023) で廃止）、後続のアーキテクチャでは局所ウィンドウと周期的なグローバル層を交互に配置している。たとえば、Gemma 2 (Gemma Team 2024) では 1:1、Gemma 3（$w = 1{,}024$、(Gemma Team 2025)）では 5:1、`gpt-oss` (OpenAI 2025) では $w = 128$ で 1:1 となっており、Llama 4 (Meta AI 2025) ではチャンク化された局所層と NoPE グローバル層を組み合わせている。

固定の位置ウィンドウにとどまらず、長文脈の訓練とデコーディングに向けて、ハードウェアに適合した**内容ベースのスパースアテンション**（content-based sparse attention、31.5 節（英語版））も登場している。その例として、**Native Sparse Attention**（NSA、(Yuan et al. 2025)）、**Mixture of Block Attention**（MoBA、(E. Lu et al. 2025)）、および **DeepSeek Sparse Attention** (DeepSeek-AI et al. 2025) が挙げられる。

10.8.4.3 FlashAttention-2、FlashAttention-3、FlexAttention

FlashAttention (Dao et al. 2022) に続く **FlashAttention-2** (Dao 2024) は、クエリの系列次元にわたって並列化し、キーループの最後まで正規化を遅延させる。Hopper GPU 上では、**FlashAttention-3** (Shah et al. 2024) が非同期の Tensor Core 行列積とワープ特化型のソフトマックス指数計算をオーバーラップさせ、インコヒーレントな直交事前回転によって FP8 をサポートする。**FlexAttention** (Dong et al. 2025) は任意のスコア変更クロージャとブロックマスクを融合 Triton カーネルへとコンパイルし、一方で並列 KV キャッシュデコーディングカーネル（29.2 節（英語版））はキー軸を分割して、オンラインソフトマックスにより部分状態 $(m_B, s_B, \mathbf{o}_B)$ を縮約する。

10.8.4.4 ゲート付き再帰とハイブリッド

現代の線形アテンションの派生手法は、第 12 章で扱う 2 つの改良によって記憶容量の課題に対処している。すなわち、(1) データ依存の忘却ゲート（**Gated Linear Attention**、(S. Yang, Wang, Shen, et al. 2024)、12.6 節）と、(2) 誤差修正型のデルタ更新（**DeltaNet**、(Schlag et al. 2021; S. Yang, Wang, Zhang, et al. 2024)、および **Gated DeltaNet**、(S. Yang et al. 2025)、12.7 節）である。実用アーキテクチャでは、厳密な連想想起を維持するために、これらのリカレント層と周期的な完全ソフトマックスアテンション層（通常は 4 層に 1 層から 8 層に 1 層の割合）を組み合わせている (Qwen Team 2025; Kimi Team 2025b; MiniMax 2025)（12.8 節）。

10.8.5 大規模モデルにおける回路

10.8.5.1 活性化パッチング、重ね合わせ、説明

ゼロ・平均アブレーション（10.7.3.3 節）の先へ進む手法として、**活性化パッチング**（activation patching）はクリーンな実行の活性化値を破損させた実行へと代入することで因果経路を切り分ける (K. Meng et al. 2022; K. Wang et al. 2023)。この手続きは ACDC (Conmy et al. 2023) や 1 次の**帰属パッチング**（attribution patching、(Nanda 2023; Kramár et al. 2024; Zhang and Nanda 2024; Heimersheim and Nanda 2024)）によって自動化されている。特徴量の数が残差ストリームの次元数を上回る**重ね合わせ**（superposition、(Elhage et al. 2022)）の状況では、**スパー**

スオートエンコーダと帰属グラフが多義的（polysemantic）な方向を単一意味（monosemantic）の特徴量へと分離する (Bricken et al. 2023; Cunningham et al. 2024; Ameisen et al. 2025)（18.6 節（英語版））。また、こうした介入的ツールは、アテンション重みが忠実な説明になり得るかをめぐる Jain and Wallace (2019) と Wiegreffe and Pinter (2019) の古典的な論争にも決着をつけている。

10.8.5.2 アルゴリズムとしてのコンテキスト内学習

誘導ヘッド（induction head）によるコピー (Olsson et al. 2022) と並び、コンテキスト内回帰の問題族で訓練された Transformer は、勾配降下法、リッジ回帰、および最小二乗法のソルバを順伝播の中で直接実装できる (Garg et al. 2022; Oswald et al. 2023; Akyürek et al. 2023)。ただし、一般的なテキスト事前学習がどの程度まで暗黙的な勾配降下法を誘導するかについては、依然として議論が続いている (Shen et al. 2024)。

10.8.6 推奨される参考資料

- **視覚的な入門資料：**3Blue1Brown の **Attention in Transformers, Step-by-Step**（2024）、Jay Alammar の **The Illustrated Transformer**（2018）、Peter Bloem の **Transformers from Scratch**（2019）、Lilian Weng の **The Transformer Family Version 2.0**（2023）。
- **基礎および回路に関する専門文献：**Elhage らの **A Mathematical Framework for Transformer Circuits**（2021）、Olsson らの **In-Context Learning and Induction Heads**（2022）、Weiss・Goldberg・Yahav の **Thinking Like Transformers**（2021）およびこれに対応する Sasha Rush の **Transformer Puzzles**。

Chapter 11

Transformer

第 10 章では、マルチヘッドアテンション層、位置エンコーディング、および長文脈へスケールするアテンションの派生手法を構築した。しかし、アテンション単体ではトークン混合演算子（token-mixing operator）にすぎない。トークンごとの非線形変換、残差接続、および正規化がなければ、アテンション層を積み重ねても訓練可能な深層ネットワークは得られないからである。本章では、アテンションを完全な **Transformer** アーキテクチャへと組み上げ (Vaswani et al. 2017)、その訓練・サービング・スケーリングの仕組みを考察する。

まず、因果的スタックを一から構築して訓練することから始める。11.1 節では、Pre-Norm 残差ストリーム（`RMSNorm` と `SwiGLU`）の内部でマルチヘッドアテンションと位置ごとのフィードフォワードネットワークを対にし、残差射影のスケーリング（$1/\sqrt{2L}$）およびクエリ・キー正規化（query–key normalization）が深さ方向にわたって信号ノルムとアテンションロジットを有界に保つ理由を示す。11.2 節では、これらの因果的ブロックをトークン埋め込みと重み共有（weight tying）を用いた言語モデリングヘッドの間に積み重ね、条件を揃えたリカレントベースラインと比較しながら WikiText-103 上でバイトレベルの GPT を訓練する。さらに、事前学習済みの GPT-2 の重み（`124M`）を読み込んで温度と top-k サンプリングを検証し、現代のオープンウェイト LLM に共通するアーキテクチャのレシピを抽出する（11.2.6 節）。素朴な自己回帰生成ではステップごとにプレフィックス全体を再計算するため、11.3 節ではキー・値キャッシュ（key–value cache、KV キャッシュ）を導入し、ハードウェアのルーフライン上で計算律速（compute-bound）なプレフィルとメモリ帯域幅律速（memory-bandwidth-bound）なデコードを対比させたうえで、ヘッド数（グループ化クエリアテンションとマルチクエリアテンション）、チャネル幅（低ランクのキー・値射影）、および系列長（アテンションシンクを伴うスライディングウィンドウ）にわたってキャッシュを圧縮する。

すべてのタスクが因果的なテキスト生成というわけではない。11.4 節では、因果的デコーダを、マスク言語モデリング（masked language modeling）によって訓練される双方向エンコーダ (Devlin et al. 2018) や、クロスアテンションで結合された系列対系列のエンコーダ・デコーダ（固定サイズの潜在ボトルネックを含む (Jaegle et al. 2021)）と対比させる。続く 11.5 節では、重複のないピクセルパッチをトークン埋め込みへ射影することで双方向エンコーダを画像に直接適用し (Dosovitskiy et al. 2021)、学習された 2D 位置構造とデータ効率を、パラメータ数を揃えた ResNet と比較する。

最後に、スケーリングと条件付き計算（conditional computation）に目を向ける。11.6 節では、パラメータ数（$N \approx 12Ld^2$）と訓練 FLOPs（$C \approx 6ND$）を数え上げ、15 モデルのグリッドを訓練し、公開されている Chinchilla のべき乗則フィット (Kaplan et al. 2020; Hoffmann et al. 2022) を用いて IsoFLOP 曲線と計算量最適（compute-optimal）な配分を描く。あわせて、運用

期間全体の推論コスト（inference cost）を考慮すると、現代の事前学習がなぜ Chinchilla のパラメータあたりトークン比をはるかに超えて訓練された、より小さなモデルへと向かうのかを説明する。さらに 11.7 節では、密なフィードフォワードネットワークをルーティングされるエキスパート群に置き換えることで保持パラメータ容量とトークンあたり FLOPs を切り離し (Shazeer et al. 2017; Fedus et al. 2022)、補助損失による負荷分散と損失を用いないルータバイアス制御を比較する (L. Wang et al. 2024)。そして 11.8 節では、正規化の配置、潜在アテンション（latent attention）、および投機的デコーディング（speculative decoding）にわたる拡張手法を概観する。

11.1 Transformer ブロック

自己アテンション（self-attention、第 10 章）は、系列の位置間で情報を混合する。すなわち、各トークンの出力は、アテンションを向ける位置の値ベクトルの重み付き平均となる。しかし、アテンション層単体では、その重み付き和の前後に線形射影（W_Q, W_K, W_V, W_O）を適用するにすぎない。位置ごとの非線形性、残差接続、および正規化がなければ、アテンション層を積み重ねてもトークンごとの階層的な特徴量を構築できず、数十層にわたって信号を安定して伝播させることもできない。

Transformer ブロック (Vaswani et al. 2017) は、マルチヘッドアテンションと位置ごとの**フィードフォワードネットワーク（FFN）**を組み合わせ、各サブ層を加算的な**残差接続**（residual connection）で包み (He et al. 2016a)、**層正規化**（layer normalization）によって活性化値を安定化させる（Figure 11.1）。現代のブロックと 2017 年の元の Transformer を分ける設計上の改良点は、次の 3 つである。

1. **正規化の配置：**正規化を残差ストリーム（residual stream）自体（**Post-Norm**）から各サブ層の入力（**Pre-Norm**）へと移すことで、深さ方向に減衰しない恒等経路が保たれる。
2. **正規化の形式：**平均中心化を伴う `LayerNorm` を二乗平均平方根（root-mean-square）による正規化（`RMSNorm`）と任意のヘッドごとの**クエリ・キー正規化**（query–key normalization、`QK-Norm`）に置き換えることで、メモリ帯域幅のオーバーヘッドを抑えつつ活性化値とアテンションロジットを有界に保つ。
3. **ゲート付きフィードフォワードネットワーク：**2 行列の ReLU・GELU MLP（$d_{\text{ff}} = 4d$）をゲート付きの `SwiGLU` ネットワーク（$d_{\text{ff}} \approx \frac{8}{3}d$）に置き換えることで、まったく同じ $8d^2$ のパラメータ数と FLOP 予算のまま表現の質を高める。

11.1.1 Transformer ブロックの構成要素

Transformer ブロックは、共有された**残差ストリーム** $\mathbf{X} \in \mathbb{R}^{B\times n\times d}$（10.7 節）上で動作する。連続する 2 つのサブ層は $\mathbf{X}$ を上書きするのではなく、ストリームから値を読み取り、更新量を計算して、その結果をストリームに加算して戻す。

- **マルチヘッドアテンション（トークン混合、**token mixing）は、系列の位置どうしが相互作用する唯一の箇所である。各トークンは系列にクエリを送り（10.3 節）、取得したコンテキストベクトルを自身のストリームに加算する。
- **位置ごとのフィードフォワードネットワーク（チャネル混合、**channel mixing）は、同じ MLP を各トークンベクトル $\mathbf{x}_i \in \mathbb{R}^d$ に独立に適用し、アテンションが集約した特徴量を変換する。密なブロックのパラメータ数の約 3 分の 2 は FFN に存在する。

現代の Transformer で用いられる **Pre-Norm** 配置（Figure 11.1）では、各サブ層は更新量を計算する前に自身の入力を正規化する。

$$\begin{aligned}\mathbf{H} &= \mathbf{X} + \mathrm{Attn}\big(\mathrm{Norm}(\mathbf{X})\big),\\ \mathbf{Y} &= \mathbf{H} + \mathrm{FFN}\big(\mathrm{Norm}(\mathbf{H})\big).\end{aligned} \tag{11.1}$$

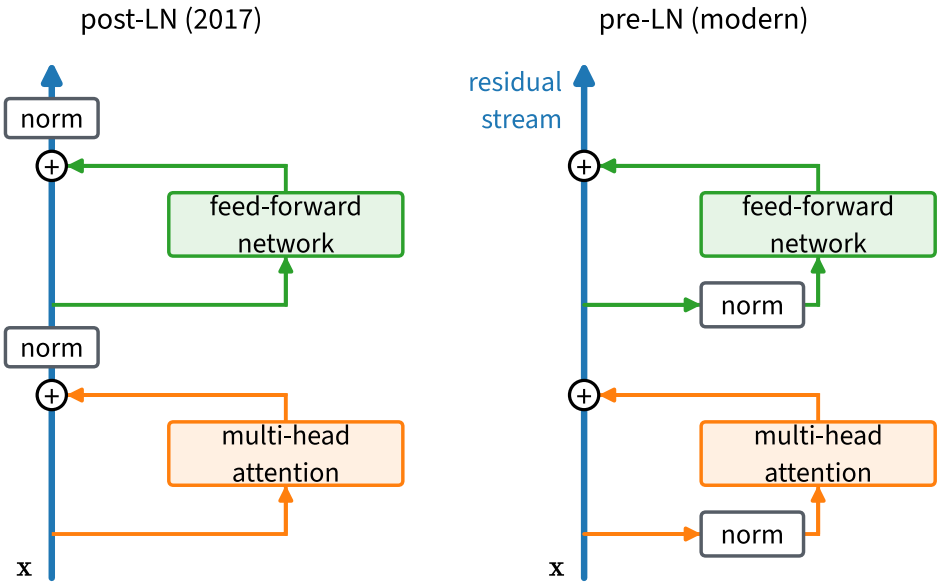


Figure 11.1: Transformer ブロック。アテンションと位置ごとの FFN が残差ストリームから値を読み取り、計算結果をストリームに加算して戻す。左は 2017 年の配置であり、正規化は各加算の後にストリーム自体の上に置かれる。右は現代の配置であり、正規化は各ブランチの上へと移り、ストリームは入力から出力まで途切れることなく通る。

11.1.2 正規化

11.1.2.1 Pre-Norm と Post-Norm

2017 年の元の Transformer (Vaswani et al. 2017) と BERT (Devlin et al. 2018) は **Post-Norm** を採用しており、各加算の後に**残差ストリーム自体の上**に正規化を置いていた。

$$\begin{aligned}\mathbf{H} &= \mathrm{Norm}\big(\mathbf{X} + \mathrm{Attn}(\mathbf{X})\big),\\ \mathbf{Y} &= \mathrm{Norm}\big(\mathbf{H} + \mathrm{FFN}(\mathbf{H})\big).\end{aligned} \tag{11.2}$$

なぜ現代のアーキテクチャは Equation 11.2 から Equation 11.1 へと切り替わったのだろうか。最小限のブロック `MiniBlock`（フラグ `pre_norm` で 2 つの配置を切り替える）を使えば、1 ステップも訓練することなく初期化の段階でその違いを直接確認できる。

```
class MiniBlock(nn.Module):
    """Attention + MLP on a residual stream; norm placement as a flag."""
    def __init__(self, num_hiddens, pre_norm):
        super().__init__()
        self.pre_norm = pre_norm
```

```
        self.attention = d2l.MultiHeadAttention(num_hiddens, 4)
        self.W_1 = nn.Linear(num_hiddens, 4 * num_hiddens)
        self.W_2 = nn.Linear(4 * num_hiddens, num_hiddens)
        self.norm1 = nn.LayerNorm(num_hiddens, eps=1e-5)
        self.norm2 = nn.LayerNorm(num_hiddens, eps=1e-5)

    def ffn(self, X):  # exact GELU
        return self.W_2(F.gelu(self.W_1(X)))

    def forward(self, X):
        if self.pre_norm:
            Y = self.norm1(X)
            X = X + self.attention(Y, Y, Y)
            return X + self.ffn(self.norm2(X))
        X = self.norm1(X + self.attention(X, X, X))
        return self.norm2(X + self.ffn(X))
```

11.1.2.2 初期化時の信号伝播とトークン崩壊

単位分散をもつランダムなトークンベクトルのバッチ $\mathbf{X}^{(0)} \in \mathbb{R}^{B\times n\times d}$ を、初期化したばかりの $L = 32$ 個のブロックに通すことを考えよう。以下の関数 `signal_stats` は、積層された各ブロックのすべての重み行列を、2017 年の Transformer と同じ Xavier 初期化で、あるいは $\mathcal{N}(0, \text{std}^2)$ の要素（`std` が指定された場合）で新たにサンプリングする。各ブロック $\ell \in \{1, \dots, L\}$ において、次の 3 つの量を追跡する。

1. **ストリーム標準偏差**（$\mathbf{X}^{(\ell)}$ の全要素にわたる標準偏差）。
2. **トークンの広がり**（token spread）。n 個の系列位置にわたる標準偏差として測られ、n 個すべてのトークンが同じベクトルへと崩壊すると消失する。
3. **クエリ勾配ノルム** $\|\partial \mathcal{L}/\partial W_q^{(\ell)}\|_F$（スタック最上部の単位スカラー損失から逆伝播してくる勾配のノルム）。

```
def signal_stats(pre_norm, std=None, num_blks=32, num_hiddens=256):
    """Token spread, stream RMS, and gradient norms of a fresh stack whose
    weight matrices are Xavier, or N(0, std^2) if std is given."""
    blks = nn.ModuleList([MiniBlock(num_hiddens, pre_norm)
                          for _ in range(num_blks)])
    init = (nn.init.xavier_uniform_ if std is None  # the 2017 initialization
            else lambda W: nn.init.normal_(W, std=std))
    for p in blks.parameters():
        if p.ndim == 2:
            init(p)
    norm = torch.linalg.vector_norm
    H = torch.randn(2, 64, num_hiddens)
    spread, rms = [], []
    for blk in blks:
        H = blk(H)
        spread.append(norm(H - H.mean(1, keepdim=True)) / norm(H))
        rms.append(H.pow(2).mean().sqrt())
    (H * torch.randn_like(H)).mean().backward()  # generic readout loss
    grad_q = [norm(blk.attention.W_q.weight.grad) for blk in blks]
    proj = {n: norm(getattr(blks[-1].attention, n).weight.grad).item()
```

```
            for n in ('W_q', 'W_k', 'W_v', 'W_o')}
    return {'spread': torch.stack(spread).tolist(),
            'rms': torch.stack(rms).tolist(),
            'grad_q': torch.stack(grad_q).tolist(), 'proj': proj}
```

なぜ Post-Norm では深さとともにトークンの多様性が失われるのだろうか。各トークンを全トークンが共有する成分と固有の成分に分け、$\mathbf{x}_i = \mu + \delta_i$ と書く。ここで $\mu = \frac{1}{n}\sum_j \mathbf{x}_j$ は n 個の位置にわたる平均である（Figure 11.2）。初期化時、アテンション重みはほぼ一様であるため、どのトークンもほぼ同じ更新量 $\mathbf{u} \approx \mathbf{W}_o\mathbf{W}_v\mu$ を受け取る。すなわち、偏差は平均化されて打ち消し合い、平均成分だけが先へ伝わる。2 つの射影はそれぞれ入力の長さの二乗を $g^2 = d\sigma^2$ 倍（σ^2 は重みの各要素の分散）にし、確率ベクトル $\mathbf{u}$ は μ とほぼ直交する。したがって、共有成分のエネルギーは $1 + g^4$ 倍に増大する一方、偏差は元のままである。続く正規化ではこれを修復できない。すべてのトークンを同じ係数で再スケーリングするため、ストリームの長さは元に戻るものの、平均と偏差のバランスは戻らないからである。

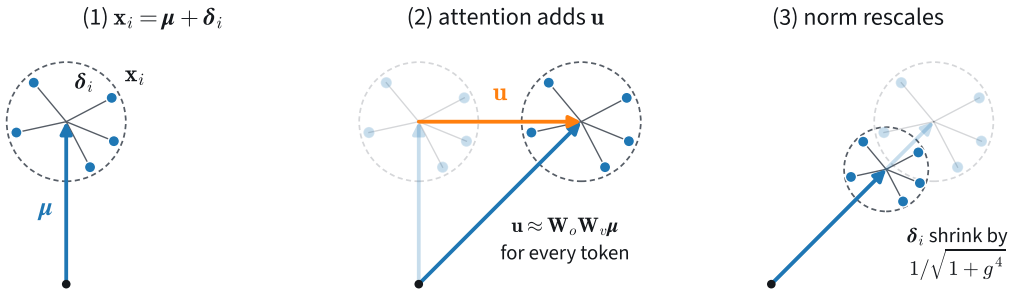


Figure 11.2: 1 つの Post-Norm ブロックにおけるトークン崩壊。各トークンは共有平均 μ と固有の偏差 δ_i に分解される。ほぼ一様なアテンションがすべてのトークンに同じ更新量 $\mathbf{u} \approx \mathbf{W}_o\mathbf{W}_v\mu$ を加算するため、平均だけが増大する。続く正規化が全トークンを単一の係数で再スケーリングすることで、偏差が縮小する。Xavier 初期化（$g = 1$）の場合を図示しており、このとき $\mathbf{u}$ は μ と同じ長さをもち、係数は $1/\sqrt{2}$ となる。

共有エネルギーとトークン固有のエネルギーの比を $\rho = n\|\mu\|^2 / \sum_i \|\delta_i\|^2$ とする。各ブロックは ρ を $1 + g^4$ 倍にする。ストリームのノルムのうち偏差が占める割合であるトークンの広がりは $s = (1+\rho)^{-1/2}$ であり、n 個の独立なランダムトークンは $\rho_0 = 1/(n-1)$ から出発する。したがって、k ブロック後には次のようになる。

$$s_k \approx \left(1 + \frac{(1+g^4)^k}{n-1}\right)^{-1/2}. \tag{11.3}$$

ひとたび平均が支配的になると、広がりはブロックごとに $1/\sqrt{1+g^4}$ 倍に縮小し、その値は $1/\sqrt{2}$ となる（Xavier 初期化、すなわち $g = 1$ の場合）。なお、Equation 11.3 に FFN は現れない。FFN は各トークンに個別に作用し、平均と偏差を区別できないため、両者の比をほとんど変えないからである（演習問題 5）。

Pre-Norm（Equation 11.1）のもとでは、残差ストリームが正規化の分母で割られることはない。L 個のブロックを展開すると、直接の和 $\mathbf{X}^{(L)} = \mathbf{X}^{(0)} + \sum_{\ell=1}^{2L} \mathbf{U}^{(\ell)}$ となる。$2L$ 個の独立なサブ層出力がストリームに分散を加えるため、スケーリングなしの Pre-Norm ストリームの RMS は

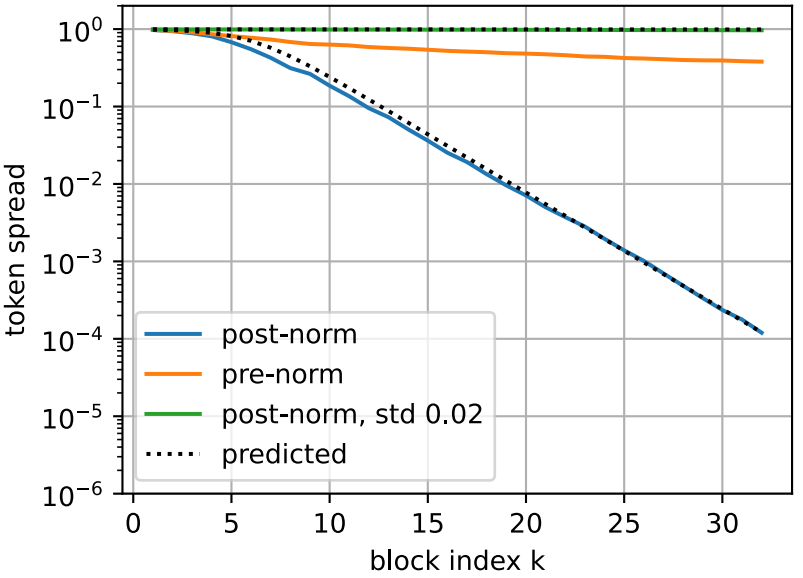
深さとともに $\sqrt{1+2\ell\sigma^2}$ のように増大する（全 $2L$ 個のサブ層が加える合計分散が $O(1)$ となるよう任意の深さ L で抑えるため、GPT-2 (Radford et al. 2019) は各ブロックの残差出力射影 `W_o` と `W_2` の初期化標準偏差を $1/\sqrt{2L}$ 倍にスケーリングしており、これは 11.2 節で実装し、演習問題 4 で検証する）。一方、Post-Norm において BERT (Devlin et al. 2018) は、Equation 11.3 の崩壊を遅らせるためにすべての線形重みをより小さな標準偏差 $\sigma = 0.02$ で初期化する（これにより $g^2 \approx 0.10$、$g^4 \approx 0.01$ となる。ただし $d = 256$ の場合）。これら 3 つの構成（`post-norm`、`pre-norm`、`post-norm, std 0.02`）を 32 ブロックにわたって比較してみよう。

```
arms = {'post-norm': (False, None), 'pre-norm': (True, None),
        'post-norm, std 0.02': (False, 0.02)}
stats, predicted = {}, {}
at = lambda xs, ks=(8, 16, 32): ' '.join(f'{xs[k - 1]:.1e}' for k in ks)
for name, (pre_norm, std) in arms.items():
    stats[name] = s = signal_stats(pre_norm, std)
    print(f'{name}: stream RMS at k=32 {s["rms"][-1]:.2f}')
    print(f'  token spread at k=8,16,32: {at(s["spread"])}')
    if not pre_norm:  # rho_0 = 1/63 for 64 random tokens; Xavier: g = 1
        g2 = 1 if std is None else 256 * std ** 2
        predicted[name] = [(1 + (1 + g2 ** 2) ** k / 63) ** -0.5
                           for k in range(1, 33)]
        print(f'  predicted:                {at(predicted[name])}')
    print(f'  W_q grad norm at k=1,8,16,32: {at(s["grad_q"], (1, 8, 16, 32))}')
    print('  grad norms at k=32: ' + '  '.join(
        f'{n} {g:.1e}' for n, g in s['proj'].items()))
```

```
post-norm: stream RMS at k=32 1.00
  token spread at k=8,16,32: 3.1e-01 2.5e-02 1.2e-04
  predicted:                 4.4e-01 3.1e-02 1.2e-04
  W_q grad norm at k=1,8,16,32: 1.3e-02 2.3e-03 1.8e-05 4.1e-09
  grad norms at k=32: W_q 4.1e-09  W_k 4.1e-09  W_v 6.4e-02  W_o 5.9e-02
pre-norm: stream RMS at k=32 5.12
  token spread at k=8,16,32: 6.8e-01 5.2e-01 3.8e-01
  W_q grad norm at k=1,8,16,32: 6.2e-02 2.0e-02 7.9e-03 2.3e-03
  grad norms at k=32: W_q 2.3e-03  W_k 2.3e-03  W_v 8.1e-02  W_o 8.4e-02
post-norm, std 0.02: stream RMS at k=32 1.00
  token spread at k=8,16,32: 9.9e-01 9.8e-01 9.7e-01
  predicted:                 9.9e-01 9.9e-01 9.9e-01
  W_q grad norm at k=1,8,16,32: 4.1e-04 4.0e-04 4.0e-04 4.0e-04
  grad norms at k=32: W_q 4.0e-04  W_k 4.1e-04  W_v 6.9e-03  W_o 7.1e-03
```

ブロック $k = 1, \ldots, 32$ にわたるトークンの広がりの実測値を、2 つの Post-Norm 条件に対する Equation 11.3 の予測（点線、$n = 64$）と並べてプロットする。

```
d2l.plot(list(range(1, 33)), [s['spread'] for s in stats.values()]
         + list(predicted.values()), 'block index k', 'token spread',
         legend=list(arms) + ['predicted'], yscale='log', ylim=(1e-6, 2),
         fmts=('C0-', 'C1-', 'C2-', 'k:', 'k:'), figsize=(4.5, 3.3),
         legend_loc='lower left')
```

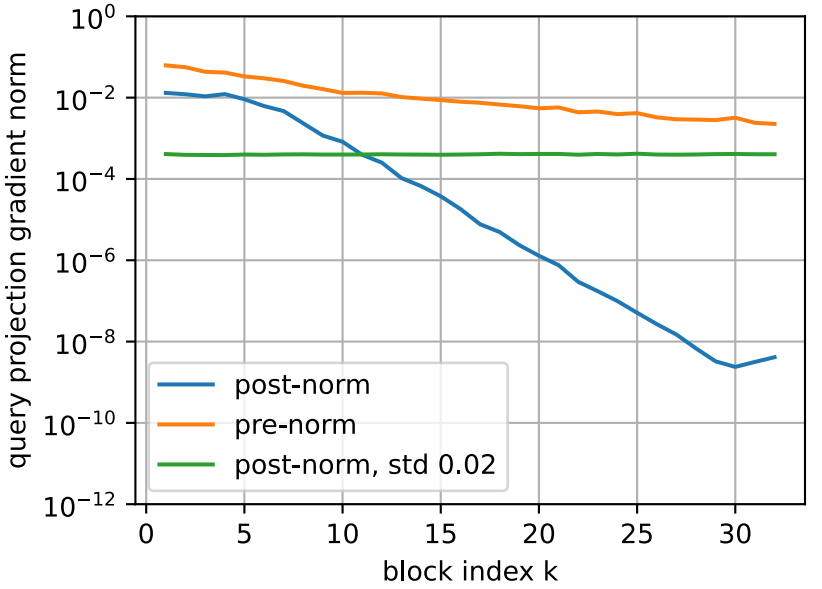


標準的な Xavier 初期化による Post-Norm（`post-norm`）では、トークンの広がりはブロック 32 までにほぼゼロ（$\sim 10^{-4}$）へと崩壊し、Equation 11.3 の予測によく一致する。系列内のすべてのトークンが、ほぼ同一のベクトルへと押し潰されているのである。実測曲線が点線をわずかに下回っているのは、FFN による小さな寄与である。Post-Norm の初期化を縮小すると（`post-norm, std 0.02`）、ブロックごとの収縮が十分に遅くなり、32 ブロックにわたってトークンの広がりが保たれる。Pre-Norm（`pre-norm`）も無縁ではなく、アテンションサブ層は同様に共有平均へ加算する。しかしストリーム自体が深さとともに増大するため、各更新の相対的な重みは前の層より小さくなり、広がりの減衰は緩やかである。Xavier スケールであっても 32 ブロック後に 3 分の 1 以上が残っている（その代わり、ストリームの RMS は $\sqrt{L}$ のオーダーで増大するが、これは $1/\sqrt{2L}$ の出力射影スケーリングと組み合わせることで抑えられる。演習問題 4 を参照）。

11.1.2.3 初期化時の勾配

トークン崩壊（token collapse）は順伝播の性質であるが、どの重みが学習できるかも決定づける。クエリ射影の勾配ノルム $\|\partial \mathcal{L}/\partial W_q^{(\ell)}\|_F$ をブロック番号 $\ell \in \{1, \dots, 32\}$ にわたってプロットする。

```
d2l.plot(list(range(1, 33)), [s['grad_q'] for s in stats.values()],
         'block index k', 'query projection gradient norm',
         legend=list(arms), yscale='log', ylim=(1e-12, 1),
         fmts=('C0-', 'C1-', 'C2-'), figsize=(4.5, 3.3),
         legend_loc='lower left')
```



`post-norm` では、クエリ勾配はブロック 1 で最大（10^{-2} 程度）となり、ブロック 32 では約 10^{-9} まで落ち込む。これは、損失から最も遠い層ほど受け取る勾配が小さくなる深い MLP の勾配消失とは逆の現象である。勾配は逆伝播の途中で失われているのではない。深いブロックが順伝播で受け取る入力に原因があるのである。ブロックに入るトークンどうしがほぼ一致してしまうと、どのキーも同じスコアを受け取り、どの値も同じになるため、クエリを動かしても出力は何も変化しない。その結果、クエリ勾配はトークンの広がりの二乗に比例して縮小し（スコアが偏差 δ_i を通じてしか異ならないことで 1 乗分、値も同様であることでさらに 1 乗分縮小する）、キー勾配もそれに伴って縮小する。他の重みは影響を受けない。ブロック 32 における `W_v` と `W_o` の勾配は、`post-norm` と `pre-norm` で同じ桁の大きさにある（上の各条件の出力の最終行）。

`pre-norm` では、恒等項 I（$\partial \mathbf{X}^{(L)}/\partial \mathbf{X}^{(\ell)} = I + \cdots$ に現れる項）が損失からの勾配をすべてのブロックへと運ぶ。こちらのクエリ勾配も一様ではなく、トークンの広がりが緩やかに減衰するにつれてスタック全体で約 30 分の 1 に低下する。それでも、Post-Norm が 6 桁以上も失うのに対し、10^{-3} 以上を保っている。Xiong et al. (2020) は、初期化時の Post-Norm Transformer における別の不均衡、すなわち出力付近のパラメータに対する勾配が大きくなる現象を解析し、Post-Norm の訓練が学習率のウォームアップに依存する一方で Pre-Norm はそれなしで訓練できる理由がここにあることを示している。

11.1.2.4 RMSNorm

標準的な LayerNorm（7.3 節）は、各トークンベクトル $\mathbf{x} \in \mathbb{R}^d$ からその平均 $\mu = \frac{1}{d}\sum_j x_j$ を引いて中心化し、標準偏差でスケーリングしたうえで、学習されるゲイン γ とバイアス β を適用する。Zhang and Sennrich (2019) は、平均中心化と加算バイアスは訓練の安定性にほとんど寄与せず、重要なのは二乗平均平方根の大きさを正規化することだと示した。μ と β を取り除くと **RMSNorm** が得られる。

$$\begin{aligned}\mathrm{RMSNorm}(\mathbf{x}) &= \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \odot \gamma, \\ \mathrm{RMS}(\mathbf{x}) &= \sqrt{\frac{1}{d}\sum_{j=1}^{d} x_j^2 + \epsilon},\end{aligned} \tag{11.4}$$

ここで、$\gamma \in \mathbb{R}^d$ は 1 で初期化され、$\epsilon = 10^{-5}$ である。RMSNorm はトークンごとに平均と分散の両方ではなく二乗平均の縮約 $\frac{1}{d}\sum_j x_j^2$ を 1 回計算するだけで済むため、LayerNorm と同等の正解率を保ちながらメモリ上の縮約演算を減らせる。

```
def rms_norm(X, weight=1, eps=1e-5):
    return X / torch.sqrt((X * X).mean(-1, keepdim=True) + eps) * weight

x = torch.tensor([1.0, 2.0, 3.0, 6.0])
rmsnorm, layernorm = nn.RMSNorm(4, eps=1e-5), nn.LayerNorm(4, eps=1e-5)
show = lambda v: ' '.join(f'{a:5.2f}' for a in v.tolist())
print(f'ours vs. nn.RMSNorm: max deviation '
      f'{(rms_norm(x) - rmsnorm(x)).abs().max():.1e}')
for name, v in (('x', x), ('3x', 3 * x), ('x + 10', x + 10)):
    print(f'{name:>6}: RMSNorm {show(rms_norm(v))} | '
          f'LayerNorm {show(layernorm(v))}')
```

```
ours vs. nn.RMSNorm: max deviation 0.0e+00
     x: RMSNorm  0.28  0.57  0.85  1.70 | LayerNorm -1.07 -0.53  0.00  1.60
    3x: RMSNorm  0.28  0.57  0.85  1.70 | LayerNorm -1.07 -0.53  0.00  1.60
x + 10: RMSNorm  0.84  0.91  0.99  1.22 | LayerNorm -1.07 -0.53  0.00  1.60
```

11.1.2.5 クエリ・キー正規化（QK-Norm）

Pre-Norm はアテンションサブ層に入る入力 $\mathbf{X}$ を正規化するが、アテンション内部ではクエリ $\mathbf{Q} = \mathrm{Norm}(\mathbf{X})\mathbf{W}_q$ とキー $\mathbf{K} = \mathrm{Norm}(\mathbf{X})\mathbf{W}_k$ に学習される射影行列 $\mathbf{W}_q, \mathbf{W}_k$ が掛けられる。訓練中に $\mathbf{W}_q$ と $\mathbf{W}_k$ の特異値が増大すると、スケール付きドット積ロジット $\mathbf{q}_i^\top \mathbf{k}_j / \sqrt{d_h}$ は重みのスケールに対して二次関数的に増大し、ソフトマックスがワンホットな飽和状態に追い込まれて勾配が消失する (Henry et al. 2020; Dehghani et al. 2023)。

QK-Norm は、アテンションスコアを計算する直前に、ヘッドごとの幅 d_h の `RMSNorm` を $\mathbf{Q}$ と $\mathbf{K}$ に適用する。コーシー・シュワルツの不等式により、単位 RMS をもつ 2 つのベクトルのドット積は大きさが高々 d_h に抑えられるため、スケール付きロジット $\mathbf{q}_i^\top \mathbf{k}_j / \sqrt{d_h}$ の絶対値は $\sqrt{d_h}$ 以下に有界となり（さらにランダムな方向に対しては単位分散をもつ）、これは $\|\mathbf{W}_q\|$ や $\|\mathbf{W}_k\|$ がどれほど大きくなっても変わらない。

```
def logit_stats(Q, K):
    """Std of the scaled logits; mean entropy of each query's weights."""
    S = Q @ K.T / math.sqrt(Q.shape[-1])
    entropy = -(F.softmax(S, -1) * F.log_softmax(S, -1)).sum(-1).mean()
    return S.std(), entropy

d, d_h = 256, 64
X = torch.randn(64, d)                                  # 64 random tokens
```

```
W_q, W_k = torch.randn(2, d, d_h) / math.sqrt(d)      # one head
for lam in (1, 4, 16):
    Q, K = X @ (lam * W_q), X @ (lam * W_k)
    s, h = logit_stats(Q, K)
    s_n, h_n = logit_stats(rms_norm(Q), rms_norm(K))
    print(f'lambda {lam:2d}: logit std {s:5.1f}, entropy {h:.2f} | '
          f'QK-norm: logit std {s_n:.2f}, entropy {h_n:.2f}')
```

```
lambda  1: logit std   1.0, entropy 3.69 | QK-norm: logit std 1.01, entropy 3.69
lambda  4: logit std  16.3, entropy 0.25 | QK-norm: logit std 1.01, entropy 3.69
lambda 16: logit std 261.4, entropy 0.01 | QK-norm: logit std 1.01, entropy 3.69
```

QK-Norm なしで Q と K を $\lambda = 4\times$ および $\lambda = 16\times$ にスケーリングすると、ロジットの標準偏差は二次関数的に跳ね上がり（≈ 16 および ≈ 256）、ソフトマックスのエントロピーはゼロに向かって崩壊する。一方、`rms_norm` を Q と K に適用すれば、ロジットの広がりとアテンションのエントロピーは重みのスケールに対して完全に不変となる。

11.1.3 フィードフォワードネットワーク：GELU から SwiGLU へ

元の Transformer のフィードフォワードネットワークは、チャネル幅を d から $d_{\text{ff}} = 4d$ へと拡大し、要素ごとの活性化関数（ReLU が Vaswani et al. (2017) で、GELU が GPT-2 と BERT で使われた）を適用してから d へと射影し直す 2 層の MLP である。

$$\begin{aligned} \text{FFN}_{\text{GELU}}(\mathbf{x}) &= \text{GELU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2, \\ \mathbf{W}_1 \in \mathbb{R}^{d\times 4d}, &\qquad \mathbf{W}_2 \in \mathbb{R}^{4d\times d}, \end{aligned} \tag{11.5}$$

これは $2 \cdot d \cdot 4d = 8d^2$ 個の重みパラメータを用いる。

Dauphin et al. (2017) および Shazeer (2020) に従い、現代の Transformer は Equation 11.5 をバイアスなしの **SwiGLU** ゲート付き線形ユニット（gated linear unit）に置き換える。これは、SiLU で活性化したゲート射影 $\mathbf{x}\mathbf{W}_g$（$\text{SiLU}(z) = z\sigma(z)$）と線形アップ射影 $\mathbf{x}\mathbf{W}_1$ の要素ごとの積をとってから、$\mathbb{R}^d$ へと $\mathbf{W}_2$ で射影し直すものである。

$$\text{FFN}_{\text{SwiGLU}}(\mathbf{x}) = \big(\text{SiLU}(\mathbf{x}\mathbf{W}_g) \odot (\mathbf{x}\mathbf{W}_1)\big)\mathbf{W}_2. \tag{11.6}$$

`SwiGLU` は形状 $d \times d_{\text{ff}}$（$\mathbf{W}_g, \mathbf{W}_1$）と $d_{\text{ff}} \times d$（$\mathbf{W}_2$）の **3** つの行列を使うため、$d_{\text{ff}} = \text{round}(\frac{8}{3}d)$ と設定すれば $3dd_{\text{ff}} \approx 8d^2$ となり、$4d$ の GELU MLP とパラメータ数・FLOP 数の両方が一致する。

```
class FeedForward(nn.Module):
    """Position-wise FFN: GELU MLP or SwiGLU at matched parameter count."""
    def __init__(self, num_hiddens, act='swiglu', bias=False):
        super().__init__()
        assert act in ('swiglu', 'gelu'), f'unknown act: {act!r}'
        self.act = act
        if act == 'gelu':
            width = 4 * num_hiddens
        else:  # 'swiglu': three matrices; width 8d/3 matches the MLP budget
            width = round(8 * num_hiddens / 3)
            self.W_g = nn.Linear(num_hiddens, width, bias=bias)
```

```
        self.W_1 = nn.Linear(num_hiddens, width, bias=bias)
        self.W_2 = nn.Linear(width, num_hiddens, bias=bias)

    def forward(self, X):
        if self.act == 'gelu':  # The tanh approximation, as in GPT-2
            return self.W_2(F.gelu(self.W_1(X), approximate='tanh'))
        return self.W_2(F.silu(self.W_g(X)) * self.W_1(X))
```

11.1.4 設定可能な TransformerBlock

ここで TransformerBlock を組み立てよう。このクラスは、Pre-Norm または Post-Norm の配置（pre_norm）、RMSNorm または LayerNorm（norm）、SwiGLU または GELU の FFN（act）、バイアスの切り替え（bias）、および差し替え可能な attn_factory と ffn_factory のフックをひとまとめにしたものである。これらのフックにより、後の節ではブロックを書き直すことなく、11.2 節の因果的 RoPE アテンションや 11.7 節の混合エキスパート（mixture of experts）層をそのまま組み込める。

```
class TransformerBlock(nn.Module):
    """Configurable transformer block: attention + FFN on a residual
    stream."""
    def __init__(self, num_hiddens, num_heads, *, dropout=0, norm='rms',
                 act='swiglu', pre_norm=True, bias=False, attn_factory=None,
                 ffn_factory=None):
        super().__init__()
        assert norm in ('rms', 'layer'), f'unknown norm: {norm!r}'
        assert num_hiddens % num_heads == 0
        self.pre_norm = pre_norm
        make_norm = nn.RMSNorm if norm == 'rms' else nn.LayerNorm
        self.norm1 = make_norm(num_hiddens, eps=1e-5)
        self.norm2 = make_norm(num_hiddens, eps=1e-5)
        self.attention = (d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout, bias=bias)
                          if attn_factory is None else attn_factory())
        self.ffn = (FeedForward(num_hiddens, act, bias=bias)
                    if ffn_factory is None else ffn_factory())
        self.dropout = nn.Dropout(dropout)

    def forward(self, X, mask=None, **attn_args):
        # attn_args go on to the attention, e.g. the RoPE tables of d2l.GPT
        if self.pre_norm:
            Y = self.norm1(X)
            X = X + self.dropout(self.attention(Y, Y, Y, mask, **attn_args))
            return X + self.dropout(self.ffn(self.norm2(X)))
        X = self.norm1(X + self.dropout(self.attention(X, X, X, mask,
                                                       **attn_args)))
        return self.norm2(X + self.dropout(self.ffn(X)))
```

11.1.4.1 パラメータ数

デフォルトの現代的なブロック（RMSNorm、SwiGLU、bias=False）を 2017 年版・GPT-2 版の変種（GELU、LayerNorm、bias=True）と $d = 256, H = 8$ で比較すると、どの構成も $\approx 12d^2 = 786{,}432$

個のパラメータ（アテンションに $4d^2$、FFN に $8d^2$）をもつことが確認できる。

```
blocks = {'default': TransformerBlock(256, 8),
          'GELU': TransformerBlock(256, 8, act='gelu'),
          '2017': TransformerBlock(256, 8, norm='layer', act='gelu',
                                   pre_norm=False, bias=True)}
for name, blk in blocks.items():
    attn, ffn, total = (d2l.num_params(m)
                        for m in (blk.attention, blk.ffn, blk))
    print(f'{name:>7}: attention {attn:,}, FFN {ffn:,}, total {total:,} '
          f'= {total / 256 ** 2:.2f} d^2')
```

```
default: attention 262,144, FFN 524,544, total 787,200 = 12.01 d^2
   GELU: attention 262,144, FFN 524,288, total 786,944 = 12.01 d^2
   2017: attention 263,168, FFN 525,568, total 789,760 = 12.05 d^2
```

11.1.5 まとめ

Transformer ブロックは、**トークン混合**（系列の位置間にわたるマルチヘッドアテンション）と**チャネル混合**（d 個の特徴量にわたる位置ごとのフィードフォワードネットワーク）を分離し、両方の更新量を共有された残差ストリームに加算する。**Pre-Norm**（Equation 11.1）は各サブ層への入力のみを正規化して残差ストリームを途切れのない恒等経路として残すのに対し、**Post-Norm**（Equation 11.2）はストリーム自体を正規化する。Xavier スケールでは、深い Post-Norm スタックはトークン間の差異を指数関数的な速さで失い（Equation 11.3）、上層ブロックのクエリ勾配とキー勾配を枯渇させる一方、各ブロックの残差出力射影（`W_o` と `W_2`）を $1/\sqrt{2L}$ 倍にスケーリングすれば、Pre-Norm のストリーム分散は $O(1)$ のまま任意の深さ L にわたって保たれる。現代のブロックは、LayerNorm の平均減算と加算バイアスを省いた **RMSNorm**（Equation 11.4）とヘッドごとの **QK-Norm** を組み合わせることで、射影重みの増大にかかわらずコーシー・シュワルツの不等式によってアテンションロジットを有界に保つ。最後に、**SwiGLU**（Equation 11.6）は 2 行列・幅 $4d$ のフィードフォワードネットワークを、幅 $d_{\text{ff}} = \text{round}(\frac{8}{3}d)$ の 3 行列によるゲート付き積に置き換え、標準的な FFN と同じ $8d^2$ のパラメータ数および FLOP 予算と一致させる。

11.1.6 演習問題

1. **RMSNorm のスケール不変性とそのヤコビ行列。** $\mathbf{y} = \text{RMSNorm}(\mathbf{x})$ が Equation 11.4 に従い、$\epsilon = 0$ かつ $\gamma = \mathbf{1}$ で作用するものとする。
    1. $\text{RMSNorm}(c\mathbf{x}) = \text{RMSNorm}(\mathbf{x})$ が任意のスカラー $c > 0$ に対して成り立つことを示せ。
    2. $\mathbf{y} = \sqrt{d}\,\mathbf{x}/\|\mathbf{x}\|_2$ を微分してヤコビ行列が $J(\mathbf{x}) = \frac{\sqrt{d}}{\|\mathbf{x}\|_2}\left(\mathbf{I} - \frac{\mathbf{x}\mathbf{x}^\top}{\|\mathbf{x}\|_2^2}\right)$ となることを示し、$J(\mathbf{x})\mathbf{x} = \mathbf{0}$ を確かめよ。$\mathbf{x}$ に平行な任意の動径方向の摂動が $J(\mathbf{x})$ の零空間に属さなければならないのはなぜか。
2. **展開された残差ストリームと勾配の恒等経路。** $2L$ 個のサブ層（L ブロックの Pre-Norm スタックを構成するサブ層）に $f_1, f_2, \ldots, f_{2L}$ と番号を振る。ここで、奇数番号はアテンションサブ層、偶数番号は FFN サブ層とする。
    1. Equation 11.1 を展開し、$\mathbf{X}^{(L)}$ を $\mathbf{X}^{(0)} + \sum_{m=1}^{2L} f_m(\text{Norm}(\mathbf{X}^{(m-1)/2}))$ として陽に表せ。

2. 連鎖律を適用して $\partial \mathbf{X}^{(L)}/\partial \mathbf{X}^{(0)}$ を $2L$ 個のヤコビ行列の積として書き下し、主要項を展開したうえで、得られる恒等項 $\mathbf{I}$ を、すべての因子に左から J_{Norm} が掛かる Post-Norm の積と対比せよ。

3. **`TransformerBlock` 内部の QK-Norm**。`attn_factory` を作成し、`d2l.MultiHeadAttention` をラップして、アテンションスコアの計算前にヘッドごとのクエリとキーへ幅 `d // h` の RMSNorm を適用するようにせよ。射影重みを $4\times$（または $16\times$）に定数倍した 16 ブロックのスタック上で、アテンションのエントロピーがゼロから離れた値に保たれることを確かめよ。

4. **深さ方向のストリーム分散予算**。Pre-Norm 配置では、すべてのサブ層が残差ストリームに更新量を加算する。`signal_stats` を修正し、`W_o` と `W_2` を標準偏差 $0.02/\sqrt{2L}$ で、他のすべての行列を標準偏差 0.02 でサンプリングするようにして、すべての行列を標準偏差 0.02 とした場合と $L \in \{8, 16, 32, 64\}$ にわたって比較せよ。スケーリングなしでは最終的なストリームの RMS が L とともに増大する一方、$1/\sqrt{2L}$ 則のもとではほぼ一定に保たれることを確かめよ。

5. **トークン崩壊のレート**。@eq-collapse-rate の予測には FFN に対応する因子が含まれていないが、これはトークン間を混合するのはアテンションサブ層だけであり、FFN は位置ごとに作用するという議論に基づいている。

    1. `MiniBlock` を修正してアテンションサブ層をスキップし（$\mathbf{H} = \text{Norm}(\mathbf{X})$）、Post-Norm の FFN のみが実行されるようにしたうえで、`signal_stats(pre_norm=False)` を再実行せよ（アテンションの重みは使われなくなるため、勾配に関する行は削除せよ）。32 ブロック後にトークンの広がりはどれだけ残っているか。
    2. 次に FFN をスキップし、アテンションサブ層を残せ。実測される広がりは Equation 11.3 に $k = 8, 16, 32$ でどれほど近づくか。
    3. FFN のみのスタックでもわずかな収縮が起きる。位置ごとのネットワークが全トークンに共通する小さなベクトルを加算できてしまうのは、GELU のどの性質によるものか。

11.2 デコーダのみの言語モデル（GPT）

単一の `TransformerBlock`（11.1 節）は、d 次元のトークン表現の系列を同じ形状 (B, T, d) の系列へと写像する。因果的な Transformer ブロックのスタックを自己回帰（autoregressive）型の **Generative Pre-trained Transformer (GPT)** (Radford et al. 2018, 2019; Brown et al. 2020) に仕立てるには、L 個のブロックを 2 つの境界層で挟み込む（Figure 11.3）。

1. **入力埋め込み：**トークン埋め込みテーブル $\mathbf{E} \in \mathbb{R}^{V \times d}$ は、離散トークン ID $x_1, \dots, x_T \in \{0, \dots, V-1\}$ を $\mathbb{R}^d$ のベクトルへと写像する。なお、アテンション内部で回転位置埋め込み（rotary position embedding）を使用しない場合は、オプションで学習可能な絶対位置埋め込み（absolute position embedding）$\mathbf{P} \in \mathbb{R}^{T_{\max} \times d}$ を加算する。
2. **最終正規化と言語モデリングヘッド：**L 個の因果的な `TransformerBlock` 層を経た後、最終正規化層が残差ストリーム $\mathbf{H}^{(L)}$ を安定化し、線形ヘッドが各位置 t を次トークンのロジット $\mathbf{Z} = \text{Norm}(\mathbf{H}^{(L)})\mathbf{E}^\top \in \mathbb{R}^{B \times T \times V}$ へと射影する。このとき、出力射影の重みは入力埋め込みテーブル $\mathbf{E}$ と**重み共有**（weight tying）する (Press and Wolf 2017; Inan et al. 2017)。

11.2.1 ブロックから言語モデルへ

各ブロックの内部では、`CausalAttention` がクエリ・キー・値の射影を融合し、**グループ化クエリアテンション**（grouped-query attention、`num_kv_heads` $\leq$ `num_heads`、10.3 節）をサポートする。

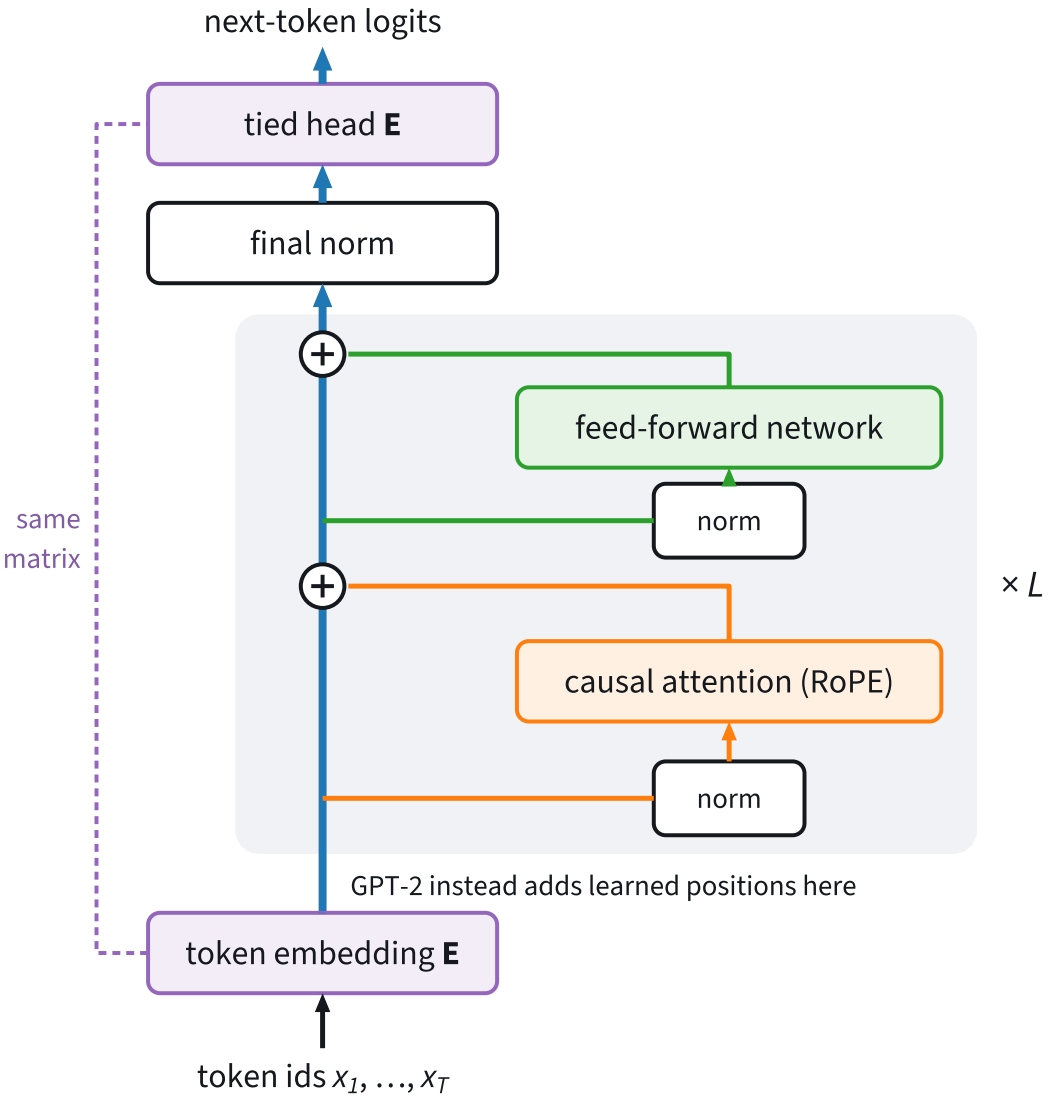


Figure 11.3: 下から上へと見た GPT の構造。テーブル $\mathbf{E}$ がトークン ID を残差ストリームに埋め込み、L 個の因果的な `TransformerBlock` 層がストリームから読み出して書き込み、最終正規化と転置テーブル $\mathbf{E}^\top$ が各位置のベクトルを語彙上のロジットへと変換する。ここに示す現代的な構成では各アテンション層の内部で RoPE によりクエリとキーを回転させるが、GPT-2 では代わりに最下部で学習された位置ベクトルを一度だけ加算する。

さらに、オプションで **RoPE**（rope=True、10.4 節）によってクエリとキーを回転させ、位置 t が位置 $s \leq t$ のみにアテンションを向けるよう因果マスクを適用する。

```
class CausalAttention(nn.Module):
    """Fused causal self-attention, optionally rotary; num_kv_heads <
    num_heads shares each key-value head across a group of queries."""
    def __init__(self, num_hiddens, num_heads, *, bias=False, rope=False,
                 num_kv_heads=None):
        super().__init__()
        self.num_heads, self.rope = num_heads, rope
        self.num_kv_heads = num_kv_heads or num_heads
        self.kv_hiddens = self.num_kv_heads * (num_hiddens // num_heads)
        self.W_qkv = nn.Linear(num_hiddens, num_hiddens + 2 * self.kv_hiddens,
                               bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def qkv(self, X, offset=0, tables=None):
        """Queries, keys, and values of X, each (batch, heads, length, head
        width), at positions offset, offset + 1, ...; tables, if given, are
        the d2l.rope_tables of these positions."""
        B, T, D = X.shape
        q, k, v = self.W_qkv(X).split([D, self.kv_hiddens, self.kv_hiddens],
                                      -1)
        q = q.reshape(B, T, self.num_heads, -1).transpose(1, 2)
        k, v = (u.reshape(B, T, self.num_kv_heads, -1).transpose(1, 2)
                for u in (k, v))
        if self.rope:
            q, k = (d2l.rope(u, offset, tables=tables) for u in (q, k))
        return q, k, v

    def forward(self, X, keys=None, values=None, mask=None, tables=None):
        # Self-attention: X also serves as the keys and the values, and the
        # mask is causal, so a block's mask argument must stay None
        if mask is not None:
            raise ValueError('CausalAttention applies its own mask')
        q, k, v = self.qkv(X, tables=tables)
        Y = F.scaled_dot_product_attention(
            q, k, v, is_causal=True,
            enable_gqa=self.num_kv_heads < self.num_heads)
        return self.W_o(Y.transpose(1, 2).flatten(2))  # (batch, length, d)
```

ここで定義する GPT クラスは、設定可能な num_blks 個の TransformerBlock モジュール（デフォルトでは norm='rms'、act='swiglu'、pos='rope'、pre_norm=True であり、GPT-2 の norm='layer'、act='gelu'、pos='learned'、bias=True もサポートする）を積み重ね、出力射影をトークン埋め込みテーブル token_emb と重み共有する。

```
class GPT(nn.Module):
    """Decoder-only transformer language model built from configurable
    blocks."""
    CausalAttention = CausalAttention  # A subclass may replace it

    def __init__(self, vocab_size, *, num_hiddens=256, num_heads=8,
```

```
                 num_blks=6, max_len=1024, pos='rope', norm='rms',
                 act='swiglu', pre_norm=True, bias=False, dropout=0,
                 num_kv_heads=None, ffn_factory=None):
        super().__init__()
        assert pos in ('rope', 'learned'), f'unknown pos: {pos!r}'
        self.pos, self.max_len, self.num_hiddens = pos, max_len, num_hiddens
        self.head_width = num_hiddens // num_heads
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        nn.init.normal_(self.token_emb.weight, std=0.02)
        if pos == 'learned':
            self.pos_emb = nn.Embedding(max_len, num_hiddens)
            nn.init.normal_(self.pos_emb.weight, std=0.02)
        attn = lambda: self.CausalAttention(
            num_hiddens, num_heads, bias=bias, rope=pos == 'rope',
            num_kv_heads=num_kv_heads)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads, dropout=dropout,
                                 norm=norm, act=act, pre_norm=pre_norm,
                                 bias=bias, attn_factory=attn,
                                 ffn_factory=ffn_factory)
            for _ in range(num_blks)])
        self.norm = (nn.RMSNorm if norm == 'rms'
                     else nn.LayerNorm)(num_hiddens, eps=1e-5)
        self.init_weights()

    def forward(self, X):                       # X: (batch, T) token ids
        H, tables = self.token_emb(X), None     # (batch, T, d)
        if self.pos == 'learned':
            H = H + self.pos_emb(torch.arange(X.shape[1], device=X.device))
        else:  # RoPE angles of positions 0, ..., T - 1, for every block
            tables = d2l.rope_tables(X.shape[1], self.head_width,
                                     device=X.device)
        for blk in self.blks:                   # L blocks, causal attention
            H = blk(H, tables=tables)
        return F.linear(self.norm(H), self.token_emb.weight)  # (batch, T, V)
```

GPT-2 (Radford et al. 2019) および 11.1.2.2 節に従い、`init_weights` は線形層と埋め込み層の重みを $\mathcal{N}(0, 0.02^2)$ からサンプリングしつつ、各ブロックの 2 つの残差出力射影（`attention.W_o` と `ffn.W_2`）を $1/\sqrt{2L}$ 倍にスケーリングする。

```
@d2l.add_to_class(GPT)
def init_weights(self):
    """GPT-2's initialization of the linear layers: N(0, 0.02^2) weights and
    zero biases; W_o and W_2, which write to the stream, get std
    0.02 / sqrt(2L)."""
    L = len(self.blks)
    for name, m in self.named_modules():
        if isinstance(m, nn.Linear):
            std = 0.02 / (2 * L) ** 0.5 if name.endswith(
                ('W_o', 'W_2')) else 0.02
            nn.init.normal_(m.weight, std=std)
            if m.bias is not None:
```

```
                nn.init.zeros_(m.bias)
```

11.2.2 WikiText-103 におけるバイトレベル GPT の訓練

ゼロからの実験を自己完結させ、トークナイザに起因するアーティファクトを排除するため、**WikiText-103** (Merity et al. 2016) の生の UTF-8 バイト列（$V = 256$）上で直接訓練する。ここでは、ウィンドウ数 $B = 64$、長さ $T = 128$ バイトのミニバッチ（ステップあたり 8,192 トークン）を用いる。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
print(f'{len(data.train) / 1e6:.0f}M training bytes, '
      f'{len(data.val) / 1e6:.2f}M validation bytes, '
      f'vocabulary of {len(data.vocab)}')
```

```
270M training bytes, 1.14M validation bytes, vocabulary of 256
```

また、補助コンストラクタ `gpt(...)` も定義しておく。これは新しく初期化された、すぐに訓練可能な `GPT` を構築する関数である。

`gpt` はモデルを GPU に転送して `torch.compile` でコンパイルし、fp32 の行列積で tf32 テンソルコアを使用できるようにする。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32

def gpt(*args, **kwargs):  # A fresh model, on the GPU, compiled
    return torch.compile(GPT(*args, **kwargs).to(d2l.try_gpu()))
```

11.2.2.1 自己回帰の目的関数

入力ウィンドウ $\mathbf{x} = (x_1, \dots, x_T)$ とシフトされたターゲットウィンドウ $\mathbf{y} = (x_2, \dots, x_{T+1})$ が与えられたとき、因果マスクにより、出力ロジットの第 t 行 $\mathbf{z}_t \in \mathbb{R}^V$ は $(x_1, \dots, x_t)$ のみに依存することが保証される。訓練では、全 T 個の位置にわたる平均交差エントロピー損失を並列に最小化する。

$$\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log \operatorname{softmax}(\mathbf{z}_t)_{x_{t+1}}. \tag{11.7}$$

初期化の段階（学習が一切進む前）では、モデルはすべてのバイトにほぼ一様な確率 $1/V$ を割り当てるため、初期の交差エントロピー損失は $\ln V = \ln 256 \approx 5.55$ nat/バイトに近い値となる。まず、シフトされた入力・ターゲットのウィンドウ対 (`X[0]`, `Y[0]`) を 1 つと、5×5 の下三角因果マスクを確認してみよう。

```
start = data.train.tobytes().find(b'the European lobster')
X, Y = data.windows(data.train, [start])  # One window and its targets
print('X[0]:', repr(data.vocab.decode(X[0, :24].tolist())))
print('Y[0]:', repr(data.vocab.decode(Y[0, :24].tolist())))
print(np.asarray(d2l.causal_mask(5)).astype(int))
```

```
X[0]: 'the European lobster ) a'
Y[0]: 'he European lobster ) an'
[[1 0 0 0 0]
 [1 1 0 0 0]
 [1 1 1 0 0]
 [1 1 1 1 0]
 [1 1 1 1 1]]
```

新しく初期化した 6 ブロックの GPT（≈ 4.79M パラメータ）を検証ミニバッチ上で評価すると、未訓練時の交差エントロピーが $\ln 256 \approx 5.55$ nat/バイトに近いことが確認できる。

```
model = gpt(len(data.vocab), max_len=128)
loss = math.log(d2l.lm_perplexity(model, [next(data.val_dataloader())]))
print(f'{d2l.num_params(model) / 1e6:.2f}M parameters; loss before '
      f'training {loss:.2f} nats per byte, log 256 = {math.log(256):.2f}')
```

```
4.79M parameters; loss before training 5.66 nats per byte, log 256 = 5.55
```

11.2.2.2 訓練レシピ（LMOptimizer）

現代の LLM の事前学習では、**AdamW**（9.5 節、標準的なモメンタムパラメータとしてデフォルトでは $\beta_1 = 0.9, \beta_2 = 0.999$、大バッチでの実行では $\beta_2 = 0.95$ を使用）に、**選択的重み減衰**（埋め込み、バイアス、1D 正規化スケールを対象外とし、2D 線形射影行列にのみ重み減衰 $\lambda = 0.1$ を適用）、**線形ウォームアップとそれに続くコサイン学習率減衰**（9.7 節）、および**大域的勾配ノルムクリッピング**（$\|\nabla \mathcal{L}\|_2 \leq 1.0$）を組み合わせる。

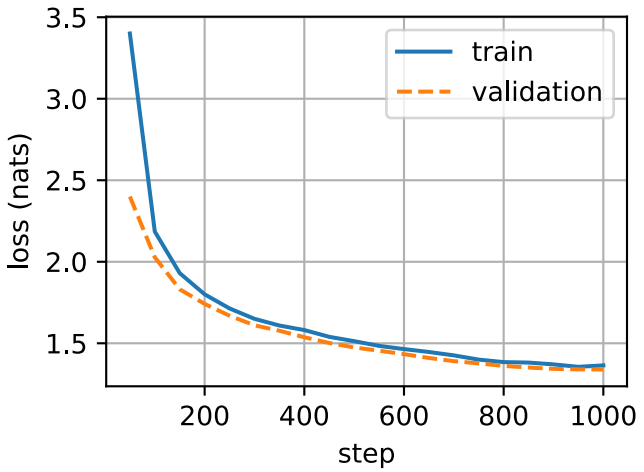
```
class LMOptimizer:
    """AdamW with weight decay on the weight matrices only, a linear warmup
    and a cosine decay of the learning rate, and gradient-norm clipping."""
    def __init__(self, model, lr, num_steps, warmup, weight_decay=0.1,
                 max_norm=1.0):
        named = list(model.named_parameters())
        # Embedding tables are named *emb* (token_emb, pos_emb) by convention
        decay = [p for n, p in named if p.ndim == 2 and 'emb' not in n]
        rest = [p for n, p in named if p.ndim != 2 or 'emb' in n]
        self.optimizer = torch.optim.AdamW(
            [{'params': decay, 'weight_decay': weight_decay},
             {'params': rest, 'weight_decay': 0.0}], lr=lr)
        self.scheduler = torch.optim.lr_scheduler.LambdaLR(
            self.optimizer, lambda t: d2l.cosine_warmup(t, num_steps, warmup))
        self.params, self.max_norm = decay + rest, max_norm

    def zero_grad(self):
        self.optimizer.zero_grad()

    def step(self):
        nn.utils.clip_grad_norm_(self.params, self.max_norm)
        self.optimizer.step()
        self.scheduler.step()
```

6 ブロックのバイトレベル GPT（$d = 256, H = 8, L = 6$、≈ 4.8M パラメータ）を 1,000 ステップ（≈ 8.2M バイト）訓練すると、検証交差エントロピーは英語 Wikipedia テキストの経験ユ

ニグラムエントロピー（3.17 nat/バイト）と経験バイグラムエントロピー（2.45 nat/バイト）の両方を大きく下回る。

```
model = gpt(len(data.vocab), max_len=128)
num_steps, every = 1000, 50
optimizer = LMOptimizer(model, lr=2e-3, num_steps=num_steps, warmup=100)
steps, train_curve, val_curve, timer = [], [], [], d2l.Timer()
for step in range(every, num_steps + 1, every):
    timer.start()
    losses = d2l.train_steps(model, data, optimizer, every)
    timer.stop()  # Training time only, without the validation passes
    steps.append(step)
    train_curve.append(sum(losses) / every)
    val_curve.append(math.log(d2l.lm_perplexity(model, data.val_dataloader())))
d2l.plot(steps, [train_curve, val_curve], 'step', 'loss (nats)',
         legend=['train', 'validation'], fmts=('-', '--'))
print(f'step {num_steps}: train {train_curve[-1]:.2f}, '
      f'validation {val_curve[-1]:.2f} nats per byte; '
      f'{timer.sum():.0f} s of training')
```

```
step 1000: train 1.36, validation 1.34 nats per byte; 14 s of training
```



参考として、訓練コーパスの先頭 10M バイトから経験ユニグラムおよびバイグラムのバイトエントロピーを直接計算し、6 ブロックの Transformer が局所的な文字統計からどれほど先へ進んでいるかを確かめる。

```
text = data.train[:10**7].astype(np.int64)  # The first 10M training bytes
V = len(data.vocab)
unigram = np.bincount(text, minlength=V) / len(text)
pairs = np.bincount(text[:-1] * V + text[1:], minlength=V * V).reshape(V, V)
bigram = pairs / np.maximum(pairs.sum(1, keepdims=True), 1)
p = unigram[unigram > 0]
print(f'unigram {-(p * np.log(p)).sum():.2f}, '
      f'bigram {-np.log(bigram[text[:-1], text[1:]]).mean():.2f} '
```

```
        f'nats per byte')
```

```
unigram 3.17, bigram 2.45 nats per byte
```

11.2.3 統制された比較

11.2.3.1 実行間のばらつきと FFN 活性化関数（SwiGLU と GELU）

純粋なアーキテクチャ上の改善を乱数シードのノイズと切り分けるため、デフォルトの SwiGLU モデルを独立な 3 つのシードで訓練し、さらにパラメータ数を揃えた GELU モデル（act='gelu'）も 3 つのシードで訓練する。

```
def train_lm(model, data, lr=2e-3, num_steps=1000, warmup=100):
    """Train with the recipe of the main run; return milliseconds per step."""
    optimizer = LMOptimizer(model, lr=lr, num_steps=num_steps, warmup=warmup)
    # One untimed call of 50 steps warms up (JAX compiles), the rest is timed
    bench = d2l.Benchmark(lambda: d2l.train_steps(model, data, optimizer, 50),
                          warmup=1, repeats=num_steps // 50 - 1)
    return 1000 * bench.time / 50

runs = {'swiglu': [val_curve[-1]], 'gelu': []}  # The main run comes first
times = {'swiglu': [], 'gelu': []}  # Milliseconds per step
for act, repeats in (('swiglu', 2), ('gelu', 3)):
    for _ in range(repeats):
        net = gpt(len(data.vocab), act=act, max_len=128)
        times[act].append(train_lm(net, data))
        runs[act].append(math.log(d2l.lm_perplexity(net,
                                                    data.val_dataloader())))
    print(f'{act:>6} ({d2l.num_params(net):,} parameters): validation '
          + ', '.join(f'{v:.3f}' for v in runs[act])
          + f'; mean {np.mean(runs[act]):.3f}, range {np.ptp(runs[act]):.3f}')
```

```
swiglu (4,788,992 parameters): validation 1.339, 1.344, 1.343; mean 1.342, range 0.005
  gelu (4,787,456 parameters): validation 1.402, 1.402, 1.402; mean 1.402, range 0.000
```

乱数シード間の実行ごとのばらつきはわずか ~ 0.01 nat/バイトにすぎないのに対し、SwiGLU は同じパラメータ数および FLOP 予算のもとで GELU を一貫して ~ 0.06 nat/バイト上回る。

この train_lm を使えば、11.1 節におけるもう 1 つの設計上の選択である正規化の配置も検証できる。初期化の時点では、11.1.2.2 節で見たように、init_weights が用いる重みスケール 0.02 において Post-Norm に問題は生じなかった。トークン同士は離れたままであり、クエリの勾配も消失せずに残る。しかし、Post-Norm が同様にうまく**訓練できる**かどうかは別の問題である。そこで、同じモデルを Post-Norm ブロック（pre_norm=False）にして、ウォームアップを 500 ステップにした場合、本実験と同じ 100 ステップにした場合、およびウォームアップなしにした場合の 3 通りで訓練する。ピーク学習率はいずれの 3 通りでも本実験と同じである。

```
for warmup in (500, 100, 0):  # Post-norm: long, pre-norm's, and no warmup
    net = gpt(len(data.vocab), pre_norm=False, max_len=128)
    train_lm(net, data, warmup=warmup)
    loss = math.log(d2l.lm_perplexity(net, data.val_dataloader()))
    print(f'post-norm, warmup {warmup:>3}: validation {loss:.3f}')
```

```
post-norm, warmup 500: validation 1.381
post-norm, warmup 100: validation 3.174
post-norm, warmup   0: validation 3.174
```

その答えはウォームアップに依存する。Pre-Norm モデルを約 1.34 nat/バイトまで訓練できる 100 ステップのウォームアップでは、Post-Norm モデルは 3.17 で止まり、ウォームアップなしの実行でも同じ結果になる。この数値はすでに上で見たものである。すなわち先ほど計算したユニグラムエントロピーであり、これら 2 つのモデルは各バイトがどれくらいの頻度で出現するかだけを学習し、直前にあるバイト列については何も学習していない。原因は初期の更新が大きすぎることにある。学習率がわずか数ステップのうちにピークへ達すると、アテンションと FFN の各サブ層の出力が、加算先の正規化された残差ストリームよりもはるかに大きく成長してしまう。その結果、正規化を経た後にはすべてのトークンがほぼ同一の最終ベクトルに行き着き、どの位置でも同じベクトルしか得られないため、バイトの出現頻度しか予測できなくなる。トークン同士を再び引き離すはずの勾配もトークンの広がりとともに縮小してしまうため（11.1.2.2 節）、訓練が回復することはない。

ウォームアップを 500 ステップにすれば、サブ層の出力はストリームより小さいまま保たれ、Post-Norm モデルも Pre-Norm モデルにわずかに及ばないものの約 1.38 nat/バイトまで訓練できる。したがって、Pre-Norm には十分なウォームアップであっても Post-Norm には不十分なのである。その境界がどこにあるかは一概には言えない。境界は明確ではなく、中間のウォームアップ長では訓練が進む実行もあれば崩壊する実行もある。これは Xiong et al. (2020) が元の Transformer について報告した挙動そのものである。すなわち、Post-Norm は長いウォームアップを必要とするのに対し、Pre-Norm はウォームアップなしでも訓練でき、より速く収束する。

11.2.3.2 条件を揃えたリカレントベースラインとの比較

同じパラメータ数（$d_{\text{rnn}} = 1920$、≈ 4.7M パラメータ）と訓練ステップ数を与えたとき、6 ブロックの GPT はリカレント言語モデル（d2l.RNNLM、8.4.2 節）と比べてどうなるだろうか。RNN の学習率を $\{10^{-3}, 2\times10^{-3}, 4\times10^{-3}\}$ にわたってスイープし、ウィンドウ内の位置 $t \in \{0, \dots, 127\}$ の関数として検証損失を比較する（loss_by_position）。

```
def rnnlm(vocab_size, num_hiddens):  # The tanh RNN LM of the Basics
    return d2l.RNNLM(d2l.RNN(256, num_hiddens), vocab_size).to(d2l.try_gpu())

@torch.inference_mode()
def loss_by_position(model, data):
    """Mean validation loss in nats at each position of the window."""
    device = next(model.parameters()).device
    total, count = 0, 0
    for X, Y in data.val_dataloader():
        logits = model(X.to(device))  # (batch, T, V)
        total += F.cross_entropy(logits.transpose(1, 2), Y.to(device),
                                 reduction='none').sum(0)
        count += len(X)
    return (total / count).tolist()
```

各候補のピーク学習率 $\{10^{-3}, 2\times10^{-3}, 4\times10^{-3}\}$ ごとに RNNLM を 2 つのシードで訓練し、最良のリカレントモデルの実行結果をウィンドウ位置 $t \in \{0, 1, 4, 16, 64, 127\}$ にわたって GPT と比較する。

```
rnn_runs = []  # (validation loss, model, ms per step) of every run
for lr in (1e-3, 2e-3, 4e-3):
    for _ in range(2):
        rnn = rnnlm(len(data.vocab), 1920)
        ms = train_lm(rnn, data, lr=lr)
        rnn_runs.append((np.mean(loss_by_position(rnn, data)), rnn, ms))
    print(f'RNN ({d2l.num_params(rnn):,} parameters), peak rate {lr:g}: '
          'validation ' + ', '.join(f'{r[0]:.3f}' for r in rnn_runs[-2:]))
best = min(rnn_runs, key=lambda run: run[0])
positions = (0, 1, 4, 16, 64, 127)
print('validation loss at positions', ', '.join(map(str, positions)))
for name, net, ms in (('GPT', model, np.mean(times['swiglu'])),
                      ('best RNN', *best[1:])):
    loss = loss_by_position(net, data)
    print(f'{name:>8}: ' + ' '.join(f'{loss[t]:.2f}' for t in positions)
          + f'; {ms:.1f} ms per training step')
```

```
RNN (4,739,072 parameters), peak rate 0.001: validation 1.461, 1.461
RNN (4,739,072 parameters), peak rate 0.002: validation 1.474, 1.459
RNN (4,739,072 parameters), peak rate 0.004: validation 1.548, 1.649
validation loss at positions 0, 1, 4, 16, 64, 127
     GPT: 2.47 2.15 1.49 1.33 1.33 1.30; 14.1 ms per training step
best RNN: 2.49 2.15 1.56 1.45 1.45 1.44; 10.6 ms per training step
```

位置 $t = 0$（文脈として 1 バイトしか利用できない位置）では、GPT と RNNLM はほぼ同じ損失から出発する。しかし、利用可能なプレフィックスの文脈が増えるにつれて（$t \to 127$）、Transformer の検証損失は RNN よりも急激に低下する。これは、自己アテンションが任意の先行バイトを $O(1)$ ホップで直接読み取り、過去の履歴を t 回の逐次的なリカレントステップを通して圧縮する必要がないからである。

11.2.4 モデルからのサンプリング

自己回帰生成（GPT.generate）では、一度に 1 トークンずつ末尾に追加していく。各ステップで現在のプレフィックスをモデルに入力し、最終位置のロジット $\mathbf{z}_t$ を**温度**（temperature）$\tau > 0$ で割り、オプションで上位の **top-**k 個を除くすべてのロジットを $-\infty$ にマスクしたうえで (Fan et al. 2018)、$\mathrm{softmax}(\mathbf{z}_t/\tau)$ から次のトークンをサンプリングする。なお、極限 $\tau \to 0$ に対応する貪欲デコーディング（greedy decoding）は、top_k=1 の場合にあたる。

```
@d2l.add_to_class(GPT)
@torch.inference_mode()
def generate(self, prefix, num_tokens, temperature=1.0, top_k=None):
    """Sample a continuation of the token-id list prefix."""
    ids = torch.tensor(prefix, device=next(self.parameters()).device)
    for _ in range(num_tokens):  # The tokens stay on the device
        logits = self(ids[None, -self.max_len:])[0, -1] / temperature
        if top_k is not None:
            cutoff = torch.topk(logits, top_k).values[-1]
            logits[logits < cutoff] = -torch.inf
        ids = torch.cat((ids, torch.multinomial(F.softmax(logits, -1), 1)))
    return ids.tolist()
```

'The history of the ' に続く 120 バイトの系列を 3 通りの (temperature, top_k) 設定でサンプ

リングすると、低い温度と top-k フィルタリングがいかに多様性と引き換えに局所的な一貫性を高めるかがわかる。

```
prefix = data.vocab.encode('The history of the ')
for temperature, top_k in ((1.0, None), (0.7, 8), (2.0, None)):
    out = model.generate(prefix, 120, temperature=temperature, top_k=top_k)
    d2l.print_wrapped(f'T={temperature}, top_k={top_k}: '
                      + repr(data.vocab.decode(out)))
```

```
T=1.0, top_k=None: 'The history of the 2010 lead in Cecil . The shiwn trees
    about his improvement appearances helped attack actual film songs , were
    aportimely'
T=0.7, top_k=8: 'The history of the story to the ship wife . \n = = =
    Control = = = \n The contrast of the final part of the time , and
    interviewers , and the'
T=2.0, top_k=None: 'The history of the
    lar□□\u038d\u038d□□□□E□;v□□□□ □□□□□否□□□□□\x1c,□ \n
    Trenreodoviti□□É□\x08q□□□□nve VS ÷tu KopaxodSq ;
    □t3nĀohjnetoKcckakizonþon U□6'
```

11.2.5 事前学習済み GPT-2（124M）の読み込み

本節の GPT クラスは `norm='layer'`、`act='gelu'`、`pos='learned'`、および `bias=True` をサポートしているため、OpenAI の 124M パラメータの **GPT-2 Small** アーキテクチャ（$V = 50{,}257, d = 768, H = 12, L = 12, T_{\max} = 1024$）をそのままインスタンス化し、公式の `model.safetensors` の重みとバイトレベル BPE トークナイザ（`gpt2_tokenizer`）を直接読み込める。

```
GPT2_URL = 'https://huggingface.co/openai-community/gpt2/resolve/main/'
d2l.DATA_HUB['gpt2-weights'] = (
    GPT2_URL + 'model.safetensors',
    '89a76996d7c6ee89b86618a265483aab73e61d50')
d2l.DATA_HUB['gpt2-merges'] = (
    GPT2_URL + 'merges.txt', '396d4d8ec90cb02f4d56e049e0e4add868bcd943')
d2l.DATA_HUB['gpt2-encoder'] = (
    GPT2_URL + 'vocab.json', 'f0223209235343bc067d7da838328bced8085ae1')

def gpt2_tokenizer():
    """GPT-2's byte-level BPE tokenizer, assembled by tiktoken."""
    import tiktoken  # lazy: d2l itself does not require tiktoken
    from tiktoken.load import data_gym_to_mergeable_bpe_ranks
    return tiktoken.Encoding(
        'gpt2', explicit_n_vocab=50257,
        pat_str=d2l.BPETokenizer.GPT2_PATTERN,
        mergeable_ranks=data_gym_to_mergeable_bpe_ranks(
            d2l.download('gpt2-merges', '../data/gpt2'),
            d2l.download('gpt2-encoder', '../data/gpt2')),
        special_tokens={'<|endoftext|>': 50256})

enc = gpt2_tokenizer()
ids = enc.encode('Attention is all you need.')
print(ids, [enc.decode([i]) for i in ids])
```

```
[8086, 1463, 318, 477, 345, 761, 13] ['Att', 'ention', ' is', ' all', ' you', ' need', '.']
```

Table 11.1: OpenAI の GPT-2 safetensors チェックポイントのキーと本節の GPT モジュールの属性との対応（Conv1D の重みは形状 (in, out) をもつ）。

チェックポイントのテンソル	GPT の属性	備考
wte.weight	token_emb	出力ヘッドと共有（V x d）
wpe.weight	pos_emb	1,024 個の学習済み位置ベクトル
h.i.ln_1, h.i.ln_2	blks[i].norm1, blks[i].norm2	アテンション前および FFN 前の LayerNorm
h.i.attn.c_attn	blks[i].attention.W_qkv	融合 QKV（Conv1D）
h.i.attn.c_proj	blks[i].attention.W_o	出力射影（Conv1D）
h.i.mlp.c_fc, h.i.mlp.c_proj	blks[i].ffn.W_1, blks[i].ffn.W_2	2 層 GELU MLP（Conv1D）
ln_f	norm	出力ヘッド前の最終 LayerNorm

nn.Linear は (out, in) の形状で格納するため、load_gpt2 はすべての Conv1D の重みを転置する。

```
def load_gpt2():
    """GPT-2 small (124M) as a GPT with the released weights, in eval
    mode."""
    from safetensors.torch import load_file  # lazy, like tiktoken above
    gpt2 = GPT(vocab_size=50257, num_hiddens=768, num_heads=12, num_blks=12,
               max_len=1024, pos='learned', norm='layer', act='gelu',
               pre_norm=True, bias=True)
    weights = load_file(d2l.download('gpt2-weights', '../data/gpt2'))
    with torch.no_grad():
        gpt2.token_emb.weight.copy_(weights['wte.weight'])
        gpt2.pos_emb.weight.copy_(weights['wpe.weight'])
        gpt2.norm.weight.copy_(weights['ln_f.weight'])
        gpt2.norm.bias.copy_(weights['ln_f.bias'])
        for i, blk in enumerate(gpt2.blks):
            modules = {f'h.{i}.ln_1': blk.norm1, f'h.{i}.ln_2': blk.norm2,
                       f'h.{i}.attn.c_attn': blk.attention.W_qkv,
                       f'h.{i}.attn.c_proj': blk.attention.W_o,
                       f'h.{i}.mlp.c_fc': blk.ffn.W_1,
                       f'h.{i}.mlp.c_proj': blk.ffn.W_2}
            for key, module in modules.items():
                W = weights[key + '.weight']  # Conv1D: (in, out)
                module.weight.copy_(W.T if W.ndim == 2 else W)
                module.bias.copy_(weights[key + '.bias'])
    return gpt2.eval()

gpt2 = load_gpt2().to(d2l.try_gpu())
print(f'{d2l.num_params(gpt2) / 1e6:.1f}M parameters')
```

```
124.4M parameters
```

読み込んだチェックポイントを検証するため、『The Time Machine』の冒頭段落における交差エントロピー（≈ 3.9 nat/トークン、すなわちパープレキシティ ≈ 49）を計算し、さらに続きのテキストを生成する。

```
text = ("The Time Machine, by H. G. Wells. The Time Traveller was "
        "expounding a recondite matter to us.")
x = torch.tensor(enc.encode(text), device=d2l.try_gpu())[None]
with torch.inference_mode():
    logits = gpt2(x)
loss = F.cross_entropy(logits[0, :-1], x[0, 1:])
print(f'per-token loss {loss.item():.2f}, '
      f'perplexity {loss.exp().item():.0f}')
```

```
per-token loss 3.89, perplexity 49
```

事前学習済みの重みが検証できたところで、gpt2.generate にプロンプトを与え、貪欲デコーディング（top_k=1）によって決定論的な続きを生成してみよう。

```
out = gpt2.generate(enc.encode('Alan Turing theorized that computers '
                               'would one day become'), 16, top_k=1)
d2l.print_wrapped(enc.decode(out))
```

```
Alan Turing theorized that computers would one day become the most powerful
    machines on the planet.

The computer is a computer that
```

確率的な top-k サンプリング（temperature=0.8, top_k=50）に切り替えて 40 個のサブワードトークンを生成すると、流暢で多様な続きが得られる。

```
out = gpt2.generate(enc.encode('The secret of a good deep learning '
                               'textbook is'), 40, temperature=0.8,
                    top_k=50)
d2l.print_wrapped(enc.decode(out))
```

```
The secret of a good deep learning textbook is that it can be read by
    anyone. The easiest way to learn a deep learning language is to read the
    book, or the software. It's also the easiest way to learn a framework.
```

11.2.6 現代的なレシピ

2019 年の load_gpt2() の構成を本節のデフォルトの GPT と比較すると、オープンウェイトである現代のデコーダのみの言語モデルを特徴づける 4 つの主要な改良点が浮かび上がる（Table 11.2）。

1. **正規化：**バイアスなしの Pre-Norm RMSNorm（11.1.2.4 節）を LayerNorm の代わりに用い、多くの場合ヘッドごとの **QK-Norm**（11.1.2.5 節）やサブ層出力の正規化（OLMo 2、Gemma 3）と組み合わせる。
2. **位置表現：**各アテンション層の内部で**回転位置埋め込み**（RoPE、10.4 節）を、固定長の学習済み位置テーブル wpe の代わりに用いる。
3. **フィードフォワードネットワーク：**バイアスなしのゲート付き **SwiGLU**（または GeGLU）を、2 層の GELU MLP の代わりに用いる。
4. **推論キャッシュとモデル容量のスケーリング：**KV キャッシュを縮小する（11.3 節）ための**グループ化クエリアテンション（GQA）**、**スライディングウィンドウアテンション**、または**低ランク潜在圧縮**（low-rank latent compression、MLA）と、トークンあたりの FLOP

数を増やさずに総パラメータ数を拡大する**混合エキスパート**（mixture of experts、MoE）層（11.7 節）。

Table 11.2: 2023〜2025 年の技術報告書で報告された 7 つのオープンウェイトモデル族のアーキテクチャ。アテンションとキャッシュ：11.3.3 節。正規化：11.1 節。位置表現：10.4 節。混合エキスパート：11.7 節。

モデル	アテンションとキャッシュ	正規化	位置表現	FFN	密または MoE
Mistral 7B（2023）(Jiang et al. 2023)	GQA 32:8、スライディングウィンドウ 4096	RMSNorm、Pre-Norm	RoPE、ベース 10^4	SwiGLU	密
Llama 3（2024）	GQA 32:8（8B）、64:8（70B）、128:8（405B）	RMSNorm、Pre-Norm	RoPE、ベース 5×10^5	SwiGLU	密
Qwen3（2025）(A. Yang et al. 2025)	GQA：64:8 密（32B）、64:4 MoE（235B-A22B）	RMSNorm、Pre-Norm、QK-Norm	RoPE、ベース 10^6、YaRN	SwiGLU	密、および MoE：128 エキスパート中 8 個活性
OLMo 2/3（2024/25）(Team OLMo et al. 2025; Team OLMo 2025)	MHA（7B）、GQA（32B）、OLMo 3：4 層中 3 層でウィンドウ 4096	サブ層出力に RMSNorm、QK-Norm	RoPE	SwiGLU	密
DeepSeek-V3（2024）(DeepSeek-AI, Liu, et al. 2024)	MLA：KV を 512 次元の潜在表現に圧縮	RMSNorm、Pre-Norm	分離型 RoPE（64 次元の共有キー）	SwiGLU	MoE：256 エキスパート + 共有 1、8 個活性
Gemma 3（2025）(Gemma Team 2025)	GQA、局所対大域 5:1、ウィンドウ 1024	RMSNorm（Pre・Post 両方）、QK-Norm	RoPE、ベースは大域 10^6、局所 10^4	GeGLU	密
gpt-oss（2025）(OpenAI 2025)	GQA 64:8、交互の層でウィンドウ 128、学習型シンク	RMSNorm、Pre-Norm	RoPE、YaRN	クランプ付き SwiGLU	MoE：128 または 32 エキスパート、4 個活性

11.2.7 まとめ

デコーダのみの GPT は、L 個の因果的な `TransformerBlock` 層をトークン埋め込みテーブル $\mathbf{E} \in \mathbb{R}^{V \times d}$ と最終正規化および重み共有された出力射影 $\mathbf{E}^\top$ の間に配置し、全 T 個の系列位置における次トークンのロジットを 1 回の並列順伝播で予測する（Equation 11.7）。**AdamW**（2D 行列にのみ選択的に重み減衰を適用）、線形ウォームアップ、コサイン減衰、および勾配クリッピング（`LMOptimizer`）を用いて訓練すると、4.8M パラメータのバイトレベル `GPT` は 1,000 ステップ以内に英語テキストのバイグラムエントロピーを大きく下回る。パラメータ数と計算量を揃えた統制された比較では、`SwiGLU` は `GELU` を約 0.06 nat/バイト上回り（シード間のばらつきは 0.01 nat）、また `GPT` は条件を揃えた `RNNLM` に対して文脈位置 t が深くなるほど着実に差を広げていく。現代のオープンウェイト LLM（Table 11.2）はいずれもこの Pre-Norm の `RMSNorm` + `RoPE` + `SwiGLU`・`GeGLU` というレシピを共有しており、主な違いは KV キャッシュをどのように圧縮するか

（11.3 節）、そして密な FFN を混合エキスパートに置き換えるかどうか（11.7 節）にある。

11.2.8 演習問題

1. **訓練レシピ（`LMOptimizer`）のアブレーション**。`LMOptimizer` は、選択的重み減衰（2D 重み行列のみに $\lambda = 0.1$）、線形ウォームアップ（100 ステップ）、コサイン減衰、および大域的勾配クリッピング（$\|\nabla \mathcal{L}\|_2 \le 1.0$）という 4 つの要素を組み合わせている。1 つずつ要素を無効化しながら、本節の 6 ブロックのバイトレベル `GPT` を 1,000 ステップ訓練せよ。(a) 埋め込みや RMSNorm のゲインを含む**すべて**のパラメータに重み減衰を適用する、(b) ウォームアップなし（`warmup=0`）、(c) 定数学習率（終始 `lr=2e-3`）、(d) 勾配クリッピングなし。検証損失を最も悪化させる単一の変更はどれか。

2. **GPT-2 チェックポイントのテンソル対応表の検証**。GPT-2 の `model.safetensors` ファイルには 148 個の重み・バイアステンソル（および 12 個の旧式の因果マスクバッファ `h.i.attn.bias`）が格納されている。`d2l.num_params(gpt2)` がファイル内の 148 個のパラメータテンソルの総要素数（$124{,}439{,}808$）と一致することを確かめる短い検証コードを書き、さらに使用しているフレームワークの線形層の重み配置に基づいて、読み込み時に `c_attn.weight` を転置する必要があるかどうかを説明せよ。

3. **0 層の重み共有埋め込みベースライン（$L = 0$）**。`GPT`（`num_blks=0`）を考える。入力トークン x_t は $\mathbf{e}_{x_t}$ として埋め込まれ、`RMSNorm` で正規化された後、$\mathbf{E}^\top$ との内積によってロジット $\mathbf{z}_t \propto \mathbf{e}_{x_t}\mathbf{E}^\top$ を生成する。

   1. 0 ブロックのモデルは、どのクラスの統計的言語モデル（n グラムの次数）を表現できるか。
   2. 入力埋め込みと出力埋め込みを重み共有すると（$\mathbf{W}_{\text{LM}} = \mathbf{E}^\top$）、対称行列 $\mathbf{E}\mathbf{E}^\top$ が対角成分 $x_{t+1} = x_t$ に最大の成分を割り当ててしまうのはなぜか（ただし $\mathbf{E}$ のすべての行が等しいノルムをもつとする）。また、重みを共有しないモデルや 1 ブロックのモデルはどのようにしてこの対称性を破るか。

4. **トークナイザ間でのバイトあたりビット数の比較（バイトレベル GPT と BPE GPT-2）**。バイトレベルモデルは交差エントロピーを nat/バイトで報告するのに対し、GPT-2 は交差エントロピーを nat/サブワードトークンで報告する。

   1. 任意の N_B バイトの UTF-8 テキスト（`enc.encode` により N_T 個のサブワードトークンにトークン化されるもの）において、トークンあたりの平均の負の対数尤度が $\mathcal{L}_{\text{tok}}$（nat/トークン）であるサブワードモデルの厳密な**バイトあたりビット数（bpb）**が $\text{bpb} = \frac{N_T\,\mathcal{L}_{\text{tok}}}{N_B \ln 2}$ となることを示せ。
   2. WikiText-103 検証セットの先頭 10,000 バイト上で、本節の 4.8M パラメータのバイトレベル `model` と事前学習済みの 124M パラメータの `gpt2` の両方をバイトあたりビット数で評価せよ。

5. **Top-p（ニュークリアス）サンプリング**。`GPT.generate` を拡張してオプション引数 `top_p`（$0 < p < 1$）を追加し (Holtzman et al. 2020)、温度スケーリング後の確率を降順にソートしたうえで、累積確率が p を超える最小のプレフィックスより後ろのトークンをマスクするようにせよ。`gpt2` にプロンプト `"Alan Turing theorized that computers would"` を与え、`top_p=0.9, temperature=0.8` を用いた場合とフィルタリングなしの `temperature=1.2` を用いた場合とで生成結果を比較せよ。

6. **QK-Norm による学習率の安定性**。本節の 6 ブロックのバイトレベル `GPT` を、ピーク学習率 $\eta \in \{2 \times 10^{-3}, 6 \times 10^{-3}, 1.5 \times 10^{-2}\}$ にわたり、標準の `CausalAttention` を用いた場合と、`RMSNorm(d_h)` を `q` と `k` に適用する変種（`CausalAttention.qkv` の内部、11.1.2.5 節）を

用いた場合とで訓練せよ。QK-Norm は安定に訓練できる最大学習率をどのように変化させるか。

11.3 KV キャッシュとキャッシュ削減

`GPT.generate`（11.2.4 節）がデコードステップ t で何をしているか詳しく見てみよう。トークン $t+1$ をサンプリングするために、プレフィックス $x_{1:t}$（長さ t）の全体に対して完全な順伝播を実行し、最後のロジットベクトル $\mathbf{z}_t \in \mathbb{R}^V$ のみを読み取って x_{t+1} を末尾に追加し、続くステップ $t+1$ では再び $x_{1:t+1}$ を全 L 個のブロックに最初から通し直している。因果アテンション（causal attention）は下三角構造をもつため、どの層においても位置 $s \le t$ の表現はトークン $1, \dots, s$ のみに依存する。したがって、位置 $1, \dots, t-1$ のキーと値をステップ t で再計算しても、ステップ $t-1$ で計算したものとまったく同じテンソルが再現されるだけである。T 個のトークンを生成する間に、この冗長な再計算によって延べ $O(T^2)$ 個のトークンが重み行列を通過することになるが、本来は T 個で済むはずである。

過去のキーと値を**キー・値（KV）キャッシュ**（key–value cache、Figure 11.4）に保存すれば、この冗長性は解消される。ステップ t において、各層は単一の新しいトークン x_t のみを $(\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t)$ へ射影し、$(\mathbf{k}_t, \mathbf{v}_t)$ を事前確保されたバッファのスロット t に書き込み、$\mathbf{q}_t$ からキャッシュ済みのプレフィックス $(\mathbf{K}_{1:t}, \mathbf{V}_{1:t})$ に対してアテンションを計算する。

しかし、冗長な FLOPs を削減すると、今度は**メモリ容量とメモリ帯域幅**（memory bandwidth）というハードウェア上のボトルネックが露呈する。単一トークンのデコードでは、トークンを 1 つ生成するたびに、わずかな回数の行列・ベクトル積を実行するためだけにモデルの重みと KV キャッシュ全体を GPU メモリ（HBM）から読み出さなければならない。コンテキストが長い場合やバッチが大きい場合には、KV キャッシュは数十ギガバイトにまで膨れ上がり、デコードのスループットを頭打ちにする。本節の前半では、キャッシュを用いた順伝播を構築し、ハードウェアのルーフライン上で**プレフィル**、**デコード**、**バッチ化**（batching）を解析する。後半（11.3.3 節）では、KV キャッシュをその 3 つの次元、すなわちヘッド数（**GQA/MQA**）、チャネル幅（**低ランクのキー・値射影**）、系列長（**アテンションシンク付きスライディングウィンドウ**）に沿って圧縮する。

11.3.1 再計算からキャッシュへ

11.3.1.1 キャッシュを用いた順伝播

各ブロックでキャッシュするのがキーと値 $(\mathbf{K}_{1:t}, \mathbf{V}_{1:t})$ であり、クエリ $\mathbf{Q}_{1:t}$ や残差状態 $\mathbf{H}_{1:t}$ ではないのはなぜだろうか。因果マスキング（causal masking）のもとでは、クエリ $\mathbf{q}_s$（$s < t$）はトークン $s+1$ を出力するときにしか使われず、その後二度と参照されない。また、**RoPE 適用後**のキー $\mathbf{k}_s$ と値 $\mathbf{v}_s$（いずれも形状は $(B, H_{\text{kv}}, 1, d_h)$）をキャッシュしておけば、過去のトークンを $\mathbf{W}_k, \mathbf{W}_v$ で再射影したり RoPE で再回転させたりする必要も一切なくなる。

`init_cache` は、ゼロ初期化したキーと値のバッファ `ks`, `vs`（形状 $(B, H_{\text{kv}}, T_{\max}, d_h)$）を L 個の各ブロックに対して事前確保する。Figure 11.5 に生成途中のバッファの 1 つを示す。位置 t より前のスロットはすでに埋まっており、現在のステップでスロット t に書き込み、残りのスロットは 0 のままマスクで隠されている。

`forward_cached(model, X, ks, vs, t)` は、スライス `X` に含まれる n 個の新しいトークンを位置オフセット `t` から処理し（RoPE に `offset=t` を渡すか `pos_emb[t:t+n]` をインデックス参照する）、それらのキーと値を `[:, :, t:t+n, :]` に書き込み、`[:, :, :t+n, :]` に対してアテンションを計算する。

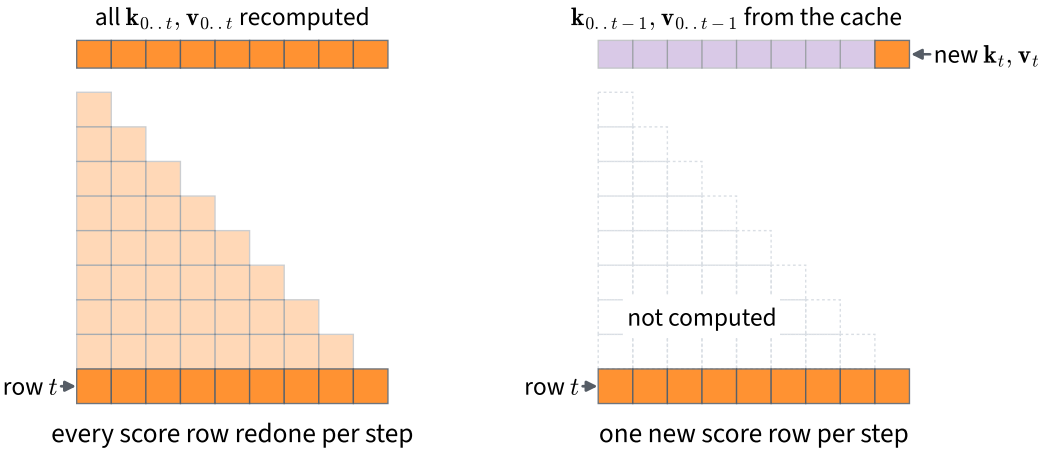


Figure 11.4: 位置 t における 1 デコードステップ。素朴な生成では、プレフィックスのすべてのキー・値・スコア行を再計算したうえで最後の行のみを使用する。KV キャッシュを用いれば、過去のキーと値はメモリから読み出され、そのステップでは新しい単一のキー・値・スコア行のみを計算する。

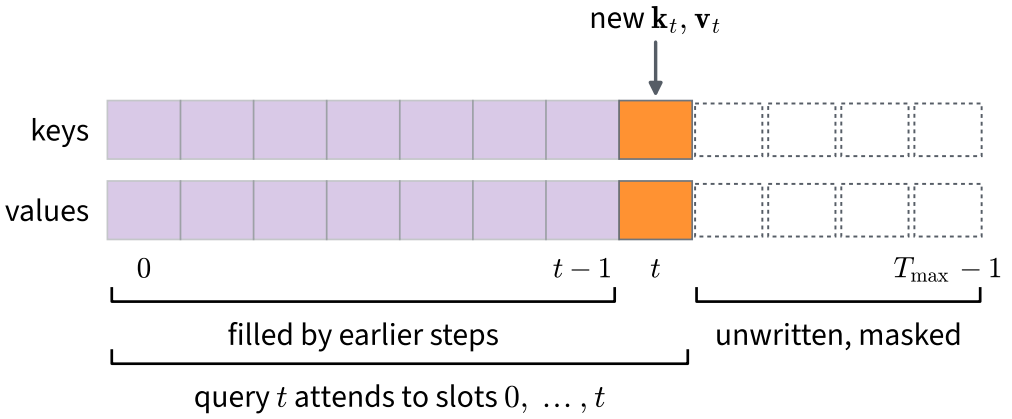


Figure 11.5: 位置 t のステップにおける 1 ブロック分のキャッシュバッファ（1 系列、1 キー・値ヘッド）。スロット $0, \ldots, t-1$ には以前のステップのキーと値が保持されている。このステップでは $\mathbf{k}_t$ と $\mathbf{v}_t$ をスロット t に書き込む。t より後のスロットは 0 のままでマスクによって隠されるため、クエリ t はスロット $0, \ldots, t$ にアテンションを向ける。

```
def init_cache(model, batch_size, max_len):
    """Zeroed keys and values: for every block one buffer of each, of
    shape (batch, kv heads, max_len, head width)."""
    attn = model.blks[0].attention
    shape = (batch_size, attn.num_kv_heads, max_len,
             attn.kv_hiddens // attn.num_kv_heads)
    p = next(model.parameters())
    cache = [torch.zeros(shape, device=p.device, dtype=p.dtype)
             for _ in range(2 * len(model.blks))]
    for u in cache:  # Fixed addresses, which a compiled step writes into
        torch._dynamo.mark_static_address(u)
    return cache[::2], cache[1::2]

def forward_cached(model, X, ks, vs, t):
    """Logits after the last of the new tokens X, which sit at positions
    t, t + 1, ...; writes their keys and values into the cache."""
    assert all(blk.pre_norm for blk in model.blks), 'pre-norm blocks only'
    T = X.shape[1]
    pos = t + torch.arange(T, device=X.device)  # t: an int or a tensor
    H, tables = model.token_emb(X), None
    if model.pos == 'learned':
        H = H + model.pos_emb(pos)
    else:  # RoPE angles of positions t, ..., t + T - 1, for every block
        tables = d2l.rope_tables(T, model.head_width, t, device=X.device)
    # Query t + i sees the filled positions 0, ..., t + i of the buffer
    mask = torch.arange(ks[0].shape[2], device=X.device) <= pos[:, None]
    for i, blk in enumerate(model.blks):  # Pre-norm, as in d2l.GPT
        attn = blk.attention
        q, k, v = attn.qkv(blk.norm1(H), t, tables)  # Rotated at t, ...
        ks[i][:, :, pos], vs[i][:, :, pos] = k, v    # Written in place
        # Attention over the whole buffer; each group of query heads
        # reads one key-value head: (batch, kv heads, group, T, max_len)
        q = q.unflatten(1, (attn.num_kv_heads, -1))
        S = q @ ks[i][:, :, None].transpose(-1, -2) / math.sqrt(q.shape[-1])
        Y = d2l.masked_softmax(S, mask) @ vs[i][:, :, None]
        H = H + attn.W_o(Y.flatten(1, 2).transpose(1, 2).flatten(2))
        H = H + blk.ffn(blk.norm2(H))
    return F.linear(model.norm(H[:, -1]), model.token_emb.weight), ks, vs
```

forward_cached を用いることで、generate_cached は生成を 2 つの段階に分割する。(1) 長さ T_p のプロンプト全体を 1 回の並列呼び出し（t = 0, n = T_p）で処理する**プレフィル**、および (2) 新たにサンプリングされたトークンをステップごとに 1 つずつ処理する（n = 1、オフセット t = T_p, T_p+1, ...）**デコード**である。

```
def sample(logits, temperature=1.0, top_k=None):
    """One token per row of logits (batch, vocab), as generate draws it."""
    logits = logits / temperature
    if top_k is not None:
        cutoff = torch.topk(logits, top_k).values[:, -1:]
        logits = logits.masked_fill(logits < cutoff, -torch.inf)
    return torch.multinomial(F.softmax(logits, -1), 1)
```

```
# One-token steps of fixed shape, compiled once and replayed as CUDA graphs
# (mode='reduce-overhead' sets the first option; the second turns off a
# one-pass softmax that does not apply to a single query over a long cache)
decode_step = torch.compile(forward_cached, dynamic=True, options={
    'triton.cudagraphs': True, 'online_softmax': False})

@torch.inference_mode()
def decode(model, logits, ks, vs, t, num_tokens, temperature=1.0,
           top_k=None):
    """Sample num_tokens tokens per sequence, the first from logits and
    each next one after a step at positions t, t + 1, ...; returns them as
    (batch, num_tokens), and the cache."""
    pos = torch.arange(t, t + num_tokens, device=logits.device)
    tokens = [sample(logits, temperature, top_k)]
    for i in range(num_tokens - 1):  # No step after the last token
        logits, ks, vs = decode_step(model, tokens[-1], ks, vs, pos[i])
        tokens.append(sample(logits, temperature, top_k))
    return torch.cat(tokens, 1), ks, vs

@d2l.add_to_class(d2l.GPT)
def generate_cached(self, prefix, num_tokens, temperature=1.0, top_k=None):
    """Sample as generate does, but never recompute the prefix."""
    assert len(prefix) + num_tokens <= self.max_len, 'cache overflow'
    X = torch.tensor(prefix, device=next(self.parameters()).device)[None]
    ks, vs = init_cache(self, 1, len(prefix) + num_tokens)
    with torch.inference_mode():  # Prefill: the whole prompt in one call
        logits, ks, vs = forward_cached(self, X, ks, vs, 0)
    tokens = decode(self, logits, ks, vs, len(prefix), num_tokens,
                    temperature, top_k)[0]
    return list(prefix) + tokens[0].tolist()
```

11.3.1.2 厳密な等価性の検証

`generate_cached` の実行時間を計測する前に、プロンプトをプレフィルしてからキャッシュを通じて 1 トークンずつステップを進める処理が、系列全体に対して一度に `model(X)` を実行した場合と（浮動小数点の丸め誤差 $\sim 10^{-6}$ の範囲内で）**まったく同じロジット**を生成することを、`pos='rope'` と `pos='learned'` の双方について確認しよう。

```
gpt = d2l.GPT
```

ここで、3 ブロックの `GPT` を回転位置埋め込み（`pos='rope'`）と学習可能な絶対位置埋め込み（`pos='learned'`）の両方でインスタンス化し、先頭の 16 トークン（全長 48 トークンの系列のうち）を 1 回の呼び出しでプレフィルしたのち、残りの 32 位置を 1 トークンずつ進め、キャッシュなしの完全な順伝播 `model(x)` に対する最大絶対誤差を計測する。

```
for pos in ('rope', 'learned'):
    model = gpt(vocab_size=97, num_hiddens=128, num_heads=4, num_blks=3,
                pos=pos).eval()
    x = torch.randint(0, 97, (1, 48))
    ks, vs = init_cache(model, 1, 48)
```

```
    with torch.inference_mode():
        full = model(x)[0, 15:]  # Logits after positions 15, ..., 47
        logits, ks, vs = forward_cached(model, x[:, :16], ks, vs, 0)
        outs = [logits]
        for t in range(16, 48):
            logits, ks, vs = forward_cached(model, x[:, t:t + 1], ks, vs, t)
            outs.append(logits)
    err = (torch.cat(outs) - full).abs().max()
    print(f'{pos}: max |full - cached| = {err:.1e}')
```

```
rope: max |full - cached| = 9.5e-07
learned: max |full - cached| = 6.0e-07
```

11.3.1.3 二次から線形への高速化の計測

12 層の GPT-2 規模のモデル（$d = 768, H = 12, L = 12$）において、n トークンに対するキャッシュなしの順伝播 1 回分の実時間（time_forward、素朴なデコードではこれを**すべての**ステップで繰り返す）と、コンテキスト長（context length）n におけるキャッシュありのデコードステップ 1 回分の実時間（time_decode）を比較すると、長文脈（long-context）の生成になぜキャッシュが不可欠であるかがわかる。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
device = d2l.try_gpu()
model = gpt(vocab_size=50257, num_hiddens=768, num_heads=12, num_blks=12,
            max_len=4096).to(device).eval()  # GPT-2's vocabulary and size
print(f'{d2l.num_params(model) / 1e6:.0f}M parameters')
```

```
124M parameters
```

続いて、d2l.Benchmark を用いて 2 つの計測用ヘルパー関数を定義する。time_forward(model, n) は n トークンに対するコンパイル済みの完全なプレフィックス順伝播を計測し、time_decode(model, batch_size, n) は 32 回の単一トークン・キャッシュ付きデコードステップ（コンテキスト長 n から開始）の実時間レイテンシの平均を求める。

```
def time_forward(model, n):
    """Seconds for a compiled forward pass over n random tokens."""
    device = next(model.parameters()).device
    x = torch.randint(0, model.token_emb.num_embeddings, (1, n),
                      device=device)
    forward = torch.compile(model)
    with torch.inference_mode():
        return d2l.Benchmark(lambda: forward(x)[:, -1]).time

def time_decode(model, batch_size, n):
    """Seconds per decode step for batch_size sequences at context n,
    averaged over 32 steps."""
    device = next(model.parameters()).device
    logits = torch.zeros(batch_size, model.token_emb.num_embeddings,
                         device=device)  # Uniform: random first tokens
    ks, vs = init_cache(model, batch_size, n + 32)
    return d2l.Benchmark(
```

```
        lambda: decode(model, logits, ks, vs, n, 33)).time / 32
```

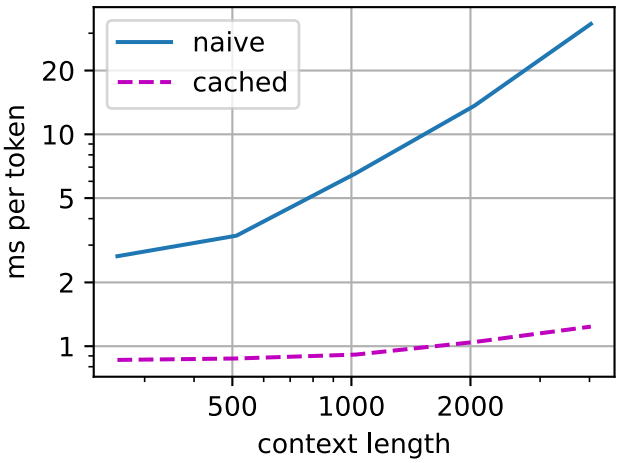
コンテキスト長 n を 256 から 4,032 トークンまで 2 のべき乗に沿って変化させると、キャッシュなしの各ステップが線形に増大するのに対し、キャッシュ付きデコードステップのレイテンシがほぼ一定に保たれるという対比が鮮明になる。

```
lengths, t_naive, t_cached = [256, 512, 1024, 2048, 4032], [], []
for n in lengths:
    t_naive.append(time_forward(model, n))
    t_cached.append(time_decode(model, 1, n))
    print(f'n={n:4d}: naive {1e3 * t_naive[-1]:5.1f} ms, '
          f'cached {1e3 * t_cached[-1]:4.1f} ms per token')
```

```
n= 256: naive   2.7 ms, cached  0.9 ms per token
n= 512: naive   3.3 ms, cached  0.9 ms per token
n=1024: naive   6.5 ms, cached  0.9 ms per token
n=2048: naive  13.6 ms, cached  1.0 ms per token
n=4032: naive  33.2 ms, cached  1.2 ms per token
```

トークンあたりのレイテンシ曲線を両対数軸でプロットすると、素朴なステップのコストがコンテキスト長 n に比例して増大する一方、キャッシュ付きステップはほぼ平坦であることがわかる（新しいクエリとキャッシュ済みの全 n 個のキーとのスコア計算は依然として必要だが、そのコストの定数係数は小さい）。

```
d2l.plot(lengths, [[1e3 * t for t in t_naive], [1e3 * t for t in t_cached]],
         'context length', 'ms per token', legend=['naive', 'cached'],
         xscale='log', yscale='log')
```



新しい 256 トークン（3,584 トークンのプロンプトに続く部分）をエンドツーエンドで生成する全体の累積時間で比べると、`model.generate` を `model.generate_cached` に置き換えることで実時間が 1 桁以上高速化される。

```
prefix = list(range(3584))  # Any token ids will do for timing
for name, gen in (('naive ', model.generate),
                  ('cached', model.generate_cached)):
    t = d2l.Benchmark(lambda: gen(prefix, 256), warmup=1, repeats=1).time
    print(f'{name}: 256 tokens in {t:4.1f} s = {256 / t:5.1f} tokens/s')
```

```
naive : 256 tokens in  8.1 s =  31.5 tokens/s
cached: 256 tokens in  0.5 s = 557.2 tokens/s
```

11.3.2 メモリ容量とハードウェアのルーフライン

ブロック数 L、バッチサイズ B、系列長（sequence length）n、および H_{kv} 個のキー・値ヘッド（各ヘッド幅 d_h）をもち、数値 1 個あたり b バイト（FP32 では $b=4$、BF16/FP16 では 2、FP8 では 1）で保存するモデルでは、各ブロックが 2 つのテンソル（$\mathbf{K}$ と $\mathbf{V}$、形状 $(B, H_{\text{kv}}, n, d_h)$）を保持する。したがって、KV キャッシュ全体のメモリ使用量は次の式で与えられる。

$$M_{\text{KV}}(n, B) \;=\; 2\, b\, B\, L\, n\, H_{\text{kv}}\, d_h \;\text{ bytes.} \tag{11.8}$$

```
def kv_cache_bytes(model, num_tokens, batch_size=1, bytes_per_elem=4):
    """Bytes of the keys and values of num_tokens positions in every block
    of a GPT, for batch_size sequences."""
    attn = model.blks[0].attention
    return (2 * len(model.blks) * attn.kv_hiddens * num_tokens * batch_size
            * bytes_per_elem)
```

kv_cache_bytes を 124M パラメータの FP32 モデルに対して $n = 4{,}096$ トークンで評価すると、確保されたキャッシュバッファのサイズが Equation 11.8（288 MiB）と一致し、すでにモデルの重みのメモリ使用量である 471 MiB の半分を超えていることが確認できる。

```
bytes_per_param = 4  # fp32 weights
weight_bytes = bytes_per_param * d2l.num_params(model)
ks, vs = init_cache(model, 1, 4096)
print(f'cache buffers {sum(u.nbytes for u in ks + vs) / 2**20:.0f} MiB, '
      f'formula {kv_cache_bytes(model, 4096) / 2**20:.0f} MiB, '
      f'weights {weight_bytes / 2**20:.0f} MiB')
```

```
cache buffers 288 MiB, formula 288 MiB, weights 471 MiB
```

11.3.2.1 ルーフライン上のプレフィルとデコード

なぜ 2,048 トークンのプレフィル 1 回あたりの所要時間は、単一トークンのデコードステップ 15 回分程度とほぼ同じ桁に収まったのだろうか。**ルーフラインモデル**（Figure 11.6）は、この違いを**演算密度**（arithmetic intensity、HBM と演算ユニットの間で転送される 1 バイトあたりに実行される FLOPs）によって説明する。

- **プレフィル（$n = 2048$ トークンの並列処理、計算律速）：**各重み行列 $\mathbf{W}$ は HBM から 1 回読み出されて Bn 個のトークンベクトルと乗算される（$2BnN_{\text{params}}$ FLOPs の演算に対し重みの転送量は $b_w N_{\text{params}}$ バイト）。これにより演算密度は $\sim 2Bn/b_w \gg 80$ FLOPs/byte となり、GPU のリッジポイントを大きく上回ってテンソルコアを飽和させる。

- **単一ストリームのデコード（ステップあたり $B = 1, n = 1$ トークン、メモリ帯域幅律速）：** 各ステップでは行列・ベクトル積としてわずか $2N_{\text{params}}$ FLOPs しか実行しない一方で、HBM から**すべての**重み $b_w N_{\text{params}}$ バイトに加えてキャッシュ $M_{\text{KV}}(n, 1)$ バイトを読み出すため、演算密度は $\leq 2/b_w \approx 0.5$ FLOPs/byte（FP32 の場合。BF16 では 1.0 FLOP/byte）となり、計算ルーフより $100\times$ 以上低い水準にとどまる。

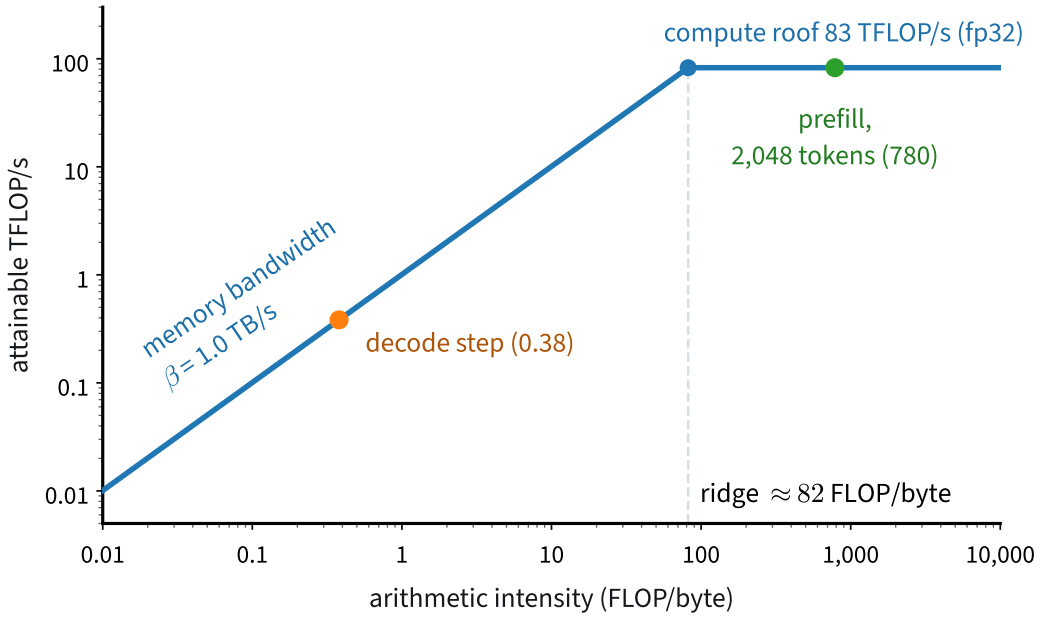


Figure 11.6: RTX 4090 における fp32 のルーフライン。達成可能なスループットは、計算ルーフ（約 83 TFLOP/s）と、演算密度にメモリ帯域幅（約 1 TB/s）を掛けた値のうち小さい方であり、両者は約 82 FLOPs/byte のリッジポイントで交わる。コンテキスト 2,048 トークンにおける GPT-2 規模のモデルのデコードステップは 0.38 FLOPs/byte であり、帯域幅の傾斜部のずっと下方に位置する。一方、2,048 トークンのプレフィルは、重みとキャッシュのみを数えた場合で約 780 FLOPs/byte となり、計算ルーフの下に位置する。

```
t_prefill = time_forward(model, 2048)
t_decode = time_decode(model, 1, 2048)
```

得られたトークンスループットと演算密度を 1 TB/s の HBM 帯域幅の上限値と比較すると、単一ストリームのデコードがメモリ帯域幅の上限付近にとどまる一方、2,048 トークンのプレフィルがそれより $100\times$ 以上高いスループットを達成する理由がわかる。

```
bandwidth = 1e12  # bytes/s, about an RTX 4090
flops = 2 * d2l.num_params(model)  # One decode step at context 2048
moved = weight_bytes + kv_cache_bytes(model, 2048)
print(f'prefill: {2048 / t_prefill:6.0f} tokens/s')
print(f'decode:  {1 / t_decode:6.0f} tokens/s, bandwidth ceiling '
      f'{bandwidth / moved:.0f} tokens/s')
print(f'FLOPs per byte: decode {flops / moved:.2f}, '
      f'prefill about {2048 * flops / moved:.0f}')
```

```
prefill: 150413 tokens/s
decode:     952 tokens/s, bandwidth ceiling 1550 tokens/s
FLOPs per byte: decode 0.38, prefill about 784
```

11.3.2.2 バッチデコード

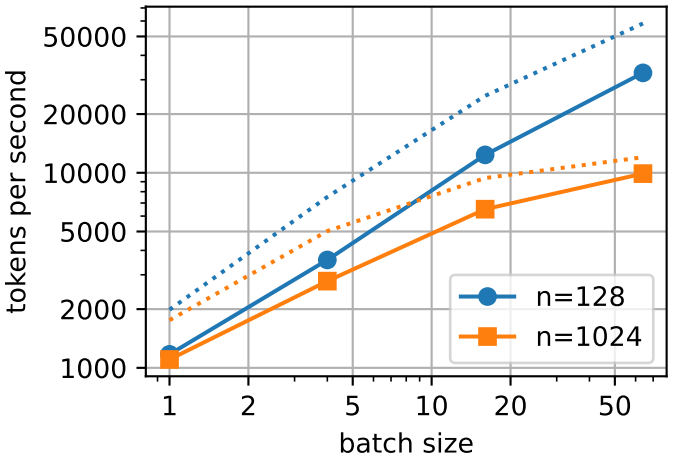
単一ストリームのデコードステップ（$B = 1$）は大半の時間を HBM からモデルの重み $M_{\text{weights}} = b_w N_{\text{params}}$ バイトを読み出すことに費やすため、B 本の系列からなるバッチを同時にデコードすれば、1 回の重みの読み出しを全 B トークンで再利用しつつ、B 個の個別の KV キャッシュ（$B\,M_{\text{KV}}(n,1)$ バイト）を読み出すだけで済む。メモリ帯域幅を $\mathcal{B}_{\text{HBM}}$（bytes/s）とすると、帯域幅で制限されるデコードスループット（tokens/s）は次の式で抑えられる。

$$\text{Throughput}(B,n) \;\le\; \frac{B\,\mathcal{B}_{\text{HBM}}}{M_{\text{weights}} + B\,M_{\text{KV}}(n,1)}. \tag{11.9}$$

バッチサイズが小さい領域（$B\,M_{\text{KV}} \ll M_{\text{weights}}$）では、スループットは B に対してほぼ線形に増加する（「無償のバッチ化」）。一方、B または n が大きくなると（$B\,M_{\text{KV}} \gg M_{\text{weights}}$）、KV キャッシュの読み出しが HBM の通信量を支配するようになり、スループットは **KV 帯域幅の上限** $\mathcal{B}_{\text{HBM}}/M_{\text{KV}}(n,1)$ で飽和する。

```
batches, measured, bounds = [1, 4, 16, 64], [], []
for n in (128, 1024):
    measured.append([b / time_decode(model, b, n) for b in batches])
    nc = kv_cache_bytes(model, n)  # One sequence's cache
    bounds.append([b * bandwidth / (weight_bytes + b * nc) for b in batches])
    print(f'n={n}: ' + ', '.join(f'{r:.0f}' for r in measured[-1])
          + ' tokens/s')
d2l.plot(batches, measured + bounds, 'batch size', 'tokens per second',
         legend=['n=128', 'n=1024'], fmts=('C0-o', 'C1-s', 'C0:', 'C1:'),
         xscale='log', yscale='log', legend_loc='lower right')
```

```
n=128: 1173, 3574, 12353, 32525 tokens/s
n=1024: 1104, 2783, 6507, 9902 tokens/s
```



11.3.3 KV キャッシュの縮小

メモリ容量（Equation 11.8）と大バッチ時のデコードスループットの上限（Equation 11.9）はどちらも系列あたりのキャッシュサイズ $M_{\text{KV}}(n,1) = 2bLnH_{\text{kv}}d_h$ によって決まるため、現代の Transformer アーキテクチャは積 $n \cdot H_{\text{kv}} \cdot d_h$ を互いに直交する 3 つの軸に沿って圧縮する。

1. **キー・値ヘッド数の削減（$H_{\text{kv}} < H$）**：グループ化クエリアテンション（grouped-query attention、GQA）(Ainslie et al. 2023) およびマルチクエリアテンション（MQA）(Shazeer 2019) は、各キー・値ヘッドを $G = H/H_{\text{kv}}$ 個のクエリヘッドで共有する（Figure 11.7）。
2. **トークンあたりの表現幅の縮小（$2H_{\text{kv}}d_h \to r$）**：低ランクのキー・値射影およびマルチヘッド潜在アテンション（Multi-Head Latent Attention、MLA）(DeepSeek-AI, Liu, et al. 2024) は、全ヘッドのキーと値を低次元の潜在ベクトルに圧縮する（11.3.3.2 節）。
3. **キャッシュするコンテキストの短縮（$n \to w + s$）**：スライディングウィンドウアテンションはウィンドウ幅 w より古いトークンを破棄する一方、先頭の $s \geq 1$ 個の**アテンションシンク**トークンを保持することで (Xiao et al. 2024)、ソフトマックスの分母を安定に保つ（11.3.3.3 節）。

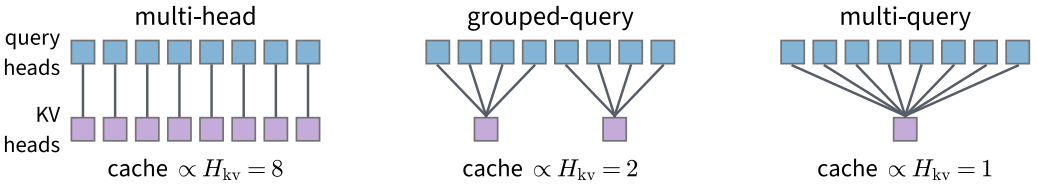


Figure 11.7: クエリヘッドとそれらが読み出すキー・値ヘッド。マルチヘッドアテンションは各クエリヘッドに固有のキー・値ヘッドを与え、グループ化クエリアテンションはクエリヘッドのグループごとに 1 つを共有し、マルチクエリアテンションはすべてのクエリヘッドで単一のヘッドを共有する。キャッシュのサイズはキー・値ヘッドの数に比例する。

11.3.3.1 軸 1：グループ化クエリアテンションとチェックポイントのアップトレーニング

`CausalAttention` クラス（11.2 節）はすでに引数 `num_kv_heads` を受け取るようになっている。$H_{\text{kv}} = H/G$ と設定すると $\mathbf{W}_{qkv}$ が小さくなり、M_{KV} は $G = H/H_{\text{kv}}$ 分の 1 に削減される。さらに Ainslie et al. (2023) は、既存のマルチヘッドのチェックポイント（$H_{\text{kv}} = H$）について、G 個ずつのキーおよび値の射影ヘッドのグループに**平均プーリング**（mean-pooling）を施し（`pool_heads`）、元の訓練計算量のわずかな割合（$\sim$ 5–10%）だけ**アップトレーニング**することで、GQA や MQA モデルへ変換できることを示した。

6 ブロックのバイトレベル `GPT`（$H = 8$）を 1,000 ステップにわたり $H_{\text{kv}} \in \{8, 2\}$ でゼロから訓練した場合（および $H_{\text{kv}} = 2$ で 50 ステップのみ訓練した場合）と、訓練済みの $H_{\text{kv}} = 8$ チェックポイントを平均プーリングまたは各グループの先頭ヘッドのみを残す方法で $H_{\text{kv}} = 2$ に変換した場合とを、ゼロショット（0 ステップ）および 50・100 ステップ（10%）のアップトレーニング後について、2 回の実行で比較する。

```
def pool_heads(target, source, pool):
    """Load the weights of source into target, which has fewer key-value
    heads. Each key (and value) projection of target is the mean of those
    of a group of consecutive heads of source (pool='mean'), or the first
    of them (pool='first')."""
    src, tgt = source.blks[0].attention, target.blks[0].attention
```

```
    group = src.num_kv_heads // tgt.num_kv_heads
    state = source.state_dict()
    for name, W in state.items():
        if name.endswith('W_qkv.weight'):  # Rows: d queries, keys, values
            q, k, v = W.split([W.shape[1], src.kv_hiddens, src.kv_hiddens])
            k, v = (u.unflatten(0, (tgt.num_kv_heads, group, -1))
                    for u in (k, v))
            k, v = ((u.mean(1) if pool == 'mean' else u[:, 0]).flatten(0, 1)
                    for u in (k, v))
            state[name] = torch.cat([q, k, v])
    target.load_state_dict(state)
    return target.to(next(source.parameters()).device)
```

それでは、6 ブロックのバイトレベル GPT を WikiText103 上で $H_{\text{kv}} \in \{8, 2\}$ の設定により訓練し、pool_heads を 'mean' と 'first' の両方のヘッドプーリング手法について 0、50、100 ステップのアップトレーニングにわたり評価しよう。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)

def trained(num_kv_heads, num_steps=1000):
    """A d2l.GPT trained from scratch with the recipe of the GPT section."""
    lm = gpt(len(data.vocab), num_kv_heads=num_kv_heads)
    optimizer = d2l.LMOptimizer(lm, lr=2e-3, num_steps=num_steps,
                                warmup=min(100, num_steps // 10))
    d2l.train_steps(lm, data, optimizer, num_steps)
    return lm

def val_loss(lm):
    return math.log(d2l.lm_perplexity(lm, data.val_dataloader()))

for run in (1, 2):
    mha = trained(8)
    print(f'run {run}, trained from scratch: H_kv=8 {val_loss(mha):.2f}, '
          f'H_kv=2 {val_loss(trained(2)):.2f}, '
          f'H_kv=2 for 50 steps only {val_loss(trained(2, 50)):.2f}')
    for pool in ('mean', 'first'):
        lm = pool_heads(gpt(len(data.vocab), num_kv_heads=2), mha, pool)
        losses = [val_loss(lm)]
        optimizer = d2l.LMOptimizer(lm, lr=3e-4, num_steps=10**9, warmup=0)
        for _ in range(2):  # Uptraining at a constant rate
            d2l.train_steps(lm, data, optimizer, 50)
            losses.append(val_loss(lm))
        print(f'  H_kv=8 converted by {pool:5s}: {losses[0]:.2f}, after 50 '
              f'steps {losses[1]:.2f}, after 100 steps {losses[2]:.2f}')
```

```
run 1, trained from scratch: H_kv=8 1.34, H_kv=2 1.35, H_kv=2 for 50 steps only 2.66
  H_kv=8 converted by mean : 2.76, after 50 steps 1.62, after 100 steps 1.47
  H_kv=8 converted by first: 2.83, after 50 steps 1.89, after 100 steps 1.71
run 2, trained from scratch: H_kv=8 1.34, H_kv=2 1.35, H_kv=2 for 50 steps only 2.66
  H_kv=8 converted by mean : 2.75, after 50 steps 1.62, after 100 steps 1.46
  H_kv=8 converted by first: 2.71, after 50 steps 1.79, after 100 steps 1.64
```

4 つのクエリヘッドでキーと値を共有しても（$H_{\text{kv}} = 2$、KV キャッシュを 4× 縮小）、ゼロから

訓練すれば測定可能なほどの品質低下は生じない（1.34〜1.35 対 1.34 nats/byte）。ヘッドプーリング直後は 2.8 nats/byte 付近から始まるが、100 ステップのアップトレーニングによって平均プーリングを施したモデルは約 1.47 まで改善し、元の性能までの差の約 90% を取り戻す。これに対し、ゼロから 50 ステップ訓練しただけでは約 2.7 にしか届かず、各グループの先頭ヘッドのみを残す方法はアップトレーニング後も平均プーリングに劣る。

11.3.3.2 軸 2：低ランクのキー・値射影

ヘッド数 H_{kv} を減らす代わりに、$2d$ 次元の連結されたキー・値ベクトル $[\mathbf{k}_t; \mathbf{v}_t] \in \mathbb{R}^{2d}$（各トークン、層 i）をランク r の部分空間（subspace、$r \ll 2d$）へ圧縮できないだろうか。これを事前学習済みの **GPT-2 Small**（124M、$d = 768$、$2d = 1536$）に対し、$T = 1024$ トークンからなる『The Time Machine』の一節を用いて直接検証する。

```
gpt2, enc = d2l.load_gpt2().to(device), d2l.gpt2_tokenizer()
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', '../data',
                     '090b5e7e70c295757f55df93cb0a180b9691891a')
tokens = torch.tensor(enc.encode(open(fname).read()), device=device)
x, calib = tokens[None, :1024], tokens[None, 1024:2048]
print(f'{len(tokens)} tokens of The Time Machine')
```

```
44073 tokens of The Time Machine
```

GPT を書き換えずにすべての層の内部でキーと値を横取りして変換できるよう、`forward_blocks(model, X, attn_fn)` を定義してブロック i のアテンション計算をコールバック `attn_fn(q, k, v, i)` に委譲する。そして、標準の因果アテンションが `gpt2(x)` と 10^{-5} 以内で一致することを確認するとともに、ベースラインとなる `tail_loss` を位置 $512 \dots 1023$ 上で求めておく。

```
def forward_blocks(model, X, attn_fn):
    """The forward pass of a d2l.GPT with attention computed by
    attn_fn(q, k, v, i) in block i, on inputs of shape (batch, heads,
    length, head width)."""
    assert all(blk.pre_norm for blk in model.blks), 'pre-norm blocks only'
    H = model.token_emb(X)
    if model.pos == 'learned':
        H = H + model.pos_emb(torch.arange(X.shape[1], device=X.device))
    for i, blk in enumerate(model.blks):
        attn = blk.attention
        Y = attn_fn(*attn.qkv(blk.norm1(H)), i)
        H = H + attn.W_o(Y.transpose(1, 2).flatten(2))
        H = H + blk.ffn(blk.norm2(H))
    return F.linear(model.norm(H), model.token_emb.weight)

def tail_loss(logits, X):
    """Mean loss of the next-token predictions from position 512 on."""
    return F.cross_entropy(logits[0, 512:-1], X[0, 513:])

causal = lambda q, k, v, i: F.scaled_dot_product_attention(
    q, k, v, is_causal=True)
with torch.inference_mode():
    logits = forward_blocks(gpt2, x, causal)
    print(f'helper vs model: max deviation '
          f'{(logits - gpt2(x)).abs().max():.1e}')
```

```
    full = tail_loss(logits, x)
print(f'full attention: loss {full:.2f}, perplexity {full.exp():.0f}')
```

```
helper vs model: max deviation 0.0e+00
full attention: loss 3.58, perplexity 36
```

書籍の続く 1,024 トークン（calib）を用いて各層の連結されたキー・値行に対する経験的な PCA/SVD 基底 $\mathbf{U}_i \in \mathbb{R}^{2d\times r}$ を求め（fit_bases）、評価対象の一節 x のキーと値を上位 $r \in \{512, 256, 128\}$ 方向へ射影する（projected）ことで、再訓練なしでも GPT-2 の KV キャッシュがどこまで圧縮可能かを計測する。

```
def kv_rows(k, v):
    """Keys and values of all heads of each token as one row: (T, 2d)."""
    return torch.cat([u[0].transpose(0, 1).flatten(1) for u in (k, v)], 1)

def fit_bases(model, X):
    """Right singular vectors of the key-value rows of X in every block."""
    bases = []
    def fit(q, k, v, i):
        bases.append(torch.linalg.svd(kv_rows(k, v), full_matrices=False)[2])
        return causal(q, k, v, i)
    forward_blocks(model, X, fit)
    return bases

def projected(bases, r):
    """Attention with keys and values projected onto the top r bases."""
    def attn_fn(q, k, v, i):
        P = bases[i][:r]  # (r, 2d), orthonormal rows
        KV = kv_rows(k, v) @ P.T @ P
        k, v = (u.unflatten(1, (k.shape[1], -1)).transpose(0, 1)[None]
                for u in KV.chunk(2, 1))
        return causal(q, k, v, i)
    return attn_fn

with torch.inference_mode():
    bases = fit_bases(gpt2, calib)
    for r in (512, 256, 128):
        loss = tail_loss(forward_blocks(gpt2, x, projected(bases, r)), x)
        print(f'rank {r} of 768: loss {loss:.2f}, '
              f'perplexity {loss.exp():.0f}')
```

```
rank 512 of 768: loss 3.55, perplexity 35
rank 256 of 768: loss 3.61, perplexity 37
rank 128 of 768: loss 4.14, perplexity 63
```

各ブロックは $2d$ 個のキーと値を 1 つの d 次元入力から 1 つの行列と 1 つのバイアスで計算するため、それらは高々 $d = 768$ 次元のアフィン部分空間に収まる（出力の「of 768」はこの次元数を指す）。事後的な線形射影であっても、GPT-2 の KV ベクトルは測定可能な劣化なしに $6\times$（$r = 256$、元の次元は $2d = 1536$）まで圧縮できる（3.61 対 3.58 nats）が、$r = 128$ まで絞ると 0.5 nat 超の悪化が生じる。一方、DeepSeek-V2/V3 の**マルチヘッド潜在アテンション（MLA）**(DeepSeek-AI, Liu, et al. 2024) のように低ランクのボトルネックをエンドツーエンドで訓練すれば、単一の 512 次元潜在ベクトルで全ヘッドのキーと値を置き換え、$> 16\times$ の圧縮を達成できる。

11.3.3.3 軸 3：スライディングウィンドウとアテンションシンク

3 つ目の軸は、キャッシュに保持する系列長 n を直近 w トークンのローリングウィンドウに制限することである（10.6 節）。しかし、事前学習済み GPT-2 に対して 1,024 トークンの一節上で純粋なスライディングウィンドウ（$w = 256$）を適用すると、後半トークン（$t \geq 512$）の交差エントロピー（cross-entropy）は ~ 3.6 nats から 8 nats 超へと跳ね上がってしまう。

その理由を探るため、`sink_mass` によって位置 $t \geq 64$ のクエリが**トークン 0**（系列の先頭トークン）に割り当てる平均アテンション確率を、GPT-2 の全 12 層のそれぞれについて計測してみよう。

```
def sink_mass(model, X):
    """Mean attention weight on token 0 of the queries from position 64
    on, one entry per block."""
    masses = []
    def watch(q, k, v, i):  # Explicit causal attention
        scores = q @ k.transpose(-1, -2) / math.sqrt(q.shape[-1])
        alpha = d2l.masked_softmax(scores, d2l.causal_mask(
            q.shape[2], device=q.device))
        masses.append(alpha[..., 64:, 0].mean())
        return alpha @ v
    forward_blocks(model, X, watch)
    return masses

with torch.inference_mode():
    masses = sink_mass(gpt2, x)
print('mean attention weight on token 0 (uniform would be about 0.003)')
print('block ' + ''.join(f'{i:6d}' for i in range(len(masses))))
print('weight' + ''.join(f'{m:6.2f}' for m in torch.stack(masses).tolist()))
```

```
mean attention weight on token 0 (uniform would be about 0.003)
block      0     1     2     3     4     5     6     7     8     9    10    11
weight  0.00  0.01  0.11  0.30  0.32  0.51  0.46  0.57  0.48  0.48  0.45  0.34
```

GPT-2 の中間層から上位層にかけて、クエリは**全アテンション重みの 30% から 60% 近くをトークン 0 に割り当てる**が、これは位置 0 にどの単語があるかによらない。ソフトマックス確率の総和は 1 でなければならないため、アテンションヘッドは局所的に一致するトークンがないとき、**すべての**因果クエリから参照可能な唯一の位置である先頭付近のトークンをデフォルトの**アテンションシンク** (Xiao et al. 2024) として利用するのである。純粋なスライディングウィンドウが位置 0 を通り過ぎると（$t \geq w$）、ソフトマックスの分母からトークン 0 が取り除かれることで、その 30–60% もの確率質量が残りのウィンドウ内トークンへ無理やり押し付けられ、残差ストリーム（residual stream）がかき乱される。

先頭のわずか $s \in \{1, 4\}$ 個のシンクトークンをキャッシュの先頭に固定したまま直近 $w = 256$ 個のトークンと併せて保持すれば（キャッシュされるトークン数は合計 $w + s$ 個）、GPT-2 の検証損失はフルコンテキストのベースラインとほぼ同等の水準まで回復する。

```
j = torch.arange(x.shape[1], device=device)
allowed = d2l.causal_mask(x.shape[1], device=device)
w = 256  # The window: each query sees itself and the 255 tokens before it
window = allowed & (j[:, None] - j[None, :] < w)
with torch.inference_mode():
```

```
    for nsink in (0, 1, 4):
        mask = window | (allowed & (j[None, :] < nsink))
        loss = tail_loss(forward_blocks(
            gpt2, x, lambda q, k, v, i: F.scaled_dot_product_attention(
                q, k, v, attn_mask=mask)), x)
        print(f'window {w} + {nsink} sink tokens: loss {loss:.2f}, '
              f'perplexity {loss.exp():.0f}')
print(f'full attention:            loss {full:.2f}, '
      f'perplexity {full.exp():.0f}')
```

```
window 256 + 0 sink tokens: loss 8.25, perplexity 3842
window 256 + 1 sink tokens: loss 3.86, perplexity 47
window 256 + 4 sink tokens: loss 3.76, perplexity 43
full attention:            loss 3.58, perplexity 36
```

11.3.4 キャッシュ削減手法の比較

Table 11.3: $M_{\text{KV}} = 2bBLnH_{\text{kv}}d_h$ の 3 つの因子に沿った KV キャッシュ削減機構のまとめ。

手法	削減対象の因子	系列あたりの保存要素数	訓練の要否
KV ヘッドの共有（GQA、MQA） (Shazeer 2019; Ainslie et al. 2023)	ヘッド数 $H_{\text{kv}} < H$	$2LnH_{\text{kv}}d_h$	要：ゼロからの訓練またはアップトレーニング（$\sim$ 5–10%）
低ランク潜在表現（MLA） (DeepSeek-AI, Liu, et al. 2024)	幅 $2H_{\text{kv}}d_h \to r$	$\approx Lnr$（小さな回転キーを追加）	MLA は要。事後的な SVD 基底でも $\sim 6\times$ まで有効
シンク付きスライディングウィンドウ (Xiao et al. 2024)	長さ $n \to w + s$	$2L(w+s)H_{\text{kv}}d_h$	$s \geq 1$ なら事前学習済みモデルにゼロショットで有効

11.3.5 まとめ

KV キャッシュは過去のトークンの RoPE 適用後のキーと値を保存することで、各デコードステップでの射影を単一の新しいトークン（`n = 1`）のみにとどめ、同一のロジットを生成しながら T トークン生成にわたる射影の計算量を $O(T^2)$ から $O(T)$ 回のトークン評価へと削減する（アテンションスコア計算も $O(T^3)$ から $O(T^2)$ へ減少する）。ハードウェアのルーフライン（Figure 11.6）上では、サービングは計算律速の**プレフィル**（$n = T_p$ 個のプロンプトトークンを高い演算密度で並列処理する段階）と、メモリ帯域幅律速の**デコード**（ステップあたり $n = 1$ トークンを低い演算密度で処理する段階）に分かれる。**バッチデコード**は並行する B 本の系列間で重みの読み出しを償却し（Equation 11.9）、スループットを B に比例して伸ばすが、やがて KV キャッシュの転送量が支配的になると $\mathcal{B}_{\text{HBM}}/M_{\text{KV}}(n,1)$ で飽和する。メモリ容量と大バッチ時のスループットの双方を引き上げるため、現代のアーキテクチャは $M_{\text{KV}} = 2bLnH_{\text{kv}}d_h$（Equation 11.8）を 3 つの直交軸（Table 11.3）、すなわちヘッド数（**GQA/MQA**）、チャネル幅（**低ランク KV / MLA**）、系列長（**アテンションシンク付きスライディングウィンドウ**）に沿って圧縮する。

11.3.6 演習問題

1. **70B モデルのキャッシュメモリとバッチ上限。** Llama 3.1 70B は $L = 80$ 層、隠れ層サイズ $d = 8192$、$H = 64$ 個のクエリヘッド（$d_h = 128$）、$H_{\text{kv}} = 8$ 個のキー・値ヘッドを

もち、BF16（$b = 2$ バイト）で保存される。

1. Equation 11.8 を用いて、単一系列の KV キャッシュサイズ $M_{\text{KV}}(n, 1)$ を最大コンテキスト長 $n = 131{,}072$（128K）において計算し、標準の MHA（$H_{\text{kv}} = 64$）を用いた場合のキャッシュサイズと比較せよ。
2. 合計 $8 \times 80 = 640$ GB の HBM（うち 140 GB は BF16 のモデル重みが占有）を備える 8-GPU ノードにおいて、最大バッチサイズ B（長さ 128K トークンの系列を残りの 500 GB に収められる本数）は、GQA（$H_{\text{kv}} = 8$）と MHA（$H_{\text{kv}} = 64$）のそれぞれでいくつになるか。

2. **バッチデコードのスループットと KV 帯域幅の上限**。1 の 70B モデルを、コンテキスト長 $n = 32{,}768$、総 HBM 帯域幅 $\mathcal{B}_{\text{HBM}} = 24$ TB/s、$M_{\text{weights}} = 140$ GB のシステム上でサービングするとする。

   1. 帯域幅による上界 Equation 11.9 を、バッチサイズ $B \in \{1, 8, 32, 128, 512\}$ において GQA（$H_{\text{kv}} = 8$）と MHA（$H_{\text{kv}} = 64$）の両方について評価せよ。
   2. バッチサイズ $B_{1/2}$ がいくつのとき、各構成は漸近的な上限 $\lim_{B\to\infty} \text{Throughput}(B, n) = \mathcal{B}_{\text{HBM}}/M_{\text{KV}}(n, 1)$ の半分に到達するか。

3. **K 個の出力系列間でのプレフィックス共有**。バッチ内の $K = 16$ 本の系列が $n_{\text{shared}} = 4096$ トークンの共通システムプロンプトを共有し、それぞれが固有のトークンを $n_{\text{unique}} = 256$ 個生成したとする。共有プレフィックスの KV キャッシュを保存し、ステップごとに全 K 系列に対して **1 回**だけ読み出す場合について Equation 11.9 の分母を修正し、70B GQA モデルにおける $K = 16$ でのスループット向上率を計算せよ。

4. **INT8 / FP8 による KV キャッシュの量子化**。`forward_blocks` 内で GPT-2 のキャッシュされるキーと値を、トークンごとの対称 8 ビット整数量子化を用いて保存せよ。すなわち、各トークンベクトル $\mathbf{u} \in \mathbb{R}^d$ に対してスケール $s = \max_j |u_j|/127$ を計算し、$\tilde{\mathbf{u}} = s \cdot \text{round}(\mathbf{u}/s)$ と量子化して、アテンション計算に $\tilde{\mathbf{u}}$ を用いる。`tail_loss`（x 上）の変化を、キーと値を 8 ビットに量子化した場合（FP32 キャッシュより $4\times$ 縮小、BF16 より $2\times$ 縮小）と 4 ビットに量子化した場合（`max / 7`、それぞれ $8\times$ / $4\times$ 縮小）とで計測せよ。

5. **アテンションシンクと仮想シンクロジット（$\textbf{softmax}_1$）**。Xiao et al. (2024) および gpt-oss（Table 11.2）が指摘しているように、標準のソフトマックスはアテンション重みの総和を 1 に強制するため、ヘッドにはアテンションシンクが必要となる。

   1. `forward_blocks` 内のソフトマックスを $\text{softmax}_1(\mathbf{s})_j = \frac{e^{s_j}}{e^b + \sum_m e^{s_m}}$ に置き換えよ（値ベクトルを一切追加せず、非トークンのシンクロジット $b = 5.0$ を分母に加える）。
   2. 純粋なスライディングウィンドウ（$w = 256, s = 0$ 個の保持シンクトークン）を GPT-2 上で $b = 5.0$ として評価せよ。分母に加えた仮想ロジットによって、$s = 0$ における損失の急増はどれだけ解消されるか。

6. **低ランクのキー・値復元の $\mathbf{W}_q$ および $\mathbf{W}_o$ への吸収**。層 ℓ が各トークン j を単一の潜在ベクトル $\mathbf{c}_j \in \mathbb{R}^r$（$r \ll d_h H$）に圧縮し、そこからヘッド m のキーと値を $\mathbf{k}_{j,m} = \mathbf{c}_j \mathbf{B}_{k,m}$ および $\mathbf{v}_{j,m} = \mathbf{c}_j \mathbf{B}_{v,m}$（$\mathbf{B}_{k,m}, \mathbf{B}_{v,m} \in \mathbb{R}^{r \times d_h}$）によって復元するとする。

   1. $\mathbf{q}_{t,m}\mathbf{k}_{j,m}^\top = (\mathbf{x}_t \mathbf{W}_{q,m} \mathbf{B}_{k,m}^\top)\mathbf{c}_j^\top$ および $(\sum_j \alpha_{t,j}^{(m)} \mathbf{v}_{j,m})\mathbf{W}_{o,m} = (\sum_j \alpha_{t,j}^{(m)} \mathbf{c}_j)(\mathbf{B}_{v,m}\mathbf{W}_{o,m})$ が成り立つことを示せ。
   2. デコード時に $\mathbf{B}_{k,m}^\top$ を $\mathbf{W}_{q,m}$ へ、$\mathbf{B}_{v,m}$ を $\mathbf{W}_{o,m}$ へ吸収することで、完全なキー $\mathbf{k}_{j,m}$ や値 $\mathbf{v}_{j,m}$ を HBM 上に一切実体化せずに済む理由を説明せよ。また、DeepSeek-V2/V3 が RoPE を幅の狭い別個の分離されたキー $\mathbf{k}_j^{\text{RoPE}}$ に適用し、$\mathbf{c}_j \mathbf{B}_{k,m}$ を直接回転させない理由を説明せよ。

7. **固定サイズのストリーミング・リングバッファキャッシュの実装。**`StreamingCache` 版の `init_cache` と `forward_cached` を作成し、層あたり $s+w$ 個のトークンスロットのみを事前確保するようにせよ（先頭の $s=4$ 個のシンク位置をスロット $0 \dots s-1$ に保持し、循環バッファ内のスロット $s+((t-s) \bmod w)$ を $t \geq s$ に対して上書きする）。そして、一定の $O(s+w)$ キャッシュメモリで 2,048 トークンを生成できることを確認せよ。なお、`gpt2` は学習済みの位置を 1,024 個しかもたないため、位置インデックスを上限なく増やせる回転位置埋め込み（`pos='rope'`）をもつモデルを用いるか、あるいは `gpt2` の各トークンにそのキャッシュスロットの位置を割り当てよ（Xiao et al. (2024) と同様の手法である）。

11.4 エンコーダ、デコーダ、およびクロスアテンション

これまでに構築してきた Transformer スタックはすべて**因果デコーダ**（causal decoder）（11.2 節）であった。すなわち、三角マスクによってトークン i のアテンション先をプレフィックス位置 $j \leq i$ に制限し、1 回の順伝播を T 個の並列な次トークン予測（next-token prediction）へと変換するものであった。推論時には未来のトークンがまだ存在しないため、トークンを 1 つずつ生成する際には因果マスキング（causal masking）が不可欠である。しかし、検索用の文書埋め込み、トークン分類、翻訳のための原言語文のエンコード、あるいは画像パッチグリッドのエンコード（11.5 節）のように、入力系列の**全体**が最初から得られている場合、位置 i が位置 $j > i$ を読み取ることを禁じると、周囲の文脈の半分を捨てることになる。

計算開始時点でどのトークンが利用可能か、また入力と出力が同じ系列に属するかに応じて、同じ `TransformerBlock`（11.1 節）から 3 つの標準的な配線（wiring）が構成できる（Figure 11.8）。

1. **エンコーダのみ（双方向自己アテンション）：**すべてのトークンがすべてのトークンにアテンションを向け（`is_causal=False`）、完全な入力系列の文脈を考慮した表現を 1 回の順伝播で計算する（BERT (Devlin et al. 2018)、RoBERTa (Liu et al. 2019)、ViT (Dosovitskiy et al. 2021)）。
2. **デコーダのみ（因果自己アテンション）：**各トークンは自身とそれ以前の位置のみにアテンションを向け（`is_causal=True`）、単一の均質なスタックでプロンプトの読み取りと自己回帰生成の両方を担う（GPT、Llama 3、Qwen3）。
3. **エンコーダ・デコーダ（双方向エンコーダ＋クロスアテンション付き因果デコーダ）：**双方向エンコーダがソース系列 $\mathbf{X}_{\text{src}}$ を文脈表現 $\mathbf{H}_{\text{enc}}$ に写像し、因果デコーダがターゲット系列 $\mathbf{Y}_{\text{tgt}}$ を、**クロスアテンション**によって $\mathbf{H}_{\text{enc}}$ にクエリを送りながら 1 トークンずつ生成する（2017 年のオリジナル Transformer (Vaswani et al. 2017)、T5 (Raffel et al. 2020)、Whisper (Radford et al. 2023)）。

11.4.1 エンコーダ、デコーダ、およびエンコーダ・デコーダアーキテクチャ

11.4.2 双方向文脈とマスクトークン予測

11.4.2.1 双方向エンコーダ

`d2l.TransformerBlock` から因果マスクを取り除くと、因果スタックは**双方向 Transformer エンコーダ**（`TransformerEncoder`）になり、トークン ID の入力バッチ（および任意のパディングマスク）を形状 (B, n, d) の文脈表現へと写像する。

```
class TransformerEncoder(nn.Module):
    """Bidirectional encoder: embeddings plus unmasked pre-norm blocks."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4, num_blks=4,
                 max_len=64):
```

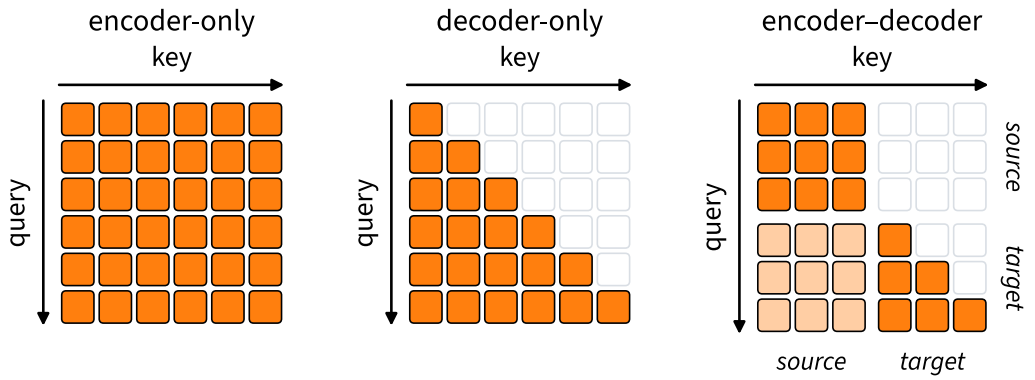


Figure 11.8: 3 つの配線とそのアテンションパターン（クエリ位置は下方向、キー位置は右方向に進み、塗りつぶされたセルはそのクエリがそのキーにアテンションを向けられることを表す）。エンコーダのみ：すべてのトークンがすべてのトークンにアテンションを向ける。デコーダのみ：各トークンは自身とその過去にのみアテンションを向ける。エンコーダ・デコーダ：ソースは自身に双方向のアテンションを向け、各ターゲットトークンはクロスアテンションによってソース全体（淡色のセル）にアテンションを向けつつ自身の過去には因果的にアテンションを向ける。3 番目のパネルには構造上の事実が 1 つ隠れている。すなわち、ソースの行とターゲットの行は 2 つの別々のスタックに属しており、すべてのデコーダブロックは最後のエンコーダブロックの出力を読み取る。

```
        super().__init__()
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        self.pos_emb = nn.Embedding(max_len, num_hiddens)
        for emb in (self.token_emb, self.pos_emb):
            nn.init.normal_(emb.weight, std=0.02)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads)
            for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)

    def forward(self, X, mask=None):
        H = self.token_emb(X) + self.pos_emb(
            torch.arange(X.shape[1], device=X.device))
        for blk in self.blks:
            H = blk(H, mask)  # by default no mask: all tokens see all
        return self.norm(H)
```

11.4.2.2 マスク言語モデリング（MLM）の目的関数

なぜ双方向スタックは 11.2 節の次トークン予測の目的関数では訓練できないのだろうか。位置 t のターゲットはトークン x_{t+1} であり、そのトークンは位置 $t+1$ の入力そのものである。マスクがなければ、わずか 1 ブロックでも位置 t が位置 $t+1$ にアテンションを向けてそこにある自身のターゲットを読み取れてしまうため（Figure 11.9）、次トークン予測は自明なコピーへと退化してしまう。

BERT (Devlin et al. 2018) で導入された**マスク言語モデリング**（masked language modeling、

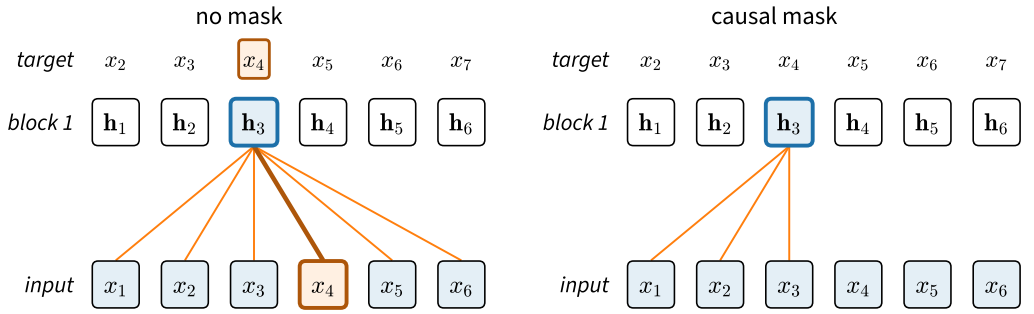


Figure 11.9: 1 ブロックにおける次トークン予測。マスクがない場合、位置 3 は自身のターゲットである x_4 を含むすべての入力にアテンションを向ける。因果マスクがある場合、位置 1 から 3 のみを読み取る。

MLM）は、次トークン予測を穴埋め課題（cloze task）に置き換える。すなわち、位置の部分集合 $M \subset \{1, \dots, n\}$（通常はトークンの $p = 15\%$）をランダムに選び、各 x_i（$i \in M$）を専用の `<mask>` トークン ID に置き換えて破損入力 $\tilde{\mathbf{x}}$ を作り、**マスクされた位置のみ**（$i \in M$）で元のトークンを復元する交差エントロピーを最小化する。

$$\mathcal{L}_{\text{MLM}}(\theta) \;=\; -\frac{1}{|M|} \sum_{i \in M} \log P_\theta(x_i \mid \tilde{\mathbf{x}}). \tag{11.10}$$

4 ブロックのバイトレベル `MaskedLM`（$d = 128, H = 4, L = 4$）を、11.2 節とまったく同じ WikiText-103 のバイトストリーム上で訓練する。その際、トークン ID `MASK = len(data.vocab)`（256）を `<mask>` 用に確保する。

```
data = d2l.WikiText103(batch_size=64, num_steps=64)
MASK = len(data.vocab)  # id of the extra <mask> embedding row

def mask_tokens(X, p=0.15):
    mask = torch.rand(X.shape, device=X.device) < p
    return torch.where(mask, MASK, X), mask

class MaskedBytes:
    """The windows of data with masked bytes; labels -100 add no loss."""
    def __init__(self, data):
        self.data = data

    def train_dataloader(self):
        return self.masked(self.data.train_dataloader())

    def val_dataloader(self):
        return self.masked(self.data.val_dataloader())

    def masked(self, batches):
        for X, _ in batches:
            Xm, mask = mask_tokens(X)
            yield Xm, torch.where(mask, X, -100)
```

```
class MaskedLM(nn.Module):
    """An encoder with an output head tied to its token embedding."""
    def __init__(self, encoder):
        super().__init__()
        self.encoder = encoder

    def forward(self, X):
        return F.linear(self.encoder(X), self.encoder.token_emb.weight)

device = d2l.try_gpu()
model = MaskedLM(TransformerEncoder(len(data.vocab) + 1)).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                              weight_decay=0.0)
losses = d2l.train_steps(model, MaskedBytes(data), optimizer, 2000)
print('masked loss at step 500/1000/2000: ' + '/'.join(
    f'{sum(losses[k-50:k]) / 50:.2f}' for k in (500, 1000, 2000)))
```

```
masked loss at step 500/1000/2000: 3.12/2.28/1.60
```

11.4.2.3 利用可能な文脈ごとの損失

11.2.3.2 節では、因果言語モデルの位置ごとの検証損失は左端（$t = 0$、1 バイトの文脈）から右端（$t = T - 1$、完全なプレフィックス文脈）に向かって単調に減少していた。では、`loss_by_position` を用いて双方向の `MaskedLM` における各ウィンドウ位置 $t \in \{0, \dots, n-1\}$ での復元損失を評価すると、どのような曲線になるだろうか。

```
def loss_by_position(model, batches):
    """Mean loss at each window position over the labels that are not -100,
    and the fraction of those labels predicted exactly."""
    device = next(model.parameters()).device
    model.eval()
    total = count = correct = 0
    with torch.no_grad():
        for X, Y in batches:
            X, Y = X.to(device), Y.to(device)
            logits, scored = model(X), Y != -100
            total += F.cross_entropy(logits.transpose(1, 2), Y,
                                     reduction='none').sum(0)  # 0 at -100
            count += scored.sum(0)
            correct += ((logits.argmax(-1) == Y) & scored).sum()
    return (total / count).cpu(), float(correct / count.sum())

def show(losses):
    return ' '.join(f'{float(v):.2f}' for v in losses)

masked = MaskedBytes(data)
pos_loss, accuracy = loss_by_position(model, masked.val_dataloader())
interior = slice(16, 48)  # positions 16 to 47 of the 64
print(f'masked accuracy {accuracy:.2f}')
print(f'loss at positions 0-3: {show(pos_loss[:4])}; 60-63: '
      f'{show(pos_loss[-4:])}; interior mean {pos_loss[interior].mean():.2f}')
```

```
masked accuracy 0.53
loss at positions 0-3: 2.36 1.78 1.54 1.52; 60-63: 1.56 1.56 1.76 2.37; interior mean 1.54
```

双方向エンコーダは左右対称な**バスタブ型（U 字型）曲線**を描く。内部のどの位置も左右両方の文脈を利用でき（約 1.5〜1.6 nats/byte）、損失が上昇するのはウィンドウ境界によって片側の文脈が断ち切られる両端（$t \approx 0$ および $t \approx n-1$、約 2.4 nats/byte）だけである。マスクされた文で `model` を評価すると、右側の文脈が内部の欠損バイトの曖昧性をどのように解消するかがわかる。

```
snippet = 'Homarus gammarus , known as the European lobster or common '
ids = torch.tensor(data.vocab.encode(snippet), device=device)[None]
for pos in (11, 22, 28):
    Xm = ids.clone()
    Xm[0, pos] = MASK
    with torch.no_grad():
        logits = model(Xm)
    probs = F.softmax(logits[0, pos], -1)
    top = probs.topk(3)
    shown = snippet[:pos] + '_' + snippet[pos + 1:]
    print(f'{shown[:40]!r}... -> '
          + ', '.join(f'{data.vocab.decode([int(i)])!r} {p:.2f}'
                      for p, i in zip(top.values, top.indices)))
```

```
'Homarus gam_arus , known as the European'... -> 'p' 0.44, 'b' 0.17, ' ' 0.12
'Homarus gammarus , kno_n as the European'... -> 'w' 0.47, 'i' 0.24, 'o' 0.10
'Homarus gammarus , known as _he European'... -> 't' 0.96, 'T' 0.02, 's' 0.01
```

11.4.3 エンコーダ・デコーダにおけるクロスアテンション

11.4.3.1 アライメントが既知の系列対系列課題

見通しのよい系列対系列（sequence-to-sequence）のアライメント問題で**エンコーダ・デコーダ**（Figure 11.8 右）を調べるため、$V = 16$ 文字のアルファベット（a〜p）と長さ $T = 12$ からなる文字列反転でモデルを訓練する。ソース文字列 $\mathbf{x} = (x_1, \dots, x_T)$ が与えられたとき、デコーダは dec_in $= (\texttt{<bos>}, x_T, x_{T-1}, \dots, x_2)$ を受け取り、反転した文字列 $\mathbf{y} = (x_T, x_{T-1}, \dots, x_1)$ を予測する。

```
V, T = 16, 12   # 16 letters, strings of length 12
BOS = V         # decoder start token

def sample_batch(batch_size, device):
    src = torch.randint(0, V, (batch_size, T), device=device)
    tgt = src.flip(1)
    dec_in = torch.cat([torch.full((batch_size, 1), BOS, device=device),
                        tgt[:, :-1]], 1)
    return src, dec_in, tgt

def to_str(ids):
    return ''.join(chr(ord('a') + int(i)) for i in ids)

src, dec_in, tgt = sample_batch(1, device)
print(f'source {to_str(src[0])!r} -> target {to_str(tgt[0])!r}')
```

```
source 'hhkmdnbmelgi' -> target 'iglembndmkhh'
```

11.4.3.2 デコーダブロックと EncoderDecoder

DecoderBlock は、因果自己アテンションと SwiGLU FFN の間に**クロスアテンション**のサブ層（d2l.MultiHeadAttention）を挿入する。クロスアテンションでは、**クエリはデコーダから得られ（$\mathbf{Q} = \mathrm{Norm}(\mathbf{Y})\mathbf{W}_q \in \mathbb{R}^{B\times T_{\text{dec}}\times d}$）、キーと値はエンコーダの最上層から得られる（$\mathbf{K} = \mathbf{H}_{\text{enc}}\mathbf{W}_k \in \mathbb{R}^{B\times T_{\text{src}}\times d}$、$\mathbf{V} = \mathbf{H}_{\text{enc}}\mathbf{W}_v \in \mathbb{R}^{B\times T_{\text{src}}\times d}$）**。エンコードされたソース $\mathbf{H}_{\text{enc}}$ の全体がデコーディング開始前にすでに存在するため、クロスアテンションでは**因果マスクを使用しない**。

```
class DecoderBlock(nn.Module):
    """Pre-norm decoder block: causal self-attention, cross-attention,
    FFN."""
    def __init__(self, num_hiddens, num_heads):
        super().__init__()
        self.norm1 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.norm2 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.norm3 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.self_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.cross_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.ffn = d2l.FeedForward(num_hiddens)

    def forward(self, X, enc, mask=None, cross_mask=None):
        if mask is None:  # causal self-attention
            mask = d2l.causal_mask(X.shape[1], device=X.device)
        Y = self.norm1(X)
        X = X + self.self_attention(Y, Y, Y, mask)
        X = X + self.cross_attention(self.norm2(X), enc, enc, cross_mask)
        return X + self.ffn(self.norm3(X))
```

EncoderDecoder は、TransformerEncoder と DecoderBlock 層のスタックおよび線形出力ヘッドを組み合わせたものである。なお、decode(dec_in, enc) は事前計算されたエンコーダ出力 enc を受け取るため、自己回帰生成の際にはソース系列に対してエンコーダが **1 回だけ**実行され、その出力を使って decode がターゲットトークンを 1 ステップずつ進める。

```
class EncoderDecoder(nn.Module):
    """A causal decoder cross-attending into a bidirectional encoder."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4,
                 num_blks=1, max_len=64):
        super().__init__()
        self.encoder = TransformerEncoder(vocab_size, num_hiddens,
                                          num_heads, num_blks, max_len)
        self.tgt_emb = nn.Embedding(vocab_size, num_hiddens)
        self.pos_emb = nn.Embedding(max_len, num_hiddens)
        for emb in (self.tgt_emb, self.pos_emb):
            nn.init.normal_(emb.weight, std=0.02)
        self.blks = nn.ModuleList([DecoderBlock(num_hiddens, num_heads)
                                   for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.head = nn.Linear(num_hiddens, vocab_size, bias=False)
```

```
    def decode(self, dec_in, enc):
        H = self.tgt_emb(dec_in) + self.pos_emb(
            torch.arange(dec_in.shape[1], device=dec_in.device))
        for blk in self.blks:
            H = blk(H, enc)
        return self.head(self.norm(H))

    def forward(self, src, dec_in):
        return self.decode(dec_in, self.encoder(src))
```

11.4.3.3 訓練と自己回帰デコーディング

EncoderDecoder を 128 個の文字列からなるランダムバッチ上で 300 ステップ訓練し、貪欲な自己回帰デコーディング（reverse）を実行すると、ほぼ完璧な文字列全体の反転が達成される。

```
def train_reversal(model, num_steps=300):
    """Train on fresh batches of 128 strings; return the loss of each step."""
    optimizer = d2l.LMOptimizer(model, lr=1e-3, num_steps=num_steps,
                                warmup=0, weight_decay=0.0)
    losses = []
    for step in range(num_steps):
        src, dec_in, tgt = sample_batch(128, device)
        logits = model(src, dec_in)
        loss = F.cross_entropy(logits.flatten(0, 1), tgt.flatten())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.detach())  # Stays on the device
    return torch.stack(losses).tolist()  # One transfer at the end

seq2seq = EncoderDecoder(V + 1).to(device)
losses = train_reversal(seq2seq)
print('loss at step 25/50/100/200/300: ' + '/'.join(
    f'{sum(losses[k-10:k]) / 10:.3f}' for k in (25, 50, 100, 200, 300)))
```

```
loss at step 25/50/100/200/300: 2.077/0.948/0.108/0.004/0.002
```

推論時には、reverse(model, src) がソース系列を 1 回エンコードし、<bos> からステップごとに 1 つずつターゲットトークンを貪欲にデコードすることで、ホールドアウトされた 1,000 個のランダム文字列に対してほぼ完璧な完全一致を達成する。

```
def reverse(model, src):
    """Greedy decoding: the encoder runs once, the decoder once per token."""
    model.eval()
    with torch.no_grad():
        enc = model.encoder(src)
        out = torch.full((len(src), T + 1), BOS, device=src.device)
        for t in range(T):
            logits = model.decode(out[:, :t + 1], enc)
            out[:, t + 1] = logits[:, -1].argmax(-1)
```

```
        return out[:, 1:]

    src, _, tgt = sample_batch(1000, device)
    pred = reverse(seq2seq, src)
    print(f'exact match on 1000 fresh strings: '
          f'{(pred == tgt).all(-1).float().mean():.3f}')
    print(f'source {to_str(src[0])!r} -> predicted {to_str(pred[0])!r}')
```

```
exact match on 1000 fresh strings: 1.000
source 'gjapkpfkolmp' -> predicted 'pmlokfpkpajg'
```

11.4.3.4 クロスアテンションのアライメントの読み取り

文字列反転では、ターゲットのステップ t はソースのトークン $T-1-t$ を出力する。したがって、クロスアテンションの重み行列は反対角線（anti-diagonal）$(t, T-1-t)$ に集中しなければならないのだろうか。そこで、1 つのエンコーダブロックと 1 つのデコーダブロックを持つ訓練済みの `EncoderDecoder`（エンコーダ自己アテンションあり：`with`）を、エンコーダがブロックを持たず埋め込みのみからなるそれ以外は同一のモデル（なし：`without`、したがってエンコーダ位置 j はソーストークン j の情報しか保持できない）と比較してみよう。

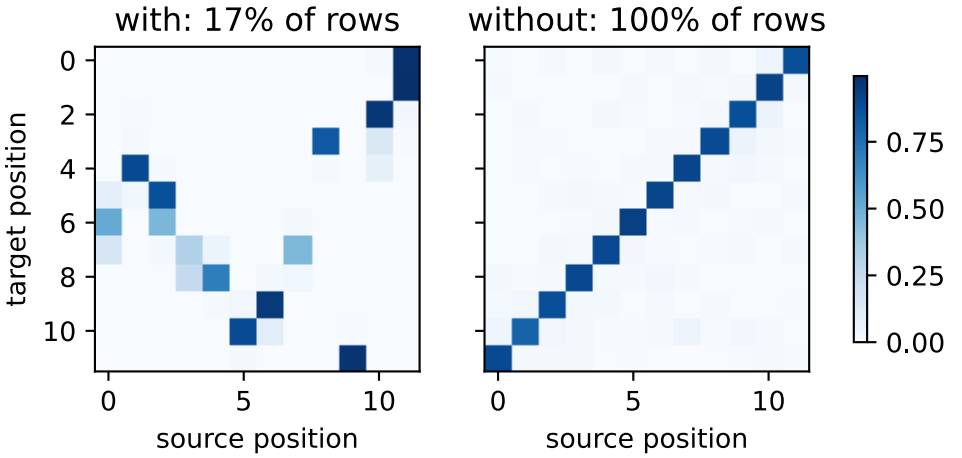
```
    def reversal_model(self_attention=True):
        model = EncoderDecoder(V + 1).to(device)
        if not self_attention:  # the encoder keeps embeddings and positions
            model.encoder = TransformerEncoder(V + 1, num_blks=0).to(device)
        train_reversal(model)
        return model

    def alignment(model):
        """Exact match on 1000 fresh strings; the head-averaged cross-attention
        of 256 strings, and how often its largest weight in a row falls on the
        true source position."""
        src, _, tgt = sample_batch(1000, device)
        match = (reverse(model, src) == tgt).all(-1).float().mean()
        with torch.no_grad():
            src, dec_in, _ = sample_batch(256, device)
            model(src, dec_in)
        maps = model.blks[0].cross_attention.attention_weights.mean(1)
        want = torch.arange(T - 1, -1, -1, device=device)  # row t: column T-1-t
        return match, (maps.argmax(-1) == want).float().mean(), maps[0].cpu()
```

エンコーダ自己アテンションあり（`with`）となし（`without`）の両方について独立した 6 つのシードを実行し、各グループで反対角線へのアライメントが最も弱いクロスアテンションのヒートマップをプロットすると、直接的な位置参照と多層ルーティングの違いが鮮明になる。

```
    encoders = {'with': [seq2seq] + [reversal_model() for _ in range(5)],
                'without': [reversal_model(False) for _ in range(6)]}
    maps, titles = [], []
    for name, models in encoders.items():
        runs = [alignment(m) for m in models]
        print(f'encoder {name} self-attention, six runs')
        print('  exact match:', ' '.join(f'{m:.3f}' for m, _, _ in runs))
```

```
        print('  rows on the true source position:',
              ' '.join(f'{100 * h:.0f}%' for _, h, _ in runs))
        _, hit, least = min(runs, key=lambda run: run[1])  # the least aligned
        maps.append(least)
        titles.append(f'{name}: {100 * hit:.0f}% of rows')
    d2l.show_heatmaps(d2l.stack(maps)[None], xlabel='source position',
                      ylabel='target position', titles=titles, figsize=(6, 3),
                      cmap='Blues')
```

```
encoder with self-attention, six runs
  exact match: 1.000 1.000 1.000 1.000 1.000 1.000
  rows on the true source position: 31% 92% 100% 45% 17% 100%
encoder without self-attention, six runs
  exact match: 1.000 1.000 1.000 1.000 1.000 1.000
  rows on the true source position: 100% 100% 100% 100% 100% 100%
```



エンコーダに自己アテンションがない場合(without)、クロスアテンションは反対角線 $(t, T-1-t)$ を直接参照してソーストークン $T-1-t$ を取り出さ**なければならない**。一方、エンコーダが双方向自己アテンションを含む場合（with）、クロスアテンションが $\mathbf{H}_{\text{enc}}$ を読み取る前に、エンコーダ層がすでに位置間でソース情報をルーティングまたは再編成できる。これは、多層ネットワークにおける生のアテンション重みがトークン対トークンの直接的な帰属マップ（attribution map）ではなく層間ルーティングを反映しているという古典的な教訓を改めて示している (Jain and Wallace 2019)。

11.4.4 インターフェースとしてのクロスアテンション：固定サイズの潜在ボトルネック

クロスアテンションで必要なのは、クエリ $\mathbf{Q} \in \mathbb{R}^{B\times M\times d}$ とキー・値 $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{B\times N\times d}$ が同じ特徴幅 d を共有することだけであるため、内部の処理長 M を生の入力長 N から切り離すことができる（Figure 11.10）。

Perceiver (Jaegle et al. 2021)（およびマルチモーダルな視覚・言語リサンプラ (Alayrac et al. 2022)）では、学習された $M \ll N$ 個の潜在クエリベクトルの配列 $\mathbf{Z}_0 \in \mathbb{R}^{M\times d}$ が、長さ N トークンの入力 $\mathbf{X} \in \mathbb{R}^{B\times N\times d}$ に対して $O(MNd)$ 回の演算でクロスアテンションを向け、その後

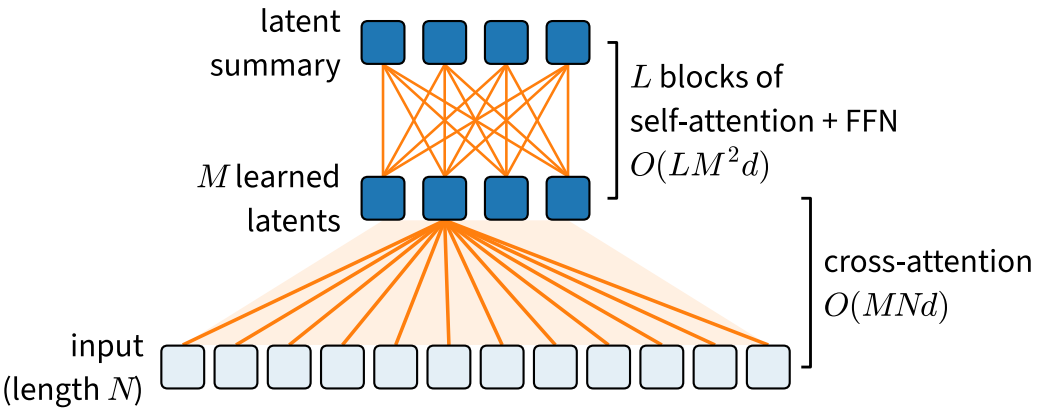


Figure 11.10: 入力長 N、学習された M 個の潜在変数、幅 d、および L 個の潜在ブロックからなる潜在ボトルネック。すべての潜在変数がすべての入力を読み取る（網掛けの帯。1 つの潜在変数による読み取りのみすべての線を描画している）。1 回のクロスアテンションによる読み取りの混合コストは $O(MNd)$ であり、後続の各潜在自己アテンション層の混合コストは $O(M^2d)$、スタック全体では $O(LM^2d)$ となる。これらの項ではバッチサイズ、ヘッド数、幅を固定とし、射影と FFN は省略している。

に L 個の自己アテンションブロックが長さ M のコンパクトな潜在配列を $O(LM^2d)$ 回の演算で（$O(LN^2d)$ ではなく）処理する。

```
class PerceiverEncoder(nn.Module):
    """M learned latents cross-attend into the input, then process among
    themselves."""
    def __init__(self, num_latents, num_hiddens, num_heads=4, num_blks=2):
        super().__init__()
        self.latents = nn.Parameter(
            0.02 * torch.randn(num_latents, num_hiddens))
        self.norm_q = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.cross_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads)
            for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)

    def forward(self, X):
        Z = self.latents.expand(X.shape[0], -1, -1)
        Z = Z + self.cross_attention(self.norm_q(Z), X, X)  # O(MN)
        for blk in self.blks:
            Z = blk(Z)                                      # O(M^2)
        return self.norm(Z)
```

11.4.5 3 つの配線の使い分け

- **エンコーダのみ（`is_causal=False`）**は、1 回の順伝播ですべてのトークンに完全な双方向文脈を与えるため、密なテキスト埋め込み、リランキング、トークン分類、および非自己回帰

型の視覚バックボーン（11.5 節）における標準的なバックボーンとなっている。
- **デコーダのみ（`is_causal=True`）**は、次トークン予測によって系列トークンの 100% （MLM の 15% に対して）で訓練でき、プロンプトと生成される続きを単一の統合されたパラメータスタックおよび KV キャッシュ（11.3 節）で処理できるため、汎用言語モデルの主流アーキテクチャとなっている。
- **エンコーダ・デコーダ（クロスアテンション付き）**は、Whisper (Radford et al. 2023) における音声スペクトログラムからテキストへの変換や、固定サイズの潜在クロスアテンション・インターフェースを通じて圧縮される高解像度の視覚・センサ配列（Figure 11.10）のように、入力と出力のモダリティや長さが異なるときに真価を発揮する。

11.4.6 まとめ

同じ `TransformerBlock` を、**双方向エンコーダ**（`is_causal=False`）、**因果デコーダ**（`is_causal=True`）、または**クロスアテンション付きエンコーダ・デコーダ**（Figure 11.8）として配線できる。マスクなしの自己アテンションでは次トークン予測のもとですべての位置が自身のターゲットを読み取れてしまうため、双方向エンコーダは**マスク言語モデリング（MLM）**（Equation 11.10）で訓練され、ウィンドウ全体にわたって対称な U 字型の `loss_by_position` 曲線を生じる。**エンコーダ・デコーダ**では、各 `DecoderBlock` がターゲットの因果自己アテンション、**クロスアテンション**（デコーダのクエリがエンコーダのキーと値にアテンションを向ける）、および位置ごとの FFN を交互に配置するため、エンコーダはソースに対して 1 回だけ実行され、デコーダが自己回帰的にステップを進める。より一般には、学習された $M \ll N$ 個のクエリベクトルによるクロスアテンション（`PerceiverEncoder`）を用いることで、長さ N トークンの入力を長さ M トークンの潜在ボトルネックへと $O(MNd + LM^2d)$ 時間で圧縮できる（Figure 11.10）。

11.4.7 演習問題

1. **BERT の 80 / 10 / 10 マスキングレシピ**。`mask_tokens` は選択されたすべての位置（バイトの 15%）を `<mask>` に置き換えている。`<mask>` はファインチューニング時や推論時には決して現れないため、Devlin et al. (2018) は選択されたトークンを 80% の割合で `<mask>` に、10% の割合で語彙から一様ランダムに選んだトークンに置き換え、残りの 10% の割合では元のトークンをそのまま残している。この 80 / 10 / 10 ルールを `mask_tokens` に実装し、10% をそのまま残し 10% をランダムにすることで、なぜエンコーダが**マスクされていない**トークンでも `<mask>` トークンと同様に正確な表現を維持せざるを得なくなるのか説明せよ。

2. **不揃いな系列対系列バッチに対するパディングマスク**。バッチ内に長さ $(s_1, \dots, s_B) \le S$ の可変長ソース文字列と長さ $(t_1, \dots, t_B) \le T_{\text{dec}}$ の可変長ターゲット文字列が含まれる場合、3 つのブールマスクが必要になる。すなわち、(a) $(B, 1, S, S)$ のエンコーダ用パディングマスク、(b) $(B, 1, T_{\text{dec}}, T_{\text{dec}})$ のデコーダ用因果＋パディングマスク、(c) $(B, 1, T_{\text{dec}}, S)$ の長方形のクロスアテンション用ソースパディングマスクである。これら 3 つのマスクをすべて構築するヘルパ関数 `make_seq2seq_masks(src_len, tgt_len, S, T_dec)` を作成し、パディングトークンを `src` に付加して `EncoderDecoder` に渡しても、パディング以外の位置におけるデコーダのロジットが変化しないことを検証せよ。

3. **MLM のマスク率 p の変化**。`MaskedLM` を 2,000 ステップにわたりマスク率 $p \in \{0.05, 0.15, 0.30, 0.50\}$ でそれぞれ訓練し、4 つのモデルすべてを同一の $p = 0.15$ の検証セット `masked.val_dataloader()` で評価せよ。p は、バッチあたりの教師あり勾配信号の数と各マスクの周囲で利用可能な破損していない文脈の量とをどのようにトレードオフしているか。

4. **因果モデルと双方向モデルにおける時間の順方向と逆方向**。因果言語モデルはテキストを左から右（$x_1 \to x_n$）へと処理する。`WikiText103` のすべてのバイトウィンドウを反転させ

て因果モデルが右から左（$x_n \to x_1$）へと予測するようにして 1,000 ステップ訓練し、バイトあたりの平均検証損失と `loss_by_position` 曲線を、左から右への因果モデルおよび双方向の `MaskedLM` と比較せよ。同時対数尤度の合計 $\sum_{t=1}^{n} \log P(x_t \mid \text{past})$ が理論上は時間反転に対して不変であるのはなぜか。また、双方向モデルは両方の片方向モデルに対してどこで優位性を得るのか。

5. **内容ベース対位置ベースのクロスアテンション・アライメント。**本節の文字列反転実験では、ランダム文字列内のどの文字も重複し得るため、重複文字を区別するには位置情報が必要である。長さ 12 の各文字列が 12 個の**相異なる**文字のランダムな順列になるよう `sample_batch` を修正し、エンコーダの順伝播から位置項を取り除いた状態で `EncoderDecoder` を訓練せよ。位置埋め込みを持たない 1 層のエンコーダが、12 文字すべてが相異なる場合でさえ文字列反転に失敗する一方で、ソート課題（入力文字をアルファベット順 `a..p` に出力する課題）ではエンコーダの位置埋め込みが一切なくても成功するのはなぜか。

6. **系列対系列の反転のための Perceiver IO。**Perceiver IO (Jaegle et al. 2022) は、`Perceiver-Encoder`（学習された M 個の潜在クエリが長さ N トークンの入力にアテンションを向ける）と、クロスアテンション**デコーダ**（N_{out} 個のターゲット位置埋め込み $\mathbf{Q}_{\text{out}} \in \mathbb{R}^{N_{\text{out}} \times d}$ が M 個の潜在ベクトル $\mathbf{Z}^{(L)} \in \mathbb{R}^{B \times M \times d}$ にクロスアテンションを向ける）とを組み合わせる。12 文字の文字列を $M \in \{2, 4, 8, 12\}$ 個の潜在変数（$d = 128$）にエンコードし、1 回の並列なクロスアテンション読み出しパスで反転文字列をデコードする `PerceiverAutoencoder` を構築せよ。ボトルネックのサイズ M に応じて反転の正解率はどのように変化するか。

11.5 Vision Transformer（ViT）

畳み込みニューラルネットワーク（第 6 章、第 7 章）は、画像に関する 2 つの幾何学的な事前仮定をアーキテクチャに直接組み込んでいる。すなわち、**空間的局所性**（spatial locality、$k \times k$ カーネルが隣接するピクセルのみを結合する性質）と**平行移動同変性**（translation equivariance、同じカーネルの重みがすべての (r, c) 位置にわたってスライドする性質）である。では、2 次元グリッド上の近傍やシフト同変性という概念を**まったく**組み込んでいない標準的な双方向 Transformer エンコーダ（11.4 節）で、競争力のある画像分類ができるのだろうか。

個々のピクセルに直接自己アテンションを適用すると、計算量はピクセル数の二乗で増大する。96×96 の画像でさえ 9,216 ピクセルあり、ヘッドあたり $9216^2 \approx 8.5 \times 10^7$ 個のアテンションペアが生じる。そこで Dosovitskiy et al. (2021) は **Vision Transformer（ViT）**（Figure 11.11）を導入し、画像を重なりのない $p \times p$ の**パッチ**のグリッドへとトークン化した。96×96 の画像は、パッチサイズを $p = 16$ とすると、$m = (96/16)^2 = 36$ 個のパッチトークンからなる短い系列になる。各パッチを幅 d に射影し、学習可能な `<cls>` トークンを先頭に付加し、学習型の 1 次元位置埋め込み（position embedding）を加えたうえで、合計 37 個のトークンを L 個の双方向 Pre-Norm Transformer ブロックに通す。

11.5.1 トークンとしてのパッチ（`PatchEmbedding`）

形状 (B, C, H, W) の画像テンソルを重なりのない $p \times p$ のパッチに分割し、各パッチに共通の線形射影 $\mathbb{R}^{Cp^2} \to \mathbb{R}^d$ を適用する操作は、**カーネルサイズ** $p \times p$、**ストライド** p、**出力チャネル数** d **の 2 次元畳み込み**を適用した後に、高さ H/p、幅 W/p の 2 次元空間グリッドを $m = (H/p)(W/p)$ 個のトークンベクトルからなる系列へと平坦化する操作と同一である。

```
class PatchEmbedding(nn.Module):
    """Image-to-sequence stem of the vision transformer."""
```

```
    def __init__(self, img_size=96, patch_size=16, num_hiddens=512,
                 num_channels=3):
        super().__init__()
        # A partial trailing patch would silently shrink the token grid
        assert img_size % patch_size == 0, 'patches must tile the image'
        self.num_patches = (img_size // patch_size) ** 2
        self.conv = nn.Conv2d(num_channels, num_hiddens,
                              kernel_size=patch_size, stride=patch_size)

    def forward(self, X):  # (batch, c, h, w) -> (batch, patches, d)
        return self.conv(X).flatten(2).transpose(1, 2)
```

4 枚の 96×96 単一チャネル画像からなるミニバッチに対し、パッチサイズ $p = 16$ で PatchEmbedding を適用すると、$(96/16)^2 = 36$ 個の幅 $d = 512$ のパッチトークンが出力される。

```
img_size, patch_size, num_hiddens, batch_size = 96, 16, 512, 4
patch_emb = PatchEmbedding(img_size, patch_size, num_hiddens, num_channels=1)
X = torch.zeros(batch_size, 1, img_size, img_size)
d2l.check_shape(patch_emb(X),
                (batch_size, (img_size//patch_size)**2, num_hiddens))
```

11.5.2 Vision Transformer のアーキテクチャ

11.5.2.1 エンコーダブロック（ViTBlock）

Dosovitskiy et al. (2021) に従い、ViTBlock は設定可能な d2l.TransformerBlock（11.1 節）を継承し、Pre-Norm の LayerNorm と、GELU を活性化関数とする 2 層の多層パーセプトロン（multilayer perceptron、MLP、隠れ層幅は mlp_num_hiddens）を用いる。デフォルトではアテンションの射影にバイアスを持たせない（use_bias=False）一方、MLP の 2 つの線形層と LayerNorm にはバイアスを残す。

```
class ViTBlock(d2l.TransformerBlock):
    """Pre-norm block with LayerNorm and an exact-GELU MLP."""
    def __init__(self, num_hiddens, mlp_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__(
            num_hiddens, num_heads, dropout=dropout, norm='layer',
            bias=use_bias, ffn_factory=lambda: nn.Sequential(
                nn.Linear(num_hiddens, mlp_num_hiddens),
                nn.GELU(),  # Exact GELU
                nn.Dropout(dropout),
                nn.Linear(mlp_num_hiddens, num_hiddens)))
```

11.5.2.2 <cls> トークン、位置埋め込み、および ViT

特定の空間位置を優遇することなく m 個のパッチ表現を画像全体の単一ベクトルへと集約するため、ViT は学習可能なパラメータベクトル cls_token（形状 $(1, 1, d)$）をインデックス 0 の先頭に付加し、学習可能な位置埋め込みテーブル pos_embedding（形状 $(1, m + 1, d)$）を $m + 1$ 個の位置すべてに加算する。そのうえで L 層の ViTBlock を実行し、LayerNorm で正規化して、インデックス 0（X[:, 0]）を線形分類ヘッドに入力する。

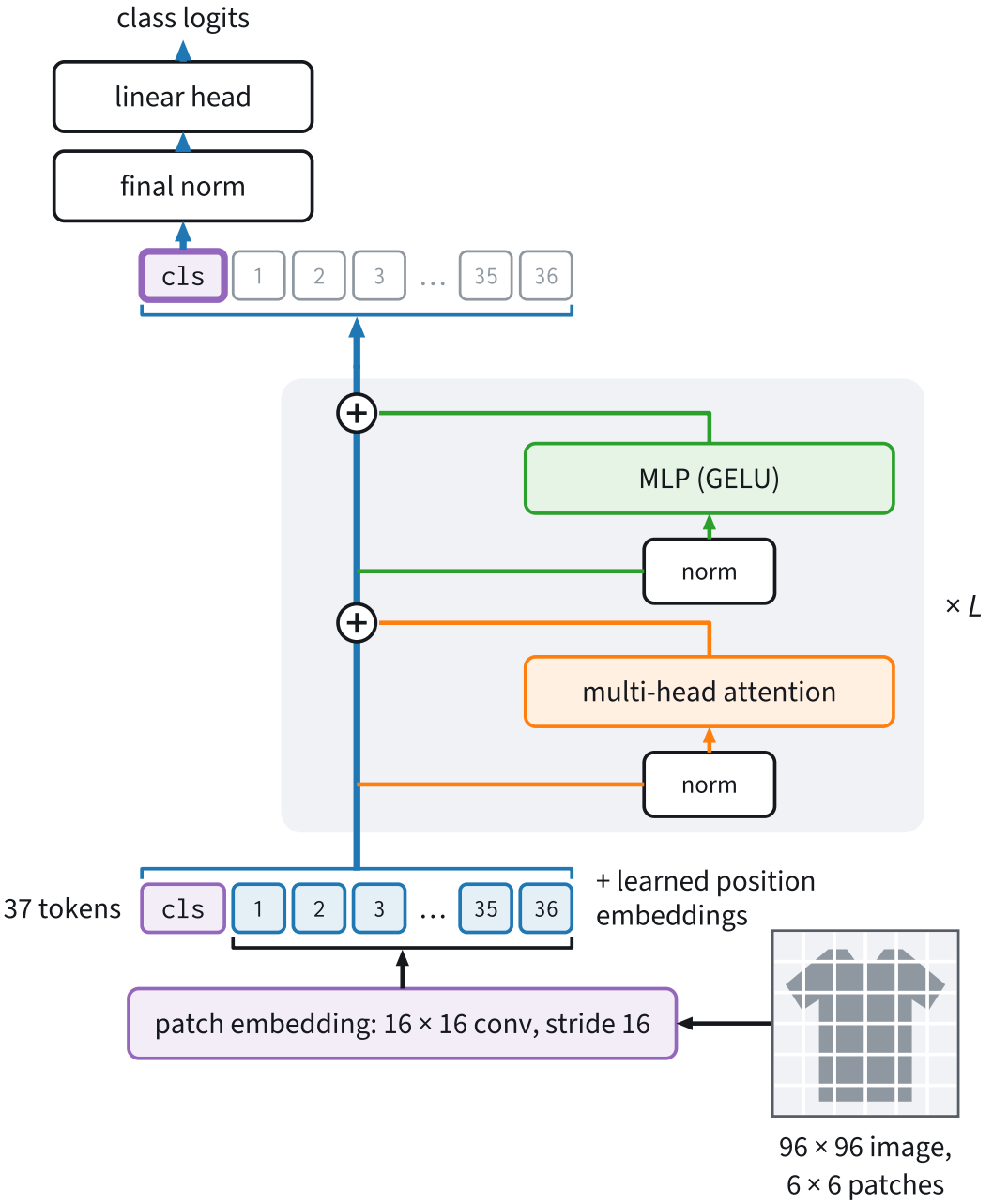


Figure 11.11: 96 × 96 画像に対する Vision Transformer。ストライド付き畳み込みによって 6 × 6 グリッドに並ぶ 16 × 16 パッチが 36 個のトークンに変換される。学習された「<cls>」トークンが先頭に付加され、学習された位置埋め込みが加算される。続いて L 個の Pre-Norm Transformer ブロックが全 37 トークンを処理し、線形ヘッドが「<cls>」トークンの最終表現を分類する。

```
class ViT(d2l.Classifier):
    """Vision transformer."""
    def __init__(self, *, img_size, patch_size, num_hiddens, mlp_num_hiddens,
                 num_heads, num_blks, num_steps, emb_dropout=0.0,
                 blk_dropout=0.0, lr=1e-3, use_bias=False, num_classes=10,
                 num_channels=1):
        super().__init__()
        self.save_hyperparameters()
        self.patch_embedding = PatchEmbedding(
            img_size, patch_size, num_hiddens, num_channels)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, num_hiddens))
        num_tokens = self.patch_embedding.num_patches + 1  # Add the cls token
        # Positional embeddings are learnable, initialized to small noise
        self.pos_embedding = nn.Parameter(
            torch.randn(1, num_tokens, num_hiddens) * 0.02)
        self.dropout = nn.Dropout(emb_dropout)
        self.blks = nn.ModuleList([
            ViTBlock(num_hiddens, mlp_num_hiddens, num_heads, blk_dropout,
                     use_bias=use_bias) for _ in range(num_blks)])
        self.head = nn.Sequential(nn.LayerNorm(num_hiddens, eps=1e-5),
                                  nn.Linear(num_hiddens, num_classes))

    def forward(self, X):
        X = self.patch_embedding(X)  # (batch, patches, d)
        cls = self.cls_token.expand(X.shape[0], -1, -1)
        X = self.dropout(torch.cat((cls, X), 1) + self.pos_embedding)
        for blk in self.blks:
            X = blk(X)
        return self.head(X[:, 0])

    def configure_optimizers(self):
        # AdamW decays weight matrices and kernels; it exempts biases, norm
        # parameters, the class token, and the position embeddings
        named = list(self.named_parameters())
        decay = lambda n, p: p.ndim >= 2 and n not in ('cls_token',
                                                       'pos_embedding')
        optim = torch.optim.AdamW(
            [{'params': [p for n, p in named if decay(n, p)]},
             {'params': [p for n, p in named if not decay(n, p)],
              'weight_decay': 0.0}], lr=self.lr, weight_decay=0.05)
        # Linear warmup over the first tenth of the steps, then cosine
        return optim, torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
```

11.5.3 Fashion-MNIST での訓練

11.5.3.1 最初の実行

2 ブロックの `ViT`（$d = 512, d_{\text{mlp}} = 2048, H = 8, L = 2, p = 16$、パラメータ数 ≈ 6.5M）を、96×96 の Fashion-MNIST 上で 10 エポック訓練する。最適化には線形ウォームアップとコサイン学習率減衰を伴う AdamW（$\lambda = 0.05$）を用いる。

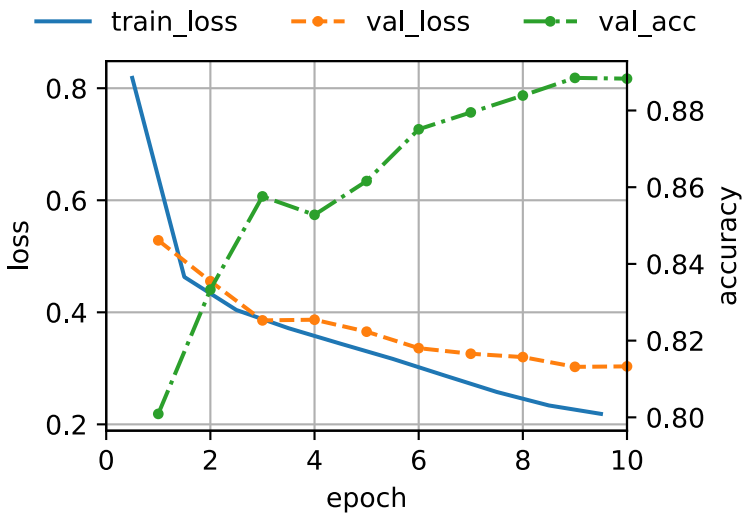
```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
num_steps = 10 * len(data.train_dataloader())  # Ten epochs of minibatches
hparams = dict(img_size=96, patch_size=16, num_hiddens=512,
               mlp_num_hiddens=2048, num_heads=8, num_blks=2,
               emb_dropout=0.1, blk_dropout=0.1, num_steps=num_steps)
```

訓練を始める前に、`model = vit()` をインスタンス化し、ランダムに初期化されたパッチ位置埋め込みのコピーを `pos_init` として保存しておく。これにより、初期化時点から訓練によってどれだけの空間構造が加わったかを後で計測できる。

```
vit = lambda: ViT(**hparams, lr=3e-4)  # A fresh model at the tuned rate
model = vit()
pos_init = model.pos_embedding[0, 1:].detach().clone()  # The initial noise
```

`model` を 10 エポックにわたり `d2l.Trainer` で適合させると、検証正解率は約 89% に達する（アップサンプリングされた 96×96 の Fashion-MNIST 画像での評価）。

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
vit_history = trainer.history
vit_params, vit_acc = d2l.num_params(model), vit_history['val_acc'][-1]
print(f'ViT: {vit_params/1e6:.1f}M parameters, '
      f'validation accuracy {vit_acc:.2f}')
```




```
ViT: 6.5M parameters, validation accuracy 0.89
```

11.5.3.2 空間位置の役割と学習された 2 次元グリッドのトポロジー

`pos_embedding` がなければ、`ViT` は $m = 36$ 個のパッチに対して**置換不変**（permutation-invariant）である。36 個のパッチをどのように置換しても、すべてのアテンション行列の $1 \ldots 36$ 行目が同じように置換されるだけで、0 行目（<cls>）は文字通りまったく変化しないからである。学習された `pos_embedding` がある場合、パッチ k は位置ベクトル $\mathbf{p}_k$ を配置先のスロット k に応じて受

け取る。訓練済みの ViT が空間配置にどれだけ依存しているかを検証するため、テスト時に各検証画像の 36 個のパッチを 5 通りのランダムな順列で並べ替える（shuffled_accuracy）。

```
def shuffle_patches(X, perm, p=16):
    """Move the p x p blocks of images X: block perm[k] goes to grid cell k."""
    n, c, h, w = X.shape
    B = X.reshape(n, c, h // p, p, w // p, p).transpose(3, 4)  # Grid of blocks
    B = B.reshape(n, c, -1, p, p)[:, :, perm]  # Blocks in raster order, moved
    B = B.reshape(n, c, h // p, w // p, p, p).transpose(3, 4)
    return B.reshape(X.shape)

def shuffled_accuracy(model, data, perm):
    """Validation accuracy on images whose blocks are moved by perm."""
    return d2l.evaluate_accuracy(model, ((shuffle_patches(X, perm), y)
                                         for X, y in data.val_dataloader()))
```

shuffled_accuracy を 36 個のパッチに対する 5 通りの固定されたランダム順列にわたって評価し、テスト時に空間配置をかき混ぜたときに正解率がどれだけ急激に低下するかを計測する。

```
rng = np.random.default_rng(0)
perms = [rng.permutation(36) for _ in range(5)]  # Five fixed shuffles
accs = [shuffled_accuracy(model, data, perm) for perm in perms]
print(f'trained ViT: {vit_acc:.3f} as is, '
      f'{min(accs):.2f} to {max(accs):.2f} with its patches shuffled')
```

```
trained ViT: 0.888 as is, 0.25 to 0.40 with its patches shuffled
```

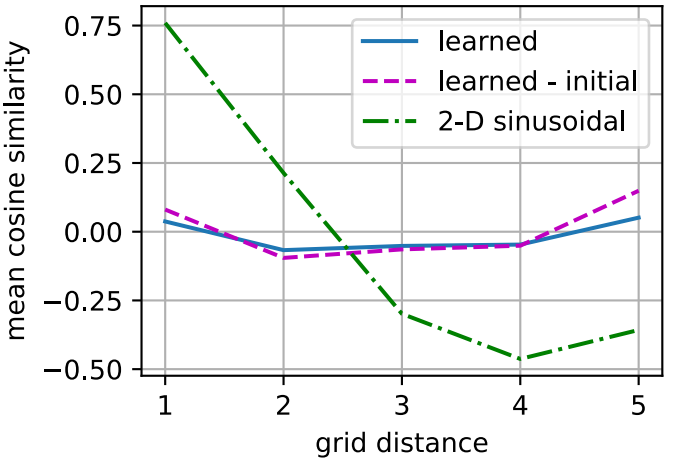
6×6 のパッチ配置をかき混ぜると、検証正解率は約 0.89 からシャッフルの仕方に応じて約 0.25〜0.45 まで低下するため、訓練済みの ViT は各パッチが置かれた位置に強く依存していることがわかる。それでもランダム予測の 0.1 を大きく上回っているのは、靴のかかとや襟の形状といった局所的な 16×16 パッチのテクスチャだけでも、「パッチの集合（bag of patches）」として大まかな衣類クラスの多くを識別できるからである。

構造を持たない 1 次元の位置テーブル pos_embedding（$36 \times d$）は、6×6 の画像グリッドが持つ 2 次元の幾何構造を実際に発見しているのだろうか。パッチの位置ベクトル間の平均コサイン類似度を、2 次元グリッド距離 $\max(|r-r'|, |c-c'|)$（6×6 グリッド上）の関数として計算すれば（grid_similarity）、この問いに答えられる。比較のため、構造上グリッドを最初からエンコードしている 2 次元正弦波符号と、そうではないランダムなベクトルについても同じ曲線を計算する。

```
def grid_similarity(P):
    """Mean cosine similarity of 36 position embeddings, by grid distance."""
    P = P - P.mean(0)                    # Drop the part shared by all
    P = P / torch.linalg.vector_norm(P, dim=1, keepdim=True)
    sim = P @ P.T                        # (36, 36) cosine similarities
    rows, cols = torch.meshgrid(torch.arange(6), torch.arange(6),
                                indexing='ij')
    ij = torch.stack((rows.flatten(), cols.flatten()), 1)  # Grid cells
    dist = (ij[:, None] - ij[None]).abs().amax(-1)  # Max of the two offsets
    return [float(sim[dist == k].mean()) for k in range(1, 6)]
```

```
S = d2l.sinusoidal_encoding(6, 256)      # 1-D codes of the 6 rows/columns
rows, cols = torch.meshgrid(torch.arange(6), torch.arange(6), indexing='ij')
fixed = torch.cat((S[rows.flatten()], S[cols.flatten()]), 1)  # (36, 512)
learned = model.pos_embedding[0, 1:].detach().cpu()  # Drop the cls slot
noise = torch.randn(36, 512)
curves = {'learned': grid_similarity(learned),
          'learned - initial': grid_similarity(learned - pos_init),
          '2-D sinusoidal': grid_similarity(fixed),
          'random': grid_similarity(noise)}
for name, curve in curves.items():
    print(f'{name:>17}:', '  '.join(f'{c:+.2f}' for c in curve))
d2l.plot(list(range(1, 6)), list(curves.values())[:3], 'grid distance',
         'mean cosine similarity', legend=list(curves)[:3])
```

```
          learned: +0.04  -0.07  -0.05  -0.05  +0.05
learned - initial: +0.08  -0.10  -0.06  -0.05  +0.15
   2-D sinusoidal: +0.76  +0.21  -0.30  -0.46  -0.36
           random: -0.03  -0.03  -0.03  -0.02  -0.03
```



10 エポック後の答えは「ごくわずか」である。隣接するパッチ（距離 1）と反対側の端にあるパッチ（距離 5）は、ランダムなベクトルよりもわずかに類似度が高くなるが（約 $+0.05$ 対 -0.03）、その中間の距離では学習された曲線は 0 をわずかに下回り、初期ノイズに訓練が付け加えた変化量は距離 5 で最大となる。対照的に、2 次元正弦波符号は距離 1 での約 0.75 から 0 を大きく下回る値まで低下する。Dosovitskiy et al. (2021) が ViT の位置埋め込みに見出した明瞭な 2 次元構造は、はるかに大規模なデータセットではるかに長く訓練することで初めて現れるものである。

11.5.4 パラメータ数を揃えた畳み込みネットワークとの比較

この 6.5M パラメータの `ViT` は、60,000 枚の画像からなるこのデータセットにおいて、パラメータ数を揃えた ResNet 風の畳み込みベースライン（`CompactResNet`、パラメータ数 ≈ 6.5M）をまったく同じコサイン減衰付きの AdamW スケジュールで 10 エポック訓練した場合と比べるとどうなるだろうか。

```
class CompactResNet(d2l.Classifier):
    """A ResNet-style CNN parameter-matched to the ViT above."""
    configure_optimizers = ViT.configure_optimizers  # The same recipe

    def __init__(self, num_steps, lr=1e-3, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
            d2l.Residual(64), d2l.Residual(64),
            d2l.Residual(128, strides=2), d2l.Residual(128),
            d2l.Residual(256, strides=2), d2l.Residual(256),
            d2l.Residual(512, strides=2),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes))

    def forward(self, X):
        return self.net(X)

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')  # He
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

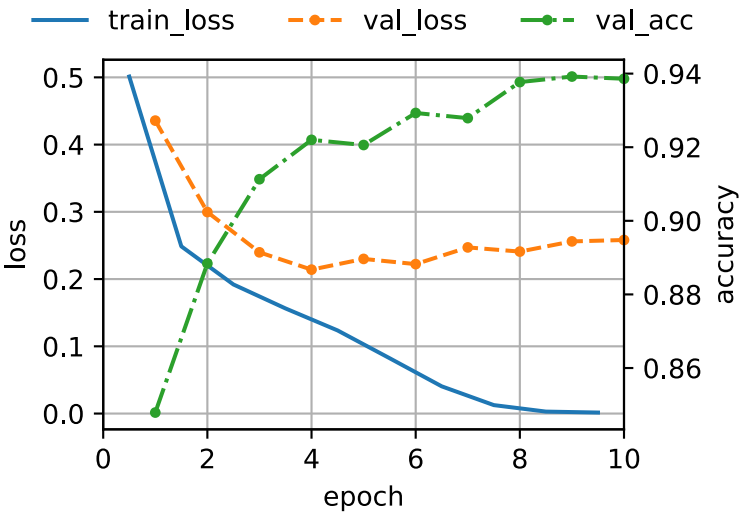
ファクトリ compact_cnn を作成し、cnn をピーク学習率 10^{-3} でインスタンス化する。

```
compact_cnn = lambda: CompactResNet(num_steps, lr=1e-3)  # A fresh CNN
cnn = compact_cnn()
```

CompactResNet を 10 エポック訓練し（さらに実行間の安定性を確認するため CompactResNet と ViT の両方で 2 つ目のシードでも訓練し）、まったく同じ 6.5M パラメータの予算で 2 つのアーキテクチャを比較する。

```
def final_accuracy(model, data):
    """Validation accuracy after the ten epochs of the first run."""
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False  # No progress plot for the repeats
    trainer.fit(model, data)
    return trainer.history['val_acc'][-1]

trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(cnn, data)
cnn_history = trainer.history
cnn_params, cnn_acc = d2l.num_params(cnn), cnn_history['val_acc'][-1]
cnn_runs = [cnn_acc, final_accuracy(compact_cnn(), data)]  # Run-to-run range
learned_runs = [vit_acc, final_accuracy(vit(), data)]  # A second ViT run
print(f'CNN: {cnn_params/1e6:.1f}M parameters, validation accuracy',
```

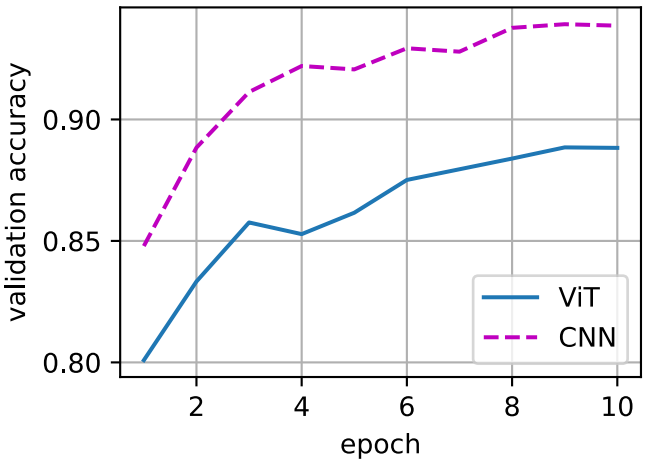
```
      ', '.join(f'{a:.3f}' for a in cnn_runs))
print(f'ViT: {vit_params/1e6:.1f}M parameters, validation accuracy',
      ', '.join(f'{a:.3f}' for a in learned_runs))
```




```
CNN: 6.5M parameters, validation accuracy 0.939, 0.939
ViT: 6.5M parameters, validation accuracy 0.888, 0.889
```

10 エポックにわたる ViT と CompactResNet の検証正解率の軌跡を重ねてプロットすると、訓練全体を通じたサンプル効率の差が浮き彫りになる。

```
d2l.plot(list(range(1, 11)), [vit_history['val_acc'], cnn_history['val_acc']],
         'epoch', 'validation accuracy', legend=['ViT', 'CNN'])
```



データ拡張や大規模な事前学習なしの 60,000 枚の小さな画像データセットでは、CompactResNet の方が速く収束し、最終的に ViT を約 5 パーセントポイント上回る。この結果は、Dosovitskiy

et al. (2021) が明らかにした根本的なトレードオフを示している。すなわち、CNN にハードコードされた 2 次元の局所性と平行移動同変性は小規模データ領域において強力な**帰納バイアス**（inductive bias）を与える一方、Vision Transformer は空間的な関係をデータから柔軟に学習するため、データ拡張なしの小さなデータセットでは CNN に後れを取るものの、数千万枚の画像で事前学習した場合（あるいは強力なデータ拡張、蒸留 (Touvron, Cord, Douze, et al. 2021)、マスク付きパッチ再構成 (He et al. 2022) を用いて訓練した場合）には CNN に匹敵するか、それを凌駕する。

11.5.5 まとめ

Vision Transformer（ViT）は、形状 (C, H, W) の画像を $m = (H/p)(W/p)$ 個のトークンからなる系列へとストライド p の畳み込み（`PatchEmbedding`）によって変換し、先頭に `<cls>` トークンを付加し、学習型の 1 次元位置埋め込みを加算したうえで、双方向の Pre-Norm Transformer エンコーダを適用する（Figure 11.11）。位置埋め込みがなければ、ViT はパッチに対して厳密に置換不変である。一方、学習型の 1 次元位置埋め込みを用いると、Fashion-MNIST 上での 10 エポックの訓練では位置テーブルに 2 次元グリッドがわずかに刻まれるにすぎないものの、訓練済みモデルはパッチ配置に強く依存するようになる。ViT は畳み込みのような組み込みの 2 次元の局所性や平行移動同変性を持たないため、データ拡張なしの小さなデータセットでゼロから訓練するとパラメータ数を揃えた ResNet の方が高い性能を示すが、大規模な事前学習スケールでは ViT が CNN を追い抜き、さらにマルチモーダルモデルに対する統一されたトークンインターフェースを提供する。

11.5.6 演習問題

1. **パッチサイズ p、パラメータ数、およびアテンションの FLOPs**。本節の 96×96 画像向け `ViT` では、パッチサイズは $p = 16$（$m = 36$ パッチ）である。
    1. `PatchEmbedding` のパラメータ数、`pos_embedding` のパラメータ数、および L 層の `ViTBlock` のパラメータ数は、パッチサイズを $p = 8$（$m = 144$）に半減させたとき、あるいは $p = 4$（$m = 576$）に 4 分の 1 にしたとき、それぞれどのように変化するか。
    2. 層あたりのアテンションスコア行列のサイズ $(m + 1)^2$ は、$p = 16$ から $p = 8$ および $p = 4$ にすると何倍に増大するか。$p = 8$ の `ViT` を 10 エポック訓練し、その実経過時間と検証正解率を $p = 16$ の場合と比較せよ。
2. **`<cls>` 読み出し対グローバル平均プーリング**。`<cls>` トークンによる `ViT` の読み出しを、$m = 36$ 個のパッチトークンにわたるグローバル平均プーリングに置き換えよ（`self.head(X[:, 1:].mean(1))` とするか、`cls_token` を完全に削除して全 36 個のトークンを平均する）。Fashion-MNIST 上で 10 エポック訓練し、検証正解率を `<cls>` 読み出しと比較せよ。
3. **位置埋め込みなしでの厳密な置換不変性**。`model.pos_embedding` をすべてゼロに設定し、`model` を評価モードに切り替えよ（ドロップアウトを無効化する）。次の性質を代数的に証明するとともに、検証バッチ上で誤差 10^{-6} 以内の精度で数値的に確認せよ。すなわち、`shuffle_patches(X, perm)` を任意の順列 `perm`（$\{0, \dots, 35\}$ 上の置換）に対して適用しても、出力ロジット `model(X)` はまったく変化しない。
4. **2 次元グリッド発見に関する統計的置換検定**。Δ を、隣接パッチ（グリッド距離 1）とグリッド距離 ≥ 4 のパッチとの間での `grid_similarity` の差（訓練済みの `model.pos_embedding` に対する値）とする。Δ の帰無分布を、`pos_embedding` の 36 行に対する 1,000 通りのランダムな順列にわたって計算し、経験 p 値を報告せよ。学習された位置埋め込みの微弱なグリッド構造は、偶然と区別できるか。
5. **サブパッチ単位対パッチ単位の平行移動感度**。CNN では、入力画像を s ピクセル平行移動させると、内部の特徴マップも（プーリングのストライドを除けば）s ピクセル平行移動す

る。一方、ViT（$p = 16$）では、画像を $s = 16$ ピクセル（1 パッチ幅ちょうど）平行移動させても各パッチが隣接するパッチスロットへ移るだけであるのに対し、$s = 8$ ピクセル（半パッチ分）平行移動させるとすべての 16×16 パッチ境界を横切ることになる。訓練済みの ViT と CompactResNet の検証正解率を、右へ $s \in \{0, 2, 4, 8, 12, 16\}$ ピクセル平行移動させた検証画像（左端はゼロパディングする）上で評価せよ。なぜ ViT は $s = 8$ において $s = 16$ と比べ非単調な落ち込みを示すのか。

6. **深さにわたる平均アテンション距離**。@Dosovitskiy.Beyer.Kolesnikov.ea.2021 は、各ヘッドの**平均アテンション距離**を、アテンション重みで加重平均した 2 次元ユークリッドピクセル距離 $\sum_{i,j=1}^{36} \alpha_{ij}\, d_{\text{pixel}}(i, j)/36$（クエリパッチ i とキーパッチ j の間の距離）として定義している。訓練済みの ViT のブロック 1 とブロック 2 に含まれる各 8 個のヘッドについて、平均アテンション距離を計算せよ。CNN で受容野が拡大していくのと同様に、浅い層のヘッドは深い層のヘッドよりも局所的にアテンションを向けているか。

11.6 スケーリング則

Transformer 言語モデル（11.2 節）を事前学習（pretraining）する際、計算コストを決定する巨視的な量が 2 つある。**非埋め込みパラメータ** N（non-embedding parameters、L 個の Transformer ブロック内のパラメータ）の個数と、オプティマイザに流し込まれる**訓練トークン数** D（training tokens）である。各トークンはパラメータあたり予測可能な回数の浮動小数点演算（floating-point operations、FLOPs）を必要とするため、訓練の計算予算 C（単位は FLOPs）を固定すると、モデルサイズ N とデータ予算 D の間に直接的なトレードオフが生じる。より大きなモデルをより少ないステップ数で訓練すべきか、それともより小さなモデルをより多くのトークンで訓練すべきか。

経験的な**ニューラルスケーリング則**（neural scaling laws）の研究 (Kaplan et al. 2020) により、検証交差エントロピー損失は滑らかなべき乗則（power law）の曲面 $\mathcal{L}(N, D)$ に従い、それが N と D の何桁にもわたる範囲で成り立つことが明らかになった。続いて Hoffmann et al. (2022) は、固定された事前学習計算量 $C \approx 6ND$ のもとで損失を最小化するには、モデルパラメータ $N^*(C)$ と訓練トークン数 $D^*(C)$ を**同じ比率**（$N^* \propto C^{1/2}, D^* \propto C^{1/2}$、パラメータあたり約 $D^*/N^* \approx 20$ トークン、Figure 11.12）でスケーリングする必要があることを示した。本節では、パラメータ数の集計式 $N \approx 12Ld^2$ と FLOP 則 $C \approx 6ND$ を導出して厳密な演算子カウンタに対して両者を検証し、15 モデルのスケーリンググリッド（5 種類のサイズと 3 種類のトークン予算の組み合わせ）を WikiText-103 上で実行したうえで、**推論コスト**（inference cost）を考慮すると現代のオープンモデルが Chinchilla の比率をはるかに超えてより小さなモデルを訓練する方向へシフトする理由を説明する。

```
from torch.nn.attention import SDPBackend, sdpa_kernel
from torch.utils.flop_counter import FlopCounterMode
```

11.6.1 パラメータ数（$N \approx 12Ld^2$）と FLOPs（$C \approx 6ND$）の算出

11.6.1.1 ブロックと埋め込み

密な L ブロックの GPT において、隠れ層の幅を d とし、標準的なマルチヘッドアテンション（$H_{\text{kv}} = H$）および SwiGLU の幅 $d_{\text{ff}} = \text{round}(\frac{8}{3}d)$（11.1 節）を用いる場合を考える。

- 各アテンションサブ層は 4 つの $d \times d$ 射影行列（$\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v, \mathbf{W}_o$）を持ち、パラメータ数は $4d^2$ 個である。

- 各 SwiGLU FFN は 3 つの行列（$\mathbf{W}_g, \mathbf{W}_1 \in \mathbb{R}^{d \times \frac{8}{3}d}$ および $\mathbf{W}_2 \in \mathbb{R}^{\frac{8}{3}d \times d}$）を持ち、パラメータ数は $8d^2$ 個である。
- したがって、各ブロックは $12d^2$ 個の行列パラメータ（および $2d$ 個の RMSNorm ゲイン）を持ち、L 個のブロック全体では次式の個数の非埋め込み行列パラメータを持つ。

$$N \;=\; 12Ld^2 \tag{11.11}$$

一方、重み共有されたトークン埋め込みテーブルが Vd 個のパラメータを追加する。

```
gpt = d2l.GPT
```

非埋め込みパラメータ数が $12Ld^2$ に等しいこと（無視できる $O(Ld)$ 個の正規化のゲインおよびバイアスの範囲内で）を、6 ブロックのバイトレベル GPT と 12 ブロックの GPT-2 Small の両方で確認する。

```
def census(model):
    L, d = len(model.blks), model.num_hiddens
    emb = d2l.num_params(model.token_emb) + (
        d2l.num_params(model.pos_emb) if model.pos == 'learned' else 0)
    total = d2l.num_params(model)
    print(f'embedding {emb / 1e6:6.2f}M ({emb / total:5.1%}),  '
          f'blocks {(total - emb) / 1e6:6.2f}M '
          f'= {(total - emb) / (L * d ** 2):.2f} L d^2,  '
          f'total {total / 1e6:.2f}M')

census(gpt(256))
census(gpt(50257, num_hiddens=768, num_heads=12, num_blks=12,
           pos='learned', norm='layer', act='gelu', bias=True))
```

```
embedding   0.07M ( 1.4%),  blocks   4.72M = 12.01 L d^2,  total 4.79M
embedding  39.38M (31.6%),  blocks  85.06M = 12.02 L d^2,  total 124.44M
```

11.6.1.2 パラメータ・トークンあたり 6 FLOPs（$C \approx 6ND$）

なぜ 1 つの訓練トークンは、約 $6N$ FLOPs を $N = 12Ld^2$ 個の線形射影（linear projection）パラメータ全体で消費するのだろうか。

1. **順伝播（トークンあたり $2N$ FLOPs）：** トークンベクトル $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ に重み行列 $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ を掛けると、$d_{\text{in}} d_{\text{out}}$ 回の積和演算 $= 2P$ FLOPs が実行される（ここで $P = d_{\text{in}} d_{\text{out}}$ はこの行列のパラメータ数である）。全 N 個の行列パラメータと D トークンにわたって合計すると、順伝播の計算量は $2ND$ FLOPs になる。
2. **逆伝播（トークンあたり $4N$ FLOPs）：** $\mathbf{Y} = \mathbf{X}\mathbf{W}$ を通して誤差逆伝播法を実行するには、トークンあたり **2** 回のサイズ P の行列積が必要になる。すなわち $\nabla_{\mathbf{X}}\mathcal{L} = (\nabla_{\mathbf{Y}}\mathcal{L})\mathbf{W}^\top$（$2P$ FLOPs）と $\nabla_{\mathbf{W}}\mathcal{L} = \mathbf{X}^\top(\nabla_{\mathbf{Y}}\mathcal{L})$（$2P$ FLOPs）であり、合計で逆伝播の計算量は $4ND$ FLOPs となる。

順伝播と逆伝播を足し合わせると、訓練計算量の基本的な近似式が得られる (Kaplan et al. 2020)。

$$C \;\approx\; 6ND. \tag{11.12}$$

($QK^\top$ と AV のアテンションスコアのドット積をコンテキストウィンドウ長 T にわたって含めると、$12LTdD$ FLOPs が加わり、$6ND$ は $1 + \frac{T}{6d}$ 倍になる。この係数は 1 に近い値となる（$T \ll 6d$ が成り立つ限り）。因果アテンション行列のマスクされた半分をスキップする実装では、この補正項は半分になる。）

11.6.1.3 ハードウェアカウンタによる計算式の検証

Equation 11.12 は、フレームワークが計測する厳密な FLOP カウントと直接照らし合わせることで、`gpt(256, act='gelu')` の単一の訓練ステップ上で検証できる。

PyTorch の `FlopCounterMode` は、すべての演算子の FLOPs をカウントする。

```
model = gpt(256, act='gelu')
X = torch.randint(0, 256, (64, 128))
Y = torch.randint(0, 256, (64, 128))
N = sum(p.numel() for p in model.parameters() if p.ndim == 2)
with sdpa_kernel(SDPBackend.MATH):  # Attention as visible matrix products
    with FlopCounterMode(display=False) as fwd:
        loss = F.cross_entropy(model(X).flatten(0, 1), Y.flatten())
    with FlopCounterMode(display=False) as bwd:
        loss.backward()
f, b = fwd.get_total_flops(), bwd.get_total_flops()
print(f'forward {f:.3e} FLOPs, {f / (2 * N * X.numel()):.3f} x 2ND')
print(f'step    {f + b:.3e} FLOPs, {(f + b) / (6 * N * X.numel()):.3f} x 6ND')
print(f'backward / forward = {b / f:.2f}')
```

```
forward 8.483e+10 FLOPs, 1.082 x 2ND
step    2.545e+11 FLOPs, 1.082 x 6ND
backward / forward = 2.00
```

実測された逆伝播の計算量は順伝播の約 2 倍（2.0–2.1×）であり、$6ND$（重み共有出力ヘッドおよびアテンションスコア項 $12LTdD$ を考慮したもの）は厳密な演算子カウントと数パーセント以内で一致する。

11.6.2 WikiText-103 における小規模スケーリング実験

11.6.2.1 5 つのモデルサイズと 3 つのデータ予算

スケーリングの挙動を直接観察するため、**15 個のモデル**からなるグリッドを WikiText-103 上で訓練する。具体的には、非埋め込みパラメータ数が $N \in [0.3\text{M}, 14\text{M}]$ にわたる 5 種類のアーキテクチャ（`(d, L, H)` が `(96, 3, 4)` から `(384, 8, 8)` まで）と、3 種類のステップ予算 $s \in \{500, 1000, 2000\}$ ステップ（訓練バイト数 $D \in \{4.10\text{M}, 8.19\text{M}, 16.38\text{M}\}$ に相当）を掛け合わせる。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
print(f'{len(data.train) / 1e6:.0f}M training bytes, '
      f'{len(data.val) / 1e6:.2f}M validation bytes, '
      f'vocabulary of {len(data.vocab)}')
```

```
270M training bytes, 1.14M validation bytes, vocabulary of 256
```

コーパスを読み込んだら、全 $5 \times 3 = 15$ 通りの `(size, budget)` 設定を、幅に応じてスケーリングしたピーク学習率 $\eta = 2 \times 10^{-3} \cdot (256/d)$ で訓練し、各モデルの非埋め込みパラメータ数 `Ns`

と検証交差エントロピー val を記録する。

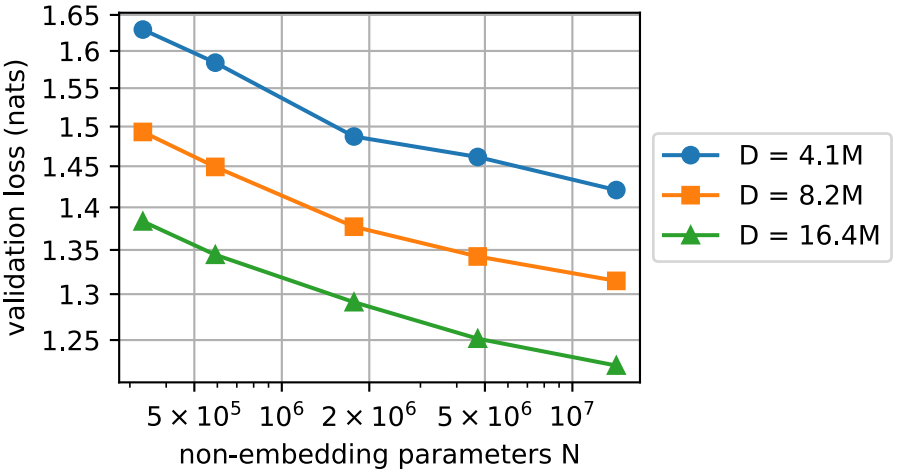
```
sizes = ((96, 3, 4), (128, 3, 4), (192, 4, 6), (256, 6, 8), (384, 8, 8))
budgets = (500, 1000, 2000)  # Steps of 64 windows of 128 bytes
Ns, train, val = [], [], {num_steps: [] for num_steps in budgets}
for d, num_blks, num_heads in sizes:
    for num_steps in budgets:
        model = gpt(len(data.vocab), num_hiddens=d, num_heads=num_heads,
                    num_blks=num_blks)
        optimizer = d2l.LMOptimizer(model, lr=2e-3 * 256 / d,
                                    num_steps=num_steps,
                                    warmup=max(100, num_steps // 10))
        losses = d2l.train_steps(model, data, optimizer, num_steps)
        val[num_steps].append(
            math.log(d2l.lm_perplexity(model, data.val_dataloader())))
    # The size of the models just trained, and the training loss of the
    # longest of their runs
    Ns.append(d2l.num_params(model) - d2l.num_params(model.token_emb))
    train.append(sum(losses[-50:]) / 50)
    print(f'd = {d:3d}, {num_blks} blocks, N = {Ns[-1] / 1e6:5.2f}M: '
          'validation ' + ' / '.join(f'{val[s][-1]:.2f}' for s in budgets)
          + f', training {train[-1]:.2f}')
```

```
d =  96, 3 blocks, N =  0.33M: validation 1.63 / 1.49 / 1.38, training 1.41
d = 128, 3 blocks, N =  0.59M: validation 1.58 / 1.45 / 1.34, training 1.37
d = 192, 4 blocks, N =  1.77M: validation 1.49 / 1.38 / 1.29, training 1.31
d = 256, 6 blocks, N =  4.72M: validation 1.46 / 1.34 / 1.25, training 1.27
d = 384, 8 blocks, N = 14.16M: validation 1.42 / 1.31 / 1.22, training 1.24
```

11.6.2.2 モデルサイズ、データ予算、および計算量に対する損失

検証交差エントロピーを、非埋め込みパラメータ数 N の対数軸に対してトークン予算 D ごとにプロットし、さらに総訓練計算量 $C = 6ND$ に対してもプロットする。

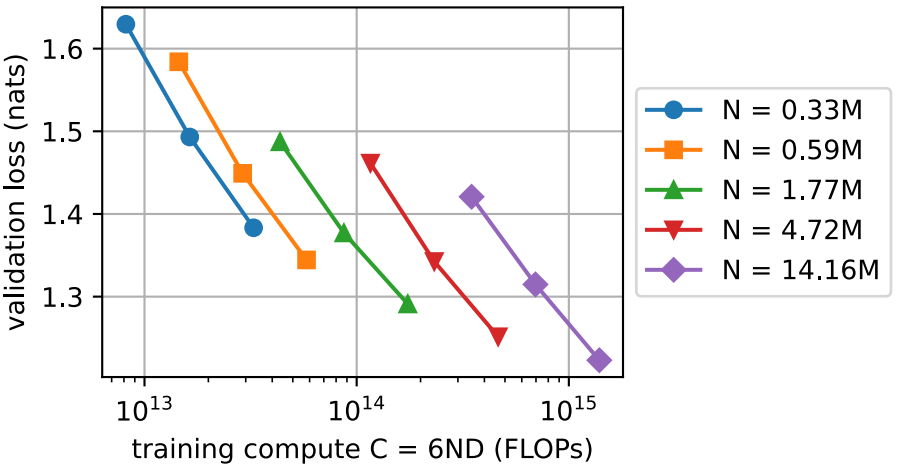
```
tokens = {s: s * 64 * 128 for s in budgets}  # Tokens read in s steps
d2l.plot(Ns, [val[s] for s in budgets], 'non-embedding parameters N',
         'validation loss (nats)',
         legend=[f'D = {tokens[s] / 1e6:.1f}M' for s in budgets],
         xscale='log', yscale='log', fmts=('C0o-', 'C1s-', 'C2^-'),
         legend_loc='outside')
```



同じ 15 個の検証損失を総訓練計算量 $C = 6ND$ に対して再プロットすると（あわせて計算量が近い 3 つの実行、すなわち $C \approx 2\text{–}3 \times 10^{14}$ FLOPs で互いに $2\times$ 以内に収まるものを出力すると）、FLOP 予算が増えるにつれて計算量最適（compute-optimal）なモデルサイズが上方へシフトする様子がわかる。

```
d2l.plot([[6 * N * tokens[s] for s in budgets] for N in Ns],
         [[val[s][i] for s in budgets] for i in range(len(Ns))],
         'training compute C = 6ND (FLOPs)', 'validation loss (nats)',
         legend=[f'N = {N / 1e6:.2f}M' for N in Ns], xscale='log',
         fmts=('C0o-', 'C1s-', 'C2^-', 'C3v-', 'C4D-'), legend_loc='outside')
for i, s in ((2, 2000), (3, 1000), (4, 500)):  # Larger N, fewer tokens
    print(f'N = {Ns[i] / 1e6:5.2f}M, D = {tokens[s] / 1e6:4.1f}M tokens: '
          f'C = {6 * Ns[i] * tokens[s]:.1e} FLOPs, '
          f'validation {val[s][i]:.2f}')
```

```
N =  1.77M, D = 16.4M tokens: C = 1.7e+14 FLOPs, validation 1.29
N =  4.72M, D =  8.2M tokens: C = 2.3e+14 FLOPs, validation 1.34
N = 14.16M, D =  4.1M tokens: C = 3.5e+14 FLOPs, validation 1.42
```



2 つのパターンが際立っている。

1. 最大のトークン予算（2,000 ステップ、$D = 16.4$M バイト）では、検証損失は小型・中型モデルにわたって $\log N$ に対してほぼ線形に減少したのち、最大のモデル（$N = 14.16$M）で横ばいに曲がる。そこでは 2,000 ステップ（パラメータあたり $D/N \approx 1.2$ バイト）であっても、14M 個のパラメータに対してはいまだデータ不足の状態にある。
2. 全 15 回の実行を総訓練 FLOPs $C = 6ND$ に対してプロットすると、その下側包絡線（lower envelope）が**パレート最適な計算フロンティア**（Pareto-optimal compute frontier）を描く。計算量が小さい領域では小型モデルが優位に立ち、計算量が大きい領域では十分なステップ数が与えられた時点で大型モデルが逆転する。

11.6.3 計算量最適な配分（Chinchilla スケーリング）

11.6.3.1 Chinchilla のパラメトリック法則

Hoffmann et al. (2022)（および Besiroglu et al. (2024) による再フィット）は、検証損失曲面 $\mathcal{L}(N, D)$ を、削減不能なテキストエントロピーの下限 L_∞ と、N および D に関する 2 つのべき乗則ペナルティの和としてモデル化する。

$$\mathcal{L}(N, D) \;=\; L_\infty \;+\; \frac{A}{N^{\alpha}} \;+\; \frac{B}{D^{\beta}}. \tag{11.13}$$

11.6.3.2 計算量最適な分割（$N^*(C)$ と $D^*(C)$）

Equation 11.13 を、N と D に関して制約 $C = 6ND$ のもとで最小化すると（$D = C/(6N)$ を代入して $\partial\mathcal{L}/\partial N = 0$ とおくと）、最適なモデルサイズ $N^*(C)$ と最適なトークン予算 $D^*(C)$ の両方についてべき乗則スケーリングが得られる。

$$N^*(C) \;=\; G\left(\frac{C}{6}\right)^{a}, \qquad D^*(C) \;=\; G^{-1}\left(\frac{C}{6}\right)^{b}, \tag{11.14}$$

ここで、各定数は次のとおりである。

$$\begin{aligned} a \;&=\; \frac{\beta}{\alpha+\beta}, \qquad b \;=\; \frac{\alpha}{\alpha+\beta}, \\ G \;&=\; \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta}}. \end{aligned} \tag{11.15}$$

$\alpha \approx \beta \approx 0.34$–$0.37$ であるため、指数は $a \approx b \approx 0.5$ を満たす。すなわち、訓練計算量 C が $100\times$ に増えると、モデルサイズ N^* と訓練トークン数 D^* はともに $\sim 10\times$ に増え、パラメータあたり約 $D^*/N^* \approx 20$ トークンという計算量最適な比率はほぼ一定に保たれる。

これは、この問いに対する最初の答えではなかった。その 2 年前、Kaplan et al. (2020) は同種のべき乗則を当てはめ、追加の計算量の大部分はモデルに投じるべきだと結論づけていた。すなわち $N^* \propto C^{0.73}$ であり、データには $D^* \propto C^{0.27}$ しか残らない（Figure 11.13）。この 2 つの研究の食い違いは、主に学習率スケジュール（learning rate schedule）に起因する。Kaplan らはすべての実行に 1 つのスケジュールを使い、より小さなトークン予算の損失を、まだ学習率が減衰していない中間チェックポイントから読み取っていた。これでは少ないトークンでの訓練が実際より悪く見えるため、データの効果が実際より小さく見積もられてしまう。また、扱ったモデルも小さく、高々 1.5B パラメータであった。Hoffmann らは最大 16B パラメータのモデルを訓

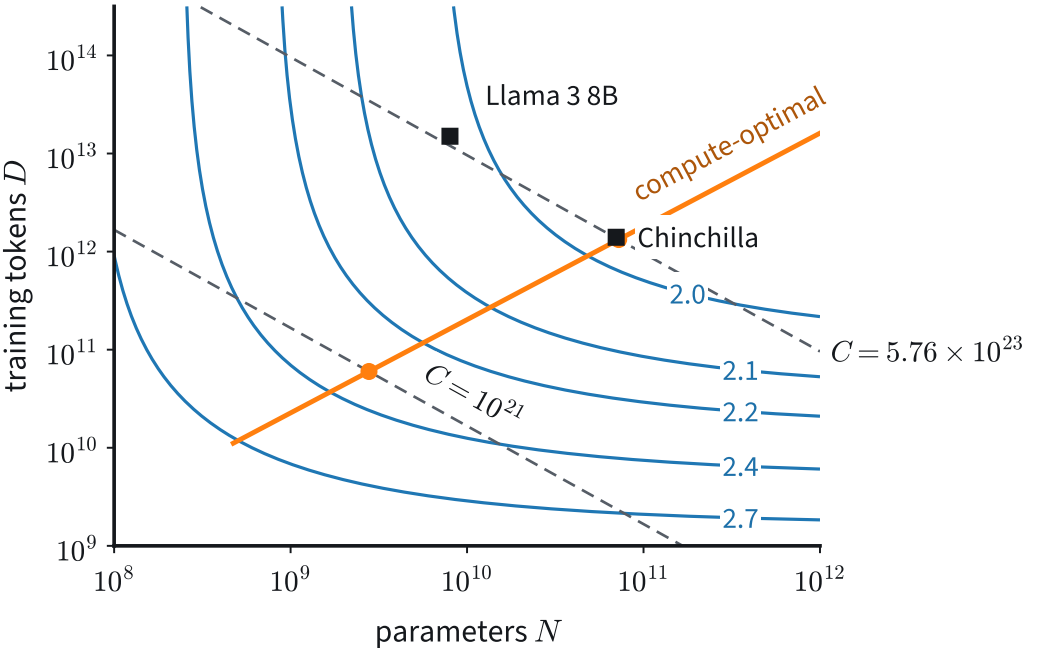


Figure 11.12: 後述する Besiroglu ら（2024）の再フィット定数を用いた、パラメータ数と訓練トークン数にわたる Chinchilla 則の損失等高線。破線：等しい計算量 $C = 6ND$ の予算線、点：各予算における計算量最適モデル、オレンジ色の実線：計算量最適経路。Chinchilla を示す四角は自身の予算の最適点上に位置し、その点を覆っている。一方、Chinchilla に近い予算で訓練された Llama 3 8B は、最適サイズの 10 分の 1 の位置にあり、最適経路から大きく外れている。

練し、コサインスケジュールを各実行の長さに合わせた。前述のグリッド実験でもこれと同じ方法をとっており、15 回の実行のそれぞれが固有のコサインホライズンを持つのはそのためである（そうしない場合に何が起きるかは、演習問題 3 で計測する）。

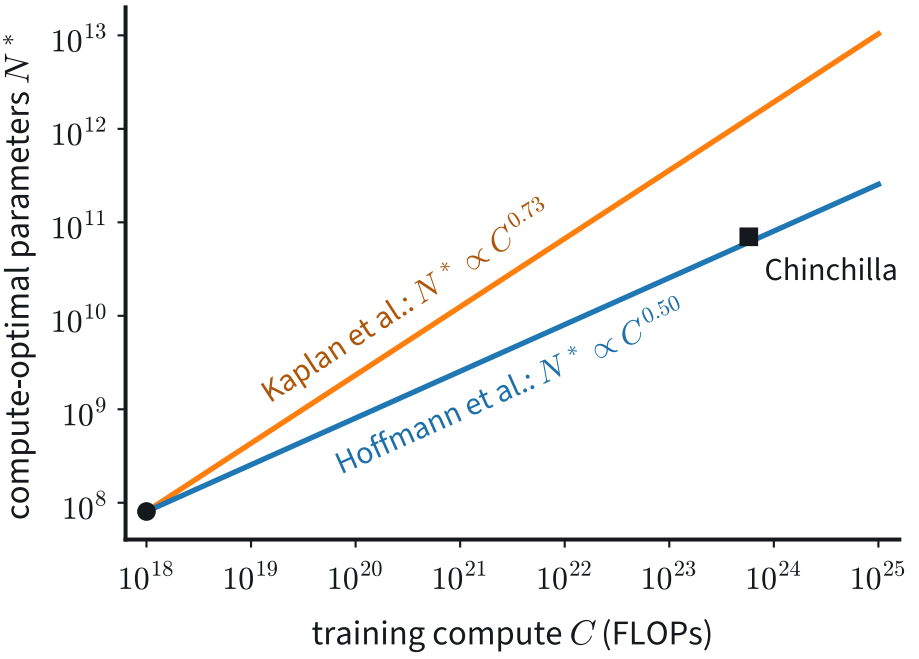


Figure 11.13: 公表された 2 つの指数、すなわち 0.73（Kaplan ら）と 0.5（Hoffmann ら）のもとでの訓練計算量に対する計算量最適モデルサイズ。いずれの直線も 10^{18} FLOPs において同じモデルを通るように引かれており、傾きのみが各論文に由来する。予算を 100× に増やすと、第 1 の指数では 29× 大きいモデルが、第 2 の指数では 10× 大きいモデルが求められる。四角は Chinchilla を示す。

以下の関数 `optimum` は、Equation 11.14 を Chinchilla の訓練予算 $C = 5.76 \times 10^{23}$ FLOPs において、Hoffmann らが公表した定数と Besiroglu らによる再フィット定数のそれぞれについて評価する。

```
fits = {'Hoffmann et al.': (1.69, 406.4, 410.7, 0.34, 0.28),
        'Besiroglu et al.': (1.82, 482.0, 2085.4, 0.348, 0.366)}

def optimum(C, L_inf, A, B, alpha, beta):
    G = (alpha * A / (beta * B)) ** (1 / (alpha + beta))
    N = G * (C / 6) ** (beta / (alpha + beta))
    return N, C / (6 * N)

for name, fit in fits.items():
    N, D = optimum(5.76e23, *fit)
    print(f'{name:17s} N* = {N / 1e9:3.0f}B, D* = {D / 1e12:.2f}T, '
          f'{D / N:3.0f} tokens per parameter')
```

```
Hoffmann et al.   N* =  32B, D* = 2.98T,  93 tokens per parameter
Besiroglu et al.  N* =  72B, D* = 1.33T,  18 tokens per parameter
```

11.6.3.3 推論を見据えた Chinchilla 最適を超える訓練

なぜ Llama 3 8B（Figure 11.12）は、8B パラメータのモデルを 15T トークン（パラメータあたり $D/N \approx 1{,}875$ トークンで、Chinchilla の比率のほぼ 100×）で訓練する道を選び、Chinchilla 最適な 70B モデルを 1.4T トークン（同じ事前学習計算量 $C \approx 7 \times 10^{23}$ FLOPs）で訓練しなかったのだろうか。

それは、Chinchilla が**事前学習の FLOPs $6ND$ のみ**を最小化し、デプロイ後のモデルのサービングに費やされる $2ND_{\text{inf}}$ の推論 FLOPs や KV キャッシュメモリを無視しているからである (Sardana et al. 2024)。一定の損失をもつ等高線 $\mathcal{L}(N, D) = \mathcal{L}_{\text{target}}$ に沿って移動してみると、8B モデルを 15T トークンで過剰訓練（over-training）すれば、Chinchilla 配分の 30B–40B モデルに近い損失に到達しつつ、その後に生成するすべての推論トークンを 4–5× 安くサービングできることがわかる。

```
L_inf, A, B, alpha, beta = fits['Besiroglu et al.']
loss = lambda N, D: L_inf + A / N ** alpha + B / D ** beta
reported = {'Chinchilla': (70e9, 1.4e12), 'Llama 3 8B': (8e9, 15e12),
            'Llama 3 405B': (405e9, 15.6e12), 'OLMo 2 7B': (7e9, 4.05e12),
            'Gemma 3 27B': (27e9, 14e12)}
for name, (N, D) in reported.items():
    N_opt, D_opt = optimum(6 * N * D, L_inf, A, B, alpha, beta)
    print(f'{name:12s} {D / N:5.0f} tokens per parameter, '
          f'{N / N_opt:4.2f} x optimal size, '
          f'loss +{loss(N, D) - loss(N_opt, D_opt):.3f}')
```

```
Chinchilla      20 tokens per parameter, 0.96 x optimal size, loss +0.000
Llama 3 8B    1875 tokens per parameter, 0.10 x optimal size, loss +0.054
Llama 3 405B    39 tokens per parameter, 0.65 x optimal size, loss +0.001
OLMo 2 7B      579 tokens per parameter, 0.18 x optimal size, loss +0.037
Gemma 3 27B    519 tokens per parameter, 0.18 x optimal size, loss +0.023
```

11.6.4 まとめ

L 層、幅 d で SwiGLU（$d_{\text{ff}} \approx \frac{8}{3}d$）を備えた密な Transformer は、$N \approx 12Ld^2$ 個の非埋め込み行列パラメータ（アテンションに $4Ld^2$、FFN に $8Ld^2$）を含み、$C \approx 6ND$ の訓練 FLOPs を D トークンにわたって消費する（順伝播で $2ND$、逆伝播で $4ND$、Equation 11.12）。**Chinchilla スケーリング則** $\mathcal{L}(N, D) = L_\infty + AN^{-\alpha} + BD^{-\beta}$（Equation 11.13）のもとでは、固定された事前学習計算量 $C = 6ND$ で検証損失を最小化すると、パラメータ数とトークン数は指数 $a = \beta/(\alpha+\beta) \approx 0.5$ および $b = \alpha/(\alpha+\beta) \approx 0.5$（Equation 11.14、Equation 11.15）に従ってスケーリングし、パラメータあたり約 $D^*/N^* \approx 20$ トークンという比率が得られる。**生涯推論計算量**（lifetime inference compute、$2ND_{\text{inf}}$）とサービング時のレイテンシを考慮すると、総計算量の最適点は等損失等高線に沿って、パラメータあたり数百〜数千トークンで過剰訓練されたより小さなモデル N の側へとシフトする。

11.6.5 演習問題

1. **アテンションスコアの FLOP 補正**（$1 + T/(6d)$）。本文の Equation 11.12 は、アテンションスコアと値の積 $\mathbf{QK}^\top$ および $\mathbf{AV}$、ならびに出力ヘッド $\mathbf{H}^{(L)}\mathbf{E}^\top$ を省略している。
    1. L ブロック、幅 d、ウィンドウ長 T のモデルにおいて、順伝播のアテンションのドット積がウィンドウあたり $4LT^2d$ FLOPs（トークンあたり $4LTd$ FLOPs）を実行し、順伝播と逆伝播を合わせたアテンションのドット積がトークンあたり $12LTd$ FLOPs

（因果対称性が FlashAttention によって利用される場合は $6LTd$）を実行することを示せ。

2. $6LTd$ と $6N = 72Ld^2$ の比をとることで、因果アテンションが加える相対的な FLOP 比率が $T/(12d)$（因果タイルスキップなしでは $T/(6d)$）となることを示せ。コンテキスト長 T^* がいくつのときに、アテンションの計算量は Llama 3 8B（$d = 4096$）の線形射影の FLOPs と一致するか。

2. **スケーリンググリッドにおける総パラメータと非埋め込みパラメータ。**本節の 5 サイズにわたる WikiText-103 スイープの各モデルではバイト語彙（$V = 256$）を用いたため、Vd は高々約 7%（対 $12Ld^2$ 比）にとどまっていた。`census` を同じ 5 つの `(d, L, H)` バックボーンに対して、代わりに BPE 語彙 $V = 128{,}256$ トークン（Llama 3）を用いた場合について実行せよ。最小モデル（`d=96, L=3`）と最大モデル（`d=384, L=8`）では、埋め込みテーブルが総パラメータのどれだけの割合を占めるか。また、なぜ Kaplan et al. (2020) はべき乗則を $N_{\text{non-emb}} = 12Ld^2$ に対して当てはめ、N_{total} に対しては当てはめなかったのか。

3. **コサイン減衰のホライズンと固定ステップホライズン。**本節の 15 実行のスケーリング実験では、各予算 $s \in \{500, 1000, 2000\}$ が固有のコサインスケジュールを実行し、学習率をステップ s でゼロまで減衰させた（スケジュールを実行の長さに合わせることは、Hoffmann et al. (2022) による Kaplan et al. (2020) への決定的な修正であった）。3 つの予算を中型モデル `(192, 4, 6)` について再実行せよ。ただし、単一の固定コサインホライズン `num_steps=2000` を `LMOptimizer` 内で用い、ステップ 500、1000、2000 における検証損失を単に読み取るものとする。$s = 500$ および $s = 1000$ で学習率を減衰させないことによって、短い実行の損失はどれだけ過大評価されるか。

4. **Chinchilla 則の 15 点グリッドへの当てはめ。**`scipy.optimize.minimize`（またはグリッド探索 + 非線形最小二乗法）を用いて、パラメータ $(L_\infty, A, B, \alpha, \beta)$（Equation 11.13）を、15 個の `(N, D, val_loss)` 点（本節の WikiText-103 スイープで計測したもの）に当てはめよ。スケーリング指数 $a = \beta/(\alpha+\beta)$ と $b = \alpha/(\alpha+\beta)$ は、このバイトレベルの 15 点グリッドからどのような値として得られるか。

5. **推論需要 D_{inf} を伴う生涯計算量の最適化。**サイズ N のモデルを D トークンで訓練し、D_{inf} 個の推論トークンをサービングするためにデプロイすると、消費される総生涯 FLOPs は $C_{\text{tot}}(N, D) = 6ND + 2ND_{\text{inf}}$ となる。

   1. 制約 $\mathcal{L}(N, D) = L_\infty + AN^{-\alpha} + BD^{-\beta} = \mathcal{L}_{\text{target}}$ に対してラグランジュ乗数法を用い、生涯計算量の最小点 (N^*, D^*) において限界比が次の関係を満たすことを示せ。

$$\frac{\beta B (D^*)^{-\beta}}{\alpha A (N^*)^{-\alpha}} = \frac{6N^*D^*}{6N^*D^* + 2N^*D_{\text{inf}}} = \frac{1}{1 + \frac{D_{\text{inf}}}{3D^*}}.$$

   2. $D_{\text{inf}} = 0$ のときにこれが Chinchilla の均衡式 $\alpha A (N^*)^{-\alpha} = \beta B (D^*)^{-\beta}$ に帰着することを確かめよ。さらに (N^*, D^*) を、$\mathcal{L}_{\text{target}} = \mathcal{L}(70\text{B}, 1.4\text{T})$ かつ $D_{\text{inf}} \in \{10\text{T}, 100\text{T}, 500\text{T}\}$ トークンの各場合について数値的に求めよ。

6. **活性化チェックポイントと $8ND$ 則。**完全なブロック単位の活性化チェックポイントは、各ブロックへの入力 $\mathbf{X}^{(\ell-1)}$ のみを順伝播（$2ND$ FLOPs）の間に保存し、誤差逆伝播法において逆方向の行列積（$4ND$ FLOPs）を計算する前に各ブロックの順伝播を一度再計算する。活性化チェックポイントによって事前学習計算量の式が $C \approx 6ND$ から $C \approx 8ND$ に変わることを示せ。また、6 を 8 に置き換えること（Equation 11.14）によって、$N^*(C)$ と $D^*(C)$ が固定された FLOP 予算 C のもとでどのようにスケーリングするかを求めよ。

11.7 混合エキスパート（MoE）

密な Transformer（11.6 節）では、すべてのトークンがすべてのパラメータを使用する。そのため、モデル容量 $N \approx 12Ld^2$ を 2 倍にすると、訓練コスト（トークンあたり $6N$ FLOPs）も推論の計算量（トークンあたり $2N$ FLOPs）も 2 倍になる。モデルの保持パラメータ容量 N_{total} を拡大しつつ、トークンごとに実行されるアクティブパラメータ数 N_{active} を増やさずに済ませることはできるだろうか。

11.1 節で見たように、フィードフォワードネットワークは密なブロックのパラメータ数の 3 分の 2（$8d^2$、ブロック全体では $12d^2$）を占め、各トークンベクトル $\mathbf{x} \in \mathbb{R}^d$ に対して独立に**位置ごと**（position-wise）に作用する。**スパース混合エキスパート**（sparse mixture-of-experts、MoE）層 (Jacobs et al. 1991; Shazeer et al. 2017; Lepikhin et al. 2021; Fedus et al. 2022) は、この独立性を活用する（Figure 11.14）。すなわち、ブロック内の単一の FFN を並列な E 個の FFN **エキスパート**群 $\{\text{FFN}_1, \dots, \text{FFN}_E\}$ に置き換え、軽量な線形**ルータ** $\mathbf{W}_r \in \mathbb{R}^{d \times E}$ を用いて各トークン $\mathbf{x}$ を上位 k 個のエキスパートのみにディスパッチする（$1 \le k \ll E$）。

11.7.1 条件付き計算：保持パラメータとアクティブパラメータ

L 層の MoE Transformer において、各ブロックがアテンションのパラメータ P_{attn}、ルーティングされる E 個の SwiGLU エキスパート（中間幅 w、エキスパートあたり $P_{\text{exp}} = 3dw$ 個のパラメータ）、および E_{shared} 個の常時アクティブな共有エキスパート（shared expert）を持つとする (Dai et al. 2024)。このとき、全 $E + E_{\text{shared}}$ 個のエキスパートをメモリに保持する必要があるが（N_{total}）、各トークンが実行するのは $k + E_{\text{shared}}$ 個のエキスパートのみである（N_{active}）。

```python
def moe_accounting(name, d, width, num_experts, num_active, num_layers,
                   shared=0, unit=1e9):
    """Expert parameters stored, and expert and router parameters active
    per token, in a stack of SwiGLU expert layers."""
    per_expert = 3 * d * width          # SwiGLU: three d x width matrices
    store = num_layers * (num_experts + shared) * per_expert
    active = num_layers * ((num_active + shared) * per_expert
                           + d * num_experts)      # + the router
    tag = 'B' if unit == 1e9 else 'M'
    print(f'{name:>12}: {store/unit:5.3g}{tag} in experts, '
          f'{active/unit:5.3g}{tag} active per token, '
          f'ratio {store/active:4.1f}x')
    return store, active

moe_accounting('ours', 128, 341, 8, 1, 2, unit=1e6)
moe_accounting('DeepSeek-V3', 7168, 2048, 256, 8, 58, shared=1)
store, active = moe_accounting('Mixtral 8x7B', 4096, 14336, 8, 2, 32)
attn = 32 * (2 * 4096 * 4096 + 2 * 4096 * 1024)  # GQA: 32 Q, 8 KV heads
emb = 2 * 32000 * 4096                           # embedding + output head
print(f'{"":>12}  with attention and embeddings: '
      f'{(store + attn + emb)/1e9:.1f}B stored, '
      f'{(active + attn + emb)/1e9:.1f}B active')
```

```
        ours:   2.1M in experts, 0.264M active per token, ratio  7.9x
 DeepSeek-V3:   656B in experts,  23.1B active per token, ratio 28.4x
Mixtral 8x7B:  45.1B in experts,  11.3B active per token, ratio  4.0x
              with attention and embeddings: 46.7B stored, 12.9B active
```

N_{total} と N_{active} がそれぞれ異なるシステム資源を規定している点に注目しよう。

- **トークンあたり FLOPs（訓練とプレフィル）**は $N_{\textbf{active}}$ に比例してスケールする（$C_{\text{train}} \approx 6N_{\text{active}}D$）。
- **VRAM 容量**は $N_{\textbf{total}}$ に比例してスケールする。全 E 個のエキスパートをデバイスメモリ上に常駐させる必要があるためである。
- **バッチデコードの帯域幅：**バッチサイズ $B = 1$ では、1 つのトークンが HBM からストリーミングするのは k 個のエキスパートのみである（N_{active} 個分の重みバイト数）。しかし、B トークンのバッチ全体で見ると、異なるトークンが異なるエキスパートを活性化させる。$B \gg E/k$ になると、ほぼ全 E 個のエキスパートがバッチ内の少なくとも 1 つのトークンによって参照されるため、各デコードステップでは HBM からほぼ $N_{\textbf{total}}$ 個分の重みバイト数をストリーミングする一方で、実行する演算は $2BN_{\text{active}}$ FLOPs にとどまる。

11.7.2 混合エキスパート層

11.7.2.1 Top-k ルーティング

各トークン $\mathbf{x} \in \mathbb{R}^d$ に対し、ルータは確率 $\mathbf{p}(\mathbf{x}) = \text{softmax}(\mathbf{x}\mathbf{W}_r) \in \Delta^{E-1}$ を計算し、集合 $T(\mathbf{x}) \subset \{1, \dots, E\}$ としてスコア上位 k 個のエキスパートを選択する。続いて、その生の確率をそのまま重み $g_e(\mathbf{x}) = p_e(\mathbf{x})$ として保持し（$T(\mathbf{x})$ 上で再正規化すると Top-1 ルータの勾配が消失してしまう。演習問題 1）、次の重み付き和を計算する。

$$\text{MoE}(\mathbf{x}) \;=\; \sum_{e \in T(\mathbf{x})} g_e(\mathbf{x})\, \text{FFN}_e(\mathbf{x}). \tag{11.16}$$

11.7.2.2 固定容量エキスパートバッファへのトークンのディスパッチ

コンパイル実行や分散訓練では、各エキスパート $e \in \{1, \dots, E\}$ はあらかじめ容量 $S = \lceil \gamma \cdot nk/E \rceil$ スロットの固定バッファを、n トークンのバッチに対して確保する。ここで、nk/E は完全に均等化されたときのエキスパートあたりの負荷であり、$\gamma \geq 1$ は**容量係数**（capacity factor）である（$\gamma = E$ とすれば $S = nk$ となり、ドロップは無効化される）。S 個を超えるトークン・エキスパート割り当てがエキスパート e に集中した場合、超過分の割り当ては**ドロップ**される（Equation 11.16 への寄与が 0 になり、該当トークンは外側のブロックの残差スキップ接続をそのまま通過する）。Figure 11.15 は、6 個のトークンがこのディスパッチ処理を通過する様子を追ったものである。

`MoELayer` は、ルータ、S スロットのバッファへのディスパッチ、およびドロップ規則を実装する。

```
class MoELayer(nn.Module):
    """Token-choice top-k mixture of SwiGLU d2l.FeedForward experts, each
    with a buffer of S = capacity_factor * n k / E slots for n tokens."""
    def __init__(self, num_hiddens, num_experts, num_active,
                 capacity_factor=1.25):
        super().__init__()
        self.num_experts, self.num_active = num_experts, num_active
        self.capacity_factor = capacity_factor
        self.router = nn.Linear(num_hiddens, num_experts, bias=False)
        self.experts = nn.ModuleList([d2l.FeedForward(num_hiddens)
                                      for _ in range(num_experts)])
        # Added to the probabilities for the selection only; zero until a
        # balancing method moves it
```

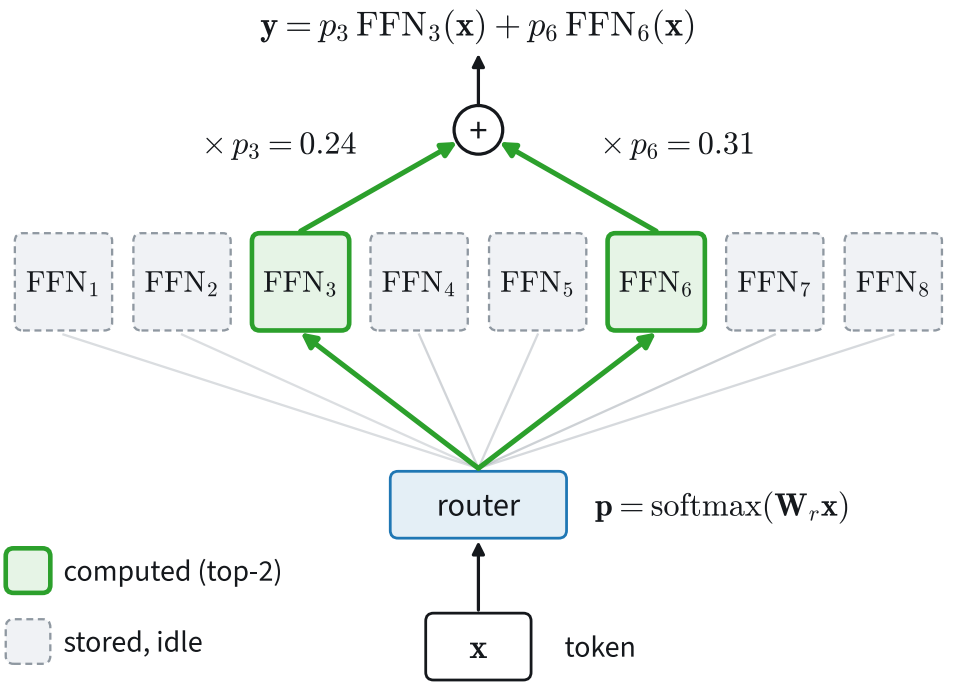


Figure 11.14: 8 個のエキスパートを持ち、トークンごとにそのうち 2 個がアクティブになる混合エキスパート層。ルータがすべてのエキスパートをスコアリングする。最も確率の高い 2 個のエキスパートが実行され、それぞれの出力が確率で重み付けされて加算される。残りの 6 個のエキスパートは保持されているが、このトークンに対しては何も処理しない。

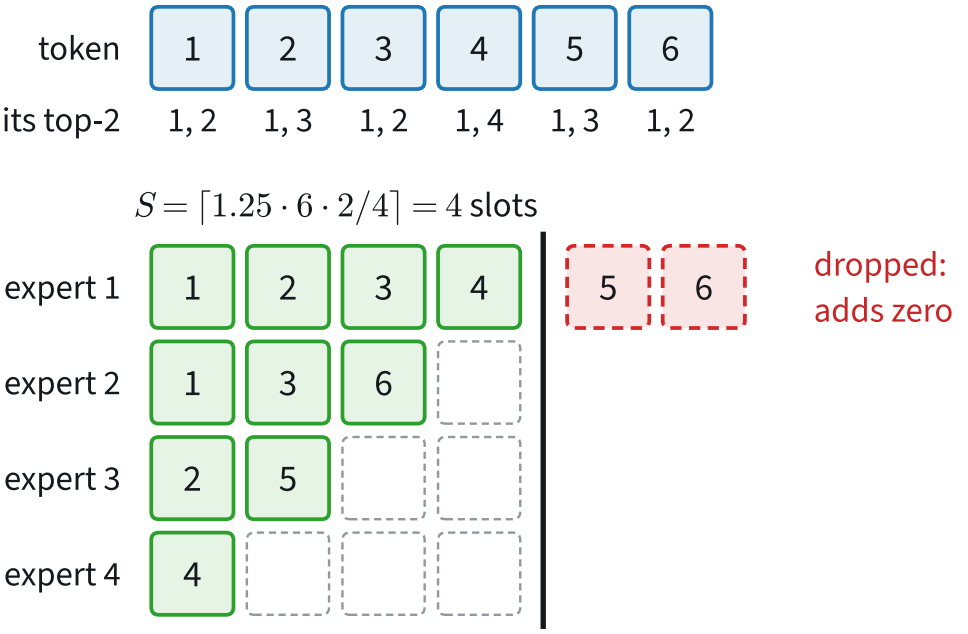


Figure 11.15: 6 個のトークンをそれぞれ上位 2 個（Top-2）ずつ 4 個のエキスパートにディスパッチする例（容量係数 $\gamma = 1.25$ により、各エキスパートのスロット数は $S = 4$）。割り当てはトークン順にエキスパートごとのキューに入る。エキスパート 1 への 5 番目と 6 番目の割り当ては S スロットを超えるためドロップされ、トークン 5 と 6 は第 2 候補のエキスパートのみで処理される。すべての割り当てがドロップされたトークンには残差のみが残る。

```
        self.register_buffer('expert_bias', torch.zeros(num_experts))
        # Assignments per expert, dropped or not, and the dropped ones,
        # counted by record() until reset_stats()
        self.register_buffer('load', torch.zeros(num_experts))
        self.register_buffer('dropped', torch.zeros(()))

    def forward(self, X):
        x = X.reshape(-1, X.shape[-1])                # (n, d): the tokens
        (n, d), E, k = x.shape, self.num_experts, self.num_active
        S = math.ceil(self.capacity_factor * n * k / E)  # slots per expert
        P = F.softmax(self.router(x), -1)             # (n, E)
        idx = (P + self.expert_bias).topk(k).indices  # (n, k)
        e = idx.flatten()                             # expert of each pair
        onehot = F.one_hot(e, E)                      # (n k, E)
        rank = (onehot.cumsum(0) * onehot).sum(1) - 1  # place in its queue
        slot = torch.where(rank < S, e * S + rank, E * S)  # E S: dropped
        buf = x.new_zeros(E * S + 1, d)               # + an overflow row
        buf[slot] = x.repeat_interleave(k, 0)
        Y = self.run_experts(buf[:-1].reshape(E, S, d)).reshape(E * S, d)
        Y = torch.cat([Y, Y.new_zeros(1, d)])[slot]   # dropped: zeros
        weights = P.gather(-1, idx)                   # (n, k): the p_i
        out = (weights[..., None] * Y.reshape(n, k, d)).sum(1)
        self.record(onehot.sum(0), (rank >= S).sum(), P)
        return out.reshape(X.shape)

    def run_experts(self, H):
        """Each expert's SwiGLU on its own buffer: (E, S, d) -> (E, S, d)."""
        W = lambda name: torch.stack(  # (E, d_in, d_out): one per expert
            [getattr(e, name).weight for e in self.experts]).mT
        return (F.silu(H @ W('W_g')) * (H @ W('W_1'))) @ W('W_2')

    def record(self, counts, num_dropped, P):
        """Add a batch's assignments per expert and drops to the totals,
        and keep its balancing loss, a measure of imbalance."""
        self.load += counts
        self.dropped += num_dropped
        f = counts / counts.sum()                     # this batch's shares
        self.aux_loss = self.num_experts * (f * P.mean(0)).sum()

    def reset_stats(self):
        self.load.zero_()
        self.dropped.zero_()
```

2 つの小さなユーティリティ関数 `moe_layers(model)` と `active_params(model)` は、ネットワーク内のすべての `MoELayer` サブモジュールを収集し、単一トークンの順伝播に関与するパラメータ数（エキスパート群の k/E とすべての共有パラメータの合計）を集計する。

```
def moe_layers(model):
    """The MoELayer modules of a model."""
    return [m for m in model.modules() if isinstance(m, MoELayer)]

def active_params(model):
```

```
    """Parameters used per token: all but the unselected experts."""
    return d2l.num_params(model) - sum(
        d2l.num_params(l.experts) * (l.num_experts - l.num_active)
        // l.num_experts for l in moe_layers(model))
```

d2l.TransformerBlock（11.1 節）は ffn_factory フックを受け取るため、密な FFN を 8 エキスパート・Top-2 の MoELayer に置き換えるにはラムダ式を 1 つ渡すだけでよい。

```
blk = d2l.TransformerBlock(256, num_heads=8,
                           ffn_factory=lambda: MoELayer(256, 8, 2))
X = torch.randn(4, 100, 256)
d2l.check_shape(blk(X), X.shape)  # The expert layer fits into the block
moe = MoELayer(256, num_experts=8, num_active=2)
print(f'{d2l.num_params(moe)} parameters stored, '
      f'{active_params(moe)} active per token')
```

```
4198400 parameters stored, 1051136 active per token
```

11.7.3 ルーティング崩壊と負荷分散

11.7.3.1 負荷の偏りと正のフィードバック

初期化の時点でさえ、ランダムなルータの重みによって、一部のエキスパートには他よりも多くのトークンが送られる。負荷分散（load balancing）の仕組みなしに訓練を進めると、**正のフィードバックループ**（positive-feedback loop）がこの非対称性を増幅し、**ルーティング崩壊**（routing collapse）を引き起こす (Shazeer et al. 2017)。すなわち、ステップ 1 でより多くのトークンを受け取ったエキスパートほど大きな勾配更新を受け、ステップ 2 で言語モデリング損失をより速く減少させ、ステップ 3 で $\nabla_{\mathbf{w}_r}\mathcal{L}$ から報酬を受けてルータのロジットがさらに高くなる。このまま放置すると、ごく少数のエキスパートが上位 k 個のスロットの大半を独占し、残りのエキスパートにはほとんどトークンが回らず何も学習できない状態に陥りうる。

11.7.3.2 2 つの解決策：補助負荷分散損失と損失不要バイアスコントローラ

1. **Switch 補助負荷分散損失** (Fedus et al. 2022)：$f_e = \frac{1}{nk}\sum_{t=1}^{n}\mathbb{1}[e \in T(\mathbf{x}_t)]$ をエキスパート e にディスパッチされたルーティングスロットの経験的な割合（n トークンのバッチ内）とし、$\bar{p}_e = \frac{1}{n}\sum_{t=1}^{n} p_e(\mathbf{x}_t)$ をエキスパート e に割り当てられたルータ確率の平均とする。訓練損失に次の補助損失による項 $\alpha\mathcal{L}_{\text{aux}}$ を加える。

$$\mathcal{L}_{\text{aux}} \;=\; E\sum_{e=1}^{E} f_e\,\bar{p}_e, \tag{11.17}$$

   この項は、高い確率 $\bar{p}_e$ が負荷の集中したエキスパート（f_e が大きいもの）に与えられることにペナルティを課し、最小値 $\mathcal{L}_{\text{aux}} = 1$ をとるのは $f_e = 1/E$ がすべての e について成り立つときである。

2. **損失不要の動的ルータバイアス制御**（loss-free dynamic router bias control、L. Wang et al. (2024)。DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024) で採用）：専門化した方が言語モデリング損失が下がる場合であっても $\nabla\mathcal{L}_{\text{aux}}$ はトークン表現を一様なルーティングへと押しやってしまうため、この手法では補助損失の勾配を用いずに負荷を分散させる（な

お DeepSeek-V3 では、これに添えてごく小さな系列単位の負荷分散損失のみを併用している）。その代わりに、各エキスパートは微分不可能な選択バイアス b_e を保持し、これを Top-k 選択の**内部でのみ**加算する（$T(\mathbf{x}) = \text{argtop}_k(\mathbf{p}(\mathbf{x}) + \mathbf{b})$ となり、ゲーティング重み $g_e(\mathbf{x})$ にはバイアスがかからない）。このバイアスは各ステップ後に、更新率 η_b の符号コントローラによって次式で更新される。

$$b_e \leftarrow b_e + \eta_b \,\text{sign}\left(\frac{1}{E} - f_e\right). \tag{11.18}$$

11.7.3.3 負荷分散手法の比較

メソッド `balance(rate)` を `MoELayer` に追加し、2 ブロックのバイトレベル `GPT`（$E = 8, k = 1$ とし、各エキスパートを全幅の FFN とすることで 1 個のアクティブなエキスパートが 1 個の密な FFN と等価になる設定）を用いて、WikiText-103 上で 800 ステップにわたり 5 つの設定を比較する。(1) 初期化状態、(2) 負荷分散なし、(3) 対照実験としての**凍結ルータ**（崩壊にルータとエキスパート間のフィードバックが必要かどうかを検証）、(4) Switch 補助損失（$\alpha = 0.01$）、および (5) 損失不要のバイアス制御（$\eta_b = 10^{-2}$）である。

```
@d2l.add_to_class(MoELayer)
@torch.no_grad()  # Not a gradient step
def balance(self, rate):
    """The controller: move each bias one step toward the even load, then
    count the next batch's assignments afresh."""
    self.expert_bias += rate * torch.sign(self.load.mean() - self.load)
    self.load.zero_()

def train_moe_lm(model, data, optimizer, num_steps, aux_weight=0.0,
                 bias_rate=0.0):
    """d2l.train_steps with a balancing loss and the bias controller."""
    device = d2l.try_gpu()
    model.to(device)
    layers, losses = moe_layers(model), []
    for l in layers:
        l.reset_stats()
    while len(losses) < num_steps:
        for X, Y in data.train_dataloader():
            ce = F.cross_entropy(model(X.to(device)).flatten(0, 1),
                                 Y.to(device).flatten())
            optimizer.zero_grad()
            (ce + aux_weight * sum(l.aux_loss for l in layers)).backward()
            optimizer.step()
            for l in layers:
                l.balance(bias_rate)
            losses.append(ce.detach())
            if len(losses) == num_steps:
                return torch.stack(losses).tolist()
```

検証セットにおけるルーティングの均衡を調べるため、`evaluate_moe` は検証交差エントロピー、降順にソートしたエキスパートごとのトークン占有率、およびドロップされた割り当ての割合を記録する。一方、`measure` は設定された容量係数での検証損失を、上限を設けないバッファ（$S = nk$、ドロップ数 0）での値と比較する。

```
def evaluate_moe(model, data):
    """Validation loss; each expert layer's shares of the validation
    tokens, busiest expert first; and the fraction of assignments dropped."""
    layers = moe_layers(model)
    for l in layers:
        l.reset_stats()
    loss = math.log(d2l.lm_perplexity(model, data.val_dataloader()))
    shares = [sorted((l.load / l.load.sum()).tolist(), reverse=True)
              for l in layers]
    dropped = sum(float(l.dropped) for l in layers) / sum(
        float(l.load.sum()) for l in layers)
    return loss, shares, dropped

def measure(model, data):
    """The validation loss, the same with buffers that drop nothing, the
    busiest expert's load in units of the even share, and the fraction
    dropped; and the shares."""
    layers = moe_layers(model)
    val, shares, dropped = evaluate_moe(model, data)
    peak = max(l.num_experts * s[0] for l, s in zip(layers, shares))
    capacity = [l.capacity_factor for l in layers]
    for l in layers:  # S = n k slots: room for every assignment
        l.capacity_factor = l.num_experts
    val_all = evaluate_moe(model, data)[0]
    for l, c in zip(layers, capacity):
        l.capacity_factor = c
    return (val, val_all, peak, dropped), shares
```

次に、WikiText103 を読み込み、small_gpt を定義する。これは 2 ブロック・バイトレベルの GPT（幅 $d = 128$）であり、capacity_factor=None のときは密な FFN、それ以外のときは 8 エキスパート・Top-1 の MoELayer を使用する。あわせて、凍結ルータの対照アーム用に freeze_router も定義する。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)

def small_gpt(vocab_size, capacity_factor=None):
    """Two blocks of width 128; with a capacity factor, each feed-forward
    network becomes eight experts with one active per token."""
    experts = lambda: MoELayer(128, 8, 1, capacity_factor=capacity_factor)
    model = d2l.GPT(vocab_size, num_hiddens=128, num_heads=4, num_blks=2,
                    ffn_factory=None if capacity_factor is None else experts)
    return model.to(d2l.try_gpu())

def freeze_router(model):
    """Keep every router at its initial weights: no gradient, no update."""
    for l in moe_layers(model):
        l.router.weight.requires_grad_(False)
```

ドライバ関数 balancing_run は、容量係数 $\gamma = 1.25$ で初期化した新規の small_gpt を、指定された負荷分散設定のもとで num_steps=800 ステップ訓練し、measure による集計結果を返す。

```
def balancing_run(data, num_steps=800, frozen=False, aux_weight=0.0,
                  bias_rate=0.0):
    """Measure a fresh small MoE GPT after num_steps steps of training with
    one balancing setting."""
    model = small_gpt(len(data.vocab), capacity_factor=1.25)
    if frozen:
        freeze_router(model)
    if num_steps:
        optimizer = d2l.LMOptimizer(model, lr=2e-3, num_steps=num_steps,
                                    warmup=100)
        train_moe_lm(model, data, optimizer, num_steps, aux_weight,
                     bias_rate)
    return measure(model, data)
```

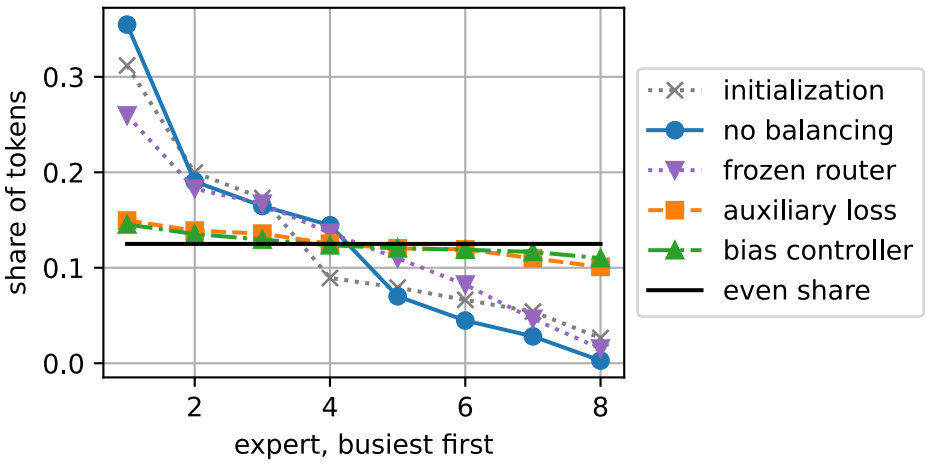
5 つの負荷分散アームのそれぞれについて独立な 2 つのシードを実行し、検証損失（バッファによるドロップあり・なしの両方）、最大エキスパート負荷、およびドロップ率を表にまとめる。

```
arms = {'initialization': dict(num_steps=0), 'no balancing': {},
        'frozen router': dict(frozen=True),
        'auxiliary loss': dict(aux_weight=0.01),
        'bias controller': dict(bias_rate=0.01)}
print(f'spaces: {(data.val == ord(" ")).mean():.1%} of the validation bytes')
print(f'{"two runs each":>15}  {"loss":^9}  {"no drops":^9}  '
      f'{"busiest":^9}  {"dropped":^7}')
shares = {}  # The first run's shares
for name, kwargs in arms.items():
    (a, shares[name]), (b, _) = [balancing_run(data, **kwargs)
                                 for _ in range(2)]
    print(f'{name:>15}  {a[0]:.2f} {b[0]:.2f}  {a[1]:.2f} {b[1]:.2f}  '
          f'{a[2]:3.1f}x {b[2]:3.1f}x  {a[3]:3.0%} {b[3]:3.0%}')
```

```
spaces: 18.9% of the validation bytes
  two runs each    loss      no drops    busiest   dropped
 initialization  5.63 5.49  5.63 5.49  2.5x 2.8x  22% 20%
   no balancing  1.75 1.81  1.70 1.73  3.3x 3.9x  24% 36%
  frozen router  1.71 1.72  1.67 1.67  2.8x 2.1x  14% 17%
 auxiliary loss  1.59 1.54  1.59 1.54  1.3x 1.2x   0%  0%
bias controller  1.58 1.56  1.57 1.55  1.3x 1.5x   0%  2%
```

両方の MoE 層にわたって平均し順位付けしたエキスパートのトークン占有率を、一様分布の基準線 $1/8 = 12.5\%$ と重ねてプロットすると、いずれの負荷分散機構もルーティング崩壊を防いでいる様子が視覚的にわかる。

```
ranks = list(range(1, 9))
mean_shares = [[sum(rank) / len(rank) for rank in zip(*s)]
               for s in shares.values()]  # Mean over the layers
d2l.plot(ranks, mean_shares + [[1 / 8] * 8], 'expert, busiest first',
         'share of tokens', legend=list(shares) + ['even share'],
         fmts=('C7x:', 'C0o-', 'C4v:', 'C1s--', 'C2^-.', 'k-'),
         legend_loc='outside')
```



負荷分散なし（no balancing）では、最も稼働率の高い上位 3 個のエキスパートがルーティング対象トークン全体の約 70% を占める一方で、最も稼働率の低いエキスパートはほとんどトークンを受け取らず、容量係数 $\gamma = 1.25$ のもとで全割り当ての 4 分の 1 から 3 分の 1 がドロップされる。ルータを凍結した場合（frozen router）は偏りがやや緩和されるものの、初期化直後の未訓練モデル（initialization）の時点ですでに負荷分散なしの場合と同程度の偏りがあるため、この規模では偏りの大部分はエキスパートの学習とルータの更新の間のフィードバックではなく入力分布に起因している。1 行あたり 2 回の実行ではこの 2 つの要因を切り分けることはできない（演習問題 3）。Switch 補助損失と損失不要のバイアスコントローラはいずれも、すべてのエキスパートを理想的な占有率 $1/E = 12.5\%$ の近くに保っている（最も負荷の高いエキスパートでもその 2 倍未満に収まり、$\gamma = 1.25$ でのドロップ率は高々数パーセントにとどまる）。

11.7.4 混合エキスパート言語モデル

11.7.4.1 アクティブパラメータ数を揃えた MoE GPT と密な GPT の比較

負荷分散された 8 エキスパート・Top-1 の MoE GPT（$N_{\text{active}} = 0.43\text{M}$、$N_{\text{total}} = 2.26\text{M}$）を、**アクティブパラメータ数とトークンあたり FLOPs が完全に同一**の密な GPT（$N_{\text{active}} = N_{\text{total}} = 0.43\text{M}$）と WikiText-103 上の 6,000 訓練ステップにわたって比較すると、スパースな条件付き容量がもたらす IsoFLOP 上の優位性が明らかになる。

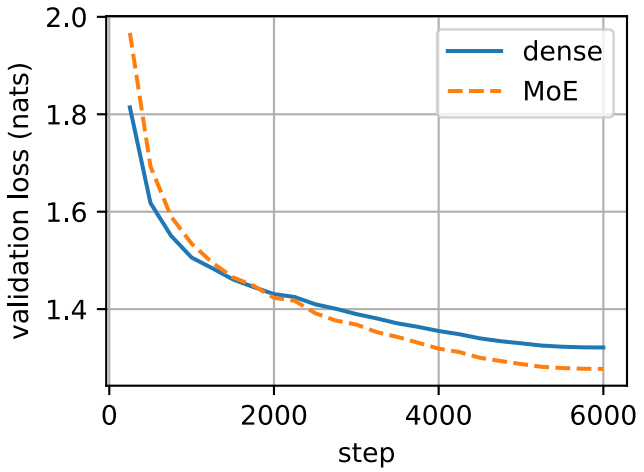
```
capacity = {'dense': None, 'MoE': 2}  # The dense model has no buffers
models = {name: small_gpt(len(data.vocab), capacity_factor=c)
          for name, c in capacity.items()}
for name, model in models.items():
    print(f'{name:>5}: {d2l.num_params(model) / 1e6:5.2f}M parameters, '
          f'{active_params(model) / 1e6:.2f}M active per token')
```

```
dense:  0.43M parameters, 0.43M active per token
  MoE:  2.26M parameters, 0.43M active per token
```

両モデルを独立に 2 回ずつ 6,000 ステップにわたり Switch 補助損失（$\alpha = 0.01$）のもとで訓練し、250 ステップごとに検証損失を評価することで、スパースモデルが密なベースラインをいつ追い抜くかを追跡する。

```
num_steps, every = 6000, 250
curves = {name: [] for name in capacity}  # Validation loss every 250 steps
for run in range(2):
    for name, c in capacity.items():  # Dense first, then MoE
        model = small_gpt(len(data.vocab), capacity_factor=c)
        optimizer = d2l.LMOptimizer(model, lr=2e-3, num_steps=num_steps,
                                    warmup=100)
        curve = []
        for _ in range(num_steps // every):  # Dense: no aux loss to add
            train_moe_lm(model, data, optimizer, every, aux_weight=0.01)
            curve.append(
                math.log(d2l.lm_perplexity(model, data.val_dataloader())))
        curves[name].append(curve)
    dense, moe = curves['dense'][-1], curves['MoE'][-1]
    ahead = next((every * (i + 1) for i, (m, d) in enumerate(zip(moe, dense))
                  if m < d), None)
    print(f'run {run + 1}: dense {dense[-1]:.3f}, MoE {moe[-1]:.3f}; '
          + (f'MoE ahead from step {ahead}, ' if ahead else 'MoE never ahead, ')
          + f'{evaluate_moe(model, data)[2]:.1%} dropped at the end')
d2l.plot(list(range(every, num_steps + 1, every)),
         [curves['dense'][0], curves['MoE'][0]], 'step',
         'validation loss (nats)', legend=['dense', 'MoE'], fmts=('-', '--'))
```

```
run 1: dense 1.321, MoE 1.277; MoE ahead from step 2000, 0.0% dropped at the end
run 2: dense 1.327, MoE 1.265; MoE ahead from step 750, 0.0% dropped at the end
```



MoE モデルは序盤こそ密なモデルに遅れをとるものの、最初の 2,000 ステップ以内に追い抜き、同じトークンあたり FLOP コストでありながら検証交差エントロピーが約 0.05 nat 低い状態で訓練を終える。

11.7.4.2 公開モデルにおける細粒度エキスパートと共有エキスパート

近年の最先端 MoE モデル群（Table 11.4）は、Mixtral 8x7B の粗粒度な 8 エキスパート・Top-2 のレシピを次の 2 つの方向で洗練させている（Figure 11.16）。

1. **細粒度エキスパート分割**（fine-grained expert segmentation）：E 個の全幅エキスパート

のそれぞれを m 個のエキスパート（幅 w/m）に分割し、各トークンを mk 個へと（全 mE 個の中から）ルーティングすると、保持パラメータ数とアクティブパラメータ数の両方を変えないまま、異なるアクティブなエキスパートの組み合わせ数を、たとえば $\binom{16}{2} = 120$ 通りから $\binom{64}{8} \approx 4.4 \times 10^9$ 通りへと拡大できる（$m = 4$ の場合）。公開モデルではさらに踏み込み、128〜256 個のエキスパートのうち 4〜8 個をアクティブにすることで、アクティブな割合も引き下げている。

2. **共有エキスパート** (Dai et al. 2024)：ルーティングを介さず**すべての**トークンに対して実行される幅の狭いエキスパートを $E_{\text{shared}} \geq 1$ 個専任させることで、共通の文法的変換や書式変換を共有エキスパートに吸収させ、ルーティングされるエキスパートが冗長性なく専門化できるようにする。

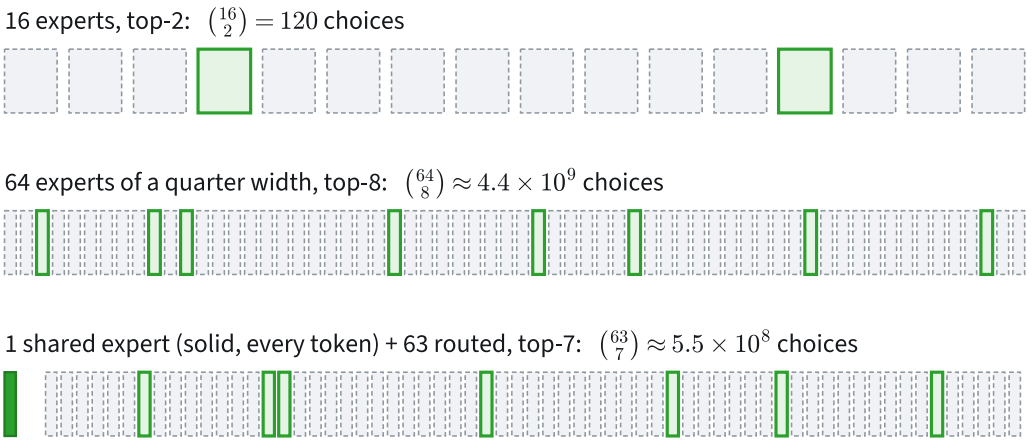


Figure 11.16: 同じエキスパートパラメータ予算を使う 3 通りの方法。各ボックスは 1 個のエキスパートを表し、その幅はサイズに比例する。強調表示されたエキスパートは、ある 1 トークンに対して選択されたものである。上段：16 個のエキスパートのうち、トークンは 2 個を使用する。中段：同じパラメータ数を 4 分の 1 幅の 64 個のエキスパートに分割し、トークンはそのうち 8 個を使用する。下段：幅の狭いエキスパートのうち 1 個をすべてのトークンが使用する共有エキスパートとし、各トークンは残りの 63 個から 7 個を選択する。3 つの行はいずれも保持パラメータ数とトークンあたりの使用パラメータ数が同一である。

Table 11.4: 公開されている MoE 言語モデルにおけるルーティング対象エキスパートの構成。

モデル	層あたりエキスパート数（E）	アクティブ数 k	エキスパート幅 / d	共有エキスパート	保持 / アクティブパラメータ数
Mixtral 8x7B (Jiang et al. 2024)	8	2	3.5	なし	46.7B / 12.9B = 3.6
DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024)	256	8	0.29	1	671B / 37B = 18
Qwen3-235B (A. Yang et al. 2025)	128	8	0.38	なし	235B / 22B = 11

モデル	層あたりエキスパート数（E）	アクティブ数 k	エキスパート幅 / d	共有エキスパート	保持 / アクティブパラメータ数
gpt-oss-120b (OpenAI 2025)	128	4	1.0	なし	117B / 5.1B = 23

11.7.5 まとめ

混合エキスパート（MoE）層（Equation 11.16）は、密な FFN をルーティングされる E 個のエキスパート FFN に置き換え、トークンごとに上位 $k \ll E$ 個のエキスパートのみを活性化させることで、**保持パラメータ数** N_{total}（メモリ容量）を**アクティブパラメータ数** N_{active}（トークンあたり FLOPs）から切り離す。負荷分散を行わないとエキスパートの稼働率は大きく偏る。この偏りのかなりの部分は初期化時点ですでに存在しており、ルータとエキスパート間の正のフィードバックによって**ルーティング崩壊**へと増幅されうるが、**Switch 補助損失** $\mathcal{L}_{\text{aux}} = E \sum_e f_e \bar{p}_e$（Equation 11.17）と**損失不要の動的ルータバイアス** $b_e \leftarrow b_e + \eta_b \text{sign}(1/E - f_e)$（Equation 11.18）はいずれもエキスパートの負荷を $1/E$ 付近に保つ。アクティブパラメータ数と訓練 FLOPs を揃えた比較では、負荷分散された MoE 言語モデルの方が密な GPT よりも低い検証交差エントロピーに到達する。近年のオープンウェイト MoE アーキテクチャ（Table 11.4）はさらに、組み合わせによる専門化を広げるためにエキスパートを**細粒度**のバンク（$E \in [128, 256], k \in [4, 8]$）へと分割し、多くの場合そこに常時アクティブな**共有エキスパート**を併用している（Figure 11.16）。

11.7.6 演習問題

1. **$k = 1$ と $k > 1$ の再正規化におけるルータ勾配**。多くの MoE 層では、選択された確率をその和で再正規化する。すなわち $g_e(\mathbf{x}) = p_e(\mathbf{x}) / \sum_{j \in T(\mathbf{x})} p_j(\mathbf{x})$ とする。
    1. $k = 1$ のときにこれと同じ再正規化を適用すると、$g_{e^*}(\mathbf{x}) = p_e(\mathbf{x})/p_e(\mathbf{x}) \equiv 1$ が定数となるため $\partial \text{MoE}(\mathbf{x})/\partial \mathbf{W}_r = \mathbf{0}$ となり、ルータがタスク損失からの勾配を一切受け取らないことを示せ。
    2. 本節の `MoELayer` は、選択された各エキスパートを生の確率で重み付けすることで（$g_{e^*}(\mathbf{x}) = p_{e^*}(\mathbf{x})$ とする Switch Transformer (Fedus et al. 2022) と同様に）この問題を回避していることをコードで確認せよ。
2. **Switch 補助損失の最小値と勾配**。$\mathcal{L}_{\text{aux}} = E \sum_{e=1}^{E} f_e \bar{p}_e$（Equation 11.17）を考える。ただし、f_e は勾配停止された定数として扱い、$\sum_e \bar{p}_e = 1$ とする。
    1. 偏微分 $\partial \mathcal{L}_{\text{aux}} / \partial s_{t,e}$（トークン t のルータロジット $s_{t,e} = (\mathbf{x}_t \mathbf{W}_r)_e$ に関する微分）を計算し、これが 0 になるのはすべての経験的負荷が等しいとき、かつそのときに限ること（$f_1 = \cdots = f_E = 1/E$）を示せ。
    2. コーシー・シュワルツの不等式を用いて、$\bar{p}_e = f_e$ のときに $\mathcal{L}_{\text{aux}} \geq 1$ が成り立ち、等号成立は $f_e = 1/E$ のとき、かつそのときに限ることを証明せよ。また、その最大値 E（単一エキスパートへの完全な崩壊 $f_1 = \bar{p}_1 = 1$ のとき）を求めよ。
3. **正のフィードバックと入力分布の偏りの切り分け**。`balancing_run` において、`frozen router` アームと `no balancing` をステップ $100, 200, 400, 800$ で比較せよ。初期状態のモデルにおける最上位エキスパートの占有率（本節のバイトレベルモデルではすでに約 3 分の 1 に達している）を起点としたとき、判別に十分な数のシードにわたって評価すると、訓練可能なルータの最上位エキスパート占有率は凍結ルータよりもそこから大きく離れるか。また、すべてのエキスパートの出力射影 `W_2` を 0 で初期化し、初期ステップですべてのエキスパートが同一の出力を生成するようにすると何が起きるか。

4. **損失不要バイアスコントローラのステップサイズに関するダイナミクス。** `balancing_run` を `bias_rate` $\eta_b \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$ で実行し、訓練ステップにわたる最上位エキスパートの占有率と検証損失をプロットせよ。コントローラのステップサイズが大きすぎる場合（$\eta_b = 10^{-1}$）、なぜエキスパートの負荷がステップごとに振動するのか。

5. **極小データ領域における MoE の過学習。** 6,000 ステップの `dense` 対 `MoE` の比較（11.7.4.1 節）を、ただし `data.train` を先頭の $65{,}536$ バイト（64 KB）のみに制限し、モデルが同じテキストを何百回も反復するようにして再実行せよ。`dense` と `MoE` の両方について訓練損失と検証損失をプロットせよ。両者がトークンあたりで全く同じ FLOPs を使用しているにもかかわらず、なぜ 2.26M パラメータの MoE は小さなコーパスをはるかに速く暗記し、0.43M パラメータの密なモデルよりも早く過学習し始めるのか。

6. **固定予算下での共有エキスパートと細粒度化。** 常時アクティブな 1 個の共有エキスパート（幅 w）に加えて Top-k 個（ルーティングされる E 個の幅 w のエキスパートから選択）を実行する `SharedMoELayer` を実装し、保持およびアクティブなエキスパート幅を揃えた条件（ブロックあたり保持 16×170、アクティブ 2×170）のもとで、`small_gpt` 内の 3 つの FFN 構成を比較せよ。(a) 粗粒度 $E = 8, k = 1, w = 340$、(b) 中粒度 $E = 16, k = 2, w = 170$、(c) 細粒度と共有 $E_{\text{shared}} = 1, E = 15, k = 1, w = 170$ とし、800 ステップ後の検証損失を比較せよ。

11.8 関連文献

本章で構築してきたアーキテクチャは、共通の設計を備えている。すなわち、Pre-Norm の残差ストリーム（11.1 節）、因果または双方向のマルチヘッドアテンション（11.2 節、11.4 節）、ヘッド・トークン・チャネルにわたって圧縮されたキー・値キャッシュ（11.3 節）、そして密に（11.6 節）またはスパースなエキスパートルーティングを通じて（11.7 節）スケールされたフィードフォワード層である。本節では、これらの基盤の上に築かれた重要なアーキテクチャ上のバリエーションと研究の方向性を概観する。

11.8.1 正規化の配置と残差ストリームの派生形

Pre-Norm（各サブ層の前に `RMSNorm` を配置する方式）は、出力から最初の層まで減衰のない恒等経路を保つため、現代の大半の Transformer で標準となっている（11.1 節）。しかし、非常に深いモデルや長時間の FP16/BF16 訓練では、正規化されない残差ストリームが深さ方向にわたって大きな活性化値ノルムを蓄積しうるほか、アテンションスコアの大きさもソフトマックスが飽和するまで上方へドリフトしかねない。この問題に対処する改良には 2 つの系統がある（Figure 11.17）。

- **サブ層出力の正規化（Sandwich-LN / Peri-LN）と Post-Norm の派生形：** 各サブ層の入力と出力の両方を正規化する構成（$x + \mathrm{Norm}(f(\mathrm{Norm}(x)))$）は、恒等スキップ接続（skip connection）を損なわずに各残差更新の大きさを有界に保つものであり、CogView (M. Ding et al. 2021) や Gemma 2/3 (Gemma Team 2024, 2025) で採用されている。一方、OLMo 2 (Team OLMo et al. 2025) は RMSNorm を各サブ層の出力側へ移し（$x + \mathrm{Norm}(f(x))$）、これを QK-Norm と組み合わせることで、加算後の正規化によって恒等経路を遮ることなく残差ストリームの増大を有界に抑えている。
- **クエリ・キー正規化（QK-Norm）：** ヘッドごとにクエリとキーへ RMSNorm を適用してから $QK^\top/\sqrt{d_h}$ を計算すると（11.1.2.5 節）、残差ストリームの増大にかかわらず、コーシー・シュワルツの不等式によってソフトマックス前のロジットが有界に保たれる (Henry et al. 2020; Dehghani et al. 2023)。この手法は ViT-22B、Gemma 3、Qwen3、OLMo 2 で標準となっている。

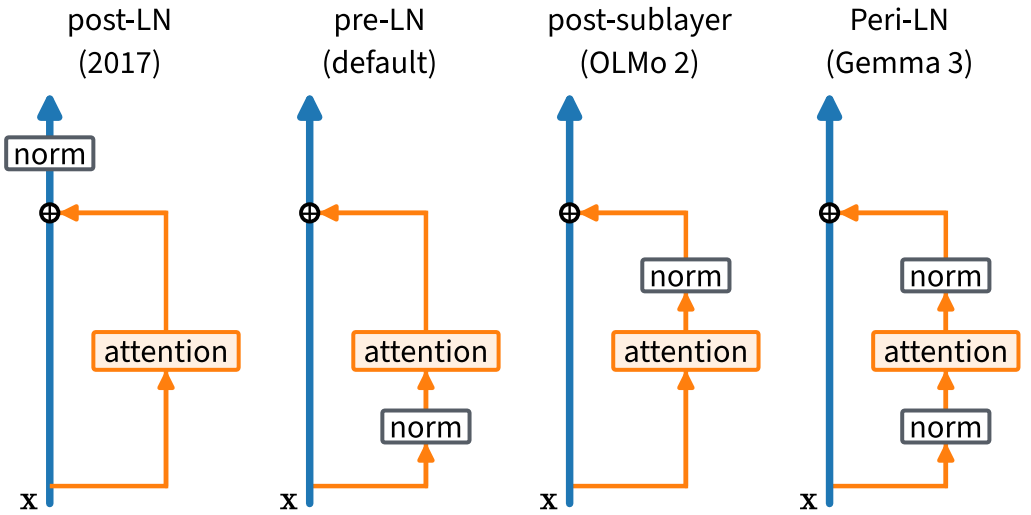


Figure 11.17: サブ層 f の前後に正規化を配置する 4 つの方法。Post-LN は残差ストリーム自体を正規化する。Pre-LN はサブ層の入力のみを正規化する。サブ層後正規化（OLMo 2）はストリームへ戻して加算する前にサブ層の出力のみを正規化する。Peri-LN（Sandwich-LN、Gemma 3）はサブ層の入力と出力の両方を正規化する。

- **深さスケーリング付き Post-Norm とロジットのソフトキャッピング：**DeepNorm (H. Wang et al. 2024) は Post-LN ブロックを維持しつつ、明示的な深さ依存の定数によってスキップ経路の重みを大きくし（$\mathrm{Norm}(\alpha x + f(x))$、ただし $\alpha = (2L)^{1/4}$）、同時にブランチの重みの初期化を $\beta = (8L)^{-1/4}$ 倍に縮小する。この両方によってストリームに対する各更新が相対的に小さくなり、Post-LN Transformer を 1,000 層まで安定化できる。別の方法として、Gemma 2 は滑らかな tanh ソフトキャッピング（$c\tanh(z/c)$）によってアテンションロジットと最終ロジットを有界に抑えているが、より新しいモデルではアテンションロジットのソフトキャッピングはおおむね QK-Norm に取って代わられている。

11.8.2 フィードフォワードネットワークと活性化関数

元の Transformer は、中間幅 $d_{\text{ff}} = 4d$ をもつ 2 行列の ReLU フィードフォワードネットワークを用いていた (Vaswani et al. 2017)。GPT-2 と BERT は ReLU を GELU に置き換えた (Hendrycks and Gimpel 2016)。Shazeer (2020) 以降、現代のほぼすべての言語モデルはゲート付き線形ユニット（gated linear unit）(Dauphin et al. 2017)、すなわち **SwiGLU**（SiLU ゲート）または **GeGLU**（GELU ゲート）を採用しており、中間幅 $d_{\text{ff}} \approx \frac{8}{3}d$ をハードウェアに適した 128 または 256 の倍数に丸めることで、3 つの射影行列（$W_{\text{gate}}, W_{\text{up}}, W_{\text{down}}$）を $8d^2$ のパラメータ数・FLOP 予算（ゲートなしの幅 $4d$ の FFN と同量）に一致させている（11.1 節）。極端に大きな幅では、**並列 Transformer ブロック**（PaLM (Chowdhery et al. 2022)、ViT-22B (Dehghani et al. 2023)）が同じ正規化済み入力からアテンションと FFN を並列に計算するため（$x + \mathrm{Attn}(\mathrm{Norm}(x)) + \mathrm{FFN}(\mathrm{Norm}(x))$）、アテンションと FFN の入力射影を単一の大規模な行列積へと融合できる。

11.8.3 長文脈推論とキャッシュ圧縮

デコーディングはメモリ帯域幅律速であるため（11.3.2.1 節）、キー・値キャッシュの通信量を削減することが効率的なサービングの中心課題となる。

- **グループ化クエリアテンションとマルチヘッド潜在アテンション：**マルチクエリアテンション（MQA）(Shazeer 2019) とグループ化クエリアテンション（grouped-query attention、GQA）(Ainslie et al. 2023) は、クエリヘッドのグループ間でキーヘッドと値ヘッドを共有する（11.3.3 節）。DeepSeek-V2 (DeepSeek-AI 2024) で導入され DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024) でも引き継がれた**マルチヘッド潜在アテンション**（multi-head latent attention、MLA）は、低ランク KV 圧縮（11.3.3.2 節）を一般化した手法である。キーと値を結合的に単一の低次元潜在ベクトル $c_t^{\mathrm{KV}} \in \mathbb{R}^{d_c}$（$d_c \ll Hd_h$）へ射影してメモリにキャッシュする一方、切り離された低次元の RoPE キーベクトルと対にすることで、デコーディング時の重み吸収（weight absorption）と回転位置エンコーディングの両立を保っている。
- **スライディングウィンドウ、アテンションシンク、ハイブリッドスタック：**直近トークンの移動ウィンドウとあわせて先頭のプロンプトトークンを保持する手法（**StreamingLLM** (Xiao et al. 2024)、11.3.3.3 節）は、ソフトマックスの分母が余剰の確率質量を先頭の「シンク」トークンで吸収するため、ストリーミング推論中も生成品質を維持できる。近年のいくつかのアーキテクチャでは、局所的なスライディングウィンドウアテンション層と完全な大域的因果アテンション層を交互に配置したり（Gemma 3 では大域層 1 層につき局所層 5 層、gpt-oss では 1 層おき）、アテンション層と線形時間の状態空間層・リカレント層を交互に配置したりしている（第 12 章）。
- **投機的デコーディング：**候補となる m 個のトークンを並列に検証する処理は、単一トークンをデコードする場合とほぼ同じメモリ帯域幅コストで実行できる（11.3.2.1 節）。そのため、**投機的デコーディング**（speculative decoding）(Leviathan et al. 2023; C. Chen et al. 2023) は高速なドラフトモデル（またはターゲットモデルに付加した追加のマルチトークン予測ヘッド）を用いて m 個のトークンを提案し、ターゲット分布を厳密に保つ棄却サンプリング（rejection sampling）規則によって、ターゲットモデルの 1 回の順伝播でそれらを検証する。

11.8.4 エンコーダ、クロスアテンション、ビジョンバックボーン

- **双方向エンコーダと検索：**オープンエンドな生成では因果デコーダが主流を占める一方、双方向エンコーダ（11.4 節）は依然として密なテキスト埋め込み、リランキング、およびトークンレベルの分類における主力である。RoBERTa (Liu et al. 2019) と DeBERTa (He et al. 2021) は BERT のマスク言語モデリング（masked language modeling）による事前学習レシピを洗練させ、ModernBERT (Warner et al. 2024) は現代的な Pre-Norm、RoPE、SwiGLU、およびパディングなしの FlashAttention スタックを双方向エンコーダへと導入している。
- **Vision Transformer と階層型ハイブリッド：**中程度のデータ規模で ViT を効果的に訓練するには、強力なデータ拡張と正則化（DeiT (Touvron, Cord, Douze, et al. 2021)）、自己教師ありのマスクパッチ予測（MAE (He et al. 2022)）または自己蒸留（self-distillation、DINOv2 (Oquab et al. 2024)）、および 2D 回転位置埋め込み（rotary position embedding）が鍵となる。マルチスケールの特徴マップを必要とする密な予測タスク（検出、セグメンテーション）に対しては、Swin Transformer (Liu et al. 2021) が階層型ピラミッドにわたってシフトされた局所 2D ウィンドウ内に自己アテンションを制限する一方、ConvNeXt (Z. Liu et al. 2022) は Transformer ブロックの設計上の選択を純粋なデプスワイズ分離可能畳み込みネットワークへと還元している。

11.8.5 スケーリング則と混合エキスパートの設計

- **計算量最適スケーリングと推論考慮型スケーリング：**@kaplan2020scaling は、言語モデルの交差エントロピー損失がモデルサイズ N、データセットサイズ D、および計算量 C に対してべき乗則に従ってスケールすることを初めて確立した。続いて Hoffmann et al. (2022)（Chinchilla）は IsoFLOP 曲線を用い、一定の事前学習計算量のもとで訓練損失が最小化されるのは、N と D を同じ比率でスケールさせたとき（$D \approx 20N$）であることを示した。現代のオープンウェイトモデル（15 兆トークンで訓練された Llama 3 8B や Qwen3 など）は、小さなモデルを Chinchilla 比（11.6 節）をはるかに超えてあえて過剰に訓練することで、運用期間全体の総推論コストに対して訓練計算量を償却している (Sardana et al. 2024)。
- **スパース混合エキスパート：**Top-K ノイジーゲーティング (Shazeer et al. 2017)、GShard (Lepikhin et al. 2021)、および Switch Transformer (Fedus et al. 2022) を基盤とする現代の MoE モデルは、主に粒度において異なっている（11.7 節）。Mixtral 8x7B (Jiang et al. 2024) が全幅のエキスパート 8 個のうち 2 個へ各トークンをルーティングするのに対し、Qwen3 (A. Yang et al. 2025) や DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024) は 100 個を超える幅の狭いエキスパートのうち 8 個を選択する。さらに DeepSeek-V3 は常時活性な共有エキスパート (Dai et al. 2024) を追加し、ごく小さな補助ペナルティだけを残して、主に動的なエキスパートごとのルータバイアス (L. Wang et al. 2024) によって負荷を分散させている。また、**スパースアップサイクリング**（sparse upcycling）(Komatsuzaki et al. 2023) によって密なチェックポイントを MoE モデルへ変換することも可能であり、訓練済みの密な FFN から各エキスパートを初期化したうえで事前学習を継続する。

Chapter 12

状態空間モデル

第 8 章のリカレントネットワークは、テキストを 1 トークンずつ順番に読み込み、それまでに読み込んだ内容を固定サイズの**状態**（state）に保持して引き継ぐ。したがって、100 万番目のトークンを生成するコストは、10 番目のトークンを生成するコストと変わらない。これに対し、第 11 章の Transformer はすべてのトークンを保持する。長いレポートに関する質問に答える際には、レポートのすべてのトークンについてキーと値を保持するため（11.3 節）、メモリもトークンあたりの計算量もコンテキスト長（context length）に応じて増大する。ところが 11.2 節では、そこで構築した GPT と同規模の単純なリカレントネットワークは、コンテキストの最初の数バイトを過ぎたところで遅れをとり、最後まで追いつくことはなかった。より優れたリカレントネットワークはその差を埋められるのだろうか。そして、増大するアテンションのコストはどの程度の長さから問題になり始めるのだろうか。

そこには 2 つの障壁が立ちはだかっている。古典的なリカレントネットワークは状態を非線形に更新するため、訓練を 1 ステップずつ進めざるを得ず、GPU の大部分をアイドル状態のまま遊ばせてしまう。また、固定サイズの状態は忘却を避けられないため、何を保持するかを選択しなければならない。その解決策は 3 つの方向から得られる。すなわち、1990 年代のゲート機構 (Hochreiter and Schmidhuber 1997; Gers et al. 2000)、制御理論における線形力学系（linear dynamical system）(Kalman 1960) を訓練可能な層として深層学習に導入した手法 (Gu et al. 2020; Gu, Goel, et al. 2022)、そしてソフトマックスを除いたアテンション (Katharopoulos et al. 2020) である。これら 3 つの流れは、並列に訓練でき、一種の線形アテンション（linear attention）を計算するリカレント層において合流する。本章で扱うすべてのリカレント層は、状態、各トークンを取り込む**更新**（update）、および出力を計算する**読み出し**（readout）を備えている。制御工学ではこのようなシステムを**状態空間モデル**（state space model）と呼ぶ。本章のタイトルではこの用語をこの広義の意味で用い、12.4 節では連続時間の線形システムという狭義の意味で用いる。

まず、何を保持するかをネットワークが学習できるようにするゲート機構と、単一の状態ベクトルが計測可能なボトルネックを生む 2014 年のエンコーダ・デコーダから始める。続いて 3 つの改良により、状態を高速かつ選択的（selective）なものにする。すなわち、すべての位置にわたって並列に評価できアテンションと実行時間を比較できる線形更新（12.3.4 節）、連続時間から離散化された複数時間スケールの減衰、そして各トークンがどれだけ書き込みどれだけ忘却するかを自ら決められるようにする入力依存のステップサイズである。さらに状態をベクトルから行列へと拡張すると、状態はキー・値ペアの連想メモリ（associative memory）となり、その記憶容量を計算できるほか、デルタ則（delta rule）によって書き込み規則を先鋭化できる。最後に、少数のフルアテンション層を挟み込むことで、いかなる固定サイズの状態も持ち得ない正確な想起（exact recall）を取り戻す。これは複数の実用言語モデルで採用されているハイブリッドな構成

パターンである (Lieber et al. 2024; Qwen Team 2025)。

初読の際には、12.2 節、12.4 節における連続時間の導出、および 12.6 節と 12.7 節のチャンク化アルゴリズムは後回しにしてよい。特定の記憶能力を 1 つずつ切り分けるよう設計された小規模な合成タスクによって各メカニズムを検証し、11.2 節のバイトレベルのテキストベンチマークによって各ミキサが言語上でどのような性能を示すかを確認する。

12.1 ゲート付きリカレントネットワーク

8.4.2 節の素のリカレントネットワークは、単語や句読点としては認識できる文字列を生成するものの、数語を超えると文法が崩れてしまう。「**the keys to the cabinet in the hall**」に続く動詞を予測する場面を考えよう。ここで「**is**」ではなく「**are**」を選ぶには、30 文字前にある「**keys**」が複数形であることを覚えている必要がある。素の RNN は毎ステップ、同じリカレント行列と tanh 非線形性を通して状態全体を上書きするため、順方向の信号も逆方向の勾配も数十ステップで急速に減衰する。リカレントネットワークは、読み取った情報を数百ステップにわたってどのように保持できるだろうか。

1990 年代後半に導入された解決策は、現在の入力と前の状態から計算される $(0, 1)$ の値をもつベクトルである**ゲート**を介して、状態のどの程度の割合を保持するかをユニットごと・ステップごとにネットワーク自身に決定させることである。ゲートが 1 の近くにとどまるユニットは、保持している内容と逆方向の勾配の両方をほぼ変化させずに次のステップへと伝える。以下では、古典的な 2 つのゲート付きアーキテクチャである長・短期記憶（long short-term memory、LSTM）とゲート付きリカレントユニット（gated recurrent unit、GRU）を構築し、合成データの遅延想起（delayed recall）および第 11 章のバイトレベル WikiText-103 ベンチマークで評価する。ゲートは長距離の記憶保持を解決する一方で、前の状態に依存するためステップごとの逐次的な訓練が依然として避けられない。このボトルネックは 12.3 節で取り除く。

本章を通じて、数式は列ベクトルを用いて単一の系列に対して記述し、コードでは行ベクトルを用いて系列のミニバッチを処理する。

12.1.1 ゲート

12.1.1.1 勾配消失

8.5.1 節で見たように、隠れ状態（hidden state）$\mathbf{h}_t \in \mathbb{R}^d$ をもつ素の RNN を通じて勾配がどのように伝播するかを振り返ろう。

$$\mathbf{h}_t = \tanh(\mathbf{W}_{\text{hh}}\mathbf{h}_{t-1} + \mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{b}_{\text{h}}). \tag{12.1}$$

$\mathbf{h}_{t-k}$ に対する摂動は、$\mathbf{h}_t$ に到達するまでに k 個のステップごとのヤコビ行列 $\mathbf{J}_s = \operatorname{diag}(1 - \mathbf{h}_s^2)\,\mathbf{W}_{\text{hh}}$ の積が掛かる。

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-k}} = \mathbf{J}_t \mathbf{J}_{t-1} \cdots \mathbf{J}_{t-k+1}. \tag{12.2}$$

単一のスカラーユニットでは、各因子は $w(1-h_s^2)$ であり、その大きさは高々 $|w|$ である。$|w| < 1$ のとき、この積は少なくとも $|w|^k$ の速さで縮小する（$w = 0.9$ では、30 ステップで約 4%、100 ステップでは $\le 0.003\%$ となる）。

行列 $\mathbf{W}_{\text{hh}}$ については、その最大特異値が 1 未満であれば、すべての因子があらゆるベクトルを縮小させるため、積は必ずゼロに収束する。多くのステップにわたると、行列の累乗は ρ^k のオー

ダーで変化する。ここで、**スペクトル半径**（spectral radius）ρ（固有値の絶対値の最大値）は $\rho \le \|\mathbf{W}_{\text{hh}}\|_2$ を満たす。$\rho \approx 1$ であり、かつ状態が小さく保たれて tanh がほぼ線形であれば、主固有ベクトル方向の成分は多くのステップにわたって生き残り得る。この境界領域の挙動は、以下の実験で実際に観察する。$\rho \approx 1$ から離れると、勾配は爆発するか（これは勾配クリッピングで上限を抑えられる）、あるいは消失する（これは事後的な再スケーリングでは回復できない）。

12.1.1.2 加法的な記憶

ヤコビ行列が単位行列となる最も単純な漸化式は純粋な累積和 $c_t = c_{t-1} + x_t$ であり、このとき $\partial c_t / \partial c_{t-k} = 1$ があらゆるラグ k に対して成り立つ。しかし、重み付けのない和は決して忘却しない。過去のあらゆる入力が等しい重みで重ね合わされたまま残り、状態の分散は系列長とともに増大する。

12.1.1.3 学習される減衰としてのゲート

関連する入力を保持しつつ無関係な入力を捨てるため、ネットワークが各ステップで保持率 $f_t \in (0,1)$、および入力率 $i_t \in (0,1)$（新たな候補内容 $\tilde{c}_t$ に乗じる割合）を選択できるようにする。

$$c_t = f_t \, c_{t-1} + i_t \, \tilde{c}_t. \tag{12.3}$$

直接的な記憶経路に沿った k ステップの導関数は保持率の積 $\partial c_t / \partial c_{t-k} = \prod_{s=t-k+1}^{t} f_s$ となる。保持率が一定の f であるとき、記憶された値は f^k に従って減衰し、その**半減期**（half-life）は $\ln 2 / \ln(1/f)$ ステップとなる。具体的には、$f = 0.9$ で約 7 ステップ、$f = 0.99$ で 69 ステップ、$f = 0.999$ で 693 ステップである。$f_t \to 1$ とすれば値とその勾配は無期限に保持され、$f_t \to 0$ とすればセルは 1 ステップで消去される。

幅 d の層では、記憶はベクトル $\mathbf{c}_t \in \mathbb{R}^d$ となり、外部に公開される隠れ状態 $\mathbf{h}_t \in \mathbb{R}^d$ を伴う。各**ゲート**は $(0,1)^d$ のベクトルを出力し、これは現在の入力 $\mathbf{x}_t$ と前の出力 $\mathbf{h}_{t-1}$ を読み取るシグモイド層によって次のように計算される。

$$\mathbf{f}_t = \sigma(\mathbf{W}_{\text{xf}} \mathbf{x}_t + \mathbf{W}_{\text{hf}} \mathbf{h}_{t-1} + \mathbf{b}_{\text{f}}). \tag{12.4}$$

毎ステップ状態全体に密行列 $\mathbf{W}_{\text{hh}}$ を掛けて tanh を適用する素の RNN とは異なり、ゲート付き記憶は各座標に固有の学習された係数 $f_{t,j} \in (0,1)$ を要素ごとに乗じて更新する。$f_{t,j} \le 1$ であるため、直接的な記憶経路に沿った勾配は爆発し得ず、さらに $f_{t,j}$ を 1 の近くに保てば消失も防げる。

実用上の細部が 2 つ残っている。第 1 に、$f_t \approx 1$ の間も書き込みが繰り返されると記憶された値 c_t が増大し得るため、LSTM は tanh を適用して $\mathbf{c}_t$ を $\mathbf{h}_t$ へと読み出す。第 2 に、ゼロバイアス初期化のもとではシグモイドゲートは $\sigma(0) = 1/2$（半減期 1 ステップ）の近くから始まるため、ゲート付きネットワークは長期の記憶を表現できるものの、デフォルトの初期状態では長期の記憶をもたない。

12.1.2 長・短期記憶

12.1.2.1 記憶セル

Hochreiter and Schmidhuber (1997) による**長・短期記憶**（LSTM）に Gers et al. (2000) の忘却ゲート（forget gate）を加えたアーキテクチャは、Equation 12.3 を中心にリカレント層を構成

する。この層は幅 d のベクトルを 2 つ保持する。1 つは要素ごとのゲート処理と加算のみによって時間発展する内部の**記憶セル**（memory cell）$\mathbf{c}_t$ であり、もう 1 つは外部に公開される**隠れ状態** $\mathbf{h}_t$ であり、後者は $\mathbf{c}_t$ を圧縮してゲートに通した表現を後続の層や次ステップのゲートに提供する。

セルへのアクセスは、**入力ゲート**（input gate）$\mathbf{i}_t$、**忘却ゲート** $\mathbf{f}_t$、**出力ゲート**（output gate）$\mathbf{o}_t$ という 3 つのシグモイドゲートが制御する。

$$\begin{aligned}\mathbf{i}_t &= \sigma(\mathbf{W}_{\text{xi}}\mathbf{x}_t + \mathbf{W}_{\text{hi}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{i}}),\\ \mathbf{f}_t &= \sigma(\mathbf{W}_{\text{xf}}\mathbf{x}_t + \mathbf{W}_{\text{hf}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{f}}),\\ \mathbf{o}_t &= \sigma(\mathbf{W}_{\text{xo}}\mathbf{x}_t + \mathbf{W}_{\text{ho}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{o}}).\end{aligned} \tag{12.5}$$

tanh 活性化関数をもつ 4 つ目の射影は、候補となる**入力ノード**（input node）$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{\text{xc}}\mathbf{x}_t + \mathbf{W}_{\text{hc}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{c}})$ を提案する。続いて、セルと隠れ状態は要素ごとの積（$\odot$）を用いて次のように更新される。

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \qquad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{12.6}$$

$\mathbf{c}_t$ を $\mathbf{h}_t$ から分離することで、ユニットは必要になるまで情報を外部に出さずに保持できる。たとえば、$c_{t-1} = 0.8$ に対して $f_t = 0.95$、$i_t = 0.1$、$\tilde{c}_t = -0.5$、$o_t = 0.2$ であれば、セルは $c_t = 0.71$ に保たれる一方、外部には $h_t = 0.2 \tanh(0.71) = 0.12$ しか出力されない（Figure 12.1）。

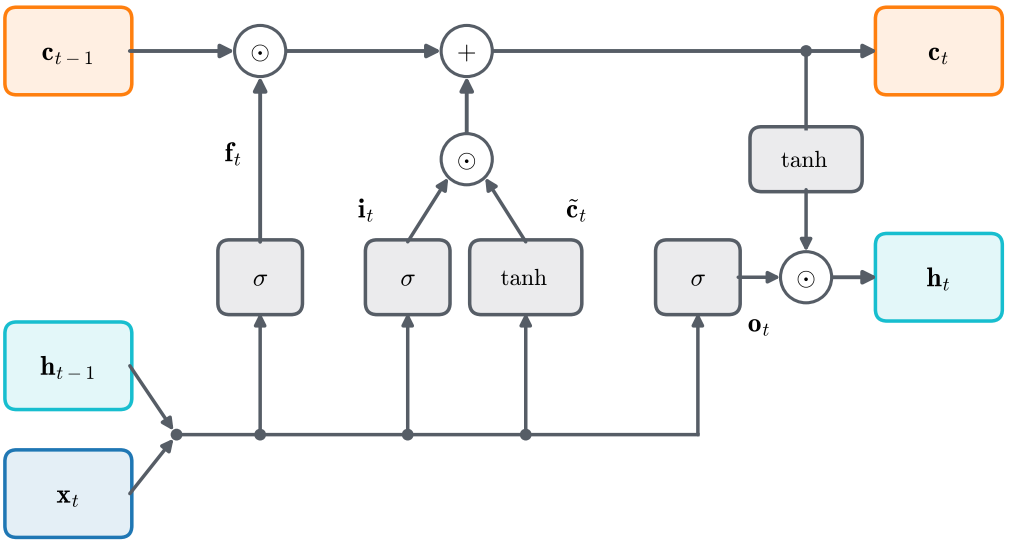


Figure 12.1: LSTM の記憶セル。前の出力 $\mathbf{h}_{t-1}$ と入力 $\mathbf{x}_t$ が、3 つのシグモイドゲートと 1 つの tanh 入力ノードからなる 4 つの層に入力される。セル $\mathbf{c}_t$ は忘却ゲートでスケールされつつ加法的に更新され、ネットワークの残りの部分が何を受け取るかは出力ゲートが制御する。

ゲートを固定したものとして扱うと、セルからセルへの直接的なヤコビ行列 $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ は単に $\text{diag}(\mathbf{f}_t)$ となる（Figure 12.1 の上部の経路）。完全な導関数には $\mathbf{h}_{t-1}$ とゲートを介した間接項も含まれるが、セルの経路は $\mathbf{f}_t$ が 1 の近くにとどまる限り減衰のないハイウェイを提供する。

12.1.2.2 実装

$\mathbf{i}_t$、$\mathbf{f}_t$、$\tilde{\mathbf{c}}_t$、$\mathbf{o}_t$ はいずれも $(\mathbf{x}_t, \mathbf{h}_{t-1})$ を入力とするため、それぞれの重みを 2 つの行列 $\mathbf{W}_{\mathrm{x}} \in \mathbb{R}^{d_{\mathrm{in}} \times 4d}$ と $\mathbf{W}_{\mathrm{h}} \in \mathbb{R}^{d \times 4d}$ に束ね、入力ごとに 1 回の行列積を計算してから結果を 4 つに分割する。

```
class LSTMScratch(nn.Module):
    """An LSTM layer computed one time step at a time."""
    def __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.num_hiddens = num_hiddens
        # Weights of i, f, c_tilde, and o, stacked along the last axis, with
        # variance 1 / fan-in
        self.W_x = nn.Parameter(torch.randn(num_inputs, 4 * num_hiddens)
                                / math.sqrt(num_inputs))
        self.W_h = nn.Parameter(torch.randn(num_hiddens, 4 * num_hiddens)
                                / math.sqrt(num_hiddens))
        self.b = nn.Parameter(torch.zeros(4 * num_hiddens))

    def forward(self, X):  # X: (batch_size, num_steps, num_inputs)
        h = c = torch.zeros(X.shape[0], self.num_hiddens, device=X.device)
        outputs = []
        for x in X.unbind(1):  # One time step at a time
            i, f, c_tilde, o = (x @ self.W_x + h @ self.W_h + self.b).chunk(4, -1)
            c = torch.sigmoid(f) * c + torch.sigmoid(i) * torch.tanh(c_tilde)
            h = torch.sigmoid(o) * torch.tanh(c)
            outputs.append(h)
        return torch.stack(outputs, 1)  # (batch_size, num_steps, num_hiddens)
```

`LSTMScratch` がフレームワークの LSTM 層とまったく同じ漸化式を計算することを検証するため、ライブラリの重みを自作クラスにコピーし、ランダムなバッチに対する出力を比較する。

ライブラリの層は融合された `nn.LSTM` である。

```
lstm = nn.LSTM(16, 32, batch_first=True)  # Stacks its weights as i, f, c~, o
cell = LSTMScratch(16, 32)
with torch.no_grad():
    cell.W_x.copy_(lstm.weight_ih_l0.T)
    cell.W_h.copy_(lstm.weight_hh_l0.T)
    cell.b.copy_(lstm.bias_ih_l0 + lstm.bias_hh_l0)
X = torch.randn(4, 9, 16)  # (batch_size, num_steps, num_inputs)
print(f'max difference: {(cell(X) - lstm(X)[0]).abs().max():.1e}')
```

```
max difference: 6.0e-08
```

出力は float32 の丸め誤差（$\sim 10^{-7}$）の範囲で一致するため、以降はより高速な融合済みのライブラリ層を使用する。

12.1.3 ゲート付きリカレントユニット

ゲート付き層には、本当に 2 つの状態ベクトルと 3 つのゲートが必要なのだろうか。Cho, Van Merriënboer, Gulcehre, et al. (2014) の**ゲート付きリカレントユニット**（GRU）は、セルと隠れ状態を単一のベクトル $\mathbf{h}_t$ に統合し、わずか 2 つのゲート（**リセットゲート** $\mathbf{r}_t$ と**更新ゲート**（update

gate）$\mathbf{z}_t$）で制御しながら、多くの系列タスクで LSTM に匹敵する性能を達成する (Chung et al. 2014)。

$$\mathbf{r}_t = \sigma(\mathbf{W}_{\text{xr}}\mathbf{x}_t + \mathbf{W}_{\text{hr}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{r}}), \qquad \mathbf{z}_t = \sigma(\mathbf{W}_{\text{xz}}\mathbf{x}_t + \mathbf{W}_{\text{hz}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{z}}). \tag{12.7}$$

リセットゲートは、$\mathbf{h}_{t-1}$ のうちどれだけを新しい**候補状態**（candidate state）の計算に通すかをフィルタリングする。

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{W}_{\text{hh}}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_{\text{h}}\right). \tag{12.8}$$

続いて、更新ゲート $\mathbf{z}_t$ が古い状態と候補状態の間を凸補間する。

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t. \tag{12.9}$$

Equation 12.9 は、忘却ゲートと入力ゲートを連動させた（$f_t = z_t, i_t = 1 - z_t$）ゲート付き和 Equation 12.3 にほかならない。各座標は $[-1, 1]$ 内の 2 つの値を凸結合するため、$\mathbf{h}_t$ は $[-1, 1]$ 内に有界に保たれ、追加の出力 tanh を必要としない（Figure 12.2）。また、GRU 層が必要とするリカレント・入力重みは $6d^2$ 個であり、LSTM の $8d^2$ 個よりも少ない。（ライブラリの融合版 GRU では、すべてのリカレント行列積を 1 回の呼び出しにバッチ化できるよう、$\mathbf{r}_t$ を $\mathbf{W}_{\text{hh}}$ との積の**後**に適用する。演習問題を参照のこと。）

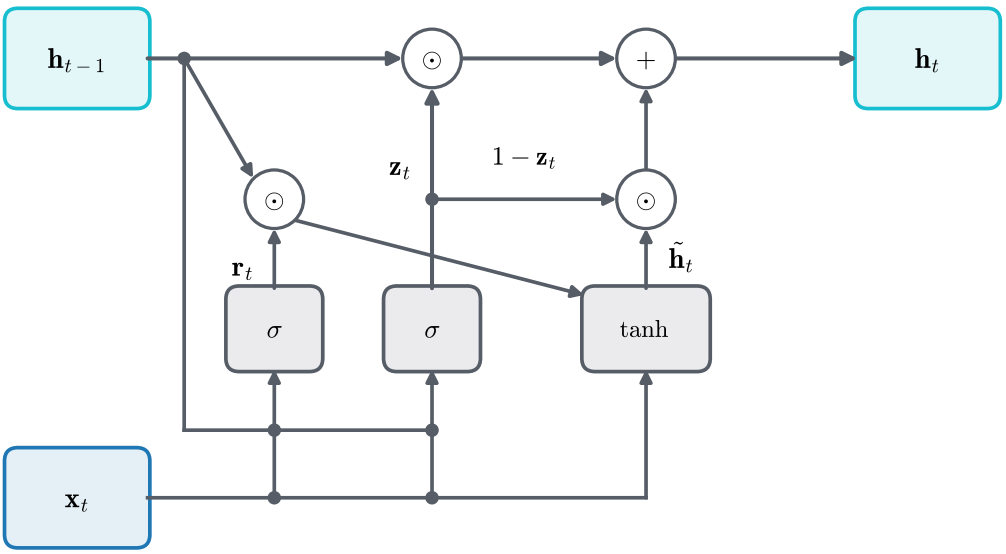


Figure 12.2: GRU セル。リセットゲートは候補 $\tilde{\mathbf{h}}_t$ の内部で前の状態をフィルタリングし、更新ゲートは古い状態と候補を混合する。

12.1.4 ゲート付きネットワークの訓練

12.1.4.1 遅延想起

ゲート付きの数式は数百ステップ持続する記憶を**可能にする**が、勾配降下法は実際にそれを見つけ出せるのだろうか。記憶の保持能力を合成データの**遅延想起**タスクで直接検証しよう。系列は8種類のターゲット記号（0〜7）のいずれか1つから始まり、その後に別の8種類の記号（8〜15）からランダムに選ばれた $T-1$ 個のフィラートークンが続き、ステップ T においてモデルは最初の記号を分類しなければならない（ランダム予測の正解率は $1/8 = 0.125$）。

```
def delayed_recall(rng, batch_size, num_steps, num_symbols=8,
                   blank_fillers=False):
    """A symbol, then num_steps - 1 fillers; the target is the symbol."""
    Y = rng.integers(0, num_symbols, (batch_size, 1))  # One query, at the end
    X = rng.integers(num_symbols, 2 * num_symbols, (batch_size, num_steps))
    if blank_fillers:  # The same filler token at every step
        X[:] = num_symbols
    X[:, :1] = Y
    return X, Y

X, Y = delayed_recall(np.random.default_rng(0), 2, 6)
print(X, Y)
```

```
[[ 6 10 10  8  8  8]
 [ 5 14 13 15 12 12]] [[6]
 [5]]
```

評価モデル（`RecallModel`）はトークンを32次元に埋め込み、幅64の単一のリカレント層に通して、最終ステップで線形分類器を適用する。本章を通じて、線形層・畳み込み層・リカレント層のすべての重み行列はLeCun正規初期化（分散 $1/n_{\text{in}}$）とゼロバイアスで初期化する（4.4節）。

`init_lecun` は線形・畳み込み・リカレントの各モジュールにLeCun正規重みとゼロバイアスを適用し、`Recurrent` はPyTorchのリカレントモジュールをラップして出力系列テンソルのみを返す。

```
def init_lecun(module):
    """LeCun normal weights and zero biases for all linear, convolutional,
    and recurrent layers."""
    for layer in module.modules():
        if isinstance(layer, (nn.Linear, nn.Conv1d)):
            fan_in = layer.weight[0].numel()  # Inputs of one output unit
            nn.init.normal_(layer.weight, std=fan_in ** -0.5)
            if layer.bias is not None:
                nn.init.zeros_(layer.bias)
        elif isinstance(layer, nn.RNNBase):  # Input and recurrent matrices
            for name, param in layer.named_parameters():
                if name.startswith('weight'):
                    nn.init.normal_(param, std=param.shape[1] ** -0.5)
                else:
                    nn.init.zeros_(param)
    return module

class Recurrent(nn.Module):
    """A batch-first recurrent layer that returns only its outputs."""
```

```
    def __init__(self, rnn):
        super().__init__()
        self.rnn = rnn

    def forward(self, X):
        return self.rnn(X)[0]
```

ここでは、6 つのリカレント構成を 3 組のペアとして比較する。

1. **素の RNN**：LeCun 正規初期化のリカレント重み（$\sigma = 1/\sqrt{64} = 1/8$）を用いるものと、対照群として $[-1/8, 1/8]$ 上の一様重み（分散は LeCun 正規初期化の $1/3$）を用いるものである。
2. **デフォルトの LSTM と GRU**：ゲートバイアスをゼロとするため、すべてのゲートが $\sigma(0) = 1/2$（半減期 1 ステップ）の近くから始まる。
3. **保持ゲート付き LSTM と GRU（`init_keep_gate`）**：LSTM の忘却ゲートバイアスと GRU の更新ゲートバイアスを $+5$（$\sigma(5) = 0.993$、半減期 ≈ 100 ステップ）に初期化し、LSTM の入力ゲートバイアスを -5（$\sigma(-5) = 1 - \sigma(5) = 0.007$）に設定する。$b_{\mathrm{i}} = -b_{\mathrm{f}}$ と対にすることで、長いフィラー列にわたって LSTM セルの大きさが $\sigma(-5)/(1 - \sigma(5)) = 1$ 以下に抑えられ、出力の tanh が決して飽和しなくなる。これは GRU に組み込まれた $1 - \mathbf{z}_t$ の連動に対応する（Jozefowicz et al. (2015) および Tallec and Ollivier (2018) の**クロノ初期化**（chrono initialization）を参照）。

一様重みの対照群は、PyTorch の `nn.RNN` におけるデフォルトの初期化と一致する。

```
class RecallModel(nn.Module):
    """Embedding, a sequence layer, and a classifier at every position."""
    def __init__(self, layer, num_hiddens, vocab_size=16, num_inputs=32,
                 num_outputs=8):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, num_inputs)
        nn.init.normal_(self.embedding.weight, std=num_inputs ** -0.5)
        self.layer = layer
        self.head = init_lecun(nn.Linear(num_hiddens, num_outputs))

    def forward(self, X):  # Logits, (batch_size, num_steps, num_outputs)
        return self.head(self.layer(self.embedding(X)))

def init_keep_gate(rnn, bias=5.0):
    """Start an LSTM's forget gate, or a GRU's update gate, near one."""
    h = rnn.hidden_size
    with torch.no_grad():
        rnn.bias_ih_l0.zero_()  # Independent of the earlier initialization
        rnn.bias_hh_l0.zero_()
        rnn.bias_ih_l0[h:2 * h] = bias  # LSTM: i, f, c~, o; GRU: r, z, n
        if isinstance(rnn, nn.LSTM):
            rnn.bias_ih_l0[:h] = -bias  # Input gate near zero
    return rnn
```

デフォルトのゼロバイアスをもつゲート付きネットワークについて、単純な算術が何を予測するかに注目しよう。すべてのゲートが $1/2$ から始まるとき、ステップ 1 とステップ T を結ぶ直接経路は順逆両方向で $2^{-(T-1)}$ 倍に減衰する。これは約 10^{-6}（$T = 20$ のときで、float32 でもまだ解像可能）であるが、10^{-15}（$T = 50$ のとき）にまで落ち込む。ステップ T の状態は、直近

のステップからの $O(1)$ の寄与に最初の記号の寄与を足し合わせるため、float32 の相対精度である $\sim 10^{-7}$ を下回る成分は読み出し層に届く前に丸め誤差で消えてしまい、記号の埋め込みへ戻る勾配も同様に小さくなる。したがって、デフォルトの LSTM と GRU は $T = 20$ では成功し $T = 50$ までには失敗する一方、保持ゲート版は $T = 200$ までのすべての遅延で成功すると予想される。

訓練用のヘルパー関数である `train_queries`、`query_accuracy`、`train_delayed_recall` を定義して保存し、本章のすべての合成想起タスクで再利用する。

```
def train_queries(model, batch, num_steps, optimizer,
                  scheduler=None):
    """Train on fresh batches; the queries are the last Y.shape[1] steps."""
    device = next(model.parameters()).device
    for step in range(num_steps):
        X, Y = (torch.as_tensor(A, device=device) for A in batch(step))
        logits = model(X)[:, -Y.shape[1]:]  # Predictions at the queries
        loss = F.cross_entropy(logits.flatten(0, 1), Y.flatten())
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        if scheduler is not None:
            scheduler.step()

def query_accuracy(model, X, Y):
    """Fraction of queries answered correctly."""
    device = next(model.parameters()).device
    X, Y = (torch.as_tensor(A, device=device) for A in (X, Y))
    with torch.inference_mode():
        return (model(X)[:, -Y.shape[1]:].argmax(-1) == Y).float().mean()

def train_delayed_recall(model, num_steps, blank_fillers=False):
    """Train with Adam at rate 0.003 on 1,500 fresh batches of 128 sequences
    of delayed recall; return the accuracy on 2,048 new sequences."""
    task = lambda rng, n: delayed_recall(rng, n, num_steps,
                                         blank_fillers=blank_fillers)
    rng = np.random.default_rng(0)
    model.to(d2l.try_gpu())
    train_queries(model, lambda step: task(rng, 128), 1500,
                  torch.optim.Adam(model.parameters(), lr=3e-3))
    return float(query_accuracy(model, *task(np.random.default_rng(1), 2048)))
```

訓練の前に、64×64 の素の RNN における 2 通りの初期化について、スペクトル半径と最大特異値も確認しておこう。

```
rng, n = np.random.default_rng(0), 64  # n hidden units
for name, W in (('LeCun normal', rng.normal(0, n ** -0.5, (n, n))),
                ('default scale', rng.uniform(-n ** -0.5, n ** -0.5, (n, n)))):
    print(f'{name:>13}: spectral radius {np.abs(np.linalg.eigvals(W)).max():.2f},'
          f' largest singular value {np.linalg.norm(W, 2):.2f}')
```

```
 LeCun normal: spectral radius 1.02, largest singular value 2.01
default scale: spectral radius 0.62, largest singular value 1.10
```

LeCun 正規初期化の行列はスペクトル半径が 1.02 であり（状態が小さいときに主固有モードが消失も爆発もしない $\rho \approx 1$ の境界線上にちょうど位置する）、一方で一様分布によるデフォルトスケールの行列はスペクトル半径が 0.62 である（1 ステップあたり ~ 0.6 倍に状態を縮小させる）。それでは、全 6 モデルのそれぞれについて 2 つの乱数シードを用い、遅延 $T \in \{20, 50, 100, 200\}$ にわたって訓練する。

```
def recurrent(layer, keep_gate=False):
    """A batch-first layer, 32 inputs to 64 hidden units, LeCun-initialized."""
    layer = init_lecun(layer(32, 64, batch_first=True))
    return init_keep_gate(layer) if keep_gate else layer

def rnn_default_scale():  # The recurrent matrix as nn.RNN draws it
    rnn = recurrent(nn.RNN)
    nn.init.uniform_(rnn.weight_hh_l0, -64 ** -0.5, 64 ** -0.5)
    return rnn

layers = {'RNN': lambda: recurrent(nn.RNN),
          'RNN, default scale': rnn_default_scale,
          'LSTM': lambda: recurrent(nn.LSTM), 'GRU': lambda: recurrent(nn.GRU),
          'LSTM, keep gate': lambda: recurrent(nn.LSTM, keep_gate=True),
          'GRU, keep gate': lambda: recurrent(nn.GRU, keep_gate=True)}
delays = [20, 50, 100, 200]
print(' ' * 18, ' '.join(f'{T:>9d}' for T in delays))
for name, layer in layers.items():
    runs = [[train_delayed_recall(RecallModel(Recurrent(layer()), 64), T)
             for _ in range(2)] for T in delays]
    print(f'{name:18s}', ' '.join(f'{a:.2f}/{b:.2f}' for a, b in runs))
```

```
                       20        50       100       200
RNN                1.00/1.00 0.25/0.26 1.00/0.35 0.23/0.24
RNN, default scale 1.00/1.00 1.00/0.12 0.12/0.12 0.12/0.13
LSTM               0.61/0.22 0.12/0.12 0.12/0.13 0.12/0.12
GRU                1.00/1.00 0.12/0.12 0.13/0.13 0.13/0.13
LSTM, keep gate    1.00/1.00 1.00/1.00 1.00/1.00 1.00/1.00
GRU, keep gate     1.00/1.00 1.00/1.00 1.00/1.00 1.00/1.00
```

この結果は、GRU および $T \geq 50$ のすべての遅延については上の解析を裏付ける一方、LSTM については解析を修正するものとなっている。

- **デフォルト対保持ゲート付きの LSTM と GRU：**ゲートバイアスがゼロのとき、GRU は $T = 20$ で成功し、両セルともランダム予測の水準（0.12）へと $T \geq 50$ で崩壊する。しかしデフォルトの LSTM は、算術的な見積もりでは成功するはずだった $T = 20$ の時点ですでに不安定である。あの見積もりは忘却ゲートしか考慮していなかった。原因を完全には切り分けていないが、おそらく LSTM では記号が半分だけ開いた入力ゲートを通ってセルに入り、さらに半分だけ開いた出力ゲートと tanh を通って外へ出る必要があるため、$T = 20$ が訓練で回復できる限界付近になってしまうのだろう。`init_keep_gate` を用いれば、両者とも 1.00 の正解率を $T = 200$ までのすべての遅延・すべての試行で達成する。
- **境界線上と縮小領域の素の RNN：**LeCun 正規初期化の RNN はスペクトル半径 $\rho \approx 1$ から始まるため、シード間のばらつきは大きいものの一部のシードでは驚くほど遠く（$T = 100$ まで）記号を運ぶ。これに対し、$\rho \approx 0.62$ のデフォルトスケール RNN は $T = 50$ でも成否が安定せず、$T = 100$ 以降はすべてのシードで失敗する。適切にバイアスを初期化したゲートは、運任せのスペクトル保持を信頼性の高いユニットごとの記憶制御へと変える。

12.1.4.2 テキストにおけるゲート付きネットワーク

11.2.3.2 節では、幅の広い単層の素の RNN はコンテキストの最初の数バイトを過ぎると複数ブロックの GPT に遅れをとり、そのまま追いつけなかった。再帰構造そのものの違いを残差接続の深さや位置ごとの MLP と混同せずに系列ミキサを対等な条件で比較するため、本章ではすべてのミキサを 11.1 節に合わせた同一の Pre-Norm 残差バックボーン（`SeqLM`）の内部に配置する。具体的には、トークン埋め込みに続き、`H = H + mixer(norm(H))` とその後の 2 層 GELU MLP `H = H + mlp(H)` を各ブロックで適用する積層構造をとり、最後に層正規化と線形出力ヘッドを置く（Figure 12.3）。

```
class SeqLM(nn.Module):
    """A language model of residual blocks: a sequence mixer, then an MLP."""
    def __init__(self, vocab_size, num_hiddens, mixers):
        super().__init__()
        d = num_hiddens
        self.embedding = nn.Embedding(vocab_size, d)
        nn.init.normal_(self.embedding.weight, std=d ** -0.5)  # Unit length
        self.mixers = nn.ModuleList(mixers)
        self.norms = nn.ModuleList([nn.LayerNorm(d) for _ in mixers])
        self.mlps = nn.ModuleList([nn.Sequential(
            nn.LayerNorm(d), nn.Linear(d, 4 * d), nn.GELU(),
            nn.Linear(4 * d, d)) for _ in mixers])
        self.norm = nn.LayerNorm(d)
        self.head = nn.Linear(d, vocab_size)
        for module in (self.mlps, self.head):  # Mixers initialize themselves
            init_lecun(module)

    def forward(self, X):
        H = self.embedding(X)  # (batch_size, num_steps, num_hiddens)
        for norm, mixer, mlp in zip(self.norms, self.mixers, self.mlps):
            H = H + mixer(norm(H))
            H = H + mlp(H)
        return self.head(self.norm(H))  # (batch_size, num_steps, vocab_size)
```

`SeqLM` 内のアテンションベースラインは `AttentionMixer` であり、これは RoPE 付きの因果マルチヘッドアテンション（11.2 節、10.4 節）、LeCun 正規初期化、およびオプションとして $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ に対するデプスワイズ因果畳み込み（後の 12.8 節で使用する）をまとめたものである。

```
class AttentionMixer(d2l.CausalAttention):
    """Causal multi-head attention, with rotary positions unless rope is
    False, as a sequence mixer; with conv_width, a causal convolution per
    channel first mixes each query, key, and value with those of the
    previous steps."""
    def __init__(self, num_hiddens, num_heads, conv_width=None, rope=True):
        super().__init__(num_hiddens, num_heads, rope=rope)
        d = 3 * num_hiddens  # Channels of q, k, and v; one filter each
        self.conv = None if conv_width is None else nn.Conv1d(
            d, d, conv_width, groups=d, padding=conv_width - 1)
        init_lecun(self)  # Like every mixer of the chapter

    def qkv(self, X, offset=0, tables=None):
        B, T, _ = X.shape
```

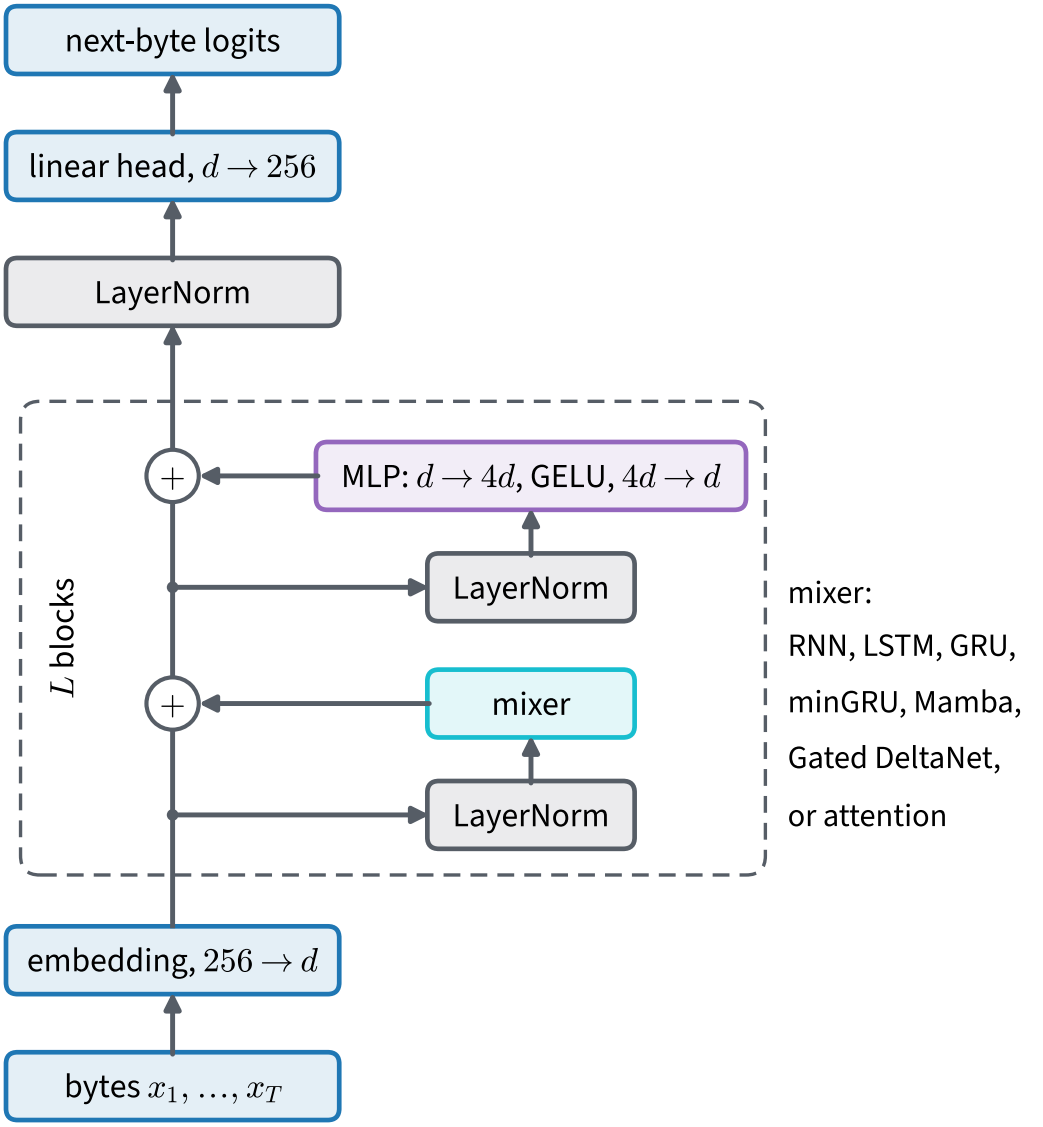


Figure 12.3: 共通バックボーン `SeqLM`。各ブロックは、それぞれ独自の層正規化を経たミキサの出力、続いて MLP の出力を残差ストリームに加算する。本章のモデル間で変わるのはミキサのみであり、言語モデルは $L = 4$ 個のブロックを積み重ねる。

```
        QKV = self.W_qkv(X)
        if self.conv is not None:  # Causal: drop the last conv_width - 1 steps
            QKV = self.conv(QKV.mT)[..., :T].mT
        q, k, v = (u.reshape(B, T, self.num_heads, -1).transpose(1, 2)
                   for u in QKV.chunk(3, -1))
        if self.rope:
            q, k = (d2l.rope(u, offset, tables=tables) for u in (q, k))
        return q, k, v  # Each (batch_size, num_heads, num_steps, head width)
```

また、本章全体で使用する 3 つの評価用ヘルパー関数も保存しておく。まず、`loss_by_position` はコンテキストウィンドウの各位置における検証データの平均交差エントロピー（単位は nat）を計算する。

```
@torch.inference_mode()
def loss_by_position(model, data):
    """Mean validation loss in nats at each position of the window."""
    device = next(model.parameters()).device
    total, count = 0, 0
    for X, Y in data.val_dataloader():
        logits = model(X.to(device))  # (batch, T, V)
        total += F.cross_entropy(logits.transpose(1, 2), Y.to(device),
                                 reduction='none').sum(0)
        count += len(X)
    return (total / count).tolist()
```

次に、`train_text` は AdamW でモデルを 1,000 ステップ訓練し（100 ステップかけて 2×10^{-3} までウォームアップしたのちコサイン減衰を適用）、`text_table` は各構成につき 2 つのシードを実行して、パラメータ数、全体および位置帯域別（`0-15`、`16-63`、`64-127`）の検証損失、ならびに 1 ステップあたりのミリ秒数を出力する。

```
def train_text(model, data, lr=2e-3, num_steps=1000):
    """Train a language model with LMOptimizer; return its validation loss
    in nats at each position and its milliseconds per training step."""
    optimizer = d2l.LMOptimizer(model, lr=lr, num_steps=num_steps, warmup=100)
    # One untimed call of 50 steps warms up (JAX compiles), the rest is timed
    bench = d2l.Benchmark(lambda: d2l.train_steps(model, data, optimizer, 50),
                          warmup=1, repeats=num_steps // 50 - 1)
    return loss_by_position(model, data), 1000 * bench.time / 50

def text_table(make_model, arms, data, num_runs=2, num_steps=1000):
    """Train num_runs fresh models make_model(*args) for each arm (a name
    and its args) for num_steps steps; print the parameters, the validation loss of every run,
    its losses at early, middle, and late positions of the window, and the
    milliseconds per step. Return the trained models of each arm."""
    T = data.num_steps
    bands = [(0, T // 8), (T // 8, T // 2), (T // 2, T)]
    width = max(len(name) for name in arms)
    header = ['validation'] + [f'{a}-{b - 1}' for a, b in bands]
    print(' ' * (width + 12), *(f'{h:>11}' for h in header), '  ms/step')
    trained = {}
    for name, args in arms.items():
```

```
            models = [make_model(*args) for _ in range(num_runs)]
            runs = [train_text(model, data, num_steps=num_steps)  # (losses, ms)
                    for model in models]
            columns = [[np.mean(loss) for loss, _ in runs]] + [
                [np.mean(loss[a:b]) for loss, _ in runs] for a, b in bands]
            print(f'{name:{width}s} {d2l.num_params(models[0]):11,d}',
                  *('/'.join(f'{v:.3f}' for v in column) for column in columns),
                  '/'.join(f'{ms:.0f}' for _, ms in runs))
            trained[name] = models
        return trained
```

ここで、リカレントミキサとアテンションミキサをバイトレベルの WikiText-103（`d2l.WikiText103`、ウィンドウ長 128 バイト、バッチサイズ 64）でベンチマークする。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
```

各 `text_model` は幅 $d = 256$ の残差ブロックを 4 つ積み重ね、素の `RNN`、`LSTM`、`GRU`、8 ヘッドの因果 `attention` に加え、パラメータ数を揃えるために幅を $d = 302$ に広げた素の RNN（パラメータ数 3.81M で、GRU の積層モデルと同数）を比較する。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32

def text_model(mixer, d):  # 256 byte values; four blocks of width d
    return SeqLM(256, d, [mixer(d) for _ in range(4)]).to(d2l.try_gpu())

rnn = lambda d: Recurrent(init_lecun(nn.RNN(d, d, batch_first=True)))
lstm = lambda d: Recurrent(init_lecun(nn.LSTM(d, d, batch_first=True)))
gru = lambda d: Recurrent(init_lecun(nn.GRU(d, d, batch_first=True)))
attention = lambda d: AttentionMixer(d, num_heads=8)
arms = {'RNN': (rnn, 256), 'LSTM': (lstm, 256), 'GRU': (gru, 256),
        'attention': (attention, 256),
        'RNN, d=302': (rnn, 302)}  # As many parameters as the GRU
```

`text_table` を呼び出すと、各アームにつき 2 回ずつ訓練を実行し、横並びの比較表を出力する。

```
trained = text_table(text_model, arms, data)
```

```
                       validation        0-15       16-63      64-127   ms/step
RNN           2,764,544 1.347/1.349 1.510/1.512 1.324/1.325 1.324/1.325 13/13
LSTM          4,343,552 1.292/1.293 1.488/1.490 1.266/1.267 1.263/1.264 32/38
GRU           3,817,216 1.297/1.299 1.490/1.492 1.271/1.273 1.269/1.271 34/38
attention     3,286,784 1.378/1.375 1.560/1.556 1.354/1.350 1.351/1.349 20/21
RNN, d=302    3,816,932 1.334/1.334 1.501/1.501 1.310/1.311 1.309/1.310 21/21
```

この表からは 3 つの知見が浮かび上がる。

1. **残差積層＋ゲート機構：**幅の狭い 4 つの tanh RNN 層を Pre-Norm 残差＋ MLP バックボーンの内部に配置するだけで ~ 1.35 nat/byte に到達し、11.2 節の 6 ブロック SwiGLU GPT にすでに肩を並べる。素の RNN ミキサを LSTM や GRU に置き換えると検証損失はさらに 0.05 nat/byte 下がって ~ 1.30 になるのに対し、素の RNN の幅を広げて GRU とパラメータ数を揃えても（`RNN, d=302`）、その差の 1/3 未満しか縮まらない（約 1.33）。

2. **短いコンテキストの活用：**この 1,000 ステップ・128 バイトの設定では、どのモデルでも損失は帯域 `0-15` から `16-63` にかけて急減し、16 バイト目以降の変化は 0.005 未満にとどまる。ここでゲートが優位に立っているのは、100 ステップ先の記憶を運んでいるからではなく、短距離のバイト文脈をより効果的に管理しているからである。この小さな訓練予算では、アテンションの積層モデル（約 1.38）は最初の 16 バイトの時点ですでにリカレントミキサに遅れをとっている。
3. **ステップあたりのコスト：**融合されたライブラリのループを用いているにもかかわらず、LSTM と GRU は訓練 1 ステップあたり素の RNN より $1.4\times$〜$3\times$ 長い時間を要する。原因は各ステップ内部の計算量にあり、素の RNN が 1 回の積で済むところで 3〜4 回のゲート積を計算するためである。素の RNN も同じように前ステップの隠れ状態 $\mathbf{h}_{t-1}$ を待つため、逐次評価そのものが原因ではない。128 ステップではその逐次依存はまだ支配的なコストになっておらず、この表からは読み取れない。そのコストを切り分けるのが 12.3.4 節の系列長スイープであり、そこでは 512 トークンから 16,384 トークンの間に GRU のステップ時間が 15 倍以上に増大する。

12.1.5 まとめ

素のリカレントネットワークでは、信号と勾配は単一の共有行列 $\mathbf{W}_{\mathrm{hh}}$ に支配されるステップごとのヤコビ行列の積を通過するため、そのスペクトル半径が 1 の近くにあるときに限り長いホライズンにわたって生き残る。ゲート付きリカレントネットワークは、学習されるシグモイドゲートで制御される要素ごとの加法的な記憶経路 $c_t = f_t c_{t-1} + i_t \tilde{c}_t$ を導入する。LSTM は（入力ゲート、忘却ゲート、出力ゲートに守られた）内部の記憶セル $\mathbf{c}_t$ を公開状態 $\mathbf{h}_t$ から分離する一方、GRU は単一の状態ベクトル上で忘却ゲートと入力ゲートを連動させ（$1 - \mathbf{z}_t$）、リセットゲート $\mathbf{r}_t$ によって再帰状態をフィルタリングする。合成データの遅延想起では、忘却ゲート・更新ゲートを 1 の近くに初期化すること（`init_keep_gate`）が決定的に重要であり、$T \geq 50$ でのランダム予測水準の正解率を 100% の想起正解率（$T = 200$）へと変える。残差バックボーン（`SeqLM`）内部でのバイトレベル WikiText-103 では、LSTM ミキサと GRU ミキサは同一幅およびパラメータ数を揃えた素の RNN の両方を 0.03〜0.05 nat/byte 上回るが、ゲート内部で $\mathbf{h}_{t-1}$ に依存するため、訓練時には依然として T 回の逐次ステップを要する。

12.1.6 演習問題

1. **値の保持。**ステップ s の文字がずっと後のステップ t における予測を決定しており、1 つの LSTM ユニットがその情報を s から t まで運ぶものとする。
   1. Equation 12.5 におけるそのユニットの各ゲートが、ステップ s、s と t の真に間の各ステップ、およびステップ t においてとるべき値を答えよ。次に、Equation 12.9 における GRU ユニットの更新ゲートについても同様に答えよ。
   2. シグモイドゲートは決して 1 に到達しない。忘却ゲートの活性化前の値が定数 b に保たれ、毎ステップその値が $\sigma(b)$ であるとする。直接経路に沿って k ステップ後に生き残る記憶値の割合はいくらか。また、1,000 ステップ後に値の 90% が生き残るためには、b はどの程度大きくなければならないか。
   3. 重みがゼロのとき、忘却ゲートバイアスを $\log(T-1)$ とすると初期の忘却ゲートが $1 - 1/T$ となり、したがって大きい T に対して記憶された値が約 e 分の 1 に T ステップで減衰することを示せ。Tallec and Ollivier (2018) の**クロノ初期化**は、各ユニットの忘却ゲートバイアスを $\log u$（ただし u は $[1, T_{\max} - 1]$ 上の一様分布に従う）として抽出し、$\mathbf{b}_{\mathrm{i}} = -\mathbf{b}_{\mathrm{f}}$ と設定する。重みがゼロのとき、これにより $\mathbf{f}_t + \mathbf{i}_t = 1$ となることを示し、得られるセル更新式を Equation 12.9 と比較せよ。
2. **出力の圧縮。**入力ノード $\tilde{\mathbf{c}}_t$ はすでに tanh で圧縮されているにもかかわらず、Equation 12.6 は出力ゲートの前で tanh を $\mathbf{c}_t$ に再び適用している。
   1. $\mathbf{c}_0 = \mathbf{0}$ から始めたとき、$\mathbf{c}_t$ のすべての要素の絶対値が t より小さいことを示せ。さら

に、すべての忘却ゲートの値が高々 $f_{\max} < 1$ であり、すべての入力ゲートの値が高々 $i_{\max}$ であるならば、すべての要素の絶対値が $i_{\max}/(1 - f_{\max})$ 未満にとどまる（t がいくらであっても成り立つ）ことを示せ。

2. 仮に出力が $\mathbf{h}_t = \mathbf{o}_t \odot \mathbf{c}_t$ であったとしよう。多数のステップにわたって値が蓄積された後、次ステップにおける各ゲートの活性化前の値はどうなるか。また、それはゲートを通過する勾配にどのような影響を与えるか。

3. **リセットの前後**。深層学習フレームワークの融合版 GRU 層は、リカレント行列積の後にリセットゲートを適用する。バイアス項の配置を除けば、候補を次の式で計算する。

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{r}_t \odot (\mathbf{W}_{\text{hh}}\mathbf{h}_{t-1}) + \mathbf{b}_{\text{h}})$$

これを Equation 12.8 の代わりに用いる。

1. 2 つの候補が任意の $\mathbf{r}_t$ と $\mathbf{h}_{t-1}$ に対して一致するのは、$\mathbf{W}_{\text{hh}}$ が対角行列であるとき、かつそのときに限ることを示せ。一般の $\mathbf{W}_{\text{hh}}$ に対して、それでも両者を一致させるリセットゲートの値は何か。
2. 全ステップの入力積 $\mathbf{W}_{\text{x}}.\mathbf{x}_t$ は、再帰が始まる前に 1 つの大きな積としてまとめて計算できる。それぞれの候補の形式のもとで、前の状態を含む行列積を各ステップで逐次的に何回実行しなければならないか。なぜ融合 GRU カーネルは上の形式を採用するのか。

4. [code] **スクラッチからの GRU**。クラス `GRUScratch` を `LSTMScratch` の編集によって実装せよ。具体的には、Equation 12.7 と Equation 12.8 に従う 3 つのヘッド、更新式 Equation 12.9、および $\mathbf{h}_t$ のみからなる状態をもたせよ。

1. LSTM で行ったのと同様に、ライブラリの GRU 層の重みを自作の層にコピーして出力を比較せよ。両者は一致しないはずである。前の演習問題を利用して自作の層を 1 行変更し、出力が一致するようにせよ。（また、候補のリカレントバイアスである `bias_hh_l0` の最後の 1/3 をリセットゲートの内側にある独立したベクトルとして保持するか、コピー前にゼロにしておくこと。）
2. $\mathbf{r}_t = \mathbf{1}$ に固定してリセットゲートを除去した層と、別の層として $\mathbf{z}_t = \mathbf{0}$ に固定して更新ゲートを除去した層を作成せよ。訓練の前に、どちらの除去が遅延想起をより大きく損なうかを Equation 12.9 から予測せよ。続いて 12.1.4.1 節の実験を実行し、予測を確かめよ。

5. [code] **セルの他の経路**。●入力が 1 つ、隠れユニットが 1 つの LSTM を考え、Equation 12.5 と Equation 12.6 のすべての量がスカラーであるとする。忘却ゲート、入力ゲート、入力ノードのリカレント重みをそれぞれ w_{hf}、w_{hi}、w_{hc} と書く。

1. 出力ゲート o_{t-1} は h_{t-2} に依存するが、c_{t-1} には依存しない。$h_{t-1} = o_{t-1}\tanh(c_{t-1})$ を用いて、全微分 $\mathrm{d}c_t/\mathrm{d}c_{t-1}$ を導出せよ。これを直接経路の値 f_t と補正項の和として書き、さらに補正項をステップ t 由来の項と導関数 $\mathrm{d}h_{t-1}/\mathrm{d}c_{t-1}$ の積に因数分解せよ。
2. ゲート、$\tilde{c}_t$、および c_{t-1} がどのような値のときにこの補正項はゼロになるか。セルが値を保持している最中に、ネットワークがゲートを通じて設定できるのはこれらの条件のうちどれか。
3. LSTM が成功する遅延長で遅延想起タスクを訓練せよ。1 つのテスト系列について、セル $\mathbf{c}_{T-k}$ に関する損失の勾配のノルムをすべての k に対して自動微分により計算し、それを直接経路のみが運ぶ勾配（$\mathbf{c}_T$ における勾配に、直近 k ステップの忘却ゲート値の積を要素ごとに掛けたもの）と比較せよ。2 つの曲線はどこで乖離し、どちらが大きくなるか。

6. [code] **双方向のリーク**。●リカレント層は系列を逆向きに読み取ることもできる。**双方向**ネットワーク (Schuster and Paliwal 1997) は、最後のステップから最初のステップへ向かって 2 つ目の層を走らせ、2 つの層の出力を連結することで、各位置の表現が系列全体に依存するようにする。これは、文中の単語へのタグ付けや翻訳のための文の符号化のように、入力全体があらかじめ与えられている場合には常に有効である。しかし、言語モデルとなると

話は別である。

1. 反転させた系列に対して 2 つ目の LSTM を走らせ、その出力を再反転して順方向 LSTM の出力と連結し、結果を幅 d に射影して戻す双方向ミキサを構築せよ。12.1.4.2 節の LSTM をこのミキサに置き換えた `SeqLM` を `train_text` で訓練し、その検証損失を順方向モデルの検証損失と比較せよ。得られた数値を説明せよ。逆方向層は位置 t でどの入力を読み取っているか。また、後続のバイトがまだ存在しない生成時には、このモデルは何ができるだろうか。
2. バイト x_t のリークのない双方向予測器は、ステップ $t-1$ 後の順方向状態とステップ $t+1$ 後の逆方向状態を結合することで、ウィンドウ内の他のすべてのバイトが与えられたもとでの x_t の確率をモデル化する。2 つの 2 値トークンからなる系列を用いて、これらの条件付き確率の t にわたる積を全系列について足し合わせても 1 になるとは限らないことを示せ。これは、その積を用いて系列全体にスコアを付けたり、パープレキシティを報告したりすることに対して何を意味するか。

12.2 系列対系列学習

前章までの言語モデルは、テキストを読み取って次のトークンを予測することで単一の系列を継続するものであった。しかし多くのタスクでは、ある系列を長さの異なる別の系列へと写像し、両者の位置は 1 対 1 に対応しない。たとえば、翻訳器が英語の文を読み取ってフランス語の文を書き出したり、音声認識器が音声クリップを単語に書き起こしたりする場合がそうである。翻訳は、2014 年にニューラルネットワークが初めて統計的フレーズベースシステムに肩を並べたベンチマークであった (Sutskever et al. 2014; Cho, Van Merriënboer, Gulcehre, et al. 2014)。そのアーキテクチャは単純であり、1 つ目のリカレントネットワークがソースを読み取り、2 つ目がターゲットを書き出し、両者の間では単一の固定幅の状態ベクトルが受け渡される。

この設計の後継にあたるクロスアテンション（第 10 章）や Transformer エンコーダ・デコーダ（11.4 節）はすでに登場したが、これらはその単一ベクトルのボトルネックを回避するために考案されたものである。では、なぜここで 2014 年の設計に立ち返るのか。そのボトルネックこそが本章の中心テーマだからである。次節以降で扱うすべてのリカレントモデルは、過去の情報を固定サイズの状態へと圧縮する。本節では、エンコーダ・デコーダの抽象化と標準的な語彙ベースの翻訳評価指標（`chrF` および `BLEU`）を整備したうえで、文字列反転ベンチマークを用い、入力が長くなるにつれて単一の固定状態ベクトルがどれだけの情報を保持できるかを直接測定する。

```
import collections
```

12.2.1 エンコーダ・デコーダの抽象化

ある系列を可変長の別の系列へと写像する**系列対系列学習**（sequence-to-sequence learning）、すなわち**系列変換**（sequence transduction）は、自然に 2 つのモジュールへと分かれる（Figure 12.4）。ソース系列を受け取って 1 つの状態に要約する**エンコーダ**と、その状態を受け取ってターゲットを 1 トークンずつ生成する**デコーダ**である。デコーダがソースについて得る情報はすべて、その状態を通過しなければならない。

ソース $x_1, \ldots, x_T$ とターゲット $y_1, \ldots, y_{T'}$（ただし T' は T と等しくなくてよい）が与えられたとき、デコーダは言語モデル（8.3 節）と同様に条件付き確率を自己回帰的に分解するが、すべての因子がエンコーダの状態 $\mathbf{c}$ で条件付けられる点が異なる。

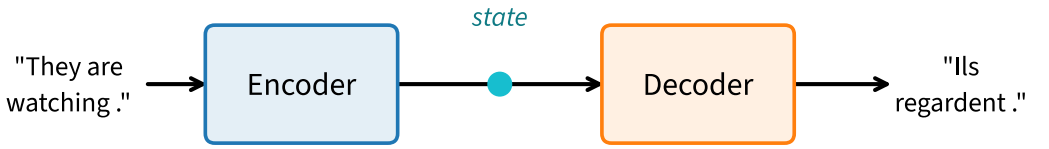


Figure 12.4: エンコーダ・デコーダアーキテクチャ。エンコーダがソース系列を読み取って状態に変換し、デコーダがその状態からターゲット系列を生成する。

$$P(y_1, \dots, y_{T'} \mid x_1, \dots, x_T) = \prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}). \tag{12.10}$$

右辺からはソース系列が消え、$\mathbf{c}$ だけが残っている点に注目しよう。この等式が成り立つのは、$\mathbf{c}$ がデコーダに必要となるソースの情報をすべて保持している場合に限られる。訓練時には教師強制（8.4.1.2 節）のもとで正解の直前ターゲットトークンを入力し、次トークンの交差エントロピーを最小化する。一方、推論時には、デコーダ自身が直前に出力したトークンを次の入力としてフィードバックし、系列終了トークン（end-of-sequence token）を生成するまでこれを繰り返す。

2014 年に 2 つの研究グループがこのリカレント設計を導入した。Cho, Van Merriënboer, Gulcehre, et al. (2014) は統計的翻訳システムの内部でフレーズ対のスコアを計算するために GRU エンコーダ・デコーダ（12.1.3 節）を用い、一方 Sutskever et al. (2014) は単一のベクトルで接続した深層 LSTM によって英語からフランス語への翻訳をエンドツーエンドで行った。特筆すべき工夫として、Sutskever らはソース文を**逆順**でエンコーダに入力し、翻訳の最初の数単語が対応するソース単語からわずか数リカレントステップの距離に置かれるようにした。これにより、状態が保持すべき総情報量は変わらないまま、勾配経路が短縮された。

リカレントなエンコーダとデコーダは後に Transformer（音声認識の Whisper (Radford et al. 2023) や視覚言語キャプション生成モデルなど）に取って代わられたが、エンコーダ・デコーダという抽象化そのものは生き残った。11.4 節で変わったのは、デコーダに渡される状態がソース全体に対する単一のベクトルではなく、ソーストークンごとに 1 つのベクトルを並べた系列になった点である。本節ではあえて 2014 年の単一ベクトルによる状態を維持する。その容量の限界こそが、ここで測定したい量だからである。

12.2.2 翻訳のスコアリング

人手による評価は遅すぎて訓練ループの内部では実行できないため、機械翻訳モデルは参照訳との語彙的な n グラムの重複によってスコア付けされる。2014 年の論文では BLEU が報告されていたが、現代の評価ベンチマークではその代わりに文字レベルの chrF が推奨されている (Popović 2015; Kocmi et al. 2021)。ここでは両方を実装し、18.4 節（英語版） でも再利用する。

chrF は、空白を除去した文字 n グラムを次数 $n = 1, \dots, 6$ にわたって比較するため、トークン化への依存を避けつつ、共通の語幹や惜しい語形変化にも部分的な評価を与えられる。各次数 n について、適合率（precision）は候補訳の n グラムのうち参照訳に含まれるものの割合であり、再現率（recall）は参照訳の n グラムのうち候補訳に含まれるものの割合である（多重集合の共通部分 `p & r` を用いることで、繰り返される n グラムは参照訳での出現回数でクリップされる）。6 つの次数にわたって適合率と再現率を平均して P と R を求め、chrF は次の F_β スコアを計算する。

$$F_\beta = \frac{(1+\beta^2)\,PR}{\beta^2 P + R}. \tag{12.11}$$

$\beta = 2$ とすると、適合率よりも再現率に大きな重みが置かれる。

```
def chrf(pred, label, n=6, beta=2):
    """chrF (Popovic, 2015): character n-gram F-score."""
    def ngrams(s, k):
        s = s.replace(' ', '')
        return collections.Counter(s[i:i + k] for i in range(len(s) - k + 1))
    prec = rec = 0.0
    for k in range(1, n + 1):
        p, r = ngrams(pred, k), ngrams(label, k)
        overlap = sum((p & r).values())
        if sum(p.values()) and sum(r.values()):
            prec += overlap / sum(p.values())
            rec += overlap / sum(r.values())
    prec, rec = prec / n, rec / n
    if prec + rec == 0:
        return 0.0
    return (1 + beta**2) * prec * rec / (beta**2 * prec + rec)
```

BLEU (Papineni et al. 2002) は、クリップされた単語 n グラム適合率 p_n（$n = 1, \ldots, k$）を測定し、短く適合率の高い断片がスコアを不正に稼げないよう、それらの重み付き幾何平均に**短文ペナルティ**（brevity penalty）を掛ける。

$$\text{BLEU} = \exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}. \tag{12.12}$$

この式は積の形をとるため、適合率 $p_n = 0$ となる次数 $n \le k$ が 1 つでもあると、文レベルの BLEU は 0 になる。（標準的なコーパスレベルの BLEU では、次数 $n \le 4$ に $1/4$ ずつ均等な重みを付け、テストセット全体でカウントを集計するため、カウントがゼロになることはまれである。）

```
def bleu(pred_seq, label_seq, k):
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

単一の文でこれら 2 つの評価指標がどのように食い違うかを見るため、完全なコピー、6 単語中 3 番目の単語の語形変化を 1 つだけ誤ったニアミス、および妥当な言い換えのスコアを計算してみよう。6 単語の文に含まれるすべての 4 グラムは 3 番目の単語を含むため、ニアミスでは $p_4 = 0$ となり、$k = 4$ の BLEU はゼロに落ち込む。一方、chrF が失うのは余分な **s** に重なる文字 n グラムだけである。

```
reference = 'we will visit our grandparents tomorrow'
candidates = {'copy': reference,
              'near miss': 'we will visits our grandparents tomorrow',
              'paraphrase': 'we are going to see our grandparents tomorrow'}
for name, candidate in candidates.items():
    print(f'{name:10s}  chrF {chrf(candidate, reference):.2f}  '
          f'BLEU {bleu(candidate, reference, k=4):.2f}')
```

```
copy        chrF 1.00  BLEU 1.00
near miss   chrF 0.91  BLEU 0.00
paraphrase  chrF 0.67  BLEU 0.00
```

ニアミスのスコアは chrF で 0.91、BLEU で 0.00 となる一方、意味を保った言い換えのスコアは chrF で 0.67、BLEU で 0.00 となる。どちらの評価指標も意味ではなく表層的な重複を数えるため、現代の評価では chrF（または過去の研究との比較にはコーパスレベルの sacreBLEU (Post 2018)）を、COMET (Rei et al. 2020) のような学習済みのニューラル評価器や LLM ジャッジと組み合わせて用いる。

12.2.3 固定状態のボトルネック

単一の状態ベクトルには、入力系列のどれだけの情報が収まるだろうか。自然言語では文の長さと構文的な希少性が交絡するため、ここでは 11.4.3.1 節の人工的な文字列反転ベンチマークを使って状態容量を切り出す。すなわち、16 個の記号からなるアルファベットから一様かつ独立に抽出された T 文字のランダム文字列を反転させるタスクである。各文字は独立であるため、すでにデコードされた文字は次の文字について何の情報も与えず、すべての文字がエンコーダの最終状態 **c** を経由しなければならない。また反転タスクでは、エンコーダが最後に読んだトークンをデコーダが最初に出力することになるため、Sutskever らの逆順入力の順序とも自然に一致する。

各ミニバッチでは、毎回新たなランダム文字列を生成する。`Reversal` は、ソース文字列と、先頭に開始トークン `V` を付加して右にシフトした反転ターゲットを、1 つのテンソル `X` にまとめて `d2l.train_steps` に渡す。

```
class Reversal:
    """Fresh batches of random strings and their reversals."""
    def __init__(self, length, batch_size=128, num_letters=16):
        self.length, self.batch_size = length, batch_size
        self.num_letters = num_letters

    def batch(self, batch_size):
        V = self.num_letters  # Letters 0 to V - 1; V is the start token
        src = torch.randint(0, V, (batch_size, self.length),
                            device=d2l.try_gpu())
        tgt = src.flip(1)
        dec_in = torch.cat([torch.full_like(tgt[:, :1], V), tgt[:, :-1]], 1)
        return torch.cat([src, dec_in], 1), tgt
```

```
    def train_dataloader(self):
        while True:
            yield self.batch(self.batch_size)
```

生のビット数で数えると、等確率な 16 種類の各文字は $\log_2 16 = 4$ ビットの情報を持つため、長さ T の文字列は $4T$ ビット（$T = 8$ で 32 ビット、$T = 64$ で 256 ビット）の情報を持つ。原理的には、64 個の単精度浮動小数点数は 2,048 ビットを占めるため、512 文字分の余裕がある。では、訓練された GRU は実際には 64 個または 256 個の浮動小数点数を通じて何文字を詰め込み、取り出せるのだろうか。

Figure 12.5 および Cho, Van Merriënboer, Gulcehre, et al. (2014) に従い、GRU エンコーダの最終状態 $\mathbf{c}$ は GRU デコーダを初期化する（$\mathbf{s}_0 = \mathbf{c}$）とともに、各デコーディングステップで入力埋め込みに連結される。

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{12.13}$$

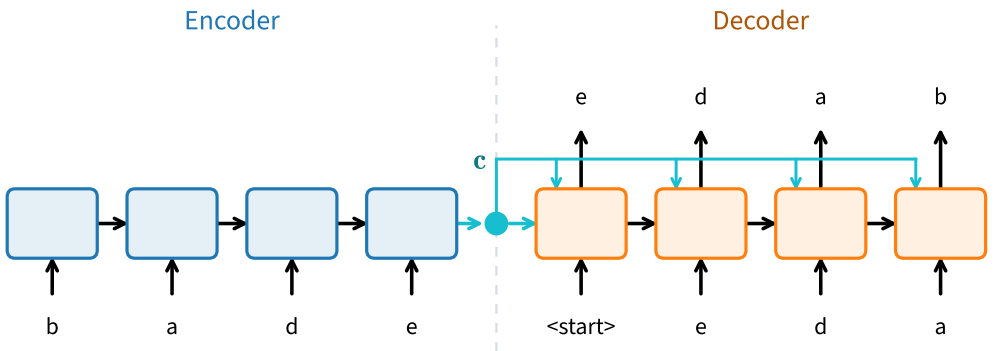


Figure 12.5: 反転タスクにおける GRU エンコーダ・デコーダを時間方向に展開した図。エンコーダはソースを読み取り、その最終状態 $\mathbf{c}$ がデコーダを初期化するとともにデコーダの各ステップに入力される。デコーダは開始トークンの後ろに右シフトされた反転文字列を読み取り、反転文字列を予測する。

```
class Reverser(nn.Module):
    """A GRU encoder whose final state starts and conditions a GRU decoder."""
    def __init__(self, num_letters, num_hiddens, num_inputs=32):
        super().__init__()  # One more embedding: the start token
        self.embedding = nn.Embedding(num_letters + 1, num_inputs)
        self.encoder = nn.GRU(num_inputs, num_hiddens, batch_first=True)
        self.decoder = nn.GRU(num_inputs + num_hiddens, num_hiddens,
                              batch_first=True)
        self.head = nn.Linear(num_hiddens, num_letters)
        d2l.init_lecun(self)
        nn.init.normal_(self.embedding.weight, std=num_inputs ** -0.5)

    def forward(self, X):  # X: the source, then the decoder input
        src, dec_in = X.chunk(2, dim=1)
```

```
        _, c = self.encoder(self.embedding(src))  # (1, batch, num_hiddens)
        context = c[0, :, None].expand(-1, dec_in.shape[1], -1)
        dec_X = torch.cat([self.embedding(dec_in), context], -1)
        return self.head(self.decoder(dec_X, c)[0])
```

新規に生成した 128 個の文字列からなるミニバッチを用い、3×10^{-3} からゼロへのコサイン減衰を伴う AdamW（`d2l.LMOptimizer` および `d2l.train_steps`）によって、11.4 節と同じ設定で各モデルを 1,000 ステップ訓練する。評価では、新規の 1,000 文字列に対する文字ごとの正解率を教師強制のもとで測定する。各文字は独立であるため、正解の直前トークンを入力しても次のターゲット文字に関する情報は一切漏洩せず、同時に初期のデコーディング誤りが状態容量の測定に交絡するのを防げるからである。

```
def reversal_accuracy(num_hiddens, length):
    """Train a fresh model for 1,000 steps; return its per-letter accuracy."""
    data = Reversal(length)
    model = Reverser(data.num_letters, num_hiddens).to(d2l.try_gpu())
    optimizer = d2l.LMOptimizer(model, lr=3e-3, num_steps=1000, warmup=0,
                                weight_decay=0.0)
    d2l.train_steps(model, data, optimizer, 1000)
    X, Y = data.batch(1000)
    with torch.inference_mode():
        return (model(X).argmax(-1) == Y).float().mean().item()
```

状態幅 64 と 256 について、長さ $T \in \{8, 16, 32, 64\}$ にわたり 2 つの乱数シードで実験を実行する（ランダム予測の正解率は $1/16 \approx 0.06$ である）。

```
lengths, widths = (8, 16, 32, 64), (64, 256)
print('width' + ''.join(f'{f"T = {T}":>12}' for T in lengths))
for num_hiddens in widths:
    row = [[reversal_accuracy(num_hiddens, T) for _ in range(2)]
           for T in lengths]
    print(f'{num_hiddens:5d}' + ''.join(f'   {a:.2f}/{b:.2f}' for a, b in row))
```

```
width       T = 8       T = 16       T = 32       T = 64
   64   0.99/0.99   0.74/0.74   0.41/0.42   0.27/0.26
  256   1.00/1.00   0.99/0.99   0.80/0.80   0.51/0.51
```

幅 64 では、正解率は $T = 8$ での 0.99 から、$T = 16$ で約 0.73、$T = 64$ で 0.26〜0.27 へと低下する。状態幅を 4 倍の 256 に増やすと、$T = 16$ まで正解率 0.99 を維持し、$T = 64$ でも約半数の文字を復元できる。それでもなお、実効容量は生の浮動小数点のビット数に比べればごく一部にすぎない。$T = 16$ の文字列が持つ情報量はわずか 64 ビット（幅 64 では状態の浮動小数点数 1 個あたり 1 ビット）であるにもかかわらず、文字の 4 分の 1 が失われている。さらに、系列長に比例して状態幅を拡大しても性能は低下する（幅 64、$T = 8$ では 0.99 に達するのに対し、幅 256、$T = 32$ では約 0.80 にとどまる）。これは、ネットワークが序盤の文字をより長い非線形なリカレント更新の連鎖にわたって伝播させなければならないためである。

ニューラル機械翻訳も 2014 年にこれとまったく同じボトルネックに直面し (Cho, Van Merriënboer, Bahdanau, et al. 2014)、それを受けて Bahdanau et al. (2015) はデコーダがすべてのエンコーダ状態を振り返って参照できるようクロスアテンションを導入した（10.2 節、11.4 節）。これにより、入力長に対して線形に増大するキャッシュのコストと引き換えに、長さ 12 の反転タスクは 300 ステップ以内に完全に解けるようになった。本章の残りの節では、これと相補的な道筋を

探る。すなわち、固定サイズの状態を保ったまま訓練を並列化し、状態をベクトルからキー・値行列メモリへと拡張し、最後に少数のアテンション層と組み合わせるアプローチである。

12.2.4 まとめ

エンコーダ・デコーダアーキテクチャは、ソースをエンコーダ状態 $\mathbf{c}$ に要約し、自己回帰デコーダを $\mathbf{c}$ で条件付けることによって、可変長のソース系列を可変長のターゲット系列へと写像する。翻訳品質は一般に、文字 n グラムの重複（chrF）や短文ペナルティを伴う単語 n グラム適合率（BLEU）で測定され、学習済みの意味的評価指標によって補完される。人工的な文字列反転タスクでは、単一の固定幅 GRU 状態ベクトルは系列長が増すにつれて急速に劣化し、状態の座標あたり 1 実効ビットを大きく下回る情報しか保持できない。この限界が、アテンション機構と、本章の残りの節で展開する構造化された高容量な再帰状態の双方への動機付けとなる。

12.2.5 演習問題

1. **BLEU 対 chrF**。ニアミスと言い換えについて $k = 1, 2, 3, 4$ で `bleu` のスコアを計算し、それぞれの値を n グラムの出現回数から説明せよ。次に、どちらの方向でもよいので chrF と BLEU が可能な限り大きく食い違う候補訳を 1 つ構成し、その候補訳が各評価指標のどの性質を突いているかを説明せよ。
2. **状態を通過するビット数**。本節の数え上げの議論によれば、N 個の数値（各 b ビット）からなる状態は、高々 2^{Nb} 通りの値しかとり得ない。
   1. 16 種類の文字からなる長さ T のあらゆる文字列を誤りなく反転させるには、$Nb \geq 4T$ が必要であることを示せ。
   2. 本節と同様に正しい直前文字を入力として測定したデコーダの文字ごとの正解率を p とする。ファノの不等式（37.6.2.3 節）と相互情報量の連鎖律を用いて、文字列に関して少なくとも $T\,(4 - H_{\mathrm{b}}(1-p) - (1-p)\log_2 15)$ ビットの情報が $\mathbf{c}$ を通過したことを示せ。ただし、H_{b} はビット単位の二値エントロピーである。
   3. 表の各欄についてこの下界を計算し、$32N$ と比較せよ。訓練されたエンコーダは数値 1 個あたり何ビットを使用しているか。また、その値は T に依存するか。
   4. 幅 64 のモデルを各長さで 4 倍長く訓練せよ。どの長さで最も大きく改善し、長さに応じた低下は解消されるか。この結果から、限界を定めているのが状態なのか訓練なのかについて何が言えるか。
3. **Sutskever の工夫の解除**。本節の反転タスクは、最初に書き出す文字が最後に読み取った文字であるという点で、Sutskever らがソースを逆順にすることで得た配置を最初から備えている。ソースを逆順でエンコーダに入力することでこの配置を打ち消し、エンコーダが読み取った順序のままデコーダが文字列をコピーしなければならないようにせよ。
   1. 反転タスクとコピータスクのそれぞれについて、各文字を読み取ってから書き出すまでのステップ数を数えよ。平均して依存距離が短いのはどちらのタスクか。また、最も短い単一の依存距離を持つのはどちらか。
   2. 幅 64、$T = 16$ で両方のタスクを訓練し、全体および各出力位置での正解率を測定せよ。どちらのタスクの方が学習しやすいか。その結果は、ソースを逆順にする理由として Sutskever et al. (2014) が挙げた議論と一致するか。
4. **毎ステップでのコンテキスト入力**。Sutskever らが行ったように、コンテキスト $\mathbf{c}$ を毎ステップの入力としては渡さず、初期状態としてのみデコーダに渡せ。それによって各長さでの正解率はどのように変化するか。また、コンテキストを繰り返し与えることが小さなモデルの助けになり得るのはなぜか。

12.3 線形漸化式と並列スキャン

12.1 節のゲート付きネットワークは数百ステップにわたって情報を保持できるが、各ステップのゲートと候補が前の状態 $\mathbf{h}_{t-1}$ に依存するため、依然として系列を 1 トークンずつ順番に処理する。$4{,}096$ トークンの系列では、GRU は $4{,}096$ 回の逐次更新を実行することになり、Transformer が大規模な行列積によって全位置を並列に処理する一方で、訓練中に GPU の大部分を遊休状態にしてしまう。リカレントネットワークにおいて、$O(1)$ 状態の自己回帰生成（autoregressive generation）を維持しつつ、全位置にわたって並列に訓練することはできるだろうか。

答えはイエスである。ゲートと候補が現在の入力 $\mathbf{x}_t$ のみに依存するようになれば、漸化式は $\mathbf{h}_{t-1}$ に関して**線形**（より正確には**アフィン**）になるからである。アフィン更新はプレフィックス和（prefix sum）と同様に結合的に合成でき (Hillis and Steele 1986)、全 T 個の状態を $\lceil \log_2 T \rceil$ 回の並列ラウンドで評価できる。以下では、スカラーの減衰から、最小 GRU（minimal GRU、`MinGRU`）として Feng et al. (2024) が導入した構造へと進み、結合的な並列スキャン（parallel scan）を実装したうえで、バイトレベルの WikiText-103 で `MinGRU` を評価し、最大 $16{,}384$ トークンまでの実時間のスケーリングを逐次型の GRU および因果アテンション（causal attention）と比較する。

12.3.1 線形漸化式

12.3.1.1 減衰する単一の数値

まず、スカラー状態 h_t が直前の値の一定割合 $a \in (0,1)$ を保持し、そこに入力 x_t を加える場合から始めよう。

$$h_t = a\, h_{t-1} + x_t, \qquad h_0 = 0. \tag{12.14}$$

Equation 12.14 を展開すると、h_t は過去の入力によって直接次のように表される。

$$h_t = \sum_{s=1}^{t} a^{t-s} x_s. \tag{12.15}$$

すべての h_t は半減期（half-life）$\ln 2 / \ln(1/a)$ ステップ（$a = 0.9$ では ≈ 7 ステップ、$a = 0.999$ では ≈ 693 ステップ）をもつ過去の入力の指数重み付き移動平均（exponentially weighted moving average）であり、勾配 $\partial h_t / \partial x_s = a^{t-s}$ もまったく同じ率で減衰する。

12.3.1.2 入力依存の減衰

ベクトル状態 $\mathbf{h}_t \in \mathbb{R}^d$ へと一般化し、入力依存（input-dependent）の要素ごとの減衰 $\mathbf{a}_t \in (0,1)^d$ と**書き込み**（write）ベクトル $\mathbf{b}_t \in \mathbb{R}^d$ を導入すると、次のアフィン漸化式（affine recurrence）が得られる。

$$\mathbf{h}_t = \mathbf{a}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_t, \qquad \mathbf{a}_t = \mathbf{a}(\mathbf{x}_t), \quad \mathbf{b}_t = \mathbf{b}(\mathbf{x}_t). \tag{12.16}$$

Equation 12.16 を展開すると、べき乗 a^{t-s} は途中の減衰の累積積（cumulative product）に置き換わる。

$$\mathbf{h}_t = \sum_{s=1}^{t} \Big(\prod_{r=s+1}^{t} \mathbf{a}_r \Big) \odot \mathbf{b}_s. \tag{12.17}$$

$\mathbf{a}_t$ と $\mathbf{b}_t$ は $\mathbf{x}_t$ のみに依存するため、漸化式を実行する前に、全 T 個の減衰ベクトルと書き込みベクトルを系列全体にわたって並列に計算できる。また、各 $\mathbf{a}_r \in (0,1)^d$ であるから、過去の書き込みを発散させることなく選択的に保持したり（$\mathbf{a}_r \approx \mathbf{1}$）、消去したり（$\mathbf{a}_r \approx \mathbf{0}$）できる。

12.3.1.3 最小 GRU

前の状態 $\mathbf{h}_{t-1}$ が 12.1.3 節の標準的な GRU に入り込む 3 つの箇所を見てみよう。

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_{\mathrm{xz}}\mathbf{x}_t + \underbrace{\mathbf{W}_{\mathrm{hz}}\mathbf{h}_{t-1}}_{(1)} + \mathbf{b}_{\mathrm{z}}),\\
\mathbf{r}_t &= \sigma(\mathbf{W}_{\mathrm{xr}}\mathbf{x}_t + \underbrace{\mathbf{W}_{\mathrm{hr}}\mathbf{h}_{t-1}}_{(2)} + \mathbf{b}_{\mathrm{r}}),\\
\tilde{\mathbf{h}}_t &= \tanh\big(\mathbf{W}_{\mathrm{xh}}\mathbf{x}_t + \underbrace{\mathbf{W}_{\mathrm{hh}}(\mathbf{r}_t \odot \mathbf{h}_{t-1})}_{(3)} + \mathbf{b}_{\mathrm{h}}\big),
\end{aligned}
$$

これに続いて $\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1-\mathbf{z}_t)\odot\tilde{\mathbf{h}}_t$ が計算される。印を付けた 3 つのリカレントな依存関係を除去すると（これによりリセットゲート $\mathbf{r}_t$ は完全に消える）、Feng et al. (2024) の**最小 GRU**（**minGRU**）が得られる。

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_{\mathrm{z}}\mathbf{x}_t + \mathbf{b}_{\mathrm{z}}), \qquad \tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{\mathrm{h}}\mathbf{x}_t + \mathbf{b}_{\mathrm{h}}),\\
\mathbf{h}_t &= \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1-\mathbf{z}_t)\odot\tilde{\mathbf{h}}_t.
\end{aligned}
\tag{12.18}
$$

これは Equation 12.16 において減衰を $\mathbf{a}_t = \mathbf{z}_t$、書き込みを $\mathbf{b}_t = (1-\mathbf{z}_t)\odot\tilde{\mathbf{h}}_t$ としたアフィン漸化式であり、層のパラメータ数を $6d^2$ から $2d^2$ へと削減する。単一の minGRU 層はすでに $\mathbf{h}_{t-1}$ に入っている情報でゲートを条件付けることができないが、複数の minGRU ブロックを位置ごとの MLP とともに積み重ねれば、時間方向の並列評価を損なうことなくその能力の一部を回復できる。各層のゲートは、1 つ下の層の状態を参照できるからである。これにより、GRU が毎ステップ 1 回の状態依存ゲーティングを行うのに対し、1 層あたり 1 ラウンドの状態依存ゲーティングが得られる。固定回数のラウンドでは実現できない処理については、12.7.3.5 節で検証する。

12.3.2 並列スキャン

12.3.2.1 アフィンステップの合成

展開された和 Equation 12.17 をすべての $t = 1, \dots, T$ に対して素朴に評価すると、$O(T^2)$ 回の演算が必要になる。そこで、関数合成の結合律（associativity）を利用する。ステップ t のアフィン写像 $\mathbf{h} \mapsto \mathbf{a}_t \odot \mathbf{h} + \mathbf{b}_t$ を組 $(\mathbf{a}_t, \mathbf{b}_t)$ で表そう。ステップ 1 に続けてステップ 2 を適用すると $\mathbf{a}_2 \odot (\mathbf{a}_1 \odot \mathbf{h} + \mathbf{b}_1) + \mathbf{b}_2$ となり、これは次の係数をもつ別のアフィン写像になる。

$$
(\mathbf{a}_1, \mathbf{b}_1) \bullet (\mathbf{a}_2, \mathbf{b}_2) = (\mathbf{a}_1 \odot \mathbf{a}_2,\ \mathbf{a}_2 \odot \mathbf{b}_1 + \mathbf{b}_2).
\tag{12.19}
$$

関数の合成は結合的であり、任意のステップ列の合成もやはり同じサイズの単一の組 $(\mathbf{a}, \mathbf{b})$ になるため、$\mathbf{h}_1, \dots, \mathbf{h}_T$ を初期状態 $\mathbf{h}_0 = \mathbf{0}$ から計算する処理は、演算 $\bullet$ のもとでの結合的なプレフィックス**スキャン**に帰着する（$\mathbf{h}_t$ は t 番目のプレフィックスの第 2 成分から読み出せる）。

12.3.2.2 倍増スケジュール

Figure 12.6 は、Hillis and Steele (1986) による並列倍増スケジュール（parallel doubling schedule）を $T = 8$ ステップの場合について示している。ラウンド $r = 0, 1, \ldots, \lceil \log_2 T \rceil - 1$ において、すべての位置はストライド 2^r だけ手前の位置と結合する。$\lceil \log_2 T \rceil$ ラウンド後には、すべての位置が完全なプレフィックスを保持する。

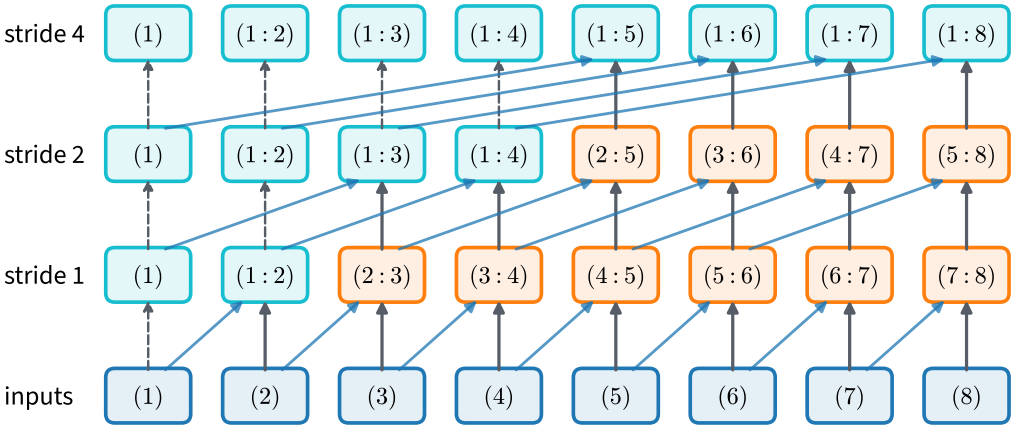


Figure 12.6: 8 ステップに対する並列スキャン。各ボックスはその値がどのステップを結合したものかを表し、$(1{:}4)$ はステップ 1 から 4 を意味する。各ラウンドでは、それぞれの位置が一定のストライドだけ手前の位置と結合し（青色の矢印）、ストライドはラウンドごとに倍増する。破線の矢印は値をそのままコピーする。青緑色のボックスは完全なプレフィックスを保持している。$\log_2 8 = 3$ ラウンド後には、すべての位置が完全なプレフィックスを保持する。

Hillis–Steele の倍増スケジュールは $O(T \log_2 T)$ 回の結合演算を $\lceil \log_2 T \rceil$ 回の並列ラウンドにわたって実行するのに対し、Blelloch (1990) による**演算量効率的**（work-efficient）なアップスイープ・ダウンスイープ木スケジュールは、$\sim 2T$ 回の結合演算をサイズが縮小していく $2\lceil \log_2 T \rceil$ 回のラウンドにわたって実行する。

簡潔な倍増スケジュールを `associative_scan` として記述する。

12.3.2.3 実装

`linear_scan` は、時間軸（軸 1）に沿ってすべての状態 $\mathbf{h}_1, \ldots, \mathbf{h}_T$（Equation 12.16）を計算し、任意の初期状態 $\mathbf{h}_0$ が与えられた場合は最初の書き込み $\mathbf{b}_1 \leftarrow \mathbf{a}_1 \odot \mathbf{h}_0 + \mathbf{b}_1$ に折り込む。

```
def associative_scan(combine, elems):
    """All prefixes of elems under an associative combine; time is axis 1."""
    s = 1
    # Autograd keeps the tensors of every round for the backward pass
    while s < elems[0].shape[1]:  # Combine every position with the one s back
        earlier = tuple(x[:, :-s] for x in elems)
        later = tuple(x[:, s:] for x in elems)
        elems = tuple(torch.cat([x[:, :s], y], 1)
                      for x, y in zip(elems, combine(earlier, later)))
        s *= 2
```

```
    return elems

def linear_scan(a, b, h0=None):
    """All states of h_t = a_t * h_{t-1} + b_t; time is the second axis."""
    a = a.expand_as(b)  # Decays may omit axes on which they do not vary
    if h0 is not None:  # Fold the initial state into the first step
        b = torch.cat([a[:, :1] * h0.unsqueeze(1) + b[:, :1], b[:, 1:]], 1)
    def combine(earlier, later):
        (a1, b1), (a2, b2) = earlier, later
        return a1 * a2, a2 * b1 + b2
    return associative_scan(combine, (a, b))[1]
```

長さ $T = 100$（2 のべき乗でない長さ）において、`linear_scan` を 1 ステップずつ進める逐次ループと照合して検証してみよう。

```
def sequential_scan(a, b):
    h, states = torch.zeros_like(b[:, 0]), []
    for a_t, b_t in zip(a.unbind(1), b.unbind(1)):
        h = a_t * h + b_t
        states.append(h)
    return torch.stack(states, 1)

a, b = torch.rand(4, 100, 8), torch.randn(4, 100, 8)
print(f'max deviation: '
      f'{(linear_scan(a, b) - sequential_scan(a, b)).abs().max():.1e}')
```

```
max deviation: 4.8e-07
```

並列スキャンと逐次ループは、float32 の丸め誤差の許容範囲（$< 10^{-6}$）内で一致する。

12.3.3 線形リカレント言語モデル

12.3.3.1 テキスト上の minGRU

`linear_scan` を 2 つの線形射影（$\mathbf{W}_z$ と $\mathbf{W}_h$）と組み合わせることで、`MinGRU`（Equation 12.18）を `d2l.SeqLM` にそのまま差し替え可能な系列ミキサとして実装できる。

```
class MinGRU(nn.Module):
    """The minimal GRU: its gate and candidate read only the input."""
    def __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.W_z = nn.Linear(num_inputs, num_hiddens)  # Update gate (decay)
        self.W_h = nn.Linear(num_inputs, num_hiddens)  # Candidate state
        d2l.init_lecun(self)

    def coefficients(self, X):
        """Decays a_t and writes b_t of the linear recurrence."""
        Z = torch.sigmoid(self.W_z(X))
        return Z, (1 - Z) * torch.tanh(self.W_h(X))

    def forward(self, X):  # X: (batch_size, num_steps, num_inputs)
```

```
        return linear_scan(*self.coefficients(X))
```

ゲートから $\mathbf{h}_{t-1}$ を除去すると、バイトレベルの WikiText-103 におけるモデル化品質はどの程度損なわれるだろうか。12.1.4.2 節とまったく同じ設定（`d2l.SeqLM` の 4 ブロック、128 バイトのウィンドウ、1,000 ステップ）を用い、まず `text_model` と 3 つの比較アーム `arms` を定義する。すなわち、`GRU`（$d = 256$、パラメータ数 3.81M）、同じ幅の `minGRU`（$d = 256$、パラメータ数 2.76M）、およびパラメータ数が等しくなるよう幅を広げた `minGRU`（`minGRU, d=302`、パラメータ数 3.81M）である。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32

def text_model(mixer, d):  # 256 byte values; four blocks of width d
    return d2l.SeqLM(256, d, [mixer(d) for _ in range(4)]).to(d2l.try_gpu())

gru = lambda d: d2l.Recurrent(d2l.init_lecun(nn.GRU(d, d, batch_first=True)))
min_gru = lambda d: MinGRU(d, d)
arms = {'GRU': (gru, 256), 'minGRU': (min_gru, 256),
        'minGRU, d=302': (min_gru, 302)}  # As many parameters as the GRU
```

128 バイトの `WikiText103` データセットを読み込み、3 つのアームすべてにわたって `d2l.text_table` を実行し、検証用に最初の訓練済み `minGRU` モデルを保持する。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
trained = d2l.text_table(text_model, arms, data)
model = trained['minGRU'][0]  # The model we look inside below
```

```
                          validation        0-15       16-63      64-127   ms/step
GRU              3,817,216 1.299/1.300 1.490/1.491 1.273/1.274 1.270/1.271 33/32
minGRU           2,764,544 1.343/1.348 1.520/1.520 1.319/1.323 1.317/1.324 26/26
minGRU, d=302    3,816,932 1.328/1.329 1.508/1.507 1.303/1.304 1.301/1.303 26/28
```

同じ幅（$d = 256$）では、`minGRU` は ~ 0.05 nat/byte の差で `GRU` の後塵を拝するが（1.35 対 1.30）、同じパラメータ数（`minGRU, d=302`）ではその差の約 5 分の 2 を縮め（~ 1.33）、12.1.4.2 節の素の tanh RNN とほぼ並ぶとともに、アテンションのベースライン（1.38）を上回り、しかも逐次ループではなく並列スキャンによって訓練できる。

12.3.3.2 訓練済みモデルの内部

`minGRU` が再帰状態（recurrent state）にどれだけ依存しているかを調べるため、訓練済みの $d = 256$ モデルを、毎ステップの前に状態をゼロにリセットした設定（`StateReset`、これにより $\mathbf{h}_t = \mathbf{b}_t$ は現在のバイトのみに依存する）で評価する。

```
class StateReset(nn.Module):
    """A minGRU layer whose state is reset to zero before every step."""
    def __init__(self, layer):
        super().__init__()
        self.layer = layer

    def forward(self, X):
        return self.layer.coefficients(X)[1]
```

```
layers = model.mixers
loss = np.mean(d2l.loss_by_position(model, data))
model.mixers = nn.ModuleList([StateReset(layer) for layer in layers])
reset = np.mean(d2l.loss_by_position(model, data))
model.mixers = layers
print(f'validation loss: {loss:.3f} nats per byte, '
      f'with the state reset at every step: {reset:.3f}')
```

```
validation loss: 1.343 nats per byte, with the state reset at every step: 2.481
```

状態をゼロにすると、検証損失は 1.35 から 2.48 nat/byte へと悪化し（実質的に 2.45 nat/byte という 11.2 節の経験的バイトバイグラムのベースラインと同等）、1.1 nat/byte 以上の改善が状態によって運ばれる文脈に由来していることが確認できる。

また、第 1 ブロックにおける更新ゲート（update gate）の減衰 $\mathbf{z}_t$ の分布を訓練前後で調べることもできる。`first_mixer_input` を `d2l.SeqLM` に追加すると、最初のミキサに入力される正規化後のトークン埋め込みを取り出せる。

```
@d2l.add_to_class(d2l.SeqLM)
def first_mixer_input(self, X):
    """What the first mixer reads: the normalized token embeddings."""
    return self.norms[0](self.embedding(X))
```

検証ミニバッチを `first_gates` に通し、未訓練の `minGRU` と訓練済みの `model` の双方について、更新ゲートが 0.5、0.9、0.99 を超える割合を集計する。

```
def first_gates(model, X):  # Gates of the first layer
    return model.mixers[0].coefficients(model.first_mixer_input(X))[0]

X, _ = next(iter(data.val_dataloader()))  # Validation text
X = X.to(next(model.parameters()).device)
with torch.inference_mode():
    Z_init = first_gates(text_model(min_gru, 256), X)  # Untrained
    Z = first_gates(model, X)
print('gate above  half-life in steps  share at initialization  trained')
for g in (0.5, 0.9, 0.99):
    print(f'{g:10}  {math.log(2) / -math.log(g):17.0f}  '
          f'{(Z_init > g).float().mean():23.1%}  '
          f'{(Z > g).float().mean():7.1%}')
```

```
gate above  half-life in steps  share at initialization  trained
       0.5                 1                    49.0%    53.4%
       0.9                 7                     1.3%    17.1%
      0.99                69                     0.0%     0.1%
```

訓練によって、半減期が ≥ 7 バイトとなるゲート（$g > 0.9$）の割合は 10 倍以上に増加し（$\sim 1\%$ から約 18% へ）、128 バイトのウィンドウ上で必要とされる単語レベルのタイムスケールと一致するようになる。一方、ゼロバイアス初期化からは、$g > 0.99$（半減期 69 バイト）に達するゲートはほとんど現れない。

12.3.4 アテンションとのコスト比較

12.3.4.1 演算量、深さ、メモリ

Table 12.1 は、幅 d の 1 層で T トークンを処理する際の漸近的な訓練時の演算量、逐次的な深さ、および自己回帰生成時の状態を比較したものである。

Table 12.1: 幅 d の 1 層における T トークンに対するコスト（定数倍を除く）。

層	訓練時の演算量	連続するステップ数	生成時のメモリ
GRU、1 ステップずつ	Td^2	T	d
minGRU、倍増スキャン	$Td^2 + Td\log_2 T$	$\log_2 T$	d
minGRU、演算量効率的スキャン	$Td^2 + Td$	$2\log_2 T$	d
因果アテンション	$Td^2 + T^2d$	数回の行列積	$2Td$

`GRU` と `minGRU` はどちらも $O(Td^2)$ の射影の演算量を実行し、デコーディング中には $O(d)$ の状態ベクトルを保持するが、`minGRU` は訓練時の逐次的な深さを T 回の行列・ベクトル積ステップから $O(\log_2 T)$ 回の要素ごとのスキャンラウンドへと削減する。因果アテンションの訓練時の逐次的な深さは $O(1)$ であるが、アテンションスコアと混合の演算量は $O(T^2d)$ として二次関数的に増大し（射影の $8Td^2$ FLOPs を $T \approx 4d$ 付近で上回る）、推論時の KV キャッシュも $2Td$ として線形に増大する。

12.3.4.2 実測コスト

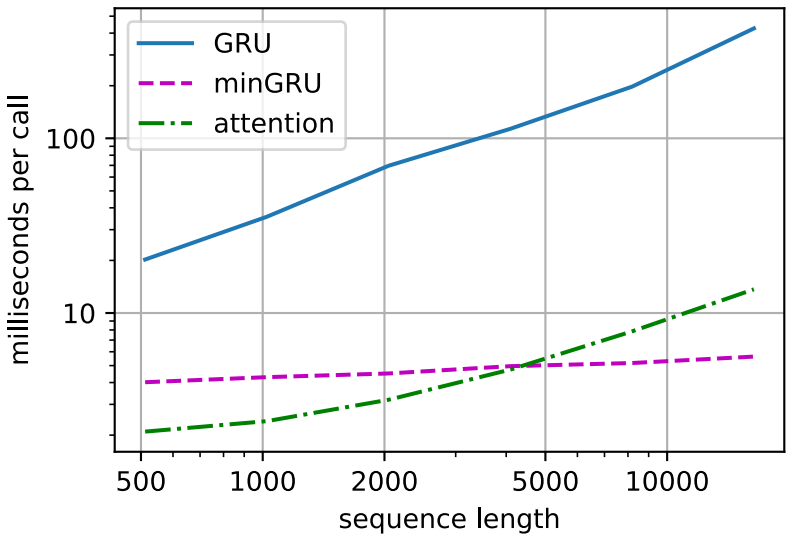
これらのスケーリング則（scaling law）が実ハードウェア上でどこで交差するかを見るため、float16 で幅 $d = 256$ とし、1 バッチあたりの総トークン数を $B \times T = 65{,}536$ に固定したまま、系列長（sequence length）T を 512 から 16,384 までスイープさせながら順伝播と逆伝播の合計時間を計測する。総トークン数を固定しているため、コストが T に対して厳密に線形な層の曲線は平坦になる一方、$O(T)$ 回の逐次ステップまたは $O(T^2)$ のペアワイズ演算量が支配的な層では、T が 2 倍になるたびに実行時間も 2 倍になる。

```
d, tokens = 256, 65536  # Width; tokens per call, in sequences of length T
device, half = d2l.try_gpu(), torch.float16  # 16-bit floats throughout
timed = {'GRU': d2l.Recurrent(nn.GRU(d, d, batch_first=True)),
         'minGRU': torch.compile(MinGRU(d, d)),
         'attention': d2l.CausalAttention(d, num_heads=4)}
for layer in timed.values():
    layer.to(device, half)

def time_call(layer, T):
    """Milliseconds for a forward and backward pass over tokens / T
    sequences of length T."""
    X = torch.randn(tokens // T, T, d, device=device, dtype=half,
                    requires_grad=True)
    return 1000 * d2l.Benchmark(lambda: layer(X).float().sum().backward(),
                                warmup=1, repeats=3).time
```

系列長 $T \in \{512, 1024, 2048, 4096, 8192, 16384\}$ にわたって `time_call` をスイープし、`GRU`、`minGRU`、`attention` の実実行時間を両対数軸上で比較する。

```
lengths = [512, 1024, 2048, 4096, 8192, 16384]
times = {name: [time_call(layer, T) for T in lengths]
         for name, layer in timed.items()}
print('T        ', *(f'{T:>7d}' for T in lengths))
for name, ms in times.items():
    print(f'{name:9s}', *(f'{t:7.1f}' for t in ms))
d2l.plot(lengths, list(times.values()), 'sequence length',
         'milliseconds per call', legend=list(times), xscale='log',
         yscale='log', figsize=(4.5, 3))
```

```
T             512    1024    2048    4096    8192   16384
GRU          20.2    35.6    69.7   113.3   197.6   425.6
minGRU        4.0     4.3     4.5     5.0     5.2     5.6
attention     2.1     2.4     3.2     4.7     7.9    13.7
```



実測されたスケーリング曲線は Table 12.1 と一致する。

- **GRU**：あらゆる長さで最も遅く、15× 以上に増大する（$T = 512$ から 16,384 まで）。T 回の逐次ステップを時間方向に並列化できないためである。
- **minGRU**：系列長が 32 倍に増加する全範囲にわたって、ほぼ平坦である。
- **attention**：$T \approx 2{,}048$ までは $O(Td^2)$ の射影とメモリトラフィックが支配的であるためほぼ平坦であるが、その後は系列長が倍増するごとに 2× へ漸近するように上向きに曲がる。$O(T^2 d)$ のアテンション項が支配的になるためである。

minGRU の倍増スキャンの増加は 50% 未満に収まる。短い系列長では高速な attention も、$T = 4{,}096$ 付近でこれと交差し、$T = 16{,}384$ では約 2.5× 遅れをとる。

minGRU にまだ欠けているのは、多重タイムスケールの記憶を構造化された方法で初期化・制御する仕組みである。この問いには、12.4 節で連続時間状態空間モデル（continuous-time state space model）を通じて取り組む。

12.3.5 まとめ

リカレント層のゲートから前の状態 $\mathbf{h}_{t-1}$ を除去すると、アフィン漸化式 $\mathbf{h}_t = \mathbf{a}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_t$ が得られ、その各ステップは $(\mathbf{a}_1, \mathbf{b}_1) \bullet (\mathbf{a}_2, \mathbf{b}_2) = (\mathbf{a}_1 \odot \mathbf{a}_2, \mathbf{a}_2 \odot \mathbf{b}_1 + \mathbf{b}_2)$ によって結合的に合成できる。結合的なスキャンを用いることで、訓練時には全 T 個の状態を $O(\log_2 T)$ 回の並列ラウンドで評価しつつ、デコーディング時には $O(1)$ 状態の逐次更新を維持できる。この簡略化を GRU に適用して得られる `MinGRU` は、バイトレベルの WikiText-103 において同じパラメータ数の素の RNN に匹敵する性能を示し（完全な GRU との差は約 0.03 nat/byte）、しかも最大 $16{,}384$ トークンまでトークンあたりほぼ一定の時間で訓練できる。これは逐次型の GRU より 1 桁以上高速であり、長文脈においては二次計算量のアテンションよりも数倍高速である。

12.3.6 演習問題

1. **合成に関する閉包性**。並列スキャンには結合律だけでなく、任意の連続するステップの合成が単一ステップと同じ個数の数値で記述でき、すべての結合演算が同じコストで実行できることが求められる。
    1. Equation 12.19 を、漸化式 $\mathbf{h}_t = \mathbf{A}_t \mathbf{h}_{t-1} + \mathbf{b}_t$（ただし $\mathbf{A}_t \in \mathbb{R}^{d \times d}$ は密行列）へと、各積の因子の順序を正しく保って一般化せよ。1 回の結合演算と 1 回の逐次ステップの積和演算数を数えよ。倍増スキャンの総演算量（約 $T \log_2 T$ 回の結合演算）とループの総演算量を $T = 4096$、$d = 64$ において比較せよ。この比較は密な遷移行列について何を示唆するか。
    2. スカラーステップ $h \mapsto \tanh(w_t h + c_t)$ に対し、$w_1, w_2 \neq 0$ である 2 ステップの合成は、どのような $h \mapsto \tanh(wh + c)$ の形（任意の w と c）にも表せないことを示せ。k ステップの合成を表現するには、スキャンは何を保持しなければならないか。
2. [code] **直接計算の経路**。スカラーの減衰に対しては、展開された和 Equation 12.17 はスキャンを使わずに、累積積 $P_t = a_1 a_2 \cdots a_t$ と 1 回の累積和から評価できる。
    1. $h_t = P_t \sum_{s=1}^{t} b_s / P_s$ が成り立つことを示せ。ライブラリの `cumprod` と `cumsum` を用いてこれを実装し、0.9 付近から抽出した減衰に対して 32 ビット浮動小数点数で `linear_scan` と比較せよ。どの長さで破綻するか、またなぜスキャンはその長さで破綻しないのか。
    2. Feng et al. (2024) は減衰と書き込みの対数を扱い、和を累積 log-sum-exp に置き換えることでこの式を修正している。この修正を実装せよ。書き込みの符号にはどのような条件が必要となり、著者らはそれをどのように実現しているか。
3. [code] **候補の非線形性**。`MinGRU` は tanh を候補 $\tilde{\mathbf{h}}_t$ に残しているのに対し、Feng et al. (2024) は線形な候補を用いている。
    1. $\mathbf{h}_0 = \mathbf{0}$ から出発し、$\mathbf{h}_t$ の各座標を候補 $\tilde{\mathbf{h}}_1, \ldots, \tilde{\mathbf{h}}_t$ の重み付き和として表せ。重みが非負でその和が高々 1 になることを示し、あらゆる長さで成り立つ $|\mathbf{h}_t|$ の上界を tanh がある場合とない場合のそれぞれについて導出せよ。
    2. tanh を取り除き、本節の言語モデルを `d2l.train_text` で 2 回訓練して、その検証損失を `MinGRU` の 2 回の実行結果と比較せよ。その差は実行間のばらつきより大きいか。観察された結果を導出した上界と関連付けて説明せよ。
4. [code] **状態の引き継ぎ**。`linear_scan` は初期状態を受け取るため、長い系列を分割して処理できる。
    1. 長さ T の系列を長さ τ のチャンクに分割し、直前のチャンクの最終状態から開始して各チャンクに対してスキャンを実行し、連結した状態が系列全体に対する 1 回の呼び出し結果と一致することを確かめよ。
    2. 次に、各チャンク境界で引き継がれる状態に対して勾配停止を適用せよ。状態自体は変化しない。ステップ t における損失の勾配への寄与のうちどれが消え、訓練済みモデルがテキストを生成する際にはどの依存関係を依然として利用できるか。打ち切り型通時的誤差逆伝播法（8.5 節）と比較せよ。

5. [code] **演算量効率的なスキャン**。●再帰的なスケジュールは倍増スケジュールよりも演算量が少ない。隣接するペアを結合して系列長を半分にし、その半分の長さの系列を再帰的にスキャンして 1 つおきのプレフィックスを得たあと、残りの各プレフィックスをもう 1 回の結合演算で求める。
   1. $T = 2^k$ に対して、各スケジュールの結合演算の回数と結合ラウンド数を数え、その両方をループの $T-1$ ステップと $T = 16384$ において比較せよ。
   2. スライスを用いて両方のスケジュールを実装し、どちらも `sequential_scan` と照合して確認したうえで、12.3.4 節の計測と同じ系列長で実行時間を計測せよ。実行時間の比は結合演算の回数の比に従うか。短い系列長では従わない場合がある理由を説明せよ。

12.4 状態空間モデル

記憶はどのくらいの期間持続すべきだろうか。文の主語は数語の間、段落の話題は数百語の間、そして小説の登場人物の正体は 1 冊の本全体にわたって重要となる。原理的には、12.3 節の minGRU はこれらの時間スケールのいずれも学習できるはずだが、そのゲートは $\sigma(0) = 1/2$（半減期 1 ステップ）の付近から出発する。ある記号を 1,000 ステップにわたって保持するには 0.999 付近の減衰（半減期 693 ステップ）が必要であり、初期減衰 $1/2$ を通じて 1,000 ステップさかのぼる勾配は 2^{-1000} 倍に縮小し、単精度では 0 にアンダーフローしてしまう。また、一様なゲート初期化では、最初から各チャネルに多様な時間スケールのバンクを与えることもできない。

連続時間の力学系（dynamical system）は、多重時間スケールの減衰に対する自然なパラメータ化を与える。**状態空間モデル**（state space model、SSM）は、各チャネルの記憶を、異なる減衰率を持ち離散的なステップサイズ Δ でサンプリングされる連続時間線形微分方程式のバンクとして扱う。この系を離散化すると、ステップごとの減衰が状態レートとチャネルのステップサイズの積で決まる線形漸化式が得られる。これにより、訓練開始前に幅広い半減期のスペクトルを初期化し、各チャネルに一度に N 個の時間スケールを持たせ、さらに時不変なダイナミクスを FFT による因果畳み込み（causal convolution）として並列に評価できる。以下では、厳密なゼロ次ホールド（zero-order hold）離散化を導出し、Gu, Gupta, et al. (2022) の対角 S4D 層を実装したうえで、1,000 ステップの遅延想起（delayed recall）において多重時間スケール初期化が長距離記憶の鍵となる要素かどうかを検証する。

12.4.1 連続時間から漸化式へ

12.4.1.1 連続時間ダイナミクス

まず、スカラー状態 $h(t)$ がレート λ で減衰しつつ、入力信号 $x(t)$ によって駆動される系から始めよう。

$$h'(t) = \lambda\, h(t) + x(t). \tag{12.20}$$

$x(t) = 0$ のとき、解は $h(t) = e^{\lambda t} h(0)$ である。安定性のためには $\lambda < 0$ が必要であり、このとき $h(t)$ は指数関数的に減衰する（$\lambda = -1$ では、割合 $e^{-1} \approx 0.37$ が連続時間の 1 単位後に残る）。一方、$\lambda > 0$ では際限なく増大する。重要な点として、λ はトークンステップあたりではなく、**単位時間あたり**の減衰率である。

状態空間モデル（SSM）は Equation 12.20 を一般化し、N 次元の状態ベクトル $\mathbf{h}(t) \in \mathbb{R}^N$ をスカラー入力 $x(t)$ で駆動してスカラー出力 $y(t)$ へと射影する。

$$\begin{aligned} \mathbf{h}'(t) &= \mathbf{A}\,\mathbf{h}(t) + \mathbf{B}\,x(t), \\ y(t) &= \mathbf{C}\,\mathbf{h}(t) + D\,x(t), \end{aligned} \tag{12.21}$$

ここで、$\mathbf{B} \in \mathbb{R}^{N}$ は入力を状態へと射影し、**状態行列**（state matrix）$\mathbf{A} \in \mathbb{R}^{N\times N}$ は状態を時間発展させ、$\mathbf{C} \in \mathbb{R}^{1\times N}$ は状態を読み出し、$D \in \mathbb{R}$ は直接のスキップ接続である。

$\mathbf{A} = \mathrm{diag}(\lambda_1, \dots, \lambda_N)$ を対角行列に制限すると、Equation 12.21 は N 個の独立な Equation 12.20 のコピーへと帰着し、それぞれがレート $\lambda_1, \dots, \lambda_N$ で時間発展して $\mathbf{C}$ により線形結合される。密行列 $\mathbf{A}$ ではステップあたり $O(N^2)$、結合的なスキャンの結合演算あたり $O(N^3)$ のコストがかかるのに対し、対角行列 $\mathbf{A}$ のコストは $O(N)$ 回のスカラー演算で済む。複素数体上ではほとんどすべての行列が対角化可能であり、$\mathbf{B}, \mathbf{C}$ が固有ベクトルによる基底変換を吸収するため、対角構造にしても表現力はほとんど損なわれない (Gu, Gupta, et al. 2022)。実レート $\lambda_n < 0$ を用いると、純粋な実数演算で計算できる代わりに、振動モード（演習問題で扱う $\lambda = -\alpha + i\omega$）は失われる。

SSM 層は d 次元の入力系列を処理する際、このような N 状態の系を座標（**チャネル**）ごとに 1 つずつ独立に走らせるため、層全体の状態は合計 $d \times N$ 個の数値となる。これは状態数が N 倍（幅 d の minGRU との比較）であることを意味する。

12.4.1.2 離散化

Equation 12.21 を離散的なトークン系列に適用するため、ステップサイズ Δ の間隔で系をサンプリングし、ステップ t が連続時間 $t\Delta$ に対応するようにする。入力を x_t で区間 $((t-1)\Delta, t\Delta]$ にわたり一定に保つと（**ゼロ次ホールド**、略して ZOH の仮定）、Equation 12.20 を厳密に積分できる。$\frac{\mathrm{d}}{\mathrm{d}t}(h + x_t/\lambda) = \lambda(h + x_t/\lambda)$ が成り立つため、シフトした状態 $h + x_t/\lambda$ は 1 ステップの間に $e^{\Delta\lambda}$ 倍に減衰し、$h_t + x_t/\lambda = e^{\Delta\lambda}(h_{t-1} + x_t/\lambda)$ となる。これを整理すると、厳密な離散線形漸化式が得られる。

$$\begin{aligned} h_t &= \bar{a}\,h_{t-1} + \bar{b}\,x_t, \\ \bar{a} &= e^{\Delta\lambda}, \qquad \bar{b} = \frac{e^{\Delta\lambda} - 1}{\lambda}. \end{aligned} \tag{12.22}$$

$\lambda = -1$ のとき $\bar{b} = 1 - \bar{a}$ となるため、Equation 12.22 は凸結合（convex combination）$h_t = \bar{a}\,h_{t-1} + (1-\bar{a})\,x_t$ となり、最小 GRU の更新式（12.3.1.3 節）において $\bar{a} = e^{-\Delta}$ を更新ゲートとしたものと一致する。ステップサイズ Δ は記憶の長さを直接制御する。すなわち、$\lambda = -1$ のとき、$\Delta = 0.01$ では $\bar{a} = 0.990$（半減期 69 ステップ）、$\Delta = 0.1$ では $\bar{a} = 0.905$（半減期 7 ステップ）、$\Delta = 1$ では $\bar{a} = 0.37$ となる。

$\lambda = -e^{\theta} < 0$ および $\Delta > 0$ とパラメータ化すれば、シグモイドゲートと同様に $\bar{a} = e^{\Delta\lambda} \in (0, 1)$ が保証される。ここで新しいのは、半減期 $\ln 2/(\Delta|\lambda|)$ が積の形に分解される構造である（Figure 12.7）。N 個のレート $\lambda_1, \dots, \lambda_N$ がチャネル内の状態を異なる相対的時間スケールへと分散させる一方、チャネルごとの単一のステップサイズ Δ が N 個すべての時間スケールを一括して引き伸ばしたり圧縮したりする。

対角行列 $\mathbf{A}$ に対しては、ZOH は要素ごとに適用され、$\bar{a}_n = e^{\Delta\lambda_n}$ および $\bar{b}_n = (e^{\Delta\lambda_n} - 1)b_n/\lambda_n$ となる（`expm1` を用いて計算することで $\Delta\lambda_n$ が小さいときの桁落ちを防ぐ。また、密な行列指数関数 $\bar{\mathbf{A}} = e^{\Delta\mathbf{A}}$ については 38.1.2.3 節を参照）。

実数値版の S4D は、$\lambda_n = -n$（$n = 1, \dots, N$、HiPPO Legendre 射影行列の対角成分 (Gu et al. 2020)）と初期化し、各チャネルのステップサイズ Δ を $[0.001, 0.1]$ から対数一様に抽出す

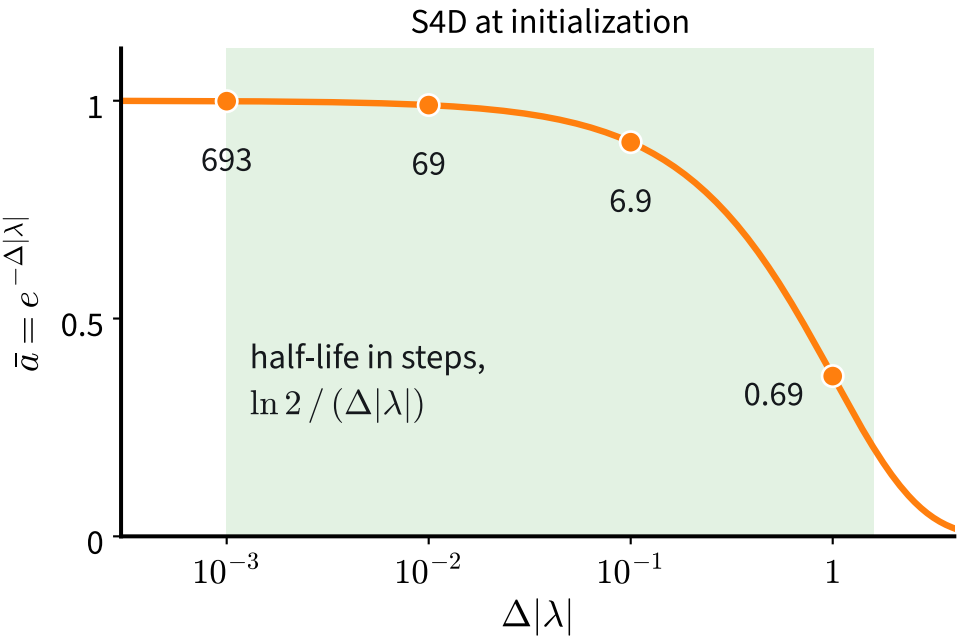


Figure 12.7: ゼロ次ホールドは実レート $\lambda < 0$ をステップあたりの減衰 $\bar{a} = e^{-\Delta|\lambda|}$ へと変換し、その値は積 $\Delta|\lambda|$ のみに依存する。図中のラベルはステップ単位の半減期 $\ln 2/(\Delta|\lambda|)$ を表す。網掛けの帯は S4D の初期化がカバーする範囲であり、Δ が 0.001 から 0.1 の間、$|\lambda|$ が 1 から 16 の間に対応する。

る。$N = 16$ のとき、初期の半減期は半ステップ未満（$\Delta = 0.1, \lambda = -16$）から 693 ステップ（$\Delta = 0.001, \lambda = -1$）にまで及び、勾配ステップを 1 回も踏む前に Figure 12.7 の網掛けの帯をカバーする。

12.4.1.3 畳み込み形式

$\bar{a}_n$、$\bar{b}_n$、$\mathbf{C}$ がトークン位置 t によって変化しないとき、その系は**線形時不変**（linear time-invariant、LTI）である。Equation 12.22 を $\mathbf{h}_0 = \mathbf{0}$ から展開すると、状態 n は $\sum_{k \geq 0} \bar{a}_n^k \bar{b}_n x_{t-k}$ を累積し、読み出し $\mathbf{C}$ が N 個の状態を足し合わせて単一の**因果畳み込み**を形作ることがわかる。

$$
\begin{aligned}
y_t &= \sum_{k=0}^{t-1} K_k \, x_{t-k} + D x_t, \\
K_k &= \sum_{n=1}^{N} c_n \bar{a}_n^k \bar{b}_n.
\end{aligned}
\tag{12.23}
$$

各チャネルのインパルス応答（impulse response）$\mathbf{K} = (K_0, K_1, \ldots)$ は、N 個の減衰する指数関数の線形結合である。したがって、時不変 SSM には 3 つの等価な見方が存在する（Figure 12.8）。

1. **連続時間常微分方程式**（continuous-time ordinary differential equation、ODE）：パラメータがレートとステップサイズを指定する。
2. **離散漸化式**：自己回帰生成の際に用いられ、トークンあたり $O(1)$ の計算時間と状態で動作する。
3. **因果畳み込み**：訓練時の全 T 位置にわたって、$O(T \log T)$ の計算量で高速フーリエ変換（fast Fourier transform、FFT）により評価される。S4 (Gu, Goel, et al. 2022) はこの畳み込みとしての見方を用いて、16,384 ステップの Path-X ベンチマークを解いた。なお、畳み込みとしての見方には時不変なダイナミクスが必要であるのに対し、12.3 節の並列スキャンはトークンごとに減衰が変化する場合でも機能する。

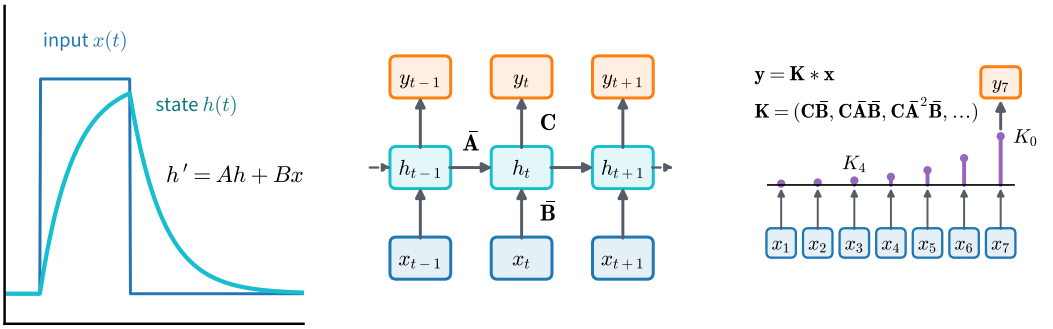


Figure 12.8: 1 つの状態空間モデルが持つ 3 つの形式。左：連続時間の状態が入力パルスに応答する。中央：離散化された系はトークンあたり 1 ステップの線形漸化式となる。右：展開すると、同じ系は単一のパルスに対する応答をカーネルとする因果畳み込みとなる。

12.4.1.4 時間スケールと学習されるカーネル

minGRU の初期ゲートバイアスを S4D の減衰に合わせて設定する以上に、複数状態を持つ対角 SSM は何をもたらすのだろうか。第 1 に、チャネルごとに単一のステップサイズ Δ が N 個

すべての時間スケールを一括してスケーリングする点であり、12.5 節ではこれを入力依存にする。第 2 に、N 個の状態を読み出し $\mathbf{C}$ で結合することにより、各チャネルは畳み込みカーネル $K_k = \sum_{n=1}^{N} c_n \bar{a}_n^k \bar{b}_n$ を単一の単調減衰ではなく N 個の指数関数の和として形作ることができる。

N 個の実指数関数は、非単調な遅延カーネルをどこまで精密に近似できるだろうか。あるチャネルが 20 ステップ前の入力を読み取る必要があるとし、鋭いスパイク（$K_{20} = 1$ で他はゼロ）またはラグ 20 付近の滑らかなガウス型のバンプを考える。単一の状態（$N = 1$）は常にラグ 0 でピークをとる。以下では、読み出し重み $\mathbf{c}$ を $N \in \{1, 4, 16\}$ 個の状態（S4D のレート $\lambda_n = -n$ と固定の $\Delta = 0.05$）に対して最小二乗法で当てはめ、相対残差誤差 $\|\mathbf{K} - \text{target}\|/\|\text{target}\|$ と必要となる読み出し重みの大きさ $|\mathbf{c}|_\infty$ の両方を調べる。

```
def fit_kernel(N, target, dt=0.05):
    """Fit the readout c of N states at S4D's rates to a target kernel."""
    rate = -np.arange(1., N + 1)
    a_bar, b_bar = np.exp(dt * rate), np.expm1(dt * rate) / rate
    basis = a_bar ** np.arange(len(target))[:, None] * b_bar  # K = basis @ c
    c = np.linalg.lstsq(basis, target, rcond=None)[0]
    K = basis @ c
    return K, c, np.linalg.norm(K - target) / np.linalg.norm(target)

k = np.arange(100)
spike = (k == 20).astype(float)      # Exactly the input 20 steps back
bump = np.exp(-(k - 20) ** 2 / 32)   # About 20 steps back, give or take 4
print(' N  peak  spike |c|  spike residual  bump |c|  bump residual')
for N in [1, 4, 16]:
    K, c, spike_error = fit_kernel(N, spike)
    _, c_bump, bump_error = fit_kernel(N, bump)
    print(f'{N:2d}  {K.argmax():4d}  {np.abs(c).max():9.0e}  '
          f'{spike_error:14.2f}  {np.abs(c_bump).max():8.0e}  '
          f'{bump_error:13.2f}')
```

```
 N  peak  spike |c|  spike residual  bump |c|  bump residual
 1     0      7e-01            0.99     7e+00           0.90
 4    17      1e+02            0.97     1e+03           0.55
16    20      2e+11            0.89     8e+10           0.03
```

$N = 1$ では、カーネルはラグ 0 でピークをとり、どちらの目標も捉えられない。$N = 16$ では、ピークはラグ 20 へと移動し、滑らかなバンプには相対誤差 0.03 で適合する一方、鋭いスパイクの近似精度は依然として低い（誤差 0.89）。しかし、`|c|` に注目してほしい。実指数関数でこの適合を実現するには、符号が交互に入れ替わりほぼ相殺し合う 10^{11} 程度の読み出し重みが必要となる。これは実指数関数 $e^{-n\Delta k}$ が基底として極端に悪条件であるためである。単位スケール程度の重みから出発する勾配降下法ではこのような相殺には到達できないため、実数値 SSM のカーネルは時間ウィンドウにわたる広い平均にとどまる（対照的に、振動を伴う複素レート $\lambda = -\alpha + i\omega$ はフーリエ基底に似た直交基底を与える。演習問題 3 を参照）。

12.4.2 対角状態空間層

12.4.2.1 S4D 層

S4D 層は、N 個の実状態を d 個の各チャネルごとに保持する。レートを $\lambda_n = -e^{\theta_n} < 0$（`log_neg_rate`）、ステップサイズを `log_dt` によってパラメータ化し、$\mathbf{B} = \mathbf{1}$ に固定したうえで（積 $b_n c_n$

のみが Equation 12.23 に現れ、$\mathbf{C}$ が学習されるため)、$(\bar{\mathbf{a}}, \bar{\mathbf{b}})$ を `discretize` 内で Equation 12.22 により計算する。

```
class S4D(nn.Module):
    """A diagonal state space layer: N states per channel, fixed dynamics."""
    def __init__(self, num_hiddens, num_states=16, dt_min=1e-3, dt_max=1e-1):
        super().__init__()
        d, N = num_hiddens, num_states
        # lambda_n = -n for n = 1..N, stored as log(-lambda) to stay negative
        self.log_neg_rate = nn.Parameter(
            torch.log(torch.arange(1., N + 1)).repeat(d, 1))
        self.log_dt = nn.Parameter(  # Log-uniform in [dt_min, dt_max]
            torch.rand(d, 1) * math.log(dt_max / dt_min) + math.log(dt_min))
        self.C = nn.Parameter(torch.randn(d, N) / math.sqrt(N))
        self.D = nn.Parameter(torch.ones(d))

    def discretize(self):
        """Zero-order hold with B = 1: decays a_bar and input weights b_bar."""
        rate = -torch.exp(self.log_neg_rate)  # (d, N)
        dt_rate = torch.exp(self.log_dt) * rate
        return torch.exp(dt_rate), torch.expm1(dt_rate) / rate
```

訓練時の順伝播では、すべてのチャネルについて長さ T のカーネル（Equation 12.23）を構成し、ゼロパディングを施した長さ $2T$ の FFT によって因果畳み込みを実行する（これにより巡回的な回り込みを防ぐ)。

```
@d2l.add_to_class(S4D)
def forward(self, X):  # X: (batch_size, num_steps, d)
    T = X.shape[1]
    a_bar, b_bar = self.discretize()
    # All powers a_bar^k for k < T: d * N * T numbers held in memory
    powers = a_bar.unsqueeze(-1) ** torch.arange(T, device=X.device)
    K = ((self.C * b_bar).unsqueeze(-1) * powers).sum(1)  # Kernel, (d, T)
    Y = torch.fft.irfft(torch.fft.rfft(X, n=2 * T, dim=1) *
                        torch.fft.rfft(K.T, n=2 * T, dim=0), n=2 * T, dim=1)
    return Y[:, :T] + self.D * X
```

自己回帰生成の際には、`step` が $d \times N$ の再帰状態を 1 トークン分だけ、$O(dN)$ の計算時間と定数メモリで進める。

```
@d2l.add_to_class(S4D)
def step(self, x, h):
    """One step: x is (batch_size, d); the state h is (batch_size, d, N)."""
    a_bar, b_bar = self.discretize()
    h = a_bar * h + b_bar * x.unsqueeze(-1)
    return (h * self.C).sum(-1) + self.D * x, h
```

FFT 畳み込み、並列の `linear_scan`、および逐次的な `step` による評価が、単精度浮動小数点の許容誤差の範囲で一致することを確かめてみよう。

```
layer = S4D(8, num_states=4)
X = torch.randn(2, 50, 8)  # (batch_size, num_steps, d)
with torch.no_grad():
    a_bar, b_bar = layer.discretize()
    H = d2l.linear_scan(a_bar, b_bar * X.unsqueeze(-1))  # (batch, T, d, N)
    Y_scan = (H * layer.C).sum(-1) + layer.D * X
    h, Y_step = torch.zeros(2, 8, 4), []
    for x in X.unbind(1):
        y, h = layer.step(x, h)
        Y_step.append(y)
    Y_conv = layer(X)
Y_step = torch.stack(Y_step, 1)
print(f'convolution vs scan: {(Y_conv - Y_scan).abs().max():.1e}, '
      f'convolution vs steps: {(Y_conv - Y_step).abs().max():.1e}')
```

```
convolution vs scan: 2.4e-07, convolution vs steps: 2.4e-07
```

3 つの見方はすべて $\sim 10^{-7}$ の精度で一致する。ここで注目すべき点として、LTI 層では FFT 畳み込みを用いることで、バッチ全体にわたる 4 次元の (B, T, d, N) 中間状態テンソルを実体化せずに済み、(d, N, T) のカーネルのべき乗のみを保持すればよい。

12.4.2.2 長距離想起

初期の時間スケールが長距離記憶を左右するかどうかを検証するため、12.1 節の遅延想起ベンチマークを再び取り上げ、遅延 $T = 100$ および $T = 1{,}000$ ステップで実験する。ここでは同一の空白フィラー（`blank_fillers=True`）を用い、内容に基づくフィルタリング（12.5 節）ではなく純粋な保持期間をテストできるようにする。

層の種類と初期の時間スケールを掛け合わせた 4 つの設定を比較する（Figure 12.9）。

1. **デフォルトの `minGRU`**：ゲートが $1/2$ 付近から出発する。
2. **デフォルトの `S4D`**：$d = 64, N = 16$、$\Delta \in [0.001, 0.1]$ であり、半減期は最大 693 ステップに達する。
3. **`S4D, short timescales`**：$\Delta \in [0.1, 1.0]$ で初期化し、初期の半減期がいずれも 7 ステップを超えないようにする。
4. **`minGRU, S4D timescales`**：更新ゲートのバイアスを `init_timescales` によって初期化し、$\sigma(b) = e^{-\Delta}$ が $\Delta \in [0.001, 0.1]$ に対して成り立つようにすることで（$b = -\log(e^{\Delta} - 1)$）、S4D の各チャネルの最も遅い状態に合わせる。

```
def init_timescales(layer, dt_min=1e-3, dt_max=1e-1):
    """Start a minGRU's gates at the decays exp(-dt) of S4D."""
    dt = dt_min * (dt_max / dt_min) ** torch.rand(layer.W_z.bias.shape)
    with torch.no_grad():  # sigmoid(b) = exp(-dt)
        layer.W_z.bias.copy_(-torch.log(torch.expm1(dt)))
    return layer
```

各モデルにつき 2 つのシードを `d2l.train_delayed_recall`（1,500 ステップの Adam、学習率 3×10^{-3}）で訓練し、訓練後の最も遅い半減期 $\ln 2 / \min_{d,n}(\Delta|\lambda_n|)$ も 2 種類の S4D について出力する。

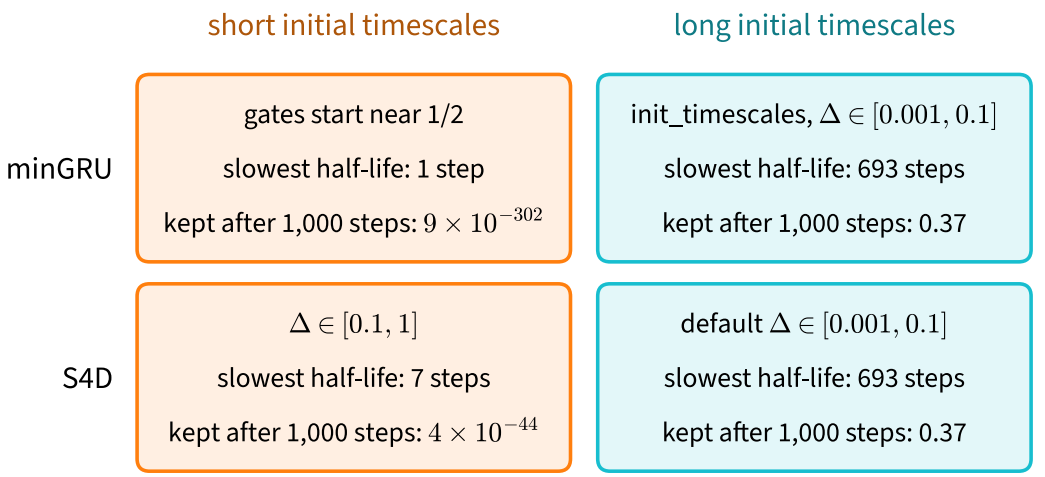


Figure 12.9: 長距離想起実験における 4 つの設定。2 種類の層（minGRU と S4D）のそれぞれに対し、短い初期時間スケールと長い初期時間スケールを組み合わせる。各セルには、初期化時の最も遅い半減期と、1,000 ステップの遅延後にその最も遅い状態が保持している格納値の割合を示す。時間スケールが結果を決めるなら列ごとに差が現れ、層の種類が決めるなら行ごとに差が現れる。

```
# (layer, embedding width): S4D keeps its width; the minGRUs map 32 to 64
layers = {
    'minGRU': lambda: (d2l.MinGRU(32, 64), 32),
    'S4D': lambda: (S4D(64), 64),
    'S4D, short timescales': lambda: (S4D(64, dt_min=0.1, dt_max=1.0), 64),
    'minGRU, S4D timescales': lambda: (
        init_timescales(d2l.MinGRU(32, 64)), 32)}

def recall(make_layer, T):
    """Train a fresh model at delay T; return its test accuracy and layer."""
    layer, num_inputs = make_layer()
    model = d2l.RecallModel(layer, 64, num_inputs=num_inputs)
    return d2l.train_delayed_recall(model, T, blank_fillers=True), layer

def slowest_half_life(layer):  # ln 2 / min(dt |lambda|), in steps
    rates = torch.exp(layer.log_dt + layer.log_neg_rate).detach()
    return math.log(2) / rates.min().item()

runs = {name: {T: [recall(make_layer, T) for _ in range(2)]  # Two runs each
               for T in [100, 1000]} for name, make_layer in layers.items()}
print(f"{'accuracy':23s} {'T=100':^11s}  {'T=1000':^11s}")
for name, by_T in runs.items():
    print(f'{name:23s}', '  '.join(' '.join(f'{acc:5.2f}' for acc, _ in r)
                                    for r in by_T.values()))
print('slowest half-life after training, in steps')
for name in ['S4D', 'S4D, short timescales']:
    print(f'{name:23s}', '  '.join(' '.join(
```

```
            f'{slowest_half_life(layer):5.0f}' for _, layer in r)
            for r in runs[name].values()))
```

```
accuracy                    T=100       T=1000
minGRU                   0.12  0.12   0.13  0.13
S4D                      1.00  1.00   1.00  1.00
S4D, short timescales    1.00  1.00   0.12  0.12
minGRU, S4D timescales   1.00  1.00   1.00  1.00
slowest half-life after training, in steps
S4D                      1028  1190   3020  2552
S4D, short timescales     119   196      7     7
```

結果は層の種類ではなく初期の時間スケールに従っている。$T = 1{,}000$ では、S4D と minGRU, S4D timescales の両方がすべての試行で 100% の正解率を達成するのに対し、デフォルトの minGRU と S4D, short timescales はランダム予測の水準（0.12）にとどまったままである。なお、S4D, short timescales が $T = 100$ では成功する理由に注目しよう。初期の最も遅い半減期は 7 ステップにすぎないものの、99 ステップにわたる減衰は $e^{-0.1\times 99} \approx 5 \times 10^{-5}$ であり、これは float32 の精度を上回るため、Adam が最も遅い半減期を 100〜200 ステップまで引き伸ばすのに十分な勾配信号が供給される。しかし、999 ステップにわたると $e^{-100} \approx 10^{-44}$ となって機械精度を下回って消失し、最も遅い半減期は 7 ステップ付近で凍りついたままとなる。

とはいえ、時不変なダイナミクスには根本的な死角がある。ラグ k にあるすべてのトークンは、それが決定的に重要な記号であれ情報を持たないフィラーであれ、まったく同じ重み K_k を受け取ってしまうのである。12.5 節では、各トークンが自身のステップサイズ Δ_t を選べるようにすることで、S4D の多重時間スケールの状態バンクと入力依存の選択機構を組み合わせる。

12.4.3 まとめ

連続時間の対角状態空間モデル $\mathbf{h}'(t) = \mathbf{A}\mathbf{h}(t) + \mathbf{B}x(t)$（レート $\lambda_n < 0$）をステップサイズ Δ のゼロ次ホールドで離散化すると、減衰 $\bar{a}_n = e^{\Delta\lambda_n}$ と半減期 $\ln 2/(\Delta|\lambda_n|)$ を持つ線形漸化式が得られる。$\lambda_n = -n$ と初期化し、Δ を $[0.001, 0.1]$ にわたって対数一様に抽出することで、S4D の各チャネルは N 個の指数減衰状態を備え、その半減期は 1 ステップ未満からほぼ 700 ステップにまで及ぶ。パラメータが時不変であるため、S4D は推論時には $O(1)$ 状態の漸化式として、訓練時には FFT による因果畳み込みとして等価に評価できる。1,000 ステップの遅延想起では、長い時間スケールの減衰から出発すること（S4D であれ minGRU のゲートバイアスへ移し替えたものであれ）が学習を可能にする決定的な要因となるが、固定されたダイナミクスでは依然として無関係な入力を選択的に無視することはできない。

12.4.4 演習問題

1. [code] **離散化規則**。ゼロ次ホールドは、Equation 12.20 を漸化式に変換するいくつかの方法の 1 つである。前進オイラー法は $\bar{a} = 1 + \Delta\lambda$、後退オイラー法は $\bar{a} = 1/(1 - \Delta\lambda)$、双一次法は $\bar{a} = (1 + \Delta\lambda/2)/(1 - \Delta\lambda/2)$ を用いる（38.2.3 節では、これら 2 つのオイラー法を微分方程式のソルバとして扱う）。
    1. 各規則について、どのようなステップサイズ Δ であれば、すべての $\lambda < 0$ に対して $|\bar{a}| < 1$ となるかを求めよ。安定な連続系を不安定にし得る規則はどれか。
    2. $\lambda = -1$ の後退オイラー法が、更新式 $h_t = z h_{t-1} + (1 - z)x_t$（ただし $z = \sigma(-\log\Delta)$）を与えること、すなわちステップサイズの負の対数を更新ゲートの活性化前の値とする GRU の混合式になることを示せ。
    3. $\lambda = -1$ および $\Delta \in \{0.1, 1, 3\}$ に対して、単位ステップ入力に対する各規則の応答をシミュレーションし、厳密解 $1 - e^{-t}$ と比較せよ。

2. [code] **遅延とカーネル**。`fit_kernel` を用いて、N 個の実指数関数がどのような記憶を表現できるかを調べよ。
   1. 遅延 5、20、50 における単位スパイクに対し、$N = 16$ 個の状態と $\Delta = 0.05$ で当てはめよ。遅延に応じて誤差はどのように変化するか。最も遅い状態と最も速い状態の半減期に関連付けて答えよ。
   2. 遅延 20 のスパイクに対し、ステップサイズ Δ を 0.01 から 0.2 の間で変化させて当てはめよ。どの Δ が最もよく適合するか。この結果は、特定の長さの記憶を必要とするチャネルのステップサイズを、訓練された S4D がどのように設定すべきかについて何を示唆するか。
   3. $\mathbf{c}$ を最小二乗法の代わりにゼロからの勾配降下法（Adam、数千ステップ）で当てはめよ。バンプとスパイクのそれぞれについて、最小二乗法の誤差にどこまで近づくか。最小二乗法で当てはめた重みから何が予測されるか。
3. [code] **複素レート**。複素レート $\lambda = -\alpha + i\omega$（$\alpha > 0$）を持つ状態は、回転しながら減衰する。
   1. ゼロ次ホールドにより $\bar{a} = e^{-\alpha\Delta} e^{i\omega\Delta}$ となること、および共役なレート・入力重み・読み出し重みを持つ 2 つの状態が実カーネル $K_k = r\, e^{-\alpha\Delta k}\cos(\omega\Delta k + \varphi)$（ただし $r \geq 0$、φ はある定数）として寄与することを示せ。このようなペアは何個の実パラメータを持つか。また、実状態のペアは何個持つか。
   2. 実レートでは表現できず、このようなペアで表現できるカーネルはどのようなものか。一定の周期で繰り返す入力を考えてみよ。
   3. `fit_kernel` を変更し、複素レート $\lambda_m = -1/2 + i\pi m$（$m = 0, \ldots, N-1$、Gu, Gupta, et al. (2022) の S4D-Lin 初期化）を用いてカーネルの実部を当てはめるようにせよ。遅延 20 のスパイクについて、誤差と重みの大きさを実レートによる当てはめと比較し、その違いを説明せよ。
4. [code] **FFT による畳み込み**。`S4D` の順伝播では、FFT の前に系列とカーネルを長さ $2T$ までパディングする。
   1. パディングを取り除いて FFT の長さを T にし、その出力を `step` の出力と比較せよ。どの出力が誤っているか、またそこに何が加算されているか。
   2. 厳密な因果畳み込みを与える最短のパディング長はいくつか。実装において、それを 2 のべき乗に切り上げることが多いのはなぜか。

12.5 選択的状態空間モデル

12.4 節の S4D 層は、ラグ k にあるすべてのトークンを同一に扱う。すなわち、k ステップ前の入力が段落の主要な名詞であろうとカンマであろうと、現在の出力には固定された畳み込み重み K_k で寄与する。しかし自然言語には**内容に基づく選択**（content-based selection）が不可欠である。大半のトークンはすぐに破棄してよい一方、少数の重要なトークンは強く書き込み、文ごとに異なる位置から数百ステップにわたって保持しなければならない。状態空間モデル（state space model、SSM）は、何をどのくらいの期間保持するかをトークンごとに決定できるだろうか。

前節までに登場した 2 つのモデルは、それぞれこの問いに対する答えの半分を備えている。

- `MinGRU`（12.3 節）は現在の入力トークンから減衰ゲートを計算するため、あるトークンを保持して次のトークンで上書きできるが、ゲートの初期値は 1 ステップの半減期であり、各チャネルは単一のスカラー状態しか持たない。
- `S4D`（12.4 節）は初期状態で 1 から ~ 700 ステップにわたる半減期をチャネルあたり N 個の状態に持つが、そのダイナミクスは時不変（time-invariant）である。

選択的状態空間モデル（selective state space model）は Gu and Dao (2023) が Mamba アーキテクチャとともに導入したものであり、これら 2 つを統合する。離散化された SSM のステップサ

イズ Δ_t は、書き込みベクトル $\mathbf{B}_t$ および読み出しベクトル $\mathbf{C}_t$ とともに、現在の入力 $\mathbf{x}_t$ の関数となる。$\Delta_t \to 0$ にすると状態は凍結され、情報を持たないフィラートークンは無視される。一方、Δ_t を大きくすると重要なトークンが書き込まれ、それまで状態が保持していた内容は減衰する。ダイナミクスがステップごとに変化するため FFT 畳み込みはもはや適用できないが、12.3 節の並列結合スキャン（parallel associative scan）を用いれば、依然として $O(\log_2 T)$ ラウンドで層を訓練できる。以下では、選択的コピーにおいて `S4D` と `MinGRU` をベンチマーク評価し、1 状態の選択的 SSM が `MinGRU` のゲートと数学的に等価であることを示す。続いて、`SelectiveSSM` と `Mamba` を省メモリなカスタム逆方向スキャン（`adjoint_scan`）によって構築し、訓練済みの Mamba 層がどのように状態を形成するかを調べ、最後にバイトレベルの WikiText-103 で評価する。

12.5.1 選択的コピー

Gu and Dao (2023) による合成ベンチマークの**選択的コピー**（selective copying）は、内容依存の保持能力を切り出して評価する（Figure 12.10）。ここでの設定では、系列は 1,024 個のフィラートークン（`0`）からなり、その中に 4 個のマーク付き記号（トークン `2..9` から一様に抽出）を**ランダム**な位置に挿入し、末尾に 4 個のクエリトークン（`1`）を続ける。j 番目のクエリトークンにおいて、モデルは j 番目のマーク付き記号を順番に出力しなければならない（ランダム予測の正解率は $1/8 = 0.125$）。記号の位置は系列ごとにランダムに変化するため、ラグ $T - s$ のみによって入力を重み付けする時不変フィルタは必ず失敗する。

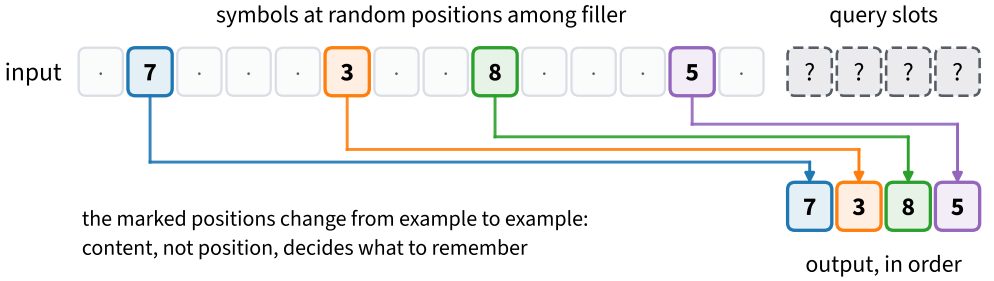


Figure 12.10: 選択的コピータスク。少数のマーク付き記号（色付き）が、系列ごとに変わる位置でフィラートークン（灰色）の中に散在している。クエリトークンの位置で、モデルはそれらの記号を順番に出力しなければならない。

```
def copy_batch(batch_size, num_fillers=1024, num_marked=4, num_symbols=8,
               device=None):
    """Symbols at random positions among fillers, then query tokens."""
    X = torch.zeros(batch_size, num_fillers + num_marked, dtype=torch.long,
                    device=device)
    X[:, num_fillers:] = 1  # Query tokens
    Y = torch.randint(2, 2 + num_symbols, (batch_size, num_marked),
                      device=device)
    positions = torch.rand(batch_size, num_fillers, device=device).argsort(1)
    X.scatter_(1, positions[:, :num_marked].sort(1).values, Y)
    return X, Y

X, Y = copy_batch(2, num_fillers=12)
print(X, Y, sep='\n')
```

```
tensor([[0, 0, 0, 7, 0, 2, 7, 0, 6, 0, 0, 0, 1, 1, 1, 1],
        [0, 0, 7, 7, 0, 0, 9, 5, 0, 0, 0, 0, 1, 1, 1, 1]])
tensor([[7, 2, 7, 6],
        [7, 7, 9, 5]])
```

選択的コピーでモデルを評価するため、`train_copy` は 1,500 ステップの Adam（`lr=3e-3`、バッチサイズ 64）にわたり `d2l.train_queries` を用いてネットワークを訓練し、新たに生成した 1,024 本のテスト系列に対するクエリ正解率を評価する。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32

def train_copy(model):
    """Train for 1,500 steps on fresh batches of selective copying; return
    the test accuracy."""
    num_steps = 1500
    device = d2l.try_gpu()
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=3e-3)
    d2l.train_queries(model, lambda step: copy_batch(64, device=device),
                      num_steps, optimizer)
    X, Y = copy_batch(1024, device=device)
    return float(torch.stack([d2l.query_accuracy(model, X_i, Y_i)  # In parts
                              for X_i, Y_i in zip(X.split(128), Y.split(128))
                              ]).mean())
```

ラッパ関数 `copy_runs` は、1 つのアーキテクチャを独立したシードで `num_runs=2` 回訓練し、パラメータ数とテスト正解率を表示して、訓練済みモデルを返す。

```
def copy_runs(name, make_model, num_runs=2):
    """Train num_runs fresh models; print their accuracies; return them."""
    models = [make_model() for _ in range(num_runs)]
    accuracies = [train_copy(model) for model in models]
    print(f'{name:20s} {d2l.num_params(models[0]):6,} parameters, accuracy '
          + ', '.join(f'{acc:.2f}' for acc in accuracies))
    return models
```

まず、2 ブロックの `d2l.SeqLM` モデル（幅 $d = 48$）において、時不変な `d2l.S4D` と入力選択的な `d2l.MinGRU` を系列ミキサとして比較ベンチマークする。

```
models = {}
for name, mixer in (('S4D', lambda: d2l.S4D(48)),
                    ('minGRU', lambda: d2l.MinGRU(48, 48))):
    models[name] = copy_runs(name, lambda: d2l.SeqLM(
        10, 48, [mixer() for _ in range(2)]))
```

```
S4D                  42,058 parameters, accuracy 0.17, 0.14
minGRU               48,202 parameters, accuracy 0.94, 0.94
```

予測どおり、時不変な `S4D` はランダム予測の水準付近（≤ 0.20）にとどまる一方、入力選択的な `minGRU` は実行ごとに約 0.75 から 0.95 の間に達する。ゲートの初期半減期が 1 ステップである（そのため 1,000 ステップの固定遅延想起では 12.4 節で失敗した）にもかかわらず、ここで `minGRU` が成功する理由に注目しよう。選択的コピーでは、マーク付き記号がたまたまクエリの近

くに配置されることがあり、そこから短いホライズンの勾配が得られるため、より長いフィラー区間を越えて記号を保持するようゲートが段階的に学習していくのである（演習問題 6）。

12.5.2 選択的ダイナミクス

12.5.2.1 入力依存のステップサイズ

選択的 SSM を 2 段階で構築する。まず、単一の状態（$N = 1$）とレート $\lambda = -1$ を持つ単一チャネルを、ゼロ次ホールド（zero-order hold、ZOH、12.4.1.2 節）のもとで考えよう。その更新式は $h_t = e^{-\Delta} h_{t-1} + (1 - e^{-\Delta}) x_t$ である。ここで Δ を固定パラメータとして扱う代わりに、$\Delta_t = \mathrm{softplus}(s_t) = \log(1 + e^{s_t}) > 0$ を現在のトークンの線形射影 $s_t = \mathbf{w}^\top \mathbf{x}_t + b$ から計算する。$-\Delta_t$ の指数をとると、厳密な代数的恒等式が得られる。

$$e^{-\Delta_t} = e^{-\log(1+e^{s_t})} = \frac{1}{1 + e^{s_t}} = \sigma(-s_t). \tag{12.24}$$

$z_t = \sigma(-s_t)$ とおくと、ZOH の更新式は $h_t = z_t h_{t-1} + (1 - z_t) x_t$ となり、`MinGRU` の厳密な凸結合と一致する（12.3.1.3 節、Gu and Dao (2023) の定理 1）。フィラートークンが $s_t = -4.6$ を生じるとき、$\Delta_t \approx 0.01$ かつ $z_t = 0.99$ となり（状態の 99% を保持し、半減期は 69 ステップ）、マーク付き記号が $s_t = 2$ を生じるとき、$\Delta_t \approx 2.1$ かつ $z_t \approx 0.12$ となる（新しい記号で状態を上書きする）。また、$\Delta_t = \mathrm{softplus}(s_t)$ を介してゲートをパラメータ化すると、S4D の多重時間スケールの初期化も直ちに適用できる。バイアス b を $\mathrm{softplus}(b) \in [0.001, 0.1]$ となるように選べば、ゲートの初期半減期は 7 から 693 ステップとなり、さらに重要な点として、単一のチャネルステップサイズ $\Delta_{t,c}$ によって N 個の状態（それぞれ異なるレート $\lambda_1, \dots, \lambda_N$ を持つ）を同時に駆動できる。

12.5.2.2 書き込みベクトルと読み出しベクトル

次に、層の H 個のチャネル（Mamba では $H = 2d$）のそれぞれに、N 次元の状態ベクトル $\mathbf{h}_t^{(c)} \in \mathbb{R}^N$ を与え、レート $\lambda = (\lambda_1, \dots, \lambda_N)^\top$ を $(-1, \dots, -N)^\top$ に初期化する。現在の H 次元入力 $\mathbf{x}_t$ から、層はチャネルごとのステップサイズ $\Delta_t \in \mathbb{R}_+^H$ と、全チャネルで共有される書き込み方向 $\mathbf{B}_t \in \mathbb{R}^N$ および読み出し方向 $\mathbf{C}_t \in \mathbb{R}^N$ を計算する。

$$\begin{aligned} \Delta_t &= \mathrm{softplus}(\mathbf{W}_\Delta \mathbf{x}_t + \mathbf{b}_\Delta), \\ \mathbf{B}_t &= \mathbf{W}_B \mathbf{x}_t, \qquad \mathbf{C}_t = \mathbf{W}_C \mathbf{x}_t. \end{aligned}$$

続いて各チャネル $c \in \{1, \dots, H\}$ は、自身の N 個の状態を更新し、次式によって出力を読み出す。

$$\begin{aligned} \mathbf{h}_t^{(c)} &= e^{\Delta_{t,c} \lambda} \odot \mathbf{h}_{t-1}^{(c)} + \Delta_{t,c}\, x_{t,c}\, \mathbf{B}_t, \\ y_{t,c} &= \mathbf{C}_t^\top \mathbf{h}_t^{(c)} + D_c\, x_{t,c}. \end{aligned} \tag{12.25}$$

Figure 12.11 は 3 つの射影がそれぞれどこに作用するかを示している。Δ_t は状態の各行をチャネルごとにスケーリングする一方、$\mathbf{B}_t$ と $\mathbf{C}_t$ は列方向に作用し、すべてのチャネルで共有される。Gu and Dao (2023) に従い、Equation 12.25 では厳密な ZOH 減衰 $e^{\Delta_{t,c}\lambda}$ を維持する一方、ZOH の入力重み $(e^{\Delta\lambda} - 1)/\lambda$ を一次のオイラー重み $\Delta_{t,c}$ で近似している（これは $\Delta|\lambda| \ll 1$ のときに一致し、大きなステップ $\Delta_{t,c}$ において $1/|\lambda|$ を超える重みでの書き込みを可能にする。演習問題 1 を参照）。Equation 12.25 に現れるすべての量の形状を以下にまとめる。

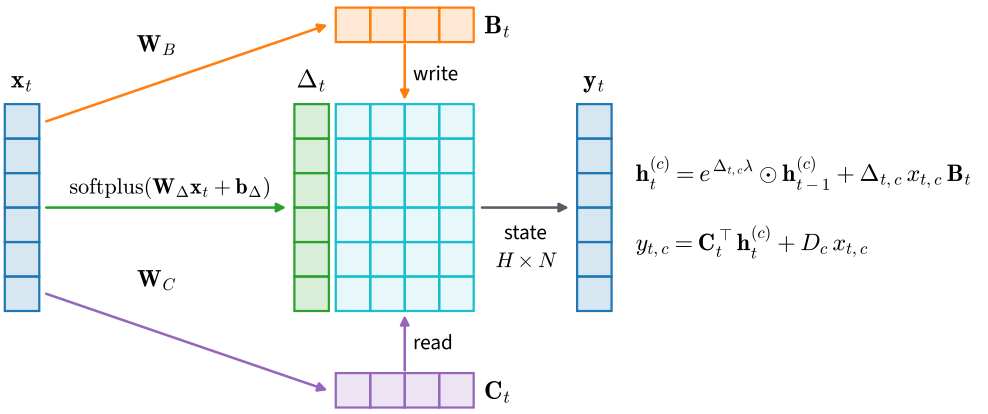


Figure 12.11: 選択的 SSM の 1 ステップ。入力 $\mathbf{x}_t$（H チャネル）から、3 つの射影によってステップサイズ Δ_t（チャネルごとに 1 つ）、書き込みベクトル $\mathbf{B}_t$、読み出しベクトル $\mathbf{C}_t$（いずれも長さ N）を計算する。第 c 行（$H \times N$ 状態の行）は $e^{\Delta_{t,c}\lambda}$ だけ減衰して $\Delta_{t,c}\, x_{t,c}\, \mathbf{B}_t$ を受け取り、出力 $y_{t,c}$ はその行を $\mathbf{C}_t$ で読み出してスキップ項 $D_c\, x_{t,c}$ を加える。

量	形状	計算方法
入力 $\mathbf{x}_t$	H	ステップ t における層の入力
ステップサイズ Δ_t	H（チャネルごとに 1 つ）	$\mathbf{x}_t$ からランク R 分解された $\mathbf{W}_\Delta \in \mathbb{R}^{H\times H}$ を経由
書き込み・読み出しベクトル $\mathbf{B}_t$、$\mathbf{C}_t$	各 N（全チャネルで共有）	$\mathbf{x}_t$ から $\mathbf{W}_B, \mathbf{W}_C \in \mathbb{R}^{N\times H}$ を経由
レート λ	チャネルあたり N	$-1, \dots, -N$ に初期化された学習パラメータ
状態 $\mathbf{h}_t^{(1)}, \dots, \mathbf{h}_t^{(H)}$	$H \times N$	Equation 12.25 の漸化式
出力 $\mathbf{y}_t$	H	$\mathbf{C}_t$ による読み出しとスキップ項 $D_c x_{t,c}$ の和

12.5.3 Mamba 層

12.5.3.1 射影・畳み込み・ゲート

`Mamba` ブロック (Gu and Dao 2023) は、チャネル混合の射影、幅 4 のデプスワイズ因果畳み込み（depthwise causal convolution）、および SwiGLU 風の乗法的な出力ゲートの内側に選択的 SSM を組み込む（Figure 12.12）。

1. 入力射影 $\mathbf{W}_{\text{in}}$ が、d 次元の入力を幅 $H = 2d$ の 2 つのブランチに拡張する。
2. 最初のブランチは、選択的 SSM に入る前に幅 4 のデプスワイズ因果畳み込みと SiLU 活性化関数を通過する。これにより、$\Delta_t, \mathbf{B}_t, \mathbf{C}_t$ は孤立した単一トークンではなく 4 トークンの局所ウィンドウに条件付けられる。
3. SSM の出力は、もう一方のブランチの SiLU 活性化値と要素ごとに掛け合わされた後、幅 d へ出力射影 $\mathbf{W}_{\text{out}}$ によって戻される。
4. パラメータ数を節約するため、$\mathbf{W}_\Delta \in \mathbb{R}^{H\times H}$ はボトルネックランク $R = \lceil d/16 \rceil$ を経由して分解される。

12.5.3.2 スキャンによる訓練

Equation 12.25 は依然として各ステップにおけるアフィン漸化式 $\mathbf{h}_t = \mathbf{a}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_t$（$H \times N$ の状態上）であるため、`d2l.linear_scan` によって全 T ステップの状態を並列に計算できる。しかし、状態をトークンあたり $H \times N$ 個の数値に拡張すると、活性化メモリは $2N$ 倍に増大する。倍増スキャンの $\lceil \log_2 T \rceil$ ラウンドを素朴に自動微分すると、$\sim 2\lceil \log_2 T \rceil$ 個もの形状 (B, T, H, N) の

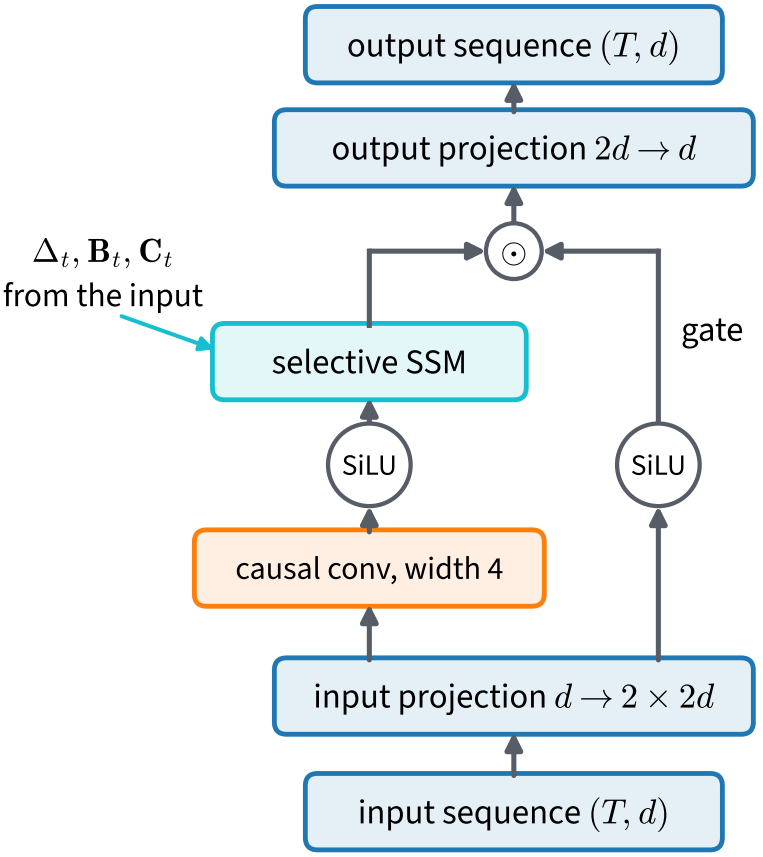


Figure 12.12: Mamba 層。入力は幅 $H = 2d$ の 2 つのブランチへ射影される。一方は短い因果畳み込み、SiLU、および選択的 SSM（ステップサイズと書き込み・読み出しベクトルをその入力から計算する）を通過し、もう一方は SiLU を通過してその結果をゲート制御する。最後の射影で幅 d に戻す。

```
class AdjointScan(torch.autograd.Function):
    """linear_scan whose backward pass is a second scan, run backward."""
    @staticmethod
    def forward(ctx, a, b):
        h = d2l.linear_scan(a, b)
        ctx.save_for_backward(a, h)
        return h

    @staticmethod
    def backward(ctx, grad_h):
        a, h = ctx.saved_tensors
        # g_t = grad_h_t + a_{t+1} g_{t+1}: a linear recurrence from the end
        a_next = torch.cat([a[:, 1:], torch.ones_like(a[:, :1])], 1)
        g = d2l.linear_scan(a_next.flip(1), grad_h.flip(1)).flip(1)
        h_prev = torch.cat([torch.zeros_like(h[:, :1]), h[:, :-1]], 1)
        return g * h_prev, g

def adjoint_scan(a, b):
    """All states of h_t = a_t * h_{t-1} + b_t, differentiated by a scan."""
    return AdjointScan.apply(a.expand_as(b), b)
```

長さ $T = 100$ のランダム系列上で、`adjoint_scan` の微分が `d2l.linear_scan` を通した標準の自動微分と float32 の精度で一致することを検証する。

```
a = torch.rand(2, 100, 8, requires_grad=True)
b = torch.randn(2, 100, 8, requires_grad=True)
w = torch.randn(2, 100, 8)  # A loss that weights every state
grads_scan = torch.autograd.grad((d2l.linear_scan(a, b) * w).sum(), (a, b))
grads_adjoint = torch.autograd.grad((adjoint_scan(a, b) * w).sum(), (a, b))
deviation = max(float((g1 - g2).abs().max())
                for g1, g2 in zip(grads_scan, grads_adjoint))
print(f'max deviation: {deviation:.1e}')
```

```
max deviation: 1.4e-06
```

カスタムの逆時間スキャンによる勾配は、自動微分と $\sim 10^{-6}$ の精度で一致する。

12.5.3.3 実装

`SelectiveSSM` は Equation 12.25 を実装し、`dt_bias` を初期化して $\mathrm{softplus}(\text{dt_bias})$ が $[\text{dt_min}, \text{dt_max}] = [0.001, 0.1]$ 上で対数一様に分布するようにする。`selective=False` を渡すと、$\Delta_t, \mathbf{B}_t, \mathbf{C}_t$ はチャネルごとの学習定数に置き換えられる（まったく同じ Mamba ブロック内での LTI 対照モデルとなる）。

```
class SelectiveSSM(nn.Module):
    """A diagonal SSM with N states for each of num_channels channels. Its
    step sizes and write and read vectors depend on the input, or, with
    selective=False, are learned constants of each channel."""
    def __init__(self, num_channels, dt_rank, num_states=16, dt_min=1e-3,
                 dt_max=1e-1, selective=True):
        super().__init__()
```

```
        H, N, R = num_channels, num_states, dt_rank
        self.selective = selective
        self.log_neg_rate = nn.Parameter(  # Rates -1, ..., -N in every channel
            torch.log(torch.arange(1., N + 1)).repeat(H, 1))
        dt = dt_min * (dt_max / dt_min) ** torch.rand(H)
        dt_bias = dt + torch.log(-torch.expm1(-dt))  # softplus(dt_bias) = dt
        if selective:
            self.W_dt = nn.Sequential(  # Step size per channel, rank R
                nn.Linear(H, R, bias=False), nn.Linear(R, H))
            self.W_B = nn.Linear(H, N, bias=False)  # Write vector
            self.W_C = nn.Linear(H, N, bias=False)  # Read vector
            d2l.init_lecun(self)
            with torch.no_grad():  # Small up-projection, step sizes dt
                nn.init.uniform_(self.W_dt[1].weight, -R ** -0.5, R ** -0.5)
                self.W_dt[1].bias.copy_(dt_bias)
        else:  # Fixed dynamics: S4D with a learned write vector
            self.dt_bias = nn.Parameter(dt_bias)
            self.B = nn.Parameter(torch.ones(H, N))
            self.C = nn.Parameter(torch.randn(H, N) / math.sqrt(N))
        self.D = nn.Parameter(torch.ones(H))
```

次に、`log_coefficients` を追加してステップごとの対数減衰 $\Delta_{t,c}\lambda$ と書き込み $\Delta_{t,c}u_{t,c}\mathbf{B}_t$（形状 `(batch_size, T, H, N)`）を計算し、さらに層の順伝播を追加して `adjoint_scan` を実行し状態を $\mathbf{C}_t$ と縮約する。

```
@d2l.add_to_class(SelectiveSSM)
def log_coefficients(self, U):
    """Log-decays and writes, each of shape (batch_size, T, H, N)."""
    if self.selective:  # Step sizes (batch_size, T, H), write vectors
        dt, B = F.softplus(self.W_dt(U)), self.W_B(U).unsqueeze(-2)
    else:  # One step size and one write vector per channel
        dt, B = F.softplus(self.dt_bias), self.B
    b = (dt * U).unsqueeze(-1) * B
    log_a = dt.unsqueeze(-1) * -torch.exp(self.log_neg_rate)
    return log_a.expand_as(b), b

@d2l.add_to_class(SelectiveSSM)
def forward(self, U):  # U: (batch_size, num_steps, num_channels)
    log_a, b = self.log_coefficients(U)
    H = adjoint_scan(torch.exp(log_a), b)  # States, (batch_size, T, H, N)
    C = self.W_C(U).unsqueeze(-2) if self.selective else self.C  # Read
    return torch.einsum('...n,...n->...', H, C) + self.D * U  # Sum over n
```

`Mamba` モジュールは `SelectiveSSM` を Figure 12.12 の 2 ブランチ構造で包み、幅 d を $2H = 4d$ へ `W_in` によって拡張し、SSM ブランチに因果デプスワイズ畳み込み `conv` と SiLU を適用したうえで、`silu(Z)` でゲート制御し、`W_out` によって元の幅へ射影して戻す。

```
class Mamba(nn.Module):
    """The Mamba layer: a gated selective SSM between two projections."""
    def __init__(self, num_hiddens, num_states=16, expand=2, conv_width=4,
                 **ssm_args):
```

```
        super().__init__()
        H = expand * num_hiddens
        self.W_in = nn.Linear(num_hiddens, 2 * H, bias=False)
        self.conv = nn.Conv1d(H, H, conv_width, groups=H,  # One per channel
                              padding=conv_width - 1)
        self.W_out = nn.Linear(H, num_hiddens, bias=False)
        d2l.init_lecun(self)  # The SSM, built next, has its own init
        self.ssm = SelectiveSSM(H, math.ceil(num_hiddens / 16), num_states,
                                **ssm_args)

    def ssm_input(self, X):
        """The input of the SSM and the gate branch."""
        U, Z = self.W_in(X).chunk(2, -1)
        U = self.conv(U.transpose(1, 2))[..., :X.shape[1]]  # Keep it causal
        return F.silu(U.transpose(1, 2)), Z

    def forward(self, X):  # X: (batch_size, num_steps, num_hiddens)
        U, Z = self.ssm_input(X)
        return self.W_out(self.ssm(U) * F.silu(Z))
```

12.5.4 訓練済み Mamba モデル

12.5.4.1 Mamba による選択的コピー

ここで、選択的コピーにおいて Mamba（チャネルあたり $N = 4$ 状態）と、その非選択的アブレーションである Mamba, not selective（selective=False とし、まったく同じ射影、幅 4 の畳み込み、SiLU ゲートを維持したもの）を並べて訓練する。

```
for name, selective in (('Mamba', True), ('Mamba, not selective', False)):
    # Compiled, since eagerly each round of the scan would be its own kernel
    models[name] = copy_runs(name, lambda: d2l.SeqLM(10, 48, [
        torch.compile(Mamba(48, num_states=4, selective=selective))
        for _ in range(2)]))
```

```
Mamba                71,242 parameters, accuracy 0.99, 1.00
Mamba, not selective 70,090 parameters, accuracy 0.38, 0.29
```

Mamba はほぼすべての記号をコピーする。正解率は 95% 以上、通常は 100% 近くに達し、minGRU を上回る。一方、非選択的な対照モデルでは正解率が 30 パーセントポイント以上低下し、実行ごとのばらつきも大きくなる。これは Gu and Dao (2023) で報告されたアブレーション結果と一致する。射影、畳み込み、およびゲートは両モデルで同一であるため、この差は選択機構のみに起因する。演習問題 3 では、4,096 個のフィラーに対するゼロショットの長さ外挿（length extrapolation）を検証する。

12.5.4.2 状態への寄与

Mamba が**どのようにして**フィラーを無視しつつ 4 個のマーク付き記号を保持しているかを調べるため、最初のクエリトークンにおける最終状態を、それ以前の各位置 $s \le T$ からの寄与へと分解する。展開された漸化式 $\mathbf{h}_T = \sum_{s=1}^{T} (\prod_{r=s+1}^{T} \mathbf{a}_r) \odot \mathbf{b}_s$ において、位置 s の減衰後の項のユークリッドノルム $\| \exp(\sum_{r=s+1}^{T} \log \mathbf{a}_r) \odot \mathbf{b}_s \|_2$ は、位置 s での書き込みがステップ T の状態にどれだけ残存しているかを表す。

```
def state_contributions(log_a, b):
    """Size of each position's term in the final state of h = a h + b."""
    cum = log_a.cumsum(1)
    terms = torch.exp(cum[:, -1:] - cum) * b  # Decayed to the last step
    return torch.linalg.vector_norm(terms.flatten(2), dim=-1)  # (batch, T)
```

log_coefficients を d2l.S4D に追加し、ステップごとの対数減衰と書き込みを (batch_size, T, d, N) という SelectiveSSM と同じ形式で取り出せるようにする。

```
@d2l.add_to_class(d2l.S4D)
def log_coefficients(self, X):
    """Log-decays and writes of the recurrence, each (batch_size, T, d, N)."""
    rate = -torch.exp(self.log_neg_rate)
    log_a = torch.exp(self.log_dt) * rate  # log a_bar = dt * lambda
    b_bar = torch.expm1(log_a) / rate
    return log_a.expand(*X.shape, -1), b_bar * X.unsqueeze(-1)
```

2 つの層のどちらを見るべきだろうか。訓練時にはどの層で選択を行うべきかは何も指定されないため、両方の層を調べる。layer_contributions は指定された層より下のブロックを実行し、最初のクエリトークンまでで打ち切ったテスト系列 X（X[:, :-3]）に対するその層の対数減衰と書き込みを取り出して、state_contributions を計算する。これを S4D および訓練済みの両方の Mamba 実行結果における両層に適用する。

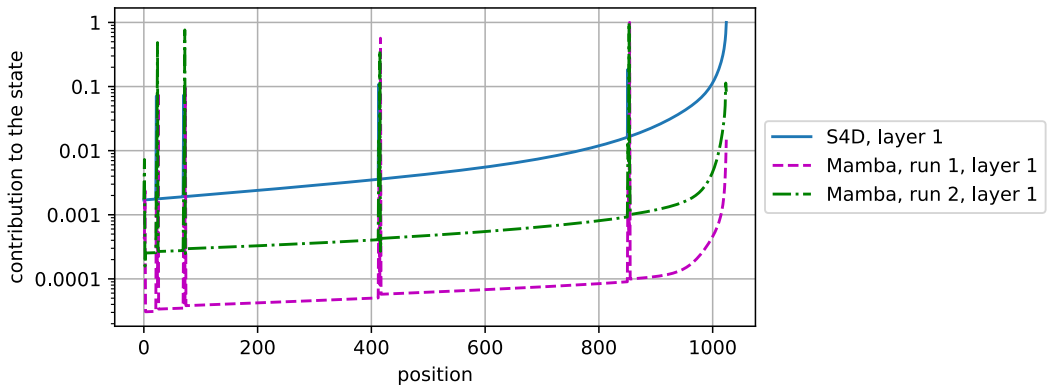
```
def layer_contributions(model, X, layer):
    """Contributions to the state of a layer at the last step of X."""
    H = model.embedding(X)
    for i in range(layer):  # The blocks below it, as in the forward pass
        H = H + model.mixers[i](model.norms[i](H))
        H = H + model.mlps[i](H)
    mixer, X = model.mixers[layer], model.norms[layer](H)
    if isinstance(mixer, d2l.S4D):
        return state_contributions(*mixer.log_coefficients(X))
    U, _ = mixer.ssm_input(X)  # Mamba: what its selective SSM reads
    return state_contributions(*mixer.ssm.log_coefficients(U))

X, _ = copy_batch(1, device=d2l.try_gpu())
X = X[:, :-3]  # Up to the first query token
probed = {'S4D': models['S4D'][0], 'Mamba, run 1': models['Mamba'][0],
          'Mamba, run 2': models['Mamba'][1]}  # Both trained Mamba runs
with torch.inference_mode():  # Both layers of each model
    contributions = {(name, layer): layer_contributions(model, X, layer)[0]
                     for name, model in probed.items() for layer in (0, 1)}
```

幅 4 の因果畳み込みは各マーク付き記号を最大 4 個の連続する位置に広げるため、symbol_windows で各記号から始まる 4 位置分のウィンドウを集める（系列全体の $\leq 1.6\%$）。各層について、最終状態のうちそれらのウィンドウに由来する割合を表示し、続いて各モデルについてその割合が大きい方の層における全 1,025 位置の寄与をプロットする。

```
def symbol_windows(X, width=4):
    """Positions of each symbol and the width - 1 after it, without repeats."""
```

```
    starts = (X[0] > 1).nonzero().flatten()  # The four symbols
    offsets = torch.arange(width, device=X.device)
    windows = (starts[:, None] + offsets).flatten()
    windows = windows.unique()  # Windows of nearby symbols may overlap
    return windows[windows < X.shape[1]]  # A window may reach past the end

windows = symbol_windows(X)
print(f'symbol windows: {len(windows) / X.shape[1]:.1%} of the positions')
share = {key: float(c[windows].sum() / c.sum())
         for key, c in contributions.items()}
print('share of the state from them  layer 1  layer 2')
for name in probed:
    print(f'{name:28s}  {share[name, 0]:7.2f}  {share[name, 1]:7.2f}')
best = {name: max((0, 1), key=lambda layer: share[name, layer])
        for name in probed}  # The layer that selects more
curves = {f'{name}, layer {layer + 1}': contributions[name, layer]
          for name, layer in best.items()}
d2l.plot(torch.arange(X.shape[1]), [(c / c.max()).cpu() for c in
                                    curves.values()],
         'position', 'contribution to the state',
         legend=list(curves), yscale='log', figsize=(6, 3),
         legend_loc='outside')  # The curves fill the axes
```

```
symbol windows: 1.6% of the positions
share of the state from them  layer 1  layer 2
S4D                             0.03     0.01
Mamba, run 1                    0.94     0.90
Mamba, run 2                    0.70     0.68
```



この割合は、1 本の系列内でたまたま記号がどの位置に落ちたかに依存する。局所的な指標となるのが**スパイク比**（spike ratio）、すなわち記号の 4 ステップウィンドウ内における最大寄与を、その直前のフィラートークンの寄与で割った値である。新たに生成した 64 本のテスト系列の記号全体にわたる中央値を、同様に両方の層で計算する。

```
X_test = copy_batch(64, device=X.device)[0][:, :-3]  # 64 new sequences
rows, starts = (X_test[:, 1:] > 1).nonzero(as_tuple=True)  # The symbols,
starts = starts + 1                                        # not at position 0
```

```
    window = (starts[:, None] + torch.arange(4, device=X.device)).clamp(
        max=X_test.shape[1] - 1)  # A window may reach past the end
    def spike_ratio(c):  # Median over the symbols
        return float((c[rows[:, None], window].amax(1) / c[rows, starts - 1])
                     .median())

    print(f'spike ratio, median over {len(rows)} symbols   layer 1   layer 2')
    with torch.inference_mode():
        for name, model in probed.items():
            spikes = [spike_ratio(layer_contributions(model, X_test, layer))
                      for layer in (0, 1)]
            print(f'{name:36s} {spikes[0]:9,.1f} {spikes[1]:9,.1f}')
```

```
spike ratio, median over 256 symbols   layer 1   layer 2
S4D                                       27.8       1.5
Mamba, run 1                           2,933.1   5,654.5
Mamba, run 2                           1,416.0   4,980.2
```

S4D のプロファイルは直近性に従う。系列の末尾に向かって滑らかに上昇し、記号ウィンドウが状態に占める割合はわずか数パーセントと、位置数の割合を少し上回る程度にすぎない。固定フィルタであってもトークンによって強く応答することはできるため、記号はその直前のフィラーより目立ちはするものの、高々数十倍にとどまり、そのトークンをより長く保持することはできない。一方 Mamba では、選択を担う層が記号を隣接するフィラーの数百倍から数万倍にまで引き上げ、4 つのウィンドウが状態の大部分（多くの場合は過半）を占める。どちらの層がこの役割を担うかは、訓練前にあらかじめ決まっているわけではない。両方の層が選択を行う実行もあれば、片方の層だけが行う実行もあり（どちらの層になる場合もある）、そのときもう一方の層のスパイク比は S4D と同程度にとどまる。表に両方の層を載せ、プロットにはより選択性の高い層を表示しているのはそのためである。

層はどのようにしてこのようなスパイクを生み出すのだろうか。自然な推測は、記号のたびに大きなステップをとることで、その記号を書き込むと同時にそれ以前のフィラーを一掃している、というものであろう。しかし、訓練済みモデルのステップサイズはそれとは異なる事実を示している。$H = 96$ 個のチャネル全体で平均すると、記号でのステップは通常フィラーでのステップと変わらない。選択はごく少数のチャネルの仕事であり、各記号は 2〜3 個のチャネルによって運ばれる。フィラー上では、これらのチャネルはステップ $\Delta_{t,c}$ を約 10^{-5} にとるため、減衰も書き込みも起こらず、状態は凍結される。一方、記号上では同じチャネルが約 0.1 から 1.5 の間のステップをとり、書き込み $\Delta_{t,c}\, x_{t,c}\, \mathbf{B}_t$ はステップに比例するため、記号は十分な強さで状態に入る。このようなステップはチャネルがそれまで保持していた内容も減衰させるが、その減衰はわずかな倍率にすぎず、直前のフィラーを消去することがスパイクに寄与する度合いは小さい。ここでの選択とは、状態のフラッシュ（一掃）ではなく、凍結された少数のチャネルへの強い書き込みなのである。

12.5.4.3 言語モデリング

最後に、Mamba、minGRU、attention をバイトレベルの WikiText-103 ベンチマーク（d2l.SeqLM、幅 $d = 256$ の 4 ブロック、128 バイトのウィンドウ、1,000 ステップ）で比較する。まず 128 バイトの WikiText103 ストリームを読み込む。

```
    data = d2l.WikiText103(batch_size=64, num_steps=128)
```

次に text_model(mixer, d) を定義し、4 つの実験条件 arms を設定する。これには、パラメータ数

を合わせた minGRU, d=309（パラメータ数 3.99M、Mamba のチャネル拡張後のパラメータ数と一致）も含める。

```
def text_model(mixer, d):  # 256 byte values; four blocks of width d
    return d2l.SeqLM(256, d, [mixer(d) for _ in range(4)]).to(d2l.try_gpu())

mamba = lambda d: torch.compile(Mamba(d))  # Compiled, as in the copying
min_gru = lambda d: d2l.MinGRU(d, d)
attention = lambda d: d2l.AttentionMixer(d, num_heads=8)
arms = {'Mamba': (mamba, 256), 'minGRU': (min_gru, 256),
        'attention': (attention, 256),
        'minGRU, d=309': (min_gru, 309)}  # As many parameters as Mamba
```

d2l.text_table を呼び出すと、各アーキテクチャを 2 つのシードで 1,000 ステップ訓練し、ウィンドウ内の位置帯ごとの検証損失とステップあたりのレイテンシを表にまとめる。

```
trained = d2l.text_table(text_model, arms, data)
```

```
                          validation        0-15       16-63      64-127   ms/step
Mamba           3,989,248 1.298/1.297 1.488/1.488 1.272/1.272 1.269/1.268 69/69
minGRU          2,764,544 1.349/1.346 1.523/1.521 1.324/1.321 1.325/1.321 27/27
attention       3,286,784 1.378/1.383 1.557/1.559 1.354/1.359 1.351/1.357 21/21
minGRU, d=309   3,991,918 1.325/1.328 1.509/1.509 1.300/1.302 1.298/1.301 27/28
```

Mamba は 1.30 nat/byte に達する。並列スキャンで訓練しながらも 12.1.4.2 節の完全な LSTM および GRU に匹敵し、同一幅の minGRU を 0.05 nat/byte、パラメータ数を合わせた minGRU, d=309 を 0.03 nat/byte 上回る。Mamba はトークンあたり $H \times N = 512 \times 16 = 8{,}192$ 個の数値を状態としてスキャンするため（サイズは 32× であり、minGRU の状態は 256 個）、非融合カーネルでの訓練ステップは 2×〜4× ほど minGRU より時間がかかる。ただし推論時には、その $32d$ 個の状態（幅 4 の畳み込みバッファの $6d$ 個を加えても）は、マルチヘッドアテンションのわずか 19 トークン分の KV キャッシュに等しい。さらに、Mamba の $H \times N$ 状態を外積による行列メモリ $\mathbf{S}_t$（$(\boldsymbol{\Delta}_t \odot \mathbf{x}_t)\mathbf{B}_t^\top$ で更新し $\mathbf{C}_t$ でクエリする）とみなす見方は、12.6 節の行列状態漸化式およびチャンク単位の行列積による訓練へと直結する。

12.5.5 まとめ

時不変の状態空間モデルは過去の入力をラグのみによって重み付けするため、選択的コピーのような内容選択的なタスクでは失敗する。選択的状態空間モデルは、各入力トークンからチャネルごとのステップサイズ $\boldsymbol{\Delta}_t = \mathrm{softplus}(\mathbf{W}_\Delta \mathbf{x}_t + \mathbf{b}_\Delta)$ と共有の書き込み・読み出しベクトル $\mathbf{B}_t, \mathbf{C}_t$ を計算する。単一の状態（$N = 1$）では、$e^{-\mathrm{softplus}(s_t)} = \sigma(-s_t)$ によって MinGRU の更新ゲートが復元される。また、N 個の状態をレート $-1, \ldots, -N$ に初期化し、対数一様な $\boldsymbol{\Delta}$ バイアスと組み合わせることで、Mamba は内容選択と多重時間スケールの記憶を両立させる。並列スキャンを逆時間スキャン（adjoint_scan）によって微分すれば、$O(\log_2 T)$ 個の倍増ラウンド中間テンソルの保存を回避できる。選択的コピーにおいて Mamba は正解率 95% 以上に達し（選択性を無効にすると > 30 ポイント低下する）、これはフィラー上で凍結されたまま各記号を大きなステップで書き込む少数のチャネルによるものである。さらにバイトレベルの WikiText-103 では、完全な LSTM および GRU に匹敵しつつ、パラメータ数を合わせた MinGRU を 0.03 nat/byte 上回る。

12.5.6 演習問題

1. **書き込み重みの近似**。Mamba はゼロ次ホールドの減衰 $e^{\Delta\lambda}$ を維持する一方、書き込みには重み Δ を用い、$(e^{\Delta\lambda}-1)/\lambda$ は用いない。
   1. $\lambda = -1$ のとき、これら 2 つの重みの差が 10% 未満となる最大の Δ を求めよ。
   2. $\lambda = -1$、書き込みベクトル 1、および厳密な重みを持つ 1 状態のモデルにおいて、0 から出発した状態が、0 とそれまでに見た入力値によって張られる区間内にとどまることを示せ。一方、入力が $[-1, 1]$ の値をとる系列で、重み Δ を用いた状態がその区間からはみ出すものを 1 つ見つけよ。訓練済み Mamba モデルの値を適切な範囲内に保っているものは何か。

2. **スキャンのメモリ量**。H チャネルと N 個の状態を持つ Mamba 層において、B 本の系列（長さ T）からなるミニバッチと 32 ビット数値を考える。誤差逆伝播法がスキャンのために保存するバイト数を、(a) 倍増スキャンの各ラウンドを通す場合、(b) 随伴スキャンを用いる場合、(c) 層の入力のみを保存して状態を再計算するカーネルを用いる場合のそれぞれについて数えよ。$H = 4096$、$N = 16$、$B = 8$、$T = 8192$ に対してこれら 3 つを計算し、GPU のメモリ容量と比較せよ。

3. [code] **より長い系列**。クエリの前に 1,024 位置がある設定で訓練した Mamba モデルを、再訓練せずに 2,048 および 4,096（`num_fillers`）の系列で評価せよ。
   1. モデルは依然として記号をコピーできるか。minGRU と比較せよ。
   2. より選択性の高い層の各チャネルについて、最も遅い状態の 1,000 フィラートークンにわたる減衰を計算せよ。4,096 個のフィラーを越えて記号を運べるチャネルは何個あるか。

4. [code] **どの部分が選択的であるべきか**。●
   1. 本節で扱った 2 つの Mamba モデルの中間には、さらに 2 つの変種が存在する。入力依存のステップサイズと固定された書き込み・読み出しベクトル $\mathbf{B}$ および $\mathbf{C}$ を持つモデルと、固定されたステップサイズと入力依存の $\mathbf{B}_t$ および $\mathbf{C}_t$ を持つモデルである。両方をそれぞれ 2 回ずつ訓練せよ。どちらがタスクを解けるか。その結果を上で測定したスパイク比と関連付けて説明せよ。
   2. 訓練済み Mamba モデルにおいて、各記号から始まる 4 位置（上で測定したウィンドウ）における両層のステップサイズを、フィラー位置全体にわたる中央値に置き換え、どちらの層も大きなステップで記号を書き込めないようにせよ。正解率はどれだけ低下するか。次に、それ以外のすべての位置におけるステップサイズをウィンドウ全体にわたる中央値に置き換え、どちらの層もフィラーをスキップできないようにして同じ実験を繰り返せ。さらに、1 層ずつ個別にこの介入を行え。結果は上で測定した寄与割合と整合するか。

5. [code] **生成**。メソッド `step` を `SelectiveSSM` と `Mamba` に追加し、SSM の状態と畳み込みの直近 3 入力を受け取って 1 トークンを処理し、出力と新しい状態を返すようにせよ。系列を 1 ステップずつ進めることで順伝播の出力が再現されることを確認せよ。層はトークンからトークンへ何個の数値を引き継ぐか。

6. [code] **短い初期記憶**。minGRU は、ゲートの初期記憶が約 1 ステップしかないにもかかわらずコピーを学習した。クエリに最も近い記号がゲートを訓練するという説明を検証せよ。
   1. 訓練前後のフィラートークンにおける minGRU のゲート値を記録し、さらに訓練済みモデルについて、記号とそのクエリの間の距離でソートしたうえで各クエリごとの正解率を個別に記録せよ。どの記号を最もよくコピーできるか。
   2. Mamba モデルのステップサイズを 0.5 から 1 の間（`dt_min=0.5, dt_max=1.0`）で初期化し、すべての状態が高々 2 ステップの初期記憶しか持たないようにして訓練せよ。

訓練予算内で記号のコピーを学習できるか。もし学習できるなら、第 1 層のフィラートークンにおけるステップサイズを訓練前後でプロットせよ。

3. クエリ前の位置数を 4,096（クエリ付近に記号が現れる頻度がより低くなる設定）に変え、さらに 12.4 節の長距離想起実験と同様に初期化した minGRU も含めて、両方の実験を繰り返せ。

12.6 行列状態漸化式

12.5 節の選択的状態空間モデル（selective state space model）は、何をどれだけ長く保持するかをトークンごとに決定するが、固定サイズの状態がいくつの異なる対応関係を保存して検索できるかはまだ問うていない。言語モデリングの多くは**連想想起**（associative recall）、すなわちキーに値を束縛し（`rate = 0.05` や「**Mira は船の機関士である**」など）、数百トークン後にそのキーが再出現した際に値を検索する能力を必要とする。Transformer は、過去のすべてのキーと値を増大し続ける KV キャッシュに保持し、クエリをそれらすべてと比較することで連想想起を解いている。固定サイズの再帰状態（recurrent state）でも同様にキー・値ペアを保存して検索できるのだろうか。また、その容量は状態サイズに対してどのようにスケールするのだろうか。

答えはイエスである。ただしそれは、状態が**行列** $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$ であり、外積（outer product）$\mathbf{v}_t \mathbf{k}_t^\top$ の加算によって更新され、行列・ベクトル積 $\mathbf{S}_t \mathbf{q}_t$ によってクエリされる場合である。この構造の具体例にはすでに 2 つ出会っている。非正規化線形アテンション（10.6 節）と、チャネル × 状態の $H \times N$ 行列を持つ Mamba（12.5 節）である。以下では、外積行列メモリの厳密な検索誤差と容量のスケーリング則（scaling law）を導出し、古くなったペアを忘却するための入力依存（input-dependent）のスカラー減衰を導入し、減衰付き行列漸化式と因果マスク付き線形アテンションの間の**状態空間双対性**（state space duality）を確立する。さらに、ハードウェア効率の高いチャンク単位の訓練アルゴリズム（`chunked_linear_attention`）を実装し、訓練された `GatedLinearAttention` 層がマルチクエリ連想想起において理論的な容量則に従うかどうかを検証する。

12.6.1 連想メモリ

12.6.1.1 ペアの書き込みと読み出し

最も単純な連想メモリ（associative memory）は、外積の和をとることでペアを保存する。$\mathbb{R}^2$ における簡単な例として、直交する単位キー $\mathbf{k}_1 = (1, 0)^\top$ と $\mathbf{k}_2 = (0, 1)^\top$ をスカラー値 $v_1 = 3$ と $v_2 = 5$ に束縛し、行ベクトル $\mathbf{S} = v_1 \mathbf{k}_1^\top + v_2 \mathbf{k}_2^\top = (3, 5)$ を作ってみよう。どちらのキーを掛けても、$\mathbf{S}\mathbf{k}_1 = 3$、$\mathbf{S}\mathbf{k}_2 = 5$ のように誤差なくその値を検索できる。しかし、3 つ目のペアとして $\mathbf{k}_3 = (1, 1)^\top / \sqrt{2}$ と $v_3 = 7$ を追加すると、$\mathbf{k}_3$ は $\mathbf{k}_1$ および $\mathbf{k}_2$ と重なりを持つ。その結果、$\mathbf{S}$ は $(3 + 7/\sqrt{2}, 5 + 7/\sqrt{2}) \approx (7.95, 9.95)$ となり、$\mathbf{k}_1$ でクエリすると 7.95 が返り、3 ではなくなる。キーが 2 次元のとき、クロストークなしで保持できる直交キーは高々 2 個である。

一般に、キーベクトル $\mathbf{k}_t \in \mathbb{R}^{d_k}$、値ベクトル $\mathbf{v}_t \in \mathbb{R}^{d_v}$、クエリベクトル $\mathbf{q}_t \in \mathbb{R}^{d_k}$ に対し、状態は行列 $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$ であり、各ステップで次の式により更新および読み出しされる。

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{v}_t \mathbf{k}_t^\top, \qquad \mathbf{y}_t = \mathbf{S}_t \mathbf{q}_t. \tag{12.26}$$

各書き込みでは、ランク 1 行列 $\mathbf{v}_t \mathbf{k}_t^\top$（方向 $\mathbf{k}_t$ を $\mathbf{v}_t$ に写す行列）を加算する。ここで、$(\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t)$ はトークン表現に対する学習された線形射影である。

12.6.1.2 線形アテンション

Equation 12.26 を $\mathbf{S}_0 = \mathbf{0}$ から展開すると、加法的な行列漸化式が非正規化の因果線形アテンション（linear attention）を計算していることがわかる（10.6 節、Katharopoulos et al. (2020)）。

$$\mathbf{y}_t = \Big(\sum_{s \le t} \mathbf{v}_s \mathbf{k}_s^\top\Big)\mathbf{q}_t = \sum_{s \le t} (\mathbf{k}_s^\top \mathbf{q}_t)\, \mathbf{v}_s .$$

右から結合して評価すると、各出力 $\mathbf{y}_t$ は過去のすべての値 $\mathbf{v}_s$ をドット積類似度 $\mathbf{k}_s^\top \mathbf{q}_t$ で重み付けする。一方、左から結合して評価すると、外積の和が固定サイズの再帰状態 $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$ として順方向に引き継がれていく。

12.6.1.3 干渉

クロストークが信号を圧倒するまでに、$\mathbf{S} = \sum_{i=1}^{n} \mathbf{v}_i \mathbf{k}_i^\top$ はいくつのペアを保存できるだろうか。保存済みの単位キー $\mathbf{k}_j$（$\|\mathbf{k}_j\| = 1$）でクエリすると、目的の値と他の $n-1$ 個のペアによる干渉とが分離される。

$$\mathbf{S}\mathbf{k}_j = \mathbf{v}_j + \sum_{i \ne j} (\mathbf{k}_i^\top \mathbf{k}_j)\, \mathbf{v}_i . \tag{12.27}$$

誤差ゼロの厳密な読み出しには互いに直交するキーが必要だが、$\mathbb{R}^{d_k}$ にはそのようなキーは高々 d_k 個しか存在しない。ランダムな単位キーについては、期待される干渉の大きさを厳密に計算できる。

命題： $\mathbf{k}_1, \dots, \mathbf{k}_n$ を $\mathbb{R}^{d_k}$ の単位球面上の一様分布に独立に従うベクトルとし、$\mathbf{v}_1, \dots, \mathbf{v}_n \in \mathbb{R}^{d_v}$ をキーとは独立に選ばれた単位ベクトル（unit vector）とする。このとき、保存された任意のキーにおける期待二乗検索誤差は次の式で与えられる。

$$\mathbb{E}\,\big\|\mathbf{S}\mathbf{k}_j - \mathbf{v}_j\big\|^2 = \frac{n-1}{d_k}. \tag{12.28}$$

証明： $\sum_{i \ne j} (\mathbf{k}_i^\top \mathbf{k}_j)\mathbf{v}_i$ の二乗ノルムを展開すると、$i \ne l$ の交差項は期待値をとると消える。なぜなら、$\mathbf{k}_j$ を条件付けたとき、$\mathbf{k}_i^\top \mathbf{k}_j$ と $\mathbf{k}_l^\top \mathbf{k}_j$ は独立で平均がゼロだからである。$\|\mathbf{v}_i\|^2 = 1$ より、$n-1$ 個の対角項 $\mathbb{E}[(\mathbf{k}_i^\top \mathbf{k}_j)^2]$ が残る。$\mathbb{R}^{d_k}$ の単位球面の回転対称性より、$\mathbf{k}_i$ の各座標の二乗期待値は $1/d_k$ であるため、$\mathbb{E}[(\mathbf{k}_i^\top \mathbf{k}_j)^2] = 1/d_k$ となる。干渉する $n-1$ 個のペアについて和をとれば $(n-1)/d_k$ が得られる。■

なお、検索誤差は負荷率 $(n-1)/d_k$ にのみ依存し、ペアが**いつ**書き込まれたかには依存しない。つまり、加法的なメモリがペアを失う原因は、時間経過ではなく過密である。

Equation 12.28 を NumPy で実験的に検証するため、`matrix_memory` は単位キーとコードブック値のペア n 個を $\mathbf{S} = \sum_{i=1}^{n} \mathbf{v}_i \mathbf{k}_i^\top$ に書き込み（オプションで共通方向の相関 ρ を付与できる）、256 個の単位ベクトルからなるコードブック上での平均二乗検索誤差と最近傍想起率の両方を返す。

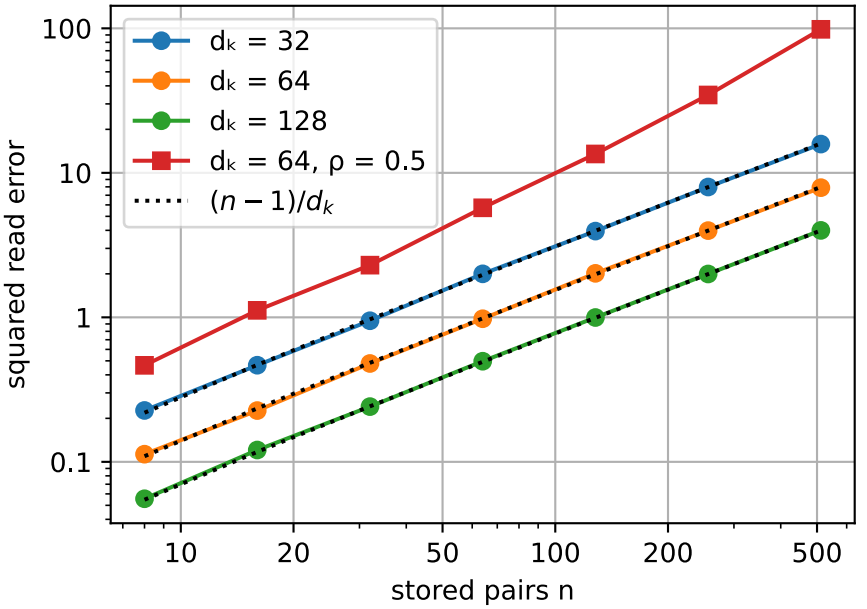
```
def matrix_memory(d_k, nums, rho=0, num_values=256, trials=20, seed=0):
    """Read error and recall of S = sum_i v_i k_i^T over random unit keys."""
    rng = np.random.default_rng(seed)
```

```
    unit = lambda X: X / np.linalg.norm(X, axis=-1, keepdims=True)
    codebook = unit(rng.standard_normal((num_values, d_k)))
    errors, recalls = [], []
    for n in nums:
        error = recall = 0
        for _ in range(trials):
            K = unit(rng.standard_normal((n, d_k)))  # Keys as rows
            shared = unit(rng.standard_normal(d_k))  # A common direction
            K = unit(np.sqrt(1 - rho**2) * K + rho * shared)
            ids = rng.integers(0, num_values, n)
            S = codebook[ids].T @ K  # Sum of v_i k_i^T
            reads = K @ S.T  # Row j is S k_j
            error += ((reads - codebook[ids]) ** 2).sum(1).mean() / trials
            recall += ((reads @ codebook.T).argmax(1) == ids).mean() / trials
        errors.append(error)
        recalls.append(recall)
    return errors, recalls
```

matrix_memory をペア数 $n \in \{8, 16, \ldots, 512\}$ にわたって、独立なランダムキー（$d_k \in \{32, 64, 128\}, \rho = 0$）および相関のあるキー（$d_k = 64, \rho = 0.5$）について評価し、二乗読み出し誤差を理論直線 $(n-1)/d_k$ と比較してプロットする。

```
nums = [8, 16, 32, 64, 128, 256, 512]
results = {f'd□ = {d_k}': matrix_memory(d_k, nums) for d_k in (32, 64, 128)}
results['d□ = 64, ρ = 0.5'] = matrix_memory(64, nums, rho=0.5)
predictions = [[(n - 1) / d_k for n in nums] for d_k in (32, 64, 128)]
d2l.plot(nums, [err for err, _ in results.values()] + predictions,
         'stored pairs n', 'squared read error',
         legend=list(results) + ['$(n-1)/d_k$'], xscale='log', yscale='log',
         fmts=('-o', '-o', '-o', '-s', 'k:', 'k:', 'k:'), figsize=(5, 3.5))
print('recall at n =    ', ' '.join(f'{n:4d}' for n in nums))
for name, (_, recalls) in results.items():
    print(f'{name:17s}', ' '.join(f'{r:4.2f}' for r in recalls))
```

```
recall at n =        8   16   32   64  128  256  512
d□ = 32           1.00 0.99 0.96 0.74 0.39 0.16 0.07
d□ = 64           1.00 1.00 1.00 1.00 0.96 0.72 0.37
d□ = 128          1.00 1.00 1.00 1.00 1.00 1.00 0.94
d□ = 64, ρ = 0.5  1.00 0.99 0.83 0.49 0.21 0.07 0.03
```



プロットと表から、2 つの性質が明確に読み取れる。

1. **想起率は状態の総要素数 $d_k d_v$ に応じてスケールする：**二乗誤差の曲線は $(n-1)/d_k$ にぴったり重なる。ところが $n = d_k = d_v = 64$ では、総二乗誤差が ≈ 1（単位信号 $\|\mathbf{v}_j\|^2 = 1$ と同じ大きさ）に達しているにもかかわらず、最近傍想起率は依然として 1.00 である（96% を維持するのは $n = 128$ のとき）。なぜか。干渉ベクトルは値の全 d_v 次元にわたって等方的に広がるため、任意の 1 つのコードブック候補ベクトルへの射影の分散が $\approx n/(d_k d_v)$ に抑えられるからである。$d_v = d_k$ のとき、d_k を 2 倍にすると状態の要素数 $d_k d_v$ は 4 倍になり、一定の正解率で想起できるペア数も 4 倍になる。実際、正解率 96% を達成するのは $n = 32$（$d_k = 32$）および $n = 128$（$d_k = 64$）であり、94% を達成するのは $n = 512$（$d_k = 128$）である（演習問題 5 では d_k と d_v を独立に変化させる）。

2. **キーの相関は削減不能な誤差の下限を生む：**キーが重み ρ で共通の方向を持つとき、ペアごとのキーの重なりは平均して $\approx \rho^2$ となり、期待二乗誤差は次のようになる。

$$\mathbb{E}\,\|\mathbf{S}\mathbf{k}_j - \mathbf{v}_j\|^2 \approx (n-1)\Big(\rho^4 + \frac{(1-\rho^2)^2}{d_k}\Big)$$

これは値の平均がゼロの場合の式である（有限のコードブックの標本平均に由来する n の二次項がさらに加わる。演習問題 4 を参照）。ρ^4 は d_k を大きくしても減少しないため、相関のあるキー（$\rho = 0.5$）では $n = d_k = 64$ における想起率が 1.00 から 0.49 へと低下する。

12.6.2 減衰

12.6.2.1 古いペアの忘却

減衰がなければ、これまでに書き込まれたすべてのペアが最大の重みのまま $\mathbf{S}_t$ に残り続けるため、干渉は系列長（sequence length）とともに際限なく増大する。各書き込みの前に状態へスカラー減衰 $a_t \in (0,1)$ を掛けることで、古いペアを徐々に薄れさせる。

$$\mathbf{S}_t = a_t\,\mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \qquad a_t \in (0,1). \tag{12.29}$$

定数減衰 a のもとでは、ℓ ステップ前に書き込まれたペアは重み a^ℓ を持つ。重みの総和は $1/(1-a)$（実効ウィンドウとして $1/(1-a)$ ペア分）となる一方、干渉の分散に寄与する二乗重みの総和は $1/(1-a^2)$ となり、全体の系列長にかかわらず定常状態の期待干渉は $1/((1-a^2)d_k)$ で上から抑えられる（演習問題 6）。入力トークンから a_t を計算するようにすれば、各ヘッドはある区間にわたって状態を保持し、話題や節の境界でそれを消去できる。（関連する変種として、ヘッドごとに固定減衰を用いる RetNet (Sun et al. 2023)、ヘッドごとに入力依存のスカラー減衰を用いる Mamba-2 (Dao and Gu 2024)、キーの座標ごとにベクトル減衰 $\mathbf{S}_{t-1}\mathrm{diag}(\alpha_t)$ を用いるゲート付き線形アテンション（Gated Linear Attention、GLA）(S. Yang, Wang, Shen, et al. 2024) がある。本節ではヘッドごとに 1 つの入力依存スカラー減衰を用いる。）

12.6.2.2 行列メモリとしての選択的状態空間モデル

Mamba の選択的 SSM における H 個のチャネル状態 $\mathbf{h}_t^{(c)} \in \mathbb{R}^N$（12.5 節、スキップ接続 $D_c x_{t,c}$ は省略）を行列 $\mathbf{S}_t \in \mathbb{R}^{H\times N}$ の行として積み重ねると、Mamba がすでに減衰付きの行列状態漸化式になっていることがわかる。

$$\begin{aligned}\mathbf{S}_t &= \mathbf{E}_t \odot \mathbf{S}_{t-1} + (\Delta_t \odot \mathbf{x}_t)\,\mathbf{B}_t^\top,\\ \mathbf{y}_t &= \mathbf{S}_t\mathbf{C}_t, \qquad \mathbf{E}_t = \exp(\mathrm{diag}(\Delta_t)\,\Lambda).\end{aligned}$$

Equation 12.29 と各項を対応させると、値ベクトルは $\mathbf{v}_t = \Delta_t \odot \mathbf{x}_t \in \mathbb{R}^H$、キーベクトルは $\mathbf{k}_t = \mathbf{B}_t \in \mathbb{R}^N$、クエリベクトルは $\mathbf{q}_t = \mathbf{C}_t \in \mathbb{R}^N$ である。唯一の違いは、Mamba の減衰行列 $\mathbf{E}_t \in \mathbb{R}^{H\times N}$ がチャネルと状態の両方にわたって変化する点である。$\mathbf{E}_t$ をヘッドあたり単一のスカラー a_t に制限すると Mamba-2 (Dao and Gu 2024) が得られ（演習問題 1）、以下に示す行列積による双対形式が導かれる。

12.6.3 双対形式

12.6.3.1 減衰を伴う展開

Equation 12.29 を $\mathbf{S}_0 = \mathbf{0}$ から展開すると、ステップ s で書き込まれたペアはステップ $t \ge s$ に到達するまでに累積減衰積 $L_{ts} = a_{s+1}a_{s+2}\cdots a_t$ によって減衰することがわかる。

$$\mathbf{y}_t = \sum_{s\le t} L_{ts}\,(\mathbf{k}_s^\top\mathbf{q}_t)\,\mathbf{v}_s, \qquad L_{ts} = a_{s+1}a_{s+2}\cdots a_t. \tag{12.30}$$

$\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t, \mathbf{y}_t$ をそれぞれ $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{Y}$ の行として積み重ね、$L_{ts} = 0$（$s > t$ のとき）とおくと、次の**双対形式**（dual form）が得られる。

$$\mathbf{Y} = (\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top)\,\mathbf{V}. \tag{12.31}$$

Equation 12.31 は**学習された入力依存の減衰マスク $\mathbf{L}$** を伴う因果線形アテンションであり、この対応関係を Dao and Gu (2024) は**状態空間双対性**（SSD）と呼んでいる。Equation 12.31 をステップごとの漸化式として評価すると計算量は $O(Td_vd_k)$、メモリは $O(d_vd_k)$ で済む。一方、完全な $T\times T$ 行列 $\mathbf{L}\odot\mathbf{Q}\mathbf{K}^\top$ を介して評価すると、並列な行列積として実行できる代わりに計算量は $O(T^2(d_k+d_v))$、メモリは $O(T^2)$ を要する。両方の形式を明示的に書いてみよう。まずは逐次的なステップごとの漸化式 `linear_attention_recurrent` から始める。

```
def linear_attention_recurrent(Q, K, V, a):
    """y_t = S_t q_t with S_t = a_t S_{t-1} + v_t k_t^T, step by step."""
    S, outputs = Q.new_zeros(V.shape[-1], K.shape[-1]), []
    for q, k, v, a_t in zip(Q, K, V, a):
        S = a_t * S + torch.outer(v, k)
        outputs.append(S @ q)
    return torch.stack(outputs)
```

次に、`decay_mask(log_a)` は累積和を用いて下三角減衰行列 $L_{ts} = \exp(\sum_{r=s+1}^{t} \log a_r)$ を構築し、これによって `linear_attention_dual` がすべての位置にわたって一度に $\mathbf{Y} = (\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ を評価できるようにする。

```
def decay_mask(log_a):
    """L[t, s] = a_{s+1} * ... * a_t for s <= t, and 0 for s > t."""
    cum = log_a.cumsum(-1)
    log_L = cum[..., :, None] - cum[..., None, :]  # Sum of log a over (s, t]
    causal = torch.ones(log_L.shape[-2:], dtype=torch.bool,
                        device=log_a.device).tril()
    return torch.exp(log_L.masked_fill(~causal, -torch.inf))

def linear_attention_dual(Q, K, V, a):
    """The same outputs as Y = (L * Q K^T) V, all positions at once."""
    return (decay_mask(torch.log(a)) * (Q @ K.T)) @ V
```

12.6.3.2 チャンク単位の計算

どちらの純粋な形式も GPU での訓練には最適ではない。トークン単位の漸化式は小さな逐次ステップを T 回実行する必要があり、完全な双対形式はヘッドごとに $T \times T$ 行列を実体化してしまうからである。**チャンク化**（chunking）は、系列を T/m 個のチャンク（各長さ m）に分割することで両者の長所を兼ね備える（Figure 12.13）。チャンクをまたぐ任意の減衰 L_{ts} は、（キーが属するチャンクの終端までの減衰）×（中間の各チャンクの減衰）×（クエリが属するチャンクの始端からの減衰）へと因数分解されるため、$\mathbf{L}$ のすべての非対角ブロックのランクは 1 になる。また、$\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top$ のすべての非対角ブロックのランクは高々 d_k となり、チャンク境界を越えて引き継がれる $d_v \times d_k$ の状態行列によって完全に要約される（演習問題 2）。

各チャンク内の位置に 1 から m までのインデックスを振ると、`chunked_linear_attention` はバッチ化された次の 3 つのステップで進む。

1. **チャンク内（対角ブロック）：** 双対形式 Equation 12.31 を、長さ m の各チャンクの内部でそれぞれの $m \times m$ 減衰マスクを用いて並列に評価する。
2. **チャンク間（チャンク境界の状態）：** $\gamma^{(c)} = a_1 \cdots a_m$ をチャンク c 全体の減衰とすると、チャンク c の終端における状態は次のチャンク単位の線形漸化式（linear recurrence）を満たす。

$$\mathbf{S}^{(c)} = \gamma^{(c)}\, \mathbf{S}^{(c-1)} + \sum_{t=1}^{m} L_{mt}\, \mathbf{v}_t \mathbf{k}_t^\top, \tag{12.32}$$

この漸化式は、`d2l.linear_scan` によって全 T/m チャンクにわたり $\lceil \log_2(T/m) \rceil$ ラウンドで評価される。

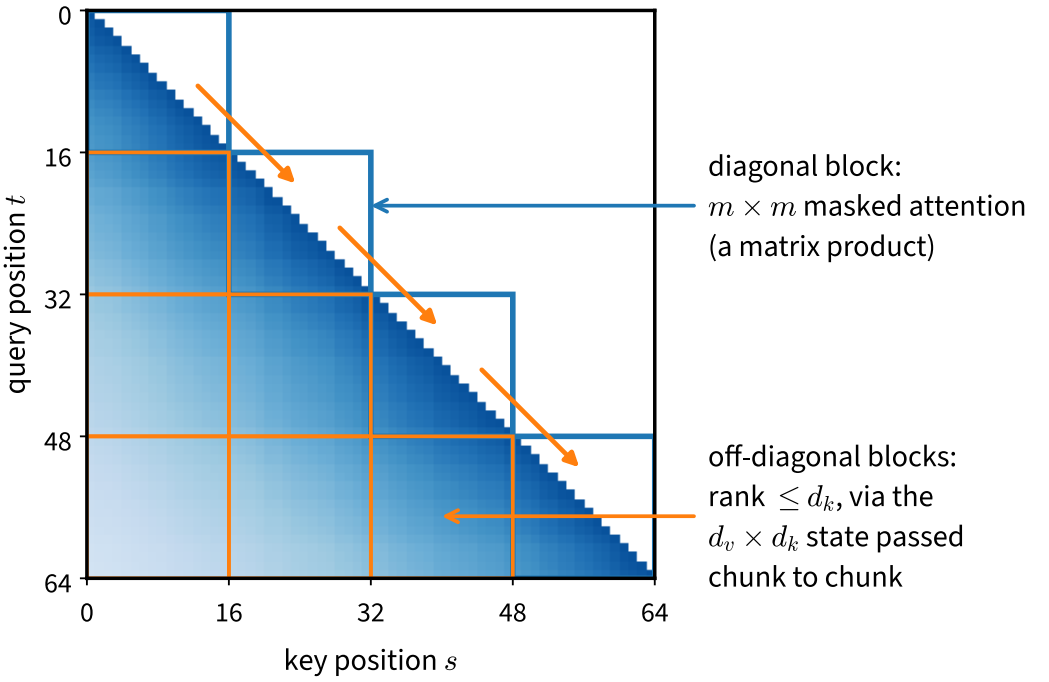


Figure 12.13: ランダムな減衰に対するマスク $\mathbf{L}$（Equation 12.31）をチャンクに分割した様子。対角ブロック（青枠）は並列に計算される小さなマスク付きアテンションの積である。対角より下の各ブロック（オレンジ枠）は $\mathbf{L}$ においてランク 1 であり、また高々ランク d_k を $\mathbf{L} \odot \mathbf{QK}^\top$ において持ち、チャンクからチャンクへ渡される状態（矢印）によって処理される。減衰が重なるにつれて陰影は薄くなっていく。

3. **過去状態の読み出し（非対角ブロック）：** $(a_1 \cdots a_t)\,\mathbf{S}^{(c-1)}\mathbf{q}_t$ を各位置 t（チャンク c 内）に加算する。

これで FLOPs の大部分は密な行列積となり、ピーク時の作業メモリは $O(T^2)$ から $O(Tm + (T/m)d_v d_k)$ へと減少する。

```
def chunked_linear_attention(Q, K, V, log_a, chunk_size):
    """y_t = S_t q_t with S_t = a_t S_{t-1} + v_t k_t^T, computed in chunks.

    Q, K: (batch_size, T, d_k); V: (batch_size, T, d_v); log_a: (batch_size, T)
    """
    B, T, m = Q.shape[0], Q.shape[1], chunk_size
    Q, K, V = (X.reshape(B, T // m, m, -1) for X in (Q, K, V))
    log_a = log_a.reshape(B, T // m, m)
    # 1. Within chunks: the dual form, one m x m mask per chunk
    Y = (decay_mask(log_a) * (Q @ K.mT)) @ V
    # 2. Across chunks: S^(c) = gamma^(c) S^(c-1) + sum_t L_mt v_t k_t^T
    cum = log_a.cumsum(-1).unsqueeze(-1)  # Log decay from the chunk's start
    writes = V.mT @ (K * torch.exp(cum[:, :, -1:] - cum))
    S = d2l.linear_scan(torch.exp(cum[:, :, -1:]), writes)  # At chunk ends
    S = torch.cat([torch.zeros_like(S[:, :1]), S[:, :-1]], 1)  # At the starts
    # 3. Every step reads the state at its chunk's start, decayed to it
    return (Y + (Q * torch.exp(cum)) @ S.mT).reshape(B, T, -1)
```

逐次的な漸化式、完全な双対形式、およびチャンクサイズ $m \in \{16, 64, 256\}$ にわたるチャンク化アルゴリズムが、長さ $T = 512$ の系列上で float32 の精度で一致することを確かめてみよう。

```
gen = torch.Generator().manual_seed(0)  # Local: later training stays unseeded
T, d_k, d_v = 512, 64, 64
Q, K, V = (torch.randn(T, d, generator=gen) for d in (d_k, d_k, d_v))
a = 0.8 + 0.19 * torch.rand(T, generator=gen)  # Decays in (0.8, 0.99)
Y = linear_attention_recurrent(Q, K, V, a)
forms = {'dual form': linear_attention_dual(Q, K, V, a)}
for m in (16, 64, 256):
    forms[f'chunks of {m}'] = chunked_linear_attention(
        Q[None], K[None], V[None], torch.log(a)[None], m)[0]
for name, Y_form in forms.items():
    print(f'{name:13s} relative deviation from the recurrence: '
          f'{(Y_form - Y).abs().max() / Y.abs().max():.1e}')
```

```
dual form     relative deviation from the recurrence: 1.5e-06
chunks of 16  relative deviation from the recurrence: 2.2e-07
chunks of 64  relative deviation from the recurrence: 2.5e-07
chunks of 256 relative deviation from the recurrence: 8.8e-07
```

すべての形式が相対誤差 $\sim 10^{-6}$ の範囲で一致する（最適なチャンクサイズ m については演習問題 3 で解析する）。

12.6.4 ゲート付き線形アテンション層

12.6.4.1 層の構成

GatedLinearAttention は、マルチヘッドのチャンク化線形アテンションを系列ミキサとしてまとめた層である。各ヘッド h に対して入力依存のスカラー減衰 $a_{t,h} = \sigma(\mathbf{w}_h^\top \mathbf{x}_t + b_h)$ を計算し、そのバイアスは init_decays によって初期の時間スケール τ_h（$a_{t,h} \approx e^{-1/\tau_h}$）がヘッド間で $[4, 256]$ ステップの範囲に対数一様に分布するよう初期化する。

```
class GatedLinearAttention(nn.Module):
    """Multi-head linear attention with one input-dependent decay per head."""
    def __init__(self, num_hiddens, num_heads, chunk_size=16, conv_width=4,
                 timescales=(4, 256)):
        super().__init__()
        d = num_hiddens
        self.num_heads, self.chunk_size = num_heads, chunk_size
        self.W_qkv = nn.Linear(d, 3 * d, bias=False)
        self.conv = nn.Conv1d(3 * d, 3 * d, conv_width, groups=3 * d,
                              padding=conv_width - 1)  # One filter per channel
        self.W_a = nn.Linear(d, num_heads)  # Decay gate, one per head
        self.W_g = nn.Linear(d, d)  # Output gate
        self.W_o = nn.Linear(d, d, bias=False)
        d2l.init_lecun(self)
        self.init_decays(*timescales)  # Memories of 4 to 256 steps by default

    def init_decays(self, tau_min, tau_max):
        """Start the decays at exp(-1/tau), tau log-uniform in range."""
        b = self.W_a.bias
        tau = torch.logspace(math.log10(tau_min), math.log10(tau_max), len(b))
        with torch.no_grad():  # log sigmoid(b) = -1 / tau
            b.copy_(-torch.log(torch.expm1(1 / tau)))
```

メモリの更新処理は memory メソッドに分離しておく。これにより、12.7 節ではブロックの残りの部分を重複して書くことなく GatedLinearAttention をサブクラス化し、デルタ則に差し替えることができる。heads_to_batch と batch_to_heads は、ヘッド軸をバッチ軸へ折り畳む操作と元に戻す操作を担う。

```
def heads_to_batch(Z, num_heads):
    """(batch_size, T, num_heads * e) -> (batch_size * num_heads, T, e)."""
    B, T, _ = Z.shape
    return Z.reshape(B, T, num_heads, -1).transpose(1, 2).flatten(0, 1)

def batch_to_heads(Y, batch_size):
    """(batch_size * num_heads, T, e) -> (batch_size, T, num_heads * e)."""
    _, T, e = Y.shape
    return Y.reshape(batch_size, -1, T, e).transpose(1, 2).flatten(2)

@d2l.add_to_class(GatedLinearAttention)
def decays(self, X):
    """Log decays, (batch_size * num_heads, T)."""
    return heads_to_batch(F.logsigmoid(self.W_a(X)), self.num_heads)[..., 0]
```

```
@d2l.add_to_class(GatedLinearAttention)
def memory(self, X, Q, K, V, log_a):
    """Outputs of the decayed matrix memory of every head. X is unused here;
    the delta rule of the next section reads its write strengths from it."""
    return chunked_linear_attention(Q, K, V, log_a, self.chunk_size)
```

順伝播（Figure 12.14）は次の手順で進む。

1. 幅 4 のデプスワイズ因果畳み込みと SiLU によって隣接トークンを混合して $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ を作る。これにより、位置 t の値トークンが単一の層の内部で位置 $t-1$ のキートークンに直接束縛できるようになる（12.8 節ではこの畳み込みをアブレーションする）。
2. クエリとキーを単位長へ ℓ_2 正規化する（Equation 12.28 の設定と一致させる）。
3. 各ヘッドの読み出しに RMS 正規化を施し、$\mathrm{SiLU}(\mathbf{W}_g \mathbf{x}_t)$ でゲートを掛けたうえで元の幅 d へ出力射影（output projection）する (S. Yang, Wang, Shen, et al. 2024)。

```
@d2l.add_to_class(GatedLinearAttention)
def forward(self, X):  # X: (batch_size, num_steps, num_hiddens)
    B, T = X.shape[:2]
    QKV = self.conv(self.W_qkv(X).mT)[..., :T].mT  # Causal: drop the tail
    Q, K, V = (heads_to_batch(Z, self.num_heads)
               for Z in F.silu(QKV).chunk(3, -1))
    Q, K = F.normalize(Q, dim=-1), F.normalize(K, dim=-1)  # Unit length
    Y = self.memory(X, Q, K, V, self.decays(X))  # (B * num_heads, T, d / h)
    Y = F.rms_norm(Y, Y.shape[-1:], eps=1e-6)  # Unit RMS per head
    return self.W_o(batch_to_heads(Y, B) * F.silu(self.W_g(X)))
```

12.6.4.2 連想想起

訓練されたニューラルネットワークは、ランダムキーに対して導出した $O(d_k d_v)$ の容量則に従って行列状態を活用するのだろうか。合成ベンチマークである**マルチクエリ連想想起**（multi-query associative recall）(Arora, Eyuboglu, Timalsina, et al. 2024) を用いてこれを検証する。各系列では、まず n 個のキー・値ペア（互いに異なるキートークンの直後に、4 クラスから選ばれた値トークンが続く組）を提示し、任意個のフィラートークンを挟んだあと、最後に全 n 個のキーをランダムな順序でクエリとして再度提示する。（オプションの `num_rebound` と `gap` は一部のキーを新しい値で上書きするためのもので、12.7 節で使用する。）

```
def associative_recall(rng, batch_size, num_pairs, num_keys=512,
                       num_values=4, num_steps=256, num_rebound=0, gap=None):
    """Key-value pairs, filler, then every key again as a query. The first
    num_rebound keys are written twice, the second time with a new value,
    at any later write or, given gap, gap writes after the first one.
    Returns the tokens (values, then keys, then the filler), the latest
    value of each query's key, and a mask of the queries for rebound keys."""
    n, r = num_pairs, num_rebound
    keys = np.argsort(rng.random((batch_size, num_keys)), 1)[:, :n]
    values = rng.integers(0, num_values, (batch_size, n + r))
    values[:, n:] = (values[:, :r] + rng.integers(  # A new value
        1, num_values, (batch_size, r))) % num_values
    if gap is None:  # Random write order, each second write after the first
        slots = np.argsort(rng.random((batch_size, n + r)), 1)
```

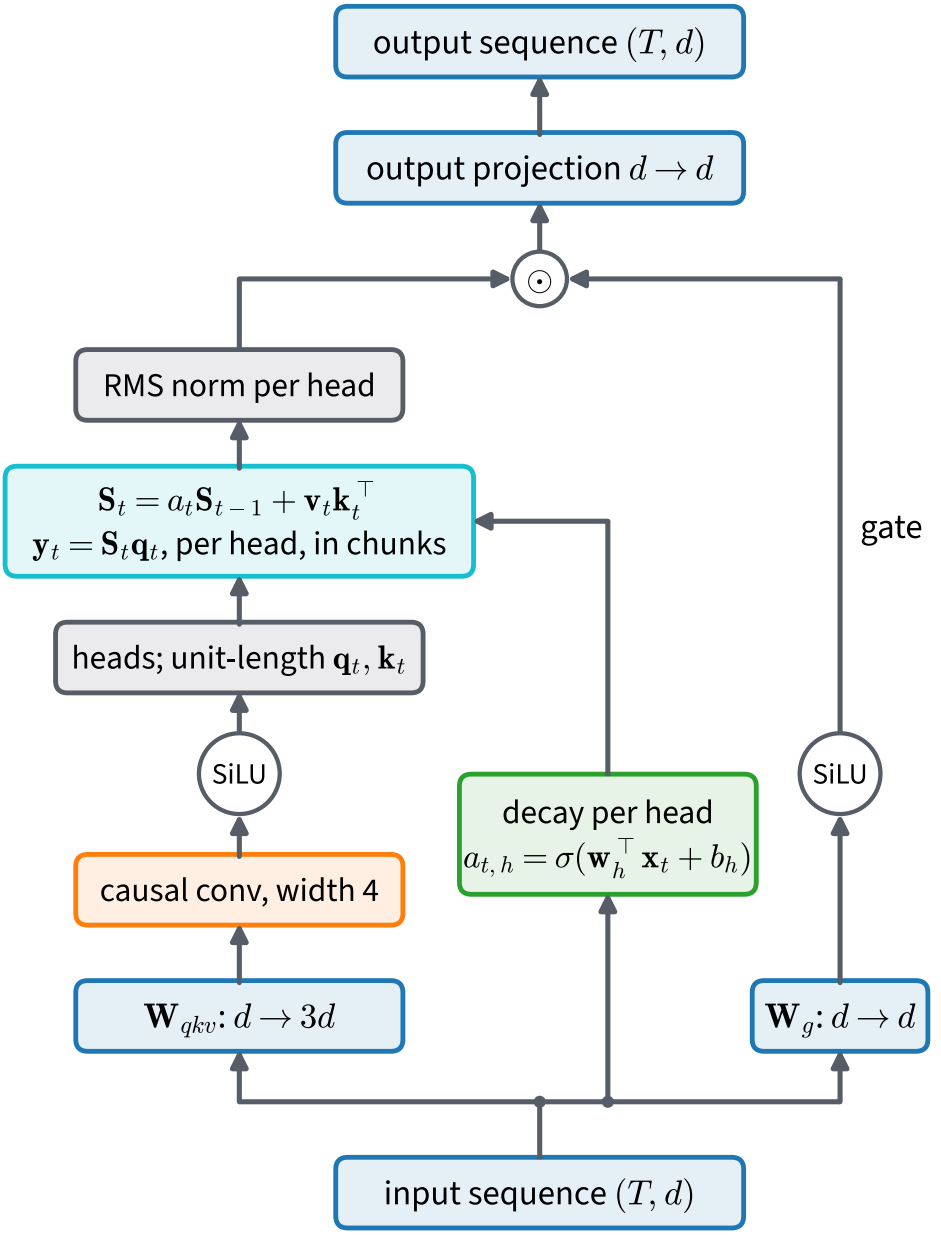


Figure 12.14: ゲート付き線形アテンション層。1 つ目の射影と短い因果畳み込みから全ヘッドのクエリ・キー・値が得られ、2 つ目の射影から各ヘッドの減衰 a_t が得られる。行列メモリはチャンク単位で計算される。その読み出しはヘッドごとに RMS 正規化され、入力から計算されたゲートを掛け合わせたのち、幅 d へ射影される。

```
        slots[:, :r], slots[:, n:] = (np.minimum(slots[:, :r], slots[:, n:]),
                                      np.maximum(slots[:, :r], slots[:, n:]))
    else:  # First writes in random order, second ones gap first writes later
        first = np.argsort(rng.random((batch_size, n)), 1)
        times = np.concatenate([first, first[:, :r] + gap - 0.5], 1)
        slots = np.argsort(np.argsort(times, 1), 1)
    X = np.full((batch_size, num_steps), num_values + num_keys)  # Filler
    rows = np.arange(batch_size)[:, None]
    X[rows, 2 * slots] = num_values + np.concatenate([keys, keys[:, :r]], 1)
    X[rows, 2 * slots + 1] = values
    order = np.argsort(rng.random((batch_size, n)), 1)  # Queries
    X[:, -n:] = num_values + np.take_along_axis(keys, order, 1)
    latest = np.concatenate([values[:, n:], values[:, r:n]], 1)
    return X, np.take_along_axis(latest, order, 1), order < r
```

4 個のペアと 1 個の再束縛キーを含む短い系列を確認してみよう（トークン `0..3` が値、`4..515` がキー、`516` がフィラーである）。

```
X, Y, rebound = associative_recall(np.random.default_rng(0), 1, 4,
                                   num_steps=16, num_rebound=1)
print('tokens: ', X[0])
print('answers:', Y[0], ' written twice:', rebound[0])
```

```
tokens:  [273   3  15   3 200   2 407   0 273   0 516 516 200 407 273  15]
answers: [2 0 0 3]  written twice: [False False  True False]
```

訓練ハーネス `train_recall` を保存しておく。この関数は、2,000 ステップの AdamW（コサインウォームアップ付き、ピーク学習率 0.01）でペア数 $n \in \{4, 8, 16, 32, 64\}$ を巡回しながらモデルを訓練し、各負荷における想起率を 1,024 本の新規系列で評価する。

```
def train_recall(model, pairs=(4, 8, 16, 32, 64), num_updates=2000,
                 **task):
    """Train on associative recall (task: options of associative_recall)
    with a cycling number of pairs, 256 sequences per update, AdamW at peak
    rate 0.01; return the accuracy per number of pairs."""
    rng = np.random.default_rng()
    length = lambda n: 32 * math.ceil(
        (3 * n + 2 * task.get('num_rebound', 0)) / 32)
    data = lambda n, size: associative_recall(
        rng, size, n, num_steps=length(n), **task)[:2]
    model.to(d2l.try_gpu())
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-2,
                                  weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lambda t: (
        d2l.cosine_warmup(t, num_updates, num_updates // 10)))
    d2l.train_queries(model, lambda step: data(pairs[step % len(pairs)],
                      256), num_updates, optimizer, scheduler)
    return {n: float(d2l.query_accuracy(model, *data(n, 1024)))
            for n in pairs}
```

モデル幅（model width）やパラメータ数（parameter count）を変え**ずに**行列状態の容量を変化させるため、2 ブロックの `SeqLM` を幅 $d = 32$ で `memory_model(num_heads)` 内に固定し、その幅

を $h \in \{8, 4, 2\}$ 個のヘッド（各次元 $e = d/h \in \{4, 8, 16\}$）に分割する。

```
def memory_model(num_heads, num_hiddens=32, vocab_size=4 + 128 + 1):
    """Two blocks of gated linear attention; tokens: values, keys, filler."""
    return d2l.SeqLM(vocab_size, num_hiddens, [GatedLinearAttention(
        num_hiddens, num_heads, timescales=(256, 2048)) for _ in range(2)])
```

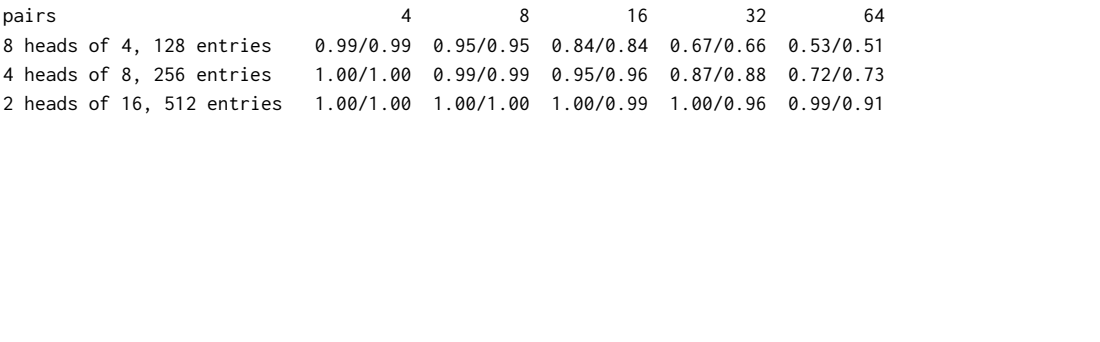
各ヘッドは $e \times e$ の状態行列を保持するため、層あたりの状態の総要素数は $h \cdot e^2 = d \cdot e \in \{128, 256, 512\}$ とスケールする一方、総パラメータ数はほぼ同一（約 37,000）に保たれる。各ヘッド構成について `train_recall` で 2 回ずつ訓練を実行する。

```
recall = {}
for num_heads in (8, 4, 2):
    d_h = 32 // num_heads
    name = f'{num_heads} heads of {d_h}, {num_heads * d_h ** 2} entries'
    models = [memory_model(num_heads) for _ in range(2)]
    recall[name] = [train_recall(model, num_keys=128) for model in models]
    print(f'{name:26s} {d2l.num_params(models[0]):,} parameters')
```

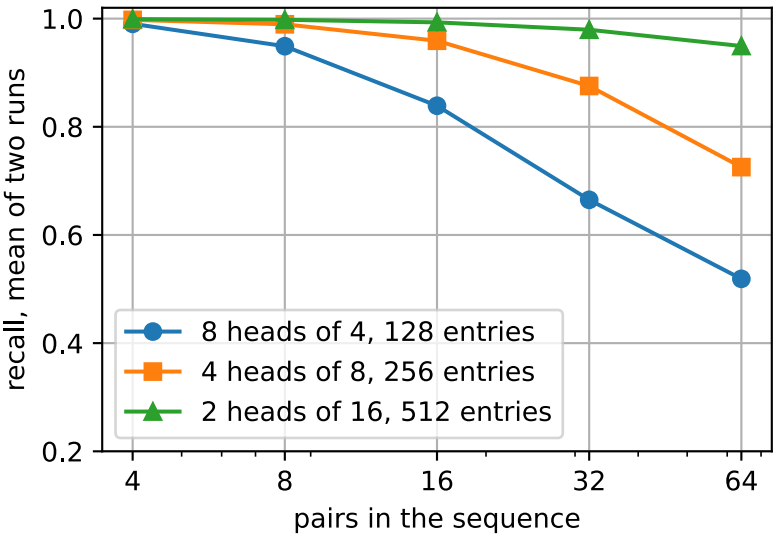
```
8 heads of 4, 128 entries  37,461 parameters
4 heads of 8, 256 entries  37,197 parameters
2 heads of 16, 512 entries 37,065 parameters
```

平均想起正解率をペア数 $n \in \{4, 8, 16, 32, 64\}$ に対してプロットおよび集計すると、検索容量が状態の総要素数に応じてどのようにスケールするかがわかる。

```
pairs = list(recall[name][0])
d2l.plot(pairs, [[np.mean([run[n] for run in runs]) for n in pairs]
                 for runs in recall.values()], 'pairs in the sequence',
         'recall, mean of two runs', legend=list(recall), xscale='log',
         ylim=(0.2, 1.02), fmts=('-o', '-s', '-^'), figsize=(4.5, 3))
d2l.plt.xticks(pairs, pairs)  # Ticks at the numbers of pairs
print(f'{"pairs":26s}', ''.join(f'{n:>11d}' for n in pairs))
for name, runs in recall.items():
    print(f'{name:26s}', ''.join(f'{runs[0][n]:6.2f}/{runs[1][n]:.2f}'
                                 for n in pairs))
```



```
pairs                              4          8         16         32         64
8 heads of 4, 128 entries    0.99/0.99  0.95/0.95  0.84/0.84  0.67/0.66  0.53/0.51
4 heads of 8, 256 entries    1.00/1.00  0.99/0.99  0.95/0.96  0.87/0.88  0.72/0.73
2 heads of 16, 512 entries   1.00/1.00  1.00/1.00  1.00/0.99  1.00/0.96  0.99/0.91
```



3 つの構成はいずれも 4 ペアの課題を解き（$\geq 98\%$）、ペア数が増えるにつれて想起率は状態の要素数に直結してスケールする。状態容量を 128 要素（8×4^2）から 256 要素（4×8^2）に倍増させると、同じ正解率で想起できるペア数もおよそ 2 倍になり（~ 0.50 から ~ 0.75 へ、64 ペアのとき）、512 要素（2×16^2）は 128 要素を ≥ 16 のすべての負荷で上回る。

ただし、すべての加法的な行列メモリには構造上の欠陥が 1 つ残っている。$\mathbf{v}_t \mathbf{k}_t^\top$ の書き込みは、$\mathbf{k}_t$ のもとにすでに保存されている値へ単純に**加算する**にすぎない点である。同じキーが 2 回目に新しい値へ束縛されると、そのキーをクエリした際には古い値と新しい値の重ね合わせが返ってしまう。キーの値を**上書きする**方法をメモリに獲得させることが、12.7 節の主題である。

12.6.5 まとめ

行列値の状態 $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$ は、外積の書き込み $\mathbf{v}_t \mathbf{k}_t^\top$ によってキー・値ペアを保存し、線形クエリ $\mathbf{S}_t \mathbf{q}_t$ によってそれらを検索する。この漸化式を展開すると因果線形アテンションになり、Mamba の選択的 SSM はキーを $\mathbf{B}_t$、クエリを $\mathbf{C}_t$ としたその一例である。ランダムな単位キーのもとでは、ペア間のクロストークによって期待二乗検索誤差 $(n-1)/d_k$（n 回の書き込み後）が生じる一方、最近傍想起の容量は状態の総要素数 $d_k d_v$ に応じてスケールする。ヘッドごとに入力依存のスカラー減衰 $a_t \in (0,1)$ を導入すると古いペアが薄れ、因果マスク付き線形アテンション $\mathbf{Y} = (\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ との**状態空間双対性**が成り立つ。この双対形式をチャンク単位で評価すれば、チャンク内の並列な行列積とチャンク間のスキャンが組み合わさり、$O(T)$ のメモリで計算できる。パラメータ数を固定したマルチクエリ連想想起では、`GatedLinearAttention` 層の状態要素数を 2 倍にすると想起できるペア数もおよそ 2 倍になるが、純粋に加法的な書き込みだけでは依然として過去に束縛されたキーを上書きできない。

12.6.6 演習問題

1. **行列メモリとしての Mamba**。$\mathbf{S}_t = \mathbf{E}_t \odot \mathbf{S}_{t-1} + (\Delta_t \odot \mathbf{x}_t)\mathbf{B}_t^\top$（ただし $\mathbf{E}_t = \exp(\mathrm{diag}(\Delta_t)\Lambda)$）の形において、減衰の対数は行 c をステップサイズ $\Delta_{t,c}$ でスケールしたレートの行列となる。
    1. Δ_t およびレート Λ にどのような条件を課せば、$\mathbf{E}_t$ が全要素 1 の行列のスカラー倍となり、更新式が Equation 12.29 の形になるか。Mamba-2 はこれらの条件のうちどれを課し、どれを入力依存のまま残しているか。

   2. 本節で扱った各減衰形式（RetNet、Mamba-2、ゲート付き線形アテンション、Mamba）について、$d_v \times d_k$ 要素からなる 1 つのヘッドの減衰を決定するトークンあたりの入力依存の数値の個数を数えよ。
2. **ランク 1 ブロック**。$p_t = a_1 a_2 \cdots a_t$ を Equation 12.31 における減衰の積とする。
   1. L_{ts} を p_t を用いて表し、行インデックスがすべて列インデックス以上であるような $\mathbf{L}$ の任意の部分行列がランク 1 を持つことを示せ。
   2. あるチャンクをそれより前のチャンクと結合する $\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top$ のすべてのブロックが、高々 d_k のランクを持つことを導け。また、`chunked_linear_attention` がそのようなブロックのクエリ側の減衰とキー側の減衰をどこで適用しているかを特定せよ。
   3. キー座標ごとの減衰 $\mathbf{S}_t = \mathbf{S}_{t-1}\mathrm{diag}(\alpha_t) + \mathbf{v}_t \mathbf{k}_t^\top$ を用いる場合、単一のマスク $\mathbf{L}$ を $\mathbf{Q}\mathbf{K}^\top$ に掛ける形にはならない。クエリ t とキー s の減衰付きスコアを書き下し、2 つのチャンクを結合するブロックのランクが依然として高々 d_k であることを示せ。
3. **演算数のカウント**。1 回の積和演算を 2 回の浮動小数点演算と数え、$d_k = d_v = d$ において主要項のみを残せ。
   1. 系列長 T、チャンクサイズ m における漸化式、双対形式、チャンク形式の演算数を数えよ。チャンク形式の演算数はアルゴリズムの 3 つのパートに分けて求めること。
   2. $m \le d$ の範囲ではチャンク形式のコストが高々漸化式の小さな定数倍に収まることを示し、$T = 4096$、$d = 64$、$m = 256$ におけるチャンク形式と双対形式の演算数の比を計算せよ。
   3. あるスケジュールの実行時間を、その演算数をスループット P で割った値に、逐次ステップあたり一定の所要時間 τ（漸化式では T ステップ、チャンク形式では T/m ステップ）を加えたものとしてモデル化する。チャンク形式の実行時間を最小化するチャンクサイズを求め、$T = 4096$、$d = 64$、毎秒 $P = 10^{13}$ 演算、$\tau = 5\,\mu$s の場合について評価せよ。このモデルによれば、チャンク形式は漸化式より何倍高速になると予測されるか。
4. **相関のあるキー**。●`matrix_memory` における $\rho > 0$ のキーは、$\sqrt{1-\rho^2}\,\mathbf{g}_i + \rho\,\mathbf{c}$（ただし $\mathbf{g}_i$ は独立なランダム単位ベクトル、$\mathbf{c}$ は 1 つの共通単位ベクトル）を正規化したものである。本文の式では、相異なる 2 つのキーの重なりを $\rho^2 + \epsilon_{ij}$（ただし ϵ_{ij} は平均ゼロ・分散 $(1-\rho^2)^2/d_k$ で独立）とモデル化し、値の平均がゼロであると仮定している。
   1. `matrix_memory` の値は 256 個のランダムなベクトルからなるコードブックから選ばれており、その平均 $\bar{\mathbf{v}}$ は厳密にはゼロではない。これにより期待二乗誤差に $(n-1)(n-2)\,\|\bar{\mathbf{v}}\|^2$ に比例する項が加わることを示し、その完全な予測式を実測誤差と比較せよ。
   2. コードブックを中心化（平均を引いてから各要素を再度正規化）し、曲線の上方への曲がりが消えることを確認せよ。
   3. $\rho \in \{0, 0.25, 0.5, 0.75\}$ について、$d_k = 64$ と $d_k = 256$ における誤差と想起率を測定せよ。どの ρ までならキーの幅を広げることが依然として有効であり、どこから誤差が $(n-1)\rho^4$ に近づくか。
5. [code] **幅の異なるキーと値**。●`matrix_memory` に独立した値の幅 `d_v` を持たせ、コードブックが $\mathbb{R}^{d_v}$ 内の 256 個のランダムな単位ベクトルを保持するようにせよ。
   1. $n/(d_k d_v) = 1/32$ を一定に保ち、$(d_k, d_v) = (64, 64)$、$(256, 16)$、$(16, 256)$ について想起率を測定せよ。想起率は積 $d_k d_v$ のみに依存するか。
   2. 期待二乗読み出し誤差は d_v に依存しないため、測定された結果をそれだけでは説明できないことを示せ。
   3. 各コードブック要素はおよそ $n/256$ 個のキーのもとに保存されている。それらと読み出しキーとの重なりが、その要素の方向に分散およそ $n/(256\,d_k)$ の成分を加えることを示し、これを用いてどのような形状で積の法則が破綻するかを説明せよ。
6. [code] **スライディングウィンドウとしての減衰**。●`matrix_memory` に定数減衰 a を追加し、末尾から ℓ ステップ前に書き込まれたペアが重み a^ℓ で最終的なメモリに残るようにせよ。
   1. $d_k = 64$、$n = 256$ において、$a \in \{1, 0.99, 0.95, 0.9\}$ のそれぞれに対し、最も古

い 4 分の 1 のペアと最も新しい 4 分の 1 のペアの想起率を別々に測定せよ。減衰は最も新しいペアに何をもたらし、最も古いペアにどのような代償を強いるか。

2. 命題の証明にならい、任意のキーにおける期待二乗干渉が高々 $1/((1-a^2)\,d_k)$ であることを示せ。さらに、二乗信号 $a^{2\ell}$ がこの上界を上回るペアの個数は高々およそ d_k/e 個にとどまることを、任意の $a<1$ に対して示せ。

12.7 デルタ則

12.6 節の加算型行列メモリは、新しいペアごとに $\mathbf{v}_t\mathbf{k}_t^\top$ を既存の $\mathbf{S}_{t-1}$ に足し合わせることで記憶する。この仕組みは、すべてのキーが 1 度しか束縛されない限り機能する。しかし実際のテキストやコードでは、キーの**再束縛**（rebind）が頻繁に起こる。たとえば、プログラムが `rate = 0.05` を代入した後に `rate = 0.07` へ更新したり、物語の中で登場人物がパリからローマへ移動したりする場合である。加算的書き込みのもとでは、2 度書き込まれたキーをクエリすると古い値と新しい値の和が返され、更新を繰り返すほど最新の値は古い値の下に埋もれてしまう。固定サイズの行列メモリにおいて、あるキーに格納された値を上書きするにはどうすればよいだろうか。

その答えは、**書き込み前に読み出す**ことである。すなわち、$\mathbf{k}_t$ に対する現在のメモリの予測（$\hat{\mathbf{v}}_t = \mathbf{S}_{t-1}\mathbf{k}_t$）を計算し、予測誤差 $\beta_t(\mathbf{v}_t - \hat{\mathbf{v}}_t)\mathbf{k}_t^\top$ のみを書き込む。この**デルタ則**（delta rule）は (Widrow and Hoff 1960; Schlag et al. 2021)、順伝播中に二乗想起損失 $\frac{1}{2}\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2$ に対する勾配降下法のオンラインステップを 1 回実行することに相当する。以下では、デルタ則の射影幾何学を導出し、ランダムなキーと相関のあるキーにおける再束縛の利点と信号浸食（signal erosion）のコストの両方を測定する。さらに、スカラー減衰と組み合わせた `GatedDeltaNet` を導入し (S. Yang et al. 2025)、チャンク単位の三角求解によって評価したうえで (S. Yang, Wang, Zhang, et al. 2024)、最後に系列メモリをオンライン回帰ソルバとして捉える統一的な視点で締めくくる。

12.7.1 加算的書き込み

12.7.1.1 2 度書き込まれたキー

まず、キーが再束縛されたときに加算的書き込みがどのように破綻するかを定量化しよう。単位キー $\mathbf{k}$ が R 回書き込まれ、それぞれの値が $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(R)}$ であるとき、加算型メモリはそれらの重みなし和を返す。

$$\mathbf{S}\mathbf{k} = \mathbf{v}^{(1)} + \mathbf{v}^{(2)} + \cdots + \mathbf{v}^{(R)}, \tag{12.33}$$

さらに、キー間干渉 $\sum_i (\mathbf{k}_i^\top \mathbf{k})\mathbf{v}_i$ がこれに加わる。スカラー減衰 $a_t \in (0,1)$ を導入すれば新しい値に比べて古い値を減衰させられるが、同じ経過時間の**すべて**のキーを無差別に減衰させてしまうため、1 度だけ書き込まれた事実も、再束縛されたキーの古い値と同じ速さで薄れてしまう（演習問題 2）。

キー間干渉から再束縛を切り離すため、`overwrite_recall` は $\mathbb{R}^{64}$ 内の互いに正規直交する 8 個のキーをテストする。具体的には、32 個の単位ベクトルからなるコードブックから値を取り出して各キーを $R \in \{1, 2, 4, 8\}$ 回書き込み、$\mathbf{S}\mathbf{k}$ に対するコードブック内の最近傍が最新の値 $\mathbf{v}^{(R)}$ と一致する頻度を測定する。

```
def overwrite_recall(write, num_writes, d=64, num_keys=8, num_values=32,
                     trials=200, seed=0):
    """Recall of the latest value after every key is written num_writes
```

```
    times."""
    rng = np.random.default_rng(seed)
    codebook = rng.standard_normal((num_values, d))
    codebook /= np.linalg.norm(codebook, axis=1, keepdims=True)
    recall = 0
    for _ in range(trials):
        keys = np.linalg.qr(rng.standard_normal((d, num_keys)))[0].T  # Rows
        S = np.zeros((d, d))
        for _ in range(num_writes):
            ids = rng.integers(0, num_values, num_keys)
            for k, v in zip(keys, codebook[ids]):
                S = write(S, k, v)
        reads = keys @ S.T  # Row j is S k_j
        recall += ((reads @ codebook.T).argmax(1) == ids).mean() / trials
    return recall

additive = lambda S, k, v: S + np.outer(v, k)
writes = [1, 2, 4, 8]
print('additive:', ' '.join(f'{overwrite_recall(additive, R):.2f}'
                            for R in writes))
```

```
additive: 1.00 0.51 0.27 0.21
```

$\mathbb{R}^{64}$ 内の正規直交する 8 個のキー間で干渉がゼロであっても、加算型メモリの想起率は 1.00（$R = 1$）から 0.51（$R = 2$、足し合わされた 2 つの値の間でのコイントス）、さらに 0.21（$R = 8$）へと低下する（演習問題 1）。

12.7.2 勾配ステップとしての書き込み

12.7.2.1 デルタ則

他の方向をできるだけ乱さずに $\mathbf{k}_t$ に束縛された値を置き換えるため、新しく入力されたペア $(\mathbf{k}_t, \mathbf{v}_t)$ に対するメモリの二乗予測誤差を測る。

$$\ell_t(\mathbf{S}) = \tfrac{1}{2}\,\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2, \tag{12.34}$$

そして、勾配降下法を 1 ステップ進める更新 $\mathbf{S}_t = \mathbf{S}_{t-1} - \beta_t \nabla_{\mathbf{S}} \ell_t(\mathbf{S}_{t-1})$ を、ステップサイズすなわち**書き込み強度**（write strength）を β_t として適用すると次のようになる。

$$\mathbf{S}_t = \mathbf{S}_{t-1} - \beta_t(\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top. \tag{12.35}$$

これが**デルタ則**である (Widrow and Hoff 1960; Schlag et al. 2021)。現在の予測 $\hat{\mathbf{v}}_t = \mathbf{S}_{t-1}\mathbf{k}_t$ を読み出し、ランク 1 の補正 $\beta_t(\mathbf{v}_t - \hat{\mathbf{v}}_t)\mathbf{k}_t^\top$ を加える。なお、この勾配ステップは訓練時と推論時の両方において、各トークンの**順伝播の内部で**実行される。単位キー $\|\mathbf{k}_t\| = 1$ に対して Equation 12.35 の直後に $\mathbf{k}_t$ をクエリすると、$\mathbf{S}_t\mathbf{k}_t = (1-\beta_t)\hat{\mathbf{v}}_t + \beta_t\mathbf{v}_t$ が返される。これは $\beta_t = 1$ のとき厳密な置換となり、$\beta_t \in (0,1)$ のとき凸結合となる（対照的に、加算的書き込み $\mathbf{v}_t\mathbf{k}_t^\top$ は線形アライメント $\mathbf{v}_t^\top \mathbf{S}\mathbf{k}_t$ に対する勾配上昇法のステップであり、飽和点を持たない）。

12.7.2.2 遷移

$\mathbf{S}_{t-1}$ を Equation 12.35 からくくり出すと、既存のメモリに作用する行列遷移が明らかになる。

$$\mathbf{S}_t = \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top . \tag{12.36}$$

単位キー $\mathbf{k}_t$ に対して、対称行列 $\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$ は $\mathbf{k}_t$ に直交するすべての部分空間を不変に保ち（固有値 1）、$\mathbf{k}_t$ の方向を固有値 $1 - \beta_t$ でスケーリングする（Figure 12.15）。

- $\beta_t = 0$：恒等変換（書き込みなし）。
- $\beta_t \in (0, 1)$：$\mathbf{k}_t$ 方向の部分縮小。
- $\beta_t = 1$：直交射影 $\mathbf{I} - \mathbf{k}_t \mathbf{k}_t^\top$。$\mathbf{k}_t$ 成分を完全に消去してから $\mathbf{v}_t \mathbf{k}_t^\top$ を書き込む。
- $\beta_t = 2$：ハウスホルダー鏡映（Householder reflection）$\mathbf{I} - 2\mathbf{k}_t \mathbf{k}_t^\top$（固有値 -1）。$\mathbf{k}_t$ 方向の符号を反転させる。

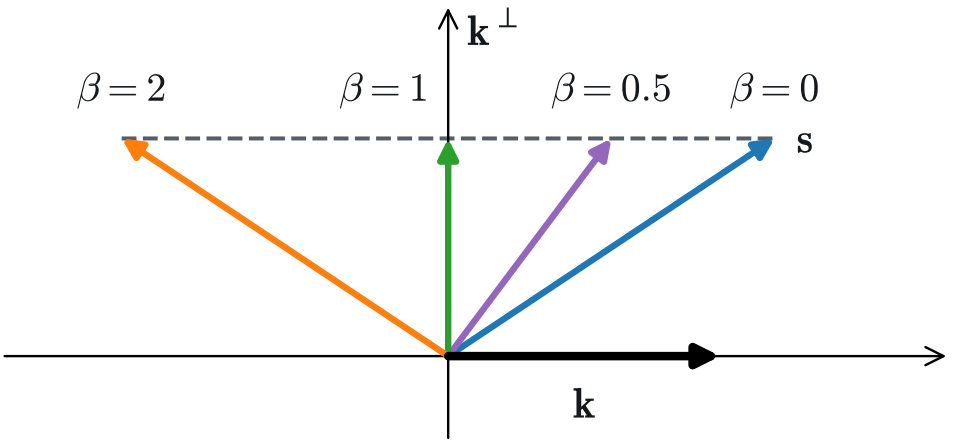


Figure 12.15: 遷移 $\mathbf{I} - \beta \mathbf{k}\mathbf{k}^\top$ が、ベクトル $\mathbf{s}$ に対して単位キー $\mathbf{k}$ とそれに直交する方向が張る平面内で作用する様子。$\mathbf{k}$ 方向の成分は $1 - \beta$ 倍され、$\beta = 0$ ではそのまま保たれ、$\beta = 0.5$ では半分になり、$\beta = 1$ では除去され、$\beta = 2$ では反転する。直交成分は変化しない。

固有値 $1 - \beta_t \|\mathbf{k}_t\|^2$ から、2 つの帰結が導かれる。

1. **単位ノルムキーが不安定性を防ぐ：**$\|\mathbf{k}_t\| = 1$ かつ $\beta_t \in [0, 2]$ のとき、遷移のすべての固有値は $[-1, 1]$ に収まるため、斉次なメモリ状態が幾何級数的に発散することはない。
2. **負の固有値が法 2 のカウント（パリティ）を可能にする：**$\beta_t \in [0, 1]$ のとき、遷移の固有値はすべて非負となり、有限精度では任意の系列長にわたってパリティを追跡できない（Grazzi et al. 2025）。β_t の範囲を 2 まで広げると、カウント対象のトークンごとに記憶状態を反転させる鏡映（固有値 -1）が可能になり、任意の長さで厳密なパリティを計算できる。12.7.3.5 節では、訓練された層がこの解を見つけられるかを検証する。

12.7.2.3 補正のコスト

正規直交するキーに対しては、$\beta_t = 1$ とすることで他のキーに一切触れずに再束縛されたキーの古い値を消去できる。では、$n > d_k$ の場合や、キーがランダムな単位ベクトルであって異なるキー $\mathbf{k}_i$ と $\mathbf{k}_t$ が非ゼロの重なり $\mathbf{k}_t^\top \mathbf{k}_i$ を持つ場合はどうなるだろうか。後続の各デルタステップは $\mathbf{k}_t$ に沿って進み、$\mathbf{k}_t$ 方向の成分を $\mathbf{v}_i \mathbf{k}_i^\top$ から射影除去するため、キー i に記憶された信号は 1 ステップあたり $1 - (\mathbf{k}_t^\top \mathbf{k}_i)^2 \approx 1 - 1/d_k$ 倍に縮小する。その結果、その後に m 回の書き込みが行われると、古いペアの信号は $(1 - 1/d_k)^m \approx e^{-m/d_k}$ しか残らない。

対象を絞った消去がいつ有効でいつ有害になるかを確認するため、delta_rule(beta) でオンラインのデルタ書き込みを実装し、correction_cost で複数の書き込みラウンドとキー間の相関にわたり幅 $d = 64$ での二乗読み出し誤差と最近傍想起率の両方を測定する。

```
def delta_rule(beta):
    """The memory after writing the pairs in order with the delta rule."""
    def fit(K, V):
        S = np.zeros((V.shape[1], K.shape[1]))
        for k, v in zip(K, V):
            S = S + beta * np.outer(v - S @ k, k)
        return S
    return fit

def correction_cost(memories, num_keys, writes=1, orthonormal=False, rho=0,
                    d=64, num_values=256, trials=20, seed=0):
    """Squared read error and recall of the latest value when each of
    num_keys keys is written `writes` times, for each memory."""
    rng = np.random.default_rng(seed)
    unit = lambda X: X / np.linalg.norm(X, axis=-1, keepdims=True)
    codebook = unit(rng.standard_normal((num_values, d)))
    error, recall = dict.fromkeys(memories, 0), dict.fromkeys(memories, 0)
    for _ in range(trials):
        if orthonormal:
            keys = np.linalg.qr(rng.standard_normal((d, num_keys)))[0].T
        else:  # Random unit keys that share one direction with weight rho
            keys = unit(np.sqrt(1 - rho**2) * unit(rng.standard_normal(
                (num_keys, d))) + rho * unit(rng.standard_normal(d)))
        ids = rng.integers(0, num_values, (writes, num_keys))  # Row = round
        K, V = np.tile(keys, (writes, 1)), codebook[ids.ravel()]
        for name, fit in memories.items():
            reads = keys @ fit(K, V).T  # Row j is S k_j
            error[name] += ((reads - codebook[ids[-1]]) ** 2).sum(1).mean()
            recall[name] += ((reads @ codebook.T).argmax(1) == ids[-1]).mean()
    return ({name: e / trials for name, e in error.items()},
            {name: r / trials for name, r in recall.items()})
```

ここでは 4 つのメモリ規則、すなわち additive、delta, β = 1、delta, β = 0.5、およびオフラインの least squares（np.linalg.lstsq により全ペアで同時に $\sum_s \|\mathbf{S}\mathbf{k}_s - \mathbf{v}_s\|^2$ を最小化する最小二乗法）を、次の 3 つの設定で比較する。(1) 各 4 回書き込まれる 8 個の正規直交キー、(2) 1 回書き込まれる $n \in \{32, 64, 128, 256\}$ 個の独立なランダム単位キー、(3) 1 回書き込まれる相関のある 64 個のキー（$\rho = 0.5$）である。

```
memories = {'additive': lambda K, V: V.T @ K,  # Sum of the v k^T
            'delta, β = 1': delta_rule(1), 'delta, β = 0.5': delta_rule(0.5),
            'least squares': lambda K, V: np.linalg.lstsq(K, V)[0].T}
cases = {'8 orthonormal keys, 4 writes': dict(num_keys=8, writes=4,
                                              orthonormal=True),
         **{f'{n} keys': dict(num_keys=n) for n in (32, 64, 128, 256)},
         '64 keys, ρ = 0.5': dict(num_keys=64, rho=0.5)}
results = {case: correction_cost(memories, **args)
           for case, args in cases.items()}
for i, measure in enumerate(['squared read error', 'recall']):
```

```
    print(f'{measure:28s}', ''.join(f'{name:>15s}' for name in memories))
    for case, result in results.items():
        print(f'{case:28s}', ''.join(f'{result[i][name]:15.2f}'
                                     for name in memories))
```

squared read error	additive	delta, β = 1	delta, β = 0.5	least squares
8 orthonormal keys, 4 writes	2.98	0.00	0.33	0.74
32 keys	0.49	0.29	0.37	0.00
64 keys	0.98	0.60	0.49	0.00
128 keys	1.99	1.06	0.67	0.50
256 keys	3.94	1.49	0.89	0.75
64 keys, ρ = 0.5	5.82	0.76	0.61	0.00
recall	additive	delta, β = 1	delta, β = 0.5	least squares
8 orthonormal keys, 4 writes	0.28	1.00	1.00	0.28
32 keys	1.00	1.00	1.00	1.00
64 keys	1.00	0.89	0.99	1.00
128 keys	0.96	0.53	0.82	1.00
256 keys	0.73	0.26	0.45	0.86
64 keys, ρ = 0.5	0.54	0.77	0.89	1.00

この表から、オンライン射影のトレードオフが浮き彫りになる。

- **再束縛（8 orthonormal keys, 4 writes）：**delta は想起率 1.00 を $\beta = 1$ と $\beta = 0.5$ の両方で達成する（二乗誤差はそれぞれ 0.00 と 0.33）のに対し、additive と重みなしの least squares は各キーに束縛された 4 つの値すべてを平均してしまい、スコアはわずか 0.28 にとどまる。
- **相関のあるキー（64 keys, ρ = 0.5）：**すべてのキーが共通の方向を持つため、additive はあらゆる値をその共有成分の上に積み重ねてしまう（二乗誤差 5.82、想起率 0.54）。一方、delta は書き込み前に共有方向の予測を差し引くため、想起率は 0.77（$\beta = 1$）および 0.89（$\beta = 0.5$）に達する。
- **過密な独立キー（128 keys、256 keys）：**ここでは、delta, β = 1 は additive よりも二乗誤差が小さい（1.06 対 1.99、$n = 128$ のとき）にもかかわらず、最近傍想起率は**低い**（0.53 対 0.96）。なぜだろうか。古い読み出し値を $\mathbf{0}$ に向かって縮小させると、二乗誤差は $\|\mathbf{0} - \mathbf{v}_j\|^2 = 1$ に向かって下がる一方で、初期のペアの信号が後続のペアに比べて浸食されるからである。より穏やかなステップ $\beta = 0.5$ を用いると、より低い二乗誤差（0.67）と大幅に高い想起率（0.82）の両方が回復する。このことから、β_t を入力依存のゲートとし、再束縛されたキーを上書きするときは $\beta_t \approx 1$ を使い、事実を蓄積するときはより小さな β_t を使えるようにする動機が得られる。

12.7.3 Gated DeltaNet

12.7.3.1 減衰と補正

デルタ則は $\mathbf{S}_{t-1}$ を現在のキー $\mathbf{k}_t$ の方向にしか変更しないため、直近のキーに直交する方向は単独では決して減衰しない。**Gated DeltaNet** は (S. Yang et al. 2025)、大域的な忘却を担う入力依存のスカラー減衰 $a_t \in (0, 1)$ と、対象キーの上書きを担うデルタ則のランク 1 方向別更新を組み合わせる。

$$\mathbf{S}_t = a_t \, \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top . \tag{12.37}$$

2 つのゲートは異なる役割を担い、両者の組み合わせによってこの層は新しいペアを 3 通りの方法で扱うことができる（Figure 12.16）。a_t が 0 に近いとき、層はそれまでに記憶した内容をすべ

て一掃して新しいペアだけを残す。これは話題の切り替わりに対する適切な応答である。$a_t \approx 1$ かつ $\beta_t = 1$ のときは、先ほどの再束縛テストと同様に、現在のキーに対応する値を置き換えつつ直交するキーには手を触れない。$a_t \approx 1$ で β_t が小さいときは、読み出し値を新しい値へ向かって途中までしか動かさないため、類似したキーに対する複数回の書き込みでは最後の 1 つが独り勝ちするのではなく平均化される。

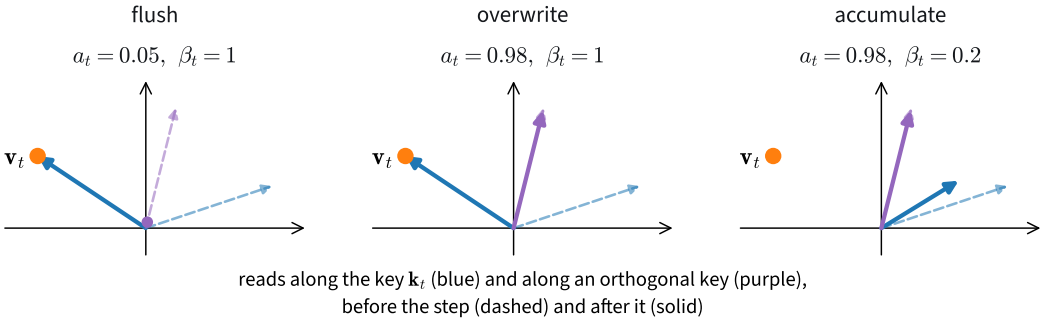


Figure 12.16: ゲートの 3 つの設定のもとでの、2×2 メモリに対する Gated DeltaNet の 1 ステップ。矢印は現在のキー $\mathbf{k}_t$ 方向（青）と直交するキー方向（紫）の読み出し値を表し、ステップ前が破線、ステップ後が実線である。減衰 a_t が 0 に近いとメモリが一掃され、新しいペアだけが残る。$a_t \approx 1$ のとき、書き込み強度 $\beta_t = 1$ では $\mathbf{k}_t$ に対応する値が $\mathbf{v}_t$ に置き換えられて他方のキーはそのまま保たれる一方、小さな β_t では読み出し値が $\mathbf{v}_t$ に向かって途中までしか移動しない。

12.7.3.2 チャンク単位の計算

遷移 $a_t(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top)$ はスカラーではなく $d_k \times d_k$ 行列であるため、素朴なトークン単位のスキャンでは $d_k \times d_k$ 行列の乗算が必要となり、$O(T d_k^3)$ のコストがかかる。S. Yang, Wang, Zhang, et al. (2024) は、**WY 表現**（WY representation）を用いて Equation 12.37 をサイズ m のチャンク単位で評価する方法を示した。まず Equation 12.37 を、減衰付きの加算的更新 $\mathbf{S}_t = a_t \mathbf{S}_{t-1} + \mathbf{u}_t \mathbf{k}_t^\top$ として書き換える。ここで、**実効補正値**（effective corrected value）は $\mathbf{u}_t = \beta_t(\mathbf{v}_t - a_t \mathbf{S}_{t-1} \mathbf{k}_t)$ である。長さ m のチャンクの内部で初期状態を $\mathbf{S}_0$ とし、$a_t \mathbf{S}_{t-1} \mathbf{k}_t = \gamma_t \mathbf{S}_0 \mathbf{k}_t + \sum_{s<t} L_{ts} (\mathbf{k}_s^\top \mathbf{k}_t) \mathbf{u}_s$（ただし $L_{ts} = \prod_{r=s+1}^{t} a_r$、$\gamma_t = \prod_{r=1}^{t} a_r$）を展開すると、狭義下三角な $m \times m$ の連立一次方程式がチャンク内の補正値 $\mathbf{u}_1, \dots, \mathbf{u}_m$ について得られる。

$$\mathbf{u}_t + \beta_t \sum_{s<t} L_{ts} \, (\mathbf{k}_s^\top \mathbf{k}_t) \, \mathbf{u}_s = \beta_t (\mathbf{v}_t - \gamma_t \mathbf{S}_0 \mathbf{k}_t).$$

右辺は $\mathbf{S}_0$ に関してアフィンであるため、`delta_rule_writes` は単位下三角な $m \times m$ の系 $\mathbf{M} = \mathbf{I} + \mathrm{diag}(\beta)\,\mathrm{tril}_{-1}(\mathbf{L} \odot \mathbf{K}\mathbf{K}^\top)$ を、2 つの右辺に対してすべてのチャンクとヘッドにわたり並列に解き、$\mathbf{W} = \mathbf{M}^{-1}(\beta \odot \mathbf{V})$ と $\mathbf{Z} = \mathbf{M}^{-1}((\beta \odot \gamma) \odot \mathbf{K})$ を求める。これにより、$\mathbf{U} = \mathbf{W} - \mathbf{Z}\mathbf{S}_0^\top$ は $\mathbf{S}_0$ が確定した時点で直ちに得られる。

```
def delta_rule_writes(K, V, beta, log_a):
    """W and Z with corrected values U = W - Z S_0^T, for every chunk.

    K: (..., m, d_k); V: (..., m, d_v); beta, log_a: (..., m)
    """
    L = d2l.decay_mask(log_a)  # Products of decays within the chunk
```

```
    gamma = torch.exp(log_a.cumsum(-1)).unsqueeze(-1)  # Decay since start
    beta = beta.unsqueeze(-1)
    M = torch.eye(K.shape[-2], device=K.device) + (
        beta * L.tril(-1) * (K @ K.mT))  # Lower triangular
    W = torch.linalg.solve_triangular(M, beta * V, upper=False)
    Z = torch.linalg.solve_triangular(M, beta * gamma * K, upper=False)
    return W, Z
```

$\mathbf{U} = \mathbf{W} - \mathbf{Z}\mathbf{S}_0^\top$ をチャンク末尾の状態 $\mathbf{S}_m = \gamma_m \mathbf{S}_0 + \mathbf{U}^\top \mathbf{D}\mathbf{K}$（ただし $\mathbf{D}$ はチャンク末尾までの減衰を並べた対角行列）に代入すると、チャンク遷移は T/m 個のチャンクにわたるアフィンな行列漸化式として表される。

$$\mathbf{S}_m = \mathbf{S}_0 \mathbf{A} + \mathbf{W}^\top \mathbf{D}\mathbf{K}, \qquad \mathbf{A} = \gamma_m \mathbf{I} - \mathbf{Z}^\top \mathbf{D}\mathbf{K}.$$

$(\mathbf{A}, \mathbf{W}^\top \mathbf{D}\mathbf{K})$ を T/m 個のチャンクにわたってスキャンするコストはわずか $(T/m)d_k^3$ 回の演算で済み（チャンク内の $O(Td_k^2)$ の行列積と $m \approx d_k$ のときに一致する）、その後は $\mathbf{U} = \mathbf{W} - \mathbf{Z}\mathbf{S}_0^\top$ およびすべての出力 $\mathbf{Y}$ が 1 回のバッチ行列積で求まる。

```
def chunked_delta_rule(Q, K, V, beta, log_a, chunk_size):
    """Outputs y_t = S_t q_t of the gated delta rule, computed in chunks.

    Q, K: (batch_size, T, d_k); V: (batch_size, T, d_v);
    beta, log_a: (batch_size, T)
    """
    B, T, m = Q.shape[0], Q.shape[1], chunk_size
    Q, K, V = (X.reshape(B, T // m, m, -1) for X in (Q, K, V))
    beta, log_a = beta.reshape(B, T // m, m), log_a.reshape(B, T // m, m)
    W, Z = delta_rule_writes(K, V, beta, log_a)
    L, cum = d2l.decay_mask(log_a), log_a.cumsum(-1)
    # Each chunk maps the memory S at its start to S A + W^T D K at its end
    to_end = torch.exp(cum[..., -1:] - cum).unsqueeze(-1)
    I = torch.eye(K.shape[-1], device=K.device)
    A = torch.exp(cum[..., -1, None, None]) * I - (Z * to_end).mT @ K
    def combine(earlier, later):  # A scan over chunks, with matrix products
        (A1, W1), (A2, W2) = earlier, later
        return A1 @ A2, W1 @ A2 + W2
    W_end = d2l.associative_scan(combine, (A, (W * to_end).mT @ K))[1]
    S = torch.cat([torch.zeros_like(W_end[:, :1]), W_end[:, :-1]], 1)
    U = W - Z @ S.mT  # Corrected values, with the memory at each start
    Y = torch.exp(cum).unsqueeze(-1) * (Q @ S.mT) + (L * (Q @ K.mT)) @ U
    return Y.reshape(B, T, -1)
```

`chunked_delta_rule` を検証するため、まずトークンごとに逐次計算する参照実装 `delta_rule_recurrent` を用意する。

```
def delta_rule_recurrent(Q, K, V, beta, a):
    """The same outputs, one step at a time."""
    S, outputs = Q.new_zeros(V.shape[-1], K.shape[-1]), []
    for q, k, v, b, a_t in zip(Q, K, V, beta, a):
        S = a_t * S
```

```
        S = S + b * torch.outer(v - S @ k, k)
        outputs.append(S @ q)
    return torch.stack(outputs)
```

続いて、チャンクサイズ $m \in \{16, 64\}$ の `chunked_delta_rule` を、単位正規化されたキーを持つ長さ $T = 256$ のランダム系列上で `delta_rule_recurrent` と比較する。

```
gen = torch.Generator().manual_seed(0)  # Local: later training stays unseeded
T, d_k, d_v = 256, 32, 32
Q, V = torch.randn(T, d_k, generator=gen), torch.randn(T, d_v, generator=gen)
K = F.normalize(torch.randn(T, d_k, generator=gen), dim=-1)
beta = torch.rand(T, generator=gen)
a = 0.8 + 0.19 * torch.rand(T, generator=gen)
Y = delta_rule_recurrent(Q, K, V, beta, a)
for m in [16, 64]:
    Y_chunked = chunked_delta_rule(Q[None], K[None], V[None], beta[None],
                                   torch.log(a)[None], m)[0]
    print(f'chunk size {m}: relative deviation '
          f'{(Y_chunked - Y).abs().max() / Y.abs().max():.1e}')
```

```
chunk size 16: relative deviation 2.1e-07
chunk size 64: relative deviation 3.1e-07
```

チャンク単位のアルゴリズムは、トークン単位の漸化式と相対誤差 $\sim 10^{-7}$ の精度で一致する。

12.7.3.3 層

`GatedDeltaNet` は `d2l.GatedLinearAttention`（12.6 節）のサブクラスであり、書き込み強度の射影 `W_beta`（ヘッドごとに $\beta_{t,h} = \sigma(\mathbf{w}_{\beta,h}^\top \mathbf{x}_t + b_{\beta,h}) \in (0, 1)$）だけを追加し、`memory` をオーバーライドして `chunked_delta_rule` を呼び出す。それ以外のすべての構成要素（幅 4 の因果畳み込み、ℓ_2 正規化されたクエリとキー、ヘッドごとのスカラー減衰、ヘッドごとの RMS 正規化、および SiLU 出力ゲート）は同一である。

```
class GatedDeltaNet(d2l.GatedLinearAttention):
    """Gated linear attention whose memory is written by the delta rule."""
    def __init__(self, num_hiddens, num_heads, chunk_size=32, conv_width=4,
                 timescales=(4, 256)):
        super().__init__(num_hiddens, num_heads, chunk_size, conv_width,
                         timescales)
        self.W_beta = nn.Linear(num_hiddens, num_heads)  # Write strength
        d2l.init_lecun(self.W_beta)

    def memory(self, X, Q, K, V, log_a):
        beta = d2l.heads_to_batch(torch.sigmoid(self.W_beta(X)),
                                  self.num_heads).squeeze(-1)
        return chunked_delta_rule(Q, K, V, beta, log_a, self.chunk_size)
```

12.7.3.4 再束縛

訓練された `GatedDeltaNet` 層は、`GatedLinearAttention` が苦戦する再束縛されたキーの上書きを実行できるだろうか。1 ブロックの `SeqLM`（$d = 64$、次元 16 のヘッド 4 個、初期減衰時間スケー

ル $[256, 2048]$）を `d2l.associative_recall` でテストする。ここでは 128 種類から選んだ 16 個のキーと 16 種類から選んだ値を用い、16 個のキーのうち 8 個を最初の束縛からわずか 2 回の書き込み後に新しい値へ再束縛する（`num_rebound=8, gap=2`）。この間隔を短く保つことで 2 つの束縛の経過時間がほぼ同じになるため、経過時間に基づく大域的減衰では、1 度しか書き込まれていないキーまで忘れることなしに新しい値と古い値を容易には切り分けられない（2 ブロックの加算型モデルは深さ方向を使って 1 回のデルタステップを近似できるため、ここでは単一ブロックを用いる。演習問題 6 を参照）。

```
def rebinding_model(layer, num_hiddens=64, num_heads=4,
                    vocab_size=16 + 128 + 1):
    """One block of `layer`; tokens: 16 values, 128 keys, the filler."""
    return d2l.SeqLM(vocab_size, num_hiddens, [layer(
        num_hiddens, num_heads, chunk_size=16, timescales=(256, 2048))])
```

`recall_by_kind` を用いて再束縛されたキー（`rebound`）と 1 度だけ書き込まれたキー（`~rebound`）の正解率を分けて集計し、`GatedLinearAttention` と `GatedDeltaNet` をそれぞれ 2 つの乱数シードで 16 ペアの再束縛タスク上で訓練する。

```
def recall_by_kind(model, X, Y, rebound):
    """Recall on the queries for keys written twice and for keys written
    once; a label of -1 matches no prediction."""
    accuracy = lambda mask: float(d2l.query_accuracy(
        model, X, np.where(mask, Y, -1))) / mask.mean()
    return accuracy(rebound), accuracy(~rebound)

task = dict(num_keys=128, num_values=16, num_rebound=8, gap=2)
test = d2l.associative_recall(np.random.default_rng(1), 2048, 16,
                              num_steps=64, **task)
print(f'{"":24s}{"parameters":>12s}{"written twice":>16s}'
      f'{"written once":>16s}')
for name, layer in (('gated linear attention', d2l.GatedLinearAttention),
                    ('Gated DeltaNet', d2l.GatedDeltaNet)):
    models = [rebinding_model(layer) for _ in range(2)]  # Two runs
    for model in models:
        d2l.train_recall(model, pairs=(16,), **task)
    twice, once = zip(*[recall_by_kind(model, *test) for model in models])
    print(f'{name:24s}{d2l.num_params(models[0]):12,d}'
          f'{twice[0]:11.3f} {twice[1]:.3f}{once[0]:11.3f} {once[1]:.3f}')
```

```
                         parameters   written twice    written once
gated linear attention       73,941      0.929 0.859      0.977 0.968
Gated DeltaNet               74,201      0.965 0.969      0.978 0.986
```

260 パラメータの `W_beta` ヘッドを追加するだけで、再束縛されたキーに対する想起率は約 0.9（`gated linear attention`）から 0.96 以上（`Gated DeltaNet`）へと上昇し、1 度だけ書き込まれたキーとの差は約 2 パーセントポイント以内に縮まる。

12.7.3.5 状態追跡

これまでの実験はすべて、何かをメモリに記憶させて後から返すことを求めていた。これとは異なる要求として、トークンごとに変化しうる状態を**追跡**するという課題がある。たとえば、一連の入れ替えの後にどのカップにボールが隠れているか、引用符が開いたままか、あるいは最も単

純な例として、これまでに読み込んだ 1 の個数が奇数か偶数かといった問題である。パリティに必要な状態はわずか 1 ビットであり、1 が来れば反転し、0 が来ればそのまま保たれる。何かを長期間保持する必要はないが、1 つでも見落とすとそれ以降の答えがすべて誤りになるため、何ひとつ飛ばすことは許されない。

GRU はプログラムと同じやり方でこの課題を解く。そのゲートは直前の状態 $\mathbf{h}_{t-1}$ を参照できるため、1 ステップで「入力が 1 なら反転する」という計算を実行できる。本章の線形漸化式は、並列訓練と引き換えにこの仕組みを手放した。すなわち、減衰と書き込みが入力のみに依存する（12.3 節）。記憶されたビットを反転させるには、状態のある座標に負の数を掛けなければならないが、minGRU の減衰 $\mathbf{z}_t \in (0,1)$ は座標を縮小できても符号を変えることは決してできない。デルタ則も、書き込み強度がシグモイドである限り同じ制約を受ける。遷移 Equation 12.36 のキー方向の固有値が $1-\beta_t \in (0,1)$ となるからである。しかし $\beta_t \in (0,2)$ とすれば、固有値は -1 に達しうる。入力が 1 のたびに値 $\mathbf{v}$ を $\beta_t = 2$ で 1 つの固定キー $\mathbf{k}$ に書き込み、0 のときには何も書き込まない層は、1 が来るたびに読み出し値を $\mathbf{S}_t\mathbf{k} = 2\mathbf{v} - \mathbf{S}_{t-1}\mathbf{k}$ へと変化させる。空のメモリから始めれば、読み出し値は $2\mathbf{v}$ と $\mathbf{0}$ の間を交互に行き来し、これがそのままパリティとなる。

では、このような構成法が存在するとして、訓練によってそれを見つけられるだろうか。2 ブロック・幅 64 の `d2l.SeqLM`（この課題で Grazzi et al. (2025) が用いた深さ）を 64 ビットのランダムな列で訓練し、各ステップでそこまでのビットのパリティを出力させる。続いて 256 ビットの列でテストすることで、規則そのものを学習したモデルと、訓練時の長さに適合しただけのモデルとを切り分ける。比較するミキサは、GRU、minGRU、`GatedDeltaNet`、および書き込み強度のシグモイドを 2 倍にした点以外は同じ層である `GatedDeltaNetReflect` の 4 つである。上の議論が正しければ、GRU と $\beta_t \in (0,2)$ を持つ層は訓練長を超える位置を含めすべての位置で正解する一方、minGRU とシグモイドのデルタ則は失敗し、訓練時より長い列になるとランダム予測の水準である 0.5 まで低下するはずである。

```
class GatedDeltaNetReflect(d2l.GatedDeltaNet):
    """Gated DeltaNet with write strengths in (0, 2), so that the transition
    can reflect the memory along the key."""
    def memory(self, X, Q, K, V, log_a):
        beta = 2 * d2l.heads_to_batch(torch.sigmoid(self.W_beta(X)),
                                      self.num_heads).squeeze(-1)
        return chunked_delta_rule(Q, K, V, beta, log_a, self.chunk_size)

def running_parity(rng, batch_size, num_steps):
    """Random bits and, at every step, the parity of the bits so far."""
    X = rng.integers(0, 2, (batch_size, num_steps))
    return X, np.cumsum(X, 1) % 2

def train_parity(model, num_updates=1000):
    """Train on strings of 64 bits; return the accuracy on positions 1-64
    and on positions 65-256 of new strings of 256 bits."""
    rng = np.random.default_rng()
    model.to(d2l.try_gpu())
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-2,
                                  weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lambda t: (
        d2l.cosine_warmup(t, num_updates, num_updates // 10)))
    d2l.train_queries(model, lambda step: running_parity(rng, 128, 64),
                      num_updates, optimizer, scheduler)
    X, Y = running_parity(np.random.default_rng(1), 1024, 256)
    return (float(d2l.query_accuracy(model, X[:, :64], Y[:, :64])),
```

```
            float(d2l.query_accuracy(model, X, Y[:, 64:])))

d = 64  # Width; the delta rules have 4 heads of size 16
parity_mixers = {
    'GRU': lambda: d2l.Recurrent(d2l.init_lecun(
        nn.GRU(d, d, batch_first=True))),
    'minGRU': lambda: d2l.MinGRU(d, d),
    'Gated DeltaNet': lambda: d2l.GatedDeltaNet(
        d, 4, chunk_size=32, timescales=(256, 2048)),
    'Gated DeltaNet, 2σ': lambda: GatedDeltaNetReflect(
        d, 4, chunk_size=32, timescales=(256, 2048))}

def parity_model(mixer):  # Tokens and outputs: the bits 0 and 1
    return d2l.SeqLM(2, d, [mixer(), mixer()])  # Two blocks
```

各ミキサについて 2 回ずつ訓練を実行し、訓練中に観測した長さである位置 1〜64 と、位置 65〜256 における正解率を報告する。

```
print(f'{"":24s}{"positions 1-64":>16s}{"65-256":>12s}')
for name, mixer in parity_mixers.items():
    (a, b), (c, e) = [train_parity(parity_model(mixer)) for _ in range(2)]
    print(f'{name:24s}{a:11.2f}/{c:.2f}{b:7.2f}/{e:.2f}')
```

```
                         positions 1-64      65-256
GRU                            1.00/1.00   1.00/1.00
minGRU                         0.52/0.56   0.50/0.50
Gated DeltaNet                 0.80/0.81   0.50/0.50
Gated DeltaNet, 2σ             1.00/1.00   1.00/1.00
```

GRU は両方の実行において、訓練長でもそれを超える領域でもすべての位置で正解している。一方、minGRU とシグモイドのデルタ則はそうならない。最初の 64 位置ではランダム予測の水準を上回り、minGRU は 0.5 をわずかに超え、Gated DeltaNet は約 0.75 から 0.8 に達するものの、どちらも規則を学習してはいない。位置 64 を超えると、両者ともコイントスと同じ 0.50 の正解率に落ち込む。書き込み強度を 2 倍にすると、この結果が一変する。`GatedDeltaNetReflect` は訓練長で 1.00 に達し、その正解率を 256 ビットまで維持する。これは偶然うまくいった 2 回の実行ではなく典型的な結果である。さらに約 50 回実行したところ、1 回を除くすべてがこのように汎化し、その例外も位置 64 以降で約 0.9 のスコアを記録した。つまり、線形性そのものが障害なのではない。線形漸化式が手放したのは状態に依存する更新であり、それにもかかわらず行列メモリが状態を追跡できるかどうかは、遷移の固有値がとりうる範囲によって決まる。minGRU は負の固有値も、自身の状態を参照するゲートも持たない。2 ブロック目を重ねても欠けている符号は補えず、モデルは失敗する。

もっとも、この鏡映は繊細である。2 倍にしたシグモイドは $\beta_t = 2$ に漸近するだけで到達はしないため、反転のたびに記憶されたビットも少しずつ縮小し、書き込み強度が 2 の手前で止まった実行では長い列で正解率が低下する（演習問題 5）。複数のカップの入れ替えを追うような、より難しい状態追跡には、1 トークンあたり複数の鏡映が必要となる。

12.7.4 回帰としてのメモリ

一歩引いて眺めると、本章のすべての行列メモリ、さらにはソフトマックスアテンションも、系列を読み進めながらキー $\mathbf{K}_t \in \mathbb{R}^{t \times d_k}$ から値 $\mathbf{V}_t \in \mathbb{R}^{t \times d_v}$ への回帰予測器をその場で当てはめる

操作として理解できる (K. A. Wang et al. 2025)。$\sum_{s\le t}\|\mathbf{S}\mathbf{k}_s-\mathbf{v}_s\|^2$ の厳密な線形最小二乗法による最小点は次の式で与えられる。

$$\mathbf{S}_t^\star=\mathbf{V}_t^\top\mathbf{K}_t\,(\mathbf{K}_t^\top\mathbf{K}_t)^{-1}, \tag{12.38}$$

ここで、$(\mathbf{K}_t^\top\mathbf{K}_t)^{-1}$ は $t<d_k$ のとき擬似逆行列に置き換えられる。Equation 12.38 の 2 つの因子に注目しよう（Table 12.2）。

- **線形アテンション**は最初の因子である非正規化交差モーメント $\mathbf{V}_t^\top\mathbf{K}_t=\sum_{s\le t}\mathbf{v}_s\mathbf{k}_s^\top$ のみを保持し、重なり合うキーを無相関化するグラム逆行列 $(\mathbf{K}_t^\top\mathbf{K}_t)^{-1}$ を省略する。
- **デルタ則**はトークンごとに Equation 12.38 に向かってオンラインの確率的勾配ステップを 1 回進めるのに対し、**Gated DeltaNet** は各ステップの前に重み減衰（weight decay、$a_t<1$）を加える。陽的な勾配ステップを近接陰的ステップ（proximal implicit step）$\arg\min_{\mathbf{S}}\frac{1}{2}\|\mathbf{S}-\mathbf{S}_{t-1}\|^2+\frac{\beta_t}{2}\|\mathbf{S}\mathbf{k}_t-\mathbf{v}_t\|^2$ に置き換えると、正規化されたステップサイズ $\beta_t/(1+\beta_t\|\mathbf{k}_t\|^2)$ を持つ Longhorn が得られ (B. Liu et al. 2024)（演習問題 3）、一方で $(\mathbf{K}_t^\top\mathbf{K}_t+\lambda\mathbf{I})^{-1}$ を陽に保持すれば逐次最小二乗法（recursive least squares）が得られる（演習問題 4）。
- **ソフトマックスアテンション**は、保存されたすべてのペアに対してクエリ時に**ノンパラメトリック**な Nadaraya–Watson カーネル回帰（10.1.2 節）を実行し、$O(t)$ の KV キャッシュというコストと引き換えにパラメトリックな圧縮を回避する。

厳密な当てはめも、1 ペアずつ最新の状態に保ち続けることができる。各ペアは $\mathbf{K}_t^\top\mathbf{K}_t$ をランク 1 の項 $\mathbf{k}_t\mathbf{k}_t^\top$ だけ変化させるため、Sherman–Morrison の公式（33.2.3.5 節）によって新たな逆行列計算を行うことなく逆行列を更新できる。これが逐次最小二乗法である。この手法は、メモリの隣にサイズ $d_k\times d_k$ の第 2 の状態を保持する（演習問題 4）。

Table 12.2: これまでに読み込んだペアに対する回帰としてのメモリ。

メモリ	当てはめる対象	方法	状態
ソフトマックスアテンション	すべての値のカーネル重み付き平均	クエリ時	全ペア
線形アテンション	$\mathbf{V}^\top\mathbf{K}$（キー補正なしの最小二乗法）	1 ペアにつき 1 回の加算	$d_v\times d_k$
ゲート付き線形アテンション	同上（直近のペアに重み付け）	1 ペアにつき 1 回の加算	$d_v\times d_k$
デルタ則	最小二乗法の近似	1 ペアにつき 1 回の勾配ステップ	$d_v\times d_k$
Gated DeltaNet	重み減衰付き最小二乗法の近似	1 ペアにつき 1 回の勾配ステップ	$d_v\times d_k$

これらすべてのアーキテクチャにおいて、学習は 2 つの時間スケールで進行する。状態 $\mathbf{S}_t$ の**高速重み**（fast weights）が順伝播中に現在のコンテキストにおけるキー・値の対応関係を当てはめる一方、ネットワークの**低速重み**（slow weights、$\mathbf{W}_{qkv},\mathbf{W}_a,\mathbf{W}_\beta,\ldots$）は、そのオンライン当てはめが有効に働くようなキー、値、クエリ、ステップサイズを生成するよう、系列をまたいで誤差逆伝播法により訓練される。しかし、どれほど最適な書き込み規則を用いても、$d_v\times d_k$ 行列のランクとビット容量は有限である。12.8 節ではその厳然たる限界を測定し、ごく一部のフルアテンション層を挟み込むことでその限界をどのように引き上げられるかを示す。

12.7.5 まとめ

加算型行列メモリは同じキーへのすべての書き込みを重ね合わせるため、再束縛されたキーをクエリすると古い値と新しい値の和が返される。デルタ則は書き込み前に現在の予測 $\mathbf{S}_{t-1}\mathbf{k}_t$ を読み出し、誤差補正 $\beta_t(\mathbf{v}_t - \mathbf{S}_{t-1}\mathbf{k}_t)\mathbf{k}_t^\top$ のみを加える。これは $\frac{1}{2}\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2$ に対する勾配降下法の 1 回のオンラインステップであり、その遷移 $\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top$ はメモリを $\mathbf{k}_t$ 方向に選択的に縮小する。対象を絞った消去はキーが再束縛される場合や相関を持つ場合に加算的書き込みを上回る一方、多数のランダムなキーが重なり合う状況で強い消去（$\beta_t = 1$）を行うと初期の信号が浸食されうる。`GatedDeltaNet` は入力依存の書き込み強度 β_t とヘッドごとのスカラー減衰 a_t を組み合わせ、下三角求解（`delta_rule_writes`）とチャンク単位のスキャン（`chunked_delta_rule`）によって漸化式を並列チャンクで評価し、連想想起において再束縛されたキーを確実に復元する。

12.7.6 演習問題

1. **順序は結果に影響しない。**一連の書き込みを行った後の加算型メモリが、書き込みの順序に依存しないことを示せ。ここから、いかに巧妙な読み出し規則であっても、加算型メモリだけからは 1 つのキーに書き込まれた R 個の値のうち最新のものを復元できないことを結論づけ、さらに R 個の値がすべて異なるときに最良の規則が達成できる想起率を求めよ。コードブックのサイズを 32 個から 1,024 個の値へ増やしたとき、8 回の書き込み後に測定される想起率はどのように変化するか。また、それはなぜか。

2. [code] **補正の代わりに減衰を用いる。** `overwrite_recall` の加算的書き込みに定数減衰を与え、$\mathbf{S} \leftarrow a\mathbf{S} + \mathbf{v}\mathbf{k}^\top$ とせよ。

    1. $R = 2, 4, 8$ に対して、最新の値が少なくとも 95% の割合で想起される最大の減衰 a を求めよ。なぜその答えは同じキーに対する 2 回の書き込みの間に何回の書き込みが挟まるかに依存するのか。また、このテストにおいてより小さな減衰が決して悪影響を与えないのはなぜか。
    2. 次に、正規直交キーを幅 64 における 16 個のランダムな単位キーに置き換えよ。そのうち 8 個をこれまで通り R 回書き直し、残りの 8 個は最初に 1 度だけ書き込んで、最後に両方の種類のキーに対する想起率を測定せよ。どちらの種類のキーにも対応できる書き込み規則はどれか。

3. **陰的ステップ。** $\frac{1}{2}\|\mathbf{S} - \mathbf{S}_{t-1}\|^2 + \frac{\beta}{2}\|\mathbf{S}\mathbf{k} - \mathbf{v}\|^2$ の最小点が $\mathbf{S}_{t-1} - \frac{\beta}{1+\beta\|\mathbf{k}\|^2}(\mathbf{S}_{t-1}\mathbf{k} - \mathbf{v})\mathbf{k}^\top$ であることを示せ（ヒント：勾配を 0 と置き、その結果に $\mathbf{k}$ を掛けてまず $\mathbf{S}\mathbf{k}$ を求めるとよい）。さらにその遷移の固有値を求め、このステップが新しいペアを厳密に想起するのは極限 $\beta \to \infty$ の場合に限られることを示せ。

4. [code] **逐次最小二乗法。** Equation 12.38 に $\lambda \to 0$ で漸近する正則化最小二乗法の当てはめは、メモリが $\mathbf{P}_t = (\mathbf{K}_t^\top\mathbf{K}_t + \lambda\mathbf{I})^{-1}$ も併せて保持すれば 1 ペアずつ更新できる。

    1. 33.2.3.5 節の Sherman–Morrison の公式を用いて、新しいペアが 1 つ与えられた後の $\mathbf{P}_t$ と $\mathbf{S}_t$ の更新式を導出し、$\mathbf{S}_t$ の更新が $\mathbf{P}_t\mathbf{k}_t$ 方向に（$\mathbf{k}_t$ ではなく）書き込むデルタ則であることを示せ。デルタ則の $d_v d_k$ 個に対し、この状態はいくつの数値を保持するか。
    2. これを実装し、上書きテストにおいてデルタ則と比較せよ。すべてのペアに対する最良の当てはめである逐次最小二乗法が、最新の値をうまく想起できないのはなぜか。また、どうすればこれを解決できるか。

5. [code] **パリティ。**●`GatedDeltaNet` の書き込み強度はシグモイドであるため、キー方向の遷移の固有値 $1-\beta_t$ は $(0, 1)$ に収まる。$2\sigma(\cdot)$ を $\sigma(\cdot)$ の代わりに用いると、固有値は $(-1, 1)$ に収まるようになる。

1. それでもなお、訓練された層は $\beta_t = 2$ に漸近することしかできない。12.7.3.5 節のパリティの構成法において、カウント対象のすべてのトークンで $\beta = 2 - \epsilon$ と置き、カウント対象のトークンが n 個続いた後の読み出し値が $\big(1 - (\epsilon - 1)^n\big)\mathbf{v}$ となることを示せ。奇数回カウントした後と偶数回カウントした後の読み出し値の差は、どれほどの速さで縮小するか。$\epsilon = 0.01$ のとき、この差が $\|\mathbf{v}\|$ の 1% 未満になるのはカウント対象のトークンが何個続いた後か。また、それは有限精度での読み出しにとって何を意味するか。
2. 12.7.3.5 節では訓練長の 4 倍でモデルをテストした。`Gated DeltaNet, 2σ`を改めて訓練し、1,024 ビットおよび 4,096 ビットの列でテストせよ。正解率が低下し始めるのはどの長さか。また、その長さは最初の小問の答えと整合しているか。

6. [code] **再束縛への別の経路**。再束縛の実験では、書き込み規則だけが値を補正できるように 3 つの設定を固定していた。これらを 1 つずつ緩和し、それぞれの結果を説明せよ。

   1. ゲート付き線形アテンションのメソッド `decays`（$\log a_t$ を返す）をオーバーライドして同じ形状のゼロを返すようにし、すべての $a_t = 1$ として減衰を無効化したうえで再度訓練せよ。書き直されたキーと 1 度だけ書き込まれたキーに対する想起率はそれぞれどのように変化するか。
   2. 減衰を元に戻し、各書き直しが後方の任意の位置で起こるようにせよ（`gap=None` を `task` に指定する）。2 つの層の間の差は縮まるか。2 つの値の経過時間を使って説明せよ。
   3. 両方の層を 1 ブロックではなく 2 ブロックで訓練せよ。2 つの加算層を組み合わせることで、どのようにして値を補正できるか。

12.8 ハイブリッドアーキテクチャ

本章で扱ってきたすべてのモデルは、ゲートから選択的ダイナミクス、さらには誤差修正型のデルタ則に至るまで、固定サイズの状態を**どのように**更新するかを改良してきた。しかし、どのモデルも状態のサイズが許す以上の情報を保持することはできない。ユーザーが 64 個の商品コードと価格の表をプロンプトに貼り付け、そのうち 20 個について尋ねたとき、Transformer は完全な KV キャッシュ全体を振り返って正確なトークンを取り出すのに対し、純粋なリカレントモデルには、どの項目が質問されるかを知る前に固定状態へ圧縮した情報しか残っていない。少数ショットプロンプティング（few-shot prompting）、検索拡張生成（retrieval-augmented generation）、コード編集などで必要となるコンテキスト内検索（in-context retrieval）こそが、固定状態の系列モデルが明確な上限に突き当たる場面である。

実運用されているリカレント言語モデルが採用する解決策は実務的なものであり、リカレントブロックの間にごく少数の完全なアテンション層を挟み込むことである。以下では、固定状態による想起に対する 2 つの基本的な限界（情報理論的なビット数え上げ限界と、キーの重なりに関する Welch の幾何学的下界）を導出し、純粋な `GatedDeltaNet`、純粋なアテンション、および `GA` ハイブリッドをマルチクエリ連想想起（multi-query associative recall）とバイトレベルの WikiText-103 上で比較する。さらに、隣接するキーと値のトークンを対にする短いデプスワイズ畳み込みをアブレーションにより検証し、256K コンテキスト長における推論時の KV キャッシュ削減量を定量化し、最後に本章で扱ったすべての系列メモリを総括する。

12.8.1 固定サイズの状態の限界

12.8.1.1 ビットの数え上げ

鳩の巣原理（pigeonhole principle）により、M 個の数値をそれぞれ b ビットで格納する状態は、高々 2^{Mb} 通りの異なる構成しかとり得ない。可能な全 V^k 通りの系列（長さ k トークン、語彙サ

イズ V）をすべて区別して再現するには、状態は次の条件を満たさなければならない (Jelassi et al. 2024)。

$$Mb \geq k \log_2 V. \tag{12.39}$$

したがって、固定状態が想起できる最大トークン数は状態サイズ M に比例して増加し、コンテキスト長 T にはまったく依存しない。これに対し、Transformer の KV キャッシュは T に比例して拡大し、飽和することがない。

実際には、線形行列メモリは生のビット数を使い切るはるか手前で幾何学的な限界に突き当たる。以下で用いる連想想起ベンチマークでは、1 つの系列に最大 64 個のペア（キーは 128 種類の記号、値は 4 種類の記号から選ばれ、テーブル情報量は高々 $128 \times 2 = 256$ ビット、64 個の値に限れば $64 \times 2 = 128$ ビット）を格納する一方、`GatedDeltaNet` 2 層（8 ヘッド、各ヘッドは 4×4 の float32 状態）は $256 \times 32 = 8{,}192$ ビット、すなわち必要量の $30\times$ を超えるビット数を保持する。なぜ行列メモリは、生のビット数を大きく下回る段階でペアを取りこぼすのだろうか。

12.8.1.2 キーの重なり

行列ヘッド $\mathbf{S} \in \mathbb{R}^{d_v \times d_k}$ を加法的に更新する場合（12.6 節）でも、最小二乗解へ向けてデルタ則で更新する場合（12.7 節）でも、任意の値に対して厳密な線形読み出し $\mathbf{S}\mathbf{k}_j = \mathbf{v}_j$ を実現するには、格納されたキーが線形独立でなければならない。ところが、$\mathbb{R}^{d_k}$ には線形独立なベクトルが高々 d_k 本しか存在しない。格納するキーの数 m がヘッド次元 d_k を超えると、訓練によってキーをどのように選んでも、キー同士は**必ず**重なり合うことになる。

命題（Welch の下界）。 $\mathbf{k}_1, \dots, \mathbf{k}_m$ を $\mathbb{R}^{d_k}$ 内の単位ベクトルとし、$m > d_k$ とする。このとき、相異なるすべてのペア $i \neq j$ にわたる二乗内積の平均は、少なくとも次の値以上である (Welch 1974)。

$$\frac{m - d_k}{d_k\,(m-1)}. \tag{12.40}$$

証明。 $\mathbf{G} = \sum_{i=1}^{m} \mathbf{k}_i \mathbf{k}_i^{\top} \in \mathbb{R}^{d_k \times d_k}$ とおく。そのトレースは $\sum_{i=1}^{m} \|\mathbf{k}_i\|^2 = m$ であるため、d_k 個の固有値の和は m となる。コーシー・シュワルツの不等式により、固有値の二乗和（$\|\mathbf{G}\|_F^2 = \sum_{i,j} (\mathbf{k}_i^{\top}\mathbf{k}_j)^2$ に等しい）は少なくとも m^2/d_k である。ここから m 個の対角項 $(\mathbf{k}_i^{\top}\mathbf{k}_i)^2 = 1$ を差し引くと、少なくとも $m^2/d_k - m = m(m-d_k)/d_k$ が $m(m-1)$ 個の非対角ペア全体の和として残る。■

次元 $d_k = 4$ のヘッドでは、$m = 64$ 個の単位キーの平均二乗重なりは少なくとも $60/(4\times 63) \approx 0.24$ となり、ランダムな単位ベクトル（$1/d_k = 0.25$）とほぼ見分けがつかない。ヘッドを $d_k = 16$ に広げると、Welch の下限は $48/(16 \times 63) \approx 0.05$ まで下がる。

ここから、マルチクエリ連想想起において 2 つの明確な実験的特徴が予測される。

1. 純粋な `GatedDeltaNet` の積層モデルは、格納ペア数 n が状態容量を超えると性能が劣化する一方、層あたりの状態要素数を 4 倍（$8 \times 4^2 = 128$ から $2 \times 16^2 = 512$ へ、幅 $d = 32$ は固定）に増やすと、劣化が始まる位置が約 $4\times$ 高いペア負荷へとシフトする。
2. 2 つのリカレントブロックのうちわずか 1 つを完全な因果アテンションに置き換える（`GA`）だけで、容量不足による性能低下は完全に解消される。アテンション層がすべてのトークンをキャッシュに保持するためである。

12.8.2 1 つのアテンション層による想起

12.8.2.1 4 つのモデル

2 ブロック構成の d2l.SeqLM モデル（$d = 32$）4 種類を、マルチクエリ連想想起（num_keys=128、num_values=4、負荷 $n \in \{4, 8, 16, 32, 64\}$）で検証する。

- **Gated DeltaNet（GG）**：8 個のサイズ 4 ヘッド（層あたり 128 個の状態要素）。
- **Gated DeltaNet, 4x state（GG, num_heads=2）**：2 個のサイズ 16 ヘッド（層あたり 512 個の状態要素、パラメータ数は同じ）。
- **attention（AA）**：2 ヘッドの因果アテンション（d2l.AttentionMixer）。$(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ に対して同じ幅 4 のデプスワイズ因果畳み込みを適用し、位置エンコーディングは用いない（rope=False）。
- **hybrid（GA）**：ブロック 1 の GatedDeltaNet に続けて、ブロック 2 に AttentionMixer を配置した構成。

```
def recall_model(layout, num_heads=8, conv_width=4, num_hiddens=32):
    """SeqLM whose mixers are given by letters: G(ated DeltaNet) with
    num_heads heads, A(ttention) with two; tokens: 4 values, 128 keys, and
    the filler."""
    make = {'G': lambda: d2l.GatedDeltaNet(
                num_hiddens, num_heads, conv_width=conv_width,
                timescales=(256, 2048)),
            'A': lambda: d2l.AttentionMixer(num_hiddens, num_heads=2,
                                            conv_width=4, rope=False)}
    return d2l.SeqLM(4 + 128 + 1, num_hiddens, [make[c]() for c in layout])
```

12.8.2.2 想起スイープ

各アーキテクチャにつき 2 つのシードを d2l.train_recall で 2,000 回更新して訓練し、ペア負荷ごとの正解率を評価する。

```
arms = {'Gated DeltaNet': dict(layout='GG'), 'attention': dict(layout='AA'),
        'hybrid': dict(layout='GA'),
        'Gated DeltaNet, 4x state': dict(layout='GG', num_heads=2)}
recall = {}
for name, arm in arms.items():
    models = [recall_model(**arm) for _ in range(2)]
    recall[name] = [d2l.train_recall(model, num_keys=128) for model in models]
    print(f'{name:25s} {d2l.num_params(models[0]):,} parameters')
```

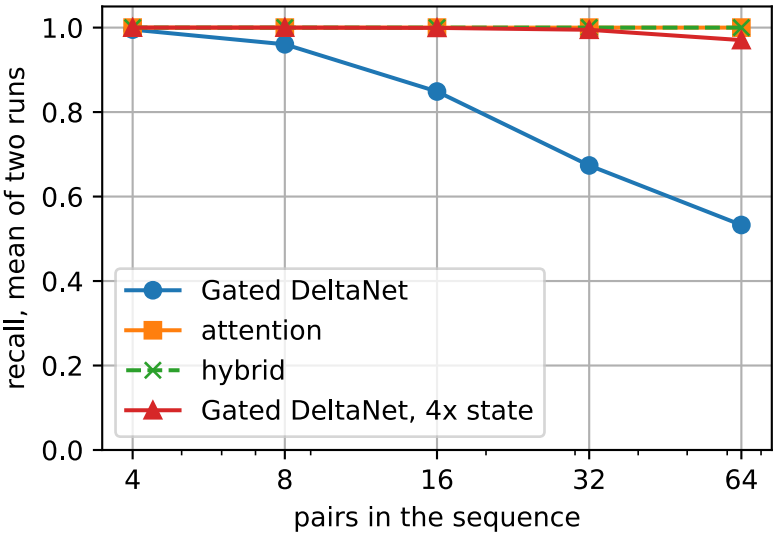
```
Gated DeltaNet            37,989 parameters
attention                 34,821 parameters
hybrid                    36,405 parameters
Gated DeltaNet, 4x state  37,197 parameters
```

続いて、各アームの 2 回の実行にわたって想起曲線を平均し、格納ペア数 $n \in \{4, 8, 16, 32, 64\}$ の関数として正解率をプロットする。

```
pairs = list(recall['hybrid'][0])
runs = {name: np.array([[run[n] for n in pairs] for run in two])
```

```
        for name, two in recall.items()}  # Two rows, one per run
d2l.plot(pairs, [r.mean(0) for r in runs.values()], 'pairs in the sequence',
         'recall, mean of two runs', legend=list(runs), xscale='log',
         ylim=(0, 1.05), fmts=('-o', '-s', '--x', '-^'), figsize=(4.5, 3),
         legend_loc='lower left')
d2l.plt.xticks(pairs, pairs)  # Ticks at the numbers of pairs
print(f'{"pairs":25s}', ''.join(f'{n:>11d}' for n in pairs))
for name, r in runs.items():
    print(f'{name:25s}', ''.join(f'{a:6.2f}/{b:.2f}' for a, b in r.T))
```

```
pairs                              4          8         16         32         64
Gated DeltaNet              1.00/0.99  0.98/0.94  0.87/0.82  0.70/0.64  0.56/0.51
attention                   1.00/1.00  1.00/1.00  1.00/1.00  1.00/1.00  1.00/1.00
hybrid                      1.00/1.00  1.00/1.00  1.00/1.00  1.00/1.00  1.00/1.00
Gated DeltaNet, 4x state    1.00/1.00  1.00/1.00  1.00/1.00  1.00/0.99  0.97/0.97
```



2 つの予測はいずれも成り立っている。

1. **GatedDeltaNet における状態容量のスケーリング：**層あたり 128 個の状態要素（GG）では、想起率は 4 ペアでの約 1.00 から、32 ペアでは 0.6〜0.7、64 ペアでは約 0.5 まで低下する。状態を層あたり 512 要素へ 4 倍に増やす（Gated DeltaNet, 4x state）と、32 ペアまで想起率は 1.00 付近を維持する。64 ペアでも大半のペアを想起でき、実行によって約 0.8〜1.0 の範囲となる。
2. **1 つのアテンション層が厳密な想起を回復する：**attention（AA）と hybrid（GA）はいずれも、すべての実行において最大 64 ペアまでのあらゆる負荷で 1.00 の想起率を達成しており、Waleffe et al. (2024) による大規模実験の知見と一致する。

12.8.2.3 短い畳み込み

なぜリカレントミキサやアテンションミキサの前に幅 4 のデプスワイズ因果畳み込みを入れたのだろうか。連想想起では、各値はトークン t において、トークン $t-1$ のキーの直**後**に到着する。局所的なトークン混合がない場合（conv_width=1）、ステップ t の単一の行列状態層は値トークン

しか見ることができず、直前のキートークンと結び付けることができない。GG で conv_width=1 と設定し、この畳み込みのアブレーションを試してみよう。

```
no_conv = [d2l.train_recall(recall_model('GG', conv_width=1), num_keys=128)
           for _ in range(2)]  # Width 1: only scales and shifts each channel
print(f'{"pairs":25s}', ''.join(f'{n:>11d}' for n in pairs))
for name, (a, b) in [('Gated DeltaNet', recall['Gated DeltaNet']),
                     ('without the convolution', no_conv)]:
    print(f'{name:25s}', ''.join(f'{a[n]:6.2f}/{b[n]:.2f}' for n in pairs))
```

```
pairs                             4          8         16         32         64
Gated DeltaNet             1.00/0.99  0.98/0.94  0.87/0.82  0.70/0.64  0.56/0.51
without the convolution    0.81/0.84  0.64/0.69  0.51/0.55  0.42/0.45  0.36/0.38
```

幅 4 の因果畳み込みを取り除くと、すべての負荷で想起率が低下する（約 10〜35 パーセントポイント低下し、4 ペアの想起率でさえ 0.8〜0.9 まで落ち込む）。

12.8.3 同規模での言語モデリング

純粋な GatedDeltaNet（GGGG）、3 対 1 のハイブリッド（GGGA）、および純粋な因果 attention（AAAA）は、バイトレベルの WikiText-103 ベンチマーク（d2l.SeqLM、幅 $d = 256$ の 4 ブロック構成）においてどのように比較されるだろうか。この問いを 2 段階で検証する。最初は本章を通じて用いてきた 128 バイトのウィンドウで、次はより長いコンテキストによって固定状態と拡大するキャッシュに差がつくかを確かめるため 1,024 バイトのウィンドウで比較する。モデルが離れた位置のコンテキストを活用できるようになるのは訓練の後半であるため、いずれの実験でも前節までの 1,000 ステップではなく 8,000 ステップ訓練する。したがって、以下の損失はこれまでのテキスト実験の表よりも低く、それらと直接比較することはできない。

12.8.3.1 短いウィンドウ

まず、128 バイトの WikiText103 データセットを読み込む。

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
```

公平な比較とするには、もう 1 つのアームが必要である。Mamba、ゲート付き線形アテンション、および Gated DeltaNet はいずれも再帰の前に幅 4 の因果畳み込みで隣接トークンを混合しているが、これまでのテキストベンチマークで用いたアテンションミキサにはそれがなかった。そのため、GatedDeltaNet が attention を上回ったとしても、その差はメモリではなく畳み込みに由来する可能性がある。そこで想起実験と同様に、4 つ目のアームとしてアテンションにも同じ畳み込みを与える。

そのために、text_model(layout, d) は 4 ブロックの d2l.SeqLM を構築する。各ブロックのミキサは 4 文字の文字列 layout に従い、'G'（GatedDeltaNet、4 ヘッド）、'A'（AttentionMixer、8 ヘッド）、'C'（幅 4 の因果畳み込みの後ろに配置した同じアテンション）、および 'R'（後述する長いウィンドウ用の GRU）に対応する。

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
text_mixers = {'R': lambda d: d2l.Recurrent(d2l.init_lecun(
                    nn.GRU(d, d, batch_first=True))),
               'G': lambda d: d2l.GatedDeltaNet(d, num_heads=4),
```

```
                'A': lambda d: d2l.AttentionMixer(d, num_heads=8),
                'C': lambda d: d2l.AttentionMixer(d, num_heads=8,
                                                  conv_width=4)}

    def text_model(layout, d):  # 256 byte values; one block per letter
        return d2l.SeqLM(256, d, [text_mixers[c](d) for c in layout]).to(
            d2l.try_gpu())
```

'GGGG'、'GGGA'、'AAAA'、'CCCC' のそれぞれについて、幅 $d = 256$ で 8,000 ステップの訓練を d2l.text_table により 2 回ずつ実行する。

```
    arms = {'Gated DeltaNet': ('GGGG', 256), 'hybrid': ('GGGA', 256),
            'attention': ('AAAA', 256),
            'attention, conv': ('CCCC', 256)}  # Width 256, four blocks each
    trained = d2l.text_table(text_model, arms, data, num_steps=8000)
```

```
                            validation        0-15       16-63      64-127   ms/step
Gated DeltaNet    3,573,536 1.161/1.158 1.397/1.399 1.132/1.128 1.124/1.120 48/50
hybrid            3,501,848 1.153/1.154 1.399/1.398 1.123/1.124 1.114/1.114 44/43
attention         3,286,784 1.182/1.185 1.410/1.412 1.152/1.155 1.146/1.151 21/21
attention, conv   3,302,144 1.150/1.154 1.397/1.397 1.121/1.124 1.110/1.116 26/26
```

128 バイトのウィンドウでは、4 つのアームのうち 3 つがほぼ同等の結果に落ち着く。Gated DeltaNet はバイトあたり約 1.16 nat、GGGA ハイブリッドと畳み込み付きアテンションは約 1.15 である。畳み込みなしの attention だけが約 0.03 遅れて 1.18 と離れている。この長さでは損失は局所的な構文に支配されており、ウィンドウの先頭 16 バイト以降はほとんど変化しない（12.1.4.2 節）ため、この表は主にミキサが隣接バイトをどれだけうまく結合できるかを測っている。

したがって、素のアテンションの遅れはすべて畳み込みの有無で説明がつく。畳み込みを加えれば、アテンションはハイブリッドと並ぶ。どちらも Gated DeltaNet を高々 0.01 上回るにすぎず、これは同一アームの 2 回の実行間に生じる差と同程度であるため、そこにランキングは読み取らない。短いウィンドウでは、行列メモリにもキャッシュにも局所的な混合では説明できないほどの優位性はなく、ハイブリッドが持つ 1 つのアテンション層も測定可能な効果をもたらさない。この層が重要になるのは、厳密な長距離参照が必要とされる場面である（演習問題 5）。では、ウィンドウが十分に長くなれば、実際のテキストでもこの層は意味を持つのだろうか。

12.8.3.2 長いウィンドウ

128 バイトのウィンドウでは、拡大するキャッシュに対して固定状態が何を失うかも、行列メモリに対してベクトル状態が何を失うかも見えてこない。そこで、1,024 バイトのウィンドウ上で 4 種のミキサ、すなわち GRU の積層（'RRRR'）、Gated DeltaNet、GGGA ハイブリッド、および畳み込み付きアテンションを訓練する。バッチサイズを 64 から 8 に下げることで、各ステップが処理するバイト数と、8,000 ステップ全体で目にするテキスト量を同じに保つ。

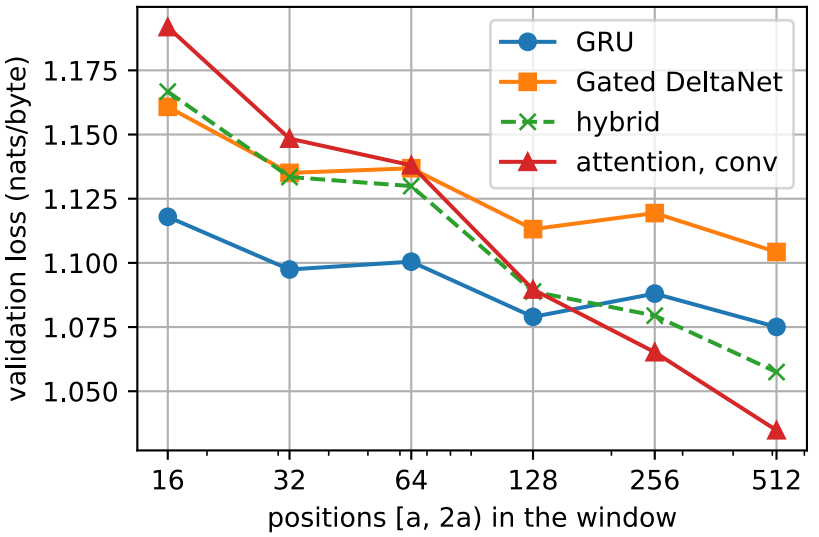
```
    long_data = d2l.WikiText103(batch_size=8, num_steps=1024)  # Same bytes/step
    long_arms = {'GRU': ('RRRR', 256), 'Gated DeltaNet': ('GGGG', 256),
                 'hybrid': ('GGGA', 256), 'attention, conv': ('CCCC', 256)}
    long_trained = d2l.text_table(text_model, long_arms, long_data,
                                  num_steps=8000)  # Long context needs training
```

```
                          validation       0-127     128-511    512-1023   ms/step
GRU                3,817,216 1.088/1.088 1.146/1.144 1.084/1.086 1.075/1.075 119/119
Gated DeltaNet     3,573,536 1.116/1.122 1.183/1.185 1.115/1.120 1.100/1.108 64/63
hybrid             3,501,848 1.083/1.083 1.182/1.184 1.082/1.083 1.058/1.057 54/55
attention, conv    3,302,144 1.073/1.065 1.199/1.190 1.076/1.070 1.040/1.030 32/32
```

ウィンドウ全体の平均をとると、ウィンドウ内のどの位置で各モデルが改善しているかが隠れてしまう。コンテキスト全体を活用するモデルは位置が進むにつれて予測精度を高め続けるのに対し、直近のバイトしか要約しないモデルは早い段階で改善が止まる。両者を見分けるため、長さが倍々に広がる位置の帯域 $[a, 2a)$（バイト位置 16〜31 から 512〜1,023 まで）ごとに、2 回の実行の検証損失を平均する。

```
losses = {name: np.mean([d2l.loss_by_position(model, long_data)
                         for model in models], 0)  # Mean of the two runs
          for name, models in long_trained.items()}
starts = [16, 32, 64, 128, 256, 512]  # Bands [a, 2a) of positions
bands = {name: [loss[a:2 * a].mean() for a in starts]
         for name, loss in losses.items()}
d2l.plot(starts, list(bands.values()), 'positions [a, 2a) in the window',
         'validation loss (nats/byte)', legend=list(bands), xscale='log',
         fmts=('-o', '-s', '--x', '-^'), figsize=(4.5, 3))
d2l.plt.xticks(starts, starts)  # Ticks at the first position of each band
print(f'{"positions":16s}', ''.join(f'{f"{a}-{2 * a - 1}":>10s}'
                                    for a in starts))
for name, band in bands.items():
    print(f'{name:16s}', ''.join(f'{v:10.3f}' for v in band))
```

```
positions            16-31     32-63     64-127    128-255   256-511   512-1023
GRU                  1.118     1.097     1.100     1.079     1.088     1.075
Gated DeltaNet       1.161     1.135     1.137     1.113     1.119     1.104
hybrid               1.167     1.133     1.130     1.089     1.079     1.057
attention, conv      1.192     1.148     1.138     1.090     1.065     1.035
```



ウィンドウを長くすると、両方の長さで訓練したすべてのモデルで性能が向上する。Gated

DeltaNet は 1.11〜1.12、ハイブリッドは 1.07〜1.08 に到達し、いずれも 128 バイトのウィンドウでの 1.16 と 1.15 から改善している。畳み込み付きアテンションは、大半の実行で約 1.07 に達する。ただし約 5 回に 1 回の実行では 1.12 付近にとどまるが、その理由は後述する。GRU は 1.09 に達するものの、ウィンドウを 1 ステップずつ逐次処理するのはこのモデルだけであるため、ステップあたりの計算時間が次に遅いモデルの約 2 倍以上かかり、大差で最も遅いアームとなっている。

短いウィンドウでは横並びだった 3 つのアームにも差が生じており、表の各列からどこで差が開くかがわかる。ウィンドウの最初の 128 バイトから最後の 512 バイトにかけて、GRU の損失は約 0.07、Gated DeltaNet の損失は約 0.08 下がる。これに対し、ハイブリッドは約 0.13、アテンションは大半の実行で約 0.16 改善する。プロットを見ると、改善が起きる位置がさらに詳しくわかる。バイト位置 127 までは固定状態モデルも互角以上に戦っており、すべての帯域で GRU が最も低く、序盤の帯域ではアテンションが最も高く、ハイブリッドは純粋な Gated DeltaNet と見分けがつかない。しかしバイト位置 128 を超えると、GRU と Gated DeltaNet は最後の帯域までに約 0.03 しか改善しなくなる。一方、ハイブリッドの損失は下がり続ける。バイト位置 128 で Gated DeltaNet を引き離し、バイト位置 256 以降は GRU も下回る。バイト位置 512〜1,023 では、Gated DeltaNet を約 0.05、GRU を約 0.02 リードする。このリードはすべて、第 4 ブロックに置かれた 1 つのアテンション層によるものである。

最も大きく損失が下がるのはアテンションだが、すべての実行でそうなるわけではない。大半の実行ではバイト位置 512〜1,023 で 1.03〜1.04 に達し、ハイブリッドを約 0.02 上回って 4 モデル中最高となる。しかし、約 5 回に 1 回の実行は遅れをとる。その場合、全体で約 1.12、最後の帯域で約 1.10 に終わり、Gated DeltaNet と変わらない。プロットは表中の 2 回の実行の平均を示しているため、その一方がこうした遅れた実行にあたると、アテンションの曲線はハイブリッドの下ではなく上で終わることになる。

この 2 種類の実行を分けるのは、たった 1 つの出来事である。最初の 1,000 ステップ前後までは、すべての実行が同じ訓練損失をたどる。その後、大半の実行では 1,000〜5,000 ステップのどこかで損失が約 0.05 急落する。一方、遅れをとった実行では損失がなだらかに下がるだけで、この急落が決して現れない。この急落こそ、アテンションがコピーを学習した瞬間である。テキストには繰り返しが現れる。人名や日付、言い回しは 1 キロバイト以内に何度も再登場することが多い。直前に読んだバイト列の過去の出現箇所を見つけ出し、そのとき後に続いたバイトを予測できるモデルは、そうした繰り返しのたびに損失を減らせる。Olsson et al. (2022) はこの回路を**誘導ヘッド**（induction head）と呼んでおり、本書でも 10.7 節で既に扱った。この回路の有無は直接テストできる。同じ 512 バイトの文章を 2 回含むウィンドウを与えると、急落を経験した実行は 2 回目のコピーを 1 回目の 0.6〜0.8 倍の損失で予測する。遅れをとった実行は、8,000 ステップすべてを通じて両方のコピーをほぼ同じ損失で予測する。コピー元となる情報がほとんどないウィンドウ先頭の 128 バイトでは、この 2 種類の実行を見分けることはできない。

遅れをとった実行は不安定なわけではない。損失が発散することはなく、勾配の大きさも他の実行と変わらない。単に形成が遅いだけなのである。回路が形成されるステップはシードによって変動するが、コサインスケジュールでは 6,000 ステップ前後を過ぎると学習率がほとんど残らないため、それまでに回路を形成できなかった実行は最後まで形成できずに終わる。短い畳み込みは想起タスクと同様ここでも有効であり、これがないと大半の実行では 5,000 ステップまでに急落が起きない。他の学習率、重み減衰なし、異なるアテンションヘッド数も試したが、形成し損ねる頻度が下がる設定は 1 つあったものの、完全にゼロになる設定はなかった。単独ではほとんど役に立たない部品を組み合わせて回路を構築しなければならないとき、このような急落は典型的に生じる現象であり、2 つのアーキテクチャにランキングをつける前に複数のシードで訓練すべき十分な理由となる。

このように、長いコンテキストと十分な訓練があれば、テキスト上でも想起タスクで見られた傾向がより緩やかな形で現れる。拡大するキャッシュが離れたバイトから最も多くの恩恵を受け、3

つのリカレント層に 1 つのアテンション層を混ぜた構成がその改善の大部分を取り戻し、固定状態のみでは改善が最も小さい。128 バイトのウィンドウには利用できるような離れたバイトが存在しないため、短いウィンドウの表からはこの差が見えないのである。

12.8.4 推論時のメモリ

自己回帰生成（autoregressive generation）においては、系列あたりのメモリ量がサービング時の最大バッチサイズと、メモリ帯域幅律速となるデコーディング速度の両方を左右する。

- **グループ化クエリアテンション層**（16 ビット浮動小数点による $n_{\text{kv}} = 8$ 個のキー・値ヘッド、各ヘッドの次元 $d_{\text{head}} = 128$）：**トークンあたり** $2 \times n_{\text{kv}} \times d_{\text{head}} \times 2 = 4$ KB を保持し、層あたり 512 MB（$T = 128$K トークン時）に達する。
- **リカレント・行列状態層**（たとえば幅 4,096、チャネル数 8,192、状態次元 128 の Mamba-2 を 16 ビット浮動小数点で表現した場合）：**コンテキスト長によらず**一定の $8{,}192 \times 128 \times 2 = 2$ MB を保持する（わずか 512 トークンを超えた時点でアテンション層の KV キャッシュに追い抜かれる）。

ハイブリッドアーキテクチャでは、拡大していく KV キャッシュを蓄積するのはアテンション層だけである（Figure 12.17）。4 つのアテンション層を持つ 32 層モデル（Jamba (Lieber et al. 2024) のように 8 層中 1 層の割合）は、256K トークン時のキャッシュを 32 GB から 4 GB へと削減する。また、Qwen3-Next (Qwen Team 2025) や Kimi Linear (Kimi Team 2025b) のような 3 対 1 の Gated DeltaNet ハイブリッドは、KV キャッシュを 4× 縮小する。

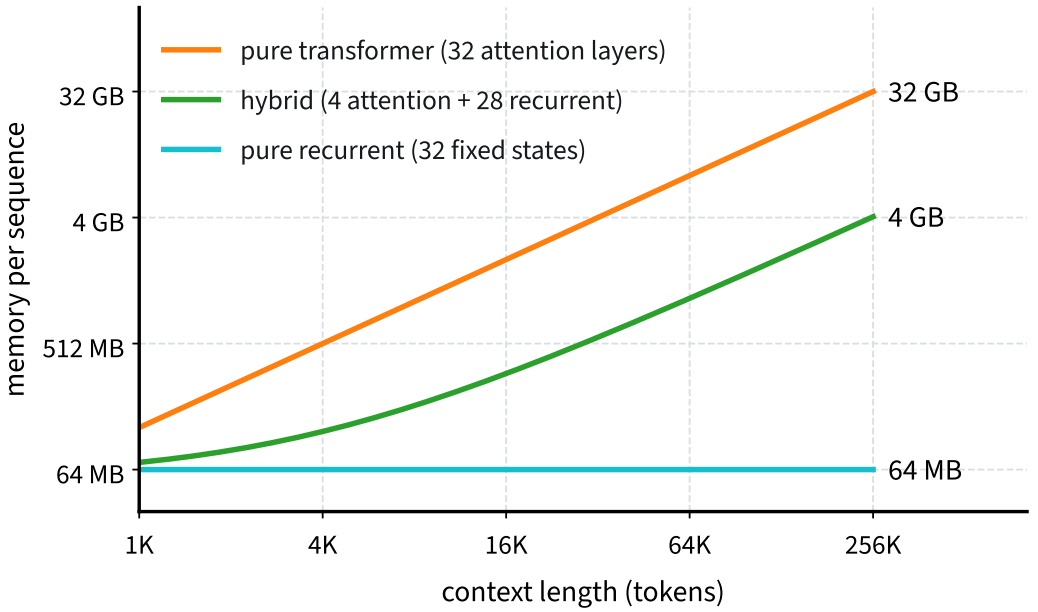


Figure 12.17: グループ化クエリアテンション（次元 128 のキー・値ヘッド 8 個）と Mamba-2 規模の再帰状態を備えた 32 層モデルにおいて、生成中に系列あたり保持されるメモリ量（16 ビット数値）とコンテキスト長の関係。Transformer のキャッシュは 256K トークンで 32 GB に達するが、純粋なリカレントモデルはあらゆる長さで 64 MB にとどまり、4 つのアテンション層を持つハイブリッドは Transformer の 8 分の 1 のキャッシュで済む。

12.8.5 本章のメモリの比較

Table 12.3 に、本章で構築したすべての系列ミキサを因果アテンションおよびハイブリッドと並べて比較する（d はモデル幅、N は状態次元、$H = 2d$ は Mamba の拡張チャネル数、h はヘッド数、T は系列長を表す）。

Table 12.3: 本章のメモリ一覧。「テキスト」列はバイトレベル WikiText-103（128 バイトのウィンドウ、モデルごとに 2 回実行）におけるバイトあたり nat 単位の検証損失であり、括弧内はパラメータ数を示す。本節のモデルは 8,000 ステップ、それ以外は 1,000 ステップ訓練したため、同じステップ数の損失同士のみ比較できる。8,000 ステップ訓練後の 1,024 バイトのウィンドウでは、大半の実行で畳み込み付きアテンションが首位となり、ハイブリッド、GRU、Gated DeltaNet がそれに続く。

メモリ	生成時の層あたり状態	訓練	合成タスク	テキスト
LSTM、GRU	$2d$、d	1 ステップずつ逐次処理	1 付近で初期化したゲートは記号を 200 ステップ保持するが、0.5 で初期化すると 50 ステップまでに失う	1.29〜1.30（4.3 M、3.8 M）
minGRU	d	スキャン	100 ステップ後はランダム予測水準だが、S4D の時間スケールを用いれば 1,000 ステップ後も想起	1.34〜1.35（2.8 M）
S4D	dN	畳み込みまたはスキャン	1,000 ステップ後も想起、選択的コピーでは高々 5 回に 1 回	未訓練
Mamba	HN	スキャン	選択的コピーで 0.98 以上	1.30（4.0 M）
ゲート付き線形アテンション	d^2/h	チャンク処理	128 要素では 64 ペアの約半分、512 要素ではさらに多くを想起	未訓練
Gated DeltaNet	d^2/h	三角求解を伴うチャンク処理	書き換えられたキーの 0.95 以上を想起（ゲート付き線形アテンションは約 0.9）	8,000 ステップ後に 1.16（3.6 M）
アテンション	$2dT$	全位置にわたる融合アテンション	あらゆる負荷ですべてのペアを想起	1.38〜1.39、8,000 ステップ後は 1.18、幅 4 畳み込み付きで 1.15（3.3 M）
ハイブリッド	再帰状態およびアテンション層あたり $2dT$	両方	2 ブロック中 1 つをアテンションにしてすべてのペアを想起	8,000 ステップ後に 1.15（3.5 M）

各列を横断して読むと、設計上のトレードオフが浮かび上がる。

- 任意の過去トークンの厳密な参照が不要な**低消費電力ストリーミングや連続的なセンサー・音声処理**では、古典的なゲート付き再帰や選択的状態空間モデル（SSM）によってコンパクトな $O(1)$ 状態での処理が可能になる。
- **長文脈の基盤モデル**では、チャンク並列のデルタ則行列再帰（GatedDeltaNet）や選択的SSM（Mamba-2）に少数の完全なアテンション層を（1:3〜1:7 の比率で）挟み込むことにより、線形時間の並列訓練、4×〜8× 小さい推論時 KV キャッシュ、および内容参照による厳密な想起を兼ね備えることができる。

12.8.6 まとめ

M 個の数値をそれぞれ b ビットで保持する固定サイズの状態は、高々 2^{Mb} 通りの履歴しか区別できない（$Mb \geq k \log_2 V$ が k 個のトークンを語彙 V から想起するための条件である）。行列状態層では、幾何学的な制約によってさらに早い段階で容量が制限される。Welch の下界により、$m > d_k$ 個の $\mathbb{R}^{d_k}$ 内の単位キーは二乗重なりの平均が少なくとも $(m - d_k)/(d_k(m-1))$ となり、加法的更新とデルタ則更新のいずれにおいてもキー間の干渉を引き起こす。マルチクエリ連想想起では、純粋な GatedDeltaNet モデルはペア数が状態要素数を超えると劣化するが、状態要素数を 4 倍にすると容量も比例して拡大し、リカレントブロックを 1 つだけ完全なアテンションに置き換える（GA）と全負荷で 100% の想起率が回復する。隣接するキートークンと値トークンを単一の層内で結び付けるには、ミキサの前に置く幅 4 の因果畳み込みが不可欠である。バイトレベルのテキストでは差はより小さく、ウィンドウ長に依存する。128 バイトのウィンドウでは、8,000 ステップの訓練後に Gated DeltaNet、ハイブリッド、および畳み込み付きアテンションが互いにバイトあたり約 0.01 nat 以内に収まり、素のアテンションに見られた約 0.03 の遅れも同じ畳み込みを加えれば解消する。1,024 バイトのウィンドウではバイト位置 128 以降で各モデルが分かれ、大半の実行でアテンションが離れたコンテキストから最大の改善を得て、ハイブリッドがそれに続き、純粋な Gated DeltaNet の改善が最も小さい。生成時にはハイブリッドのうちアテンション層のみが拡大する KV キャッシュを必要とするため、推論時のキャッシュメモリはアテンション層の割合に応じて削減される。

12.8.7 演習問題

1. **数え上げ限界**。モデルが長さ k トークンの系列（語彙サイズ V）を読み込み、それを再現しなければならないとする。
   1. Equation 12.39 を、読み込みと書き出しの間の状態が M 個の b ビット数値からなり、出力がその状態の決定論的な関数であるモデルについて証明せよ。
   2. 次に、系列が一様に選ばれるとき、モデルが高々確率 ϵ で誤った系列を出力してもよいとする。このとき $Mb \geq k \log_2 V + \log_2(1-\epsilon)$ が成り立つことを示せ。（ヒント：各状態が正しくデコードできる系列の個数を数える。）
   3. 語彙サイズ 50,000 のトークンを、48 層の Mamba-2 状態（各層は 16 ビット数値で 2 MB）を持つモデルは最大いくつコピーできるか。この規模のどのモデルも実際にはその上限に遠く及ばないのはなぜか。
2. **多数のユーザーのためのメモリ**。80 GB のメモリを持つ GPU が、Figure 12.17 のキャッシュサイズを持つ 32 層モデルをサービングしており、GPU メモリのうち 16 GB は重みに使われているとする。
   1. 長さ 128K トークンの系列を、この GPU は Transformer、4 つのアテンション層を持つハイブリッド、およびリカレントモデルのそれぞれで同時にいくつサービングできるか。
   2. キー・値ヘッド数を減らしたグループ化クエリアテンションと、キャッシュの 8 ビット数値化は、それぞれキャッシュを半分にする。両方を Transformer とハイブリッドに適用し、同じ計算を繰り返せ。2 つの変更のうちハイブリッドの再帰状態も縮小できるのはどちらか。また、両方の変更を適用した後、ハイブリッドのメモリに占める

再帰状態の割合はいくらか。

3. **重なりと訓練**。
   1. 訓練によって、ヘッドのキーをランダムなキーよりどれだけ改善できるか。下界 Equation 12.40 をランダムな単位キーの平均二乗重なり $1/d_k$ で割り、64 個のキーについて次元 4、8、16、32 での比を求めよ。これらのヘッドのうち、訓練されたキーがランダムなキーより明確に優れ得るのはどれか。
   2. [code] スイープ実験で用いた小規模な Gated DeltaNet モデルを 4,000 回および 8,000 回の更新（`num_updates` を `d2l.train_recall` に指定）で訓練せよ。32 ペアおよび 64 ペアの想起率はハイブリッドに近づくか。その結果から、この小規模モデルに不足しているのは訓練量と容量のどちらだといえるか。
   3. [code] キーの候補数を 64 種類および 512 種類としてキーを抽出せよ（`num_keys` を変更し、それに合わせて `recall_model` の語彙も変更する）。1 つの系列が格納するペア数は依然として高々 64 個である。それにもかかわらず、キーの候補数によって想起率が変化し得るのはなぜか。
4. [code] **アテンションをどこに、どれだけ置くか**。
   1. 第 1 ブロックにアテンションを置いたハイブリッド `AG` を訓練し、その想起率を `GA` と比較せよ。アテンション層の位置は結果に影響するか。どの層が各キーとその後続の値を結び付けられるかを踏まえて考察せよ。
   2. 同じ幅を持つ 4 ブロックのモデル `GGGG`、`GGGA`、`GAGA` を訓練せよ。リカレント層を 2 つ増やすと、層あたりの状態を 4 倍にしたときと同じだけ想起曲線は押し上がるか。また、4 層中 1 層のアテンションでも依然としてすべてのペアを想起できるか。
5. [code] **テキストにおける想起**。バイトレベル WikiText-103 の検証ウィンドウにおいて、同じウィンドウ内のより前の位置に既に出現した少なくとも 8 バイトの文字列を完成させるバイトの位置に印を付けよ。
   1. 訓練済みの Gated DeltaNet、ハイブリッド、およびアテンションモデル（`trained`）の損失を、印を付けた位置とそれ以外のすべての位置とで比較せよ。想起におけるアテンションの優位性は、テキストが繰り返される箇所に現れるか。
   2. 512 バイトのウィンドウ（`num_steps=512` とし、ステップあたりのウィンドウ数を減らす）で同じ実験を繰り返せ。印を付けた位置の割合はどう変化し、それに伴ってモデル間の差はどう変わるか。
6. [code] **コピー回路の形成を観察する**。後半が前半の繰り返しになっている 1,024 バイトの検証ウィンドウを作成せよ。畳み込み付きアテンション（`'CCCC'`）を通常の 1,024 バイトのウィンドウ上で訓練し、250 ステップごとに訓練損失と、繰り返しウィンドウの前半の損失に対する後半の損失の比を記録せよ。
   1. この比が 1 を下回るのは何ステップ目か。また、同じステップで訓練損失も急落するか。
   2. 複数のシードで繰り返せ。急落が起きるステップはどの程度変動するか。また、8,000 ステップ以内に急落が起きない実行はどのような挙動を示すか。

12.9 関連文献

本章では、現代の長文脈モデルがアテンションと組み合わせるリカレント層を構築し、その起源を LSTM までさかのぼり、固定サイズの状態がどこで容量の限界に達するかを計測してきた。この分野の文献の大半は発表から 5 年未満のものであり、そのすべてが今後も残るとは限らない。本節では、本章で扱わなかった拡張手法や未解決問題とあわせて、参照実装およびチュートリアルを概観する。

12.9.1 再帰とアテンションの組み合わせ

12.9.1.1 実用におけるハイブリッドモデル

12.8 節の演習問題では、ハイブリッドモデルにどれだけのアテンションが必要か、またスタック内のどこに配置すべきかを問うた。3 つ目の問いは、スライディングウィンドウアテンションで大域アテンション（global attention）を代替できるかという点である。条件を揃えた Waleffe et al. (2024) の 8B パラメータ比較では、24 個の Mamba-2 層と 4 個のアテンション層（MLP を含む全層の約 8%）を組み合わせ、位置埋め込み（positional embedding）が不要になるよう先頭に Mamba-2 層を置き、アテンション層を等間隔に配置している。この比率において、ハイブリッドモデルは 12 個の標準ベンチマークすべてで Transformer に匹敵するかそれを上回ったのに対し、純粋な Mamba モデルは 5-shot MMLU と電話帳ルックアップで遅れをとった。

実用モデルにおける具体的な混合比率は多様である。Qwen3-Next (Qwen Team 2025) と Kimi Linear (Kimi Team 2025b) はデルタ則の漸化式（12.7 節）の周囲で 4 層に 1 層の割合でアテンション層を用い、Jamba (Lieber et al. 2024; Jamba Team 2024) は 8 層に 1 層を用い、Nemotron-H (NVIDIA 2025) は Mamba-2 層を維持している。Griffin (De et al. 2024) は大域アテンションを完全に廃し、2 つのゲート付き対角漸化式層と 1 つのスライディングウィンドウアテンション層（10.6 節）を交互に配置している。ただし、ウィンドウは直近の w トークンしか保持しないため、コンテキストがウィンドウ幅を超えると厳密なルックアップ性能は劣化する。

12.9.1.2 Transformer からの蒸留

リカレントモデルやハイブリッドモデルをゼロから事前学習する代わりに、事前学習済みの Transformer のアテンション射影から行列状態層を初期化し（12.6 節）、蒸留（distillation）によって変換することもできる。Junxiong Wang et al. (2024) は一部のアテンション層を残してハイブリッドを構成する一方、Bick et al. (2024) は元の事前学習予算の 1% 未満にあたる 30〜50 億トークンを用いて、アテンションマップ、隠れ状態、および最終出力を段階的に整合させている。

12.9.1.3 固定状態と完全なキャッシュの間

固定サイズの状態と完全な KV キャッシュの間を補間する設計が 2 つある。Based (Arora, Eyuboglu, Zhang, et al. 2024) は、$\exp(\mathbf{q}^\top \mathbf{k})$ に対する 2 次テイラー近似と厳密なスライディングウィンドウアテンションを組み合わせ、状態サイズと想起性能を直接トレードオフさせる。対数線形アテンション（log-linear attention、(Guo et al. 2025)）は、過去へさかのぼるにつれて長さが 2 倍ずつ増える 2 進（dyadic）履歴ブロックごとに 1 つの行列状態を保持するため、メモリは $O(\log T)$ で増大し、Mamba-2 や Gated DeltaNet にそのまま適用できる。

12.9.2 行列メモリへの書き込み

12.9.2.1 固定状態モデルの表現力

系列タスクの中には、状態サイズをどれだけ大きくしても、対角または時不変な線形漸化式では解けないものがある。5 つのカップの下に隠されたボールを一連の入れ替えにわたって追跡する問題、すなわち対称群（symmetric group）S_5 における置換の合成を考えよう。Merrill et al. (2024) は、対角遷移または固定遷移を持つ状態空間モデル（S4 や Mamba を含む）が、Transformer と同様に、定数深さしきい値回路クラス（constant-depth threshold circuit class）TC^0 に属する関数しか計算できないことを証明している。5 要素以上の置換の合成は NC^1（対数深さ回路）完全であり、したがって TC^0 の外にある（ただし $\mathsf{TC}^0 = \mathsf{NC}^1$ でない限り）。（カップが 3 つまたは 4 つの場合は、対称群が可解群（solvable group）となるため TC^0 の内部にとどまる。）

デルタ則の遷移 $\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$ は入力依存かつ非対角であるため、TC^0 の障壁によっては排除され

ない。しかし Grazzi et al. (2025) は、有限精度のもとでは、遷移の固有値が $[0,1]$ 内に限定されるいかなる線形漸化式も、任意の長さにおいてパリティを計算できず、したがって入れ替えも追跡できないことを示している。標準の `GatedDeltaNet` はシグモイドによる $\beta_t \in (0,1)$ を用いるため、この制約を受ける。$\beta_t \in (0,2)$ を許容すれば、遷移は負の固有値をとれるようになる。実際、$\beta_t = 2$ かつキーが $(\mathbf{e}_i - \mathbf{e}_j)/\sqrt{2}$ のとき、$\mathbf{I} - 2\mathbf{k}_t\mathbf{k}_t^\top$ は座標 i と j を入れ替える鏡映（reflection）となる。12.7.3.5 節では、この派生モデルをパリティ課題で訓練している。

12.9.2.2 一般化デルタ則、指数ゲート、ネットワークメモリ

- **一般化デルタ則：**RWKV-7 (Peng et al. 2025) は、各座標を固有のレートで減衰させ、あるキーベクトルに沿って消去しつつそれとは異なるキーベクトルに沿って書き込むことでデルタ則のランク 1 射影の共有を解き、任意の正規言語（regular language）を認識可能な非対称遷移を実現している。DeltaProduct (Siems et al. 2025) はトークンごとに複数の鏡映 $\prod_i (\mathbf{I} - \beta_i \mathbf{k}_i \mathbf{k}_i^\top)$ を合成し、1 ステップでより高ランクの置換を表現できるようにしている。
- **指数ゲート：**シグモイド入力ゲートは、単一トークンの書き込み重みを最大 1 に制限する。xLSTM (Beck et al. 2024, 2025) の mLSTM は、指数入力ゲートを逐次正規化項の状態および 10.5 節のオンライン対数最大値（log-max）安定化と組み合わせて用いる。
- **ネットワークメモリ：**TTT (Y. Sun et al. 2024) と Titans (Behrouz et al. 2025) は、12.7 節におけるオンライン勾配の視点を一般化し、行列 $\mathbf{S}_t$ をトークンごとの勾配ステップで更新される線形層や小規模な MLP に置き換える（Titans ではモメンタムと重み減衰も併用する）。非線形な高速重みが、同じパラメータ数の行列よりも状態の 1 バイトあたり多くの対応関係を想起できるかどうかは、依然として未解決の問いである。

12.9.3 状態空間モデルと長い畳み込み

12.9.3.1 多項式射影による記憶

12.4 節では、S4D のレートを $\lambda_n = -n$ と初期化した。この選択は HiPPO (Gu et al. 2020) に由来する。HiPPO は、次数 $(N-1)$ の直交多項式による入力履歴 $x(s)$（$s \le t$）の最良近似の係数を保持する手法である。これらの係数は、固定された下三角の HiPPO 行列 $\mathbf{A}$ を持つ線形常微分方程式（ODE）を満たす。S4 (Gu, Goel, et al. 2022) はこの系を時不変にし（指数関数的に時間伸縮された履歴上の Legendre 多項式に対応する (Gu et al. 2023)）、高速なカーネル評価のために $\mathbf{A}$ を正規行列（normal matrix）と低ランク補正の和へと分解した。S4D (Gu, Gupta, et al. 2022) は、対角成分（複素固有値、または実対角成分 $-1, \dots, -N$）のみを残すだけで、はるかに単純な仕組みでほぼ同等の性能を回復できることを示した。S5 (Smith et al. 2023) はチャネルごとのスカラー SSM を、並列スキャン（12.3.2 節）で評価される単一の多入力・多出力対角 SSM に置き換えており、Gu の博士論文 (Gu 2023) がこの一連の研究を体系的にまとめている。

12.9.3.2 Mamba の後継モデル、直接的な線形漸化式、長い畳み込み

- **Mamba-2 と Mamba-3：**Mamba-2 (Dao and Gu 2024) は各ヘッド内でスカラー減衰を共有することで、訓練を 12.6 節の行列積に基づくチャンク形式へと帰着させる。Mamba-3 (Lahoti et al. 2026) は、台形離散化（trapezoidal discretization）、回転追跡のための複素状態、および自己回帰デコーディング時の演算密度（arithmetic intensity）を高める多入力更新を導入している。
- **Linear Recurrent Unit（LRU）：**@Orvieto.Smith.Gu.ea.2023 は、単位円板内の複素固有値と入力正規化 $\sqrt{1-|\lambda|^2}$ を用いて離散対角漸化式を直接パラメータ化し、Long Range Arena において連続時間 SSM に匹敵する性能を達成するとともに、Griffin で用いられている実数値版 RG-LRU の動機を与えた。

- **長い畳み込み：**H3 (Fu et al. 2023) はシフト SSM と乗法的連想 SSM を直列に連結し、Hyena (Poli et al. 2023) は SSM カーネルを完全に廃し、FFT によって $O(T \log T)$ で評価される陰的にパラメータ化された全長フィルタに置き換えた。一般の長い畳み込み（long convolution）は $O(1)$ デコーディングのためのコンパクトな再帰状態を持たないため、言語モデルでは主にリカレント SSM が選ばれてきたが、ゲノミクス分野では長い畳み込みが依然として広く用いられている。

12.9.4 言語を超えて

言語モデリング以外にも、画像パッチ上の双方向スキャンが Vision Mamba (Zhu et al. 2024) や VMamba (Y. Liu et al. 2024) を支えており、SaShiMi (Goel et al. 2022) ではマルチスケールの S4 スタックが生の音声波形をモデル化している。さらに、100 万ヌクレオチド規模のゲノムコンテキストは、HyenaDNA (Nguyen et al. 2023) では長い畳み込みにより、Caduceus (Schiff et al. 2024) では逆相補鎖同変（reverse-complement-equivariant）な双方向 Mamba 層によってモデル化されている。

12.9.5 参照実装とチュートリアル

- **実装：**The Annotated S4（Rush & Karamcheti, 2022）は、完全な S4 を 1 行ずつ構築する。state-spaces/mamba（Gu & Dao）は Mamba および Mamba-2 の `ssd_minimal` の参照実装を提供しており、mamba-minimal は単一ファイルの PyTorch 版 Mamba を与えている（12.5 節）。flash-linear-attention は、GLA、DeltaNet、および Gated DeltaNet 向けの Triton カーネルを提供している（12.6 節、12.7 節）。Gated DeltaNet for Linear Attention（Raschka, 2025）は Gated DeltaNet ハイブリッドをゼロから構築するチュートリアルを提供しており、zoology は 12.6 節および 12.8 節で使用した合成連想想起スイートを提供している。
- **解説とサーベイ：**On the Tradeoffs of SSMs and Transformers および State Space Duality (Mamba-2), parts I–IV（Gu & Dao, 2024〜2025）、A Visual Guide to Mamba and State Space Models（Grootendorst, 2024）、DeltaNet Explained, parts I–III（Yang, 2024）、ならびに Test-Time Regression（Wang, Shi & Fox, 2025）がある。最後の文献は、12.7 節の末尾で扱ったメモリの統一的な回帰としての見方をまとめている。

Chapter 13

計算性能

ここまでは、どのモデルを訓練するときも、ネットワークを書き、訓練ループを開始して完了を待つだけであり、ステップ時間とメモリは所与のものとして扱ってきた。しかし、実行に何日もかかる場合や、モデルがカードに収まらない場合、あるいは 2 基目の GPU を追加するとかえって訓練が遅くなる場合には、そのやり方は通用しなくなる。GPU により多くの処理を与えるため、11.2 節の GPT の幅を 2 倍に広げたものを考えてみよう。1 回の訓練ステップの算術演算量を GPU のピーク演算速度で割れば、そのステップが取り得る最短時間が求まるはずだが、実測したステップにはその数倍の時間がかかる。残りの時間はどこへ消えているのだろうか。

コンピュータアーキテクトは、深層学習よりはるか前からこの問いに直面していた。1990 年代までにはプロセッサの速度がメモリを大きく引き離し、多くのプログラムが実行時間の大部分をデータ待ちに費やすようになっていた。2009 年に Williams、Waterman、Patterson は、その帰結を**ルーフライン**モデルとして定式化した (Williams et al. 2009)。すなわち、プログラムは演算器が許す速度を超えて動作することはなく、また必要なバイト数をメモリが供給できる速度を超えて動作することもない。Python から駆動される GPU には、各演算を発行するホスト側のコストという 3 つ目の制約が加わる。これら 3 つのうちどれがステップを律速しているかによって、どの変更が高速化につながるかが決まる。この視点に立てば、大きなミニバッチや 16 ビット数値（5.5 節）といった前章までの選択の理由が説明できるとともに、次のような方法論が得られる。ステップの算術演算量とバイト数を数え、実行時間を予測し、実測して、その差を説明するのである。

すべての土台となる 13.1 節では、1 バイトの移動あたりに演算が実行する FLOPs である**演算密度**（arithmetic intensity）を導入し、この方法論を実際に適用する。ハードウェアの節（13.2 節）では、ピーク演算性能と帯域幅がどこから生じるのかを説明する。コンパイル（13.3 節）と 13.4 節のメモリ手法は、多くの実務者が日常的に使う単一 GPU 向けのツールである。続く 3 つの節では GPU を増やし、第一原理からのデータ並列（data parallelism、13.5 節）と実運用におけるデータ並列（13.6 節）を扱ったのち、重み、勾配、オプティマイザ状態を複数のカードにシャーディングする手法（13.7 節）を導入して、単一の GPU には収まらないほど大きなモデルを訓練できるようにする。最後に、ケーススタディ（13.8 節）では上述の GPT に立ち返り、1 基および 4 基の GPU 上でそのステップ時間の内訳をミリ秒単位で明らかにする。

本章では、GPU コードの実行時間とピークメモリを計測でき（5.7 節）、bf16 と tf32 を理解しており（5.5 節）、GPT とそのキー・値キャッシュに触れていること（11.2 節、11.3 節）を前提とする。すべての計測値は、4 枚の RTX 4090 カード、PyTorch 2.11、JAX 0.12 を備えた 1 台のホスト上で取得したものである。手元のマシンでは実行時間こそ異なるだろうが、FLOPs とバイト数のカウントは変わらず、それらを予測へと変換する方法論も変わらない。

13.1 性能モデル

これまで訓練してきたすべてのネットワークは算術演算の大半を行列積に費やしており、GPU はそれをピーク速度に近い速さで計算すると暗黙のうちに仮定してきた。しかし、その仮定は誤っていることが多い。本書のノートブックを実行している RTX 4090 では、2 つの 8192×8192 行列の bf16 行列積はハードウェアのピーク性能である 165 TFLOP/s から数パーセント以内で動作する一方、256×256 の行列積は適切に計測してもピークの 2% 未満にしか達しない。同じ演算、同じチップであるにもかかわらずである。訓練ステップは多数の演算の連なりであり、その中には小さな演算も含まれる。それらの大半がピークのごく一部でしか動作していないとき、より高速な GPU や 2 枚目のカードを導入してもほとんど改善しない。時間が計算ではなく待ちに費やされているからである。

GPU 上のあらゆる演算は、算術演算器、そこにデータを供給するメモリ帯域幅、あるいは Python から演算を発行する際の固定的なホスト側のコストという 3 つの資源のいずれかによって制限される。演算が実行する FLOPs と移動させるバイト数という机上で数えられる 2 つの量によって、最初の 2 つのどちらが制約になるかが決まる。そこに呼び出しを発行するためのホスト時間 h という 1 つの実測定数を加えれば、3 つ目の制約も予測できる。本章全体を通じたアプローチは、FLOPs とバイト数を数え、実行時間を予測し、実測し、その差を説明することである。

```
import os
import re
import time
# Detach PyTorch's profiler from the GPU when a profile ends; without this,
# every kernel launch after the first profile stays about 30% slower
os.environ['TEARDOWN_CUPTI'] = '1'
```

PyTorch では、fp32 の行列積で tf32 テンソルコアが使われるよう、float32 行列積の精度を `'high'` に設定しておく。

```
# Run fp32 matrix products on the tf32 tensor cores (PyTorch's default is
# full fp32). The bf16 products below are unaffected.
torch.set_float32_matmul_precision('high')
```

13.1.1 演算密度とルーフライン

実際に時間を計測する前でも、演算の形状からどの資源がそれを制限するかを予測できる。この考え方は 11.3.2.1 節でも簡単に用い、プレフィルが自己回帰デコーディングよりもはるかに高速にトークンを処理できる理由を説明したが、ここではそれを本格的に展開する。

13.1.1.1 FLOPs とバイト数のカウント

あらゆるテンソル演算は GPU に 2 つの要求を課す。1 つは**浮動小数点演算**（floating-point operations、FLOPs。加算または乗算のそれぞれを 1 回と数える）で測られる算術演算の実行であり、もう 1 つは GPU のメインメモリとオンチッププロセッサの間でのバイト単位のデータ移動である。

GPU にはそのそれぞれに対して個別のバジェット（上限）がある。RTX 4090 のメインメモリが供給できるのは最大で $\beta = 1.008 \times 10^{12}$ bytes/s（約 1 TB/s）である。ピーク算術演算速度 P は数値形式と演算器に依存し、テンソルコア上の bf16 行列積は $P = 165 \times 10^{12}$ FLOP/s に達する一方、tf32 行列積と標準的な fp32 算術演算は定格の 2.52 GHz ブーストクロックにおいてその

半分の速度（82.6 TFLOP/s）で動作する。定格クロックは公称値であり、実際のカードは温度や消費電力に応じて数パーセント速く、あるいは遅く動作する。したがって、実測速度が P の前後約 5% 以内に収まっていれば、ルーフに達しているものとみなす。

どちらのバジェットが先に尽きるかは、両者の比である**演算密度**（arithmetic intensity）によって決まる。

$$I \;=\; \frac{\text{FLOPs performed}}{\text{bytes moved}}.$$

行列積 $\mathbf{XW}$（ただし $\mathbf{X} \in \mathbb{R}^{B\times D}$、$\mathbf{W} \in \mathbb{R}^{D\times F}$）、すなわち B 行のバッチに適用される線形層を考えよう。この演算は $2BDF$ FLOPs（内積の各項につき乗算 1 回と加算 1 回）を実行し、最良の場合でも $\mathbf{X}$ と $\mathbf{W}$ を 1 回ずつ読み出し、出力を 1 回書き込む（合計 $BD + DF + BF$ 個の数値）。数値 1 個あたり b バイトとすると、その演算密度は次のようになる。

$$\begin{aligned} \frac{2BDF}{b\,(BD + DF + BF)} \;&=\; \frac{2}{b}\left(\frac{1}{F} + \frac{1}{B} + \frac{1}{D}\right)^{-1} \\ &\approx\; \frac{2B}{b} \quad \text{for } B \ll D, F. \end{aligned} \tag{13.1}$$

中辺の式から、(B, D, F) のうち最小のものが演算密度を支配することがわかる。バッチサイズ B が層の次元よりはるかに小さいとき、演算密度は B に比例して増加する。メモリから読み出された各重みが、B 個すべての行にわたって再利用されるからである。

幅 $D = F = 2048$、bf16（$b = 2$）の 1B パラメータの GPT では、1,024 トークンの系列 8 本からなる訓練ミニバッチは $B = 8192$ 行となり、Equation 13.1 より $I \approx 910$ FLOP/byte となる（幅 D によって制限される）。単一系列の自己回帰生成（autoregressive generation）では $B = 1$ であり、$I \approx 1$ FLOP/byte となる。同じ層であっても、訓練とデコーディングの間で演算密度はほぼ 3 桁にわたって変化する。さらにその下限に位置するのが $y_i = x_i + 1$ のような要素ごとの演算であり、テンソルのサイズにかかわらず、移動する $2b$ バイトあたり 1 FLOP しか実行しない（bf16 では $I = 0.25$ FLOP/byte）。

13.1.1.2 ルーフラインモデル

ルーフラインモデル (Williams et al. 2009) は、演算密度をスループットの上界へと変換する。すなわち、いかなる演算も計算ルーフ P より速くは動作せず、メモリが速度 $I\beta$ でデータを供給するペースより速くも動作しない。

$$\text{performance} \;\le\; \min(P,\; I\beta). \tag{13.2}$$

両対数軸上では、この上界は傾きをもつ帯域幅の直線と水平な計算ルーフが**リッジポイント** $I^* = P/\beta$ で交わる形になる（Figure 13.1）。I^* 未満では演算は**帯域幅律速**（bandwidth-bound）となる。メモリが算術演算器に十分なデータを供給できないため、すでにオンチップにあるデータに対する追加の算術演算は実質的に無償で行える。I^* を超えると演算は**計算律速**（compute-bound）となる。算術演算器が飽和しているため、データ移動のバイト数を減らしても高速化しない。本章全体を通じて、β はメインメモリ（DRAM）の帯域幅を表す。72 MB のオンチップ L2 キャッシュに収まるほど小さいオペランドは、このルーフを超えることがある。

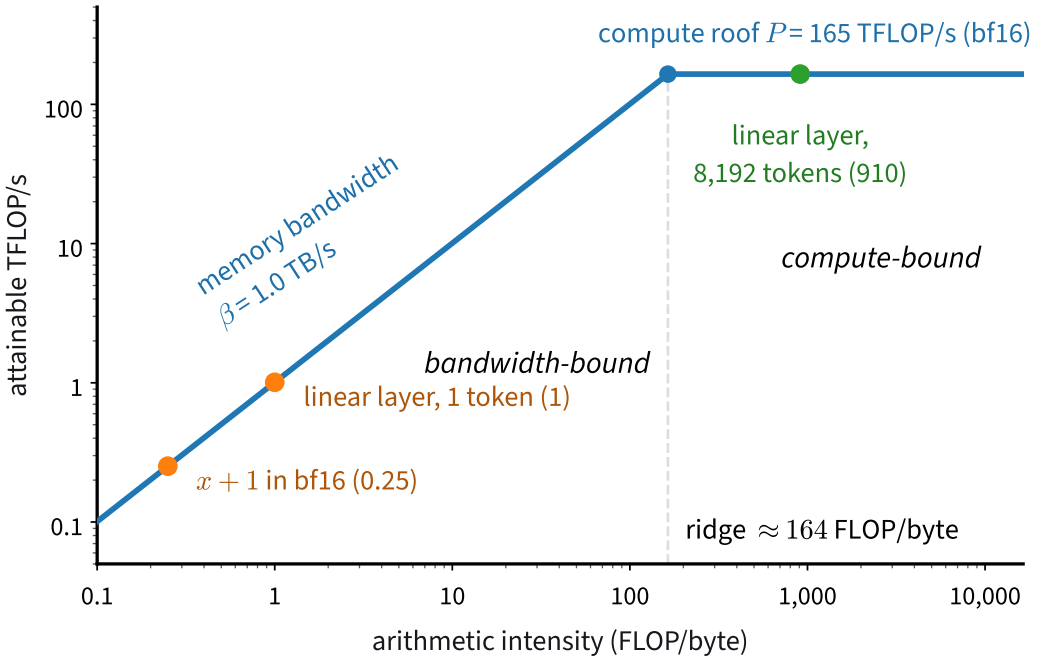


Figure 13.1: RTX 4090 のルーフライン。達成可能なスループットは、計算ルーフ P と、演算密度にメモリ帯域幅 β を掛けた値のうち小さい方となる。図中の点は本文の例（bf16 での $x + 1$、1 トークンに適用した幅 2048 の線形層、および 8,192 トークンのミニバッチに適用した同じ層）を示す。リッジポイントは 164 FLOP/byte 付近にある。

13.1.1.3 リッジポイント

RTX 4090 上の bf16 テンソルコア行列積では、リッジポイントは $I^* = 165.2/1.008 \approx 164$ FLOP/byte であり、tf32 行列積（$P = 82.6$ TFLOP/s）では $I^* \approx 82$ FLOP/byte である。現代のデータセンター向け GPU では、HBM の帯域幅よりもテンソルコアの FLOP/s の方が速く向上してきたため、リッジポイントはさらに高い（280〜300 FLOP/byte 前後）。これらのしきい値を超えるのは大規模な行列積だけであり、要素ごとの演算、正規化、単一トークンのデコーディングはいずれもリッジポイントを大きく下回る。

正方行列同士の bf16 行列積（$B = D = F = n$）では $I = n/3$ となるため、ルーフラインモデルからは $n \gtrsim 500$ を満たすすべての行列積が計算律速になると予測される。この予測を検証するには、GPU の実行時間を正確に計測する必要がある。

13.1.2 GPU コードの時間計測

GPU へのディスパッチは非同期であるため（5.7.2.1 節）、Python はカーネルをキューに入れると即座に次の処理へ進む。同期をとらずにカーネル呼び出しの前後で実経過時間を測ると、計測されるのは Python が処理をエンキューするのに要した時間であり、GPU がそれを実行するのに要した時間ではない。正確な時間計測では、`d2l.Benchmark` が行うように、まずデバイスをウォームアップし、計測の開始前と停止前に同期をとり、多数回の反復にわたって平均をとる。

13.1.2.1 同期ポイント

Python が GPU の完了を待ってブロックするのは、デバイス上にある値を要求したときだけである。たとえば、テンソルを Python のスカラーや NumPy 配列に変換するとき、それを表示するとき、その値で条件分岐するとき（`if loss < 0.1:`）、あるいは出力の形状がテンソルの値に依存する演算（ブールマスキング `x[x > 0]` など）を実行するときである。

ホストが 1 ステップを発行するのに時間 h を要し、GPU がそれを実行するのに t_{dev} を要するとしよう。同期のないループではホストディスパッチとデバイス実行がオーバーラップするのに対し、毎ステップでスカラーを読み出すと両者が直列化され、さらにデバイスからホストへの往復レイテンシ σ が加わる。

$$t_{\text{once}} \approx \max(h, t_{\text{dev}}), \qquad t_{\text{every}} \lesssim h + t_{\text{dev}} + \sigma.$$

1024×1024 行列の自身との積を計算し、その総和を GPU 上で累積する 2 つの 1,000 ステップのループを比較しよう。一方は累積和を毎ステップ Python の `float` に変換し、もう一方は最後に 1 回だけ読み出す。

```
x = torch.ones(1024, 1024, device=d2l.try_gpu())
s = torch.zeros((), device=x.device)
s += (x @ x).sum()
float(s)                          # Warm up: load the kernels

torch.cuda.synchronize()
t0 = time.perf_counter()
for _ in range(1000):
    s += (x @ x).sum()
    total = float(s)              # Reads the value: a barrier every step
read_every = time.perf_counter() - t0
```

```
    torch.cuda.synchronize()
    t0 = time.perf_counter()
    for _ in range(1000):
        s += (x @ x).sum()            # Stays on the device: no barrier
    total = float(s)                  # One read at the end
    read_once = time.perf_counter() - t0
    print(f'read every step: {read_every:.3f} s   read once: {read_once:.3f} s')
```

```
read every step: 0.063 s   read once: 0.040 s
```

ステップごとの読み出しを行わなければ、1 ステップあたり 3 回の呼び出しに約 40 µs のホストディスパッチ時間（tf32 によるデバイス処理の約 26 µs より長い）がかかるため、GPU はホストの後を追って動作する。毎ステップ `float(s)` を読み出すとホストとデバイスが交互に動作せざるを得なくなり、ループは約 1.6 倍遅くなる。

したがって、訓練ループでは評価指標をデバイス上で累積し、ログ出力の間隔でのみ読み出すべきである。

13.1.2.2 ベンチマークヘルパ

これ以降、演算時間の計測には `d2l.Benchmark` を用いる。このヘルパはまずウォームアップ呼び出しを実行してコンパイルやカーネル読み込みのオーバーヘッドを吸収し、続いて 2 つのデバイスバリアの間で `repeats` 回の呼び出しを計測する。最初の確認として、4096×4096 の bf16 行列積は演算密度が $I \approx 1365$ FLOP/byte とリッジポイントを大きく上回るため、ルーフラインモデルからは $2 \cdot 4096^3 / P \approx 0.83$ ms と予測される。

```
    a = torch.randn(4096, 4096, device=d2l.try_gpu(), dtype=torch.bfloat16)
    d2l.Benchmark(lambda: torch.mm(a, a), desc='4096 x 4096 product')
```

```
4096 x 4096 product: 0.90 ms/call
```

実測時間は、予測値である 0.83 ms の約 10% 以内に収まっている。

13.1.3 3 つのレジーム

GPU プログラムの実経過時間は、計算律速なカーネル、帯域幅律速なカーネル、そしてデバイスがホストを待つアイドル時間の隙間に分けられる (He 2022)。3 つの実験によって各レジームを切り分けよう。

13.1.3.1 帯域幅ルーフと計算ルーフ

まず、活性化値からなる B 行の行列に、$d \times d$（$d = 16{,}384$、512 MiB）の bf16 重み行列を掛ける。Equation 13.1 より、その演算密度はおよそ B であり、$B = 164$ 付近でリッジポイントを横切る。リッジポイント未満では、どの呼び出しでも B にかかわらず 512 MiB の重みをストリーミングしなければならず（1.008 TB/s で約 0.53 ms）、リッジポイントを超えると $2Bd^2$ FLOPs が実行時間を支配して B に比例して増加する（$B = 256$ で 0.83 ms、$B = 1024$ で 3.32 ms）。

```
    peak_tflops = 165.2    # RTX 4090: dense bf16 on the tensor cores
    bandwidth_tbs = 1.008  # RTX 4090: GDDR6X memory
    print(f'{torch.cuda.get_device_name()}: ridge point '
```

```
      f'{peak_tflops / bandwidth_tbs:.0f} FLOP/byte')
d, b = 16384, 2  # Width and bytes per bf16 number
W = torch.randn(d, d, dtype=torch.bfloat16, device=d2l.try_gpu())  # 0.54 GB
for B in (1, 4, 16, 64, 128, 256, 1024):
    X = torch.randn(B, d, dtype=torch.bfloat16, device=W.device)
    t = d2l.Benchmark(lambda: X @ W).time
    flops, nbytes = 2 * B * d * d, b * (B * d + d * d + B * d)
    t_mem, t_comp = nbytes / bandwidth_tbs / 1e12, flops / peak_tflops / 1e12
    roof = 'memory' if t_mem > t_comp else 'compute'
    print(f'B = {B:4d}: bound {1e3 * max(t_mem, t_comp):4.2f} ms ({roof:7s}), '
          f'measured {1e3 * t:4.2f} ms, {flops / t / 1e12:5.1f} TFLOP/s')
```

```
NVIDIA GeForce RTX 4090: ridge point 164 FLOP/byte
B =    1: bound 0.53 ms (memory ), measured 0.57 ms,   0.9 TFLOP/s
B =    4: bound 0.53 ms (memory ), measured 0.57 ms,   3.7 TFLOP/s
B =   16: bound 0.53 ms (memory ), measured 0.57 ms,  15.0 TFLOP/s
B =   64: bound 0.54 ms (memory ), measured 0.58 ms,  58.9 TFLOP/s
B =  128: bound 0.54 ms (memory ), measured 0.64 ms, 106.9 TFLOP/s
B =  256: bound 0.83 ms (compute), measured 0.90 ms, 152.2 TFLOP/s
B = 1024: bound 3.33 ms (compute), measured 3.34 ms, 164.6 TFLOP/s
```

1 つの重み行列で両方のルーフに到達している。$B = 1$ から $B = 128$ までは算術演算量が 128× に増えるにもかかわらず実行時間は 0.6 ms 付近でほぼ一定にとどまり、$B = 256$ 以上では計算ルーフ付近で B に比例して増加する。1,024 トークンを一度にサービングすると、1 トークンをサービングする場合の 1,024× の算術演算を、6× 未満の時間で実行できる。

13.1.3.2 要素ごとの演算

次に、リッジポイントを大きく下回る領域にとどまったまま、8192×8192 の fp32 テンソル（256 MiB）に対して要素あたりの算術演算を変えてみよう。試すのは加算、乗算、正弦（sin）、シグモイドである。いずれも 256 MiB を読み出して 256 MiB を書き込む（合計 512 MiB、1.008 TB/s で 0.53 ms）。

```
x = torch.randn(8192, 8192, device=d2l.try_gpu())  # 256 MiB in fp32
for desc, f in [('add', lambda x: x + 1), ('mul', lambda x: x * 1.5),
                ('sin', torch.sin), ('sigmoid', torch.sigmoid)]:
    print(d2l.Benchmark(lambda: f(x), desc=desc))
```

```
add: 0.59 ms/call
mul: 0.59 ms/call
sin: 0.59 ms/call
sigmoid: 0.58 ms/call
```

4 つすべてが約 0.6 ms で完了し、約 0.9 TB/s の実効メモリ帯域幅を達成する。メモリのデータ移動量が実行時間を決めるため、超越関数を含む演算（`sin`、`sigmoid`）であっても `x + 1` より余分な時間はかからない。

即時実行（eager execution）では、要素ごとの演算を連ねるとこのメモリ通信量が積み重なる。以下に示す GELU の tanh 近似（`gelu_ish`）は、8 個の要素ごとの演算を連鎖させている。6 個の単項演算がそれぞれ 1 個のテンソルを読み書きし（計 12 個のテンソル）、2 個の二項演算（`x + 0.04 * x**3` と最後の `0.5 * x` との乗算）がそれぞれ 2 個のテンソルを読み出して 1 個を書き込むため（計 6 個のテンソル）、合計で 256 MiB のテンソル 18 個分、すなわち単一の加算の 9× のバ

イト数を移動させることになる。

```
def gelu_ish(x):
    """The tanh approximation of GELU with rounded constants: a chain of
    eight elementwise operations."""
    return 0.5 * x * (1 + torch.tanh(0.8 * (x + 0.04 * x**3)))

print(d2l.Benchmark(lambda: x + 1, desc='1 op'))
print(d2l.Benchmark(lambda: gelu_ish(x), desc='chain'))  # Eager: 8 kernels
```

```
1 op: 0.59 ms/call
chain: 5.25 ms/call
```

予測どおり、融合されていない演算連鎖には `x + 1` の $8\times$ から $9\times$ の時間がかかる。**カーネル融合**（kernel fusion、13.3 節）は、この連鎖を `x` を 1 回だけ読み出して出力を 1 回だけ書き込む単一のカーネルへと統合する。

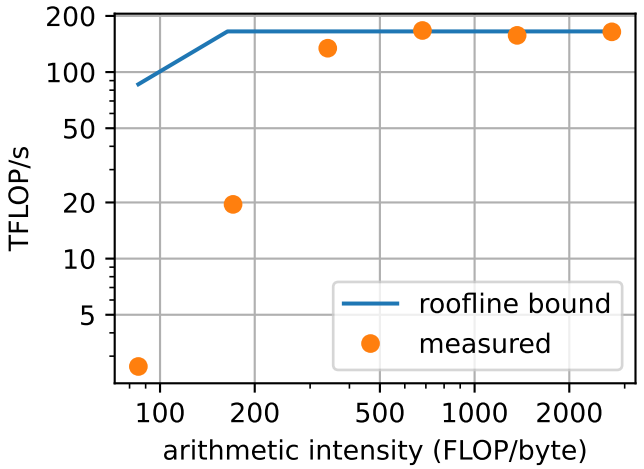
13.1.3.3 サイズ別の行列積

正方行列の bf16 行列積は、$I = n/3$ が $n = 500$ 付近で 164 を超えた途端に計算ルーフへ到達するのだろうか。n を 256 から 8,192 までスイープして確かめてみよう。

```
sizes = [256, 512, 1024, 2048, 4096, 8192]
times = []
for n in sizes:
    a = torch.randn(n, n, device=d2l.try_gpu(), dtype=torch.bfloat16)
    times.append(d2l.Benchmark(lambda: torch.mm(a, a), repeats=100).time)
```

達成された TFLOP/s を演算密度 $n/3$ に対して、ルーフライン上界 Equation 13.2 と並べてプロットする。

```
measured = [2 * n**3 / t / 1e12 for n, t in zip(sizes, times)]
intensity = [n / 3 for n in sizes]  # FLOP/byte of a square bf16 product
ridge = peak_tflops / bandwidth_tbs
roof_x = [intensity[0], ridge, intensity[-1]]
roof_y = [min(peak_tflops, bandwidth_tbs * i) for i in roof_x]
d2l.plot([roof_x, intensity], [roof_y, measured],
         'arithmetic intensity (FLOP/byte)', 'TFLOP/s',
         legend=['roofline bound', 'measured'],
         xscale='log', yscale='log', fmts=('-', 'o'))
```



呼び出し 1 回あたりの正確なレイテンシ `t`（マイクロ秒単位）とスループット `tf` を行列サイズ `n` ごとに出力すると、小さい n における一定時間の下限が明確になる。

```
for n, t, tf in zip(sizes, times, measured):
    print(f'n = {n:4d}: {1e6 * t:7.1f} us per call, {tf:5.1f} TFLOP/s')
```

```
n =  256:    12.7 us per call,   2.6 TFLOP/s
n =  512:    13.7 us per call,  19.5 TFLOP/s
n = 1024:    16.0 us per call, 134.3 TFLOP/s
n = 2048:   102.8 us per call, 167.1 TFLOP/s
n = 4096:   873.6 us per call, 157.3 TFLOP/s
n = 8192:  6690.3 us per call, 164.3 TFLOP/s
```

$n = 4096$ と 8192 では、行列積はクロック変動幅である 5% の範囲内で 165 TFLOP/s のルーフに達している。しかし小さいサイズでは、測定点はルーフラインを大きく下回り、両対数傾き 3 で上昇している。これは演算量が n^3 でスケールする一方で、呼び出し 1 回あたりの時間が一定であることの特徴である。$n = 256$ から 1024 にかけて演算量は $64\times$ に増えるにもかかわらず、呼び出しあたりの時間は $2\times$ 未満しか変化しない。

この一定時間がホストとデバイスのどちらで費やされているかを確かめるため、256×256 の行列積と同じ形状の `x + 1` の実行時間を計測し、さらにプロファイラ用ヘルパ `d2l.kernel_times`（後続の 13.1.4 節で構築する）を用いて呼び出し 1 回あたりに GPU 自体が稼働している時間を計測する。

```
a = torch.randn(256, 256, device=d2l.try_gpu(), dtype=torch.bfloat16)
for desc, f in [('256 x 256 product', lambda: torch.mm(a, a)),
                ('x + 1 on 256 x 256', lambda: a + 1)]:
    wall = d2l.Benchmark(f, repeats=100).time
    busy = sum(d2l.kernel_times(f, steps=100).values()) / 1e3  # Seconds
    print(f'{desc:18s}: {1e6 * wall:5.1f} us per call, '
          f'device busy {1e6 * busy:4.1f} us')
```

```
256 x 256 product :  21.0 us per call, device busy  3.2 us
x + 1 on 256 x 256:  10.9 us per call, device busy  1.0 us
```

いずれの場合も GPU が実際に稼働しているのはわずか数マイクロ秒であり、残りはホストディ

スパッチ時間 h（Python、フレームワーク、ドライバ）である。256×256 の行列積では、演算の中で最も高コストな部分はそれを要求する Python の 1 行なのである。呼び出しが立て続けにキューへ入れられると、ホストとデバイスはオーバーラップし、各呼び出しの所要時間は $\max(h, 2n^3/P)$ となる。小さい n ではホストが次の呼び出しを発行する前にデバイスが処理を終えてしまうため、プログラムは**オーバーヘッド律速**（overhead-bound）のレジームに入る。計算時間がディスパッチ時間を上回る屈曲点は $n \approx (hP/2)^{1/3}$ 付近にある。

$h \approx 13\ \mu s$ の場合、屈曲点は $n = 1{,}000$ 付近にある。

これら 3 つの下限をすべてまとめると、ディスパッチされる単一演算に対する基本的な時間下界が得られる。

$$t \gtrsim \max\left(h,\ \frac{\text{FLOPs}}{P},\ \frac{\text{bytes}}{\beta}\right). \tag{13.3}$$

$h = 10\ \mu s$ とすると、演算量が $hP \approx 1.6 \times 10^9$ FLOPs 未満かつデータ移動量が $h\beta \approx 10$ MB 未満のあらゆる演算は、その起動時間だけで決まる。さらに、屈曲点 $n \approx (hP/2)^{1/3}$ は P とともに大きくなるため、より高速な GPU ほどオーバーヘッド律速を避けるためにより大きな演算を必要とする。

13.1.3.4 診断と対策

訓練ステップ全体では 3 つのレジームが混在し、最も大きな割合を占めるものがそのステップのレジームを決める（Figure 13.2）。Table 13.1 に、各レジームの症状と標準的な対策をまとめる。

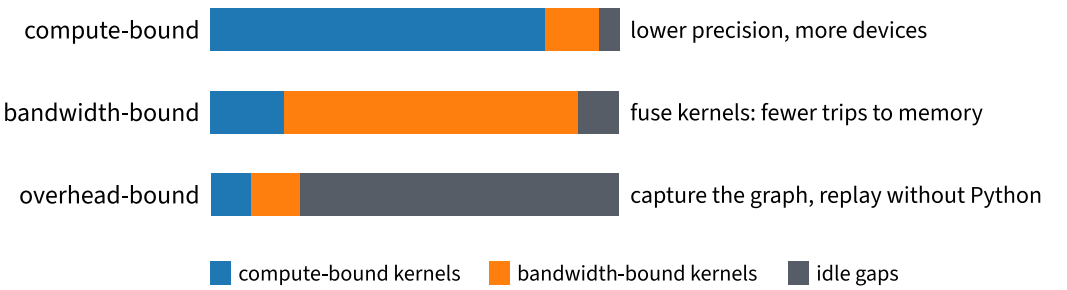


Figure 13.2: 3 つのレジームの模式図。1 ステップの時間が計算律速なカーネル、帯域幅律速なカーネル、アイドル時間の隙間のどこに費やされているかによって適用すべき対策が決まる。各バーには代表的な対策を 1 つずつ示してある。

Table 13.1: 3 つのレジームの症状と標準的な対策。

レジーム	症状	標準的な対策
計算律速	デバイスが稼働中、演算密度がリッジポイント超、時間が FLOPs に応じて増加	算術演算量の削減、より高速なテンソルコアを備えた数値形式、デバイス数の追加

レジーム	症状	標準的な対策
帯域幅律速	デバイスが稼働中、演算密度がリッジポイント未満、時間が移動バイト数に応じて増加	行列積の演算密度を高めるバッチ拡大、より狭い数値形式、カーネル融合、保存の代わりの再計算、チップ上でデータを再利用するアルゴリズム
オーバーヘッド律速	カーネル間でデバイスがアイドル状態、呼び出しあたりの時間がサイズに非依存	例あたりの呼び出し回数を減らすバッチ拡大、プログラム全体のキャプチャと Python を介さないリプレイ

13.1.4 訓練ステップのプロファイリング

プロファイラはすべての GPU カーネルの所要時間を記録するため、デバイスの稼働時間を実経過時間と比較したり、デバイス時間をカーネルの種類ごとに内訳表示したりできる。プロファイリングを行う前に、1,024 個の例からなるミニバッチに対する MLP（$4096 \to 4096 \to 4096 \to 10$）の fp32（tf32 行列積を使用、$P = 82.6$ TFLOP/s）における SGD 1 ステップの実行時間を予測してみよう。

13.1.4.1 ステップ時間の予測

2 つの 4096×4096 層はそれぞれ、順伝播の行列積 1 回あたり $2 \cdot 1024 \cdot 4096^2 \approx 3.44 \times 10^{10}$ FLOPs を実行する。1 回の訓練ステップではこのような行列積が 5 回実行される（順伝播で 2 回、重み勾配で 2 回、2 番目の隠れ層に対する入力勾配で 1 回）。$I \approx 340$ FLOP/byte（tf32 のリッジポイントである 82 を上回る）であるため、各行列積には $3.44 \times 10^{10} / (82.6 \times 10^{12}) \approx 0.42$ ms かかる。SGD の更新は 3,360 万個の fp32 重みとその勾配をそれぞれ読み出して新しい重みを書き込むため（$12 \times 33.6 \times 10^6 \approx 0.40$ GB）、1 TB/s で約 0.40 ms かかる。小さな最終層と ReLU を無視すると、このステップの所要時間は次のようになるはずである。

$$5 \times 0.42 \text{ ms} + 0.40 \text{ ms} \;\approx\; 2.5 \text{ ms}.$$

```
net = nn.Sequential(nn.Linear(4096, 4096), nn.ReLU(),
                    nn.Linear(4096, 4096), nn.ReLU(),
                    nn.Linear(4096, 10)).to(d2l.try_gpu())
X = torch.randn(1024, 4096, device=d2l.try_gpu())
opt = torch.optim.SGD(net.parameters(), lr=0.01)

def step():
    opt.zero_grad()
    net(X).mean().backward()
    opt.step()

wall = d2l.Benchmark(step).time
```

`kernel_times` はフレームワークのプロファイラの下で関数を `steps` 回実行し、起動されたすべてのカーネルについて呼び出し 1 回あたりのデバイス上の平均ミリ秒を返す。

```
def kernel_times(f, steps=10):
    """Device milliseconds per call of f, for every kernel that f runs."""
    with torch.profiler.profile(
            activities=[torch.profiler.ProfilerActivity.CUDA],
            acc_events=True) as prof:
        for _ in range(steps):
            f()
        torch.cuda.synchronize()
    return {e.key: e.device_time_total / 1e3 / steps
            for e in prof.key_averages()
            if e.device_type == torch.autograd.DeviceType.CUDA}
```

フレームワークのプロファイラは、https://ui.perfetto.dev で閲覧可能なタイムライントレースをエクスポートすることもできる。

prof.export_chrome_trace は、kernel_times 内のプロファイルのトレースを書き出す。

13.1.4.2 プロファイルの読み方

show_profile はデバイスの稼働時間をステップの実経過時間と比較し、行列積カーネルが占める割合を計算して、上位のカーネルを一覧表示する。

```
PRODUCT_KERNELS = 'gemm|xmma|cutlass|matmul|sdpa'
# Kernel names that mark a matrix product: cuBLAS and CUTLASS kernels (gemm,
# xmma, cutlass, which also names PyTorch's fused attention), XLA's fused
# products (gemm_fusion_dot), and matmul or sdpa kernels

def show_profile(kernels, step_ms, rows=6):
    """Busy device time against the wall-clock time step_ms of a step, the
    share of the matrix products, and the kernels with the most time."""
    busy = sum(kernels.values())
    products = sum(ms for name, ms in kernels.items()
                   if re.search(PRODUCT_KERNELS, name, re.I))
    print(f'wall clock {step_ms:.2f} ms per step, device busy {busy:.2f} ms '
          f'({busy / step_ms:.0%}), matrix products {products / busy:.0%}')
    for name, ms in sorted(kernels.items(), key=lambda kv: -kv[1])[:rows]:
        name = re.sub(r'^void |\(anonymous namespace\)::|\w+::|\(.*', '', name)
        print(f'{ms:6.2f} ms {ms / busy:4.0%}  {name[:52]}')
```

ここで show_profile を 3 層 MLP のステップのカーネル時間計測結果（kernel_times(step)）に適用し、デバイスの稼働時間を実測された実経過時間 1000 * wall と比較する。

```
show_profile(kernel_times(step), 1000 * wall)
```

```
wall clock 2.67 ms per step, device busy 2.65 ms (99%), matrix products 80%
  0.85 ms  32%  Kernel2<cutlass_80_tensorop_s1688gemm_256x128_16x3_n
  0.82 ms  31%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  0.42 ms  16%  multi_tensor_apply_kernel<TensorListMetadata<2>, Bin
  0.40 ms  15%  Kernel2<cutlass_80_tensorop_s1688gemm_128x256_16x3_n
  0.06 ms   2%  vectorized_elementwise_kernel<4, BinaryFunctor<float
  0.02 ms   1%  vectorized_elementwise_kernel<4, launch_clamp_scalar
```

実測されたステップ時間は予測値 2.5 ms の約 10% 以内に収まっており、ステップのほぼ全時間にわたって GPU が稼働している。行列積（**gemm** カーネル）がデバイス時間の 3/4 以上を占めている。

3 つの `gemm` 行は 5 つの大きな行列積を集約したものであり、`multi_tensor_apply_kernel` は予測どおり 0.4 ms かかっているバッチ化された SGD 更新である。

なお、融合されていないオプティマイザの更新には、バッチサイズに依存しない一定の時間がかかる。バッチサイズ 128 では、5 つの行列積がそれぞれ約 0.07 ms まで縮小するため（帯域幅律速に転じる）、0.4 ms の更新処理の方が 5 つの行列積の合計よりも大きくなる。

13.1.5 まとめ

GPU 演算の所要時間は $t \gtrsim \max(h,\ \text{FLOPs}/P,\ \text{bytes}/\beta)$ によって下から抑えられる。ここで、h はホストディスパッチのオーバーヘッド、P はピーク算術演算スループット、β はメモリ帯域幅である。演算密度 $I = \text{FLOPs}/\text{bytes}$ は後半の 2 つを比較するものであり、リッジポイント $I^* = P/\beta$（RTX 4090 の bf16 では約 164 FLOP/byte）未満では演算は帯域幅律速となり、それを超えると計算律速となる。線形層では、演算密度はバッチサイズとともにスケールし、単一トークンのデコーディング時の約 1 FLOP/byte から訓練時の数百 FLOP/byte まで変化する。正確なベンチマークにはループ外でのデバイス同期とウォームアップ実行（`d2l.Benchmark`）が不可欠であり、`d2l.kernel_times` と `d2l.show_profile` を用いれば机上の予測をカーネルごとに検証できる。

13.1.6 演習問題

1. **最小の次元。** Equation 13.1 における積 $\mathbf{XW}$ の演算密度の厳密な式（$B \ll D, F$ の近似を行う前のもの）は、あらゆる形状に対して成り立つ。$m = \min(B, D, F)$ とおく。
   1. 演算密度が $2m/(3b)$ と $2m/b$ の間にあることを示せ。ここから、リッジポイントが 164 FLOP/byte の GPU 上で bf16 を用いる場合、164 未満の次元をもつすべての行列積はルーフラインモデルにおいて帯域幅律速となることを結論づけよ。
   2. この評価式を 3 つの bf16 行列積に適用せよ。すなわち、13.1.4 節の MLP の最終層（$4096 \to 10$、サイズ 4,096 のバッチ）、1 つのヘッドのアテンションスコア $\mathbf{QK}^\top$（ただし $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{T \times 64}$）、および 64 チャネルから 64 チャネルへの 1×1 畳み込み（6.4.3 節、画像 128 枚のバッチで各画像は 32×32 ピクセル）である。それぞれがリッジポイントに到達するためには、どの次元が大きくなる必要があるか。
   3. マルチヘッドアテンション層は、すべてのヘッドのスコアを 1 つのバッチ化されたカーネルで計算する。ヘッドをバッチ化するとスコアの演算密度は変化するか。
2. [code] **バッチサイズとリッジポイント。** bf16 における積 $\mathbf{XW}$（ただし $\mathbf{X} \in \mathbb{R}^{B \times 8192}$、$\mathbf{W} \in \mathbb{R}^{8192 \times 8192}$）を考えよ。
   1. Equation 13.1 の厳密な式を用いて、この行列積が手元の GPU のリッジポイントに到達するバッチサイズを求め、小さい B の近似と比較せよ。
   2. Equation 13.3 から、同時にデコーディングする系列数が $B = 1$ と $B = 64$ の場合についてこの行列積の所要時間を予測し、両方を `d2l.Benchmark` で実測したうえで、ステップあたりの追加時間がほとんどかからずに系列数を増やせるのはどのバッチサイズまでか求めよ。
3. [code] **隠れたバリア。** 以下の各行が訓練ループのステップごとに 1 回実行されるとする。ただし、`x` と `g` は GPU テンソル、`c` は Python の浮動小数点数である。対象とする行は `x[x > 0].sum()`、`print(x.shape)`、`x.argmax()`、`if g.norm() > c: g *= c / g.norm()`、および `x * torch.tensor(0.5, device=x.device)` である。
   1. どの行がホストにデバイスの完了を待たせるかを判定し、それぞれの理由を述べよ。

2. 同期を発生させるすべての行を書き換え、ホストバリアなしに同じ結果を計算するようにせよ。
3. `torch.cuda.set_sync_debug_mode('warn')` を用いて解答を確認せよ。

4. **計測時間からの診断。** RTX 4090（$P = 165$ TFLOP/s、$\beta = 1.008$ TB/s）上の 4 つの bf16 演算について、`d2l.Benchmark` が呼び出し 1 回あたり以下の時間を報告したとする。演算 A は 2 つの 8192×8192 行列の積を 7.4 ms で計算し、演算 B は 8192×8192 行列に行ごとのソフトマックスを 0.30 ms で適用し（要素あたり約 5 FLOPs、読み出し 1 回と書き込み 1 回）、演算 C は 2 つの 256×256 行列の積を 10 µs で計算し、演算 D は 2 つの 1536×1536 行列の積を 0.10 ms で計算する。
   1. 各演算について、演算密度ならびに達成している P と β に対する割合を計算せよ。リッジポイントに対する演算密度の大小と、近づいているルーフのそれぞれによって各演算を分類せよ。2 つの分類が食い違うのはどの演算か。
   2. どちらのルーフにも近くない各演算について、ホストディスパッチレイテンシ h が原因である場合と、すべてのストリーミングマルチプロセッサを占有するにはスレッドブロックのタイル数が不足している場合とを区別し、プロファイラのタイムラインがそれぞれどのように見えるかを述べよ。
5. [code] **ディスパッチのパイプラインモデル。**●1 回の GPU 呼び出しを、ホストエンキュー時間 h、デバイス実行時間 t_{dev}、および最後のバリアレイテンシ σ という 3 つの所要時間でモデル化せよ。
   1. 立て続けにエンキューされた R 回の呼び出しの後に 1 回のバリアを置くと、所要時間が $R\max(h, t_{\text{dev}}) + \min(h, t_{\text{dev}}) + \sigma$ になることを示せ。$t_{\text{dev}} < h$ のとき、`d2l.Benchmark` は何を報告するか。
   2. 13.1.2 節のループについて、毎ステップ読み出すループの所要時間と最後に 1 回だけ読み出すループの所要時間の比を、h、t_{dev}、σ を用いて導出せよ。
   3. 行列の辺の長さ $256, 512, 1024, 2048, 4096$ に対して両方のループを実行して h、σ、t_{dev} を推定し、ステップあたりのデバイス時間がいくらを超えれば毎ステップのスカラー読み出しのコストが 10% 未満になるかを求めよ。
6. [code] **タイルとウェーブ。**●GPU は n_{SM} 個のストリーミングマルチプロセッサ（SM）から構成される（RTX 4090 では $n_{\text{SM}} = 128$）。行列積カーネルが $n \times n$ の出力を 128×128 のタイルに分割し、各 SM が一度に 1 つのタイルを速度 P/n_{SM} で計算し、タイルは最大 n_{SM} 個ずつのウェーブ単位で実行されると仮定せよ。
   1. このモデルで達成可能なピークスループットに対する割合を n と n_{SM} の関数として導出し、13.1.3.3 節でのスイープ実測値と比較せよ。
   2. $n = 2048, 2176, 2304, 2432, 2560$（128 の連続する倍数）についてピークに対する割合を予測・実測し、実測値が単純なウェーブモデルから乖離する箇所を説明せよ。
   3. $n = 512$ の行列積をより多くのワークアイテムに分割してすべての SM を参加させる方法を 2 つ挙げ、それぞれが追加するオーバーヘッドを述べよ。

13.2 ハードウェア

前節では、GPU をピーク演算速度 P とメモリ帯域幅 β という 2 つの定数で特徴づけ、1 バイトあたり I FLOPs を実行する演算の速度が高々 $\min(P, I\beta)$ に抑えられることを示した。RTX 4090 では、bf16 の行列積に対して $P = 165$ TFLOP/s、$\beta \approx 1$ TB/s である。つまり、このカードがメモリから 1 バイトを読み出す間に、165 回の浮動小数点演算を実行できる計算になる。これら 2 つの数値はどこから来るのだろうか。また、なぜ 1 バイトの移動には 1 FLOP の演算よりはるかに長い時間がかかるのだろうか。

1B パラメータの GPT を用いて、テキストを 1 トークンずつ生成する場合を考えよう（11.3 節）。bf16 では重みが 2 GB を占め、トークンを 1 つ生成するたびにそのすべてを読み出す必要がある。1 トークン分の算術演算（2×10^9 FLOPs）はテンソルコアで 12 µs で終わる一方、実効帯域幅

0.9 TB/s で 2 GB の重みを読み出すには 2.2 ms かかる。したがって、単一ストリームでのデコード速度は高々約 450 tokens/s にとどまり、演算ユニットは 99% 以上の時間を遊休状態で過ごすことになる。

この隔たりは 3 つの物理的事実によって説明できる。第 1 にデータはチップ間の境界を越えなければならず、第 2 にオフチップ配線はオンチップのトランジスタよりも微細化に伴うスケーリングが遅く、第 3 に 1 バイトのデータ移動はそれを用いた演算よりも何桁も多くのエネルギーを消費する。これらが相まって、リッジポイント P/β が数百に達し、ハードウェアの世代を追うごとに上昇してきた理由が説明される。本節では具体例として、本書のノートブックを実行しているワークステーション（RTX 4090 GPU 4 基搭載、2026 年後半時点）を用いる。なお、コンピュータアーキテクチャの詳細については Hennessy and Patterson (2011) が深く扱っており、ハードウェアの選定については 39.4 節で議論する。

13.2.1 メモリ階層

プログラムが扱うすべてのバイトは、容量と速度をトレードオフするメモリ階層（memory hierarchy）のどこかに置かれている（Figure 13.3）。オンダイのレジスタと SRAM キャッシュは約 0.1 GB を保持し、数十 TB/s の帯域幅を提供する。GPU のオフダイ DRAM、すなわちデータセンタ向けカードの**広帯域メモリ**（high-bandwidth memory、HBM）やコンシューマ向けカードの GDDR は、数十〜数百 GB を保持し、帯域幅は 1〜8 TB/s である。ホスト DRAM は CPU ソケットあたり約 0.5 TB/s で数百 GB を保持し、NVMe フラッシュは約 14 GB/s で数 TB を保持し、そのさらに外側にはネットワークストレージが位置する。

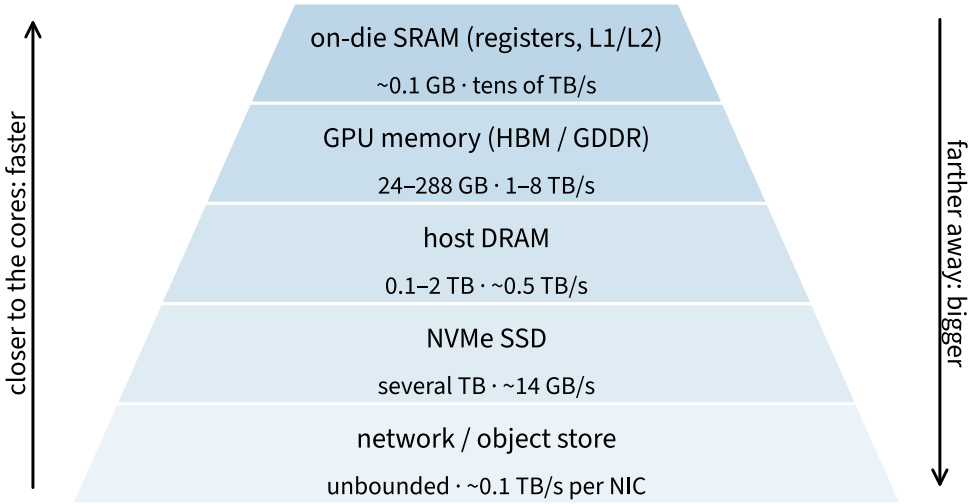


Figure 13.3: 2026 年後半時点のメモリ階層。下の階層へ行くほど保持できるバイト数は何桁も増えるが、毎秒転送できるバイト数は何桁も減る。

13.2.1.1 帯域幅とレイテンシ

Figure 13.4 は、訓練ジョブが利用するデータ経路を 14 GB/s（NVMe）から 8 TB/s（NVIDIA B200 の HBM）まで、ほぼ 3 桁にわたって並べたものである。どの段階の差も 1 つの法則で説明できる。すなわち、**帯域幅はデータ配線の本数と 1 本あたりのビットレートの積に等しい**。RTX 4090 は 384 本の配線を用いて各 21 Gb/s（合計約 1 TB/s）で GDDR6X メモリにアクセスするの

に対し、PCIe 4.0 ×16 スロットは片方向あたり 16 レーン、各 16 Gb/s（合計 32 GB/s）しかもたない。配線 1 本あたりの速度はほぼ同じだが、PCIe 側は配線数が 24 分の 1（24× の差）しかないのである。

配線の本数を決めるのは距離である。HBM スタックは GPU パッケージ内部でダイから数ミリメートルの位置に置かれ、数千本の短い配線で接続されている。これに対し、基板配線で扱えるのは数百本、スロットやケーブルでは数十レーンにとどまる。**ダイから 1 段階離れるごとに配線の本数が、ひいては帯域幅が約 1 桁ずつ減少する**。B200 は HBM を 8 TB/s で読み出し、他の GPU には NVLink 5 経由の 0.9 TB/s（各 200 Gb/s の約 36 レーンを使用）で接続し、ホストには PCIe 5.0 経由の 64 GB/s で接続する。

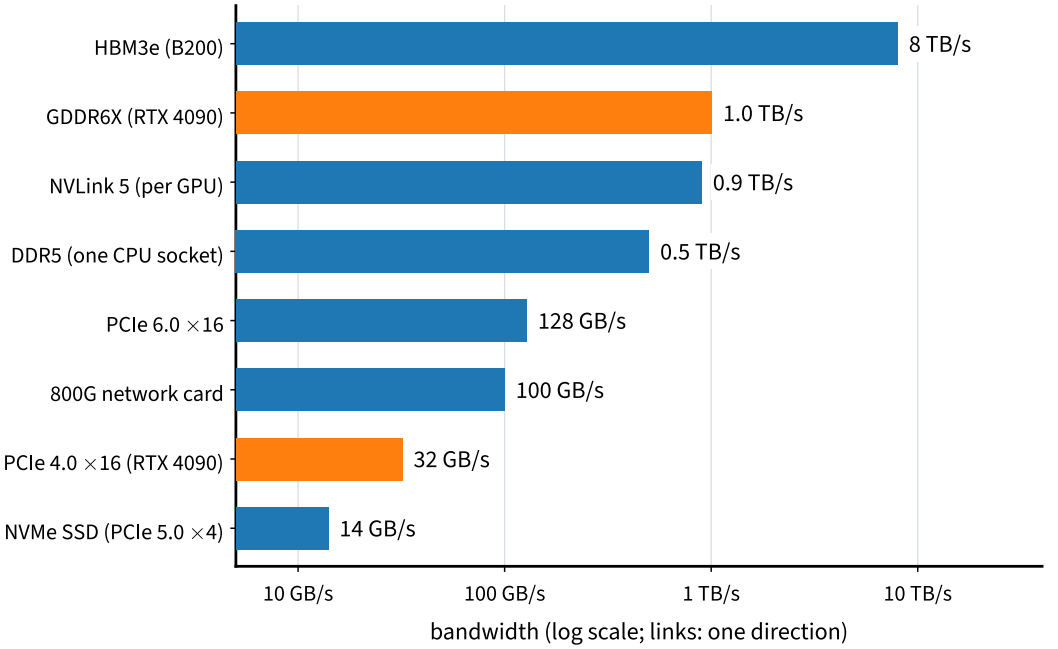


Figure 13.4: 訓練ジョブが使用するデータ経路の帯域幅（2026 年後半時点）。リンクは片方向の値である。本書のマシンのメモリとホストリンクをオレンジ色で示す。

帯域幅が定常状態での流量を支配するのに対し、**レイテンシ**は最初の 1 バイトが到着するまでの時間である（Figure 13.5、RTX 4090 のマイクロベンチマークは Luo et al. (2024) による）。Python からディスパッチされる 1 回の Eager 演算には 10〜100 µs のホスト時間がかかる（そのうちハードウェアのカーネル起動は 1〜5 µs である）。10 µs あれば RTX 4090 は 10 MB をストリーム転送できるため、小さな演算ではメモリや計算ではなくディスパッチのオーバーヘッドが支配的になる。

レイテンシと帯域幅は**アクセスパターン**を通じて相互作用する。DRAM が定格の帯域幅を発揮できるのは、連続したアドレスにアクセスする場合だけである。ここで 1 GiB の fp32 ベクトル `x[idx]` をギャザーする実験として、`idx` を恒等置換にした場合とランダム置換にした場合をそれぞれ試してみよう。GPU メモリは 32 バイトの**セクタ**単位でデータを転送するため、ランダムな 4 バイトの読み出しは毎回 32 バイトのセクタ全体を取得することになる一方、インデックスと出力のベクトルは連続的にストリーム転送される（全体として移動するバイト数は約 3× になる）。

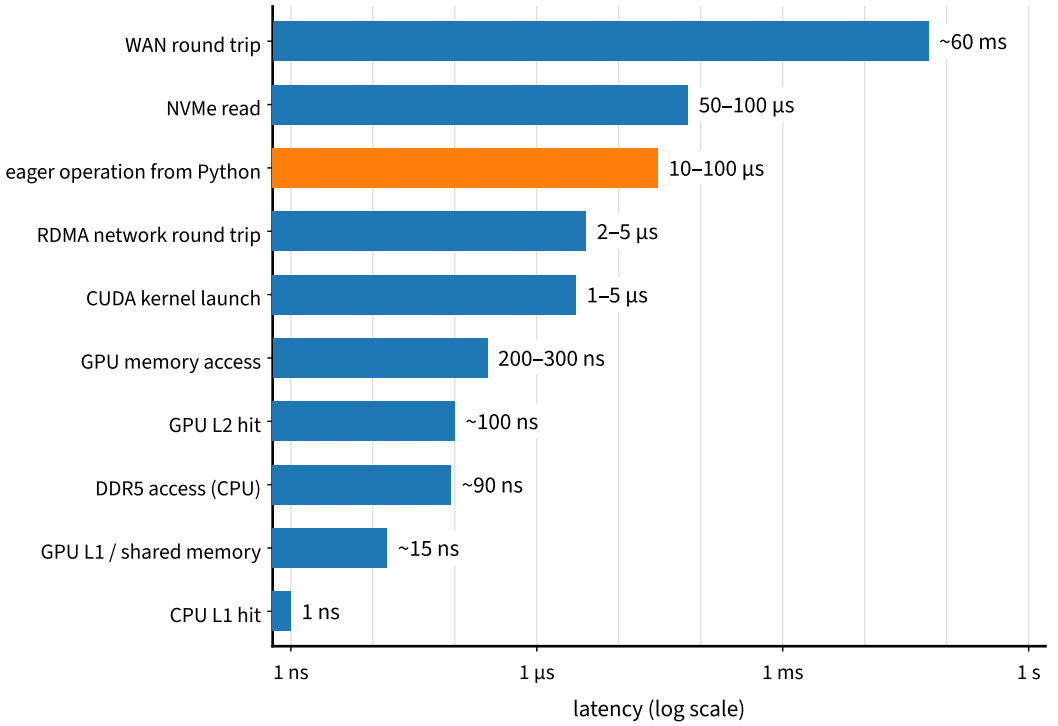


Figure 13.5: 対数スケールで示したレイテンシ。強調表示された項目は、Python からディスパッチされるフレームワーク演算 1 回あたりのホスト時間であり、カーネル起動はその一部を占める。

```
n = 256 * 1024 * 1024
x = torch.randn(n, device=d2l.try_gpu())  # 1 GiB fp32
times = {}
for desc in ('identity', 'random'):
    idx = (torch.arange(n, device=x.device) if desc == 'identity' else
           torch.randperm(n, device=x.device))
    # Bytes per element: the index and the output stream; x is read as
    # 4 bytes in order, or as a whole 32-byte sector at a random address
    nbytes = idx.dtype.itemsize + 4 + (4 if desc == 'identity' else 32)
    times[desc] = d2l.Benchmark(lambda: x[idx]).time
    print(f'{desc:8s}: {1e3 * times[desc]:5.1f} ms, '
          f'{nbytes * n / times[desc] / 1e12:.2f} TB/s at {nbytes} B/element')
    del idx  # Free the 2 GiB index before making the next one
print(f'random / identity: {times["random"] / times["identity"]:.1f}x')
```

```
identity:   4.8 ms, 0.90 TB/s at 16 B/element
random   :  37.4 ms, 0.32 TB/s at 44 B/element
random / identity: 7.8x
```

恒等置換によるギャザーが約 0.9 TB/s でストリーム処理されるのに対し、ランダムなギャザーは約 37 ms かかり（8–11× 遅い）、1 回の読み出しあたり 32 バイトのセクタ全体を転送量に算入してもピーク帯域幅の 3 分の 1 しか達成できない。32 バイトのセクタという粒度に加え、DRAM は数 KB 規模の開いた**行**（row）からデータを供給する仕組みになっている。連続するアドレスは開いた行を共有できるが、ランダムなアドレスは新しい行を開くレイテンシを繰り返し支払ううえ、アドレス変換キャッシュのミスも引き起こす。

密行列の積や埋め込み参照（1 トークンあたり数千バイトの連続する行を読み出す）は効率よくストリーム転送できる。対照的に、細粒度のランダムアクセス（疎行列、グラフ探索、ハッシュテーブル）は β のごく一部しか出せない。より一般に、n バイトの連続ブロックを立ち上がりレイテンシ α で転送するには $t(n) = \alpha + n/\beta$ の時間がかかり、$n_{1/2} = \alpha\beta$ でピーク帯域幅の半分に達する (Hockney 1994)。NVMe ドライブ（$\alpha \approx 50$–100 µs、$\beta = 14$ GB/s）では $n_{1/2} \approx 1$ MB となるため、ディスク上のデータセットは個々のデータ例ごとではなく MB 規模のシャード単位でシャッフルされる。

13.2.2 演算性能とメモリ帯域幅

13.2.2.1 面積と外周

演算ユニットがシリコンダイの内部を埋め尽くす（面積に比例し $O(n^2)$ でスケールする）のに対し、I/O パッドは外周、すなわち**ショアライン**に沿って並ぶ（辺の長さに比例し $O(n)$ でスケールする。Figure 13.6）。トランジスタの微細化も同じ効果をもたらす。最小加工寸法が半分になると論理回路の密度は 4 倍になるが、ダイ端におけるはんだバンプのピッチははるかに緩やかにしか縮小しない。

アーキテクトは、このショアラインのボトルネックを 2 つの方法で緩和している。

1. **シリコンインターポーザ上の HBM**：GPU ダイの隣に DRAM ダイを垂直に積層することで、数百本の基板配線を数千本の短く高密度なインターポーザ配線に置き換える（B200 では 8 TB/s に達するのに対し、CPU ソケットでは約 0.5 TB/s にとどまる）。
2. **マルチダイパッケージ**：リソグラフィの制約により単一ダイの面積は**レチクル限界**（reticle limit、約 858 mm^2）で頭打ちになるため、B200 のようなフラッグシップのアクセラレータはレチクル限界サイズのダイ 2 枚を 1 つのパッケージ上で結合し、外周を 2 倍にして 8 個の HBM スタックを収容している。

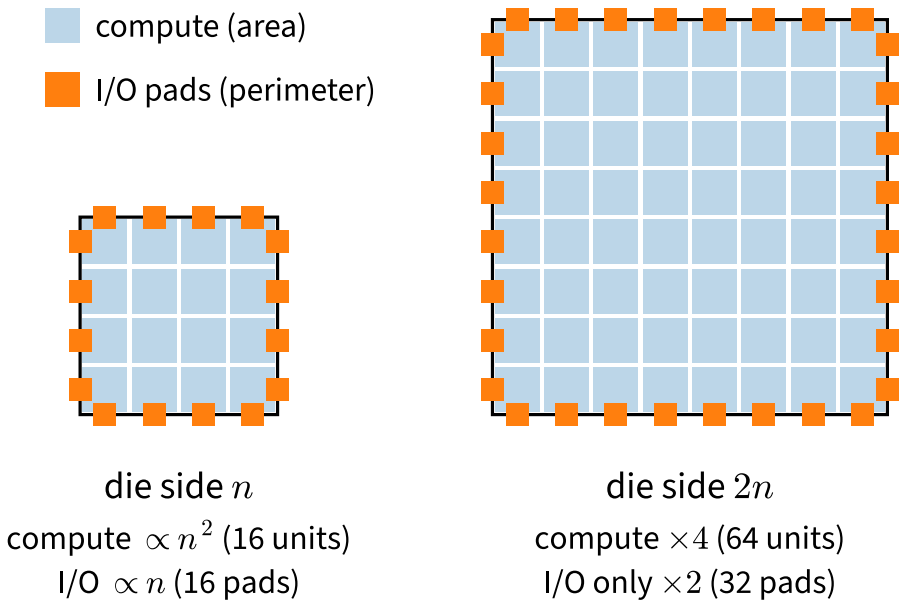


Figure 13.6: 演算は面積、I/O は外周で決まる。ダイの一辺を 2 倍にすると演算能力は 4 倍になるが、I/O のための余地は 2 倍にしかならない。

13.2.2.2 世代間のトレンド

V100（2017 年）から A100、H100 を経て B200（2024 年）に至る NVIDIA のデータセンタ向け世代を通じて、密な 16 ビット演算の FLOP/s は 1 世代あたり約 $2.6\times$、メモリ帯域幅は約 $2.0\times$、メモリ容量は約 $1.8\times$ のペースで拡大した (Gholami et al. 2024)。その結果、リッジポイントは V100 の 139 FLOP/byte から H100 の 295 FLOP/byte（B200 では 281 FLOP/byte）へと倍増した。したがって、演算密度（arithmetic intensity）が 200 FLOP/byte の演算は、V100 では計算律速であったが、H100 や B200 では帯域幅律速となる。

13.2.2.3 演算あたりのエネルギー

消費電力もまた、帯域幅に対する第 2 の厳格な上限を定める。45 nm CMOS プロセス（Figure 13.7、Horowitz (2014)）では、fp32 の乗算 1 回が 3.7 pJ、8 ビット整数の加算 1 回が 0.03 pJ であるのに対し、オフチップ DRAM から 64 ビットを読み出すには 1.3〜2.6 nJ、すなわちおよそ **500 回の fp32 乗算**に相当するエネルギーを要する。オンチップ SRAM の読み出しはその中間に位置する（距離に応じて 10〜100 pJ）。現代の微細化されたプロセスノードではすべての数値が縮小するものの、この階層関係自体は変わらない。データを移動させることは、データに対して演算するよりもはるかに多くのエネルギーを消費するのである。カーネル融合、狭い数値形式、および活性化値の再計算が電力を節約するのは、それらが時間を短縮するのとまったく同じ物理的理由による。

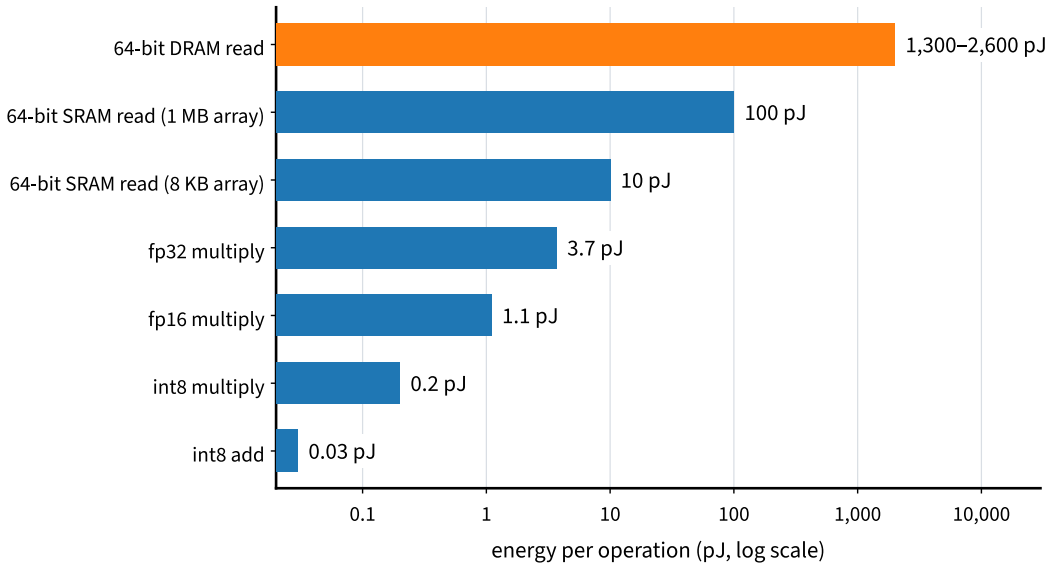


Figure 13.7: 45 nm プロセスにおける演算あたりのエネルギー。64 ビットの DRAM 読み出し 1 回は、fp32 の乗算約 500 回分に相当するエネルギーを消費する。

13.2.3 GPU

13.2.3.1 ストリーミングマルチプロセッサ

GPU はダイ面積の大半を数千個の演算ユニットに割り当てており、それらは**ストリーミングマルチプロセッサ**（SM、RTX 4090 では 128 基）へとまとめられている。各 SM は**ワープ**と呼ばれる 32 個のグループ単位でスレッドをロックステップ実行する。巨大なキャッシュに頼って DRAM のレイテンシを隠蔽するかわりに、SM は毎サイクル実行するワープを切り替え、メモリ待ちでストールしたワープがある間も別のワープが演算を続けられるようにする。

リトルの法則 (Little 1961) により、帯域幅 $\beta = 1.0$ TB/s のメモリパイプを 250 ns のレイテンシにわたって満たし続けるには、常時 $1.0\,\text{TB/s} \times 250\,\text{ns} \approx 250$ KB がインフライト（未完了の転送中）でなければならない（SM あたり約 2 KB、すなわち SM あたり 16 個のワープがそれぞれ 1 スレッドにつき 4 バイトのロードを待っている状態に相当する）。250 KB をインフライトに保てるだけの並列性を引き出せないカーネルは、帯域幅律速ではなくレイテンシ律速となる。

13.2.3.2 テンソルコアと数値形式

テンソルコアは小さな行列タイルの乗算と累積を単一命令で実行し、TPU のシストリックアレイと同様に、命令のフェッチとデコードをタイル全体にわたって償却する (Kung 1988; Jouppi et al. 2017)。H100 では、bf16 テンソルコアが 989 TFLOP/s を達成するのに対し、スカラー fp32 は 67 TFLOP/s であり（15× の差）、コンシューマ向けの RTX 4090 ではこの比が 2× となる。

RTX 4090 のピークスループットは、仕様から直接導出できる。128 基の SM × 128 個の fp32 演算器 = 16,384 個の演算器があり、それぞれが 2.52 GHz のブーストクロックで毎サイクル 1 回の融合積和演算（2 FLOPs）を実行するから、次のようになる。

$$P_{\text{fp32}} = 16{,}384 \times 2 \times 2.52 \times 10^{9}\ \text{s}^{-1} \approx 82.6\ \text{TFLOP/s}.$$

このカードのテンソルコアは、tf32 を同じ 82.6 TFLOP/s で実行し、bf16（fp32 累積あり）を $2\times = 165.2$ TFLOP/s で、fp8 を公称 $4\times = 330.4$ TFLOP/s で実行する。本章では tf32 と bf16 の速度を実測するが、fp8 の速度は本書のコードでは扱わない公称値である。

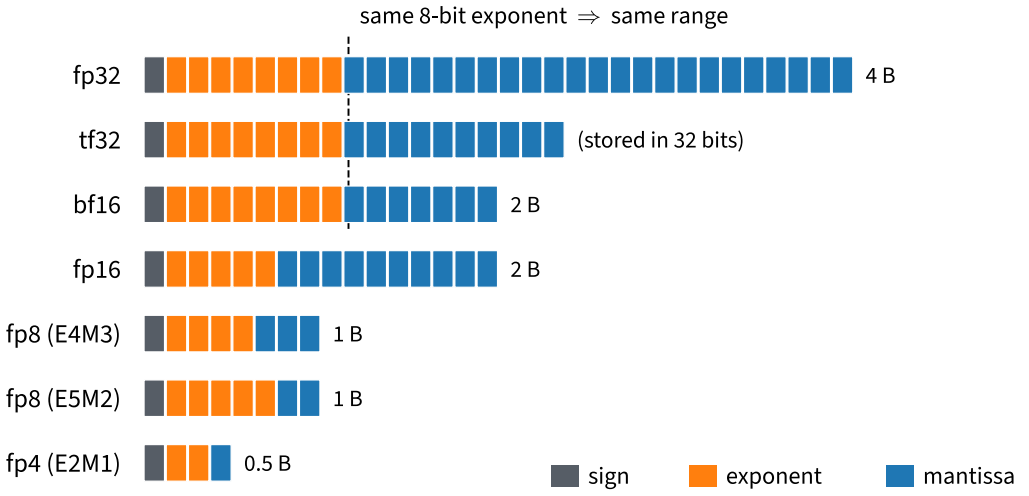


Figure 13.8: ビット単位で描いた浮動小数点形式。fp32、tf32、bf16 は 8 ビットの指数部を共有するため、表現範囲も同じである。tf32 はテンソルコアの演算モードでありメモリ上の値は 32 ビットで格納されるが、それ以外の形式では 1 段階狭まるごとに 1 値あたりのバイト数が半分になる。

Figure 13.8 が示すように、数値形式を狭めることにはルーフライン上で 2 つの効果がある。

1. **ビット幅を半分にすると移動バイト数が半減する**（帯域幅律速のカーネルが最大 $2\times$ 高速化され、重みのメモリ使用量も半減する）。ただし tf32 は例外であり、テンソルコア内部では 19 ビットの値（仮数部 10 ビット）で計算するものの、メモリ上には依然として 32 ビット浮動小数点数を格納する。
2. **ビット幅を半分にすると、通常はテンソルコアの計算ルーフ P が 2 倍になる。**

8 ビット以下の形式では明示的なスケール係数が必要になる。fp8 では、テンソル単位またはブロック単位のスケールを併用しつつ、順伝播の活性化値と重みに E4M3、勾配に E5M2 を用いる (Micikevicius et al. 2022)。4 ビット（fp4 / E2M1、-6 から 6 までの 15 通りの値しか符号化できない）になると、テンソル単位のスケーリングは外れ値に対して破綻するため、ハードウェアは 16 要素（NVFP4）または 32 要素（OCP MX）のブロックごとに 1 つの共有スケールを用いる**マイクロスケーリング**を適用する (Rouhani et al. 2023)。

13.2.3.3 アクセラレータの仕様

Table 13.2 に、2 つのデータセンタ向け GPU と 2 つのコンシューマ向けカードの比較を示す。

Table 13.2: 広く普及している 4 つの GPU。ブーストクロックにおける fp32 累積ありの密なテンソルコアスループットを TFLOP/s（TF）単位で示す。コンシューマ向けカード 2 機種の fp8 および fp4 の値は公称値であり、fp8 は bf16 の 2 倍、RTX 5090 の fp4 は実測値ではなくベンダの公称代表値に基づく。リンクは片方向の値である。

	H100 SXM	B200	RTX 4090（本機）	RTX 5090
メモリ容量 C	80 GB HBM3	180 GB HBM3e	24 GB GDDR6X	32 GB GDDR7
メモリ帯域幅 β	3.35 TB/s	8 TB/s	1.0 TB/s	1.8 TB/s
bf16	989 TF	2,250 TF	165 TF	210 TF
fp8	1,979 TF	4,500 TF	330 TF	419 TF
fp4	—	9,000 TF	—	1,676 TF
リッジポイント（bf16）	295 FLOP/B	281 FLOP/B	164 FLOP/B	117 FLOP/B
GPU 間リンク	NVLink 4、450 GB/s	NVLink 5、900 GB/s	PCIe 4.0 ×16（ホスト経由）、≤ 32 GB/s	PCIe 5.0 ×16（ホスト経由）、≤ 64 GB/s
消費電力	700 W	1,000 W	450 W	575 W

13.2.4 GPU 外のデータ経路

13.2.4.1 ホスト

算術演算を実行するのは GPU であるが、ホスト CPU も依然として訓練のボトルネックになり得る 3 つのタスクを担っている。すなわち、カーネルのディスパッチ（呼び出しあたり h）、データ読み込みと前処理パイプラインの実行、および PCIe を介したホストからデバイスへの転送の中継である。

GPU の DMA **コピーエンジン**が SM の演算と並行して PCIe 経由でデータを転送できるのは、転送元がページロック（**ピン留め**）ホストメモリにある場合だけである。通常の**ページャブル**メモリからの転送では、まず CPU が内部のピン留めステージングバッファへデータをコピーしなければならない。256 MiB の配列に対して、ページャブルメモリとピン留めメモリの転送速度を比較してみよう。

```
x_pageable = torch.randn(64 * 1024 * 1024)  # 256 MiB on the host
x_pinned = x_pageable.pin_memory()
gpu = d2l.try_gpu()
for desc, src in [('pageable', x_pageable), ('pinned', x_pinned)]:
    t = d2l.Benchmark(lambda: src.to(gpu, non_blocking=True)).time
    print(f'{desc}: {src.nbytes / t / 1e9:.1f} GB/s')
```

```
pageable: 14.2 GB/s
pinned: 23.1 GB/s
```

ピン留めメモリからの転送は約 23 GB/s に達する。これは PCIe 4.0 ×16 の理論上限である 32 GB/s の約 4 分の 3 であり、GPU メモリと比べると $40\times$ 遅い。一方、ページャブルメモリからの転送はホストメモリ上での余分なコピーが加わるためさらに遅くなる。GPU メモリに収まるデータセットはデバイス上に常駐させるべきであり、ホストからストリーム供給するミニバッチにはピン留めホストメモリを用いて演算とオーバーラップさせるべきである。

`DataLoader` で `pin_memory=True` を指定すると、ミニバッチはピン留めメモリ上に準備される。

13.2.4.2 GPU 間のリンク

データセンタ向け GPU は専用の NVLink ファブリック（片方向あたり H100 で 450 GB/s、B200 で 900 GB/s）を介して通信する。一方、コンシューマ向けの RTX 4090 カードは NVLink も PCIe のピアツーピア（P2P）アクセスも備えていない。Figure 13.9 に示すように、本書の 4 GPU ワークステーションでは GPU 間を行き来するすべてのバイトが一方の PCIe 4.0 リンクを上り、ホスト DRAM を経由して、もう一方のリンクを下らなければならないため、GPU 間転送はピン留めコピーの速度である約 23 GB/s（NVLink 5 より $40\times$ 遅い）に制限される。

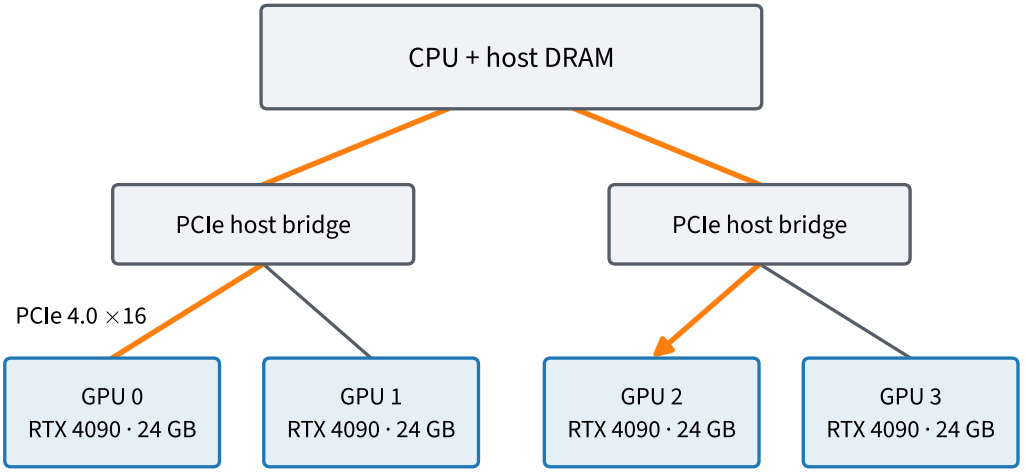


Figure 13.9: `nvidia-smi topo` が報告する本書のマシンの構成。2 つの PCIe ホストブリッジの配下に 4 基の RTX 4090 が 2 基ずつ接続されており、ピアツーピアアクセスも NVLink もない。強調表示された経路は、ある GPU から別の GPU へ向かうすべてのバイトがたどるルートである。

1B パラメータの GPT（fp32 勾配で $S = 4$ GB）の場合、勾配を 23 GB/s で 1 回移動させるだけで本書のワークステーションでは少なくとも 0.17 s かかる。NVLink 5 経由なら 5 ms 未満で済むのに対し、これは bf16 のピークスループットにおける 8,192 トークンの 1 ステップあたりの演算時間（0.3 s）の半分以上を占める。

13.2.5 ルーフライン上のワークロード

言語モデルの推論は 2 つの領域にまたがる（11.3.2.1 節）。**プレフィル**は n トークンのプロンプトを並列に処理し、（層の幅に達するまで）$I \approx n$ の演算密度を達成して計算ルーフ上で動作する。一方、**デコード**は 1 ステップにつき 1 トークンを生成し、bf16 の重み全 $2N$ バイトと KV キャッシュを読み出すため、演算密度は $I \approx 1$ FLOP/byte となる。

13.2.5.1 単一ストリームデコードとバッチデコード

単一ストリームのデコードは帯域幅律速であるため、そのトークン生成速度は次の式で上から抑えられる。

$$\text{tokens/s} \lesssim \frac{\beta}{\text{bytes per token}} = \frac{\beta}{2N + \text{KV bytes}}. \tag{13.4}$$

モデルの重みがカードのメモリ容量 C を埋め尽くすとき、単一ストリームでの最大デコード速度は**メモリスイープ時間**（memory sweep time）C/β、すなわち GPU DRAM の全バイトを 1 回読み出すのに要する時間の逆数になる。Table 13.2 の 4 つの GPU すべてにおいて、容量と帯域幅には 7–8× の開きがあるにもかかわらず、C/β はいずれも **20 ms** 付近に収まっている（H100 と RTX 4090 で 24 ms、B200 で 22.5 ms、RTX 5090 で 18 ms）。したがって、GPU メモリをいっぱいに使う任意の密なモデルは、コンシューマ向けカードであれフラッグシップのデータセンタ向け GPU であれ、単一ストリームをおよそ 40〜55 tokens/s でデコードすることになる。なお、この上限は各トークンがすべての重みを読み出すことを前提としている。混合エキスパート（mixture-of-experts）モデル（11.7 節）は、ルータがそのトークンのために選択したエキスパートのみを読み出すため、C/β が示唆するよりも高速にデコードできる。

同時並行する B 本の系列をバッチ化すると、重みの読み出しが B トークンにわたって償却され、重みに関する演算密度は $I \approx B$ FLOP/byte へと上昇する（B がリッジポイントに達するまで。13.1.3.1 節で実測したとおり）。ただし、各系列は依然として自身の KV キャッシュを個別に読み出す必要がある（11.3.2.2 節）。

13.2.5.2 経験則

Table 13.3 に、本章の残りの節を通じて用いる主要なハードウェア定数をまとめる。

Table 13.3: 経験則：Table 13.2 の広く普及している GPU における代表的なオーダー（最新のデータセンタ向け製品は上限を上回る）、本書のマシン（ピアツーピアアクセスなしの RTX 4090 4 基）における値、およびそれぞれが重要である理由。

物理量	代表的な大きさ	本書のマシン	重要である理由
呼び出しあたりのホスト時間 h	10 µs〜0.1 ms	13〜100 µs（13.1 節）	小さな演算では GPU を稼働させ続けられない
bf16 ピーク演算速度 P	0.17〜2.3 PFLOP/s	165 TFLOP/s	大規模な行列積の速度上限
GPU メモリ帯域幅 β	1〜8 TB/s	約 0.9 TB/s（定格 1.0）	要素ごとの演算とデコードの速度上限
リッジポイント P/β	120〜300 FLOP/byte	164 FLOP/byte	これ未満ではメモリ帯域幅が実行時間を決める
スイープ時間 C/β	約 20 ms	24 ms	メモリに収まる最大の密モデルは 1 ストリームを約 40〜55 tok/s でデコード
ホストからデバイス（ピン留め）	32〜64 GB/s（PCIe 4/5）	約 23 GB/s	訓練データはデバイス上またはピン留めメモリに置く
GPU 間	450〜900 GB/s（NVLink）	ホスト経由で $\leq$ 23 GB/s（13.5 節）	毎回の勾配交換にかかる時間を決める
パラメータあたりバイト数（Adam）	16（fp32）、18（混合精度）	16〜18	メモリに収まるモデルサイズを決める（13.4 節）
トークンあたりバイト数（デコード）	$2N$ + KV キャッシュ（bf16）		tokens/s $\lesssim \beta$/bytes per token

13.2.6 まとめ

帯域幅は配線の本数と 1 本あたりの速度の積で決まる。配線がダイの外周に沿って外へ出るのに対し演算ユニットは面積を埋め尽くすため、またオフチップのデータ移動は算術演算の数百倍ものエネルギーを要するため、リッジポイント P/β は数百 FLOP/byte に達し、ハードウェアの世

代を追って上昇してきた。メモリは開いた DRAM 行から 32 バイトのセクタ単位でデータを転送するため、細粒度のランダム読み出しは β のごく一部しか達成できず、β を飽和させるには複数のワープにわたって $\beta \times$ latency バイトがインフライトである必要がある。より狭い数値形式はメモリ通信量を半減させ、通常は P を 2 倍にする。最後に、単一ストリームでのデコードにおけるトークン生成速度は $\beta/(2N + \text{KV bytes})$ で上から抑えられ、現代の GPU ではメモリスイープ時間 C/β がいずれも約 20 ms であることから、どのカードであってもメモリに収まる最大の密なモデルは単一ストリームを約 40〜55 tokens/s でデコードする。

13.2.7 演習問題

1. [code] **レイテンシと帯域幅の実測**。●ブロック転送モデル $t(n) = \alpha + n/\beta$（ただし $n_{1/2} = \alpha\beta$、13.2.1 節）は、1 回の同期呼び出しを記述する。

    1. 13.1.3 節の要素ごとの乗算 `x * 1.5` に対する $n_{1/2}$ を、$\alpha = h$ および β に 13.1 節での計測値を用いて予測せよ。
    2. 1 MB から 2 GB までのテンソルサイズに対して `x * 1.5` の実行時間を計測し、実効帯域幅をサイズに対して両対数軸でプロットしたうえで、領域の遷移を Figure 13.5 および Figure 13.3 のオンダイ L2 キャッシュ容量と関連づけよ。
    3. 4 KB から 256 MB までの転送に対して 13.2.4.1 節のホストからデバイスへの計測を繰り返し、α と漸近的な帯域幅をフィッティングにより求めよ。

2. **配線数と伝送速度**。 13.2.1 節では、帯域幅を配線の本数と 1 本あたりの速度の積として表した。

    1. DDR5-6400 DIMM は 64 本のデータ配線をもち、各配線は 6.4 Gb/s で動作する。HBM3e スタックは 1,024 本のデータ配線をもち、B200 は 8 個のスタックを搭載している。DIMM 1 枚の帯域幅、0.5 TB/s の CPU ソケットに必要な DIMM の枚数、および Table 13.2 にある B200 の 8 TB/s から逆算される配線 1 本あたりのデータ速度を計算せよ。CPU ソケットと B200 の差を生み出している主な要因は何か。
    2. B200 の 2 枚のダイはそれぞれ、8 個の HBM スタックのうち 4 個に隣接している。各ダイ上の SM がメモリの両半分を同じ頻度で読み出し、全体で 8 TB/s になるとき、パッケージ上のブリッジは片方向あたりどれだけの帯域幅を担う必要があるか。

3. **データ供給不足の入力パイプライン**。訓練ジョブの GPU 演算部が、デコード済み画像を 2 GB/s の速度で処理できるとする。各 CPU ワーカーは 2 つの逐次ステージを実行する。まず NVMe から圧縮された JPEG を 1 GB/s（圧縮時サイズ）で読み出し、次に 1 つの CPU コア上で 0.3 GB/s（デコード後出力サイズ、サイズが 8× に拡大）でデコードする。続いて訓練ループは、演算を始める前にデコード済みの各バッチをページャブルなホストメモリから GPU へ 12 GB/s で同期コピーする。

    1. ワーカー 1 個がバッチを供給する速度（デコード後の bytes/s 単位）はいくらか。また、どちらのステージが律速となるか。
    2. GPU にデータを供給し続けるには、何個の並列ワーカープロセスが必要か。
    3. ページャブルメモリからの同期コピーによって各ステップの時間はどの程度の割合だけ延びるか。また、24 GB/s のピン留めメモリを用いた非同期コピーに切り替えると、これはどう変わるか。

4. **リンクのリッジポイント**。リッジポイント P/β は帯域幅 β をもつ任意のデータ経路に適用でき、1 バイトがその経路を通過する間にデバイスが実行できる FLOPs 数を表す。

    1. bf16 における P/β を、RTX 4090 については自身の DRAM とホスト経由の GPU 間経路（23 GB/s）のそれぞれに対して、B200 については自身の HBM と NVLink 5 の片方向（900 GB/s）のそれぞれに対して計算せよ。

2. データ並列訓練（13.5 節）では、各 GPU がパラメータ数 N のモデルを保持し、1 ステップあたり T トークンを 1 トークンにつき約 $6N$ FLOPs で処理したのち、GPU あたり約 $4N$ バイトを送信するオールリデュース（allreduce）によって bf16 勾配を合算する。通信と演算のオーバーラップがない場合、T がリンクのリッジポイントに比例する（N に依存しない）しきい値を超えるとき、かつそのときに限り、通信時間が演算時間以下になることを示せ。
3. このトークン数のしきい値を、NVLink 5、23 GB/s のピン留めコピー速度、および本書のワークステーションにおけるデフォルトのホスト経由 NCCL の実効速度である 2.4 GB/s（13.6 節）のそれぞれについて評価せよ。

5. **起動律速のデコード**。Eager モードでの単一ストリームデコードでは、GPU が各ブロックの重みを読み出すのと並行して、CPU がすべてのブロックのすべてのカーネルを起動する。L 個ある幅 d の各ブロックが $12d^2$ 個の bf16 重みを保持し、1 トークンあたり $K = 12$ 個のカーネルを、カーネル 1 個につきホスト時間 $h = 8$ µs で起動するとする。
   1. ホストの起動時間と重みの読み出し時間のどちらが律速となるかが L に依存しないことを示し、RTX 4090 と B200 のそれぞれにおける境界幅 d^* を求めよ。
   2. RTX 4090 上の GPT-2 Small（$d = 768$、$L = 12$）について両方の上限を評価せよ。どちらの上限が律速となるか。
   3. RTX 4090 から B200 への d^* の変化は、より高速な GPU における CUDA グラフのキャプチャとリプレイ（13.3 節）の重要性について何を示唆するか。

6. [code] **ブロック単位でシャッフルしたギャザー**。●13.2.1 節のギャザーのベンチマークを繰り返せ。ただし、個々の要素ではなく連続する k 要素のブロックを $k \in \{1, 2, 4, \ldots, 1024\}$ に対してランダムに置換せよ。実効帯域幅をバイト単位のブロックサイズに対してプロットし、遷移が生じる境界サイズを 32 バイトのセクタサイズおよび KB 規模の DRAM 行サイズと比較せよ。

7. **レイテンシ隠蔽**。●リトルの法則（13.2.3 節）をメモリレイテンシの隠蔽に適用せよ。
   1. 250 ns のレイテンシの下で B200 の 8 TB/s の HBM を飽和させるのに必要なインフライトのバイト数を計算し、RTX 4090 の 250 KB と比較せよ。DRAM のレイテンシがほぼ一定のまま帯域幅が世代ごとに約 2× のペースで拡大することは、カーネルが引き出すべき並列性に対して何を意味するか。
   2. ポインタチェイシング（ポインタ追跡）では、各ロードのアドレスが直前のロード結果に依存するため、1 本の連鎖につきインフライトにできるリクエストは 1 個に限られる。DDR5 のレイテンシが 90 ns、キャッシュラインが 64 バイトの CPU において、単一の連鎖で得られる帯域幅と、0.5 TB/s のソケットを飽和させるのに必要な独立した連鎖の本数を計算せよ。

13.3 計算グラフとコンパイル

13.1 節では、8 個の演算からなる活性化処理の連鎖 `gelu_ish` は、同じテンソルに対する単一の加算と比べて 8〜9 倍の時間がかかった。各演算が個別の GPU カーネルとして実行され、そのたびに入力を DRAM から読み出して出力を書き戻していたためである（配列転送が 18 回発生しており、本来の 2 回では済まなかった）。また、小さなテンソルでは、同じ演算列がもう 1 つの理由でも遅くなる。すべての演算が Python からディスパッチされ、GPU が各カーネルを実行する時間よりも、ホストがカーネルを起動する時間のほうが長くなるからである。

これら 2 つのコストはいずれも、Python が到達した順に演算を 1 つずつ実行する**即時実行**（eager execution）に起因する。実行前にプログラム全体をコンパイラへ渡せば、両方の問題に対処できる。第 1 に、**演算融合**（operation fusion）は要素ごとの演算列を単一のカーネルに統合して

入力の読み出しと出力の書き込みを各 1 回に抑え、帯域幅律速（bandwidth-bound）のコストを解消する。第 2 に、**カーネルリプレイ**は一連のカーネル起動を記録してホストからの 1 回の呼び出しでリプレイし、オーバーヘッド律速（overhead-bound）のコストを解消する。計算律速（compute-bound）となる大規模な行列積そのものは、このどちらからも直接の恩恵を受けないが、その前後の要素ごとの演算をコンパイルすることには依然として十分な効果がある。

```
import os
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
# Detach PyTorch's profiler from the GPU when a profile ends; without this,
# every kernel launch after the first profile stays about 30% slower
os.environ['TEARDOWN_CUPTI'] = '1'
```

13.3.1 グラフキャプチャ

13.3.1.1 計算グラフ

ニューラルネットワークの演算は、基本演算をノードとし、テンソルを運ぶエッジで結んだ有向の**計算グラフ**（computation graph）をなす（Figure 13.10）。即時実行の自動微分（1.5 節）でも順伝播の最中に逆伝播用のテープを記録するが、その記録は順伝播自体を最適化するには遅すぎる。テープが出来上がった時点では、すべての演算がすでにホストのディスパッチ時間 h を支払い、GPU メモリとの往復を終えてしまっているからである。プログラムを最適化するには、演算が実行される前に計算グラフを取得する**グラフキャプチャ**が必要になる。

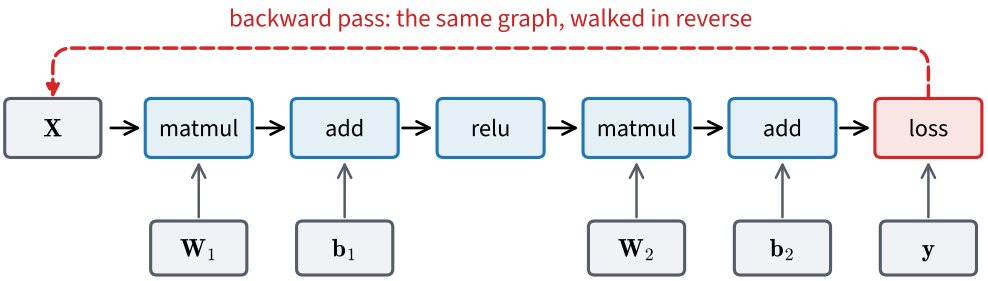


Figure 13.10: 損失を含む 2 層ネットワークの計算グラフ。自動微分はこのグラフを逆方向にたどる。順伝播の前にグラフ全体を持つコンパイラは、グラフを書き換えることができる。

13.3.1.2 トレース

現代のフレームワークは**トレース**によってグラフをキャプチャする。すなわち、形状と dtype だけを持ち具体的な数値を持たないシンボリックな**プレースホルダ**に対して、Python 関数を 1 回実行するのである（Figure 13.11）。プレースホルダに対するすべてのテンソル演算はグラフのノードとして記録され、静的な形状やループ上限のみに依存する通常の Python の制御フローはトレース中に展開される。続いてコンパイラは、キャプチャしたグラフを融合 GPU カーネルへとローワリングし、入力の形状と dtype が一致する後続の呼び出しで再利用できるようキャッシュする。

PyTorch では、`torch.compile` が TorchDynamo によって Python バイトコードを捕捉し (Ansel

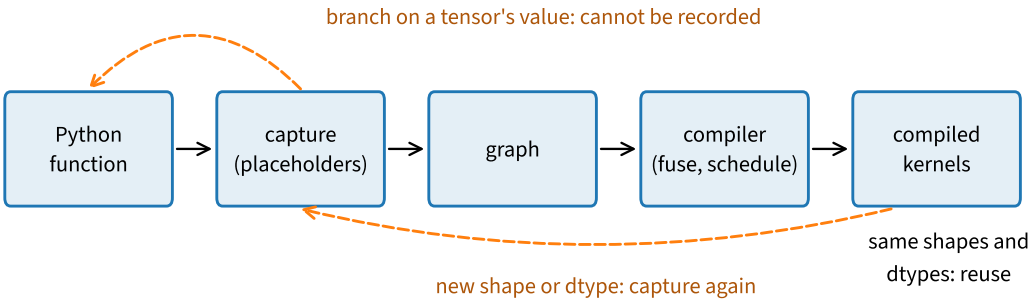


Figure 13.11: グラフキャプチャ。最初の呼び出しでプレースホルダを用いて関数をトレースし、コンパイラが記録されたグラフをカーネルへと変換する。同じ形状と dtype を持つ後続の呼び出しは、そのカーネルを再利用する。テンソルの値に依存する分岐は記録できず、新しい形状や dtype が現れると新たなキャプチャが始まる場合がある。

et al. 2024)、テンソル演算をグラフへとトレースしたうえで、AOTAutograd を用いて逆伝播グラフも事前にキャプチャし、Inductor によって両者を融合された Triton および C++ カーネルへとコンパイルする。Dynamo は入力の形状と dtype に**ガード**を設定するため、後続の呼び出しではガードを通過すればそのままコンパイル済みカーネルへ直行できる。

13.3.1.3 値に依存する分岐

プレースホルダは具体的な数値を持たないため、トレーサは `if x.sum() > 0:` のようなデータ依存の Python の分岐を評価できない。グラフキャプチャの仕組みが取り得る対応は次の 3 通りである。

1. 具体例の入力が通った分岐で**凍結する**。高速だが、将来の入力がもう一方の分岐を通るべきときに、警告なしに誤った結果を返す。
2. エラーを出して**拒否する**。プログラマは、**選択演算**（select、両方の分岐を評価して一方を選ぶ）または**条件分岐演算**（conditional op）というグラフプリミティブで分岐を表現しなければならない。
3. **グラフブレーク**を挿入して Python に**フォールバックする**。条件式の箇所で最初のグラフを終了し、実体化された値に対して Python に分岐を評価させ、2 つ目のコンパイル済みグラフで処理を再開する。

静的なテンソル形状に基づく分岐（`if x.shape[0] > 4:`）はトレース時に確定するため、特別な扱いは不要である。フレームワークがデータ依存の分岐をどのように扱うか見てみよう。

```
def f(x):
    x = x * 2
    if x.sum() > 0:        # Python needs the value of this comparison
        return x + 1
    return x - 1

explanation = torch._dynamo.explain(f)(torch.randn(8, device=d2l.try_gpu()))
print(f'graphs: {explanation.graph_count}, '
      f'breaks: {explanation.graph_break_count}')
for graph in explanation.graphs:
```

```
        print(graph.code.strip())
```

```
graphs: 2, breaks: 1
def forward(self, L_x_ : torch.Tensor):
    l_x_ = L_x_
    x = l_x_ * 2;  l_x_ = None
    sum_1 = x.sum()
    gt = sum_1 > 0;  sum_1 = None
    return (gt, x)
def forward(self, L_x_ : torch.Tensor):
    l_x_ = L_x_
    sub = l_x_ - 1;  l_x_ = None
    return (sub,)
```

デフォルトでは、torch.compile はグラフブレークを挿入する。グラフ 1 が x * 2 と x.sum() > 0 を計算して Python に制御を返し、if を評価したうえで、選択された分岐に対応するグラフ 2 をディスパッチする。代わりに fullgraph=True を渡す（または torch.export を使う）とグラフブレークは拒否される。一方、分岐を torch.where(x.sum() > 0, x + 1, x - 1) で書き換えれば単一のグラフとしてキャプチャできる。従来の torch.jit.trace は、例として与えた入力が通った分岐で凍結する。

13.3.1.4 形状と再コンパイル

コンパイラはカーネル（タイルサイズ、ループ上限、メモリレイアウトなど）をテンソル形状に合わせて特化させるため、エポック末尾の短いミニバッチや可変の系列長のように入力形状が変化すると、コンパイラがシンボリック次元へ一般化しない限り再コンパイルが発生する。実務では、入力を少数のバケット化された形状にパディングするか、長さが変化する軸を動的軸として指定する。

```
compilations = []
def recording_backend(gm, example_inputs):  # Called once per compilation
    compilations.append(gm)
    return gm.forward                       # Run the captured graph as is

g = torch.compile(lambda x: torch.sin(x) + 1, backend=recording_backend)
for n in (8, 8, 16, 32):
    g(torch.ones(n, device=d2l.try_gpu()))
print(f'compilations: {len(compilations)}')
print(compilations[-1].code.strip())
```

```
compilations: 2
def forward(self, s77 : torch.SymInt, L_x_ : torch.Tensor):
    l_x_ = L_x_
    sin = torch.sin(l_x_);  l_x_ = None
    add = sin + 1;  sin = None
    return (add,)
```

Dynamo は最初の呼び出し（$n = 8$）でコンパイルし、2 回目の呼び出し（$n = 8$）ではそのグラフを再利用する。続いて $n = 16$ が与えられるとシンボリック次元（SymInt s77）を用いて再コンパイルし、その後の $n = 32$ は追加のコンパイルなしで処理する。

13.3.2 演算融合

13.3.2.1 メモリトラフィックの算出

即時実行では、`gelu_ish(x)` は 8 個の要素ごとのカーネル（配列を 1 個読み出して 1 個書き出す単項演算 6 個と、配列を 2 個読み出して 1 個書き出す二項演算 2 個）を実行し、合計 $10 + 8 = 18$ 個のサイズ nb バイトの配列を転送する。これに対し、融合カーネルは `x` を 1 回読み出し、7 個の中間値をすべてレジスタに保持して、出力を 1 回だけ書き出す（Figure 13.12）。これにより、メモリトラフィックは次の倍率で削減される。

$$\frac{\text{bytes moved, eager}}{\text{bytes moved, fused}} = \frac{(10+8)\,nb}{(1+1)\,nb} = 9.$$

融合カーネルでさえ約 12 FLOPs / 8 バイト（fp32 の場合、$I \approx 1.5 \ll 82$ FLOP/byte）しか演算しないため、どちらの版も帯域幅律速であり、融合によって `gelu_ish` は約 $9\times$ 高速化するはずである。

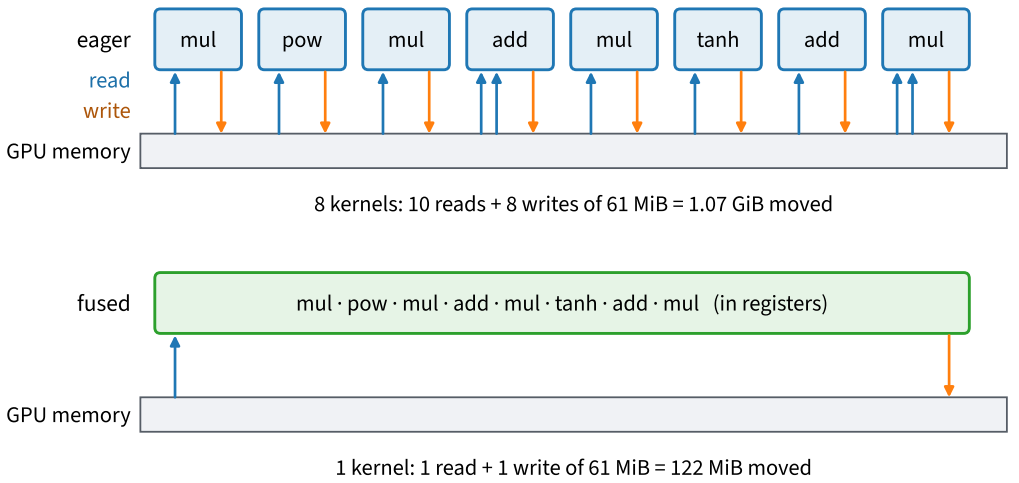


Figure 13.12: `gelu_ish` のメモリトラフィック。即時実行は 8 個のカーネルを実行して配列を 10 回読み出し 8 回書き出すのに対し、融合カーネルは入力を 1 回読み出し、7 個の中間結果をレジスタに保持して、結果を 1 回だけ書き出す。

```
x = torch.randn(4000, 4000, device=d2l.try_gpu())
compiled = torch.compile(d2l.gelu_ish)
assert torch.allclose(d2l.gelu_ish(x), compiled(x), atol=1e-6)  # Same answer
arrays_moved = {'eager': 18, 'compiled': 2}  # In units of x.nbytes
for desc, fn in (('eager', d2l.gelu_ish), ('compiled', compiled)):
    b = d2l.Benchmark(lambda: fn(x), desc=desc)
    print(f'{b}, {arrays_moved[desc] * x.nbytes / b.time / 1e9:.0f} GB/s')
```

```
eager: 1.25 ms/call, 920 GB/s
compiled: 0.15 ms/call, 878 GB/s
```

どちらの版も 0.6〜0.9 TB/s でメモリをストリーミングし、コンパイル版は 18 回の DRAM 転送のうち 16 回を省くだけで $8\times$ から $11\times$ 高速に動作する。これはバイト数から予測した $9\times$ とよ

く一致している。

Transformer ブロックの内部では、活性化の融合はどれほど重要だろうか。幅 $d = 2048$、FFN の隠れ幅 $4d = 8192$ を持つ bf16 の 1B パラメータ GPT において、融合されていない 8 演算の活性化を M トークンに対して実行すると $18 \cdot 4dM \cdot 2 = 144\,dM$ バイトが転送される一方、その前後にある 2 つの FFN 行列積は $2 \cdot 2 \cdot M \cdot d \cdot 4d = 16\,d^2 M$ FLOPs を実行する。両者の所要時間の比はトークン数 M に依存しない。

$$\frac{t_{\text{activation}}}{t_{\text{products}}} = \frac{144\,dM/\beta}{16\,d^2 M/P} = \frac{9I^*}{d} \approx 0.72$$

ただし、上の数値は $I^* = 164$ FLOP/byte、$d = 2048$ とした場合である。融合しなければ、活性化には 2 つの行列積の 4 分の 3 近い時間がかかるが、融合すれば 10 分の 1 未満に収まる。

13.3.2.2 融合のパターン

コンパイラが日常的に融合するパターンは次の 2 つである。

- **要素ごとの演算列**（活性化関数、残差加算、ドロップアウトマスク、オプティマイザの更新）：各スレッドが出力の要素 i をレジスタ内で計算する。
- **行リダクション**（ソフトマックス、LayerNorm、RMSNorm）：各行がオンチップ SRAM に収まるため、カーネルは行を 1 回読み出し、リダクションと正規化をオンチップで計算して、出力を 1 回だけ書き出す。

コンパイラは新しいアルゴリズムを発明するわけではない。計算律速の行列積を高速化することはできないし、FlashAttention（10.5 節、(Dao et al. 2022)）のようなタイル化アルゴリズムを自動的に発見することもできない。FlashAttention はオンラインソフトマックスの再スケーリングによって $T \times T$ のアテンション行列を実体化せずに済ませる手法であり、CUDA や Triton で手書きされている (Tillet et al. 2019)。

13.3.3 訓練ステップのコンパイル

ここからは、幅 1,024、`gelu_ish` 活性化を持つ 4 層 MLP に対し、512 個の例からなるバッチ上での完全な訓練ステップ（順伝播、二乗損失、逆伝播、SGD 更新）をコンパイルしてみよう。

```
class MLP(nn.Module):
    """Linear layers with the gelu_ish chain between them."""
    def __init__(self, width=1024, depth=4):
        super().__init__()
        self.layers = nn.ModuleList(nn.Linear(width, width)
                                    for _ in range(depth))

    def forward(self, X):
        for layer in self.layers[:-1]:
            X = d2l.gelu_ish(layer(X))
        return self.layers[-1](X)

def train_step(net, X, Y, lr=0.1):  # Forward, backward, SGD update
    loss = ((net(X) - Y) ** 2).mean()
    grads = torch.autograd.grad(loss, list(net.parameters()))
    with torch.no_grad():
        for p, g in zip(net.parameters(), grads):
```

```
            p -= lr * g
    return loss
```

まず 1 ステップの所要時間を予測してみよう。順伝播の 4 回の行列積と逆伝播の 7 回の行列積（最初の層は入力に関する勾配を必要としない）で約 $11 \times 2 \cdot 512 \cdot 1024^2 \approx 1.2 \times 10^{10}$ FLOPs を実行し（tf32 の 82.6 TFLOP/s で 0.14 ms）、16 MiB のパラメータの読み出しと更新で約 96 MiB を転送するため（1 TB/s で 0.10 ms）、実際のデバイス処理は合計**約 0.25 ms** となる。ところが即時実行では、100 個を超える個別の演算をそれぞれ $h \geq 10$ µs かけて発行するため（ホストの起動オーバーヘッドだけで 1 ms 超）、即時実行のステップはオーバーヘッド律速になる。

コンパイルには最初の呼び出し時に t_{compile} のコストがかかり、1 ステップあたり $t_{\text{eager}} - t_{\text{compiled}}$ だけ時間が短縮されるため、元が取れるまでのステップ数は次の式で与えられる。

$$n^* = \frac{t_{\text{compile}}}{t_{\text{eager}} - t_{\text{compiled}}} \tag{13.5}$$

このステップ数を実行した時点で損益分岐点に達する。

```
net = MLP().to(d2l.try_gpu())
X, Y = (torch.randn(512, 1024, device=d2l.try_gpu()) for _ in range(2))
compiled_step = torch.compile(train_step)
t_first = d2l.Benchmark(lambda: compiled_step(net, X, Y),
                        warmup=0, repeats=1).time  # Includes compilation
t_eager = d2l.Benchmark(lambda: train_step(net, X, Y)).time
t_compiled = d2l.Benchmark(lambda: compiled_step(net, X, Y)).time
saved = t_eager - t_compiled  # Per step; a difference of two short times
print(f'first call: {t_first:.1f} s\n'
      f'eager: {1000 * t_eager:.2f} ms, compiled: {1000 * t_compiled:.2f} ms\n'
      'break-even: ' + (f'about {float(f"{t_first / saved:.1g}"):,.0f} steps'
                        if saved > 0 else 'never, compiled is not faster'))
# explain is internal to torch; it clears the compile cache and recompiles
print('graphs:', torch._dynamo.explain(train_step)(net, X, Y).graph_count)
```

```
first call: 0.6 s
eager: 2.93 ms, compiled: 2.08 ms
break-even: about 700 steps
graphs: 2
```

デフォルトの `torch.compile` による高速化は 2× 未満にとどまり、デバイス処理の 0.25 ms と比べて依然として 5〜10 倍の時間がかかる。`torch.autograd.grad` によって順伝播・逆伝播とパラメータ更新の間にグラフブレークが生じるうえ、Inductor が生成するラッパが依然として Python から各カーネルを個別に起動するためである。

13.3.4 カーネルリプレイ

融合によって行列積同士を統合することはできず、残った各カーネルの実行時間が短い場合には、カーネルごとのホストからの起動がステップ時間の大半を占めるようになる。幅 256 のバイアスなし線形層と tanh 活性化を 60 層重ねたネットワークに、64 個の例からなるバッチを入力する場合を考えよう（1 回の呼び出しあたり 120 演算）。各層は 8.4×10^6 FLOPs の演算と 384 KiB の転送しか行わないが（ルーフライン上では 0.5 µs 未満であり、ハードウェアのカーネル起動下限

にカーネルあたり約 1〜3 µs が加わって実際のデバイス実行時間は約 4〜5 µs)、即時実行では 1 演算ごとに h（合計 120 h）を支払うことになる。

カーネルごとのホストディスパッチをなくすため、NVIDIA の GPU は **CUDA グラフ**をサポートしている。ランタイムが一連のカーネル起動と固定されたバッファアドレスを 1 回記録し、その後はホストからの 1 回の起動で一連の処理全体をリプレイする仕組みである。記録時にバッファアドレスと形状が固定されるため、リプレイ前に入力を固定バッファへコピーし、次のリプレイ時には出力が上書きされる。

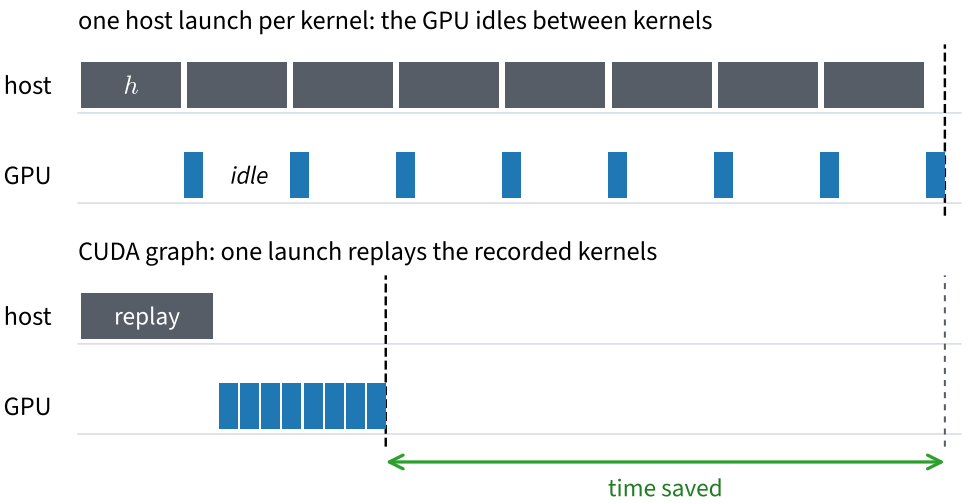


Figure 13.13: 短いカーネルを多数重ねた場合のホストとデバイスのタイムライン。カーネルごとに 1 回ずつ起動すると、GPU は毎回ホストを待つことになり、カーネル間でアイドル状態になる。CUDA グラフは 1 回の起動で一連の処理全体をリプレイするため、カーネルが隙間なく連続して実行される。

Figure 13.13 に 2 つのタイムラインの比較を示す。個々のカーネル自体の実行時間はどちらでも同じであり、リプレイが取り除くのはカーネル間のアイドル時間である。

```
deep = nn.Sequential(*[m for _ in range(60)  # 120 operations per call
                       for m in (nn.Linear(256, 256, bias=False), nn.Tanh())])
deep = deep.to(d2l.try_gpu())
x = torch.randn(64, 256, device=d2l.try_gpu())

fused = torch.compile(deep)  # Inductor kernels, launched one at a time
replayed = torch.compile(deep, mode='reduce-overhead')  # As a CUDA graph
with torch.no_grad():  # Inference: every call is a complete step
    fused(x)
    replayed(x)  # Compiles and warms up; the next call records
    times = {}
    for desc, f in (('eager', deep), ('compiled', fused),
                    ('reduce-overhead', replayed)):
        t = times[desc] = d2l.Benchmark(lambda: f(x), repeats=100).time
        print(f'{desc}: {1000 * t:.2f} ms/call, '
              f'{1e6 * t / 120:.1f} µs per operation')
```

```
    # The kernels of the compiled stack, against the time of a call
    d2l.show_profile(d2l.kernel_times(lambda: fused(x)),
                     1000 * times['compiled'], rows=3)
```

```
eager: 2.14 ms/call, 17.8 µs per operation
compiled: 2.02 ms/call, 16.8 µs per operation
reduce-overhead: 0.28 ms/call, 2.4 µs per operation
wall clock 2.02 ms per step, device busy 0.31 ms (15%), matrix products 55%
  0.17 ms   55%  Kernel2<cutlass_80_tensorop_s1688gemm_64x64_16x6_tn_
  0.08 ms   25%  splitKreduce_kernel<32, 16, int, float, float, float
  0.06 ms   20%  triton_poi_fused_tanh_0
```

デフォルトの `torch.compile`（`fused`）は即時実行とほとんど速度が変わらない。Inductor は行列積を cuBLAS に委譲するため tanh をそこに融合できず、さらにすべてのカーネルを Python から 1 つずつ起動するままであるためだ（呼び出し時間のうち GPU が稼働しているのは約 15% にすぎない）。`mode='reduce-overhead'` によって CUDA グラフを有効にすると、一連の処理全体が 1 回の起動でリプレイされ、呼び出し時間は約 7× 短縮されて実際の GPU 実行時間である約 0.3 ms まで下がる。

`mode='reduce-overhead'` を 13.3.3 節の MLP 訓練ステップに適用すると、同様にステップ時間は 1 ms 未満へと短縮され、GPU は予測どおりの 0.25 ms 稼働し、残りの時間は `autograd.grad` のグラフブレークにおける Python 処理に費やされる。

```
replayed_step = torch.compile(train_step, mode='reduce-overhead')
t = d2l.Benchmark(lambda: replayed_step(net, X, Y)).time
busy = sum(d2l.kernel_times(lambda: replayed_step(net, X, Y)).values())
print(f'training step, reduce-overhead: {1000 * t:.2f} ms/call, '
      f'GPU busy {busy:.2f} ms')
```

```
training step, reduce-overhead: 0.71 ms/call, GPU busy 0.25 ms
```

13.3.5 コンパイルの限界

コンパイルが割に合うのは、プログラムが n^* ステップを超えて実行され、かつ帯域幅律速の演算列やホスト起動オーバーヘッドに多くの時間を費やしている場合に限られる。実務上、次の 3 つの注意点に留意しておこう。

- **テンソルの値に依存する分岐：**スカラーの損失のログ出力、`if loss < 0.1:` のような判定、データ依存のインデックス参照（`x[x > 0]`）は、グラフブレークやエラーを引き起こす。ホスト側での値の読み取りはコンパイル対象関数の外に出し、グラフ内の分岐は `where` / `select` や `cond` で表現する。
- **変化する形状：**可変の系列長やエポック末尾の短いミニバッチは、繰り返しの再コンパイルを引き起こすことがある。最後のミニバッチは切り捨てるかパディングし、可変長の系列はバケット化する。
- **デバッグ：**コンパイル済みカーネルは Python 上で 1 行ずつ実行されるわけではない。まず即時実行モードで動作検証とデバッグを済ませてからコンパイルする。

13.3.6 まとめ

即時実行は演算を 1 つずつ実行するため、カーネルごとにホストディスパッチのレイテンシ h を支払い、要素ごとの演算のたびに DRAM との往復が発生する。トレースは実行前にシンボリック

なプレースホルダ上で計算グラフをキャプチャする。その際、データ依存の分岐は選択・条件分岐プリミティブで書き換えなければグラフブレークを招き、形状の変化は次元をシンボリックにしない限り再コンパイルを引き起こす。演算融合は要素ごとの演算列やオンチップの行リダクションを単一のカーネルに統合し（`gelu_ish` のメモリトラフィックと実行時間を 9 分の 1 に削減する）、CUDA グラフのリプレイは小さなモデルやバッチにおけるカーネルごとのホスト起動オーバーヘッドを解消する。

13.3.7 演習問題

1. [code] **値に依存する分岐**。 13.3.1 節の関数 `f` は、`if` の箇所でグラフブレークを起こす。
   1. `f` を `fullgraph=True` でコンパイルし、エラーメッセージを確認せよ。次に分岐を `torch.cond` で書き換え、`torch._dynamo.explain` によって単一のグラフがキャプチャされることを確認し、`torch.cond` と `torch.where` を比較せよ。
   2. コンパイルした元の関数とコンパイルした `torch.where` 版について、`d2l.Benchmark` を用いて 8 要素の入力と 4000×4000 の入力での実行時間を計測し、各サイズでの違いを説明せよ。
   3. 両方の分岐を評価する `torch.where` のほうが、グラフブレークよりも高コストになるような 2 つの分岐の例を挙げて説明せよ。
2. [code] **形状の変化**。`torch.compile` は入力形状にガードを設定し、ガードが失敗すると再コンパイルする（13.3.1 節）。
   1. `torch.tanh(x) * 2 + 1` を `recording_backend` でコンパイルし、長さ $8, 16, 32, \dots, 8192$ のベクトルに対して呼び出せ。コンパイル回数を数え、`torch._logging.set_logs(recompiles=True)` で再コンパイルの理由を調べよ。
   2. `dynamic=False` を指定した場合と、最初の呼び出し前に `torch._dynamo.mark_dynamic(x, 0)` を呼んだ場合とで同じ実験を繰り返せ。それぞれ何回コンパイルが発生するか。また、`mark_dynamic` が望ましいのはどのような場合か。
3. [code] **行列積に続く融合**。●コンパイルは大規模な行列積をそのままにしつつ、直後に続く要素ごとの演算列を融合する。`gelu_ish(a @ a)` を $n \times n$ の bf16 行列 `a` に対して実行する場合を考える（GPU のピーク性能を P、帯域幅を β、リッジポイントを $I^* = P/\beta$ とする）。行列積を計算律速（$2n^3/P$ 秒）、後続の演算列を帯域幅律速（即時実行モードでは $m = 18$ 個、融合時は 2 個の配列を転送し、各配列は n^2 個の $b = 2$ バイト要素からなる）としてモデル化せよ。
   1. 即時実行に対するコンパイル実行の高速化率の上界 $S(n)$ を n, m, b, I^* で表せ。また、RTX 4090（$I^* \approx 164$）において $n = 1024, 4096, 16384$ での値を計算し、さらに n のうち $S(n) = 1.1$ となる値を求めよ。
   2. `gelu_ish(a @ a)` の即時実行時間とコンパイル実行時間を $n = 256, 512, \dots, 8192$ について計測し、実測された高速化率を $S(n)$ と重ねてプロットし、両者が乖離する領域を説明せよ。
4. **リダクションの融合**。●$R \times d$ の fp32 行列に対する数値的に安定な行ごとのソフトマックスは、行の最大値、減算、指数関数、行の和、除算という 5 つの即時実行演算として書ける。
   1. 5 つの即時実行カーネルと、各行を SRAM に保持する単一の融合カーネルのそれぞれについて、DRAM の読み出し回数と書き込み回数を数え（Rd に比例する項のみを残す）、帯域幅律速における高速化率を答えよ。
   2. 1 行が SRAM 容量を超える場合について、最大値と和を別々のパスで計算する融合カーネルと、$m_j = \max(m_{j-1}, x_j)$ および $s_j = s_{j-1} e^{m_{j-1} - m_j} + e^{x_j - m_j}$ を更新するオンラインの 1 パスリダクションを用いた融合カーネルのそれぞれに必要な最小パス数を求めよ。また、$s_d = \sum_{j=1}^{d} e^{x_j - m_d}$ が成り立つことを確かめよ。
5. [code] **オプティマイザのコンパイル**。●13.3.3 節の訓練ステップを `torch.optim.Adam`（`zero_grad`、`backward`、`step`）を用いて書き換えよ。
   1. `torch._dynamo.explain` でこのステップを調べよ。いくつのグラフがキャプチャされ、

どこでグラフブレークが発生するか。
2. SGD の 1 回の更新と Adam の 1 回の更新（9.5 節）で読み書きされるパラメータと同サイズの配列の最小個数をそれぞれ数え、両オプティマイザについて即時実行ステップとコンパイル済みステップの時間を計測し、更新処理のコンパイルによる改善幅が一方で大きくなる理由を説明せよ。

6. [code] **オーバーヘッドの交差点。**●13.3.4 節の 60 層スタックを用いて、カーネルリプレイの効果がなくなる境界を調べよ。
1. 幅 256 のまま深さ $L \in [10, 200]$ にわたって即時実行とリプレイ実行の時間を計測し、$t(L) = c + sL$ を当てはめて傾きを比較せよ。
2. $\max(\text{FLOPs}/P, \text{bytes}/\beta) = h$ から、1 層の行列積の実行時間と起動時間が等しくなる幅を予測せよ。次に深さ 60 で幅を 64 から 4,096 まで変えて即時実行とリプレイ実行の時間を計測し、両者が交差する幅を予測と比較せよ。
3. 即時実行のスタックを `d2l.kernel_times` と `d2l.show_profile` でプロファイルし、デバイスの稼働割合および起動 1 回あたりの実時間をコンパイル済みプロファイルと比較せよ。

13.4 メモリと精度

これまでの節では訓練ステップがどれほど高速に実行されるかを問うてきたが、そもそも GPU メモリに収まらないステップは実行すらできない。fp32 の Adam で訓練する 1B パラメータの GPT は、パラメータあたり 4 つの fp32 値（重み、その勾配、および Adam の 2 つのモーメント推定値）を保持するため、トークンを 1 つも読み込む前に 24 GiB の RTX 4090 のうち $16N \approx 15.1$ GiB を消費する。残りの約 9 GiB に、順伝播が誤差逆伝播法（backpropagation）のために保存する**活性化値**（activation）を収めなければならず、これにはトークンあたり数メガバイトを要する。その結果、1,024 トークンの系列は 2〜3 本しかカードに収まらないのに対し、大規模言語モデル（large language model）は 50 万〜数百万トークンのグローバルバッチで訓練される (Brown et al. 2020)。

訓練ステップのメモリは、N に比例する固定のパラメータ項と、トークン数 n に比例する活性化項からなる。本節では、この GPT について両者を数え上げ、実測ピークメモリと照合して検証したうえで、活性化項またはマイクロバッチサイズに働きかける 3 つの手法を学ぶ。すなわち、16 ビットで計算する**混合精度**（mixed precision）、活性化値を保存する代わりに逆伝播中に再計算する**活性化チェックポイント**（activation checkpointing）、および大きなグローバルバッチを逐次的なマイクロバッチに分割する**勾配累積**（gradient accumulation）である。これら 3 つのいずれも、$16N$ バイトのパラメータ状態を縮小することはない。その状態を複数の GPU に分割する方法は 13.7 節の主題である。

```
import os
import re
from torch.utils.checkpoint import checkpoint
# Detach PyTorch's profiler from the GPU when a profile ends; without this,
# every kernel launch after the first profile stays about 30% slower
os.environ['TEARDOWN_CUPTI'] = '1'
```

13.4.1 訓練ステップのメモリ

13.4.1.1 パラメータあたりのメモリ

パラメータメモリはオプティマイザに依存する。通常の fp32 SGD は重みと勾配を保持し（パラメータあたり 8 バイト）、モメンタム付き SGD はこれに速度を加え（12 バイト）、Adam は 2 つ

のモーメント推定値を保持する（**パラメータあたり 16 バイト**、5.7.3.2 節）。混合精度訓練では、これら 4 つすべてを fp32 で保持したうえで、ステップ中に 16 ビットの重みのコピーを保持する（**パラメータあたり 18 バイト**）。

基準とするモデルは `d2l.GPT`（11.2 節）であり、幅 $d = 2048$、16 ヘッド、$L = 18$ 個の Pre-Norm SwiGLU ブロック、語彙サイズ $V = 50{,}257$、および学習可能な位置埋め込みを備える。各ブロックは約 $12d^2 \approx 50$M パラメータを保持する（18 ブロック全体で約 0.9B、さらに重み共有されたトークン埋め込みに約 0.1B）。メタデバイス（形状のみを扱うデバイス）上でモデルを検査すれば、4 GB の重みを確保することなく N を確認できる（5.2.2.2 節）。

```
config = dict(vocab_size=50257, num_hiddens=2048, num_heads=16,
              num_blks=18, pos='learned')
with torch.device('meta'):  # Shapes and dtypes only, no data
    model = d2l.GPT(**config)
N = d2l.num_params(model)
print(f'{N / 1e9:.2f} billion parameters, 16N = {16 * N / 2**30:.1f} GiB')
```

```
1.01 billion parameters, 16N = 15.1 GiB
```

13.4.1.2 トークンあたりのメモリ

順伝播の間、各層は逆伝播で必要となる中間活性化値を保存し、逆伝播の間にはパラメータ勾配が生成されるにつれてこれらの活性化値が層ごとに解放される（Figure 13.14）。したがって、メモリは逆伝播の開始時点でピークに達する。

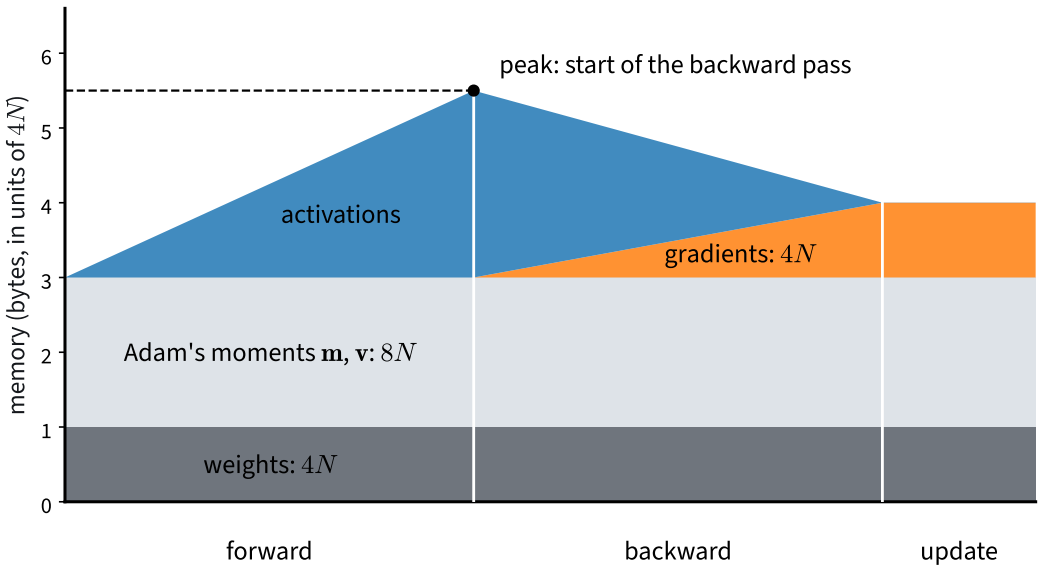


Figure 13.14: Adam を用いた訓練ステップのメモリ（$4N$ バイト単位）。重み（$4N$）と Adam のモーメント推定値（$8N$）はステップ全体を通して保持される。活性化値は順伝播中に蓄積され、逆伝播中に勾配（$4N$）が現れるにつれて解放される。メモリは逆伝播の開始時点でピークに達する。

Transformer ブロックが誤差逆伝播法のために何を保存するかは、次の 3 つの規則で決まる。

1. 線形層 $\mathbf{y} = \mathbf{W}\mathbf{x}$ は入力 $\mathbf{x}$ を保存する（$\nabla_{\mathbf{W}} = (\nabla_{\mathbf{y}})\mathbf{x}^\top$ であるため）。
2. 要素ごとの非線形性 f は入力または出力を保存する。
3. 要素ごとの積 $\mathbf{p} \odot \mathbf{q}$ は両方の因子を保存する。

幅 $d = 2048$ を単位としてトークンあたりで数え上げると、次のようになる。

- **アテンションサブ層：**アテンション前の RMSNorm が入力（d）を保存し、QKV 射影が正規化後の出力（d）を保存する。融合された因果アテンションは $\mathbf{Q}, \mathbf{K}, \mathbf{V}$（$3d$）とその出力（$d$、および $T \times T$ のスコア行列の代わりにヘッド・トークンあたりスカラーの対数正規化項のみ）を保存し、出力射影はアテンション出力を保存する（エイリアスであれば 0、ヘッド結合時にコピーされる場合は d）。合計は $6d$–$7d$ である。
- **SwiGLU FFN サブ層：**FFN 前の RMSNorm が入力（d）を保存し、ゲート射影とアップ射影が正規化後の出力（d）を共有する。さらに、$\mathrm{SiLU}(\mathbf{g}) \odot \mathbf{u}$ とダウン射影がゲート $\mathbf{g}$、活性化後のゲート $\mathrm{SiLU}(\mathbf{g})$、アップ射影 $\mathbf{u}$、およびそれらの積という 4 本のベクトル分（幅 $\frac{8}{3}d \approx 10.7d$）を保存する。合計は約 $12.7d$ である。

合計すると、融合された 1 ブロックはトークンあたり $18.7d$–$19.7d \approx$ **20d** 個の値を保存する（$d = 2048$ の fp32 で **160 KiB** であり、ゲートなしの GPT-3 ブロックに対する $16d$ という Korthikanti et al. (2022) の集計と比較できる）。融合アテンションは逆伝播中に $T \times T$ の重みをオンチップで再計算するため、この集計値の中に系列長 T に応じて増大するものは一切ない。最後に、交差エントロピー損失（cross-entropy loss）はトークンあたり $V = 50{,}257$ 個の fp32 対数確率（**196 KiB**）を保存し、これは Transformer ブロック 1 つ分全体よりも大きい。

この集計を実験的に検証するため、まず `saved_bytes` を定義し、`gpt_loss` の逆伝播のために保持されるテンソルの合計バイト数を求める。

```
def saved_bytes(f, *args):
    """Bytes of the tensors that autograd keeps for the backward pass of f."""
    storages = {}
    def pack(t):  # Called on every tensor saved for the backward pass
        storages[t.untyped_storage().data_ptr()] = (
            t.untyped_storage().nbytes())
        # A detached alias shares the storage. Returning t itself would form
        # a reference cycle between a saved output and the graph that made
        # it, and the graph, with all its tensors, would never be freed
        return t.detach()
    with torch.autograd.graph.saved_tensors_hooks(pack, lambda t: t):
        f(*args)
    return sum(storages.values())

def gpt_loss(model, X, bf16=False):
    """Cross-entropy of a GPT's predictions, with the inputs as targets."""
    with torch.autocast(X.device.type, torch.bfloat16, enabled=bf16):
        return F.cross_entropy(model(X).flatten(0, 1), X.flatten())
```

1 ブロックおよび 2 ブロックの GPT モデルに対してバッチサイズ $b \in \{1, 2\}$ で `saved_bytes` を評価することにより、`bytes_per_token` は単一の Transformer ブロックが要するトークンあたりの限界活性化メモリ（`block`）を、埋め込み・出力ヘッド・損失の分（`head`）から分離する。

```
def bytes_per_token(config, bf16=False, num_steps=1024):
    """Bytes stored per token by one block, and by the rest of a GPT."""
    gpu, per_token = d2l.try_gpu(), []
```

```
    for L in (1, 2):  # GPTs with one and with two blocks
        model = d2l.GPT(**{**config, 'num_blks': L}).to(gpu)
        X1, X2 = (torch.randint(0, config['vocab_size'], (b, num_steps),
                                device=gpu) for b in (1, 2))
        per_token.append((saved_bytes(gpt_loss, model, X2, bf16)
                          - saved_bytes(gpt_loss, model, X1, bf16))
                         / num_steps)
    return per_token[1] - per_token[0], 2 * per_token[0] - per_token[1]

block, head = bytes_per_token(config)
print(f'one block: {block / 2**10:.0f} KiB per token '
      f'({block / (4 * config["num_hiddens"]):.1f} fp32 values of width d)')
print(f'embedding, output layer, and loss: {head / 2**10:.0f} KiB per token')
```

```
one block: 149 KiB per token (18.7 fp32 values of width d)
embedding, output layer, and loss: 212 KiB per token
```

scaled_dot_product_attention、RMSNorm、および SiLU が単一の融合カーネルとして実行されるため、PyTorch は理論上の最小値であるブロックあたり $18.7d$ 個の値（149 KiB）と、埋め込み・出力射影・損失の 212 KiB をちょうど保存する。

13.4.1.3 メモリバジェット

パラメータ状態（マイクロバッチ間で勾配が持続する場合は $16N$ バイト）に、1 ブロック・1 トークンあたり a バイトと、埋め込み・出力ヘッド・損失のトークンあたり a_{out} バイトを合わせると、n トークンに対するステップのメモリバジェットが得られる。

$$M(n) \approx 16N + n\,(La + a_{\text{out}}) \;\le\; M_{\text{GPU}}. \tag{13.6}$$

24 GiB のカード上の 18 ブロック GPT（$16N = 15.1$ GiB であり、活性化値には 8.9 GiB が残る）について、生の集計値は次のように予測する。

トークンあたり $La + a_{\text{out}} \approx 2.8$ MiB（約 3,200 トークン、すなわち長さ 1,024 の系列 3 本分）である。

13.4.1.4 実測ピークメモリ

保存された活性化値に加えて逆伝播が一時的に何を確保するかを調べるため、2 ブロック版の GPT において 1,024〜8,192 トークンにわたり、重みを除いたピークメモリを計測する。

```
gpu, V = d2l.try_gpu(), config['vocab_size']
model = d2l.GPT(**{**config, 'num_blks': 2}).to(gpu)
gpt_loss(model, torch.randint(0, V, (1, 1024), device=gpu)).backward()
peaks = {}  # The step above was a warm-up
for n in (1024, 2048, 4096, 8192):  # Tokens per step, in sequences of 1,024
    X = torch.randint(0, V, (n // 1024, 1024), device=gpu)
    model.zero_grad(set_to_none=True)
    peaks[n] = d2l.peak_memory(lambda: gpt_loss(model, X).backward())
    print(f'{n:5d} tokens: peak {peaks[n] / 2**30:.2f} GiB above the weights')
for n in (2048, 4096):  # Growth per token from n to 2n tokens
    slope = (peaks[2 * n] - peaks[n]) / n
```

```
    print(f'{n:5d} to {2 * n:5d} tokens: measured {slope / 2**10:.0f} KiB '
          f'per token; counted {(2 * block + head) / 2**10:.0f} KiB')
```

```
1024 tokens: peak 1.53 GiB above the weights
2048 tokens: peak 1.77 GiB above the weights
4096 tokens: peak 3.53 GiB above the weights
8192 tokens: peak 7.06 GiB above the weights
2048 to  4096 tokens: measured 905 KiB per token; counted 511 KiB
4096 to  8192 tokens: measured 903 KiB per token; counted 511 KiB
```

サイズが小さいうちは固定オーバーヘッドがまだ現れる。それが無視できるようになると、ピークはトークン数に比例して増加し、このトークンあたりの増加量こそが集計値と比較すべき数値となる。

実測ピークはトークンあたり約 905 KiB ずつ増加し、集計した 511 KiB より 393 KiB 大きい。この超過分は、順伝播の活性化値がまだすべて生存している逆伝播の開始時点で生じる。すなわち、交差エントロピーの逆伝播が対数確率に関する勾配とロジットに関する勾配という、トークンあたり $V = 50{,}257$ 個の fp32 値からなる 2 つの一時テンソルを実体化するためである（保存テンソルの集計値に対してトークンあたり $8V \approx 393$ KiB の超過）。

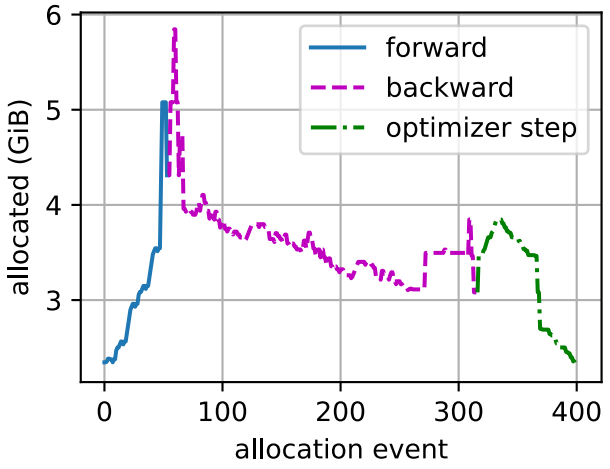
PyTorch のアロケータ履歴 API を使えば、4,096 トークンに対する Adam ステップ全体のメモリ確保タイムラインを可視化できる。

```
X = torch.randint(0, V, (4, 1024), device=gpu)
opt = torch.optim.Adam(model.parameters())
gpt_loss(model, X).backward()  # A first step creates the moment estimates
opt.step()
opt.zero_grad(set_to_none=True)

# _snapshot, _record_memory_history and the keys of their records are a
# private API of PyTorch 2.11 and may change between versions
events = lambda: torch.cuda.memory._snapshot()['device_traces'][0]
start = torch.cuda.memory_allocated()
torch.cuda.memory._record_memory_history()  # Record from here on
loss = gpt_loss(model, X)
forward_end = len(events())
loss.backward()
backward_end = len(events())
opt.step()
opt.zero_grad(set_to_none=True)
trace = events()
torch.cuda.memory._record_memory_history(enabled=None)
```

符号付きのバイト変化量（`alloc` から `free_completed` を引いたもの）を `trace` 全体にわたって累積し、`forward_end` と `backward_end` で区切ることで、ステップの 3 つのフェーズにわたる生存メモリの軌跡をプロットする。

```
sign = {'alloc': 1, 'free_completed': -1}  # Other actions change nothing
sizes = torch.tensor([sign.get(e['action'], 0) * e['size'] for e in trace])
series = ((start + sizes.cumsum(0)) / 2**30).tolist()
bounds = [0, forward_end, backward_end, len(trace)]
phases = list(zip(bounds[:-1], bounds[1:]))
```

```
d2l.plot([list(range(a, b)) for a, b in phases],
         [series[a:b] for a, b in phases], 'allocation event',
         'allocated (GiB)', legend=['forward', 'backward', 'optimizer step'])
```



トレースは $12N$（重みと 2 つの Adam モーメント）から始まり、順伝播中にブロックを進むにつれて線形に上昇し、ロジットおよび逆伝播の開始時点でスパイクしたのち、活性化値が解放されて勾配が確保されるにつれて段階的に減少していく。

13.4.2 混合精度

混合精度訓練（5.5 節）では、マスター重み（master weights）、勾配、オプティマイザのモーメント、正規化、残差ストリーム（residual stream）、および損失を fp32 に保ちながら、行列積と中間活性化値を 16 ビット浮動小数点で実行する。これには、fp32 と同じ 8 ビットの指数部範囲を共有し損失スケーリング（loss scaling）を必要としない bf16、あるいは損失スケーリングを伴う fp16（5.5.3.3 節）を用いる。

ブロック内の正規化に入力される 2 つの残差ストリームは fp32（4 バイト）のままである一方、残りの 18 個分（ブロックあたりに保存される約 $20d$ 個の値のうち）は bf16（2 バイト）に下がるため、1 ブロックが保存する活性化値は fp32 のサイズの約 $(2 \cdot 4 + 18 \cdot 2)/(20 \cdot 4) = 0.55$ 倍に縮小する。

```
block16, head16 = bytes_per_token(config, bf16=True)
print(f'one block: {block16 / 2**10:.0f} KiB per token in bf16, '
      f'{block16 / block:.2f} of fp32')
print(f'embedding, output layer, and loss: {head16 / 2**10:.0f} KiB per token')
```

```
one block: 87 KiB per token in bf16, 0.58 of fp32
embedding, output layer, and loss: 306 KiB per token
```

1 ブロックあたりの活性化バイト数は fp32 の 0.58–0.60 倍に減少する。しかし、ステップ中に重みの bf16 コピーを保持することで、固定のパラメータメモリが $16N = 15.1$ GiB から $18N = 17.0$ GiB へと増加し、活性化値には 7.0 GiB しか残らなくなる。活性化値の節約分と追加の重みコピーの分は大部分が相殺し合うため、24 GiB カード上での最大トークン容量は 15〜27% しか拡大しない。

bf16 混合精度の主な利点は容量よりも速度にある。RTX 4090 では、bf16 の Tensor Core は 165 TFLOP/s で動作し、tf32 および fp32 の 82.6 TFLOP/s の 2 倍に達する。Tensor Core の効果（tf32 対スカラー fp32）と 16 ビットオペランドの効果（bf16 対 tf32）を切り分けるため、計算律速（compute-bound）となる幅 8,192 の 3 層 MLP において、2,048 個の例からなるバッチに対する SGD の 1 ステップをベンチマークする。このステップは 1.1×10^{12} FLOPs を要し、ルーフライン下限は 82.6 TFLOP/s で 13.3 ms、165 TFLOP/s で 6.7 ms であり、これに帯域幅律速の SGD 更新の約 1.3 ms が加わる。

```
sizes = [(64 * 64, 8192), (8192, 8192), (8192, 10)]  # (inputs, outputs)
mlp = nn.Sequential(nn.Linear(*sizes[0], bias=False), nn.ReLU(),
                    nn.Linear(*sizes[1], bias=False), nn.ReLU(),
                    nn.Linear(*sizes[2], bias=False)).to(gpu)
for layer in mlp[::2]:  # Weights drawn from N(0, 0.02^2)
    nn.init.normal_(layer.weight, std=0.02)
X_mlp = torch.randn(2048, 64 * 64, device=gpu)
y_mlp = torch.randint(0, 10, (2048,), device=gpu)
opt = torch.optim.SGD(mlp.parameters(), lr=0.01)

def step(bf16):
    with torch.autocast(X_mlp.device.type, torch.bfloat16, enabled=bf16):
        l = F.cross_entropy(mlp(X_mlp), y_mlp)
    l.backward()
    opt.step()
    opt.zero_grad(set_to_none=True)
```

次に、MLP の順伝播および逆伝播の行列積について正確な積和演算数（`macs`）と合計 `flops` を数え、各精度形式をミリ秒と TFLOP/s でベンチマークする関数 `report` を定義する。

```
# Two FLOPs per multiply-add, in the forward pass, the weight gradients, and
# the input gradients of every layer but the first
macs = sum(i * o for i, o in sizes)  # Per example and pass
flops = 2 * len(X_mlp) * (3 * macs - sizes[0][0] * sizes[0][1])
print(f'{flops:.2e} FLOPs per step')

def report(desc, f):
    """Print the time per call of f and the arithmetic rate it achieves."""
    t = d2l.Benchmark(f).time
    print(f'{desc}: {1000 * t:5.2f} ms, {flops / t / 1e12:3.0f} TFLOP/s')
```

```
1.10e+12 FLOPs per step
```

それでは `report` を、厳密なスカラー演算の `'fp32'`、Tensor Core を用いる `'tf32'`、および混合精度の `'bf16'` にわたって実行しよう。

```
torch.set_float32_matmul_precision('highest')  # True fp32
report('fp32', lambda: step(False))
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
report('tf32', lambda: step(False))
report('bf16', lambda: step(True))
```

```
fp32: 23.15 ms,  48 TFLOP/s
tf32: 15.01 ms,  73 TFLOP/s
```

```
bf16:  9.68 ms, 114 TFLOP/s
```

スカラー演算器上の真の fp32 は、積和演算とアドレス計算のすべてが命令発行スロットを奪い合うため、ピークである 82.6 TFLOP/s の約 60% にとどまる。一方、Tensor Core 上の tf32 は同じピークの約 90% に達し、保存される型を変えることなくステップ時間を 3 分の 1 短縮する。さらに tf32 から bf16 へ移行すると、ステップはもう 1.5–1.75× 高速化する（fp32 の SGD 更新と重みのキャストは縮小しないため、下限と更新時間を合わせた $6.7 + 1.3 = 8.0$ ms に近づく）。

13.4.3 活性化チェックポイント

カードに収まるトークン数を何倍にも増やすには、1 ステップで保存する活性化値を減らさなければならない。**活性化チェックポイント**（5.7.3.3 節、(Chen et al. 2016)）は、順伝播中に各 Transformer ブロックの入力（トークンあたり d 個の fp32 値、すなわち $4d$ バイト）のみを保存し、逆伝播中にブロック内部の活性化値を 1 ブロックずつ再計算する。これにより、余分な順伝播 1 回分（約 33% の計算量増であり、トークンあたり $8N$ FLOPs と $6N$ から増加）と引き換えに、L ブロックにわたるトークンあたりのピークメモリを $La + a_{\text{out}} + \Delta$ から高々次の値まで削減する。

$$L \cdot 4d + a + a_{\text{out}} + \Delta,$$

ここで、a は現在再計算されている単一ブロックの分であり、Δ は集計値に対する実測ピークの超過分である（13.4.1.4 節）。

```
M_gpu, d, L = 24 * 2**30, config['num_hiddens'], config['num_blks']
slope = (peaks[8192] - peaks[4096]) / 4096  # Growth per token, measured
excess = slope - (2 * block + head)  # Measured peak above the count
budgets = {  # Configuration: (fixed bytes, bytes per token at the peak)
    'fp32, count': (16 * N, L * block + head),
    'fp32, measured': (16 * N, L * block + head + excess),
    'bf16, measured': (18 * N, L * block16 + head16 + excess),
    'fp32, checkpointing': (16 * N, L * 4 * d + block + head + excess),
    'bf16, checkpointing': (18 * N, L * 4 * d + block16 + head16 + excess)}
for name, (fixed, per_token) in budgets.items():  # M(n) <= M_gpu
    print(f'{name:19s}: {per_token / 2**20:.2f} MiB per token, room for '
          f'{(M_gpu - fixed) / per_token:6,.0f} tokens')
```

```
fp32, count        : 2.83 MiB per token, room for  3,229 tokens
fp32, measured     : 3.22 MiB per token, room for  2,844 tokens
bf16, measured     : 2.21 MiB per token, room for  3,272 tokens
fp32, checkpointing: 0.88 MiB per token, room for 10,435 tokens
bf16, checkpointing: 0.91 MiB per token, room for  7,956 tokens
```

ブロックごとのチェックポイントを用いると、18 ブロック全体で保存されるのはトークンあたりわずか $18 \times 8 = 144$ KiB（出力層と損失だけの分よりも小さい）となり、トークンバジェットは 3 倍以上（10,000〜15,000 トークン）に拡大する。なお、チェックポイント下では bf16 の方が fp32 よりも収まるトークン数が実際に**少なく**なることに注目したい。bf16 の重みコピーによる追加の $2N$ バイトが、（PyTorch では）より大きくなる bf16 の出力層および損失と相まって、再計算される単一ブロックでの活性化値の節約分を上回るためである。この段階では、出力ロジットと交差エントロピー損失がトークンあたりのメモリを支配するようになる。出力射影と損失を独自のチェックポイント内でトークンのチャンクごとに計算すれば、わずか $2dV/(6N) \approx 3\%$ の追加 FLOPs でそのボトルネックを解消できる（演習問題 6）。

長さ 1,024 トークンの系列 2 本を用いて、4 ブロックのスタック上でブロックごとのチェックポイントを計測してみよう（4 つのブロック入力と最終出力を合わせて、トークンあたり $5 \times 4d = 40$ KiB が保存されるはずである）。

```
blocks = d2l.GPT(**{**config, 'num_blks': 4}).to(gpu).blks
def stack(H, remat):
    for blk in blocks:
        H = checkpoint(blk, H, use_reentrant=False) if remat else blk(H)
    return H.square().mean()

d = config['num_hiddens']
H1, H2 = (torch.randn(b, 1024, d, device=gpu, requires_grad=True)
          for b in (1, 2))
for remat in (False, True):
    stored = (saved_bytes(stack, H2, remat)
              - saved_bytes(stack, H1, remat)) / 1024
    stack(H2, remat).backward()  # Warm-up; allocates the gradients
    peak = d2l.peak_memory(lambda: stack(H2, remat).backward())
    t = d2l.Benchmark(lambda: stack(H2, remat).backward()).time
    print(f'checkpointing={remat!s:5}: stores {stored / 2**10:4.0f} KiB per '
          f'token, peak {peak / 2**30:.2f} GiB, {1000 * t:.0f} ms per step')
```

```
checkpointing=False: stores  606 KiB per token, peak 1.25 GiB, 47 ms per step
checkpointing=True : stores   40 KiB per token, peak 0.42 GiB, 59 ms per step
```

チェックポイントにより、保存されるテンソルは予測どおりちょうどトークンあたり 40 KiB に減少し、ピークの活性化メモリは大幅に削減され、ステップ時間は約 25% 延びる（余分な順伝播 1 回分の上限である 33% 以内に収まっている）。ブロック全体を再計算する代わりに、**選択的再計算**（selective recomputation）(Korthikanti et al. 2022) は高コストな行列積の出力を保存し、SwiGLU のゲート積のような低コストな要素ごとの活性化値のみを再計算する（演習問題 5）。

13.4.4 勾配累積

より多くの活性化値を保存することなく数十万トークンのグローバルバッチ $B_{\text{global}} = m\, B_{\text{micro}}$ に到達するため、**勾配累積**はグローバルバッチを m 個の逐次的なマイクロバッチに分割し、それらの勾配を累積して（各マイクロバッチの平均損失を m で割る）、最後にオプティマイザを 1 回実行する。

勾配累積は単一の大きなバッチと同じ算術演算を実行するが、2 番目以降の各マイクロバッチは累積中の勾配アキュムレータと新しい勾配を読み出してその和を書き戻す（$12N$ バイトのメモリトラフィック）一方、マイクロバッチ自体は $6Nn$ FLOPs の演算を n トークンに対して実行する。ピーク計算時間に対する比として、このアキュムレータのトラフィックは次のコストを要する。

$$\frac{12N/\beta}{6Nn/P} = \frac{2I^*}{n},$$

ここで、$I^* = P/\beta$ はリッジポイント（ridge point、13.1 節）である。tf32（$I^* = 82$）でマイクロバッチが $n = 1{,}024$ トークンの場合、そのオーバーヘッドは累積を行う $m - 1$ 個の各マイクロバッチで約 16% となるが、$n = 4{,}096$ トークンでは 4% に低下する。

2 ブロックの GPT において、4,096 トークンを 1 バッチとして処理する場合と、1,024 トークンのマイクロバッチ $m = 4$ 個として処理する場合とで、数値的な等価性ならびにメモリと時間の

オーバーヘッドの両方を検証する。

```
X = torch.randint(0, V, (4, 1024), device=gpu)  # 4,096 tokens per step

def accumulate(model, X, m):
    """Gradient of the loss on X in .grad, accumulated over m micro-batches."""
    model.zero_grad(set_to_none=True)
    for Xb in X.chunk(m):
        (gpt_loss(model, Xb) / m).backward()  # / m: the loss averages

accumulate(model, X, 1)
full = [p.grad.clone() for p in model.parameters()]  # One-batch gradient
for m in (1, 4):
    model.zero_grad(set_to_none=True)
    peak = d2l.peak_memory(accumulate, model, X, m)
    assert max((p.grad - g).norm() / g.norm()
               for p, g in zip(model.parameters(), full)) < 1e-2
    f = lambda: accumulate(model, X, m)
    times = 1000 * np.array([d2l.Benchmark(f).time for _ in range(5)])  # ms
    kernels = d2l.kernel_times(f)        # Device ms per step, per kernel
    busy = sum(kernels.values())
    products = sum(ms for name, ms in kernels.items()
                   if re.search(d2l.PRODUCT_KERNELS, name, re.I))
    print(f'{m} x {4096 // m:4d} tokens: peak {peak / 2**30:.2f} GiB above '
          f'the weights, {np.median(times):.0f} ms per step '
          f'({times.min():.0f} to {times.max():.0f}), device busy '
          f'{busy:.0f} ms\n{"":16s}matrix products {products:.0f} ms, '
          f'other kernels {busy - products:.0f} ms')
```

```
1 x 4096 tokens: peak 3.53 GiB above the weights, 83 ms per step (83 to 83), device busy 83 ms
                 matrix products 69 ms, other kernels 13 ms
4 x 1024 tokens: peak 1.92 GiB above the weights, 99 ms per step (99 to 99), device busy 99 ms
                 matrix products 74 ms, other kernels 25 ms
```

重みを除いたピークメモリは約半分に低下し（$4N = 0.77$ GiB の持続的な勾配がマイクロバッチ間で確保されたまま残る一方、活性化メモリが 4 分の 1 に減少するため）、アキュムレータの加算および小さなマイクロバッチでの Tensor Core 効率のわずかな低下により、ステップ時間は 10〜20% 増加する。したがって、最適なマイクロバッチは常にメモリに収まる最大のものとなる。

13.4.5 メモリ手法の選択

Table 13.4 は、それぞれの手法を必要とする症状別にメモリ手法をまとめたものである。

Table 13.4: 症状別のメモリ手法、ならびに Equation 13.6 への効果とコスト。

症状	手法	メモリへの効果	コスト
行列積がステップ時間を支配	bf16 混合精度	活性化値 $\times 0.6$、重みコピーにより固定状態 $+2N$	8 ビット仮数部の丸め（fp16 は損失スケーリングも必要）
有用なマイクロバッチが収まらない	活性化チェックポイント	ブロックあたりの保存活性化値をトークンあたり a から $4d$ バイトへ削減	余分な順伝播 1 回分（時間が約 25〜33% 増加）

症状	手法	メモリへの効果	コスト
出力層と損失が支配	チャンク化交差エントロピー損失	a_{out} と Δ を一度に 1 トークンチャンク分に限定	出力射影の余分なパス 1 回分（FLOPs が約 3% 増加）
グローバルバッチがマイクロバッチ容量を超過	勾配累積	なし（グローバルバッチが m 倍に拡大）	アキュムレータのトラフィック $2I^*/n$、マイクロバッチごとの重み再読み出し
$16N$ のパラメータ状態だけでカードが埋まる	重み・勾配・オプティマイザ状態のシャーディング（13.7 節）	各 GPU 上で $16N/k$（k 基に分割）	GPU 間の集団通信

単一の RTX 4090 上の 1B パラメータ GPT では、bf16 混合精度によりチェックポイントなしで 1,024 トークンの系列が 2〜3 本収まり、256 個のマイクロバッチを累積することで 524,288 トークンのグローバルバッチに到達する。しかし、トークンあたり $6N$ FLOPs では、1 回のグローバルステップに 3.2×10^{15} FLOPs を要し、カード 1 枚ではステップあたり少なくとも 19 s、100,000 ステップ（52B トークン）では 3 週間以上かかる。単一の GPU にこの GPT の 1 ステップを収めることはできても、訓練を完走するには複数の GPU が必要となる（13.5 節）。

13.4.6 まとめ

訓練ステップのメモリは、固定のパラメータ項（fp32 の Adam では $16N$ バイト、bf16 の重みコピーを伴う場合は $18N$）と、トークンあたりの活性化項に分かれる。3 つの微分規則（線形層は入力を保持し、要素ごとの関数は入力または出力を保持し、積は両方の因子を保持する）により、融合された SwiGLU Transformer ブロックはトークンあたり約 $20d$ 個の値を保存し、交差エントロピー損失は V 個の対数確率を保存する。実際のピークは損失部分での逆伝播の一時配列によってこの集計値を上回り得るため、実測が必要となる。混合精度は Tensor Core 上での行列積を高速化する一方で総メモリ容量をほとんど変化させず、活性化チェックポイントは約 25% の追加時間と引き換えにブロック入力（トークンあたり $4d$ バイト）のみを保存し、勾配累積はメモリに収まるマイクロバッチから大きなグローバルバッチを構築し、$2I^*/n$ の割合でアキュムレータのトラフィックを追加する。

13.4.7 演習問題

1. **ピークが生じる位置**。@fig-memory-anatomy では、訓練ステップのピークを逆伝播の開始時点に置いている。n 個の同一な層からなるスタックを考え、各層のパラメータが w バイトを占有し、その順伝播が a バイトの活性化値を保存するものとする。逆伝播の開始時点では勾配は存在せず、各層の逆伝播はその a バイトの活性化値を解放して w バイトのパラメータ勾配を確保するものと仮定する。
    1. 逆伝播が j 個の層（全 n 層中）を処理した後に活性化値と勾配が保持しているバイト数を書き下せ。また、ピークが $a > w$ のときは逆伝播の開始時点にあり、$a < w$ のときはその終了時点にあることを示せ。
    2. 本節の GPT の 1 ブロックは約 $12d^2$ 個のパラメータを保持し、トークンあたり約 $20d$ 個の fp32 値を保存する。ステップあたりのトークン数がいくつを超えると、ピークが逆伝播の開始時点にくるか。
    3. m 個のマイクロバッチにわたる勾配累積のもとでは、最初のマイクロバッチの勾配が後続のマイクロバッチの実行中も確保されたまま残る。勾配累積を用いた場合のピークを求め、単一のフルバッチステップよりもピークを低くするような m の値を求めよ。

2. [code] **オプティマイザステップのメモリ。**13.4.1.4 節の 2 ブロック GPT を、`torch.optim.Adam` の 3 つの設定（`foreach=False`、デフォルト、`fused=True`）で構築せよ。
   1. 各オプティマイザで 2 ステップ実行せよ。2 ステップ目において、`opt.step()` の実行中に追加で確保されるピークバイト数を勾配サイズ $4N$ を単位として計測せよ。
   2. パラメータテンソルを各実装がどのように更新するか（一度に 1 テンソルずつ、バッチ化された複数テンソルの要素ごとの演算、または単一の融合カーネル）に基づいて、3 つの計測結果を説明せよ。オプティマイザステップが訓練ステップ全体のピークを決定し得るのはどのような場合か。
3. **勾配累積下でのクリッピング。**ノルム τ への勾配クリッピングは、$\mathbf{g}$ を $\mathbf{g}\min(1, \tau/\|\mathbf{g}\|)$ に置き換える。$\mathbf{g}_1, \dots, \mathbf{g}_m$ を $1/m$ 倍にスケールされたマイクロバッチ勾配とし、その和 $\mathbf{g}$ がグローバルバッチ勾配であるとする。
   1. ループ内で各 $\mathbf{g}_i$ をアキュムレータに加算する前にクリッピングする場合、累積された勾配が到達し得る最大ノルムはいくらか。また、マイクロバッチごとのクリッピングがグローバルなクリッピングと一致するのはどのようなときか。
   2. 係数 S の fp16 損失スケーリングのもとでは、最後のマイクロバッチの後にアキュムレータは $S\mathbf{g}$ を保持している。ループは S によるアンスケール、クリッピング、およびオプティマイザの更新ステップをどの順序で実行しなければならないか。また、マイクロバッチ間で S が変化すると何が問題になるか。
4. [code] **fp16 の勾配ウィンドウ。**●fp16 は 5 ビットの指数部を持つため、2^{-14}〜65,504 の正規化数と 2^{-24} までの非正規化数を表現できる。損失スケーリング (Micikevicius et al. 2018) は、`backward()` の前に損失に S を掛け、その後で勾配を S で割る。真の fp32 による参照勾配とともに、`mlp`、`X_mlp`、`y_mlp`（13.4.2 節）を用いて以下を行え。
   1. 参照勾配のすべての非ゼロ要素にわたって $\log_2 |g|$ のヒストグラムをプロットし、スケール係数 S として、スケール後の要素 Sg の 99.9% を fp16 の正規化数の範囲内に収めるものを求めよ。
   2. $S = 2^0, 2^2, \dots, 2^{24}$ に対して、`torch.autocast('cuda', dtype=torch.float16)` のもとで勾配を計算し、S でアンスケールして相対誤差 $\|\hat{\mathbf{g}} - \mathbf{g}\|/\|\mathbf{g}\|$ をプロットせよ。`torch.bfloat16` でも同様に行え。S のどの範囲で fp16 は bf16 よりも高精度になるか、またそれはなぜか。
   3. バッチサイズが 2,048 から 32,768 に増えると、有効な S の範囲はどのようにシフトするか。
5. [code] **選択的再計算。**●ブロックごとのチェックポイントは各ブロックの入力のみを保存する。`torch.utils.checkpoint.create_selective_checkpoint_contexts` を `context_fn` 引数経由で `checkpoint` に渡すと、選択した演算（`torch.ops.aten.mm.default` など）の出力をキャッシュし、残りを再計算する。
   1. 本節の GPT の 1 ブロックについて、3 つのポリシー（ブロックごとの完全なチェックポイント、すべての `mm` 出力のキャッシュ、および FFN のゲート射影とアップ射影を除くすべての `mm` 出力のキャッシュ）のもとで、保存されるテンソルと再計算される行列積を列挙せよ。
   2. 後半の 2 つのポリシーを 13.4.3 節の 4 ブロックスタックに実装し、トークンあたりの保存バイト数、ピークメモリ、およびステップ時間を比較せよ。
6. [extended] **チャンク単位の損失計算。**●●ブロックごとのチェックポイントのもとでは、本節の GPT の出力層と交差エントロピー損失は、損失がトークンあたり $V = 50{,}257$ 個の対数確率とその勾配を実体化するため、全 18 ブロックの合計よりもトークンあたり多くのメモリを消費する。
   1. トークンをサイズ $n_c = 1024$ のチャンクごとに処理するよう出力射影と交差エントロピー損失を記述し、ロジットと対数確率が一度に 1 チャンク分しか存在しないよう各チャンクをチェックポイントでラップせよ。損失と勾配がチャンク化しない版と一致することを検証せよ。
   2. ブロックごとのチェックポイントとチャンク化損失を用いた場合の、24 GiB カード

上での 18 ブロック GPT の最大マイクロバッチサイズを予測せよ。また、2 ブロック GPT 上でトークンあたりのメモリ増加率とステップ時間を計測せよ。

13.5 第一原理からのマルチ GPU

これまでの訓練実行ではすべて 1 基の GPU しか使ってこなかったが、本書のビルド用ワークステーションには 4 基搭載されている。k 基すべての GPU を使う最も単純な方法は**データ並列**（data parallelism）である。すなわち、すべての GPU にモデルの完全なレプリカと各ミニバッチの $1/k$ を与え、得られた勾配を **allreduce** によって GPU 間で合計し、すべてのカードで同じパラメータ更新を適用する。

このワークステーションでは、GPU 間通信はホスト DRAM を経由して中継される PCIe 4.0 を通るため（13.2.4.2 節）、各リンクは公称 32 GB/s に制限される。これはカード上のオンボードメモリの 1 TB/s より約 $30\times$ 遅い。2 基目の GPU がステップを短縮できるのは、引き受けた計算にかかる時間が、新たに加わる勾配交換（およびホストディスパッチのオーバーヘッド）を上回る場合に限られる。本節では、データ並列を手作業で構築し、集団通信ライブラリで使われる**リング allreduce** アルゴリズムを導出し、個別に測定した定数からマルチ GPU のステップ時間を予測するコストモデルを構築する。このモデルの検証には Fashion-MNIST 上の LeNet を用いる。そのステップはごく小さいため、ホストや通信の固定オーバーヘッドが鮮明に現れる。

```
import os
import time
# Detach PyTorch's profiler from the GPU when a profile ends; without this,
# every kernel launch after the first profile stays about 30% slower
os.environ['TEARDOWN_CUPTI'] = '1'
```

PyTorch では、固定形状のカーネルが定常状態の速度で動作するよう、tf32 の行列積と cuDNN の畳み込み自動チューニングも有効にしておく。

```
torch.set_float32_matmul_precision('high')
torch.backends.cudnn.benchmark = True  # Time convolution algorithms per shape
```

13.5.1 3 つの分割方法

モデルとそのミニバッチを複数のデバイスに分割する基本的な方法は 3 つある（Figure 13.15）。

- **データ並列**は、すべてのデバイスにモデルを複製し、ミニバッチをデバイス間で分割して、ステップごとに 1 回だけ勾配を合計する。単一 GPU に収まる任意のモデルで動作するが、各デバイスは依然として fp32 Adam の訓練状態 $16N$ バイトすべてを保持する（13.4 節）。
- **パイプライン並列**（pipeline parallelism）は、層のスタックをデバイス間で分割し、ステージ境界で順方向に活性化値を、逆方向に勾配を受け渡す（演習問題 1）。
- **テンソル並列**（tensor parallelism）は、個々の重み行列をデバイス間で分割し（2 基の 3 GB GPU にまたがる 2012 年の AlexNet のように (Krizhevsky et al. 2012)）、層ごとに複数回通信する（演習問題 3、13.7 節）。

13.5.2 手作業によるデータ並列

k 個のデバイス上でのデータ並列の 1 ステップは 4 つのステージからなる（Figure 13.16）。ミニバッチを k 個の均等なシャードに分割し、各シャードの勾配をそれぞれのデバイス上で計算し、

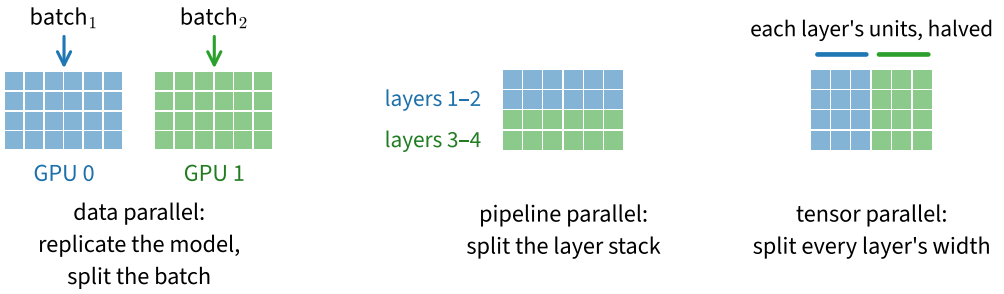


Figure 13.15: モデルを 2 つのデバイスに分割する 3 つの方法（各デバイスのセルを固有の色で示す）。モデルを複製してバッチを分割する（データ並列）、層のスタックを分割する（パイプライン並列）、各層の幅を分割する（テンソル並列）。

すべてのデバイスがその和を保持するよう allreduce で k 個の勾配を合計し、すべてのデバイスで同一のパラメータ更新を適用する。すべてのレプリカが同一のパラメータから開始して同一の更新を適用するため、重みを再コピーしなくても同期された状態が保たれる。

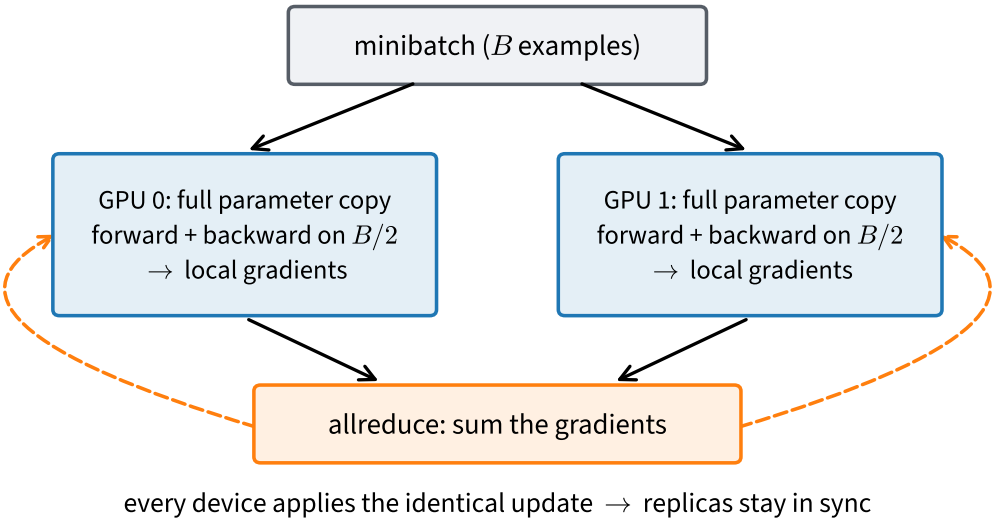


Figure 13.16: 2 基の GPU 上でのデータ並列の 1 ステップ。ミニバッチを分割し、各レプリカで同一の順伝播と逆伝播を実行し、allreduce で勾配を合計してから、すべてのデバイスで同一の更新を適用する。

13.5.2.1 モデルとデータ

デバイス間のコピーがすべて見えるように、LeNet（6.6 節、ReLU 活性化関数と He 初期化を使用）をパラメータテンソルの明示的なリストを受け取る関数として記述する。その 61,706 個の fp32 パラメータは 247 KB、Fashion-MNIST の 60,000 枚の訓練画像は 188 MB に収まる。これはデータセットのコピーを各 GPU 上に保持できるほど小さいため、ホストからのデータ読み込みが計測時間に入り込むことはない。

```
def new_params():
    """LeNet's parameters on the CPU: He-initialized weights, zero biases."""
    params = []
    for shape in [(6, 1, 5, 5), (16, 6, 5, 5), (120, 400), (84, 120),
                  (10, 84)]:
        fan_in = math.prod(shape[1:])
        params += [torch.randn(shape) * math.sqrt(2 / fan_in),
                   torch.zeros(shape[0])]
    return params

def lenet(params, X):
    W1, b1, W2, b2, W3, b3, W4, b4, W5, b5 = params
    H = F.avg_pool2d(F.relu(F.conv2d(X, W1, b1, padding=2)), 2)
    H = F.avg_pool2d(F.relu(F.conv2d(H, W2, b2)), 2).flatten(1)
    H = F.relu(F.linear(H, W3, b3))
    H = F.relu(F.linear(H, W4, b4))
    return F.linear(H, W5, b5)

def to_device(split, device):
    """A whole Fashion-MNIST split on `device`: images in [0, 1], labels."""
    return split.data.to(device).unsqueeze(1) / 255, split.targets.to(device)
```

少なくとも 2 基の GPU が利用可能であることを確認し、Fashion-MNIST の訓練分割を両方のデバイスに複製したうえで（train_data）、以降のすべての実験で共有する単一のパラメータ初期化 base を生成する。

```
assert d2l.num_gpus() >= 2, 'This section needs two GPUs'
data = d2l.FashionMNIST()
devices = [d2l.try_gpu(i) for i in range(2)]
train_data = [to_device(data.train, d) for d in devices]  # A copy each
test_data = to_device(data.val, devices[0])
base = new_params()          # One initialization for every run below
print(f'{sum(p.numel() for p in base)} parameters in {len(base)} tensors; '
      f'images {tuple(train_data[0][0].shape)} on each of 2 devices')
```

```
61706 parameters in 10 tensors; images (60000, 1, 28, 28) on each of 2 devices
```

13.5.2.2 ブロードキャストと allreduce

ブロードキャストは初期パラメータをすべてのデバイスに 1 度だけコピーし、**allreduce** はデバイスごとに 1 つのテンソルを合計して、その総和をすべてのデバイスに残す。

ここでは単純な**スター** allreduce を記述する（デバイス 0 がすべてのテンソルを集めて合計し、その和を各デバイスにコピーして戻す）。

```
def broadcast(params, device):
    """A copy of every tensor in `params` on `device`, ready for autograd."""
    return [p.to(device, copy=True).requires_grad_() for p in params]

def allreduce(data):
    """Sum the tensors in `data`, one per device, onto every device."""
```

```
    for i in range(1, len(data)):    # Star: collect and add on device 0 ...
        data[0] += data[i].to(data[0].device)
    for i in range(1, len(data)):    # ... then copy the sum back
        data[i].copy_(data[0])
```

allreduce をテストするため、デバイス 0 に [1.0, 1.0]、デバイス 1 に [2.0, 2.0] を配置し、両方のデバイスが要素ごとの和 [3.0, 3.0] を受け取ることを確認しよう。

```
x = [torch.full((1, 2), i + 1.0, device=d) for i, d in enumerate(devices)]
print('before:', [x_j.tolist() for x_j in x])
allreduce(x)
print('after: ', [x_j.tolist() for x_j in x])
```

```
before: [[[1.0, 1.0]], [[2.0, 2.0]]]
after:  [[[3.0, 3.0]], [[3.0, 3.0]]]
```

13.5.2.3 データ並列ステップ

同じサイズの k 個のシャードがあるとき、シャードごとの平均損失の勾配 k 個の平均は、ミニバッチ全体の勾配に一致する（演習問題 2）。allreduce の呼び出しにはメッセージサイズによらず固定のレイテンシがかかるため、LeNet の 10 個の勾配テンソルを単一の平坦なベクトルに連結し、ステップごとに 1 回だけ allreduce を実行する。`communicate=False` を渡すと allreduce がスキップされ、純粋なデバイス計算の時間を計測できる。

```
def shard_grads(params, data, idx):
    """Gradient of the mean loss on each device's share of the minibatch
    `idx` (example indices), as one flat vector per device."""
    grads = []
    for ps, (X, y), i in zip(params, data, idx.chunk(len(params))):
        i = i.to(X.device, non_blocking=True)  # Don't wait for the device
        F.cross_entropy(lenet(ps, X[i]), y[i]).backward()
        grads.append(torch.cat([p.grad.flatten() for p in ps]))
        for p in ps:
            p.grad = None              # The next backward starts from zero
    return grads

def train_step(params, data, idx, lr, communicate=True):
    """One data-parallel SGD step on the minibatch `idx`; without
    communication, every device steps on its own shard's gradient."""
    grads = shard_grads(params, data, idx)
    if communicate:
        allreduce(grads)               # One message per device and step
    with torch.no_grad():
        for ps, g in zip(params, grads):
            for p, g_p in zip(ps, g.split([p.numel() for p in ps])):
                p -= lr / len(params) * g_p.view_as(p)  # Mean of k grads
```

13.5.2.4 1 ステップ検証

実行時間を計測する前に、2 基の GPU 上でのデータ並列の 1 ステップが、同じ 256 個の例に対する単一 GPU 上での 1 ステップと同じパラメータ更新を生成することを確認する（tf32 下での畳

み込みアルゴリズムの違いによって k で割り忘れるなどのバグが隠れないよう、厳密な fp32 演算を用いる)。

```
idx = torch.randperm(len(data.train))[:256]
updates = []
torch.set_float32_matmul_precision('highest')  # True fp32 in matmuls ...
with torch.backends.cudnn.flags(enabled=True, allow_tf32=False):  # ... convs
    for k in (1, 2):
        params = [broadcast(base, d) for d in devices[:k]]
        train_step(params, train_data[:k], idx, lr=0.2)
        updates.append([p.detach().cpu() - b
                        for p, b in zip(params[0], base)])
torch.set_float32_matmul_precision('high')
largest = max(u.abs().max() for u in updates[0])
diff = max((u1 - u2).abs().max() for u1, u2 in zip(*updates))
print(f'largest update {largest:.1e}, largest difference {diff:.1e}')
```

```
largest update 2.4e-02, largest difference 3.0e-08
```

単一 GPU と 2 基の GPU による更新は、fp32 の丸め誤差（約 10^{-7}）の範囲で一致する。

13.5.3 リング allreduce

スター allreduce はすべての通信トラフィックをデバイス 0 経由でルーティングするため、ハブとなるデバイスは $(k-1)S$ バイト（勾配のサイズを S とする）を送受信しなければならない。デバイスを追加するたびにハブの負荷は増大する一方、他の各デバイスが扱う通信量は双方向とも S バイトにとどまる（Figure 13.17 左）。

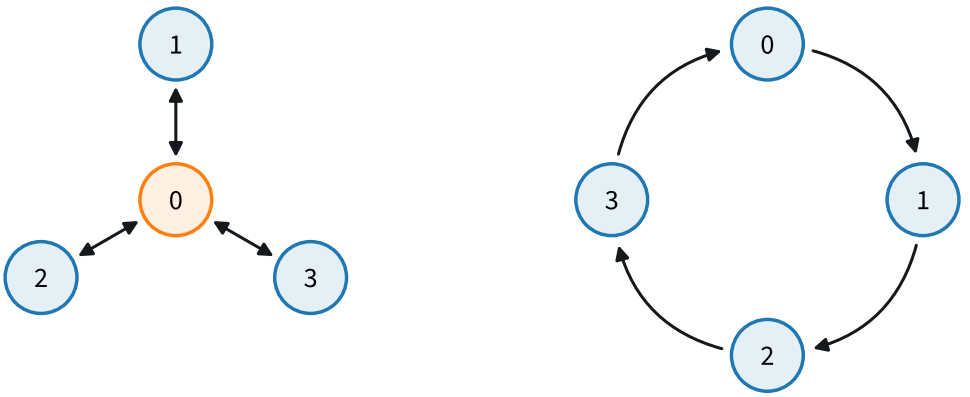


Figure 13.17: $k = 4$ におけるスター allreduce とリング allreduce の通信量。スターのハブは $(k-1)S$ バイトを送受信するが、リングではどのデバイスも $2(k-1)S/k$ バイトを送受信し、これは $2S$ 未満に収まる（任意の k に対して成立）。

リング allreduce (Patarasuk and Yuan 2009; Sergeev and Del Balso 2018) はこのハブをなくす。k 個のデバイスを論理的なリング状に並べてトラフィックを均等に分散させ、各デバイスの

S バイトの勾配を k 個のチャンク（各 S/k バイト）に分割し、それぞれ $k-1$ ステップからなる 2 つのフェーズを実行する（Figure 13.18）。

1. **reduce-scatter**：$k-1$ ステップのそれぞれにおいて、すべてのデバイスは後続のデバイスに 1 つのチャンクを送り、先行のデバイスから受け取ったチャンクを自身のローカルコピーに加算する。$k-1$ ステップ後、各デバイスは互いに異なる 1 つのチャンクについて完全な総和を保持する。
2. **all-gather**：続く $k-1$ ステップにわたり、加算が完了したチャンクをリングに沿って巡回させ、すべてのデバイスが合計済みの k 個のチャンクすべてを保持する状態にする。

各デバイスは 1 フェーズあたり $(k-1)S/k$ バイトを送信するため、合計の送信量は次のようになる。

$$\frac{2(k-1)}{k}\,S \quad \text{bytes per device.} \tag{13.7}$$

デバイスあたりの通信量は S（$k=2$ の場合）であり、リングに加わるデバイスがいくつになっても厳密に $2S$ 未満にとどまる。さらに、これら 2 つのフェーズは単独でも有用である。シャーディングを用いた訓練（13.7 節）では、合計された勾配を分割するのに reduce-scatter を使い、シャード化された重みを集めるのに all-gather を使う。

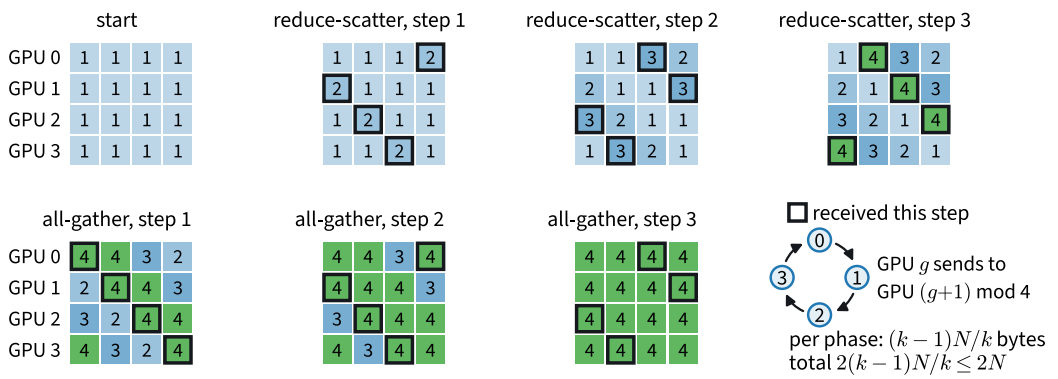


Figure 13.18: 4 基の GPU 上でのリング allreduce をステップごとに計算した様子。reduce-scatter はリングに沿ってデバイスごとに 1 つのチャンクを累積し、続いて all-gather が完成したチャンクを配布する。各セルの数字はそのチャンクが 4 つ分の寄与のうちいくつを保持しているかを表し、4 になったチャンクで加算が完了する。枠線付きのセルはそのステップで GPU が受信したチャンクであり、最後のパネルのリングはどの GPU がどの GPU へ送信するかを示す。

この 2 フェーズのリングを NumPy で実装し、各デバイス上で累積されるチャンクと、デバイスあたりの合計送信バイト数 `sent` の両方を追跡する。

```
def ring_allreduce(xs):
    """Ring allreduce of the vectors xs, one per device; returns every
    device's result and the bytes that each device sent."""
    k = len(xs)
    chunks = [np.array_split(x.copy(), k) for x in xs]  # chunks[j][c]
    sent = 0
    for s in range(k - 1):  # Reduce-scatter: add the received chunk
```

```
            msgs = [chunks[j][(j - s) % k].copy() for j in range(k)]
            for j, msg in enumerate(msgs):
                chunks[(j + 1) % k][(j - s) % k] += msg
            sent += msgs[0].nbytes
        for s in range(k - 1):  # All-gather: replace by the received chunk
            msgs = [chunks[j][(j + 1 - s) % k].copy() for j in range(k)]
            for j, msg in enumerate(msgs):
                chunks[(j + 1) % k][(j + 1 - s) % k] = msg
            sent += msgs[0].nbytes
        return [np.concatenate(c) for c in chunks], sent
```

ランダムな 1,024 要素の float ベクトルを用いて `ring_allreduce` を $k \in \{2, 4, 8, 16\}$ 個のデバイスでテストすると、すべてのデバイスが正確な総和を受け取りつつ、送信量が $2(k-1)S/k$ バイトになっていることが確認できる。

```
rng = np.random.default_rng()
for k in (2, 4, 8, 16):
    xs = [rng.standard_normal(1024, dtype=np.float32) for _ in range(k)]
    results, sent = ring_allreduce(xs)
    correct = all(np.allclose(r, sum(xs), atol=1e-5) for r in results)
    print(f'k = {k:2d}: every device holds the sum: {correct}; '
          f'sent per device: {sent / xs[0].nbytes:.2f} S')
```

```
k =  2: every device holds the sum: True; sent per device: 1.00 S
k =  4: every device holds the sum: True; sent per device: 1.50 S
k =  8: every device holds the sum: True; sent per device: 1.75 S
k = 16: every device holds the sum: True; sent per device: 1.88 S
```

デバイスあたりの通信バイト数は $2S$ 未満に抑えられるものの、リングでは $2(k-1)$ 回の逐次ホップを実行し、各ホップで起動レイテンシを支払うことになる。多数の GPU 上で小さなメッセージをやり取りする場合、集団通信ライブラリは深さ $\log_2 k$ の木アルゴリズムに切り替える（演習問題 5）。

13.5.4 通信コストとスケーリング

13.5.4.1 1 ステップのコストモデル

レイテンシ・帯域幅モデル (Hockney 1994; Thakur et al. 2005) を用いると、各 allreduce 呼び出しは固定の起動レイテンシ α に加え、各デバイスが送信する 1 バイトあたり β_{link} の時間（allreduce の**バス帯域幅**（bus bandwidth）の逆数）を要する。通信と計算のオーバーラップがない場合、k 個のデバイス上でグローバルバッチ B と c 回の allreduce 呼び出しを伴う 1 ステップは次の時間を要する。

$$t_{\text{step}}(k) \approx t_{\text{compute}}(B/k) + c\,\alpha + \frac{2(k-1)}{k}\,S\,\beta_{\text{link}}, \tag{13.8}$$

ここで、$c = 0$（$k = 1$ のとき）である。2 つ目のデバイスがステップを短縮できるのは、$t_{\text{compute}}(B) - t_{\text{compute}}(B/2) > c\alpha + S\,\beta_{\text{link}}$ が成り立つ場合に限られる。

13.5.4.2 allreduce のレイテンシと帯域幅

2 基の GPU 上で、α は 1 要素の float の allreduce によって測定し、β_{link} は 2^{26} 要素の float（268 MB）の allreduce によって測定する。

```
def allreduce_time(n, devices, calls):
    """Seconds per allreduce of n floats per device."""
    x = [torch.zeros(n, device=d) for d in devices]
    return d2l.Benchmark(lambda: allreduce(x), warmup=10 * calls,
                         repeats=3 * calls, devices=devices).time

alpha = allreduce_time(1, devices, calls=100)  # Our star: two copies in turn
n = 2**26                                      # 64M floats ...
S = 4 * n                                      # ... 268 MB (256 MiB) per device
beta_link = (allreduce_time(n, devices, calls=3) - alpha) / S  # s per byte
print(f'alpha = {1e6 * alpha:.0f} us, beta_link = {1e9 * beta_link:.2f} '
      f'ns/byte (bus bandwidth {1e-9 / beta_link:.1f} GB/s)')
```

```
alpha = 56 us, beta_link = 0.11 ns/byte (bus bandwidth 9.5 GB/s)
```

Equation 13.8 が勾配融合（gradient fusion）の効果を予測できるか確かめるため、LeNet の 10 個の勾配テンソル（合計 $S = 247$ KB）を個別に allreduce する場合（$c = 10$）と、単一の平坦なベクトルに連結して allreduce する場合（$c = 1$）を比較する。両者の差は 9α になるはずである。

```
grads = [[torch.zeros_like(p, device=d) for p in base] for d in devices]
flat = [torch.cat([g.flatten() for g in G]) for G in grads]
def separately():
    for g in zip(*grads):              # One allreduce per tensor
        allreduce(list(g))
n_bytes, c = flat[0].nbytes, len(base)
for name, f, calls in (('separately', separately, c),
                       ('at once', lambda: allreduce(flat), 1)):
    t = d2l.Benchmark(f, warmup=1000, repeats=300, devices=devices).time
    print(f'{name:10s}: {1e6 * t:4.0f} us measured, '
          f'{1e6 * (calls * alpha + n_bytes * beta_link):4.0f} us predicted')
```

```
separately:  572 us measured,  587 us predicted
at once   :   81 us measured,   82 us predicted
```

247 KB の勾配では、余分な 9 通のメッセージの封筒（9α）のほうが、その中身である全バイトの転送よりも多くのコストがかかる。実用的なデータ並列ライブラリが、通信前に小さな勾配を大きなバケットにまとめるのはこのためである（13.6 節）。

13.5.4.3 1 基および 2 基の GPU 上での LeNet

2 基の GPU で LeNet を実行する前に、256 個の例からなるミニバッチに対する 1 基の GPU 上でのローカルな計算をプロファイリングしてみよう。

```
params = [broadcast(base, devices[0])]
step = lambda: train_step(params, train_data[:1], idx, 0.2, communicate=False)
t = d2l.Benchmark(step, warmup=500, repeats=150, devices=devices[:1]).time
d2l.show_profile(d2l.kernel_times(step, steps=100), 1e3 * t, rows=4)
```

```
wall clock 2.91 ms per step, device busy 0.30 ms (10%), matrix products 43%
  0.05 ms  16%  Kernel<cutlass_tensorop_s1688dgrad_optimized_tf32_12
  0.04 ms  12%  implicit_convolve_sgemm<float, float, 128, 5, 5, 3,
  0.03 ms  10%  wgrad_alg0_engine<float, 128, 5, 5, 3, 3, 3, false,
  0.03 ms   9%  avg_pool2d_backward_out_cuda_frame<float, float, int
```

ステップの大半で GPU はアイドル状態にある。バッチサイズ 256 の LeNet はオーバーヘッド律速（overhead-bound）であり、ステップ時間はホストディスパッチ時間 h_{step} に等しい。そのため、1 基の GPU でバッチを 256 から 128 に半減させてもステップ時間はほとんど変わらない一方、1 つのホストスレッドから 2 基の GPU を駆動すると約 $2h_{\text{step}}$ かかる（Figure 13.19）。

```
params, t_local = [broadcast(base, d) for d in devices], {}
for k, b in ((1, 256), (1, 128), (2, 128)):
    t_local[k, b] = d2l.Benchmark(
        lambda: train_step(params[:k], train_data[:k], idx[:k * b], 0.2,
                           communicate=False),
        warmup=500, repeats=150, devices=devices[:k]).time
    print(f'{k} GPU(s), {b} examples each: {1e3 * t_local[k, b]:.2f} ms')
```

```
1 GPU(s), 256 examples each: 2.91 ms
1 GPU(s), 128 examples each: 2.80 ms
2 GPU(s), 128 examples each: 5.60 ms
```

単一ホストのディスパッチ下限 $k\,h_{\text{step}}$ をデバイス計算時間 $t_{\text{dev}}(B/k)$ とあわせて組み込むと、Equation 13.8 は次のように精密化される。

$$t_{\text{step}}(k) \approx \max\bigl(k\,h_{\text{step}},\; t_{\text{dev}}(B/k)\bigr) + c\,\alpha + \frac{2(k-1)}{k}\,S\,\beta_{\text{link}}. \tag{13.9}$$

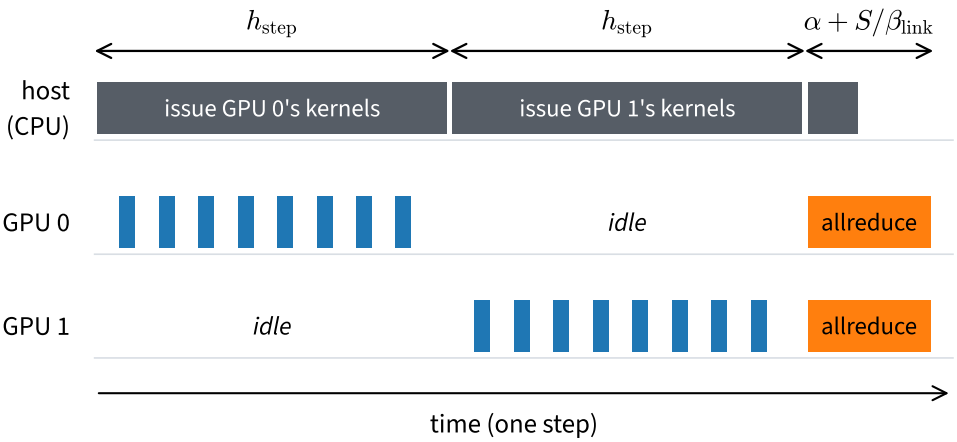


Figure 13.19: 2 基の GPU 上でのホスト律速ステップ。1 つのホストがまず GPU 0 の処理を発行し、次に GPU 1 の処理を発行する様子として描いている。ホストが一方の GPU を処理している間、もう一方の GPU はアイドル状態になるため、カーネルが短いステップではホスト時間の約 $2h_{\text{step}}$ に通信時間を加えた時間がかかる。

両方の予測をエンドツーエンドで検証するため、訓練ループ `train` を記述する。この関数は、

`orders` 内の事前生成されたデータ例の順列ごとに 1 エポックを実行し、（最初のエポックを除いた）平均ステップ時間と検証正解率を返す。

```
def train(params, train_data, test_data, orders, batch_size=256, lr=0.2):
    """Data-parallel SGD on the devices that hold `params` and `train_data`,
    one epoch per example order; returns seconds per step, test accuracy."""
    devices = [X.device for X, _ in train_data]
    times = []
    for order in orders:
        t0 = time.perf_counter()
        for idx in order.split(batch_size):
            train_step(params, train_data, idx, lr)
        for d in devices:  # Wait until every device has finished
            torch.cuda.synchronize(d)
        times.append((time.perf_counter() - t0)
                     / math.ceil(len(order) / batch_size))
    X, y = test_data
    with torch.no_grad():
        acc = (lenet(params[0], X).argmax(1) == y).float().mean().item()
    return sum(times[1:]) / (len(times) - 1), acc
```

それでは、グローバルバッチサイズ 256 で `train` を $k \in \{1, 2\}$ 基の GPU 上で 5 エポック実行し、測定された 2 GPU のステップ時間 `t_step[2]` を、素朴な予測 Equation 13.8 と単一ランチャモデル Equation 13.9 の両方と比較しよう。

```
orders, t_step = [torch.randperm(len(data.train)) for _ in range(5)], {}
for k in (1, 2):
    params = [broadcast(base, d) for d in devices[:k]]
    t_step[k], acc = train(params, train_data[:k], test_data, orders)
    print(f'{k} GPU(s): {1e3 * t_step[k]:.2f} ms per step, '
          f'test accuracy {acc:.2f}')
exchange = alpha + n_bytes * beta_link    # One allreduce of the gradient
print(f'2 GPUs predicted: {1e3 * (t_local[1, 128] + exchange):.2f} ms per '
      f'step by eq_dp_cost, {1e3 * (2 * t_step[1] + exchange):.2f} ms with '
      f'one launcher')
```

```
1 GPU(s): 2.88 ms per step, test accuracy 0.87
2 GPU(s): 5.69 ms per step, test accuracy 0.87
2 GPUs predicted: 2.88 ms per step by eq_dp_cost, 5.85 ms with one launcher
```

どちらの実行も同じテスト正解率に到達し、Equation 13.9 は 2 GPU での速度低下を正確に予測している。バッチ 256 では LeNet の勾配交換はごくわずか（約 0.1 ms）であり、ボトルネックは単一のホストランチャから 2 基の GPU に処理を発行する部分にある。

13.5.4.4 強スケーリングと弱スケーリング、およびより大きなミニバッチ

マルチ GPU のベンチマークは 2 つの流儀のいずれかに従う。**強スケーリング**（strong scaling）は、グローバルバッチ B を固定したまま k を増やし（同じステップをより速く終えることを目指す）、**弱スケーリング**（weak scaling）は、GPU あたりのバッチ b を固定してグローバルバッチを kb へと拡大する（例/秒で表したスループットを $k\times$ にすることを目指すが、グローバルバッチが大きくなると最適化のダイナミクスが変わる、9.8 節）。強スケーリングは**アムダールの法則**（Amdahl 1967）の領域である。ステップのうち並列化できない部分（ここではホストランチャと

勾配交換）が、GPU をどれだけ増やしても高速化率の上限を決めてしまう。弱スケーリングは**グスタフソンの法則** (Gustafson 1988) の領域である。プロセッサ数に応じて計算量が増えると、逐次部分の相対的な割合は縮小し、スループットを伸ばし続けることができる。

Equation 13.9 より、2 基目の GPU が弱スケーリングのスループットを向上させる（$2\,t(1)/t(2) > 1$）のは、次の条件が成り立つときである。

$$\max(2h_{\text{step}},\ t_{\text{dev}}(b)) + \alpha + S\,\beta_{\text{link}} \ <\ 2\max(h_{\text{step}},\ t_{\text{dev}}(b)).$$

$t_{\text{dev}}(b) \le h_{\text{step}}$ のときは、2 基目の GPU を追加しても決して速くならない。$t_{\text{dev}}(b)$ が $2h_{\text{step}}$ と勾配交換時間を大きく上回るようになると、高速化率は 2 に近づく。これを GPU あたり $b \in \{256, 4096, 32768\}$ 個の例にわたって検証しよう。

```
for b in (256, 4096, 32768):                # Examples per GPU
    batch = torch.randint(len(data.train), (2 * b,))  # With replacement
    steps = [lambda k=k: train_step(params[:k], train_data[:k],
                                    batch[:k * b], 0.2) for k in (1, 2)]
    times = [[], []]                        # Three timed windows per k,
    for w in range(3):                      # taken in turn for k = 1 and 2
        for k in (1, 2):
            params = [broadcast(base, d) for d in devices]  # Equal replicas
            times[k - 1].append(d2l.Benchmark(
                steps[k - 1], warmup=10 if w == 0 else 0,
                repeats=max(20, 2**18 // b), devices=devices[:k]).time)
    t = np.median(times, axis=1)
    t_dev = sum(d2l.kernel_times(steps[0]).values()) / 1e3
    if b == 256:              # The host sets this step's time (profile above)
        h_step = t[0]
    predicted = 2 * t[0] / (max(2 * h_step, t_dev) + exchange)
    speedups = 2 * np.array(times[0]) / np.array(times[1])
    print(f'b = {b:5d}: 1 GPU {1e3 * t[0]:6.2f} ms (device busy '
          f'{1e3 * t_dev:5.2f} ms), 2 GPUs {1e3 * t[1]:6.2f} ms\n'
          f'           speedup {predicted:.2f} predicted, '
          f'{2 * t[0] / t[1]:.2f} measured '
          f'(windows {speedups.min():.2f} to {speedups.max():.2f})')
```

```
b =   256: 1 GPU   2.83 ms (device busy  0.30 ms), 2 GPUs   5.83 ms
           speedup 0.99 predicted, 0.97 measured (windows 0.92 to 0.98)
b =  4096: 1 GPU   3.01 ms (device busy  3.00 ms), 2 GPUs   6.11 ms
           speedup 1.05 predicted, 0.99 measured (windows 0.98 to 0.99)
b = 32768: 1 GPU  25.26 ms (device busy 25.35 ms), 2 GPUs  25.53 ms
           speedup 1.99 predicted, 1.98 measured (windows 1.98 to 1.98)
```

$b = 256$ では 2 基目の GPU による高速化は得られないが、$b = 32{,}768$ ではデバイスの計算時間が $2h_{\text{step}}$ と 247 KB の勾配交換を大きく上回り、予測どおりスループットがほぼ 2 倍になる。

13.5.5 まとめ

データ並列はモデルを k 個のデバイスに複製し、デバイスごとにミニバッチの $1/k$ に対する勾配を計算し、更新の前に allreduce で勾配を合計する。リング allreduce はこの総和を reduce-scatter とそれに続く all-gather に分解し、デバイスあたり $2(k-1)S/k < 2S$ バイトを $2(k-1)$ ホップにわたって送信する。本節のステップコストモデル Equation 13.9 は、デバイス計算時間、

単一ホストのディスパッチオーバーヘッド $k\,h_{\text{step}}$、呼び出しごとのレイテンシ $c\alpha$、および帯域幅コスト $2(k-1)S\beta_{\text{link}}/k$ を組み合わせたものであり、小さな勾配テンソルを連結することでなぜ 9α を節約できるのか、そして GPU あたりのデバイス計算がホストディスパッチと通信の両方を上回って初めて 2 基目の GPU が訓練を高速化できるのはなぜかを説明する。

13.5.6 演習問題

1. **データ並列とパイプライン並列の比較**。バイアスなしの L 個の線形層（幅 d、fp32）のスタックを、2 基の GPU 上でバッチサイズ B で訓練する場合を考える。データ並列はすべての重み勾配を allreduce し、2 ステージのパイプライン並列（Figure 13.15）は各 GPU に $L/2$ 層を配置して、分割境界をまたいで順方向に $B \times d$ 個の活性化値を、逆方向に勾配を送信する。
   1. 各方式で 1 ステップあたりに転送されるバイト数を数え、$L = 32, d = 4096, B = 256$ に対して両者を評価し、データ並列のほうが転送バイト数が少なくなる (B, L, d) の条件を求めよ。
   2. バッチを m 個のパイプライン用マイクロバッチに分割したとき (Huang et al. 2019)、2 ステージのパイプラインにおける各 GPU のバブル（アイドル）割合を導出し、$m = 1$ および $m = 8$ に対して評価せよ。
2. **勾配の正規化**。k 個のシャード（合計 B 個の例からなるバッチを分割したもの）にわたる 2 つのデータ並列勾配レシピを比較せよ。レシピ A は各デバイスで例ごとの損失を合計し、デバイス間で勾配を合計してから B で割る。レシピ B は各デバイスで例ごとの損失を平均し、k 個のデバイス勾配を平均する。
   1. すべてのシャードが B/k 個の例を保持するとき、両者がともに厳密なフルバッチの平均勾配を与えることを示せ。
   2. トークン全体で損失を平均する言語モデルにおいて、2 つのシャードがそれぞれ 3,000 トークンと 1,000 トークンを保持しているとする。レシピ B が各シャード内の 1 トークンに割り当てる相対的な重みを求め、トークン数が不均等な場合でも厳密になるようレシピ B を修正せよ。
3. **強スケーリングの高速化率**。シャードの計算時間を $t_{\text{compute}}(s) = t_0 + s\,t_1$ とモデル化し、α を無視すると、Equation 13.8 より固定されたグローバルバッチ B において次式が成り立つ。
$$t_{\text{step}}(k) = t_0 + \frac{B\,t_1}{k} + \frac{2(k-1)}{k}\,S\,\beta_{\text{link}}. \tag{13.10}$$
   1. 強スケーリングの高速化率 $R(k) = t_{\text{step}}(1)/t_{\text{step}}(k) > 1$ が、ある 1 つの $k > 1$ で成り立つのは、すべての $k > 1$ で成り立つとき、かつそのときに限ることを示し、$\lim_{k\to\infty} R(k)$ を求めよ。
   2. $N = 11.2$M 個の fp32 パラメータ、$t_0 = 2$ ms、$t_1 = 0.2$ ms/例、$B = 256$ とするとき、$R(2)$、$R(4)$、および $\lim_{k\to\infty} R(k)$ をバス帯域幅 20 GB/s と 5 GB/s のそれぞれで評価し、少しでも高速化が得られる最小のバス帯域幅を求めよ。
   3. 5 GB/s における $R(4)$ および $\lim_{k\to\infty} R(k)$ を、勾配を bf16 で通信する場合（S が半分になる）について再計算せよ。
4. [code] **手作業によるリング allreduce**。●
   1. k 個のデバイス（それぞれ S バイトを保持）について、スター allreduce とリング allreduce で移動する総バイト数と、最も負荷の高いデバイスが送受信するバイト数を比較せよ。また、k 本すべての双方向リンクが同時に動作する場合と、すべての転送が単一のバスを逐次的に共有する場合のそれぞれについて、両者の所要時間の比を予測せよ。
   2. PyTorch で `ring_allreduce(data)` を（`chunk` と明示的な `.to()` コピーを用いて）実装し、`allreduce` と比較して $k \in \{2, 3, 4\}$ で検証したうえで、デバイスあたり 64M 個の float に対して $k = 2$ および $k = 4$ で両者をベンチマークせよ。

5. **allreduce におけるレイテンシ。**●n バイトの 1 ホップに $\alpha_{\text{hop}} + n\,\beta_{\text{link}}$ かかるとき、$2(k-1)$ ステップのリング allreduce の所要時間は

$$t_{\text{ring}} = 2(k-1)\left(\alpha_{\text{hop}} + \frac{S}{k}\,\beta_{\text{link}}\right), \tag{13.11}$$

であるのに対し、二分木 allreduce（$2\log_2 k$ ステップ、各ステップ S バイトで上向きに総和をとり下向きにブロードキャストする）の所要時間は

$$t_{\text{tree}} = 2\log_2 k\,\left(\alpha_{\text{hop}} + S\,\beta_{\text{link}}\right). \tag{13.12}$$

である。

1. 各アルゴリズムのレイテンシ項と帯域幅項が k に伴いどうスケールするかを比較せよ。$k = 8$、$\alpha_{\text{hop}} = 10$ µs、バス帯域幅 20 GB/s のとき、木のほうが高速になる境界のメッセージサイズ S を求めよ。
2. $k = 2$ において、半性能メッセージサイズ $S_{1/2}$（固定レイテンシが転送時間と等しくなるサイズ）を α と β_{link} を用いて表し、13.5.4 節での測定値から評価して、13.6 節の勾配バケットのサイズと比較せよ。

13.6 データ並列の実践

13.5 節で手書きしたデータ並列（data parallelism）ループでは、2 基の GPU で LeNet がかえって遅くなった。その理由は 2 つある。単一の Python スレッドが両方の GPU へカーネルを逐次的にディスパッチしていたこと、そして出力に近い層の勾配はずっと早い段階で揃っていたにもかかわらず、逆伝播全体が完了するまでオールリデュースが待機していたことである。

実用データ並列ツールは、これら両方のボトルネックを解消する。GPU ごとに 1 つのホストプロセスを実行するか、全 GPU にまたがる単一のシャーディングされたプログラムをコンパイルし、さらに勾配を**バケット**にまとめて逆伝播中に各バケットが埋まり次第そのオールリデュースを開始するのである。逆伝播の計算の背後に勾配交換をどれだけ隠蔽できるかを検証するため、ここでは LeNet から 64×64 の Fashion-MNIST 画像に対する ResNet-18 へと対象を移す（画像 1 枚あたり 22 億回の積和演算、45 MB の勾配 1 バイトあたり約 300 訓練 FLOPs を要し、画像 256 枚の 1 ステップあたり 1 基の RTX 4090 を約 100 ms 稼働させ続ける）。並列ステップが何を計算しているかを 1 ステップチェックで検証したうえで、1、2、4 基の GPU にわたる弱スケーリング（weak scaling）と強スケーリング（strong scaling）を計測し、コストモデルと比較する。

```
import json
import os
import random
import sys
```

13.6.1 実用データ並列

13.6.1.1 実行モデル

現代のあらゆるフレームワークは、次の 2 種類の**単一プログラム複数データ**（single program, multiple data、SPMD）設計のいずれかを用いて k 基の GPU を駆動する。

1. **マルチプロセス（k 個の複製＋集団通信ライブラリ）：**ランチャが k 個の独立した Python プロセス（GPU ごとに 1 つ、それぞれが $\{0, \dots, k-1\}$ の固有の**ランク**と専用の CPU スレッドを持つ）を起動し、NCCL を介して勾配を同期する。各プロセスの内部では任意の Python コードを実行できる。

2. **コンパイラ分割型の単一プログラム：**単一デバイス向けのステップを記述し、デバイスメッシュ上での入力のシャーディングを注釈として指定すると、コンパイラが計算を分割し、集団通信演算（collective operations）を自動的に挿入・スケジューリングする (Xu et al. 2021)。

PyTorch はマルチプロセス設計を採用しており、`torchrun` が k 個のプロセスを起動し、`DistributedDataParallel`（DDP）がモデルをラップする。ノートブック自体は単一プロセスであるため、ここで定義するヘルパ関数 `torchrun` は自己完結したスクリプトを一時ファイルに書き出し、`torchrun --standalone --nproc-per-node=k` を実行して、各ランクの JSON 出力をパースする。

```
import glob
import subprocess
import tempfile

def torchrun(script, nproc, *args, env=None):
    """Run Python source in nproc processes; return their JSON lines."""
    with tempfile.TemporaryDirectory() as tmp:
        path = os.path.join(tmp, 'script.py')
        with open(path, 'w') as f:
            f.write(script)
        launcher = os.path.join(os.path.dirname(sys.executable), 'torchrun')
        # Each process prints to a file of its own (--redirects 1), so that
        # the lines of different processes cannot interleave
        run = subprocess.run(
            [launcher, '--standalone', f'--nproc-per-node={nproc}',
             '--log-dir', tmp, '--redirects', '1', path, *map(str, args)],
            capture_output=True, text=True, env={**os.environ, **(env or {})})
        if run.returncode != 0:
            raise RuntimeError(run.stderr[-3000:])
        logs = sorted(glob.glob(f'{tmp}/*/attempt_0/*/stdout.log'),
                      key=lambda log: int(log.split(os.sep)[-2]))  # By rank
        return [json.loads(line) for log in logs for line in open(log)
                if line.startswith('{')]

def init_distributed():
    """Join the process group of torchrun; return the rank, the number of
    processes, and the GPU of this process."""
    device = torch.device('cuda', int(os.environ['LOCAL_RANK']))
    torch.cuda.set_device(device)
    torch.distributed.init_process_group('nccl', device_id=device)
    return (torch.distributed.get_rank(), torch.distributed.get_world_size(),
            device)
```

13.6.1.2 逆伝播と通信のオーバーラップ

実用フレームワークは、層の逆順（誤差逆伝播法が勾配を生成する順序）で勾配を数十メガバイト単位の**バケット**に詰め込み、バケット内の最後の勾配が揃い次第、各バケットの非同期オールリデュースを起動する (S. Li et al. 2020)（Figure 13.20）。

バケットサイズは、呼び出しごとのレイテンシ α とオーバーラップのトレードオフを決める。極小のバケットでは α を何度も支払うことになり、単一の巨大なバケットでは逆伝播が完了するまで通信を開始できない。理想的なオーバーラップの**バケットモデル**のもとでは（すなわち、次のバケットが埋まる前に各バケットの通信を終えられるほど GPU 間リンクが高速であり、かつ通

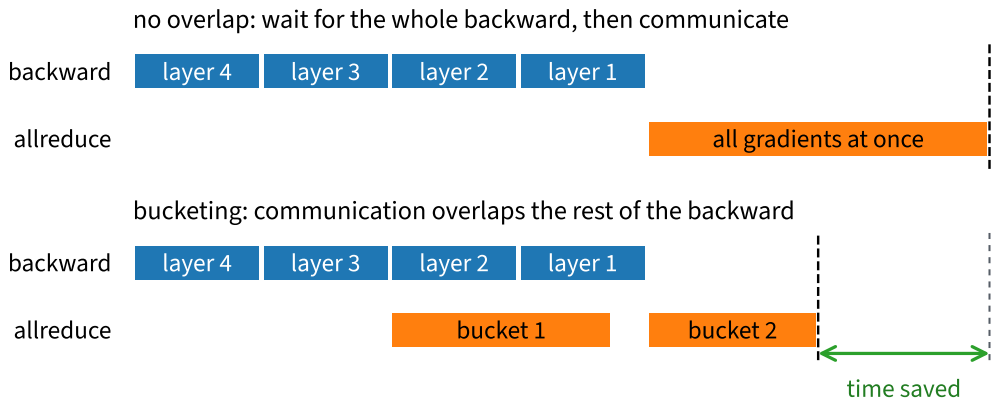


Figure 13.20: 上：オールリデュースが逆伝播全体の完了を待つ場合。下：勾配をバケットにまとめ、逆伝播が各バケットの勾配を生成し終え次第、そのオールリデュースを開始する場合。逆伝播の終了後に実行されるのは最後のバケットのみであり、それより前のバケットが隠蔽されるのは、逆伝播がまだ実行されている間に各オールリデュースが完了する場合に限られる。

信が同時実行中の逆伝播カーネルを減速させないと仮定すると)、最後のバケットを除くすべてのバケットが逆伝播の背後に隠蔽される。

DDP は最初のバケットを 1 MiB で閉じ、それ以降のバケットを 25 MiB（`bucket_cap_mb`）で閉じる。ResNet-18 の 45 MB の勾配に対し、DDP は約 9、26、9 MB の 3 つのバケットを形成する（最後の 9 MB のバケットには、最も遅く計算される入力側の層が入る)。

13.6.1.3 訓練ステップ

7.4 節の小画像向け ResNet-18（第 32 章（英語版） でも利用するため `#@save` しておく）を定義し、その勾配バイト数 S（fp32 で 45 MB）を数える。

```
def resnet18(num_classes, in_channels=1):
    """ResNet-18 for small images: a 3x3 stem without max-pooling, He
    initialization."""
    def stage(num_channels, first=False):
        return [d2l.Residual(num_channels, use_1x1conv=not first,
                             strides=1 if first else 2),
                d2l.Residual(num_channels)]
    net = nn.Sequential(
        # No stem bias: the batch normalization after the conv removes it
        nn.Conv2d(in_channels, 64, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(64), nn.ReLU(),
        *stage(64, first=True), *stage(128), *stage(256), *stage(512),
        nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(512, num_classes))
    with torch.no_grad():  # Create the lazy parameters of the blocks
        net.eval()(torch.zeros(1, in_channels, 32, 32))
    for m in net.modules():  # He initialization, zero biases
        if isinstance(m, (nn.Conv2d, nn.Linear)):
            nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
            if m.bias is not None:
```

```
                nn.init.zeros_(m.bias)
    return net.train()
```

次に、Fashion-MNIST を読み込み、ResNet-18 のパラメータ数 num_params と fp32 での勾配データ量 S を計算する。

また、DDP がオールリデュースを起動する前に逆順で詰め込む 1 MiB および 25 MiB の勾配バケット buckets も計算する。

```
d2l.FashionMNIST()                  # Download once, before any process runs
net = resnet18(10)
num_params = d2l.num_params(net)
S = 4 * num_params                  # Gradient bytes in fp32
# DDP's buckets: the gradients in the order in which the backward pass
# produces them, the reverse of the parameters', the first bucket closed
# at 1 MiB and every later one at 25 MiB
buckets, size = [], 0
for p in reversed(list(net.parameters())):
    size += 4 * p.numel()
    if size >= (25 if buckets else 1) * 2**20:
        buckets, size = buckets + [size], 0
buckets += [size] if size else []
last_bucket = buckets[-1]           # Bytes of the bucket sent last
print(f'{num_params / 1e6:.1f}M parameters, S = {S / 1e6:.0f} MB in buckets '
      f'of {", ".join(f"{b / 1e6:.1f}" for b in buckets)} MB')
```

```
11.2M parameters, S = 45 MB in buckets of 9.5, 26.5, 8.7 MB
```

続いて、64×64 の Fashion-MNIST 画像に対するデータ並列の訓練ステップを定義する。

TRAIN_DDP は resnet18 を DDP でラップし、プロセス間で train_step の実行時間を計測する（オプションとして model.no_sync() およびカーネルプロファイリングも備える）。

```
TRAIN_DDP = r'''
import contextlib, json, sys, torch
import torch.distributed as dist
from torch.nn import functional as F
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
torch.backends.cudnn.benchmark = True  # Tune convolutions for fixed shapes
# b examples per GPU and step; sync=0 skips the gradient allreduce, and
# profile=1 also reports the device time per step of NCCL and other kernels
b, num_steps, seed, sync, profile = map(int, sys.argv[1:])
torch.manual_seed(seed)                # The same permutations everywhere

data = d2l.FashionMNIST().train        # A copy on every GPU, at 64 x 64
X = F.interpolate(data.data.to(device)[:, None] / 255, 64, mode='bilinear')
y = data.targets.to(device)
model = DDP(d2l.resnet18(10).to(device), device_ids=[device])
opt = torch.optim.SGD(model.parameters(), lr=0.1)
```

```
def minibatches():
    """This process's b examples of every global minibatch of k * b."""
    while True:
        perm = torch.randperm(len(y), device=device)
        for s in range(len(y) // (k * b)):
            yield perm[(s * k + rank) * b:][:b]

def train_step(i):
    opt.zero_grad(set_to_none=True)
    # Under no_sync, which must enclose the forward pass as well, each GPU
    # keeps its own gradient
    with model.no_sync() if not sync else contextlib.nullcontext():
        F.cross_entropy(model(X[i]), y[i]).backward()
    opt.step()

batches = minibatches()
t = d2l.Benchmark(lambda: train_step(next(batches)), warmup=10,
                  repeats=num_steps).time
result = {'rank': rank, 'step_time': t}
if profile:
    kernels = d2l.kernel_times(lambda: train_step(next(batches)))
    result['comm'] = sum(ms for name, ms in kernels.items() if 'nccl' in name)
    result['compute'] = sum(kernels.values()) - result['comm']
print(json.dumps(result))
dist.destroy_process_group()
'''
```

13.6.1.4 1 ステップチェック

LeNet とは異なり、ResNet-18 は各チャネルをミニバッチの平均と分散で正規化するバッチ正規化（7.3 節）を含んでいる。256 個の例からなるミニバッチを 2 基の GPU に分割したとき、各 GPU は自身の 128 例のシャード統計量で正規化するのだろうか（これは 1 基の GPU 上で 128 例のマイクロバッチ 2 つにわたって勾配累積した場合と一致する）。それとも、GPU 間で 256 例の大域的な統計量を同期するのだろうか。2 基の GPU による更新を、単一 GPU におけるこれら両方の参照結果と比較して確かめてみよう。

```
CHECK_DDP = r'''
import copy, json, sys, torch
import torch.distributed as dist
from torch.nn import functional as F
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
torch.backends.cudnn.allow_tf32 = False  # Full fp32 convolutions
torch.manual_seed(int(sys.argv[1]))      # The same model and data everywhere
X = torch.rand(256, 1, 64, 64, device=device)
y = torch.randint(0, 10, (256,), device=device)
model = d2l.resnet18(10).to(device)
ref = copy.deepcopy(model)               # An unwrapped copy for one GPU
```

```
def update(net, shards):
    """Parameter change of one SGD step on the mean loss over the shards."""
    before = [p.detach().clone() for p in net.parameters()]
    for X_j, y_j in shards:
        (F.cross_entropy(net(X_j), y_j) / len(shards)).backward()
    with torch.no_grad():
        for p, b in zip(net.parameters(), before):
            p -= 0.1 * p.grad
    return [p.detach() - b for p, b in zip(net.parameters(), before)]

ddp = update(DDP(model, device_ids=[device]),
             [(X.chunk(k)[rank], y.chunk(k)[rank])])
if rank == 0:
    gap = lambda u: max((a - b).abs().max().item() for a, b in zip(ddp, u))
    halves = update(copy.deepcopy(ref), list(zip(X.chunk(k), y.chunk(k))))
    print(json.dumps({'largest': max(u.abs().max().item() for u in ddp),
                      'full': gap(update(ref, [(X, y)])),
                      'halves': gap(halves)}))
dist.destroy_process_group()
'''
if d2l.num_gpus() >= 2:
    r = d2l.torchrun(CHECK_DDP, 2, random.getrandbits(32))[0]
    print(f"largest update {r['largest']:.1e}\n"
          f"largest difference from one GPU on all 256: {r['full']:.1e}\n"
          f"largest difference from one GPU on two accumulated halves: "
          f"{r['halves']:.1e}")
```

```
largest update 1.7e-02
largest difference from one GPU on all 256: 1.0e-03
largest difference from one GPU on two accumulated halves: 1.2e-07
```

デフォルトでは、PyTorch DDP は各 GPU のシャードをシャードごとの局所統計量で正規化する。そのため、2 分割して累積した場合とは約 10^{-7} の精度で一致する一方、フルバッチ統計量を用いた場合とは数パーセント異なる（`nn.SyncBatchNorm` に変換すれば、追加の集団通信と引き換えに大域統計量が同期される。演習問題 4）。

13.6.2 4 基の GPU でのスケーリング

ここでは、$k \in \{1, 2, 4\}$ 基の GPU にわたって弱スケーリング（GPU あたり 256 例）と強スケーリング（全体で 512 例）を計測する。

```
def step_time(k, b, num_steps=50):
    """Seconds per DDP step on k GPUs with b examples per GPU."""
    ranks = d2l.torchrun(TRAIN_DDP, k, b, num_steps, random.getrandbits(32),
                         1, 0)          # Allreduce the gradients, no profile
    return max(r['step_time'] for r in ranks)
```

まず、NCCL を介した単体の 256 MB オールリデュースで β_{link} を測定し、弱スケーリングのもとでの露出通信時間（exposed communication time）$t_{\text{step}}(k) - t_{\text{step}}(1)$ に対する次の 2 つの予測を比較する。

1. **オーバーラップなし：**$S = 45$ MB のリングオールリデュース全体が露出する（$\frac{2(k-1)}{k} S\beta_{\text{link}}$）。
2. **完全なバケットオーバーラップ：**逆伝播の終了後に最後のバケットまたはバッファ（$last_bucket$）のみが露出する。

```
ALLREDUCE = r'''
import json, torch
import torch.distributed as dist
from d2l import torch as d2l

_, _, device = d2l.init_distributed()
x = torch.zeros(2**26, device=device)   # 64M floats: 256 MB
t = d2l.Benchmark(lambda: dist.all_reduce(x), repeats=10).time
print(json.dumps({'beta_link': t / x.nbytes}))  # Each GPU sends S at k = 2
dist.destroy_process_group()
'''
def allreduce_beta(env=None):
    """beta_link of a bare 256 MB allreduce on two GPUs: the seconds
    per byte that each GPU sends."""
    return max(r['beta_link'] for r in d2l.torchrun(ALLREDUCE, 2, env=env))
```

`allreduce_beta()` を呼び出すと実測リンクコスト `beta_link` が得られる。これを用いて、$k \in \{2, 4\}$ 基の GPU に対する `exposed[k]` を、オーバーラップなしモデル（$2(k-1)S\beta_{\text{link}}/k$）と完全オーバーラップモデル（`last_bucket`）の両方について計算する。

```
if d2l.num_gpus() >= 2:
    beta_link = allreduce_beta()
    print(f'bus bandwidth = {1e-9 / beta_link:.2f} GB/s '
          f'(beta_link = {1e9 * beta_link:.2f} ns/byte)')
    exposed = {}                    # Predicted exposed seconds per step
    for k in (2, 4):
        ring = 2 * (k - 1) / k      # Bytes that each GPU sends, per byte of S
        exposed[k] = (ring * S * beta_link, ring * last_bucket * beta_link)
        print(f'k={k}: exposed {1e3 * exposed[k][0]:.0f} ms without overlap '
              f'({ring:.1f} S beta_link), {1e3 * exposed[k][1]:.0f} ms with '
              f'full overlap (last bucket)')
```

```
bus bandwidth = 2.44 GB/s (beta_link = 0.41 ns/byte)
k=2: exposed 18 ms without overlap (1.0 S beta_link), 4 ms with full overlap (last bucket)
k=4: exposed 27 ms without overlap (1.5 S beta_link), 5 ms with full overlap (last bucket)
```

それでは、GPU あたり 256 例（`step_time(k, 256)`）で $k \in \{1, 2, 4\}$ 基の GPU にわたる弱スケーリングをベンチマークし、実測されたスループット、スケーリング効率、および露出オーバーヘッド `weak[k] - weak[1]` を両方の予測と比較しよう。

```
ks = [k for k in (1, 2, 4) if k <= d2l.num_gpus()]
weak = {k: step_time(k, 256) for k in ks}
for k in ks:
    print(f'{k} GPU(s): {k * 256 / weak[k]:5.0f} examples/s, speedup '
          f'{k * weak[1] / weak[k]:.2f}, efficiency {weak[1] / weak[k]:.0%}')
    if k > 1:  # Predicted: the one-GPU step plus the exposed communication
```

```
        none, full = (weak[1] / (weak[1] + e) for e in exposed[k])
        print(f'{"":10s}predicted {none:.0%} without overlap, {full:.0%} '
              f'with full overlap; exposed {1e3 * (weak[k] - weak[1]):.0f} ms')
```

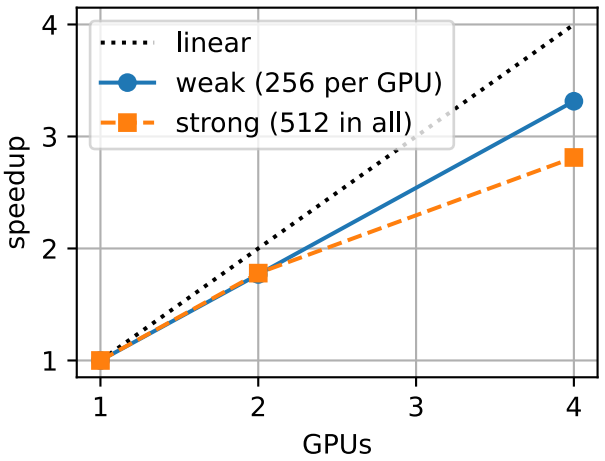
```
1 GPU(s):  2285 examples/s, speedup 1.00, efficiency 100%
2 GPU(s):  4039 examples/s, speedup 1.77, efficiency 88%
          predicted 86% without overlap, 97% with full overlap; exposed 15 ms
4 GPU(s):  7573 examples/s, speedup 3.31, efficiency 83%
          predicted 80% without overlap, 95% with full overlap; exposed 23 ms
```

バス帯域幅が約 2.4 GB/s の場合、オーバーラップなしの 45 MB のオールリデュースには 2 基の GPU で約 18 ms、4 基で約 28 ms かかる。弱スケーリングの実測では、4 基の GPU は単一 GPU の 3.1–3.3× のスループット（スケーリング効率 77〜83%）を達成した。これは**オーバーラップなし**の予測とよく一致しており、完全オーバーラップのバケット予測を大きく下回っている。

強スケーリング（グローバルバッチを 512 例に固定するため、通信量が一定のまま k 基の各 GPU が $512/k$ 例を処理する設定）のもとでは、$k = 4$ で効率がさらに低下する。

```
strong = {k: step_time(k, 512 // k) for k in ks}
for k in ks:
    print(f'{k} GPU(s), {512 // k} examples each: speedup '
          f'{strong[1] / strong[k]:.2f}, efficiency '
          f'{strong[1] / strong[k] / k:.0%}')
d2l.plot(ks, [ks, [k * weak[1] / weak[k] for k in ks],
              [strong[1] / strong[k] for k in ks]], 'GPUs', 'speedup',
         legend=['linear', 'weak (256 per GPU)', 'strong (512 in all)'],
         fmts=('k:', '-o', '--s'))
```

```
1 GPU(s), 512 examples each: speedup 1.00, efficiency 100%
2 GPU(s), 256 examples each: speedup 1.78, efficiency 89%
4 GPU(s), 128 examples each: speedup 2.81, efficiency 70%
```



13.6.2.1 バケットモデルの仮定

なぜオーバーラップによって通信がほとんど隠蔽されなかったのだろうか。バケットモデルは、(1) 各バケットのオールリデュースが逆伝播の最中に開始すること、(2) NCCL が単体ベンチマークと同等の速度で動作すること、(3) NCCL が並行して動作しても逆伝播の計算カーネルが減速しないこと、の 3 つを仮定している。GPU 0 を 3 つの構成（1 基の GPU、通信なしの 2 基の GPU、通信ありの 2 基の GPU）にわたってプロファイリングすることで、これら 3 つの仮定すべてを検証できる。

```
def profile_step(k, b, communicate=True):
    """Seconds per DDP step on k GPUs with b examples per GPU, and device
    milliseconds per step on GPU 0 of computation and of NCCL's kernels;
    without communication, every GPU steps on its own shard's gradient."""
    ranks = d2l.torchrun(TRAIN_DDP, k, b, 50, random.getrandbits(32),
                         int(communicate), 1)
    return (max(r['step_time'] for r in ranks), ranks[0]['compute'],
            ranks[0]['comm'])
```

GPU あたり $b = 256$ 例で、単一 GPU、勾配同期なしの 2 基の GPU、オールリデュースありの 2 基の GPU という 3 つの構成にわたって `profile_step` を実行し、実経過ステップ時間をデバイスの計算時間と NCCL カーネル時間に分離する。

```
if d2l.num_gpus() >= 2:
    for k, communicate in ((1, True), (2, False), (2, True)):
        t, compute, comm = profile_step(k, 256, communicate)
        name = f'{k} GPU' + ('s' if k > 1 else '') + (
            '' if communicate else ' without communication')
        print(f'{name + ":":30s}{1e3 * t:6.1f} ms per step; GPU 0 computes '
              f'{compute:6.1f} ms, communicates {comm:5.1f} ms')
```

```
1 GPU:                          112.1 ms per step; GPU 0 computes  111.9 ms, communicates   0.0 ms
2 GPUs without communication:   111.9 ms per step; GPU 0 computes  111.5 ms, communicates   0.0 ms
2 GPUs:                         126.8 ms per step; GPU 0 computes  124.6 ms, communicates  20.0 ms
```

通信なしの 2 基の GPU は単一 GPU と全く同じ速度でステップを実行し、NCCL のカーネルは 20〜22 ms 動作する（18 ms の単体ベンチマークと整合する）。破綻しているのは 3 つ目の仮定である。すなわち、**NCCL が逆伝播と並行して動作すると、計算カーネル自体が 8〜13 ms 遅くなる**のである。ピアツーピアアクセスを持たない GPU では、NCCL は独自の GPU SM カーネルを使ってホストメモリ経由でデータをコピーするため、本来であれば逆伝播が利用するはずのストリーミングマルチプロセッサとメモリ帯域幅を占有してしまう。

非公式の NCCL フラグ（`NCCL_SHM_USE_CUDA_MEMCPY=1`、NCCL 2.28〜2.30 に存在）を用いると、単体のマイクロベンチマークにおいては、これらのホストメモリ経由のコピーを GPU の DMA コピーエンジンへオフロードできる。

同じフラグを DDP で使用すると、オールリデュースが逆伝播計算とオーバーラップした際にデッドロックを引き起こす。

```
if d2l.num_gpus() >= 2:
    # An undocumented parameter of NCCL's transport through host memory. It
    # exists in NCCL 2.28.9 (bundled with PyTorch 2.11) and 2.30.4 (with
    # jaxlib 0.12); the library of NCCL 2.31.2 no longer contains it
```

```
    beta_ce = allreduce_beta({'NCCL_SHM_USE_CUDA_MEMCPY': '1'})
    print(f'bus bandwidth {1e-9 / beta_link:.2f} GB/s by default, '
          f'{1e-9 / beta_ce:.2f} GB/s with the copy engines '
          f'({beta_link / beta_ce:.1f}x)')
```

```
bus bandwidth 2.44 GB/s by default, 11.53 GB/s with the copy engines (4.7x)
```

NVLink（片方向あたり 450〜900 GB/s、13.2.4.2 節）で接続されたデータセンタ向け GPU 上では、ResNet-18 の 45 MB のオールリデュースは 1 ミリ秒を優に切る時間で完了し、ほぼ線形のスケーリングを達成する。しかし理想的なスケーリングが得られたとしても、データ並列は依然としてパラメータとオプティマイザ状態の全 $16N$ バイトをすべてのカード上に複製する。13.7 節では、この状態を複数の GPU に分割する手法を扱う。

13.6.3 まとめ

実用データ並列は、NCCL を介して同期する k 個の独立したプロセス、あるいはコンパイラ分割型の単一プログラムのいずれかによって k 基の GPU を駆動し、勾配を 25〜30 MiB のバケットにまとめてオールリデュースを逆伝播とオーバーラップさせる。単一 GPU の参照結果に対する 1 ステップチェックを行えば、バッチ正規化がシャードごとの統計量を用いているのか、追加の集団通信を介して大域統計量を用いているのかが明らかになる。ホストメモリ経由で通信する 4 基の RTX 4090（バス帯域幅 2.4 GB/s）では、弱スケーリングのもとで単一 GPU の 3.1–$3.3\times$ のスループットが得られたが、カーネルプロファイリングの結果、ホスト中継の NCCL カーネルによる SM 競合が、通信と逆伝播計算のオーバーラップで節約された時間の大部分を帳消しにしていることがわかる。

13.6.4 演習問題

1. [code] **データの分割**。`TRAIN_DDP` では、ランク r は共有されたランダム順列から $(sk+r)b$ を始点とするスライスを各ステップ s で取得する。
   1. このオフセットから `rank` を取り除き、すべてのプロセスが同じ b 個の例を取得するようにせよ。$k=2$ における `step_time` と平均化された勾配がどのように振る舞うかを予測し、各ランクでミニバッチのチェックサムを出力して検証せよ。
   2. オフセットを元に戻したうえで、各ランクに異なる乱数シードを与えよ。ミニバッチ分割のどの性質が失われるか。
   3. 最初のステップでランク間に同一のデータシャードが存在することを検出する、軽量な実行時アサーションを提案せよ。
2. **弱スケーリングの外挿**。k 基の GPU 上での 1 ステップに、単一 GPU 時間 t_1 と、リング通信量 $2(k-1)S/k$ に比例する露出通信時間がかかるとする。
   1. 弱スケーリング効率 $E(k) = t_1/t_{\text{step}}(k)$ を単一の定数を用いて表し、実測された $E(2)$ でその定数を較正したうえで、$E(4)$ に対する予測を 13.6.2 節の実測値と比較せよ。
   2. a 個のマイクロバッチ（GPU あたり 256 例、最後のマイクロバッチでのみ通信する）にわたって勾配累積を行う場合、整数 a のうち $E(4)$ を 95% 以上に引き上げる最小の値を求め、グローバルバッチサイズがどのように変化するか述べよ。
3. [code] **バケットサイズ**。●バケットオーバーラップ（Figure 13.20）を次の仮定のもとでモデル化せよ。時間 t_b の逆伝播が等サイズの n 個のバケットを時刻 $i\,t_b/n$ $(i=1,\dots,n)$ に順次放出し、各バケットのオールリデュースが逐次的に $\alpha + \tau/n$ を要するものとする（ただし $\tau = S\,\beta_{\text{link}}$）。

1. 逆伝播終了後の露出通信時間が次の式で与えられることを示せ。

$$t_{\text{exposed}}(n) = \max\Big(\alpha + \frac{\tau}{n},\ n\alpha + \tau - \frac{n-1}{n}\,t_b\Big), \tag{13.13}$$

さらに、最適な n を $\tau < t_b$ の場合と $\tau > t_b$ の場合のそれぞれについて求めよ。

2. `bucket_cap_mb`（DDP の引数）を $k = 2$ において 1 MB から 100 MB まで変化させ、実測された露出通信時間をこのモデルと比較せよ。

4. [code] **同期バッチ正規化**。●13.6.1 節の 1 ステップチェックにより、DDP はデフォルトで各シャードを局所統計量によって正規化することがわかった。
   1. `model`（`CHECK_DDP` 内）を `nn.SyncBatchNorm.convert_sync_batchnorm` で変換してから DDP でラップし、同じチェックを繰り返せ。今度はどちらの単一 GPU 参照結果と一致するか。
   2. `SyncBatchNorm` によって 1 ステップあたりに追加される集団通信の回数を数え、$k = 2$ および $k = 4$ での `step_time` を測定したうえで、その所要時間がバイト数に β_{link} を掛けた値を上回る理由を説明せよ。

13.7 訓練状態のシャーディング

13.4 節で見たように、fp32 の Adam を用いた訓練では、重み、その勾配、および Adam の 2 つのモーメント推定値にそれぞれ 4 バイトずつ、パラメータあたり合計 16 バイトを保持する。18 ブロック・1B パラメータの GPT では、この固定状態だけで 1 トークンも読み込む前に 24 GiB カードのうち 15.1 GiB を占め、さらにデータ並列（data parallelism、13.5 節、13.6 節）はこの 16 バイトすべてを各 GPU に複製する。GPT の深さを 2 倍の 36 ブロック（1.92B パラメータ、状態サイズ 28.6 GiB）にすると、1 基の GPU と同様、4 基の 24 GiB GPU によるデータ並列でも訓練できなくなる。

1 枚のカードに収まらないモデルを k 基の GPU で訓練できるだろうか。答えは可能である。なぜなら、GPU がブロックの重み全体を必要とするのはそのブロックの順伝播または逆伝播を計算している間だけであり、全勾配や完全なオプティマイザ状態（optimizer state）はそもそも一切必要としないからである。したがって、k 基の各 GPU は重み、勾配、およびオプティマイザのモーメントを $1/k$ ずつ保持し、各ブロックの実行直前にその重みをオンザフライで集め、各ブロックの勾配が生成され次第すぐに reduce-scatter すればよい。

```
import json
```

まず、形状のみを保持するメタデバイス上で、36 ブロックの GPT が持つ $16N$ バイトの訓練状態（training state）が単一の RTX 4090 の容量 24 GiB を超えることを確認しよう。

```
config = dict(vocab_size=50257, num_hiddens=2048, num_heads=16,
              num_blks=36, pos='learned')
k = 2                       # GPUs, from the all-gather below on
with torch.device('meta'):  # Shapes and dtypes only, no data
    N = d2l.num_params(d2l.GPT(**config))
print(f'{N / 1e9:.2f} billion parameters, 16N = {16 * N / 2**30:.1f} GiB')
if d2l.num_gpus():
    total = torch.cuda.mem_get_info()[1]
    print(f'one GPU holds {total / 2**30:.1f} GiB')
```

```
1.92 billion parameters, 16N = 28.6 GiB
one GPU holds 23.5 GiB
```

13.7.1 オプティマイザ状態、勾配、重みの分割

標準的なデータ並列では、k 基すべての GPU が Adam のモーメント推定値の同一なコピーを保持し、同一のオプティマイザ更新を実行する。**ZeRO**（**Zero Redundancy Optimizer**）ファミリー (Rajbhandari et al. 2020) は、累積的な 3 つの段階によってこの冗長性を排除する（Table 13.5）。

- **ZeRO Stage 1** はオプティマイザ状態のみを分割する。各 GPU はパラメータの $1/k$ について Adam の 2 つのモーメントを保持する（パラメータあたり $8 + 8/k$ バイト）。allreduce（= reduce-scatter + all-gather、13.5.3 節）の代わりに、GPU 間で勾配を reduce-scatter することで、各 GPU が自身の $1/k$ シャードに対応する合算済み勾配を受け取り、自身のパラメータを更新したうえで、更新後の重みを all-gather する。移動するバイト数はデータ並列と完全に同じである。
- **ZeRO Stage 2** は勾配も分割する（パラメータあたり $4 + 12/k$ バイト）。逆伝播中に各勾配バケットが埋まり次第すぐに reduce-scatter され、自身が所有しない $1 - 1/k$ の勾配シャードは解放される。これもデータ並列に対する追加の通信バイトコストはゼロである。
- **ZeRO Stage 3**、すなわち**完全シャーディングデータ並列**（Fully Sharded Data Parallelism、FSDP）(Y. Zhao et al. 2023) は、重みもシャーディングする（パラメータあたり $16/k$ バイト）。順伝播または逆伝播でブロックを実行する直前に all-gather でそのブロックの重み全体を組み立て、ブロックが完了した直後に所有しない重みシャードを解放し、そのブロックの勾配は reduce-scatter する（Figure 13.21）。

Table 13.5: k 基の GPU 上で fp32 の Adam を用いた場合の逆伝播終了時点における GPU あたりの訓練状態、および $k = 2$ における 36 ブロック・19.2 億パラメータの GPT での値。

各 GPU の保持内容	重み	勾配	Adam 状態	パラメータあたりバイト数	GPT（$k = 2$）
データ並列	すべて	すべて	すべて	16	28.6 GiB
ZeRO Stage 1	すべて	すべて	$1/k$	$8 + 8/k$	21.4 GiB
ZeRO Stage 2	すべて	$1/k$	$1/k$	$4 + 12/k$	17.9 GiB
ZeRO Stage 3（FSDP）	$1/k$	$1/k$	$1/k$	$16/k$	14.3 GiB

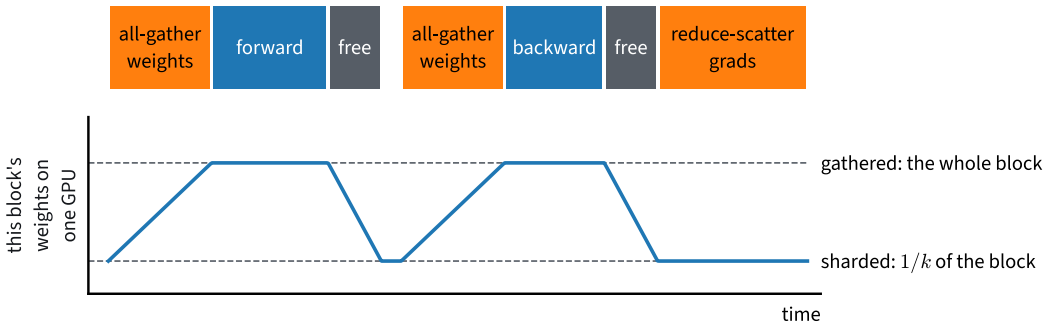


Figure 13.21: 1 基の GPU 上における FSDP 下での 1 ブロックの重みのライフサイクル。使用していない間、ブロックはシャーディングされており、各 GPU はその $1/k$ を保持する（他のブロックは常にシャーディングされたままである）。順伝播のために all-gather がブロック全体を組み立て、終了後に解放する。逆伝播のために 2 回目の all-gather が再びブロックを組み立てて終了後に解放し、reduce-scatter によって各 GPU には担当分の勾配が残る。

FSDP はステップごとに $S = 4N$ バイトに対する 2 回の all-gather（順伝播と逆伝播）と 1 回の reduce-scatter を実行するため、リング allreduce が 2 回の集団通信（collective）フェーズで済むところを 3 フェーズ要し、**データ並列の $1.5\times$ の通信量**を移動させる。Table 13.6 に並列訓練で使われる主要な 4 つの集団通信演算をまとめ、Figure 13.22 にそれぞれが 2 基の GPU 間で何を移動させるかを示す。

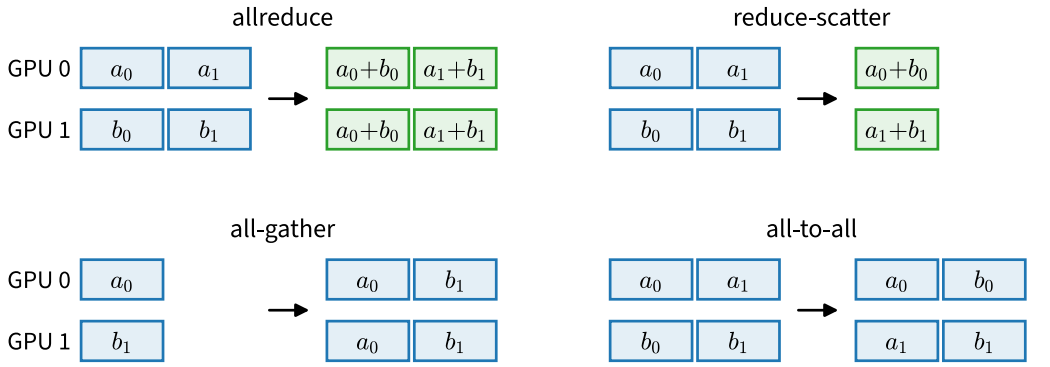


Figure 13.22: 2 基の GPU 間における実行前後のデータ移動として見た 4 つの集団通信。GPU 0 はチャンク a_0, a_1、GPU 1 はチャンク b_0, b_1 を初期状態とし、all-gather では GPU g は自身のシャードであるチャンク g のみを初期状態とする。元のチャンクを青、和を緑で示す。

Table 13.6: データ並列訓練およびシャーディング訓練を支える集団通信演算。

集団通信	各 GPU が最終的に得る結果	使用箇所
allreduce	全体の和	データ並列における勾配
reduce-scatter	和の $1/k$ シャード 1 つ	ZeRO および FSDP における勾配
all-gather	全シャードを連結したもの	ZeRO および FSDP における重み
all-to-all	各 GPU からの異なるスライス	混合エキスパート（11.7 節）におけるトークンルーティング

13.7.2 メモリと通信

$k = 2$ 基の GPU 上で、GPU あたり 1,024 トークンの系列 2 本とブロックごとの活性化チェックポイント（activation checkpointing、13.4.3 節）を用いて 36 ブロックの GPT を FSDP で実行したときのピークメモリとステップ時間を予測してみよう。

- **メモリ：**シャーディングされた訓練状態は GPU あたり $16N/2 = 14.3$ GiB を占める。チェックポイント保存された活性化値にトークンあたり約 1 MiB（2,048 トークンで約 2 GiB）を要し、さらにその時点で集められている 1〜2 個のブロック（各約 0.2 GB）を加えても、GPU あたりのピークメモリは高々**約 17 GiB** となり、24 GiB を十分に下回る。
- **通信：**3 つの集団通信フェーズはそれぞれ GPU あたり $(k-1)S/k$ バイトを送信するため、合計の通信時間は次式となる。

$$t_{\text{comm}} = \frac{3(k-1)}{k}\, S\, \beta_{\text{link}}. \tag{13.14}$$

FSDP の重要なスケーリング特性に注目しよう。**GPU を増やすと GPU あたりの状態メモリは k で割られて減る一方、各 GPU がステップごとに送信するバイト数は $1.5S$（$k = 2$ の**

とき）から $3S$（k が大きいとき）へと増加する。1 ステップに占める通信の割合を減らせる手段は、GPU あたりに処理するトークン数を増やすか、より高速なインターコネクトを使うことだけである。

まず、256 MiB の all-gather における β_{link} を $k = 2$ 基の GPU 間で計測する。

```
GATHER = r'''
import json, torch
import torch.distributed as dist
from d2l import torch as d2l

_, k, device = d2l.init_distributed()
x = torch.zeros(2**26 // k, device=device)  # This GPU's part
y = torch.empty(2**26, device=device)       # 256 MiB, gathered
t = d2l.Benchmark(lambda: dist.all_gather_into_tensor(y, x), repeats=10).time
print(json.dumps({'beta_link': t / ((k - 1) / k * y.nbytes)}))  # Per byte sent
dist.destroy_process_group()
'''
assert d2l.num_gpus() >= k, 'The cells from here on need two GPUs'
beta_link = max(r['beta_link'] for r in d2l.torchrun(GATHER, k))
print(f'bus bandwidth = {1e-9 / beta_link:.1f} GB/s in an all-gather on {k} '
      f'GPUs (beta_link = {1e9 * beta_link:.2f} ns/byte)')
```

```
bus bandwidth = 2.3 GB/s in an all-gather on 2 GPUs (beta_link = 0.44 ns/byte)
```

実測した β_{link} と RTX 4090 の tf32 ピーク性能である 82.6 TFLOP/s（`peak_flops`）を用いて、GPU あたりの状態メモリ（$16N/k$）、通信時間（$3(k-1)S\beta_{\text{link}}/k$）、およびチェックポイント保存時の計算時間（$8NbT/P$）に関する机上予測を評価しよう。

```
b, T = 2, 1024        # Sequences of T tokens per GPU and step
S = 4 * N             # Bytes of the fp32 weights, and of the gradients
peak_flops = 82.6e12  # tf32 FLOP/s of the RTX 4090
print(f'state per GPU, 16N/k: {16 * N / k / 2**30:.1f} GiB')
print(f'communication, 3(k-1)S beta_link/k: '
      f'{3 * (k - 1) / k * S * beta_link:.1f} s per step')
print(f'arithmetic at the peak, 8N bT/P: '
      f'{8 * N * b * T / peak_flops:.2f} s per step')
```

```
state per GPU, 16N/k: 14.3 GiB
communication, 3(k-1)S beta_link/k: 5.0 s per step
arithmetic at the peak, 8N bT/P: 0.38 s per step
```

ホストメモリ経由のバス帯域幅が約 2.2 GB/s の場合、ステップあたり $1.5S = 11.5$ GB のデータ移動には約 5 s かかるのに対し、2,048 トークン分の算術演算はわずか約 0.4 s で終わる。0.4 s の計算では 5 s の通信を隠蔽できないため、1 ステップには約 5 s かかるはずである。

13.7.3 2 基の GPU によるシャーディング訓練

それでは、シャーディングされた 36 ブロック GPT のステップを 2 基の GPU 上で実行する。その際、どの単一 GPU も 28.6 GiB の状態全体を一度に実体化しないようモデルを構築する。

PyTorch では、`'meta'` デバイス上でモデルを構築し、チェックポイント化した各ブロックと最上位モデルを `fully_shard`（`torch.distributed.fsdp`）でラップしたうえで、`model.to_empty(`

device=device) によりこのランクのシャードのみを実体化し、インプレースで初期化する。

```
TRAIN_FSDP = r'''
import json, sys, torch
import torch.distributed as dist
from torch import nn
from torch.nn import functional as F
from torch.distributed.fsdp import fully_shard
from torch.utils.checkpoint import checkpoint
from d2l import torch as d2l

class Checkpointed(nn.Module):
    """A block whose activations the backward pass recomputes."""
    def __init__(self, blk):
        super().__init__()
        self.blk = blk

    def forward(self, X, **kwargs):
        return checkpoint(self.blk, X, use_reentrant=False, **kwargs)

config, b, T = json.loads(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3])
_, k, device = d2l.init_distributed()
torch.set_float32_matmul_precision('high')  # fp32 products on tf32 cores
torch.manual_seed(0)        # The same seed in every process
with torch.device('meta'):  # Shapes only: no GPU holds the whole model
    model = d2l.GPT(**config)
model.blks = nn.ModuleList(Checkpointed(blk) for blk in model.blks)
for blk in model.blks:      # Each block is gathered and released on its own
    fully_shard(blk)
fully_shard(model)          # The rest: the embeddings and the final norm
model.to_empty(device=device)  # This GPU's shards, not yet initialized
with torch.no_grad():       # Initialize the shards as d2l.GPT does
    for m in model.modules():
        if isinstance(m, nn.RMSNorm):
            m.reset_parameters()
    for emb in (model.token_emb, model.pos_emb):
        nn.init.normal_(emb.weight, std=0.02)
    model.init_weights()
opt = torch.optim.Adam(model.parameters(), lr=1e-4, fused=True)
# The same tokens on every GPU: they only set the shapes of the step
X = torch.randint(0, config['vocab_size'], (b, T), device=device)

def step():
    loss = F.cross_entropy(model(X).flatten(0, 1), X.flatten())
    loss.backward()
    opt.step()
    opt.zero_grad()

step()                      # Warm-up; creates Adam's moment estimates
times = [d2l.Benchmark(step, warmup=0, repeats=1).time for _ in range(3)]
kernels = d2l.kernel_times(step, steps=1)
comm = sum(ms for name, ms in kernels.items() if 'nccl' in name.lower())
print(json.dumps({'peak': torch.cuda.max_memory_allocated(), 'times': times,
```

```
                  'comm': comm / 1e3}))
    dist.destroy_process_group()
    '''
```

続いて TRAIN_FSDP を k 個のプロセスにわたって d2l.torchrun で起動し、ランク 0 からピーク確保メモリ（peak_bytes）、ステップ所要時間（times）、および NCCL 通信時間（comm）を収集する。

```
ranks = d2l.torchrun(TRAIN_FSDP, k, json.dumps(config), b, T)
peak_bytes = max(r['peak'] for r in ranks)
times, comm = ranks[0]['times'], ranks[0]['comm']  # GPU 0
```

最後に、実測された GPU あたりのピークメモリ、ステップ所要時間の中央値、および GPU 0 が NCCL 通信カーネルに費やした時間を出力する。

```
print(f'per GPU: peak memory {peak_bytes / 2**30:.1f} GiB; '
      f'{np.median(times):.2f} s per step ({min(times):.2f} to '
      f'{max(times):.2f})\nof each step, GPU 0 spends {comm:.2f} s in '
      f'communication kernels')
```

```
per GPU: peak memory 16.0 GiB; 5.73 s per step (5.70 to 5.74)
of each step, GPU 0 spends 5.69 s in communication kernels
```

両方の GPU はカードあたりピーク約 16 GiB（予測上界の 17 GiB 未満）で 36 ブロックのモデルを保持しており、各ステップの所要時間は約 5.5 s で、GPU 0 はステップの 90% 超を NCCL 通信カーネルに費やしている。これは机上予測である 5 s から 1/6 以内に収まっている。NVLink 5（片方向あたり 900 GB/s）を用いれば、同じ 11.5 GB の集団通信はわずか約 13 ms で完了し、0.4 s の計算時間の 1/30 未満となる。

この比較から、**いつ**シャーディングすべきかもわかる。訓練状態が妥当なサイズのマイクロバッチとともに単一 GPU に収まるモデル（18 ブロックの GPT など）では、データ並列と勾配累積（gradient accumulation、グローバルステップごとに 1 回だけ通信する）を組み合わせたほうが、PCIe 経由の FSDP（マイクロバッチごとに重みを 2 回再収集する）よりもはるかに高速に訓練できる。

13.7.4 まとめ

データ並列は fp32 Adam の状態 $16N$ バイトすべてを各 GPU に複製する。ZeRO はこの状態を k 基の GPU に分割する。Stage 1 はオプティマイザのモーメントをシャーディングし（パラメータあたり $8 + 8/k$ バイト）、Stage 2 は勾配もシャーディングするが（パラメータあたり $4 + 12/k$ バイト）、どちらもステップあたりに移動するデータ量はリング allreduce と完全に同じ $2(k-1)S/k$ バイトである。Stage 3（FSDP）は重みもシャーディングし（パラメータあたり $16/k$ バイト）、順伝播と逆伝播のたびに各ブロックの重みを収集し勾配を reduce-scatter することで、データ並列の $1.5\times$ の通信量（GPU あたり $3(k-1)S/k$ バイト）で動作する。シャーディングにより、単独のカードには収まらない 28.6 GiB の状態を持つ 36 ブロックの GPT を、2 基の 24 GiB GPU で訓練できるようになる。

13.7.5 演習問題

1. **シャーディングのコスト。**●GPU あたり・ステップあたりの通信量を、fp32 の重みまたは勾配 1 セット分のバイト数である $S = 4N$ を単位として数えよ。

   1. ZeRO Stage 3 のある変種では、収集した各ブロックを順伝播から逆伝播が完了するまでメモリ上に保持し続ける。これによりどの集団通信が不要になり、Table 13.5 のメモリ削減量のうちどれだけを手放すことになるか。
   2. 混合精度 FSDP は、各 GPU のマスター重みと Adam シャードを fp32 に保ったまま、重みの all-gather と勾配の reduce-scatter を bf16 で実行する。ステップあたりの転送バイト数を fp32 データ並列と比較し、パラメータあたりのメモリ量を Table 13.5 の最終行と比較せよ。
   3. $k = 2$ 基の GPU 上での 36 ブロック GPT において、実測した β_{link} のもとで、チェックポイント保存時の計算時間（83 TFLOP/s でトークンあたり $8N$ FLOPs）が通信時間 Equation 13.14 と釣り合うには、1 ステップで GPU あたり何トークン必要か。その活性化値はカードに収まるか。また、FSDP がマイクロバッチごとに重みを再収集する場合に勾配累積が役に立たないのはなぜか。

2. [extended] **オプティマイザ状態のシャーディング。**●ZeRO Stage 1 (Rajbhandari et al. 2020) を、`TRAIN_DDP`（13.6 節）および `torch.optim.Adam` を出発点として手作業で実装せよ。
   1. パラメータテンソルをランク間で分割し、各ランクがパラメータの約 $1/k$ を所有して自身の所有テンソルに対してのみ Adam を実行したうえで、更新後の各テンソルをその所有元から `dist.broadcast` でブロードキャストするようにせよ。数ステップ後にパラメータが通常の DDP Adam と一致することを検証し、`torch.distributed.optim.ZeroRedundancyOptimizer` と比較せよ。
   2. シャーディングの有無それぞれについて $k = 2$ および $k = 4$ での確保済み GPU メモリとピーク GPU メモリを計測し、DDP の allreduce と所有元からのブロードキャストによる総送信バイト数を、Stage 1 の reduce-scatter と all-gather による総送信バイト数と比較せよ。

3. [code] **テンソル並列とデータ並列の比較。**●MLP ブロック $\mathbf{Y} = \phi(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2$（ただし $\mathbf{X} \in \mathbb{R}^{n \times d}$、$\mathbf{W}_1 \in \mathbb{R}^{d \times 4d}$、$\mathbf{W}_2 \in \mathbb{R}^{4d \times d}$）を k 基のデバイスに分散することを考える。テンソル並列（Figure 13.15、(Shoeybi et al. 2019)）では、$\mathbf{W}_1$ を列方向に、$\mathbf{W}_2$ を行方向に分割する。
   1. 各デバイスが通信なしに $n \times d$ の部分出力を計算でき、その k 個の部分出力を合算すると $\mathbf{Y}$ が得られることを示せ。順伝播にはどの集団通信が必要で、逆伝播で $\nabla_{\mathbf{X}}$ を求めるにはどの集団通信が必要か。
   2. デバイスあたり・ステップあたりに送信される値の個数をテンソル並列とデータ並列で比較し、テンソル並列のほうがデータ送信量が少なくなるトークン数 n の上限を求めよ。それにもかかわらず、テンソル並列が高速なノード内 NVLink ドメインに限定されるのはなぜか。
   3. `parallelize_module` を `ColwiseParallel()` および `RowwiseParallel()` とともに 1D の `DeviceMesh` 上で用い、`CommDebugMode` によって順伝播と逆伝播の集団通信の回数を数えよ。

4. **次ブロックのプリフェッチ。** FSDP における順伝播で、1 ブロックの all-gather 時間を g、計算時間を c とする。オーバーラップなしでは L ブロックに $L(g + c)$ かかるが、プリフェッチではブロック $l + 1$ の all-gather をブロック l の計算中に発行する。
   1. プリフェッチによって順伝播の時間が $g + (L - 1)\max(g, c) + c$ に短縮されることを示し、どのようなスケジュールでも $L\max(g, c)$ より速くできない理由を説明せよ。
   2. g と c の値を 36 ブロック GPT の 1 ブロック（50M パラメータ、GPU あたり 2,048 トークン、83 TFLOP/s）について本節の PCIe リンクと NVLink 5（900 GB/s）のそれぞれで求め、各インターコネクトでプリフェッチがどれだけ時間を短縮するか述べよ。

13.8 ケーススタディ：Transformer の最適化

本章の各節では、訓練ステップを遅くする原因とそれを取り除く変更を 1 つずつ切り分けてきた。具体的には、ホストディスパッチのオーバーヘッド、小さなカーネルによるメモリトラフィック、ビット幅の広い数値形式、そして単一 GPU の限界である。実際の訓練ステップではこれら複数が同時に発生し、1 つを取り除くとボトルネックが別の箇所へと移る。本節では実務家と同じように 1 つのモデルを取り上げ、その訓練ステップをどこまで高速化できるか、またどの順序で変更を適用すべきかを検証する。

対象とするモデルは、11.2 節の GPT の幅を 2 倍の 512 に広げたもの（1900 万パラメータ）である。128 バイトのウィンドウ 64 個（8,192 トークン）に対する 1 回の訓練ステップは、約 $6 \times 1.9 \times 10^7 \times 8{,}192 \approx 9 \times 10^{11}$ FLOPs の演算を実行する。内訳は、1 パラメータ・1 トークンあたり順伝播で 2 FLOPs、逆伝播で 4 FLOPs である。RTX 4090（bf16 で 165 TFLOP/s）では、この算術演算に必要な時間は 6 ms 未満である。ところが、AdamW を用いた通常の fp32 ステップには約 25 ms かかる。数値形式だけで約 6 ms（その 19 ms の差のうち）を占めており、これは tf32 テンソルコアが bf16 の半分の速度で動作するため 11 ms を要するからである。残りの 13 ms ほどは、テンソルコアがピーク未満で動作しているかアイドル状態にある時間である。

ここでも 13.1 節の手順に従い、演算量と転送量を数え、予測し、計測し、その差を説明する。各変更を加える前に直前のステップのプロファイルから得られる高速化の上限を求め、適用後に実測値と比較する。各変更はそれまでの変更の上に累積的に積み重ねて計測する。本節の最後には、最速化したステップでもモデルが正しく訓練できることを確認し、残りの実行時間の内訳を明らかにする。

```
import os
import re
from torch.utils.checkpoint import checkpoint
```

```
torch.set_float32_matmul_precision('high')  # fp32 products on tf32 cores
# Detach PyTorch's profiler from the GPU when a profile ends; without this,
# every kernel launch after the first profile stays about 30% slower
os.environ['TEARDOWN_CUPTI'] = '1'
```

13.8.1 実験設定

本節のすべての変更は同一の基準で評価するため、モデル、データ、およびスループットのベンチマークは一貫して固定する。

13.8.1.1 モデルとデータ

訓練するのは `d2l.GPT`（11.2 節）であり、それぞれ 8 個のアテンションヘッドを持つ 6 個の Pre-Norm ブロック、回転位置埋め込み（rotary position embedding）、RMSNorm、SwiGLU、および出力層と重み共有された入力埋め込みから構成される。データは、256 バイトの語彙上で WikiText-103 からランダムなオフセットで切り出した 128 バイトのウィンドウである（`d2l.WikiText103`、9.3 節）。

行列積が十分に計算律速（compute-bound）の領域に入るよう、幅を 256 から 512 に倍増させる（13.1 節）。$n \times d$ の活性化行列と $d \times d'$ の重み行列の積は、$2ndd'$ FLOPs の演算を実行し、$b(nd + dd' + nd')$ バイト（1 要素あたり b バイト）を転送する。$n = 8{,}192$ トークン・幅 $d = 512$ の場合、ブロック内の各射影は 125–185 FLOP/byte（tf32、$b = 4$）および 250–370 FLOP/byte

（bf16、$b = 2$）に達し、RTX 4090 のリッジポイント（82 FLOP/byte：tf32、164 FLOP/byte：bf16）を大きく上回る。したがって、より高速な数値形式を使えば、これらの行列積の時間はそのまま短縮される。幅 256 ではリッジポイントをまたぐことになる（演習問題 6）。

270 MB の訓練コーパスは、フラットなバイト配列として GPU メモリに収まる。次のセルでは、このコーパスを GPU 上に配置し、ホスト間の転送なしにデバイス上で直接ミニバッチをサンプリングする `windows` を定義して、モデルを構築する。

```
device = d2l.try_gpu()
data = d2l.WikiText103(batch_size=64, num_steps=128)
corpus = torch.tensor(data.train, device=device)  # One byte per token

def windows(corpus, batch_size, num_steps=128):
    """Endless minibatches of windows at random offsets, each position with
    the next byte as its target."""
    offsets = torch.arange(num_steps + 1, device=corpus.device)
    while True:
        starts = torch.randint(len(corpus) - num_steps, (batch_size, 1),
                               device=corpus.device)
        W = corpus[starts + offsets].long()
        yield W[:, :-1], W[:, 1:]

model = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
num_params = d2l.num_params(model)
print(f'{num_params / 1e6:.1f}M parameters, '
      f'{len(corpus) / 1e6:.0f}M training bytes on the GPU')
```

```
19.0M parameters, 270M training bytes on the GPU
```

1 パラメータあたり 16 バイト（重み、勾配、および 2 つの Adam モーメント）では、19M パラメータの訓練状態が占めるメモリ使用量はわずか 0.3 GiB であり、勾配累積（gradient accumulation）もシャーディングも必要ない（13.4 節、13.7 節）。また、コーパスを GPU 上に保持することで、訓練ステップを入力パイプラインから切り離せる（13.2 節）。

13.8.1.2 スループットの計測

ミニバッチのサンプリング、順伝播と逆伝播、およびオプティマイザの更新を含むエンドツーエンドの訓練スループットを、毎秒トークン数で計測する。

設定項目	値
ハードウェア	RTX 4090 GPU 4 基のうち 1 基（データ並列では 2 基および 4 基）
モデル	`d2l.GPT`：6 ブロック、幅 512、8 ヘッド、バイト語彙、コンテキスト長 128
データ	毎ステップ GPU 上でランダムに抽出する WikiText-103 の 128 バイトウィンドウ
オプティマイザ	AdamW（lr $= 10^{-3}$、重み減衰 0.01）
数値精度	fp32（行列積は tf32、bf16 の導入前まで）
時間計測	`d2l.Benchmark`：ウォームアップ 20 ステップの後、デバイス同期を挟む 3 個のウィンドウ（各 50 ステップ）

設定項目	値
結果	3 ウィンドウの毎秒トークン数の中央値、直前の採用構成に対する比率、ウィンドウ間の相対的なばらつき

tokens_per_second は d2l.Benchmark（13.1 節）を通じてこれらの規則を適用する。record は各結果を保存し、直前の採用構成に対する高速化率と、3 つの計測ウィンドウ間の相対的なばらつき（range/median）を表示する。kept=False を渡した構成は分岐として保存され、後続の構成の土台にはならない。

```
def tokens_per_second(step, batches, warmup=20, repeats=50, num_windows=3):
    """Median training throughput of `step` on minibatches drawn from
    `batches`, over `num_windows` timed windows of `repeats` steps each, and
    the spread of the windows (largest minus smallest) relative to it."""
    X, _ = next(batches)
    speeds = [math.prod(X.shape) / d2l.Benchmark(
                  lambda: step(*next(batches)), warmup if i == 0 else 0,
                  repeats).time for i in range(num_windows)]
    return np.median(speeds), np.ptp(speeds) / np.median(speeds)

results, branches = {}, {}
def record(name, measured, kept=True):
    """Store a measured throughput and print it with its ratio to the last
    kept configuration and the spread of its windows; a configuration that
    is not kept is stored in `branches`."""
    speed, spread = measured
    previous = list(results.values())[-1] if results else speed
    (results if kept else branches)[name] = speed
    print(f'{name}: {speed:,.0f} tokens/s ({speed / previous:.2f}x), '
          f'windows within {spread:.1%}')
```

ホスト側の処理が多いステップはホストのバックグラウンド動作の影響を受けやすく、コンパイル済みやグラフキャプチャ済みのステップよりもウィンドウ間のばらつきが大きくなる。日々のホスト負荷の変動は 1 回の実行における連続したウィンドウ間の変動よりも大きいため、高速化率は小数第 1 位まで読み取り、数パーセント程度の小さな改善はプロファイルがそれを予測している場合にのみ信頼する。

13.8.2 単一 GPU

まず単一 GPU 上で、11.2 節のベースライン実装を出発点として、1 つずつ変更を加えながら訓練ステップを最適化する。

13.8.2.1 ベースライン

ベースラインは、AdamW を用いた 11.2 節の標準的な訓練ステップであり、順伝播、交差エントロピー損失、逆伝播、およびパラメータ更新からなる。

このステップは Eager モードで実行される。すなわち、Python と自動微分が順伝播と逆伝播の各演算を個別のカーネル起動としてディスパッチする一方、AdamW（fused=True）は少数の融合カーネルですべてのパラメータとモーメントを更新する。make_step は訓練ステップをまとめた関数であり、その引数 forward と bf16 は以降の項で導入する変更に対応する。

```
def make_step(model, forward=None, bf16=False, warmup=0):
    """An AdamW training step of `model` on a minibatch (X, Y); `forward`
    replaces the model's forward pass, `bf16` turns on autocast, and a
    `warmup` > 0 raises the learning rate linearly to 1e-3 over as many
    steps."""
    forward = model if forward is None else forward
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                                  weight_decay=0.01, fused=True)
    if warmup:
        schedule = torch.optim.lr_scheduler.LambdaLR(
            optimizer, lambda i: min(1, (i + 1) / warmup))
    def step(X, Y):
        with torch.autocast('cuda', torch.bfloat16, enabled=bf16):
            loss = F.cross_entropy(forward(X).flatten(0, 1), Y.flatten())
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
        if warmup:
            schedule.step()
        return loss
    return step

step = make_step(model)
record('baseline', tokens_per_second(step, windows(corpus, 64)))
```

```
baseline: 317,736 tokens/s (1.00x), windows within 1.9%
```

予想どおり、ベースラインではステップごとのホスト処理の影響により、ウィンドウ間のばらつきが後の構成に比べて約 10 倍大きい。ステップの実行時間の内訳を調べるため、`profile` は `d2l.kernel_times` で 5 ステップ分を記録し、`d2l.show_profile`（13.1.4 節）によって内訳（ステップの実経過時間に対するデバイスの総カーネル時間、融合アテンションカーネルを含む行列積が占める割合、および上位カーネル）を表示し、最終的な時間集計のためにプロファイルを保存する。

```
profiles = {}
def profile(step, batches):
    """Kernel profile of `step`, the configuration recorded last, on
    minibatches drawn from `batches`: printed against the throughput that
    `record` measured, and stored in `profiles` under the same name."""
    name, speed = list(results.items())[-1]
    X, _ = next(batches)
    kernels = d2l.kernel_times(lambda: step(*next(batches)), steps=5)
    d2l.show_profile(kernels, 1e3 * math.prod(X.shape) / speed, rows=4)
    profiles[name] = kernels

profile(step, windows(corpus, 64))
```

```
wall clock 25.78 ms per step, device busy 24.44 ms (95%), matrix products 64%
  3.28 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  3.08 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  2.84 ms  12%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  2.41 ms  10%  vectorized_elementwise_kernel<4, BinaryFunctor<float
```

デバイスはステップのほぼ全期間にわたってカーネルを実行しており、行列積がその時間の約 3 分の 2 を占めている。ホスト側の処理はおおむね追いついているため、このステップのボトルネックはデバイス側の処理にある。

行列積以外の残り 3 分の 1 は、多数の短いカーネル（SwiGLU、回転位置埋め込み、残差加算、RMSNorm）に細分化されている。それぞれがデバイスメモリを読み書きするため、メモリ帯域幅によって律速されている（13.3.2 節）。

13.8.2.2 ホストオーバーヘッドと小さなカーネル

行列積以外に費やされる時間には 2 つの改善策があり、プロファイルからそれぞれの改善幅の上限が求まる。この上限を与えるのが**アムダールの法則**（Amdahl's law）である (Amdahl 1967)。ある変更によってステップの割合 f が s 倍に高速化されるとすると、ステップ全体の高速化率は $1/(1-f+f/s)$ となり、s をどれほど大きくしても $1/(1-f)$ 未満にとどまる。これを 2 つの改善策に当てはめると次のようになる。

1. **ホストオーバーヘッドの削減**は、デバイスがホストを待つアイドル状態の隙間を対象とする（13.1 節）。その高速化率は、ステップに占めるアイドル時間の割合によって上から抑えられる。
2. **カーネル融合**（kernel fusion）は、帯域幅律速（bandwidth-bound）な要素ごとのカーネルの連鎖を統合し、中間配列をオンチップにとどめる。その高速化率は、行列積以外のデバイス時間によって上から抑えられる。

このベースラインは 2 番目の領域にある。GPU がアイドル状態にあるのはステップの数パーセントにすぎない一方、行列積以外のカーネルが 8.8 ms（デバイス時間 24.5 ms のうち）を消費している。仮にコンパイルによって行列積以外のすべてのカーネルとアイドル隙間をゼロにできたとしても、ステップ時間は行列積の 15.6 ms までしか縮まらず、高速化率は高々 1.6× である。`torch.compile`（13.3 節）は、行列積と融合済みオプティマイザ更新をそのまま維持しつつ、順伝播と逆伝播にわたる要素ごとの演算を融合する。また、`mode='reduce-overhead'` と `dynamic=False` を指定すると静的形状の CUDA グラフもキャプチャされ（13.3.4 節）、現時点での数パーセントのホスト待ちアイドル時間を解消するとともに、後の最適化でデバイス実行時間が短縮された際にホストディスパッチがボトルネックになるのを防ぐ。まず、コンパイルしたモデルの出力が Eager 実行時のロジットと一致することを確認する。

```
compiled = torch.compile(model, dynamic=False, mode='reduce-overhead')
X, _ = next(windows(corpus, 64))
high = model(X)
error = ((compiled(X) - high).norm() / high.norm()).item()
print(f'relative difference of the compiled logits: {error:.1e}')
assert error < 1e-4
step = make_step(model, compiled)
record('+ compile', tokens_per_second(step, windows(corpus, 64)))
```

```
relative difference of the compiled logits: 1.5e-05
+ compile: 368,688 tokens/s (1.16x), windows within 0.0%
```

コンパイル後のロジットは有効数字 4 桁まで一致する（わずかな差は、融合カーネル内で fp32 の丸め順序が変わった影響が tf32 の行列積で増幅されるために生じる）。スループットは 10% 以上向上しており、上限である 1.6× の範囲内に十分収まっている。

2 回目のプロファイルから、ステップがどのように変化したかがわかる。

```
profile(step, windows(corpus, 64))
```

```
wall clock 22.22 ms per step, device busy 22.27 ms (100%), matrix products 71%
  4.01 ms  18%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  3.05 ms  14%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  2.83 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  1.44 ms   6%  fmha_cutlassB_f32_aligned_64x64_k64_sm80
```

デバイスは依然としてステップのほぼ全期間で稼働しており、行列積がデバイス時間の約 3 分の 2 以上を占めるようになった。

ステップあたりのデバイス時間は 24.5 ms から 22.3 ms に減少し、カーネル間の隙間も消滅した。Inductor の融合によって 2.2 ms（行列積以外の時間の 4 分の 1）が削られ、CUDA グラフリプレイによってホスト側の起動隙間が約 1 ms 埋められたためである。

13.8.2.3 低精度化

GPU が常時稼働し、デバイス時間の大部分を行列積が占めるようになったため、次に取り組むべきは行列積の高速化である。tf32 から bf16 に切り替えると、RTX 4090 のテンソルコアのピークスループットは 2 倍の 165 TFLOP/s になり、bf16 の活性化値のメモリ使用量は半分になる（13.4 節）。行列積がステップの割合 $f \in [2/3, 0.7]$ を占め、行列積のみが 2 倍に高速化されるとすると、ステップ時間は元の $1 - f/2$ に短縮される。これは約 $1.5\times$ の高速化に相当する（行列積以外のカーネルのデータ転送量も半減すれば $2\times$ に近づく）。

ここでは 13.4.2 節の混合精度（mixed precision）のレシピを適用する。すなわち、行列積（アテンションを含む）と中間の要素ごとの演算は bf16 で実行する一方、マスター重み、勾配、Adam モーメント、正規化、損失、および残差ストリーム（residual stream）は fp32 のまま維持する（残差ストリームを fp32 に保つことで、小さなブロック更新が bf16 の 8 ビット仮数部で丸め消えるのを防ぐ）。bf16 は fp32 と同じ指数部の範囲を持つため、損失スケーリング（loss scaling）は必要ない。時間計測の前に、bf16 の順伝播が fp32 のロジットと bf16 の精度範囲内（約 1%）で一致することを確認する。

`torch.autocast` は演算ごとに混合精度を適用し（5.5 節）、行列積とアテンションの入力を bf16 にキャストする一方、埋め込み、正規化、残差ストリーム、および損失のソフトマックスを fp32 に保つ。

```
X, _ = next(windows(corpus, 64))
with torch.autocast('cuda', torch.bfloat16):
    low = compiled(X).float()
high = compiled(X)
error = ((low - high).norm() / high.norm()).item()
print(f'relative difference of the bf16 logits: {error:.1e}')
assert error < 0.01
step = make_step(model, compiled, bf16=True)
record('+ bf16', tokens_per_second(step, windows(corpus, 64)))
```

```
relative difference of the bf16 logits: 2.0e-03
+ bf16: 643,350 tokens/s (1.74x), windows within 0.1%
```

bf16 のロジットは fp32 と 1% 以内で一致する。

スループットは約 $1.75\times$ に向上し、$1.55\times$ という予測値（71% を占める行列積の時間だけを半減させた場合）を上回る。これは、bf16 の GEMM が tf32 よりも高い稼働率で動作することに

加え、中間の融合カーネルが 4 バイトの浮動小数点数ではなく 2 バイトの bf16 活性化値を転送するためである。なお、`F.scaled_dot_product_attention` はベースラインの時点から融合された FlashAttention カーネルを使用している。

13.8.2.4 ミニバッチの拡大

ミニバッチサイズを大きくすると、ステップあたりの固定コストをより多くのトークンに償却できる。デバイスはすでにステップ全体を通して稼働しているため、償却できるのはトークン数ではなくパラメータ数に比例するデバイス処理のみである。

- AdamW は重み、勾配、2 つのモーメントを読み込み、重みと 2 つのモーメントを書き込む（fp32 で 28 bytes/param）。
- 混合精度では、順伝播で fp32 のマスター重みを bf16 にキャストし、逆伝播で bf16 の重み勾配を fp32 にキャストし直す（12 bytes/param）。

合計すると転送量は 40 bytes/param、すなわち 0.76 GB/step となる。メモリ帯域幅 1.0 TB/s では、この固定処理に約 0.75 ms かかり、これは $1/16$ 程度（12–13 ms かかるバッチサイズ 64 の bf16 ステップに対して）に相当する。ミニバッチを $8\times$ に増やして 64 から 512 ウィンドウ（65,536 トークン）にすると、この固定コストの $7/8$ が償却され、約 5% のスループット向上が予測される。

単一 GPU での改善幅が小さくても、これには十分な価値がある。データ並列（data parallelism）がよくスケールするのは、GPU あたりのステップ時間が勾配のオールリデュースより十分に長い場合に限られるからである（13.8.3 節）。ただし、バッチサイズが臨界バッチサイズ（critical batch size）を超えるとサンプル効率が低下する点には注意が必要である（9.8 節、演習問題 5）。活性化メモリ（activation memory）はバッチサイズに比例して増加するため、重みとオプティマイザ状態を超えるピークメモリ使用量もあわせて記録する（13.4.1.4 節）。

```
    torch.cuda.reset_peak_memory_stats()
    start = torch.cuda.memory_allocated()
    record('+ batch 512', tokens_per_second(step, windows(corpus, 512)))
    memory = (torch.cuda.max_memory_allocated() - start) / 2**30
    print(f'memory beyond weights and optimizer state: {memory:.1f} GiB')
```

```
+ batch 512: 653,141 tokens/s (1.02x), windows within 0.0%
memory beyond weights and optimizer state: 8.6 GiB
```

バッチサイズ 512 では、活性化メモリは GPU の 24 GiB のうち約 3 分の 1 を使用し、スループットは数パーセント向上する（演習問題 2 では、複数のバッチサイズにわたってステップあたりの固定コストとトークンあたりのコストを分離する）。3 回目のプロファイルは、バッチサイズ 512 における内訳を示す。

```
    profile(step, windows(corpus, 512))
```

```
wall clock 100.34 ms per step, device busy 100.33 ms (100%), matrix products 57%
 11.62 ms  12%  ampere_bf16_s1688gemm_bf16_128x128_ldg8_f2f_stages_3
 10.78 ms  11%  ampere_bf16_s16816gemm_bf16_128x64_ldg8_f2f_tn
  7.45 ms   7%  Kernel2<cutlass_80_tensorop_bf16_s16816gemm_relu_bf1
  5.62 ms   6%  triton_poi_fused__unsafe_view_mm_mul_silu_silu_backw
```

デバイスはステップ全体を通して稼働している。行列積（融合アテンションを含む）が時間の 60% 弱を占め、RMSNorm、SwiGLU、回転位置埋め込み、および fp32 と bf16 の間のキャストを行う帯域幅律速の融合カーネルが残りを占める。

13.8.2.5 活性化チェックポイント

活性化チェックポイント（activation checkpointing、13.4.3 節）は、順伝播では各ブロックの入力のみを保存し、逆伝播でブロック内部の活性化値を再計算することでメモリを節約する。バッチサイズ 512 で使用しているのは GPU メモリの 3 分の 1 にすぎないため、ここではチェックポイント化は不要であり、ステップを遅くするだけである。この速度低下は次の 2 通りの方法で予測できる。

1. **FLOP 数による予測**：順伝播の再計算により、トークンあたりのブロック演算量は 6 から 8 FLOPs/param に増えるため、チェックポイントなしのスループットの $6/8 = 0.75$ になると予測される。
2. **順伝播の時間計測による予測**：以下では、単体の順伝播時間 t_{fwd} をバッチサイズ 512 で計測し、$1/(1 + t_{\text{fwd}}/t_{\text{step}})$ を計算する。

```
X512, _ = next(windows(corpus, 512))  # One minibatch of the last step
with torch.no_grad(), torch.autocast('cuda', torch.bfloat16):
    t_forward = d2l.Benchmark(lambda: compiled(X512)).time
```

`t_forward` をステップ全体の所要時間 `t_step`（バッチサイズ 512）と比較することで、すべてのブロックの順伝播を再実行した際に維持されるスループットの割合 `1 / (1 + t_forward / t_step)` が予測できる。

```
t_step = 512 * 128 / results['+ batch 512']  # Seconds per step
print(f'forward pass {1e3 * t_forward:.1f} ms of a {1e3 * t_step:.1f} ms '
      f'step: repeating it keeps at least {1 / (1 + t_forward / t_step):.2f} '
      f'of the throughput')
```

```
forward pass 30.8 ms of a 100.3 ms step: repeating it keeps at least 0.77 of the throughput
```

2 つの推定は一致する。順伝播はステップの約 3 分の 1 を占めるため、これを繰り返しても少なくとも 4 分の 3 のスループットが維持されるはずである（埋め込みと出力射影はチェックポイント化しないため、実際にはもう少し高くなる）。ここで各ブロックを `Checkpointed` でラップし、ロジットの一致を確認したうえで、スループット（`kept=False`）と活性化メモリの両方を計測する。

`Checkpointed` は `torch.utils.checkpoint.checkpoint` を呼び出す。

```
class Checkpointed(nn.Module):
    """A module that keeps only its input for the backward pass."""
    def __init__(self, module):
        super().__init__()
        self.module = module

    def forward(self, X, **kwargs):
        return checkpoint(self.module, X, use_reentrant=False, **kwargs)

remat = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
remat.load_state_dict(model.state_dict())
remat.blks = nn.ModuleList(Checkpointed(blk) for blk in remat.blks)
assert torch.allclose(remat(X), model(X), atol=1e-6)
step = make_step(remat, torch.compile(remat, dynamic=False,
                                      mode='reduce-overhead'), bf16=True)
with torch.no_grad(), torch.autocast('cuda', torch.bfloat16):
```

```
    forward = d2l.kernel_times(lambda: compiled(X512), steps=5)  # Alone
torch.cuda.reset_peak_memory_stats()
start = torch.cuda.memory_allocated()
batches = windows(corpus, 512)
record('+ checkpointing', tokens_per_second(step, batches), kept=False)
memory = (torch.cuda.max_memory_allocated() - start) / 2**30
print(f'memory beyond weights and optimizer state: {memory:.1f} GiB')

def split(kernels):
    """Device ms per step of the matrix products and of all other kernels."""
    products = sum(ms for name, ms in kernels.items()
                   if re.search(d2l.PRODUCT_KERNELS, name, re.I))
    return np.array([products, sum(kernels.values()) - products])

alone = split(forward)
added = (split(d2l.kernel_times(lambda: step(*next(batches)), steps=5))
         - split(profiles['+ batch 512']))
print(f'forward pass alone:  products {alone[0]:4.1f} ms, other kernels '
      f'{alone[1]:4.1f} ms\nadded to the step:   products {added[0]:4.1f} ms, '
      f'other kernels {added[1]:4.1f} ms')
```

```
+ checkpointing: 556,390 tokens/s (0.85x), windows within 0.0%
memory beyond weights and optimizer state: 3.2 GiB
forward pass alone:  products 17.4 ms, other kernels 13.4 ms
added to the step:   products 14.5 ms, other kernels  3.1 ms
```

チェックポイント化により、活性化メモリは約 3 分の 2 削減される一方、単体の順伝播時間から予測された値よりも高いスループットが維持される。

スループットは $0.85\times$ に低下する（下限の $0.77\times$ は上回る）。カーネルの内訳からわかるとおり、ブロック内の行列積の再計算で 14 ms が増加する（出力射影を含む順伝播の全行列積の 17 ms に近い）一方、行列積以外のカーネルの再計算による増加はわずか 3 ms にとどまる（単体での 13 ms に対して）。Inductor が再計算される要素ごとの演算を、それを利用する逆伝播カーネルに直接融合するため、再計算に伴うメモリトラフィックがほぼゼロになるからである。

モデルやバッチサイズがデバイスメモリを超える場合にはチェックポイント化が不可欠であるが、ここではメモリに十分余裕があるため、以降の構成には含めない。

13.8.3 データ並列

最速化した単一 GPU のステップでも、マシンに搭載された 4 基の RTX 4090 のうち 3 基はアイドル状態のままである。複数 GPU で実行する前に、13.5.4 節のコストモデルを使ってデータ並列による高速化率を予測する。

13.8.3.1 高速化率の予測

データ並列（13.5 節）では、k 基の GPU のそれぞれがモデルのレプリカを保持し、自身のミニバッチで勾配を計算して、サイズ S バイトのオールリデュースを c 個のバケットに分けて実行することで勾配を合算する。そのステップ時間は次のとおりである。

$$t_{\text{compute}}(B/k) + c\,\alpha + \frac{2(k-1)}{k}\,S\,\beta_{\text{link}}.$$

ピアツーピアアクセスのないこの PCIe マシンでは、GPU 間の転送はホストメモリを経由して約 2.4 GB/s のバス帯域幅で行われ（13.6 節）、$\beta_{\text{link}} \approx 0.42$ ns/byte となる。弱スケーリング（weak scaling、GPU あたりのバッチサイズを固定）のもとでは、バケットあたりのレイテンシ $c\,\alpha$ は 1 ミリ秒未満であり、数十ミリ秒のステップ時間に比べれば無視できるため、オーバーラップなしのステップ時間は次のように簡略化される。

$$t_{\text{step}}(k) \approx t_{\text{step}}(1) + t_{\text{allreduce}}(k), \tag{13.15}$$

これより、スループットの倍率は $k\,t_{\text{step}}(1)/(t_{\text{step}}(1) + t_{\text{allreduce}}(k))$ となる。2 基目の GPU を追加してスループットが向上するのは、$t_{\text{allreduce}}(2) < t_{\text{step}}(1)$ の場合に限られる。

この 19M パラメータの GPT では、fp32 の勾配が占めるサイズは $S = 76$ MB であり、GPU 2 基では $t_{\text{allreduce}}(2) \approx S\beta_{\text{link}} \approx 31$ ms、GPU 4 基では $t_{\text{allreduce}}(4) \approx 1.5\,S\beta_{\text{link}} \approx 47$ ms となる。

- **GPU あたりバッチ 64 のコンパイル済み fp32**（$t_{\text{step}}(1) \approx 22$ ms）：オールリデュースにステップ自体より長い時間がかかる。Equation 13.15 からは、GPU 2 基では**減速**（$\sim 0.85\times$）し、GPU 4 基でも $\sim 1.3\times$ にとどまると予測される。
- **GPU あたりバッチ 512 のコンパイル済み bf16**（$t_{\text{step}}(1) \approx 90$–$100$ ms）：ステップがオールリデュースよりはるかに長いため、$\sim 1.5\times$（GPU 2 基）および 2.6–2.7×（GPU 4 基）の高速化が予測される。

フレームワークは勾配オールリデュースのバケットを逆伝播とオーバーラップさせるため（Figure 13.20）、通常 Equation 13.15 は高速化率の下限を与え、それを上回るマージンが逆伝播の計算の裏にどれだけ通信を隠蔽できたかを表す。ただし、これは絶対的な下限として保証されているわけではない。NCCL のコピー用カーネルが同じ GPU をめぐって逆伝播と競合し（13.6.2 節）、この競合がマージンを食い潰すことがあるからである。

13.8.3.2 高速化率の実測

`data_parallel(k, bf16, batch_size)` は、k 基の GPU における訓練スループットとあわせて、同じ GPU 群で 76 MB の勾配バッファに対する単体のオールリデュース時間を計測する。

`DistributedDataParallel` は `d2l.torchrun`（13.6 節）を介して GPU ごとに 1 プロセスを実行し、バケット化されたオールリデュースを逆伝播とオーバーラップさせる。CUDA グラフなしでコンパイルし（デバイス実行時間がホストディスパッチを隠蔽するのに十分長いため）、同一条件のもとで $k = 1$ もベンチマークする。

```
DDP_GPT = r'''
import json, sys, torch
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
bf16, batch_size = sys.argv[1] == 'bf16', int(sys.argv[2])
torch.set_float32_matmul_precision('high')
torch._inductor.config.compile_threads = 1  # k processes compile at once
data = d2l.WikiText103(batch_size, 128)
corpus = torch.tensor(data.train, device=device)
model = DDP(d2l.GPT(len(data.vocab), num_hiddens=512).to(device))
step = d2l.make_step(model, torch.compile(model, dynamic=False), bf16)
batches = d2l.windows(corpus, batch_size)
```

```
    seconds = d2l.Benchmark(lambda: step(*next(batches)), 20, 50).time
    grads = torch.zeros(d2l.num_params(model), device=device)
    allreduce = d2l.Benchmark(lambda: torch.distributed.all_reduce(grads),
                              repeats=10).time
    if rank == 0:
        print(json.dumps({'speed': k * batch_size * 128 / seconds,
                          'allreduce': allreduce}))
    torch.distributed.destroy_process_group()
    '''

    # The CUDA graphs of the compiled steps above hold memory on GPU 0, which
    # the process of rank 0 needs: reset the compiler and return the memory
    torch.compiler.reset()
    torch.cuda.empty_cache()

    def data_parallel(k, bf16, batch_size):
        """Tokens per second on k GPUs, and seconds per allreduce of all
        gradients."""
        r = d2l.torchrun(DDP_GPT, k, 'bf16' if bf16 else 'fp32', batch_size)[0]
        return r['speed'], r['allreduce']
```

ここで data_parallel を、短い fp32 ステップ（batch_size=64）と長い bf16 ステップ（batch_size=512）の両方について $k \in \{1, 2, 4\}$ 基の GPU にわたって実行し、オーバーラップなしの予測値 Equation 13.15 と実測された高速化率を比較する。

```
    for bf16, batch_size in ((False, 64), (True, 512)):
        tokens = batch_size * 128  # Per GPU and step
        one, _ = data_parallel(1, bf16, batch_size)
        print(f'{"bf16" if bf16 else "fp32"}, batch {batch_size} per GPU: '
              f'{one:,.0f} tokens/s on one GPU')
        for k in (2, 4):
            if k <= d2l.num_gpus():
                speed, allreduce = data_parallel(k, bf16, batch_size)
                predicted = k * tokens / (tokens / one + allreduce)
                print(f'  {k} GPUs: allreduce {1e3 * allreduce:3.0f} ms, '
                      f'speedup predicted {predicted / one:.2f}x, '
                      f'measured {speed / one:.2f}x')
```

```
fp32, batch 64 per GPU: 364,947 tokens/s on one GPU
  2 GPUs: allreduce  31 ms, speedup predicted 0.83x, measured 0.91x
  4 GPUs: allreduce  48 ms, speedup predicted 1.27x, measured 1.36x
bf16, batch 512 per GPU: 662,971 tokens/s on one GPU
  2 GPUs: allreduce  31 ms, speedup predicted 1.52x, measured 1.63x
  4 GPUs: allreduce  48 ms, speedup predicted 2.69x, measured 2.92x
```

単体のオールリデュースは GPU 2 基で 31 ms、GPU 4 基で 48 ms かかり、リンク速度からの見積もりと 1 ミリ秒以内で一致する。逆方向の 2 つの予測はいずれも的中した。バッチ 64 では、GPU 2 基は GPU 1 基より遅くなり、GPU 4 基でも高速化率は 1.5× 未満にとどまる。一方、bf16 のバッチ 512 では、GPU 4 基で単一 GPU の約 3× のスループット（本節の単一の変更として最大の高速化）に達し、通信のオーバーラップによって実測高速化率がオーバーラップなしの下限 Equation 13.15 を上回っている。（なお、片方向あたり 900 GB/s を持つ NVLink マシンであれば、76 MB のオールリデュースは 1 ミリ秒未満で完了し、バッチ 64 でもほぼ線形にスケールす

る。）

13.8.4 結果

ここで単一 GPU のすべての最適化を並べて総括し、最終的なステップでもモデルが適切に訓練されることを検証したうえで、残りの実行時間の内訳を整理する。

13.8.4.1 累積比較

下の表に単一 GPU の各構成を順に示す。各 `step` の比率は直上の行に対する相対値、`total` はベースラインに対する累積高速化率であり、チェックポイント化は採用しなかった分岐として区切り線の下に示す。`TFLOP/s` は 1 パラメータ・1 トークンあたり 6 FLOPs としてスループットを演算速度に換算した値であり、`MFU` はさらに RTX 4090 の bf16 ピーク性能である 165.2 TFLOP/s で割った**モデル FLOPs 利用率**（model FLOPs utilization）である Chowdhery et al. (2022)（演習問題 1 ではアテンションスコアの FLOPs も加える）。

```
flops = 6 * num_params  # Per token, without the attention scores
peak = {'tf32': 82.6e12, 'bf16': 165.2e12}  # FLOP/s of the RTX 4090
print(f'{"configuration":18}{"tokens/s":>10}{"step":>7}{"total":>7}'
      f'{"TFLOP/s":>9}{"MFU":>6}')
first = previous = next(iter(results.values()))
for name, speed in results.items():
    print(f'{name:18}{speed:10,.0f}{speed / previous:6.2f}x'
          f'{speed / first:6.2f}x{speed * flops / 1e12:9.1f}'
          f'{speed * flops / peak["bf16"]:6.0%}')
    previous = speed
print('-' * 57)
for name, speed in branches.items():  # Measured on top of the last row
    print(f'{name:18}{speed:10,.0f}{speed / previous:6.2f}x{"":7}'
          f'{speed * flops / 1e12:9.1f}{speed * flops / peak["bf16"]:6.0%}'
          '  (not kept)')
```

```
configuration       tokens/s   step  total  TFLOP/s   MFU
baseline             317,736  1.00x  1.00x     36.2   22%
+ compile            368,688  1.16x  1.16x     42.1   25%
+ bf16               643,350  1.74x  2.02x     73.4   44%
+ batch 512          653,141  1.02x  2.06x     74.5   45%
---------------------------------------------------------
+ checkpointing      556,390  0.85x            63.5   38%  (not kept)
```

単一 GPU の最適化を累積すると訓練スループットは 2 倍になり、RTX 4090 の bf16 ピーク性能の約 50% に達する。改善のほぼすべては行列積の高速化（bf16）とその前後のメモリトラフィックの削減（カーネル融合）によるものであり、ホストオーバーヘッドの除去とミニバッチの拡大による寄与はプロファイルの予測どおりそれぞれ数パーセントにとどまる。

なお、各高速化率は適用順序における位置（演習問題 7）とハードウェアのバランスに依存する。より高速な GPU（あるいはより小さなモデル）ではデバイス実行時間が短くなるため、ホストディスパッチの除去がはるかに大きな効果を持つようになる。

13.8.4.2 訓練の検証

どれほどスループットが高くても、最適化によって訓練が密かに壊れてしまっては意味がない。そこで正当性を検証するため、同一に初期化した 2 つのモデル（一方はベースラインのステッ

プ、もう一方は最終的なコンパイル済み bf16 ステップを使用）を、512 ウィンドウからなる同じ 200 個のミニバッチ（〜 13M トークン）で訓練する。その際、AdamW における後半の損失スパイクを防ぐため、50 ステップの線形学習率ウォームアップを用いる（11.2 節）。両モデルが 0.15 nats/byte 以内で最後の 20 ステップの平均損失および検証損失（`d2l.lm_perplexity`、9.3 節）において一致すること、さらに両者とも 2.45 nats/byte という 11.2 節のバイグラムのベースラインを上回ることを確認する。

```
learner = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
reference = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
reference.load_state_dict(learner.state_dict())  # One initialization
steps = {'baseline': make_step(reference, warmup=50),
         'final': make_step(learner, torch.compile(
             learner, dynamic=False, mode='reduce-overhead'), bf16=True,
             warmup=50)}
batches = windows(corpus, 512)
minibatches = [next(batches) for _ in range(200)]  # The same for both
losses = {name: torch.stack([step(X, Y).detach()
                             for X, Y in minibatches]).tolist()
          for name, step in steps.items()}
validation = {name: math.log(d2l.lm_perplexity(net, data.val_dataloader()))
              for name, net in (('baseline', reference), ('final', learner))}
```

ステップ 1、50、100、200 における訓練損失の推移を、最後の 20 ステップの平均（`late`）および検証データの対数パープレキシティ（`validation`）と並べて比較し、両方の実行がバイグラムのしきい値である 2.45 nats/byte を下回り、かつ 0.15 nats/byte 以内で一致することを検証する。

```
checkpoints = (1, 50, 100, 200)
late = {name: np.mean(curve[-20:]) for name, curve in losses.items()}
print(f'{"loss at step":14}' + ''.join(f'{i:>7}' for i in checkpoints)
      + f'{"mean of last 20":>17}{"validation":>12}')
for name, curve in losses.items():
    print(f'{name:14}' + ''.join(f'{curve[i - 1]:7.2f}' for i in checkpoints)
          + f'{late[name]:17.2f}{validation[name]:12.2f}')
    assert validation[name] < 2.45  # Better than the bigram model
for values in (late, validation):  # The final step trains like the baseline
    assert abs(values['baseline'] - values['final']) < 0.15
```

```
loss at step        1     50    100    200  mean of last 20  validation
baseline         5.66   2.52   2.03   1.49             1.52        1.47
final            5.66   2.51   2.06   1.50             1.53        1.47
```

両曲線とも $\log 256 \approx 5.55$ から始まり、全 200 ステップにわたって互いによく追随し、バイグラムのしきい値を大きく下回るほぼ同一の検証損失に到達する。ただし、バッチサイズ 512 での 200 ステップ（13M トークン）の最終損失は、コサイン減衰を用いたバッチサイズ 64 での 1,000 ステップ（8M トークン、11.2 節）よりも高い点に注意したい。生のトークンスループットは、目標損失への到達時間と同一ではない（演習問題 5）。

13.8.4.3 最終ステップの時間内訳

最後に、`account` によってベースラインと最終構成のプロファイルを 8,192 トークンあたりのミリ秒数に正規化し、総実経過時間を理想的なピーク演算時間、実際の行列積カーネル時間、および行列積以外のカーネル時間と比較する。

```
def account(kernels, tokens, speed):
    """Milliseconds per 8,192 tokens of a step of `tokens` tokens that runs
    at `speed` tokens/s: the step, its matrix products, and all its other
    kernels."""
    per_step = 8192 / tokens  # Steps per 8,192 tokens
    products = per_step * sum(  # Kernel names as in d2l.show_profile
        ms for name, ms in kernels.items()
        if re.search(d2l.PRODUCT_KERNELS, name, re.I))
    other = per_step * sum(kernels.values()) - products
    return 1e3 * 8192 / speed, products, other

print(f'{"ms per 8,192 tokens":24}{"step":>6}{"at peak":>9}'
      f'{"products":>10}{"other":>7}')
for name, tokens, fmt in (('baseline', 64 * 128, 'tf32'),
                          ('+ batch 512', 512 * 128, 'bf16')):
    wall, products, other = account(profiles[name], tokens, results[name])
    arithmetic = 1e3 * 8192 * flops / peak[fmt]  # At the peak of the format
    print(f'{name + " (" + fmt + ")":24}{wall:6.1f}{arithmetic:9.1f}'
          f'{products:10.1f}{other:7.1f}')
```

```
ms per 8,192 tokens       step  at peak  products  other
baseline (tf32)           25.8    11.3      15.6    8.8
+ batch 512 (bf16)        12.5     5.7       7.1    5.4
```

8,192 トークンあたりで見ると、最終ステップは行列積に約 7 ms（80% 超の効率、bf16 の理論値 5.7 ms 比）、その合間の帯域幅律速カーネルに 5 ms を費やしている。これは縮約演算が全 FLOPs の 99% 超を実行しながら時間では約 60% を占めるという、典型的なルーフライン上の内訳である (Ivanov et al. 2021)。7 ms 対 5 ms に分かれたステップは、演算の高速化（たとえば fp8）単独でも、行列積以外のメモリトラフィックの除去単独でも半分にはできないため、さらに $2\times$ の高速化を達成するにはその両方が必要になる。

13.8.5 まとめ

カーネルプロファイルを用いれば、訓練ステップのボトルネックを分類し、候補となる各最適化を適用する前にその改善幅の上限を見積もることができる。具体的には、ホストディスパッチの削減で節約できるのは高々デバイスのアイドル時間、カーネル融合で節約できるのは高々行列積以外のデバイス時間であり、テンソルコアの数値精度スループットを倍増させたときの高速化率は $1/(1-f/2)$ から 2 の間（行列積がステップの割合 f を占める場合）となる。1 つのボトルネックを取り除くと残りの処理が占める割合が変化するため、各最適化は数値的等価性を検証しながら累積的に計測しなければならない。単一の RTX 4090 上では、19M パラメータの GPT のステップ速度は 2 倍に向上して 45〜50% MFU に達した。その改善のほぼすべては bf16 の行列積とカーネル融合によるものであり、最終的なステップはピークに近い行列縮約と帯域幅律速カーネルにおおむね 60 対 40 の割合で分かれた。最後に、データ並列のコストモデル $t_{\text{step}}(1) + t_{\text{allreduce}}(k)$ は、短い fp32 ステップにおける GPU 2 基での減速（22 ms のステップ対 31 ms の PCIe オールリデュース）と、バッチ 512 の bf16 ステップにおける GPU 4 基での約 $3\times$ の高速化（90–100 ms のステップ対 48 ms のオールリデュース）の両方を的確に予測した。

13.8.6 演習問題

1. [code] **モデル FLOPs 利用率**。13.8.1 節の GPT は、$L = 6$ 個のブロック（幅 $d = 512$、コンテキスト長 $T = 128$）を持つ。SwiGLU の丸めを除けば、各ブロックは行列積に $12d^2$

個の重みを保持し、訓練には重み 1 個・1 トークンあたり約 6 FLOPs を要する。

1. アテンションスコアの積 $\mathbf{QK}^\top$ と値の集約は、重みを持たない行列積である。1 トークン・1 ブロックあたりの訓練 FLOPs を数えよ（逆伝播は順伝播の 2 倍とし、マスクされた要素も含めること）。重みの FLOPs に対するその比率が $T/(6d)$ となることを示し、$T = 128$ および $T = 1{,}024$ に対してこの比率を求めよ。
2. アテンションの FLOPs を含めて累積比較表の `TFLOP/s` 列と `MFU` 列を再計算し、あわせて各構成固有の数値形式のピーク性能（bf16 では 165 TFLOP/s、tf32 では 83 TFLOP/s）に対する割合も求めよ。
3. 13.8.2 節の構成またはカーネルのうち、理論上のモデル FLOPs とハードウェアが実際に実行する FLOPs が異なるものを 2 つ挙げ、それぞれの差の向きを述べよ。

2. [code] **ステップの固定コスト**。バッチサイズ B ウィンドウにおける bf16 のステップ時間を $t(B) = a + bB$ とモデル化する（a はステップあたりの固定コスト、b はウィンドウあたりのコストである）。
   1. $B = 32, 64, 128, 256, 512$ でステップ時間を計測し、最小二乗法によって a と b を当てはめ、$B = 64$ のステップ時間のうち a が占める割合を報告せよ。
   2. 本文では、AdamW の更新と fp32・bf16 間の重みキャストを ~ 0.75 ms/step と見積もった。これは a のどれだけの割合を説明するか。B に依存しない他の処理を特定し、$B = 32$ におけるプロファイルと照合して確認せよ。
   3. 線形当てはめを用いて $B = 1{,}024$ および $B = 2{,}048$ におけるスループットを予測し、それらが 24 GiB の GPU メモリに収まるか確認したうえで、収まるものについて実測せよ。
3. [code] **再計算のコスト**。13.8.2 節の活性化チェックポイントは、FLOP 数による見積もりと単体の順伝播時間による見積もりの両方を上回った。
   1. 単体の順伝播とチェックポイント化されたステップを `d2l.kernel_times` でプロファイリングせよ。チェックポイント化によって追加されたカーネルを単体の順伝播カーネルと突き合わせ、そのまま再出現するカーネルと逆伝播カーネルに融合されるカーネルはそれぞれどれか答えよ。
   2. 6 個のブロックのみの順伝播時間（埋め込みと出力層を除く）を計測し、スループットの下限を再計算せよ。
   3. 1 つおきのブロックのみをチェックポイント化する場合について、実行前に自身の計測値からスループットとメモリ使用量を予測し、その後に実測して比較せよ。
4. [code] **因果性ゲート**。欠陥のあるアテンションカーネルが因果マスクを密かに無視してしまうと仮定する。
   1. ベースラインと学習対象モデルの両方がこの欠陥カーネルを使用していた場合、訓練損失、検証ウィンドウ損失、および 2.45 nats/byte のバイグラムしきい値（13.8.4 節）がいずれもこの欠陥を検出できない理由を説明せよ。
   2. ミニバッチ内のすべての系列の最後の入力トークンに摂動を与え、それより前のすべての位置のロジットが数値誤差の範囲内で変化しないことを検証するテストを作成せよ。これを `learner` と、アテンションから因果マスクを省いたコピーの両方で実行せよ。
5. **スループット対目標到達時間**。9.8 節のモデルでは、バッチサイズ b で目標損失に到達するために必要な訓練例の数を $E(b) = E_{\min}(1 + b/b_{\text{noise}})$ と見積もる。ここで b_{noise} は勾配ノイズスケールである。
   1. バッチ 512 のバッチ 64 に対するスループット比を $\rho > 1$ とする。バッチ 512 がより短い実経過時間で目標損失に到達するための必要十分条件が

$$b_{\text{noise}} > \frac{512 - 64\rho}{\rho - 1}, \tag{13.16}$$

   であることを示し、$\rho = 1.2$、$\rho = 1.7$、および 13.8.4 節で実測された ρ に対してこのしきい値を求めよ。

2. 13.8.2 節の各変更、弱スケーリングのデータ並列、および Muon（9.6 節）への切り替えを、それぞれ $E(b)$ を変化させるか、トークンスループットを変化させるか、あるいはその両方を変化させるかによって分類せよ。

6. [code] **幅 256**。`num_hiddens=256` とすると、`d2l.GPT` は 11.2 節で訓練したアーキテクチャと一致する。
   1. $8{,}192 \times 256$ 行列と 256×256 行列の出力射影 $\mathbf{W}_o$ について、4 バイトの tf32 と 2 バイトの bf16 における演算密度を計算し、両者を RTX 4090 のルーフライン（Figure 13.1）上に配置して、この積に対する bf16 の高速化率を予測せよ。
   2. `num_hiddens=256` でベースラインと 13.8.2 節の最初の 2 つの最適化を実行し、`profile` と `account` でプロファイリングして、実測された行列積の高速化率をルーフラインによる予測値と比較せよ。
7. [code] **変更の順序**。累積比較表の各高速化率は、それより前に適用された最適化に依存する。
   1. ベースラインのプロファイルと 13.8.2.3 節の上限式を用いて、未コンパイルのベースラインに直接 bf16 を適用した場合の高速化率を予測せよ。
   2. その高速化率を実測し、自身の予測値および累積比較表の `+ bf16` 行と比較せよ。さらに bf16 の上にコンパイル（表の 2 行目）を適用し、カーネルプロファイルを用いて両方の比率がどのように変化したかを説明せよ。

13.9 関連文献

本章の各節では、それぞれの考え方について代表的な例を 1 つずつ扱ってきた。すなわち、1 つのメモリ帯域幅を持つ 1 つのルーフライン、1 基の GPU とそのリンク、フレームワークごとに 1 つのコンパイラ、1 枚のカード上でメモリを節約する 3 つの方法、1 つの集合通信アルゴリズム（collective algorithm）、モデルの状態を GPU 間で分割する 1 つの方法、そして 1 台のマシン上で高速化した 1 つの Transformer である。オープンアクセスの 2 冊の書籍が、本章全体の格好の伴侶となる。『How to Scale Your Model』（Austin et al., 2025）は TPU と GPU におけるルーフライン、シャーディング記法、および集合通信のコストを導出しており、『The Ultra-Scale Playbook』（Hugging Face, 2025）はメモリバジェット、データ並列、ZeRO、およびマルチノード並列性を実測値とともに概説している。13.1 節の 3 つのレジームに関する手短な入門としては、Horace He のエッセイ (He 2022) が参考になる。

13.9.1 計測とモデル化

13.9.1.1 より精緻な性能モデル

13.1 節のルーフライン (Williams et al. 2009) は、上下どちらの方向にも誤差を生じる。一部のカーネルがその帯域幅ルーフを上回って動作するのは、72 MB の L2 キャッシュや共有メモリに収まるテンソルが β より速く到達するためである。キャッシュ考慮型ルーフライン（cache-aware roofline）(Ilic et al. 2014) はメモリ階層の各レベルごとに 1 本の帯域幅ルーフを引き、Yang et al. (2020) は深層学習カーネルについてこれらのレベルごとのバイト数をハードウェアカウンタから読み取っている。逆に、同時に移動中のバイト数（bytes in flight）が少なすぎるためにルーフを下回るカーネルもある（リトルの法則、13.2.3 節）。Volkov (2016) は、1 つのスレッド内の独立した命令が追加のワープの代わりになり得ること、そして従来の GPU 性能モデルがリトルの法則を誤って適用した結果、スループットを最大 1.7 倍誤って予測していたことを示している。

13.9.1.2 デバイスタイマとシステムプロファイラ

`d2l.Benchmark`（13.1 節）はクロックを読み取る前にキューを空にするため、実際の訓練ループでは速度低下を招く。CUDA イベント（`torch.cuda.Event`）はキューを空にすることなく GPU ス

トリーム上でタイムスタンプを記録するが、その区間には GPU がホストを待機していた空白時間も含まれる。13.1.4 節のプロファイラは、1 つのプロセスについてビジー時間とアイドル時間を分離し、Perfetto 用のトレースを書き出す。NVIDIA の Nsight Systems は全プロセスにわたる OS スレッド、メモリコピー、NCCL 呼び出しを 1 つのタイムライン上に配置し、一方 Nsight Compute は単一のカーネルを両方のルーフに照らして計測する。

13.9.1.3 入力パイプライン

画像や音声で訓練する読者は、Python のオーバーヘッドよりも先にデータローダの供給不足（starvation）に直面することが多い。次のミニバッチを CPU がデコードしてデータ拡張するあいだ、GPU が待機してしまうからである。ホストの空きコア数が GPU の必要とする数より少ない場合（13.2.4.1 節）、デコード処理を GPU へ移すことができる（たとえば NVIDIA DALI を用いる）。Mohan et al. (2021) は複数の訓練ジョブにわたるデータストール（data stall）を分析し、訓練の特性を考慮したキャッシュ（training-aware cache）とハイパーパラメータスイープ間での前処理の共有によって、ストール時間の大部分を解消している。

13.9.1.4 利用率と目標損失までの時間

ケーススタディの締めくくりで行ったカーネルの内訳分析（13.8.4.3 節）は、Ivanov et al. (2021) が BERT に対して実施したものであり、行列積のあいだにある帯域幅律速のカーネルを融合することでエンコーダ層を $1.3\times$ 高速化した。ケーススタディの MFU 列は、大規模な訓練実行で報告される標準的な評価指標である (Chowdhery et al. 2022)。**ハードウェア FLOPs 利用率**（hardware FLOPs utilization、HFU）は再計算された FLOPs も数え上げるため、活性化チェックポイントのもとでは MFU を上回る (Korthikanti et al. 2022)。モデルが本来必要とする計算のみを評価対象とするため、システム間を公平に比較できるのは MFU だけである。

毎秒トークン数によるスループットは、たとえ収束により多くのステップを要する変更であっても、1 ステップを短縮するあらゆる変更を高く評価してしまう。そのため modded-nanoGPT スピードラン (Jordan and contributors 2024) では、8 基の GPU 上で GPT を一定の検証損失まで訓練するのにかかる実経過時間によって変更を順位付けしており、コンパイル済みカーネルや低精度形式を Muon オプティマイザ（9.6 節）と組み合わせている。

13.9.2 ハードウェアと数値形式

13.9.2.1 その他のアクセラレータ

ルーフライン解析は、Table 13.2 以外のアクセラレータにもそのまま適用できる。AMD の Instinct MI355X、Google の TPU7x、Intel の Gaudi 3、AWS の Trainium2 はいずれも、積層 HBM と行列演算ユニットを組み合わせている。これらの bf16 におけるリッジポイントは約 230 FLOP/byte（Trainium2）から 450 超（Gaudi 3）に及び、本章の RTX 4090 における 164 FLOP/byte の $1.4\times$ から $3\times$ に達するため、演算融合やビット幅の狭い形式がいっそう重要になる。

13.9.2.2 ユニファイドメモリ

重みがメモリに収まる限り、シングルストリームのデコーディングは、メモリ帯域幅を重みのバイト数で割った速度を超えて動作することはない（13.2.5 節）。Apple の M3 Ultra は、最大 512 GB のメモリを CPU と GPU のあいだで 819 GB/s の帯域幅で共有する。8 ビット表現の 70B モデル（約 70 GB）は 24 GB のカードには収まらないが、M3 Ultra 上では帯域幅の上限から毎秒約 12 トークンの生成が可能になる。このようなユニファイドメモリ搭載マシンは、計算ルーフの低さと引き換えに大容量を備えており、訓練よりもシングルストリームの推論に適している。

13.9.2.3 ブロックスケール形式の詳細

13.2.3 節のブロックスケールは、わずかなビット数のオーバーヘッドを伴う。OCP マイクロスケーリング形式 (Rouhani et al. 2023) では、32 個の値で共有される 8 ビットの 2 のべき乗スケールにより、値あたり 0.25 ビットが追加される。NVIDIA の NVFP4 は、16 個の値で共有する fp8 スケール（値あたり 0.5 ビット）に加えてテンソルごとに 1 つの fp32 スケールを持つことで、2 倍単位の範囲内にとどまるのではなく、各ブロックの最大値を 6（fp4 の範囲の上限）の近くに写像できる。いずれの方式でも、4 ビットの重みや活性化値に必要なのは多くても 4.5 ビットである。

13.9.3 コンパイラとカーネル

13.9.3.1 融合カーネルの記述

13.3 節のコンパイラは、要素ごとの演算の連鎖やリダクションを融合するが（PyTorch 2 については (Ansel et al. 2024)）、FlashAttention のようなタイル化を伴うアルゴリズム自体の書き換えまでは見つけ出せない。Triton (Tillet et al. 2019)（および JAX の Pallas）は、テンソルタイルをプログラミングの単位とする。すなわち、実装者がタイリングとオンチップバッファリングを指定し、コンパイラがスレッド、ベクトル命令、共有メモリを管理する。FlashAttention-2 (Dao 2024) はクエリタイルにわたって並列化し、FlashAttention-3 (Shah et al. 2024) は Hopper GPU 上でデータ移動をテンソルコアの積演算および指数計算とオーバーラップさせるとともに、fp8 サポートを追加している。

13.9.3.2 大規模環境でのコンパイル時間

小さなステップのコンパイル（13.3.3 節）は数秒で済むが、大規模なモデルのコンパイルにはプロセスおよび形状ごとに数分かかることがある。領域コンパイル（regional compilation）は、繰り返されるブロック（Transformer 層など）をすべてのコピーに対して 1 回だけコンパイルし（`torch.compiler.nested_compile_region`）、永続キャッシュはコンパイル済みアーティファクトをプロセス間で共有する（`torch.compiler.save_cache_artifacts`、`jax_compilation_cache_dir`）。オートチューニング（PyTorch の `mode='max-autotune'`、XLA では標準搭載）は、コンパイル時に複数の行列積タイリングをベンチマークし、初回のコンパイル時間と引き換えに各ステップを高速化する（13.3.3 節）。

13.9.3.3 訓練におけるカーネルリプレイ

記録された CUDA グラフは、形状とメモリアドレスを固定する（13.3.4 節）。`mode='reduce-overhead'` の背後にある CUDA グラフツリーは、共有メモリプール内で形状ごとに 1 回ずつ順伝播と逆伝播を記録する。ステップをまたいで保持されるテンソル（ログに記録した損失など）は、クローンしておかない限り次のリプレイ時に上書きされる。また、ランタイムがステップ境界を自動検出できない場合には、`torch.compiler.cudagraph_mark_step_begin` で明示的に印を付ける。

13.9.3.4 エクスポートと初期のグラフ設計

キャプチャされたグラフは Python の外部でも実行できる。`torch.export` はグラフブレイクを一切許さずに厳密な全体グラフをキャプチャし、JIT コンパイルされた JAX 関数はモバイル、Web、およびサービング用ランタイムへのデプロイに向けて StableHLO へと変換（lowering）される。Theano (Bergstra et al. 2010) や TensorFlow 1 (Abadi et al. 2016) といった初期のフレームワークでは、ユーザーが静的なシンボリックグラフを明示的に構築する必要があったが、トレーシングコンパイラは通常の即時実行（eager）コードから同じプログラム全体のグラフを復元する (Frostig et al. 2018)。

13.9.4 メモリ

13.9.4.1 より小さなオプティマイザ状態

13.4.1 節におけるパラメータあたり 16 バイトのうち、Adam の 2 つのモーメント推定値は 8 バイトを占める。これらを GPU 間でシャーディングする（13.7 節）代わりに、8-bit Adam (Dettmers et al. 2022) は両方のモーメントをブロックごとに量子化して正解率を損なうことなくパラメータあたり 2 バイトに抑える。一方、Adafactor (Shazeer and Stern 2018) は一次モーメントを省略し、二次モーメントを行和と列和へと分解（factorize）する。

13.9.4.2 ホストメモリとディスクへのオフロード

ホスト DRAM は GPU メモリよりも容量が 1 桁大きく、かつ安価である。ZeRO-Offload (Ren et al. 2021) は、fp32 のマスター重みと Adam のモーメントをホストメモリに保持してオプティマイザの更新を CPU 上で実行し、GPU 上には 16 ビットの重みと勾配のみを残す。さらに ZeRO-Infinity (Rajbhandari et al. 2021) は、オフロード先を NVMe ドライブまで拡張する。

13.9.4.3 再計算対象の選択

ブロックの入力のみを保存する手法（13.4.3 節、(Chen et al. 2016)）は、トレードオフ曲線上の 1 点にすぎない。二項チェックポイント法（binomial checkpointing）(Griewank and Walther 2008) は連鎖構造において保存する状態数を固定したもとで再計算量を最小化し、Checkmate (Jain et al. 2020) は一般の計算グラフを整数線形計画法によって解く。現代のコンパイラもまた、選択的再計算を自動的に適用するか（PyTorch における AOTAutograd の最小カットパーティショナ）、明示的なポリシー（JAX の `jax.checkpoint_policies`）を通じて適用する。

13.9.4.4 断片化と事前割り当て

ステップごとにテンソルの形状が変化すると、PyTorch のキャッシングアロケータが空きメモリを非連続な断片へと分割してしまうことがある（予約済みだが未割り当てを意味する「reserved but unallocated」）。`PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True` を設定すると、セグメントをインプレースで拡張できるようになる。JAX では、デフォルトのアロケータが起動時に GPU メモリの 75% を事前割り当てする（`XLA_PYTHON_CLIENT_PREALLOCATE=false` で事前割り当てを無効化でき、`XLA_PYTHON_CLIENT_ALLOCATOR=vmm` で CUDA の仮想メモリ管理を使用する）。

13.9.5 通信とスケール

13.9.5.1 リングを超える集合通信アルゴリズム

リング allreduce（13.5.3 節）は移動する総バイト数を最小化するが、$2(k-1)$ 回の逐次ホップを要する。ツリーアルゴリズムはホップ数を $\log_2 k$ のオーダーに抑え、NCCL は相補的な 2 つの二分木を組み合わせることで、すべてのデバイスが一方の木では葉となり、もう一方の木では内部ノードとなるようにしている。k が 2 のべき乗のとき、再帰的半減・倍増法（recursive halving and doubling）はリングと同じバイト通信量を $2\log_2 k$ ステップで達成する (Thakur et al. 2005)。

13.9.5.2 パラメータサーバーと非同期更新

集合通信ライブラリが標準となる以前、分散訓練ではパラメータサーバー上で勾配を集約し (Dean et al. 2012; Li et al. 2014)、非同期更新やローカル SGD を支えていた (Stich 2019)。古い勾配（stale gradient）は最適化のダイナミクスを変化させてしまうため、現在では密なモデルに対して同期型の allreduce や ZeRO（(S. Li et al. 2020; Rajbhandari et al. 2020)）が標準となっ

ている一方、巨大な疎の推薦テーブル（第 42 章（英語版））では依然としてパラメータサーバーが広く使われている。

13.9.5.3 並列化インターフェースと単一マシンを超える拡張

PyTorch は各並列化戦略をデバイスメッシュ上の組み合わせ可能な API（`DistributedDataParallel`、`fully_shard`、`torch.distributed.tensor.parallel`、`torch.distributed.pipelining`）として公開しているのに対し、JAX は GSPMD / Shardy (Xu et al. 2021) でコンパイルされる名前付きメッシュ上の配列配置によってそれらすべてを宣言的に表現する。マルチノードのテンソル並列・パイプライン並列・エキスパート並列 (Shoeybi et al. 2019; Narayanan et al. 2021; Lepikhin et al. 2021)（第 23 章（英語版））、fp8 訓練 (Micikevicius et al. 2022; DeepSeek-AI, Liu, et al. 2024)（21.3 節（英語版））、連続バッチングと PagedAttention (Yu et al. 2022; Kwon et al. 2023)（第 30 章（英語版））、および重みの量子化（第 28 章（英語版））については、「言語モデル」パートで詳しく扱う。

Chapter 14

強化学習

これまで本書で扱ってきたモデルはすべて、静止したデータセット上で訓練されていた。モデルが何を予測しても、次にどの例が現れるかは一切変わらなかったのである。ここで、数学の問題に取り組む言語モデルを考えてみよう。モデルは 1,000 トークンにわたる推論を書き連ねるが、答えが正しいかどうかを判定器が告げるのは最後だけである。どのトークンにも個別のラベルはなく、1 トークン出力するごとにモデルが次に書ける内容が変化する。動く台車の上で棒のバランスを取る制御器も同じ立場にある。台車を押すたびに次に補正すべき角度が変わり、得られる報酬（reward）は棒が倒れずに立っている時間だけである。Figure 14.1 は、意思決定主体である**エージェント**と、それが働きかける世界である**環境**（environment）の間のこのループを描いている。

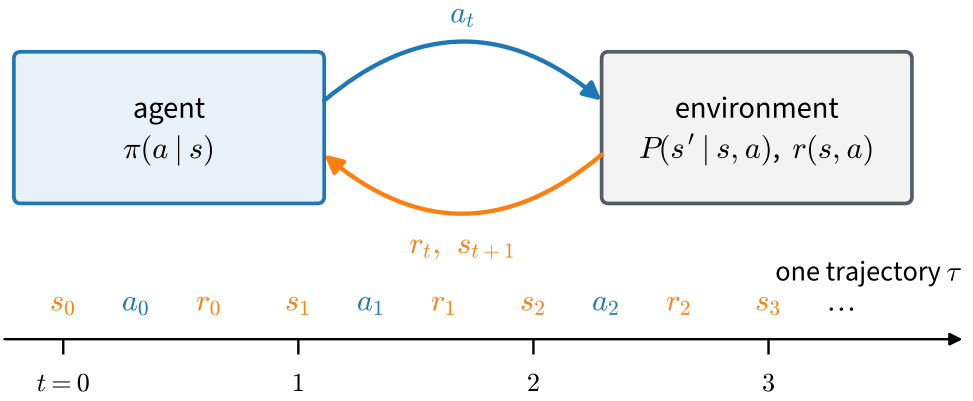


Figure 14.1: エージェントと環境の間のループ。ステップ t において、エージェントは方策 π に従って行動 a_t（青）を選択する。遷移確率 P と報酬 r に規定される環境は、報酬 r_t と次の状態 s_{t+1}（オレンジ）を返す。このやり取りを繰り返すことで、軌跡 $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ が生成される。

ここから 2 つの問いが生じる。遅れて届く報酬に対してどの意思決定が貢献したのか、そして何がよい意思決定かを誰にも示されないまま、学習者はどのようにしてそれを見つけ出せるのか。1 つ目の問いに対し、ルールが既知の場合の答えを 1950 年代に示したのが Richard Bellman である。各状態（state）にこれから得られる報酬によってスコアを付けることで、長いホライズンにわたる問題を 1 ステップの先読みの連鎖へと変換したのである (Bellman 1957b)。1980 年代

には Andrew Barto、Richard Sutton、Chris Watkins が試行錯誤によってそれらのスコアを学習し (Barto et al. 1983; Sutton 1988; Watkins 1989)、続いて Ronald Williams が意思決定の規則である**方策**（policy）を試行から得られた報酬によって直接改善する方法を示した (Williams 1992)。学習者は自らデータを収集するため、犯した失敗によって次に学ぶ対象そのものが変化する。モデルが自らの誤りから立て直す練習を一切しない場合、模倣だけでは不十分になる理由もここにある。

本章は 3 つの段階で進む。第 1 に、マルコフ決定過程（Markov decision process）を定式化し（14.1 節）、ルールが既知の場合の厳密な最適方策（optimal policy）を計算し（14.2 節）、サンプリングされた遷移から行動価値（action value）を学習し（14.3 節）、探索（exploration）のリグレットを定量化する（14.4 節）。第 2 に、長いホライズンにおいてエキスパートのデモンストレーションが見落とす問題を調べ（14.5 節）、既知の物理法則のもとで連続系の動的計画法（dynamic programming）がどのようになるかを、線形二次レギュレータ（linear-quadratic regulator、LQR）、反復 LQR（iterative LQR、iLQR）、およびリシーディングホライズン型のモデル予測制御（model predictive control、MPC）を通じて学ぶ（14.6 節）。最後に、スコア関数（score function）の勾配によって方策を直接最適化し（14.7 節）、ベースラインとグループ相対重みによって分散（variance）を抑え（14.8 節）、ルックアップテーブルをニューラルネットワークに置き換えたうえで、本章と次章の諸手法を体系的に整理する（14.9 節）。

すべての例は PyTorch と JAX で動作し、その大半はあらゆる推定値を厳密解と照らし合わせて検証できるほど小規模である。続く第 15 章ではブートストラップ型クリティック、経験再生（experience replay）、学習されたダイナミクスモデル、固定されたオフラインデータセットを扱い、「言語モデル」パートでは選好に基づく事後学習（post-training）をより深く掘り下げる。

参考資料と関連文献

以下に挙げる無料公開の資料では、理論的な解説、講義コース、実装、および実証的な指針が得られる。

教科書

- Sutton and Barto『Reinforcement Learning: An Introduction』(Sutton and Barto 2018) は、価値手法、方策手法、プランニング、関数近似を包括的に解説した入門書である。Sutton と Barto は、強化学習の基礎を築いた貢献により 2024 年のチューリング賞を受賞した。
- Szepesvári『Algorithms for Reinforcement Learning』(Szepesvári 2010) は、本章のアルゴリズムとその理論保証を 100 ページ未満でまとめている。
- Bertsekas『A Course in Reinforcement Learning』(Bertsekas 2025) は、最適制御と動的計画法の観点から強化学習を展開している。
- Agarwal、Jiang、Kakade、Sun『Reinforcement Learning: Theory and Algorithms』(Agarwal et al. 2019) は、本章で定性的に述べるサンプル計算量のレートを与えている。
- Lattimore、Szepesvári『Bandit Algorithms』(Lattimore and Szepesvári 2020) は、14.4 節の背後にある理論の全体像を扱っている。
- 14.6 節については、Kalman (1960) および Anderson and Moore (2007) が線形二次レギュレータ（LQR）を展開し、微分動的計画法 (Mayne 1966) と反復 LQR (Li and Todorov 2004) が後ろ向きのリカッチパスを非線形軌跡最適化へと拡張している。また、Rawlings et al. (2017) は制約付きのリシーディングホライズン型モデル予測制御を網羅し、Tedrake (2023) は劣駆動ロボティクスと軌跡最適化を扱っている。

講義コース

- Berkeley CS285: Deep Reinforcement Learning は、方策勾配、価値手法、モデルベース学習、オフライン強化学習を大学院レベルで扱っている。

- Stanford CS234: Reinforcement Learning は、基礎、探索、および理論保証を伴う方策勾配を扱い、対象範囲が本章に最も近い講義コースである。
- Stanford CS224R: Deep Reinforcement Learning は、模倣学習やオフライン手法を含め、実践面を扱っている。
- CMU 10-703: Deep Reinforcement Learning and Control は、アルゴリズムを制御および物理的な身体性と結びつけて扱っている。
- MIT 6.7920: Reinforcement Learning: Foundations and Methods は、14.2 節で導入する動的計画法の理論を展開している。

注釈付きの実装

- CleanRL (Huang, Rousslan F. J. Dossa, et al. 2022) は、14.9 節以降で扱う主要な深層強化学習アルゴリズムについて、ベンチマーク検証済みの単一ファイル実装を提供している。
- Stable-Baselines3 (Raffin et al. 2021) は同じアルゴリズム群のライブラリ実装を提供しており、付属の Zoo には調整済みのハイパーパラメータが用意されている。
- Gymnasium (Towers et al. 2024) は、これらの章全体で使用する環境インターフェースのドキュメントを提供している。

実証的な実践

- Spinning Up in Deep RL (Achiam 2018) は、スコア関数から SAC までの簡潔なカリキュラムであり、動作するコードと並べて読むことを想定して書かれている。
- The 37 Implementation Details of Proximal Policy Optimization (Huang, Rousslan Fernand Julien Dossa, et al. 2022) は、アルゴリズムの説明では省略されがちな実装上の選択が PPO の性能と再現性にどう影響するかを記録している。

14.1 マルコフ決定過程

凍った湖を渡るロボット、ボードゲームを対局するプログラム、あるいは 1 トークンずつ回答を書き進める言語モデルを想像してみよう。どの選択も次に起きることを変える。ロボットは別のマスへ滑り込み、ゲームは別の局面に至り、回答は別の展開をたどる。報酬は最後にしか得られないかもしれず、同じコマンドでも複数の結果が生じうる。うまく行動する方法を問う前に、まずこうした問題を書き下す枠組みが必要になる。すなわち、意思決定者が各ステップで何を知り、何ができ、その結果何が起き、何を達成しようとしているかである。

Bellman は 1957 年のマルコフ的決定過程に関する論文で、この記述に定式を与えた (Bellman 1957a)。その着想は、過去のうち将来に関わるすべてを**状態**（state）に集約し、行動の結果がその行動をとった状態のみに依存するようにすることであった。このただ 1 つの仮定によって、問題は扱いやすくなる。将来が現在の状態のみに依存するならば、状態ごとに 1 つの数値を与えるだけで、そこにいることの良さを表せる。こうして得られたモデルである**マルコフ決定過程**（Markov decision process、MDP）は、オペレーションズリサーチの標準的な道具となった (Puterman 1994)。動的計画法（dynamic programming）から報酬を用いた言語モデルの訓練に至るまで、本章と次章で扱うあらゆる手法はこのモデルの上で動作する。

本節では、すべての量を手計算で確かめられるほど小さい、16 マスの凍った湖を題材とする。それでもこの湖には、時間にわたる行動を難しくする性質が備わっている。滑りやすい氷の上では、エージェントが意図した方向とは別の場所へ滑ってしまうことが少なくないのである。まずシミュレータから湖の遷移表を読み出して各部に名前を付け、続いてエージェントが最大化する収益と、それを有界に保つ割引を定義する。さらに、状態が何を含まなければならないかを問い、一見有用に見えながら同じ場所を周回するだけでエージェントに報酬を与えてしまうボーナスを検証する。本節の締めくくりには、さらに小さな 2 つ目の例として、算数の問題に回答し判定器によってスコア付けされる言語モデルを扱う。

本章では Table 14.1 の記号を用いる。本節で使う記号は以下で定義し、残りの記号は後の各節において初出箇所でそれぞれ定義する。

Table 14.1: 強化学習の章で用いる記号。V^π 以降の行は後の各節に属し、λ 以降の行は第 15 章に属する。探索率 ϵ とステップごとの不一致率 ε は異なる記号であり、応答 y と価値ターゲット y_t も同様である。0 以外の添字または引数を持つ μ は平均（アームの平均、ガウス分布の平均など）を表す。

記号	意味	初出箇所
$\mathcal{S}, \mathcal{A}$	状態の集合と行動の集合	14.1 節
s_t, a_t, r_t	ステップ t における状態、行動、報酬	14.1 節
$P(s' \mid s, a)$	遷移カーネル：到達先が s' となる確率（状態 s、行動 a のもと）	14.1 節
$r(s, a)$	行動 a の状態 s における期待報酬	14.1 節
μ_0	初期状態分布（言語モデルではプロンプトの分布）	14.1 節
τ	軌跡 $(s_0, a_0, r_0, s_1, \ldots)$	14.1 節
γ	割引率	14.1 節
$R(\tau)$	軌跡の収益 $\sum_t \gamma^t r_t$	14.1 節
T	エピソードのステップ数、または時間制限	14.1 節
$\pi(a \mid s), \pi(s)$	方策：行動 a を状態 s でとる確率、および決定論的方策が返す行動	14.1 節, 14.2 節
$\ell(s)$	マス s からゴールまでのマンハッタン距離	14.1 節
$\Phi(s)$	シェイピングボーナスのポテンシャル	14.1 節
$x, y, r(x, y)$	プロンプト、応答、および応答に対する検証器のスコア	14.1 節
y_t	価値更新のターゲット：ステップ t からの収益の推定値	14.1 節, 14.9 節
V^π, Q^π	π のもとでの状態からの期待収益、状態行動対からの期待収益	14.2 節
V^*, Q^*	最適方策のもとでの同様の期待収益	14.2 節
$\mathcal{T}$	ベルマン作用素：価値のベクトルに適用する 1 ステップ分の先読み	14.2 節
δ, δ_t	1 回の遷移、およびステップ t の遷移における時間差分誤差	14.3 節
α	時間差分更新のステップサイズ	14.3 節
ϵ	ϵ 貪欲な行動選択における探索率	14.3 節
π_e	行動方策（手法が学習する間に行動を選択する方策）	14.3 節
κ	不確実性ボーナスの重み	14.4 節
K	プロンプトあたりにサンプリングされる応答数	14.4 節
π_θ, θ	パラメータ θ を持つ方策	14.1 節（検証器タスクの選好テーブル）、14.5 節
d_t^π, d_γ^π	s_t の π のもとでの分布、および t にわたるその割引平均	14.5 節, 14.7 節
ε	クローン方策がエキスパートと一致しないステップごとの確率	14.5 節

記号	意味	初出箇所
$\mathbf{x}_t$, $\mathbf{u}_t$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{Q}$, $\mathbf{R}$	連続状態と連続制御、線形ダイナミクス行列と二次コスト行列	14.6 節
$\mathbf{P}_t$, $\mathbf{K}_t$	リッカチの残余コスト行列と線形状態フィードバックゲイン行列	14.6 節
$J(\theta)$	π_θ の期待収益	14.7 節
η	方策更新のステップサイズ	14.7 節
$\hat{G}_t$	残りの累積報酬：ステップ t 以降の報酬の割引和	14.8 節
b	収益から差し引くベースライン	14.8 節
η_V	価値ベースライン更新のステップサイズ	14.8 節
A^π	アドバンテージ $Q^\pi - V^\pi$	14.8 節
λ	λ 収益の重み	15.1 節
w, w^-	価値ネットワークのパラメータ、およびその固定コピーまたは Polyak 平均コピーのパラメータ	15.1 節, 15.4 節
n	n ステップターゲットまたは計画の深さ	15.1 節, 15.6 節
ρ_t	重要度比 $\pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$（ステップ t）	15.2 節
ϵ_{clip}	比のクリッピング帯の半幅	15.2 節
Δ_{KL}	1 回の更新あたりの KL バジェット	15.2 節
π_{ref}, β	参照方策、および KL ペナルティの重み・ソフトアクタークリティックの温度	15.3 節, 15.5 節
$\hat{r}$	当てはめた報酬	15.3 節, 15.6 節
ν	ターゲットネットワークの Polyak 更新率	15.5 節
$\hat{f}_\phi$	学習された遷移モデル	15.6 節

```
import gymnasium as gym
```

14.1.1 モデル

Figure 14.2 に湖を示す。これは Gymnasium スイート (Towers et al. 2024) に含まれる Frozen-Lake であり、本章の表形式を扱う各節で共通して用いる。エージェントは左上のマスから出発し、報酬 1 が得られる右下のゴールへの到達を目指す。4 つのマスは穴になっており、穴またはゴールに入るとその試行は終了する。**滑りやすい**（slippery）氷の上では、コマンドを出すと確率 $1/3$ で意図した方向へ進み、直交する 2 方向へもそれぞれ確率 $1/3$ で進む。グリッドの端から外へ出ようとする移動では、エージェントはその場にとどまる。

ゴールから 3 手の距離にあるマス 9 にいるエージェントを考えよう。4 つのコマンドのうち 3 つは、いずれも 3 回に 1 回の確率でマス 5 の穴に滑り落ちてしまう。**下**だけは穴に落ちないが、**下**も 3 回に 2 回は横へ滑ってマス 8 または 10 に移る。**下**の安全性には、横に滑って手数を失うだけの価値があるだろうか。コマンド同士を比較するには、各コマンドがどこへ至りうるか、そこからエージェントに何ができるか、そして最後に何を受け取るかを知る必要がある。

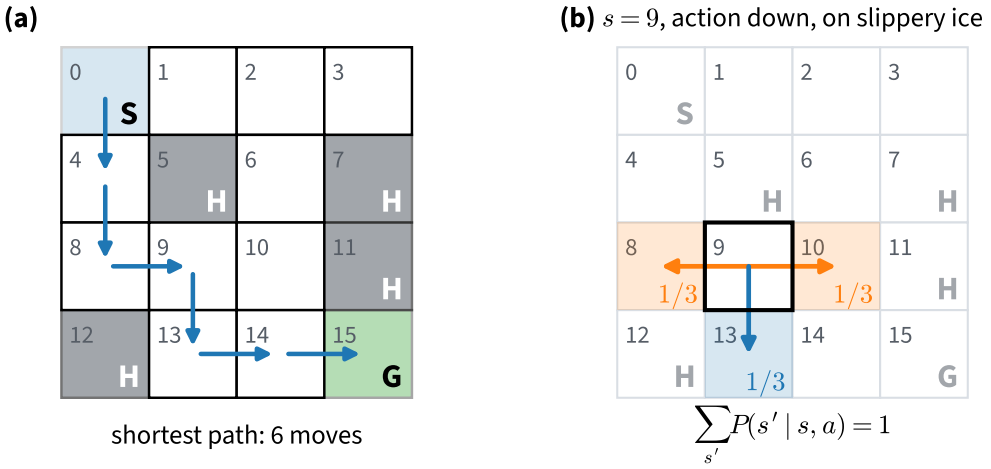


Figure 14.2: 湖。(a) S はスタート、G はゴールを表し、H と記された灰色のマスは穴である。各マスには状態のインデックスが付されている。滑らない氷の上では、最短経路は 6 回の移動でゴールに至る。(b) 滑りやすい氷の上での 1 回のコマンド：状態 $s = 9$ から行動**下**をとると、マス 13、8、10 のいずれかにそれぞれ確率 $1/3$ で到達する。意図した移動は等確率な 3 通りの結果の 1 つにすぎず、確率の和は 1 になる。

14.1.1.1 状態・行動・遷移カーネル

湖における**状態**はエージェントがいるマスである。状態の集合 $\mathcal{S} = \{0, 1, \ldots, 15\}$ は、行ごとに各マスへ番号を振る。**行動**（action）は**左**、**下**、**右**、**上**の 4 つのコマンドであり、0 から 3 として符号化され、集合 $\mathcal{A}$ をなす。滑りやすい氷の上では 1 つの行動が次の状態に関する分布を定め、Gymnasium はこの分布を直接読み出せる表として保持している。

```
env = gym.make('FrozenLake-v1', is_slippery=True)
LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
ARROWS = '<v>^'                          # one glyph per action, for printing
env.unwrapped.P[9][DOWN]
```

```
[(0.3333333333333333, 8, 0, False),
 (0.3333333333333333, 13, 0, False),
 (0.3333333333333333, 10, 0, False)]
```

出力には、（確率、次の状態、報酬、終端フラグ）という形式の結果が 3 つ並んでいる。マス 9 から**下**のコマンドを出すと、Figure 14.2 のとおり、エージェントはそれぞれ確率 $1/3$ で 8、13、10 のいずれかに到達する。記号で表せば、この表は**遷移カーネル**（transition kernel）$P(s' \mid s, a)$、すなわち状態 s' に状態 s から行動 a のもとで到達する確率である。各対 (s, a) に対して数値 $P(\cdot \mid s, a)$ は確率分布をなすため、$\sum_{s' \in \mathcal{S}} P(s' \mid s, a) = 1$ が成り立つ。

4 つ目の構成要素は**報酬**（reward）である。シミュレータはゴールに入るステップに対して 1 を与え、それ以外のすべてのステップには 0 を与えるため、1 ステップの報酬はそのステップの到達先に依存する。ここでは到達先のマスにわたって平均をとった量、すなわち**期待報酬**（expected reward）$r(s, a)$（行動 a の状態 s における平均報酬）を扱う。この平均化によって期待収益が一切変わらないことは、演習問題 3 で示す。以上 4 つの対象を合わせたものが MDP である。

$$\text{MDP}: \quad (\mathcal{S}, \mathcal{A}, P, r). \tag{14.1}$$

最初の 3 つは環境（environment）を記述する。一方、報酬はタスクを記述するものであり、タスクを設定する者がこれを選ぶ。この違いが、後の報酬設計の節の要となる。文献によっては、以下で定義する割引率をこの組に加えたり、報酬を確率変数としたり次の状態に依存させたりすることもある。その場合でも本節の $r(s,a)$ が期待報酬の役割を担うため、この簡潔な形式によって本章の議論が一般性を失うことはない。

14.1.2 収益・割引・ホライズン

エージェントは初期状態 s_0 で出発する。これは**初期状態分布**（start-state distribution）μ_0 から抽出され、湖の場合にはすべての確率質量をマス 0 に置く。各ステップ t において、エージェントは状態 s_t にあり、行動 a_t をとり、報酬 r_t を受け取って、次の状態 $s_{t+1} \sim P(\cdot \mid s_t, a_t)$ へと遷移する。これらの量の系列を**軌跡**（trajectory）と呼び、次のように書く。

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots).$$

軌跡に現れる報酬 r_t はシミュレータから実際に与えられた実現値であり、期待報酬 $r(s_t, a_t)$ は s_t と a_t を与えたもとでのその平均である。

行動を選ぶ規則を**方策**（policy）と呼び、$\pi(a \mid s)$ で行動 a を状態 s において選ぶ確率を表す。最も単純な方策は、どの状態でも 4 つのコマンドをそれぞれ確率 $1/4$ で選ぶものである。これをマス 0 から 1 回実行し、各ステップを表示してみよう。

```
rng = np.random.default_rng(8)
s, _ = env.reset(seed=8)
terminated = truncated = False
total, t = 0.0, 0
while not (terminated or truncated):
    a = int(rng.integers(4))
    s_next, r, terminated, truncated, _ = env.step(a)
    print(f"t={t:>2}  s={s:>2}  a={ARROWS[a]}  s'={s_next:>2}  r={r:.0f}")
    total, s, t = total + r, s_next, t + 1
print(f'terminated={terminated}, truncated={truncated}, total reward={total:.0f}')
```

```
t= 0  s= 0  a=>  s'= 0  r=0
t= 1  s= 0  a=v  s'= 0  r=0
t= 2  s= 0  a=<  s'= 4  r=0
t= 3  s= 4  a=^  s'= 4  r=0
t= 4  s= 4  a=<  s'= 4  r=0
t= 5  s= 4  a=v  s'= 8  r=0
t= 6  s= 8  a=>  s'= 9  r=0
t= 7  s= 9  a=^  s'=10  r=0
t= 8  s=10  a=>  s'=11  r=0
terminated=True, truncated=False, total reward=0
```

この実行記録には、遷移カーネルの働きがそのまま現れている。$t=0$ ではエージェントは**右**のコマンドを出すが、氷の上で上側の壁に向かって滑ったためマス 0 にとどまる。$t=2$ では**左**のコマンドを出しながら、**下**へ滑ってマス 4 に移る。そして $t=8$ ではマス 10 から**右**のコマンドを出してマス 11 の穴に入り、`terminated=True` となって試行が終了する。一度もゴールに入らな

かったため、報酬はすべて 0 である。ゴールに到達した軌跡を評価するには、遅れて得られる報酬をどれだけ重く評価するかをまず決めなければならない。

14.1.2.1 割引と実効ホライズン

報酬を単純に足し合わせないのはなぜだろうか。第 1 に、軌跡は無限に長くなりうる。常に上のコマンドを出す方策は、湖の最上段の行から決して出ない。上は壁にぶつかり、横へ滑る 2 通りも同じ行に沿った移動になるからである。毎ステップ正の報酬があるなら、そのような軌跡に沿った和は際限なく増大してしまう。また一般に、いま得られる報酬は後になって得られる同じ報酬よりも価値が高い。この 2 つの理由から**割引率**（discount factor）$0 \le \gamma < 1$ を導入し、ステップ t の報酬に重み γ^t を掛ける。軌跡の**収益**（return）は、次の割引和で定義される。

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{14.2}$$

$\gamma = 0.95$ のとき、1 ステップ先の報酬の重みは 0.95 であり、20 ステップ先の報酬の重みは約 0.36 となる。目的は、カーネルからのサンプリング、初期状態、およびエージェント自身の選択にわたる平均の意味で、収益ができるだけ大きくなるように行動することである。この**期待収益**（expected return）を最大化するのは、独立な試行を数多く繰り返したときに平均して得られる量がこれにほかならないからである。$\gamma < 1$ の有限 MDP では、この問題に明確な解が存在する (Puterman 1994)。あらゆる初期状態から同時に最良となる単一の方策が存在し、しかもそれは決定論的かつ時刻に依存しないもの、すなわちマスごとに 1 つのコマンドを対応させるものとして選べる。したがって最良の方策はエージェントの出発点に依存せず、初期状態分布が 1 つのマスに集中していても何も損なわれない。

割引はまた、エージェントがどれだけ先まで見通すかも制御する。すべての報酬が $|r_t| \le r_{\max}$ を満たすならば、割引によって収益は有限に保たれ、ステップ k 以降の部分も次のように抑えられる。

$$|R(\tau)| \le \frac{r_{\max}}{1-\gamma},$$
$$\Big|\sum_{t \ge k} \gamma^t r_t\Big| \le \frac{\gamma^k\, r_{\max}}{1-\gamma}.$$

1 つ目の上界により目的関数は有限になる。2 つ目の上界は、ステップ k 以降の報酬が収益を動かしうる幅が、最大可能値のせいぜい γ^k 倍にとどまることを表している。このことから、割引が届く範囲をステップ数で測る尺度として**実効ホライズン**（effective horizon）$1/(1-\gamma)$ が導かれる。γ が 1 に近いとき $\gamma^{1/(1-\gamma)} \approx e^{-1}$ であるため、重みは 1 ホライズンあたり約 e 分の 1 に減衰する。一定の報酬が続く列に対しては、割引重み全体の 95% が $\log(0.05)/\log(\gamma)$ ステップ、すなわち約 3 ホライズン以内に収まる。次の表は、4 通りの割引率についてこれら両方のステップ数と、滑らない氷での 6 手の最短経路の最後でゴール報酬に掛かる重み γ^5 をまとめたものである。

```
print(f'{"gamma":>6} {"horizon 1/(1-gamma)":>20} {"t: gamma^t < 0.05":>18} '
      f'{"gamma^5":>8}')
for g in [0.5, 0.9, 0.95, 0.99]:
    t95 = int(np.ceil(np.log(0.05) / np.log(g)))
    print(f'{g:>6} {1 / (1 - g):>20.0f} {t95:>18} {g**5:>8.3f}')
```

gamma	horizon 1/(1-gamma)	t: gamma^t < 0.05	gamma^5
0.5	2	5	0.031
0.9	10	29	0.590
0.95	20	59	0.774
0.99	100	299	0.951

本章全体で用いる割引率 $\gamma = 0.95$ では、ホライズンは 20 ステップ、重みの 95% は 59 ステップ以内に収まり、6 手で到達するゴールの価値は $\gamma^5 \approx 0.774$ となる。一方、$\gamma = 0.5$ では同じゴールの価値は約 0.03 にすぎない。Figure 14.3 にこれらの量を示す。割引率はタスクの一部であり、設計者が選ぶものである。γ を小さくするとエージェントは近視眼的になるが、不正確なモデルを用いて計画を立てる場面ではかえって信頼性が高まる場合があることを演習問題 8 で確認する。

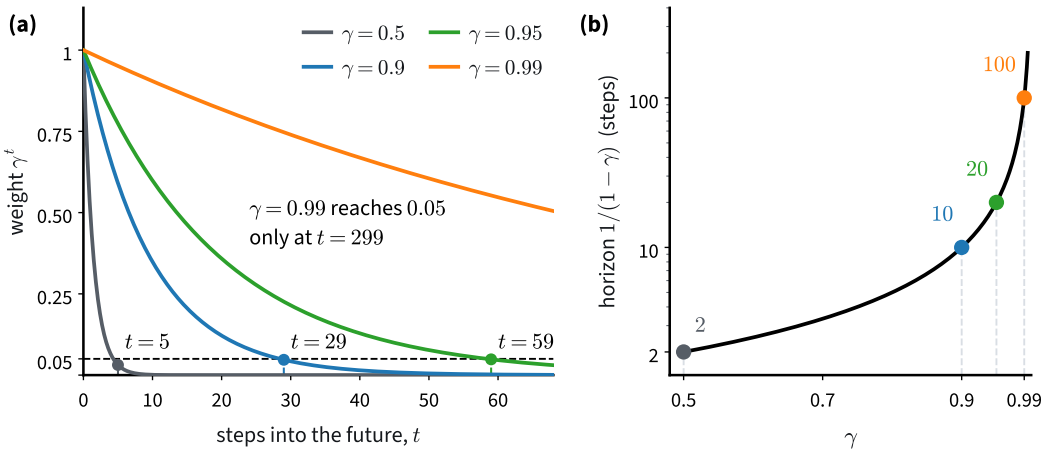


Figure 14.3: 割引は実効ホライズンを定める。(a) 重み γ^t（t ステップ先の報酬に対する重み）が 0.05 を下回るのは、$t = 5$ ($\gamma = 0.5$)、$t = 29$ ($\gamma = 0.9$)、$t = 59$ ($\gamma = 0.95$) であり、$t = 299$ ($\gamma = 0.99$) に至ってようやく下回る。(b) 対数軸で見たホライズン $1/(1-\gamma)$：$\gamma = 0.5$ で 2 ステップ、$\gamma = 0.95$ で 20 ステップ、$\gamma = 0.99$ で 100 ステップとなる。

14.1.2.2 配列としてのモデル

動的計画法はカーネル・報酬・割引率を何度も参照し、後の表形式を扱う各節でも学習器の結果をその厳密計算と照らし合わせて検証する。Gymnasium が公開しているタプルの辞書はこの用途には扱いにくい。次の状態にわたる平均を 1 回の行列積で計算できるよう、P を (s, a, s') で添字付けされた 1 つの配列として、r を (s, a) で添字付けされた 1 つの配列として持ちたい。`TabularMDP` はこれら 3 つの対象を保持するクラスである。その `from_gym` メソッドはシミュレータの表から配列を構築し、同じ次の状態に至る結果の確率を足し合わせるとともに、結果にわたって報酬を平均することで期待報酬 $r(s, a)$ を求める。本章の表形式の各節で使用するため、このクラスを `d2l` ライブラリに保存しておく。

```
class TabularMDP:
    """A finite MDP as dense arrays: P[s, a, s'] and r[s, a]."""
    def __init__(self, P, r, gamma):
        self.P, self.r, self.gamma = P, r, gamma
        self.num_states, self.num_actions = r.shape

    @classmethod
```

```
    def from_gym(cls, env, gamma):
        """Read the transition table Gymnasium exposes as env.unwrapped.P."""
        n_s, n_a = env.observation_space.n, env.action_space.n
        P, r = np.zeros((n_s, n_a, n_s)), np.zeros((n_s, n_a))
        for s, actions in env.unwrapped.P.items():
            for a, outcomes in actions.items():
                for p, s_next, reward, _ in outcomes:
                    P[s, a, s_next] += p      # several outcomes may share s'
                    r[s, a] += p * reward     # r(s,a) is the expected reward
        return cls(P, r, gamma)
```

$\gamma = 0.95$ として湖のモデルを構築し、先ほどシミュレータから読み出したコマンドに対応する行を表示してみよう。

```
gamma = 0.95
mdp = TabularMDP.from_gym(env, gamma)
print(mdp.P[9, DOWN])
```

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.33333333 0.         0.33333333 0.
 0.         0.33333333 0.         0.        ]
```

タプルのリストは 16 個の数値からなる密な行へと変わり、列 8、10、13 が $1/3$ で、それ以外は 0 になっている。配列ごとに 1 つずつ、計 2 つの確認で構築結果を検証しよう。P のすべての行の和が 1 であること、そしてゴールに入れる箇所でのみ報酬が非ゼロであることである。

```
print(f'largest |row sum - 1|: {np.abs(mdp.P.sum(axis=-1) - 1).max():.1e}')
for s, a in np.argwhere(mdp.r > 0):
    print(f'r(s={s}, a={ARROWS[a]}) = {mdp.r[s, a]:.3f}')
```

```
largest |row sum - 1|: 0.0e+00
r(s=14, a=v) = 0.333
r(s=14, a=>) = 0.333
r(s=14, a=^) = 0.333
```

P のどの行も和は 1 になっている。報酬配列は、ゴールに隣接する唯一の氷のマスであるマス 14 を除いてすべて 0 である。マス 14 では**下**、**右**、**上**がいずれも確率 $1/3$ でゴールに到達するため期待報酬 $1/3$ を持つ一方、**左**の 3 通りの結果はいずれもゴールにはならない。この湖では、ゴールに入るそのステップでのみ報酬が与えられるのである。

14.1.2.3 エピソード・終端・打ち切り

過程を終了させる状態を**終端**（terminal）状態と呼ぶ。湖では穴とゴールが終端状態である。出発から終端状態に至るまでの一連の実行を**エピソード**と呼ぶ。シミュレータはそこでエピソードを停止するため、リセットが必要になる。Gymnasium の表、ひいては `TabularMDP` では、P のすべての行が確率分布になるよう、各終端状態をどの行動でも報酬 0 で自分自身に戻る吸収状態（absorbing state）として格納している。吸収状態ではそれ以上何も得られないため、どちらの記述でも収益は同じになる。

すべてのエピソードが有限ステップで終わる場合、収益を有限に保つための割引は不要であり、$\gamma = 1$ が許容される。このとき収益は、T ステップからなるエピソードの報酬の単純和になる。湖では、常に**上**を選ぶ方策が最上段を永遠に回り続けるため、すべてのエピソードが終わるとは

限らないが、それでも報酬が非負で 1 本の軌跡が高々 1 しか得ないため $\gamma = 1$ としても問題はない。しかし一般にはそうはいかない。1 ステップあたり $+0.05$ の報酬があれば、同じ常に上を選ぶ方策の非割引収益は無限大に発散してしまう。したがって、建物の暖房制御のように終わりのないタスクでは $\gamma < 1$ を用いる。そのようなタスクに対するもう 1 つの一般的な選択肢は長期的なステップあたり平均報酬の最大化であるが、本章では扱わない。

Gymnasium はエピソードが停止する 2 つの場合を区別している。`terminated=True` は、過程が終端状態に入ったことを意味する。それ以上は何も起こりえず、継続することの価値は 0 である。一方 `truncated=True` は、本来なら過程が続くはずの状態にエージェントがいるまま、通常は時間制限によって記録が打ち切り（truncation）になったことを意味する。`gym.make` は FrozenLake にこの制限を設けており、100 ステップ経過すると、たとえエージェントがまだゴールへ到達できる氷の上に立っていても `truncated=True` でエピソードが終了する。Figure 14.4 に両者の違いを示す。したがって、打ち切られたエピソードにおける報酬の和は、打ち切り後の分がすべて欠けているため本来の収益ではない。最後の状態から継続する価値を推定する手法は、この 2 つを別々に扱う必要がある。終端後の価値は 0 であるが、打ち切り後の価値は最後の状態が持つ本来の**価値**（value）$V(s)$、すなわち s 以降の期待収益であり、その厳密な定義は 14.2 節で与える。本章で継続の価値を推定する手法はすべて、`terminated` のときに限ってそれを 0 とおく。

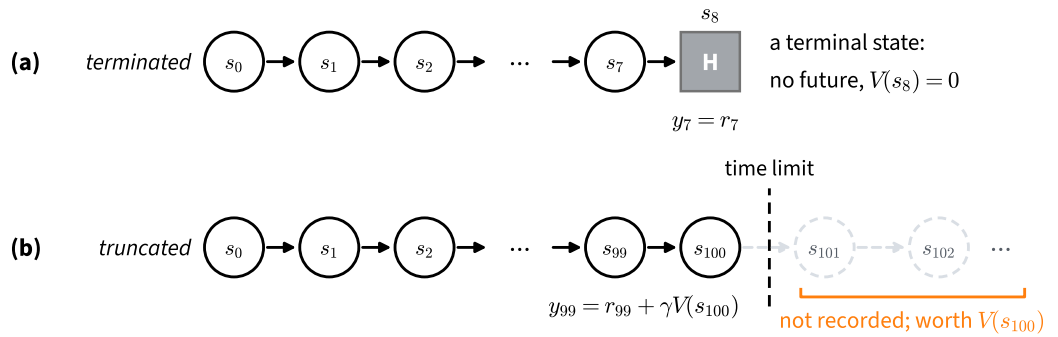


Figure 14.4: エピソードが停止する 2 つの場合。(a) 終端（Terminated）：エピソードが終端状態である穴 s_8 に入る。その先には何も続かないため、そこから継続する価値 $V(s_8)$ は 0 であり、最後のステップに対するターゲット y_7（そのステップから数えた収益）は最後の報酬 r_7 のみとなる。(b) 打ち切り（Truncated）：本来なら過程が続く氷の上にエージェントが立っている途中で、100 ステップの時間制限によって記録が打ち切られる。記録されなかった将来には最後に記録された状態から継続する価値 $V(s_{100})$ があり、これは 0 ではないため、最後のステップからの収益の推定値 $y_{99} = r_{99} + \gamma V(s_{100})$ にはこれを含めなければならない。

14.1.3 状態の選び方

遷移カーネル $P(s' \mid s, a)$ が条件付けるのは現在の状態と行動だけであり、それ以外のものは一切含まれない。これが**マルコフ性**（Markov property）である。すなわち、現在の状態がわかっていれば、それ以前の履歴は次の状態について新たな情報を何も付け加えない。この性質こそがモデルを小さく保つ鍵である。将来が現在の状態のみに依存するため、どの経路をたどってそこに到達したかにかかわらず、ある状態にいることの良さを状態ごとの 1 つの数値に要約できる。湖においては、この性質は構成上そのまま成り立つ。シミュレータは先ほど読み出した行 `P[s][a]` から次のマスをサンプリングし、滑りを毎回新たに抽選するからである。しかし一般のシステムでこの性質が成り立つかどうかは、何を状態と呼ぶかにかかっている。

14.1.3.1 状態拡張

記録された状態 s_t が車両の位置であり、行動 a_t がその加速度であるとする。次の位置は現在の速度にも依存するが、現在の位置だけでは速度はわからない。連続する 2 つの位置があれば速度を推定できるため、次の位置は過去 2 時点の位置に依存することになり、次式で表される。

$$s_{t+1} = f(s_t, a_t, s_{t-1}), \tag{14.3}$$

したがって、位置単独ではマルコフにならない。ここで状態を（位置、速度）の対として定義すれば問題は解消する。ニュートン力学により、現在の対と加えた加速度から次の位置と速度が一意に決まるからである。つまりマルコフ性とは、物理系そのものだけでなく、状態をどう表現するかという表現の性質なのである。

この構成法を**状態拡張**（state augmentation）と呼ぶ。将来を予測するために必要な情報が含まれるまで状態を拡大するのである。もっとも、拡大された状態は巨大になりうるし、必要な情報が未知であったり、保持するには大きすぎたり、そもそも観測不可能であったりすることもある。タスクの一部として時間制限が課されている場合は単純な例である。100 ステップしか許されていないとき、あるマスでの最良のコマンドは残りステップ数に依存しうるが、その残り回数を状態に加えればマルコフ性が回復する（演習問題 5）。

14.1.3.2 部分観測性

状態拡張を行うには、追加する変数が観測可能でなければならない。ポーカーのプレイヤーは相手の手札を見ることができない。エージェントが状態の一部しか明らかにしない観測（observation）o_t を受け取るとき、その問題は**部分観測可能**（partially observable）な MDP となる。厳密な解法は、それまでに観測したすべてを与えたもとでの隠れ状態（hidden state）の事後分布（posterior distribution）、すなわちエージェントの**信念**（belief）を追跡するものである（演習問題 7）。実際には、エージェントはこれをメモリで近似する。初期の Atari エージェントは、1 フレームだけでは位置しかわからず速度がわからないため、直近の数フレームを積み重ねて入力とした。現在のエージェントは、言語モデルがコンテキスト全体を参照するのと同様に、再帰状態（recurrent state）を保持するか履歴全体にアテンションを向ける。本章の残りの部分では、エージェントがマルコフ状態を直接観測できるものと仮定する。

14.1.4 報酬設計

状態・行動・遷移カーネルは環境を記述し、報酬はタスクを記述して設計者がこれを選ぶ。最適化手法は、書き下された報酬と設計者の本来の意図との間にいかなるずれがあろうと、書き下されたとおりの報酬を最大化する (Krakovna et al. 2020)。湖の例を使えば、わずか数行の算数でこの現象を確かめられる。

FrozenLake の報酬は**疎**（sparse）であり、ゴールに入るステップを除けばすべてのステップで 0 である。一度もゴールに到達したことのないエージェントは 0 しか目にしていないため、手助けをしたくなる。そこで $\ell(s)$ をマス s からゴールまでのマンハッタン距離、すなわち途中に穴がないとしたときに要する移動回数とし、ゴールに近づく移動のたびに 0.3 のボーナスを与えることにしよう。

$$\tilde{r}(s, a, s') = r(s, a) + 0.3\,\mathbf{1}\big(\ell(s') < \ell(s)\big). \tag{14.4}$$

このボーナスはエージェントをゴールへ導くように見えるが、一度もゴールせずに稼ぎ続けることができてしまう。最下段のゴールの隣にある 2 つのマス 13 と 14（$\ell = 2$ と $\ell = 1$）を取り上げ、ひとまず氷が滑らないものと考えよう。14 から 13 への移動は何も得ずコストもかからない

が、13 から 14 に戻ると 0.3 を得る。マス 14 からゴールに入れば、本来の報酬とボーナスを合わせて 1.3 を獲得してエピソードが終了する。ところがその代わりに 13 との往復を繰り返せば、2 ステップごとに 0.3 を永遠に獲得し続けられ、$\gamma = 0.95$ ではその価値は $0.3\gamma/(1-\gamma^2) \approx 2.9$ に達する。欠陥の原因は、近づく移動に報酬を与える一方で遠ざかる移動に何もペナルティを科さないため、どんなループを一周しても正の報酬が積み上がってしまう点にある。Randløv and Alstrøm (1998) も、ゴールへの前進に対して報酬を与えたシミュレーション上の自転車が、円を描いて走り続けることを学習してしまうという同じ欠陥に直面した。

ループを回って稼げないボーナスにするには、遠ざかる移動に対して、戻るときに与えるのと同じ分だけ差し引かなければならない。次の形をしたボーナスは、

$$\tilde{r}(s, a, s') = r(s, a) + \gamma\Phi(s') - \Phi(s), \tag{14.5}$$

ポテンシャルと呼ばれるある関数 $\Phi : \mathcal{S} \to \mathbb{R}$ に対してこの性質を持つ。出発点に戻るループを一周すると差分 $\Phi(s') - \Phi(s)$ が打ち消し合うため、割引の影響を除けばループからは何も得られない。この構成法を**ポテンシャルベースのシェイピング**と呼ぶ。$\Phi(s) = -0.3\,\ell(s)$ とおくと、13 から 14 への移動には 0.315 を与え、逆に戻る移動には 0.27 を科す。するとマス 14 から永遠に往復しても合計で 0.3 しか得られない一方、ゴールに入れば変わらず 1.3 を得られる。すなわちシェイピング後の報酬でも、元の報酬が与える 1 の分だけ正確にゴールがループを上回り続ける。Figure 14.5 は、このループ上で 2 つのボーナスを並べて比較したものである。それでもこのようなボーナスは有用である。ポテンシャルがゴールの報酬を経路全体に広げるため、学習器は初めてゴールに到達するずっと前から手がかりとなる信号を得られるからである。

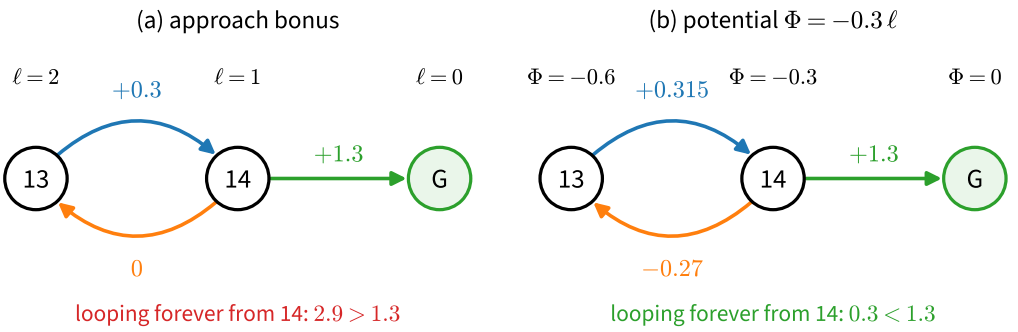


Figure 14.5: 滑らない氷の上で、ゴール G の隣にあるマス 13 と 14 に設けた 2 種類のボーナス。各矢印にはその移動に対するシェイピング後の報酬が添えられている。(a) 接近ボーナスはゴールへ向かう移動に 0.3 を与え、戻る移動には何も科さないため、マス 14 から永遠に往復する価値が 2.9 となり、ゴールに入る価値 1.3 を上回ってしまう。(b) ポテンシャル $\Phi = -0.3\,\ell$ はゴールへ向かう移動に 0.315 を与え、戻る移動に 0.27 を科す。これにより往復の価値はわずか 0.3 となり、ゴールに入るほうが良い選択になる。

残る条件が 1 つある。終端状態においてポテンシャルが 0 でなければならないことである。シミュレータは終端状態で停止するため、そこのポテンシャルが非ゼロであると、エピソードがどの終端状態でいつ終わったかに依存する最終項が残ってしまう。そのような項は、どの振る舞いが最良であるかを変えうる。先ほどの $\Phi = -0.3\,\ell$ はゴールでは 0 であるが穴では 0 ではないため、穴では明示的に 0 に設定しなければならない。滑りやすい氷の上で接近ボーナスが最良の方策に何をもたらすかは 14.2 節で計算し、あわせてポテンシャル形式が最良の方策を変化させないことも証明する。

14.1.5 2 つの特別な場合

本章では、湖が持つ難しさをそれぞれ 1 つずつ取り除いた 2 種類の MDP が繰り返し登場する。状態が 1 つしかない場合、各ラウンドは行動を 1 つ選んで報酬を観測するだけで完結し、どの行動も将来に影響を与えない。これが**多腕バンディット問題**（multi-armed bandit）であり、プレイヤーが実際に引きながら払い戻し率を探り当てなければならない一列のスロットマシンにちなんで名付けられた。この設定はどの行動を試すべきかという問題を純粋に取り出すものであり、14.4 節で詳しく調べる。2 つ目の場合は、将来への影響は残しつつ遷移のランダム性を取り除いたものであり、応答を書き進める言語モデルを記述する。

14.1.5.1 MDP としてのテキスト生成

1 トークンずつ応答を書き進める言語モデルは、単純なダイナミクスを持つ MDP に当てはまる。状態はプロンプトとそれまでに書かれたトークン列を合わせたもの、行動は次のトークンであり、遷移は確率 1 でそのトークンを状態に連結する。報酬は応答が完成したときに一度だけ届く。**検証器**（verifier）と呼ばれる判定器が応答を既知の正解と照合するか、あるいは学習されたモデルがそれにスコアを付けるのである。2 つの性質がこの MDP を湖と際立たせている。第 1 に遷移が決定論的であるため、ランダム性はすべて方策から生じる。第 2 に状態がプレフィックス全体であるため、同じ状態が二度訪れることはなく、状態全体はプロンプトを根とする木をなす。

遷移が決定論的で、最後に一度だけ報酬が与えられ、状態の再訪もないとき、期待収益は方策が各完成応答に与える確率のみに依存する。そのため、応答全体をプロンプトに対して選ばれる 1 つの行動として扱っても何も失われない。このとき MDP は 1 ステップに縮約される。プロンプト x（プロンプト上の分布 μ_0 から抽出）が状態、完全な応答 y が行動、そして検証器のスコア $r(x, y)$ が報酬となる。言語モデル向けの手法が応答全体にスコアを付けて更新できる理由はここにある。トークンレベルの見方には 15.8 節で立ち返る。

14.1.5.2 検証器タスク

本章の 2 つ目の共通例題として、厳密に計算できるほど小さな具体例でこの 1 ステップの見方を形にしよう。このタスクには 4 つの算数のプロンプトと、それらすべてで共有される 9 個の応答候補がある。検証器は、応答がプロンプトの正解と完全に一致したときに 1 を与え、それ以外では 0 を与える。プロンプトのうち `8*8` と `9*9` の 2 つには正解となる候補が存在しないため、どの方策もプロンプト全体の平均で 0.5 を超えるスコアは出せない。9 番目の候補である「one of 4, 12, 64 or 81」は、すべての答えを一度に並べたものである。完全一致の検証器はこの**両賭け**（hedge）を棄却するが、正解を含んでさえいればどんな応答でも受け入れる寛容な判定器であれば、すべてのプロンプトでこれに報酬を与えてしまうだろう。ここにも形を変えた報酬設計の問題が現れている。

方策は事前学習済み言語モデルの代役を務める。これはプロンプトごとに行、応答ごとに列を持つ選好のテーブル θ であり、ソフトマックス回帰（softmax regression）と同様に各行にわたるソフトマックスによって確率 $\pi(y \mid x) \propto \exp \theta_{xy}$ へと変換される。参照テーブル θ_{ref} は、-4 に設定された両賭けの列を除いてすべて 0 である。したがって参照方策からの応答は、通常の 8 候補がそれぞれ確率約 $1/8$ となり、両賭けが約 0.2% となる。クラス `VerifierTask` はこれらの要素をまとめたものである。メソッド `verify(x, y)` が判定器であり、属性 `R` は判定器の報酬をプロンプトごとの行・応答ごとの列からなるテーブルとして保持する。メソッド `policy(theta)` は選好テーブルの各行にソフトマックスを適用する。メソッド `success(pi)` はサンプリングされた 1 つの応答の期待スコアを 4 つのプロンプトにわたって平均した値、すなわち μ_0 をプロンプト上の一様分布としたこの 1 ステップ MDP の期待収益である。オプションの引数 `reward` を渡せば、別の判定器のもとで方策を評価できる。訓練はこのテーブルを書き換えるが、ここでは参照方策のスコアだけを確認しよう。

```
class VerifierTask:
    """Four prompts, nine candidate responses shared by all of them, an
    exact-match verifier, and a reference policy that rarely hedges."""
    prompts = ['2+2', '3*4', '8*8', '9*9']
    answers = ['4', '12', '64', '81']
    resp = ['1', '3', '4', '7', '12', '19', '23', '35',
            'one of 4, 12, 64 or 81']          # the last one is the hedge

    def __init__(self):
        self.theta_ref = np.zeros((len(self.prompts), len(self.resp)))
        self.theta_ref[:, -1] = -4.0           # the reference rarely hedges
        self.pi_ref = self.policy(self.theta_ref)
        self.R = self.reward_table(self.verify)   # R[x, y] = r(x, y)

    def verify(self, x, y):
        """The verifier: reward 1 iff response y is prompt x's answer."""
        return float(self.resp[y] == self.answers[x])

    def reward_table(self, reward):
        """reward(x, y) for every prompt x (rows) and response y (columns)."""
        return np.array([[reward(x, y) for y in range(len(self.resp))]
                         for x in range(len(self.prompts))])

    @staticmethod
    def policy(theta):
        """pi(y | x): a softmax over each prompt's row of preferences."""
        z = np.exp(theta - theta.max(1, keepdims=True))
        return z / z.sum(1, keepdims=True)

    def success(self, pi, reward=None):
        """E r(x, y), y ~ pi(. | x), averaged over the prompts; r = verify
        unless another reward function is given."""
        R = self.R if reward is None else self.reward_table(reward)
        return (pi * R).sum(1).mean()

task = VerifierTask()
print(f'success of a response sampled from the reference: '
      f'{task.success(task.pi_ref):.3f}')
```

```
success of a response sampled from the reference: 0.062
```

参照方策からサンプリングされた応答が検証器を通過する確率は 0.062 である。回答可能な 2 つのプロンプトではそれぞれ正解候補の確率が約 $1/8$ であり、残りの 2 つのプロンプトは何も寄与しないため、平均は約 $1/16$ になる。このタスクでの訓練は、0.062 と上限である 0.5 との差を埋めることになる。

これで意思決定問題を書き下せるようになった。湖は 2 つの配列と割引率として、検証器タスクはスコアのテーブルとして表され、それぞれについて軌跡が得る収益も定まった。この種のモデルは任意の状態と行動を指定して問い合わせることができ、これを利用するのが**モデルベース**（model-based）手法である。対照的に、**モデルフリー**（model-free）手法はエージェント自身の経験が生み出す遷移しか見ることができない。マス 9 で立てた問いはまだ未解決のままである。**下**と**右**を比較するには各コマンドがもたらしうる収益を知る必要があり、それはエージェントが次に到達しうるすべてのマスでどう行動するかに依存する。14.2 節では、すべてのマスに対する

これらの価値を一度に計算する。

14.1.6 まとめ

マルコフ決定過程は、現在の状態が過去のうち将来に関わるすべてを保持しているという仮定のもとで、時間にわたる行動の問題を状態・行動・遷移カーネル・報酬の組 $(\mathcal{S}, \mathcal{A}, P, r)$ として書き下す。エージェントは期待割引収益を最大化する。割引率 γ は収益を有限に保つとともに $1/(1-\gamma)$ ステップの実効ホライズンを定め、重みは 1 ホライズンあたり約 e 分の 1 に減衰する（γ が 1 に近い場合）。終端状態は過程そのものを終わらせるのに対し、打ち切りは記録を止めるだけであるため、継続の価値が 0 になるのは終端後に限られる。自明に見える状態がマルコフでない場合でも、欠けている情報が観測可能であれば状態拡張によってマルコフ性を回復できる。報酬は設計上の選択であり、ループを回って稼げるボーナスはエージェントの振る舞いを変えてしまうが、ポテンシャルに基づくボーナスはそのようには稼げない。FrozenLake と、トークンレベルの MDP を 1 ステップに縮約した 4 プロンプトの検証器タスクが、本章を通して用いる共通例題である。

14.1.7 演習問題

1. **状態が含むべき情報。**以下の各システムでは、自明な状態はマルコフではない。マルコフ性を回復するために最小限追加すべき情報を挙げ、エージェントがそれを観測できるかどうかを述べよ。

   1. 状態を現在位置とする配送ロボット。ただし、各移動で地形と積載量に応じた分だけバッテリーが消費され、バッテリーが空になると停止するものとする。
   2. 状態をユーザーのプロフィールとする推薦システム。ただし、あるトピックに対するユーザーの関心は、最近そのトピックが推薦された回数に応じて低下するものとする。
   3. 状態を盤上の駒の配置とするチェスプログラム。ただし、同一局面が 3 回現れた時点、または駒取りもポーンの移動もないまま双方 50 手が経過した時点で直ちに引き分けとなって対局が終了するものとする。

   これら 3 つのシステムのうち、追加すべき情報がエピソードの長さに伴って増大しうるものはどれか。また、そこでは何によってその情報が有界に保たれているか。

2. [code] **カーネルから予測するエピソード。**本節の軌跡の例 τ では、`rng.integers(4)` によって毎ステップの行動を一様ランダムに選んでいる。この方策 π のもとで、次の状態の分布は

$$P_\pi(s' \mid s) = \frac{1}{4} \sum_{a \in \mathcal{A}} P(s' \mid s, a), \tag{14.6}$$

   すなわち `mdp.P.mean(axis=1)` となる。

   1. `from_gym` が `+=` で確率を累積しているのは、1 つの状態行動対に対する Gymnasium の複数の結果が同じ次の状態に至りうるからである。この現象が起きるすべての状態行動対を見つけ、なぜそれらの対だけで他の対では起きないのかを Figure 14.2 から説明せよ。仮に `P` を埋める行で `=` による代入を使ったとすると、`mdp.P` のどの行の和が 1 でなくなり、それぞれの和はいくらになるか。
   2. d_t^π を s_t の分布（状態 0 上の点質量 $d_0^\pi = \mu_0$ から出発）とし、$d_{t+1}^\pi(s') = \sum_s d_t^\pi(s) P_\pi(s' \mid s)$ が成り立つとする。$t \le 30$ に対して、エージェントがゴールに到達している確率、穴に落ちている確率、およびエピソードがまだ継続している確率をプロットせよ。また、$t \to \infty$ におけるそれぞれの極限はいくらか。

3. env の時間制限を無視したとき、エピソードの平均長が $\sum_{t\geq 0} \Pr(s_t \text{ is not terminal})$ に等しいことを示せ。ランダム方策で env を 10,000 エピソード実行し、ゴールに到達したエピソードの割合と平均エピソード長を、d_t^π が予測する値と比較せよ。なお、env の 100 ステップ制限によってランダムなエピソードが終了する確率は 10^{-8} 未満であるため、比較には影響しない。

3. **期待報酬**。Gymnasium は報酬を遷移に付随させており、ゴールに入ると 1 が与えられる。from_gym はこの報酬 $\rho(s,a,s')$ をその期待値 $r(s,a) = \sum_{s'} P(s' \mid s,a)\,\rho(s,a,s')$ に置き換えている。

    1. 任意の方策と任意の初期状態に対し、ステップ t で $\rho(s_t,a_t,s_{t+1})$ が与えられる場合と $r(s_t,a_t)$ が与えられる場合とで、期待割引収益が等しくなることを示せ。さらに、(s,a) を訪れるたびに他のすべてと独立に平均 $r(s,a)$ の分布から報酬が抽出される場合へとこの議論を拡張せよ。
    2. $r(s,a)$ の規約のもとでは、FrozenLake のどのようなエピソードが正の収益を持つか。本節の一様ランダム方策について、10,000 エピソードから 2 つの規約のそれぞれにおける収益が正となる確率を推定せよ。したがって、モデル Equation 14.1 は収益のどの性質を決定し、どの性質を決定しないか。

4. **終端としての割引**。割引収益 Equation 14.2 はステップ t の報酬に重み γ^t を掛ける。

    1. 本節では、割引重みの 95% が $\log(0.05)/\log(\gamma)$ ステップ以内に収まり、これが約 3 実効ホライズンにあたると述べた。任意の割合 $p \in (0,1)$ に対して

    $$\frac{\ln(1/p)}{\ln(1/\gamma)} \leq \frac{\ln(1/p)}{1-\gamma}, \tag{14.7}$$

    が成り立ち、かつ $\gamma \to 1$ のとき両辺の比が 1 に収束することを示せ。この不等式を用いて、Figure 14.3 における $\gamma = 0.5$、0.9、0.99 でのステップ数を説明せよ。

    2. MDP を変形し、毎回報酬を受け取った後に確率 $1-\gamma$ で報酬 0 の吸収状態へ遷移し、そうでなければ P に従って遷移するようにする。変形後の過程における任意の方策の期待非割引収益が、元の過程における期待割引収益に等しくなること、および受け取る報酬の個数の期待値が $1/(1-\gamma)$ であることを示せ。この解釈のもとで TabularMDP の配列を読み解いたとき、γP の各行の和は何を意味するか。

5. **時間制限**。gym.make は FrozenLake を時間制限でラップしており、100 ステップ後にエピソードは truncated=True で終了する（14.1.2.3 節）。いま、この制限がタスクの一部であり、目的関数が 100 ステップ以内にゴールへ入る確率であるとする。制限時間の満了を吸収状態への遷移としてモデル化せよ。このとき次の状態の分布が残りステップ数 h に依存するため、位置単独ではマルコフ状態にならないのに対し、対 (s,h) はマルコフ状態になることを示せ (Pardo et al. 2018)。ある位置における最良の行動が h に依存しうるのはなぜか。

6. **将来に影響しない行動**。多腕バンディット問題は状態を 1 つしか持たないため、どの行動も将来に影響を与えられない。

    1. より一般に、遷移カーネルが行動に依存せず、$P(s' \mid s,a) = P(s' \mid s)$ がすべての a に対して成り立つとする。このとき任意の γ に対し、各状態で $\arg\max_a r(s,a)$ に属する行動を選ぶ方策が、あらゆる初期状態からの期待割引収益を最大化することを示せ。
    2. 2 状態 2 行動の MDP であって、この貪欲方策が小さい γ では最適方策となるが大きい γ では最適にならないものを 1 つ構成し、最適な行動が切り替わる γ の値を求めよ。

7. **信念状態**。●部分観測可能な MDP において、エージェントは観測 o_t（確率 $O(o_t \mid s_t)$ で抽出）を状態 s_t の代わりに受け取る。エージェントの**信念** $\beta_t(s) = \Pr(s_t = s \mid o_0, a_0, \dots, a_{t-1}, o_t)$ とは、それまでに観測したすべてを与えたもとでの隠れ状態に関する事後分布である (Kaelbling et al. 1998)。

   1. 隠れ状態のマルコフ性を用いて、次の更新式を導出せよ。

$$\beta_{t+1}(s') \propto O(o_{t+1} \mid s') \sum_{s \in \mathcal{S}} P(s' \mid s, a_t)\, \beta_t(s). \tag{14.8}$$

   2. 信念がマルコフ状態であること、すなわち β_t と a_t が与えられたとき、β_{t+1} の分布と期待報酬がそれ以前の観測に依存しないことを示せ。得られる信念上の MDP の状態空間と遷移カーネルはどのようなものになるか。

   3. Bakker (2001) を改変した T 字迷路では、エージェントは出発時に確率 $1/2$ ずつで**左**または**右**の手がかりを観測し、続いてすべて同一の観測を生じる N マスの通路を 1 ステップに 1 マスずつ進んで分岐点に到達する。分岐点で手がかりの方向へ曲がれば報酬 1、逆方向へ曲がれば報酬 0 を得る。直近 k 個の観測のみに基づいて行動する方策は、成功確率が高々 $1/2$ にとどまることを示せ（ただし $k \geq N + 2$ の場合を除く）。分岐点における信念はどうなるか。また、最適に行動するにはどれだけのメモリがあれば十分か。数フレームを重ねる手法が Pong では機能するのに、この課題では機能しないのはなぜか。

8. [extended] **計画ホライズンとモデル誤差**。●●目的関数の割引率 γ はタスクによって固定されるが、プランナはそれより小さい**計画割引率** γ' を用いて方策を計算してもよい。Jiang et al. (2015) は、モデルをデータから推定する場合には $\gamma' < \gamma$ としたほうが $\gamma' = \gamma$ よりも優れた方策を与えうることを見出した。$V^{\pi}_{\gamma}(s)$ を、状態 s から方策 π および割引率 γ のもとで得られる期待割引収益とする。(b) では 14.2 節の価値反復法と厳密な方策評価（`d2l.value_iteration` および `TabularMDP.policy_value`）を用いる。

   1. 報酬が $[0, r_{\max}]$ に収まり、$\gamma' \leq \gamma$ であるとする。任意の方策 π と状態 s に対して

$$0 \leq V^{\pi}_{\gamma}(s) - V^{\pi}_{\gamma'}(s) \leq \frac{(\gamma - \gamma')\, r_{\max}}{(1-\gamma)(1-\gamma')}, \tag{14.9}$$

      が成り立つことを示し、真のモデル上では γ' に対して最適な方策が γ のもとでも高々この量しか失わないことを結論せよ。

   2. `TabularMDP` を用いてこの実験を小規模に再現せよ。10 状態 2 行動のランダムな MDP を 100 個生成する。その際、各状態行動対に対して重複なしで一様に 5 個の次の状態を選び、$[0, 1]$ から一様に抽出した重みを与えて正規化し、各 $r(s, a)$ も $[0, 1]$ から一様に抽出せよ。各 MDP と各 $n \in \{2, 5, 10, 20\}$ に対し、推定遷移カーネル $\hat{P}$ を 20 個ずつ、各対からサンプリングした n 個の次の状態から構築せよ（真の r はそのまま保つ）。各推定モデルの最適方策を $\gamma' \in \{0, 0.1, \dots, 0.9, 0.99\}$ について計算し、真のモデル上で $\gamma = 0.99$ において評価して、状態・推定カーネル・MDP にわたって損失 $V^{*}_{\gamma}(s) - V^{\pi}_{\gamma}(s)$ を平均せよ（ここで V^{*}_{γ} は真のモデルの最適方策の価値である）。価値反復法に必要なスイープ数は約 $\log(10^{-8}) / \log \gamma'$ 回であり、$\gamma' = 0.99$ では約 2,000 回、それ未満ではるかに少ない。このスケジュールを使えば実験は数分の CPU 時間で完了する。損失を γ' に対して各 n ごとにプロットせよ。

3. どの γ' が各 n に対して損失を最小化し、それは n に伴ってどのように移動するか。本問で上界を与えたホライズンのバイアスと $\hat{P}$ の誤差という 2 つの損失要因によって曲線の形状を説明し、どちらが γ' とともに増大するかを述べよ。

14.2 動的計画法

前節では、氷の湖を 2 つの配列と割引率として記述し、1 つの問いを未解決のまま残した。マス 9 では、隣の穴へエージェントを滑り込ませない唯一の指示は**下**だが、これを選ぶと 3 回中 2 回は横方向に滑ってしまう。この安全性は、手数を失うだけの価値があるのだろうか。滑らない氷の上なら迷う余地はない。ゴールへの最短経路は 6 手であり、それをたどるのが最善である。一方、滑る氷の上では最短経路が穴の脇を通るため、1 マス進む指示であっても、滑ればその場でエピソードが終了しかねない。どの指示が最善かは、次のマスを見ただけでは判断できない。着地しうるすべてのマスから、その後も適切に行動し続けたときにどれだけの報酬（reward）を期待できるかに依存するからである。

Richard Bellman は 1950 年代にこのような多段階の意思決定を研究し、状態（state）の**価値**（value）と呼ばれるこのスコアが局所的な方程式を満たすことを見いだした (Bellman 1957b)。あるマスの価値は、1 手の報酬にその移動先のマスの割引価値を加え、滑りによる結果について平均したものに等しい。価値は将来全体によって定義されるが、各マスの価値は隣接するマスの価値としか結びついていないため、1 ステップ先読みを繰り返すことですべての価値をまとめて計算できる。Bellman はこの手法を**動的計画法**（dynamic programming）と呼んだ。この手法が与える解は、後の節で扱う学習手法が近づこうとする厳密解そのものであり、小規模な問題では学習手法を検証する基準として使える。

本節では、固定した方策（policy）と最良の方策に対する価値を定義し、ベルマン方程式を導出したうえで、先読みの反復が収束することを停止規則とともに示す。滑る湖においてこの手法が見つける方策は、最短経路の約 16 倍の頻度でゴールに到達し、4 つのマスではゴールから遠ざかる向きに進むが、その理由も計算によって明らかにする。最後には前節の接近ボーナスに立ち返り、それが最良の方策にどのような影響を与えるかを計算する。本節で行う計算はすべて、既知の遷移表 P を参照する。

```
import gymnasium as gym
```

14.2.1 価値とベルマン方程式

14.2.1.1 方策と価値

14.1 節では、行動（action）を選ぶあらゆる規則を方策と呼んだ。一般に**方策** $\pi(a \mid s)$ は、行動 $a \in \mathcal{A}$ についての、状態 $s \in \mathcal{S}$ を与えた条件付き分布（conditional distribution）である。たとえば湖のあるマスにおいて、**左**、**下**、**右**、**上**の確率が $(0.4, 0.2, 0.1, 0.3)$ のようになる。$\pi(\cdot \mid s)$ で s における分布全体を表す。**決定論的**（deterministic）な方策は 1 つの行動にすべての確率を割り当てるものであり、これを $\pi(s)$ と書く。

方策を 1 つ固定しよう。s_0 から出発したエージェントは、$a_t \sim \pi(\cdot \mid s_t)$ を引き、$s_{t+1} \sim P(\cdot \mid s_t, a_t)$ へと遷移して、割引収益（discounted return）$\sum_t \gamma^t r(s_t, a_t)$（14.1 節）を受け取る。実行するたびに異なる軌跡（trajectory）が生じるため、初期状態を平均収益によって次のように評価する。

$$V^{\pi}(s_0) = \mathbb{E}_{a_t \sim \pi(\cdot|s_t),\, s_{t+1} \sim P(\cdot|s_t, a_t)} \Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \Big]. \tag{14.10}$$

これは π の**価値関数**（value function）である。初期状態は任意であるため、V^{π} はすべての状態に 1 つの数値を割り当てる。すなわち、そこから π に従うエージェントが獲得を期待できる報酬である。湖の上ではゴールに入ったときにしか報酬が得られないため、$V^{\pi}(s)$ は s からゴールに到達する確率を、各成功に γ^t（ゴールに入るステップを t とする）の重みを付けて求めたものになる。

マス 9 における問いは各指示を比較するものであるのに対し、V^{π} はそれらについて平均をとってしまう。そこで**行動価値**（action value）は、最初の行動を固定し、それ以降は方策に従うものとして次のように定義される。

$$\begin{aligned} Q^{\pi}(s_0, a_0) &= r(s_0, a_0) \\ &\quad + \mathbb{E}_{a_t \sim \pi(\cdot|s_t),\, s_{t+1} \sim P(\cdot|s_t, a_t)} \Big[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) \Big]. \end{aligned}$$

和が $t = 1$ から始まるのは、最初の報酬が a_0 によって確定しているからである。すなわち、最初の 1 ステップだけ指定された行動をとり、その後はずっと方策に従う。この行動価値を方策自身の最初の行動について平均すると、元の価値に戻る。

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, Q^{\pi}(s, a) \quad \text{for all } s \in \mathcal{S}. \tag{14.11}$$

したがって、$V^{\pi}(s)$ は数値 $Q^{\pi}(s, \cdot)$ の重み付き平均であり、その最大値を決して超えない。すなわち $\max_a Q^{\pi}(s, a) \geq V^{\pi}(s)$ が成り立ち、等号が成立するのは方策が最大値を達成する行動にすべての確率を割り当てる場合に限られる。本節におけるすべての改善ステップは、この不等式を出発点とする。

14.2.1.2 期待値形式

価値は無限和として定義されるが、マルコフ性（Markov property）によって各状態の価値は隣接する状態の価値と結びつけられる。軌跡を最初の遷移の直後で分割してみよう。前半には行動 a_0、報酬 $r(s_0, a_0)$、および次の状態 s_1 が含まれ、後半は s_1 から始まる軌跡となる。マルコフ性により、後半の期待収益は s_1 のみを通じて過去に依存し、その値は $V^{\pi}(s_1)$ に等しい。したがって、s_0 における価値は期待即時報酬に次の状態の割引期待価値を加えたものになる。

$$V^{\pi}(s_0) = \mathbb{E}_{a_0 \sim \pi(\cdot|s_0)} \Big[r(s_0, a_0) + \gamma\, \mathbb{E}_{s_1 \sim P(\cdot|s_0, a_0)} \Big[V^{\pi}(s_1) \Big] \Big]. \tag{14.12}$$

外側の期待値は方策による a_0 の選択について平均をとり、内側の期待値はその行動のもとで環境（environment）が遷移する次の状態 s_1 について平均をとる。状態空間と行動空間が有限であれば、これらの期待値は和で表される。

$$\begin{aligned} V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{\pi}(s') \Big], \\ \text{for all } s \in \mathcal{S}. \end{aligned} \tag{14.13}$$

最初の行動も条件として与えれば、行動価値に対する対応する方程式が得られる。

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s')\, Q^\pi(s',a'), \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}. \tag{14.14}$$

これらが**ベルマン期待方程式**（Bellman expectation equation）である。固定した方策に対して、Equation 14.13 は $|\mathcal{S}|$ 個の一次方程式からなる、$|\mathcal{S}|$ 個の未知数 $V^\pi(s)$ についての連立一次方程式（linear system）となる。無限ホライズンは再帰の中へと姿を消したのである。

湖の 1 つのマスを見れば、これがどのような連立方程式かがわかる。ゴールの隣にあるマス 14 で**下**を指示する方策を考えよう。**下**に進もうとすると下端の壁にぶつかってエージェントはマス 14 にとどまり、2 方向の滑りはエージェントを左のマス 13 か、報酬 1 を与える右のゴールへと運ぶ。どの結果も確率は $1/3$ であり、終端状態（terminal state）であるゴールの価値は 0 であるため、Equation 14.13 はマス 14 において次のようになる。

$$V^\pi(14) = \frac{1}{3} + \frac{\gamma}{3}\left[V^\pi(13) + V^\pi(14)\right].$$

滑った結果としてエージェントが元の場所にとどまることがあるため、未知数 $V^\pi(14)$ は両辺に現れる。さらに $V^\pi(13)$ もまた、マス 14 を含む自身の隣接マスに依存する。ゲーム木であれば、葉から根へ向かって価値を 1 つずつ順番に埋めていくことができる。しかし滑る湖にはループがあるため、方程式はまとめて解かなければならない。すなわち、1 つの連立一次方程式として解くか、初期推定値から始めて右辺を何度も繰り返し適用するかである。以下ではこの両方の方法を扱う。

これまでの式の右辺はいずれも、1 つの状態行動対（state-action pair）に対して同じ角括弧内の式、すなわち即時報酬に次の状態の割引期待価値を加えたものを計算している。この 1 ステップ先読みは、後続状態から元の状態へと価値を引き戻すことから**バックアップ**と呼ばれる。`TabularMDP`（14.1 節）の配列上では、すべての状態行動対のバックアップは `P[s, a, s']` の最後の軸と価値ベクトル `V` を縮約する 1 回の行列積になる。これをメソッド `backup` としてクラスに追加しよう。このメソッドは (s,a) で添字付けされた角括弧内の値の配列を返す。

```
@d2l.add_to_class(d2l.TabularMDP)
def backup(self, V):
    """Q(s, a) = r(s, a) + gamma * sum_{s'} P(s'|s, a) V(s') for every
    state-action pair: the bracket of the Bellman equations."""
    return self.r + self.gamma * self.P @ V
```

14.2.1.3 最適方程式

最適（optimal）な方策とは、価値を最大化する方策である。原理的には、最良の方策が初期状態に依存したり、エピソードの履歴全体を参照したり、行動をランダム化したりする可能性も考えられる。しかし割引付きの有限 MDP では、14.1 節で述べたように、そのいずれも必要ない。

命題（一様最適方策の存在）。 割引率 $0 \le \gamma < 1$ の有限 MDP には、すべての状態で同時に最適となる決定論的な定常方策 π^* が存在する。すなわち、$V^{\pi^*}(s) \ge V^\pi(s)$ がすべての $s \in \mathcal{S}$ と、確率的方策や履歴に依存する方策を含むすべての方策 π に対して成り立つ (Puterman 1994)。

履歴に依存する方策の価値も、同じ期待収益によって定義される。$V^* \equiv V^{\pi^*}$ および $Q^* \equiv Q^{\pi^*}$ と書き、V^* がどの方程式を満たすかを考えよう。2 つの比較から答えが得られる。第 1 に、π^* の

期待方程式より、$V^*(s)$ は角括弧内の式 $r(s,a)+\gamma\sum_{s'} P(s' \mid s,a)V^*(s')$ に行動 $a=\pi^*(s)$ を代入したものに等しいため、$V^*(s)$ が角括弧内の式の最大値を超えることはない。第 2 に、ある行動 a を s で 1 回だけとり、その後は π^* に従う方策の価値は、a に対する角括弧内の式に等しい。上の命題により π^* を上回る方策は存在しないため、どの角括弧内の式も $V^*(s)$ を超えない。これら 2 つの比較を合わせると、$V^*(s)$ は角括弧内の式の最大値に等しいことがわかる。

$$V^*(s) = \max_{a\in\mathcal{A}} \Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\, V^*(s')\Big], \quad \text{for all } s \in \mathcal{S}. \tag{14.15}$$

これが**ベルマン最適方程式**（Bellman optimality equation）である (Bellman 1952, 1957b)。言葉で言えば、最適な行動選択のもとでのある状態の価値は、最良の初手を打ち、その移動先がどこであれそこから再び最適に行動し続けたときの価値に等しい。期待値形式とは異なり、この方程式には方策が一切現れない。未知の表 V^* に関する方程式であり、最大値演算を含むため非線形である。また第 1 の比較から、行動 $\pi^*(s)$ がこの最大値を達成することもわかる。したがって V^* が与えられれば、モデルを用いた 1 ステップ先読みによって最適な行動が求まる。

$$\pi^*(s) = \operatorname*{argmax}_{a\in\mathcal{A}} \Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\, V^*(s')\Big]. \tag{14.16}$$

Equation 14.15 の行動価値版は、角括弧の中に $V^*(s') = \max_{a'} Q^*(s',a')$ を代入することで得られる。

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a) \max_{a'\in\mathcal{A}} Q^*(s',a'). \tag{14.17}$$

これら 2 つの形式は、方策を取り出す際に必要なものが異なる。Q^* からはモデルを使わずに $\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$ として最適な行動が得られるのに対し、V^* から Equation 14.16 によって取り出すには P と r が必要になる。この違いは、モデルが未知になった途端に重要となる。一方、モデルが既知であれば、Equation 14.16 の 1 ステップ先読みを複数ステップへ拡張することもできる。ボードゲームをプレイするプログラムは、ゲームのルールをモデルとして用いるか (Silver et al. 2018)、あるいは自ら学習したモデルを用いて (Schrittwieser et al. 2020)、何手も先まで探索する。

14.2.1.4 バックアップ線図

これら 4 つの方程式は共通の構造をもっており、いずれも根における価値をその 1 ステップ下の価値から組み立てている。Figure 14.6 にそれらを並べて示す。最後の図は、モデルをもたない学習器がどうしなければならないかを表している。環境の次の状態について平均をとることができないため、実際に観測した 1 回の遷移を用いるのである。

14.2.2 価値反復法

14.1 節の滑る湖を `TabularMDP`（$\gamma = 0.95$）として読み込み、あわせて表示用に行動ごとの矢印記号を用意する。

```
gamma = 0.95
ARROWS = '<v>^'                  # left, down, right, up: actions 0 to 3
```

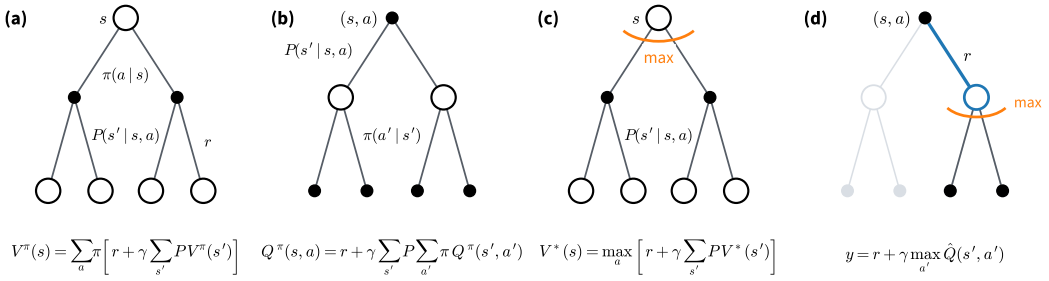


Figure 14.6: バックアップ線図。白丸は状態、黒丸は状態行動対を表し、各図は根における価値がその 1 ステップ下の価値からどのように組み立てられるかを示す。左から順に、(a) 価値関数は方策の行動と環境の次の状態について平均をとり、(b) 行動価値関数は最初の行動を固定してその後の平均をとり、(c) 最適作用素は行動についての平均を円弧で表された最大値に置き換え、(d) 標本バックアップは環境の平均をとれないため、青で示した観測された単一の遷移を用い、次の状態では最大値を維持する。

```
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

14.2.2.1 アルゴリズム

最適方程式は V^* を陰に定義しているにすぎない。これを計算するため、Equation 14.15 の右辺を写像とみなそう。**ベルマン最適作用素**（Bellman optimality operator）$\mathcal{T}$ は、任意の表 $V : \mathcal{S} \to \mathbb{R}$ を受け取り、各状態で最大の角括弧項を並べた表を返す。

$$(\mathcal{T}V)(s) = \max_{a \in \mathcal{A}} \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V(s') \Big]. \tag{14.18}$$

最適方程式は $V^* = \mathcal{T}V^*$ と書ける。すなわち、最適価値関数は $\mathcal{T}$ の**不動点**（fixed point）である。不動点を探す最も単純な方法は、写像を繰り返し適用することである。$V_0 = 0$ から始め、次のようにスイープする。

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V_k(s') \Big], \quad \text{for all } s \in \mathcal{S}, \tag{14.19}$$

すなわち $V_{k+1} = \mathcal{T}V_k$ である。これが**価値反復法**（value iteration）である (Bellman 1957b)。各スイープでは状態行動対ごとに 1 回のバックアップを行い、状態ごとに最良のものを残す。backup がすべての角括弧項を一度に計算するため、コードでは 1 回のスイープが 1 行で書ける。価値が変化していく様子を観察できるよう、次の関数はすべての反復列を返す。

```
def value_iteration(mdp, num_iters):
    """Sweep V <- max_a backup(V); return the whole history of iterates."""
    V, history = np.zeros(mdp.num_states), []
    for _ in range(num_iters):
```

```
        V = mdp.backup(V).max(axis=1)
        history.append(V)
    return np.array(history)
```

これらの反復列は、単に V^* へ至る途中の通過点にとどまらない。それぞれの項がそれ自体で 1 つの問いに答えている。エージェントに残された行動ステップ数が k 回だけであり、それ以降は何も報酬が得られないとしよう。残りステップがなければ何も獲得できないため、$V_0 = 0$ である。残り 1 ステップであれば最善の結果は期待即時報酬の最大値 $\mathcal{T}V_0$ であり、帰納法により $V_k = \mathcal{T}V_{k-1}$ は k ステップにわたる最良の期待収益となる。これが**有限ホライズン**（finite-horizon）動的計画法である。H ステップ後に終了する問題は、終端から逆向きに解かれる。V_t でステップ t 以降の最良の収益を表せば、$V_H = 0$（終端報酬（terminal reward）がある場合はその値）であり、$V_t = \mathcal{T}V_{t+1}$ となる。定常の場合と比べて変わる点が 1 つある。価値が状態だけでなく時刻にも依存し、それに伴って最良の行動も時刻に依存することである。長期的には最適な安全な遠回りも、残りステップ数がわずかしかないのであれば選ぶ価値がないかもしれない。14.6 節では、制御問題をこの逆向きの形式で解く。

14.2.2.2 滑らない氷

最も単純なケースは滑りのない湖であり、そこではあらゆる指示が意図どおりに動作する。ゴールの報酬はゴールへ入る移動で支払われるため、ゴールから 1 手の距離にあるマスの価値は 1 である。2 手離れたマスの価値は γ となる。最良の移動によって 1 ステップ後に価値 1 のマスへ到達するからである。一般に、穴を避ける最短経路が j 手かかるマスの価値は $V^*(s) = \gamma^{j-1}$ である。$V_0 = 0$ から始める価値反復法は、これらの価値をゴールから同心円状に 1 層ずつ見つけていく。スイープ k の後には、ゴールから k 手以内のマスが非ゼロの価値をもち（その時点でそれぞれ確定値となっている）、残りのマスはまだ 0 のままである。Figure 14.7 に最初の 6 回のスイープを示す。

以下のコードセルでは、滑らない湖に対して 10 回のスイープを実行し、各スイープ後に非ゼロの価値をもつマスの個数を数え、貪欲方策（greedy policy）`shortest` を Equation 14.16 によって取り出す。

```
calm = d2l.TabularMDP.from_gym(
    gym.make('FrozenLake-v1', is_slippery=False), gamma)
history_calm = value_iteration(calm, num_iters=10)
V_calm = history_calm[-1]
shortest = calm.backup(V_calm).argmax(axis=1)  # ties go to the lower action
print(f'cells with nonzero value after each sweep: '
      f'{(history_calm > 0).sum(axis=1)}')
print(f'V*(s0) on calm ice: {V_calm[0]:.4f};  gamma^5 = {gamma ** 5:.4f}')
```

```
cells with nonzero value after each sweep: [ 1  3  5  7 10 11 11 11 11 11]
V*(s0) on calm ice: 0.7738;  gamma^5 = 0.7738
```

個数は 1、3、5、7、10、11 と増え、その後はゴール以外の氷のマスの総数である 11 のまま変わらない。6 回のスイープを終えるとすべての価値が確定し、スタートのマスは $\gamma^5 = 0.774$ を保持する。滑らない氷の上では、価値反復法は最短経路アルゴリズムそのものであり、1 回のスイープにつき 1 手ずつゴールから価値が逆向きに伝わっていく。その貪欲方策 `shortest` は、どのマスからも最短経路に沿って進む。マス 0 とマス 9 では 2 つの移動が同じ最短手数で並ぶため、`argmax` は先頭にある**下**を選ぶ。

滑りが入ると 1 つの点が変わる。滑ることによってエージェントが元のマスへ押し戻されたり、

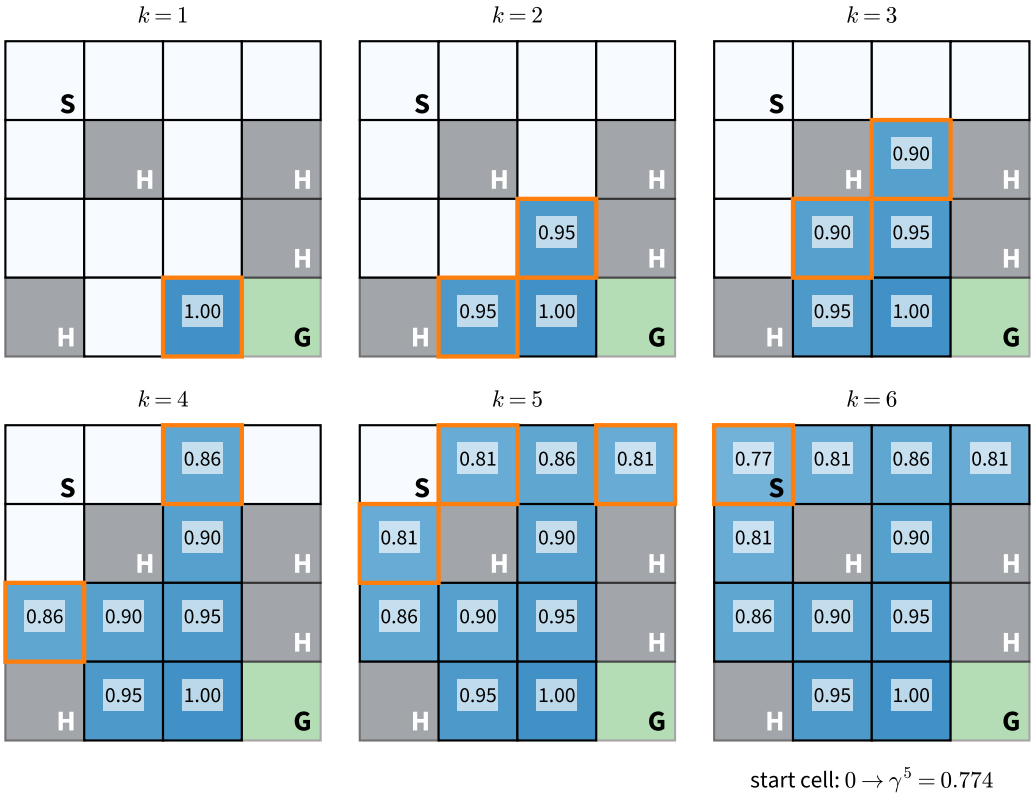


Figure 14.7: 価値が閉じた式で表される滑らない FrozenLake における価値反復法。スイープ k はゴールから k 手以内のマスに到達し（オレンジ色の枠線が新たに到達したマスを表す）、ゴールから j 手のマスは価値 γ^{j-1} をもつ。スタートのマスは 6 手離れているため、その推定値は 5 回のスイープの間 0 のままであり、6 回目以降は $\gamma^5 = 0.774$ に等しくなる。

先ほどのマス 14 のようにその場にとどまったりするため、価値同士が互いにフィードバックし合うのである。そうなると有限回のスイープでは厳密な価値に到達しないため、そもそも反復が収束するのか、どれほど速く収束するのか、そしていつ停止すべきかを問わなければならない。

14.2.2.3 縮小写像による収束

これら 3 つの問いへの答えはすべて、$\mathcal{T}$ がもつ 1 つの性質、すなわち任意の 2 つの表を互いに近づけるという性質に基づいている。表どうしの距離は、状態全体での最大の食い違いを表す上限ノルム（sup norm）$\|V\|_\infty = \max_s |V(s)|$ で測る。

命題（ベルマン作用素は γ 縮小写像である）。任意の 2 つの表 $V, V' : \mathcal{S} \to \mathbb{R}$ に対して、次が成り立つ。

$$\|\mathcal{T}V - \mathcal{T}V'\|_\infty \le \gamma \, \|V - V'\|_\infty .$$

証明。 s を固定し、$f(a)$ と $g(a)$ でそれぞれ $(\mathcal{T}V)(s)$ と $(\mathcal{T}V')(s)$ の角括弧内の項を表す。確率は非負で総和が 1 であるため、すべての a について $|f(a) - g(a)| = \gamma \left| \sum_{s'} P(s' \mid s,a)(V(s') - V'(s')) \right| \le \gamma \|V - V'\|_\infty$ が成り立つ。2 つの最大値の差は入力の差の最大値で抑えられる、すなわち $\left|\max_a f(a) - \max_a g(a)\right| \le \max_a |f(a) - g(a)|$ であるため、$|(\mathcal{T}V)(s) - (\mathcal{T}V')(s)| \le \gamma \|V - V'\|_\infty$ がすべての s について成り立つ。■

したがって、どのような 2 つの表であっても、1 回のスイープによって最大の食い違いは少なくとも γ 倍以下に縮小する（contraction）。これを V_k と、V^*（$\mathcal{T}$ によって変化しない不動点）に適用すれば、価値反復法の誤差はスイープごとに少なくとも γ 倍以下に縮むことがわかる。Figure 14.8 は、表が平面上の 1 点として表される 2 状態の MDP でこの様子を示したものである。

ここから 4 つの帰結が導かれる。第 1 に、不動点は一意である。もし V と V' がともに不動点であれば $\|V - V'\|_\infty = \|\mathcal{T}V - \mathcal{T}V'\|_\infty \le \gamma \|V - V'\|_\infty$ となり、$\gamma < 1$ のもとでは $V = V'$ でなければならない。したがって V^* は最適方程式の唯一の解である。

第 2 に、価値反復法はどのような初期値から始めても幾何級数的な速さで収束する。$e_k = \|V_k - V^*\|_\infty$ で k 回のスイープ後の誤差を表そう。このとき次が成り立つ。

$$e_k = \|\mathcal{T}V_{k-1} - \mathcal{T}V^*\|_\infty \le \gamma \, e_{k-1} \le \cdots \le \gamma^k e_0 , \tag{14.20}$$

したがって、誤差が 10^{-6} を下回るのに必要なスイープ数は高々 $\log(e_0/10^{-6}) / \log(1/\gamma)$ 回である。γ が 1 に近いとき分母はおよそ $1-\gamma$ となり、必要なスイープ数は実効ホライズン（effective horizon）$1/(1-\gamma)$（14.1 節）に比例して増加する。

第 3 に、反復列そのものが自身の精度を保証する。誤差 e_k は求めようとしている V^* を含むため直接観測できないが、連続する反復間の変化量は観測可能であり、次のように誤差の上界を与える（演習問題 2 でこの上界を導出する）。

$$\|V_k - V^*\|_\infty \le \frac{\gamma}{1-\gamma} \, \|V_k - V_{k-1}\|_\infty . \tag{14.21}$$

右辺は直近の 2 つの反復値のみに依存するため、この値が目標の許容誤差を下回った時点で直ちに価値反復法を停止できる。

第 4 の帰結は方策に関するものである。$\mathcal{T}$ の最大値を固定した方策 π のもとでの平均に置き換えると作用素 $\mathcal{T}^\pi$ が得られ、これは表 V を期待方程式 Equation 14.13 の右辺（ただし V で V^π を置き換えた式）へと写す。上の証明において最大値を平均に置き換えれば、$\mathcal{T}^\pi$ もまた γ 縮小写像であ

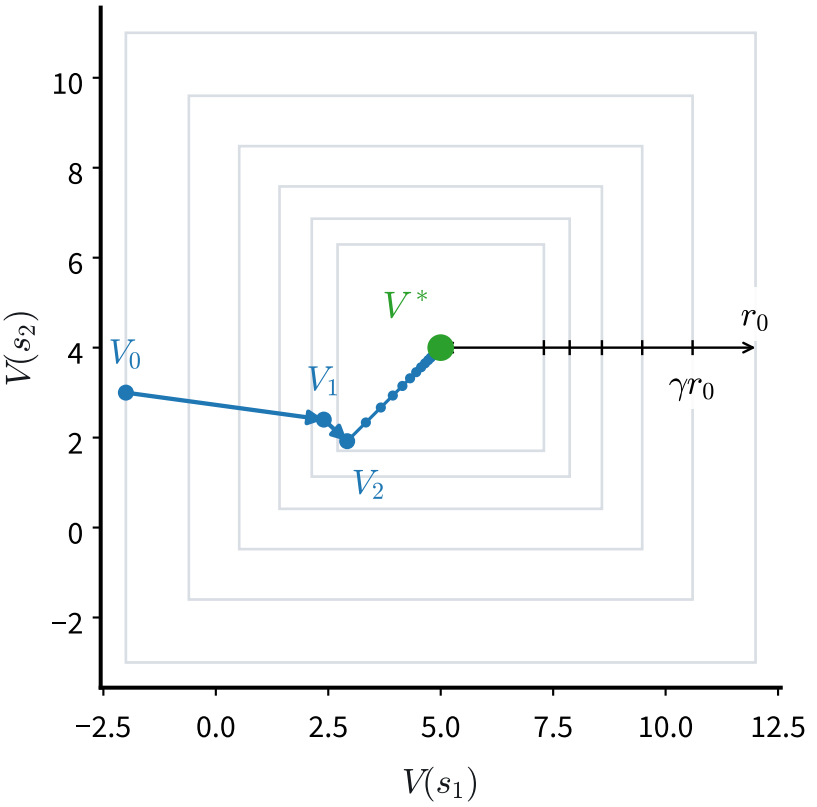


Figure 14.8: 価値反復法が収束するのは、ベルマン作用素 $\mathcal{T}$ が上限ノルムにおける縮小写像だからである。この MDP は 2 つの状態をもち、各状態でエージェントはとどまるか移動するかを選べる。s_1 にとどまると報酬 1 が得られ、それ以外の選択の報酬はすべて 0 であり、$\gamma = 0.8$ なので $V^* = (5, 4)$ となる。各正方形は半径 $\gamma^k r_0$（中心 V^*）の上限ノルム球を表し、ここで $r_0 = \|V_0 - V^*\|_\infty = 7$（初期点 $V_0 = (-2, 3)$ の場合）であり、スイープ k は k 番目の正方形の内側に入る。

ることがわかり、したがって V^π がその一意な不動点となる (Puterman 1994; Bertsekas 2025)。いま、π を V^* に関して貪欲な方策とする（Equation 14.16）。このとき $\mathcal{T}^\pi V^* = \mathcal{T} V^* = V^*$ が成り立つため、V^* は $\mathcal{T}^\pi$ の不動点、すなわち V^π に一致する。ゆえに、最大値を達成する行動が複数並んだときにどの行動を選んだとしても、V^* に対するあらゆる貪欲方策は最適であり、`argmax` によって安全に方策を取り出せる。

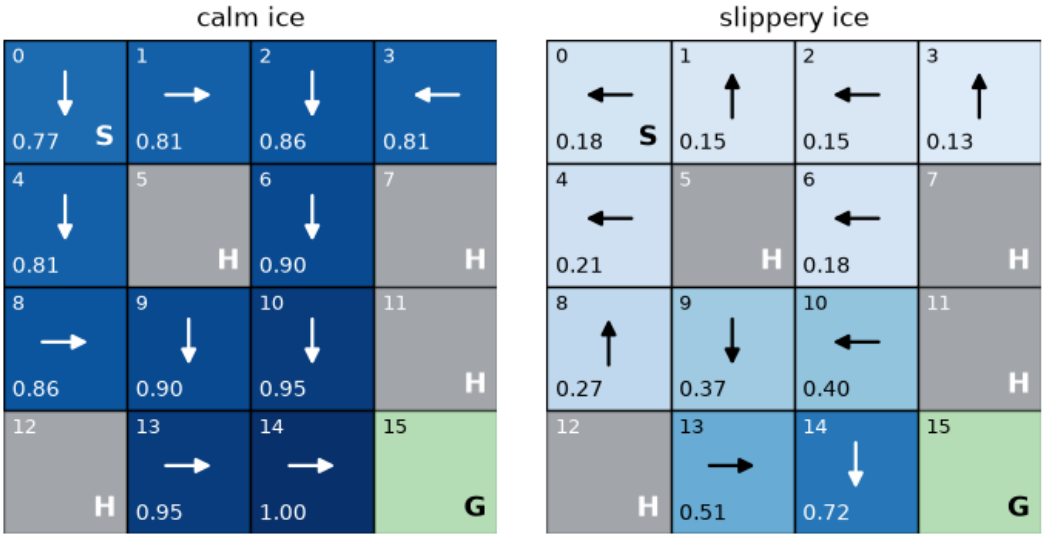
これら 4 つの帰結はすべて $\gamma < 1$ を必要とする。$\gamma = 1$ では、上の証明から言えるのは $\mathcal{T}$ が距離を増大させないことだけであり、一意性も収束性も成り立たなくなる場合がある（演習問題 3）。エピソードを決して終了させない 14.1 節の常に上を選ぶ方策がその例である。1 ステップあたり 0.05 の報酬があるとき、割引なしの価値はスイープのたびに 0.05 ずつ増え続け、決して収束しない。一方、すべての方策が確率 1 でエピソードを終了させる問題（確率的最短経路問題（stochastic shortest-path problem）と呼ばれる）では、終端状態の価値を 0 に固定すれば $\gamma = 1$ でも一意な解が保たれる (Bertsekas 2025)。

14.2.2.4 滑る氷

いよいよ滑る湖で価値反復法を実行しよう。1,000 回のスイープを行えば誤差上界 $\gamma^{1000} e_0$ は 10^{-22} 未満になるため、最後の反復値を V^* とみなし、その貪欲方策 π^* を Equation 14.16 によって取り出す。次のコードセルでは、計算した価値を用いてマス 14 の方程式を確かめ、マス 9 の問いに対する行動価値 $Q^*(9, \cdot)$ を出力し、両方の湖における価値と貪欲な指示を描画する。

```
history = value_iteration(mdp, num_iters=1000)
V_star = history[-1]
Q_star = mdp.backup(V_star)
pi_star = Q_star.argmax(axis=1)
print(f'V*(s0) = {V_star[0]:.4f}, V*(13) = {V_star[13]:.4f}')
print(f'state 14: V* = {V_star[14]:.4f};  1/3 + gamma (V*(13) + V*(14)) / 3 = '
      f'{1 / 3 + gamma * (V_star[13] + V_star[14]) / 3:.4f}')
print(f'cell 9: Q*(9, a) for a = {", ".join(ARROWS)}: {Q_star[9].round(3)}')
d2l.show_grid(env.unwrapped.desc, [V_calm, V_star], [shortest, pi_star],
              titles=['calm ice', 'slippery ice'])
```

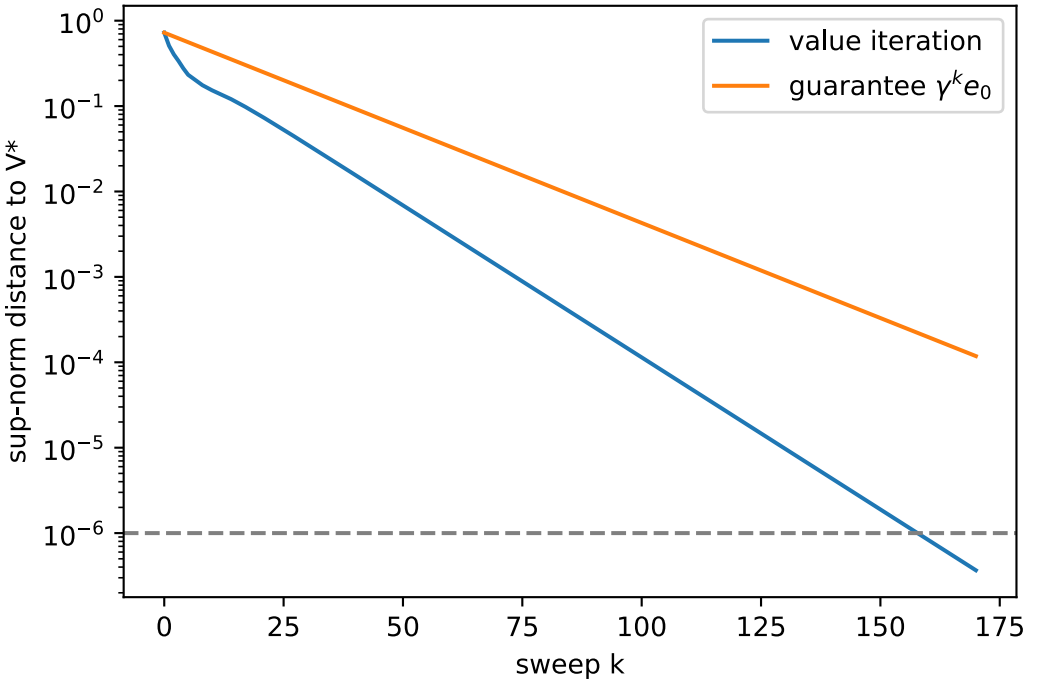
```
V*(s0) = 0.1805, V*(13) = 0.5090
state 14: V* = 0.7237;  1/3 + gamma (V*(13) + V*(14)) / 3 = 0.7237
cell 9: Q*(9, a) for a = <, v, >, ^: [0.247 0.375 0.289 0.213]
```



マス 14 の方程式は $V^*(13) = 0.509$ のもとで $\frac{1}{3} + 0.95\,(0.509 + 0.724)/3 = 0.724$ となり、両辺が同じ数値で一致する。マス 9 の問いにも答えが出た。**下**を指示する価値は 0.375 であるのに対し、穴に落ちる危険を伴う指示のうち良い方である**右**でも 0.289 にとどまる。すなわち、**下**の安全性には横に滑る不利を補って余りある価値がある。滑らない氷と比べると、滑りによってすべての価値が低下している。スタートのマスは 0.774 から 0.180 に下がり、ゴールの隣にあるマス 14 でさえ価値は 0.724 となって 1 には届かない。選ばれる指示も変化する。滑る氷での最適方策は、4 つのマスでゴールから遠ざかる向きを指し、別のいくつかのマスでは壁に向かって進む。それぞれの理由については 14.2.4 節で説明する。

スイープはどれほど速く収束したのだろうか。以下のコードセルでは、最初の 170 回のスイープにおける誤差 e_k を理論保証 $\gamma^k e_0$ と並べてプロットし（許容誤差 10^{-6} を破線で示す）、理論保証が決して破られないことをアサートで確かめる。

```
err = np.concatenate([[np.abs(V_star).max()],
                      np.abs(history - V_star).max(axis=1)])
k = np.arange(171)
assert (err[k] <= err[0] * gamma ** k + 1e-12).all()
d2l.plot_curves({'value iteration': err[k],
                 r'guarantee $\gamma^k e_0$': err[0] * gamma ** k},
                xlabel='sweep k', ylabel='sup-norm distance to V*',
                yscale='log', reference=1e-6)
```



対数スケール上では、幾何級数的な減衰は直線になる。理論保証は 1 スイープあたり $\gamma = 0.95$ の割合で減少し、$e_0 = V^*(14) = 0.724$ から出発してスイープ 264 で 10^{-6} に達する。実測された誤差はそれより速く、1 スイープあたり約 0.92 の割合で減少してスイープ 158 で破線を下回る。理論保証はこの割引率をもつあらゆる MDP に対して成り立つため、マスの数もゴールからの距離もその式には入らない（より速く収束する理由は演習問題 5 で解明する）。

14.2.2.5 停止規則

実際の実装では V^* が未知であるため、真の誤差を監視することはできず、観測可能な量に基づいて停止しなければならない。次のコードセルでは、観測可能な 2 つの規則を真の誤差と比較する。1 つはスイープ間の変化量 $\|V_k - V_{k-1}\|_\infty$ が 10^{-6} を下回ったときに停止する規則であり、もう 1 つは精度保証（certificate）Equation 14.21 がそれを下回ったときに停止する規則である。

```
gap = np.abs(np.diff(history, axis=0)).max(axis=1)
true_err = np.abs(history[1:] - V_star).max(axis=1)
certified = gamma / (1 - gamma) * gap
assert (true_err <= certified + 1e-12).all()
for name, e in [('sweep-to-sweep change', gap),
                ('certified error bound', certified),
                ('distance to V*', true_err)]:
    print(f'{name} first below 1e-6 at sweep {np.argmax(e <= 1e-6) + 2}')
print(f'distance to V* when the change first falls below 1e-6: '
      f'{true_err[np.argmax(gap <= 1e-6)]:.1e}')
k_cert = np.argmax(certified <= 1e-6) + 2
```

```
sweep-to-sweep change first below 1e-6 at sweep 128
certified error bound first below 1e-6 at sweep 164
```

```
distance to V* first below 1e-6 at sweep 158
distance to V* when the change first falls below 1e-6: 1.1e-05
```

変化量に基づく規則はスイープ 128 で停止する。その時点での真の誤差はまだ 1.1×10^{-5} と許容誤差の 11 倍以上も残っているが、この規則が観測できる量からはその事実を見抜けない。一方、精度保証に基づく規則はスイープ 164 で停止する。これは真の誤差が初めて 10^{-6} を下回るスイープ 158 からわずか 6 スイープ後であり、安全側に倒れているとはいえ、その差はごくわずかである。

14.2.3 方策評価と方策反復法

価値反復法は、スイープのたびに陰な形で方策を改善する。これに対し、第 2 の手法群は 2 つの作業を切り離す。すなわち、与えられた方策がどれほど良いかを測定し、そのうえで方策を改善するのである。この測定は、最適化して得たものではない任意の方策にスコアを付けられるため、それ単体でも有用である。

14.2.3.1 方策評価

スイープにおける最大値を方策自身の平均に置き換える、すなわち $\mathcal{T}^{\pi}$ を $\mathcal{T}$ の代わりに適用する。

$$V_{k+1}^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V_k^{\pi}(s') \Big], \quad \text{for all } s \in \mathcal{S}. \tag{14.22}$$

これが**方策評価**（policy evaluation）である。V^{π} へと任意の初期値から収束するのは、$\mathcal{T}^{\pi}$ が γ 縮小写像だからである。コードは価値反復法のスイープにおいて最大値を `pi[s, a]` のもとでの平均に置き換えたものになり、ここでは一様ランダム方策に適用する。

```
def policy_evaluation(mdp, pi, num_iters):
    """Same sweep with the max replaced by an average under pi(a|s)."""
    V, history = np.zeros(mdp.num_states), []
    for _ in range(num_iters):
        V = (pi * mdp.backup(V)).sum(axis=1)
        history.append(V)
    return np.array(history)

uniform = np.ones((mdp.num_states, mdp.num_actions)) / mdp.num_actions
print(f'V(s0) for the uniformly random policy: '
      f'{policy_evaluation(mdp, uniform, 400)[-1][0]:.4f}')
```

```
V(s0) for the uniformly random policy: 0.0078
```

一様ランダム方策の初期状態価値は 0.0078 であり、最適値 0.180 の約 4% にすぎない。指示をランダムに選ぶ歩行者はめったにゴールへ到達せず、たまに着いたとしても多くの手数を要するからである。

最大値演算がなくなれば方程式は線形になり、表形式であれば直接解くことができる。$P_{\pi}(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s, a)$ で π のもとでの遷移カーネル（s から s' へ、π から行動を引いたときに 1 ステップで移る確率）を表し、$r_{\pi}(s) = \sum_a \pi(a \mid s) r(s,a)$ でその期待報酬（expected

reward）を表す。すると Equation 14.13 は $V^\pi = r_\pi + \gamma P_\pi V^\pi$ と書け、その解は次のようになる。

$$V^\pi = (I - \gamma P_\pi)^{-1} r_\pi. \tag{14.23}$$

行列 $I - \gamma P_\pi$ は $\gamma < 1$ のとき正則である。γP_π があらゆるベクトルを上限ノルムにおいて縮小させるためである（演習問題 1 では、この逆行列を割引訪問回数として解釈する）。表形式であれば、連立一次方程式を 1 回解くだけで V^π が厳密に求まる。スイープはこれと同じ計算を 1 回の縮小ごとに進めるものであり、直接解くには大きすぎる問題へ拡張できる形式となっている。この直接解法を `TabularMDP` に追加しよう。

```
@d2l.add_to_class(d2l.TabularMDP)
def policy_value(self, pi):
    """V^pi = (I - gamma P_pi)^{-1} r_pi exactly, for pi[s, a] the
    probability of action a at state s (one-hot rows: deterministic).
    Needs gamma < 1 here: from_gym's terminal states loop on themselves."""
    P_pi = np.einsum('sa,sat->st', pi, self.P)
    r_pi = (pi * self.r).sum(axis=1)
    return np.linalg.solve(np.eye(self.num_states) - self.gamma * P_pi,
                           r_pi)

print(f'V(s0) for the uniformly random policy, solved exactly: '
      f'{mdp.policy_value(uniform)[0]:.4f}')
```

```
V(s0) for the uniformly random policy, solved exactly: 0.0078
```

どちらの方法でも 0.0078 が得られる。`einsum` は遷移カーネルの各行を行動について方策の確率で平均することで P_π を構成する。これらの配列上で直接解法を使うには $\gamma < 1$ が必要である。`from_gym` は各終端状態をどの行動でも自分自身へ戻る状態として格納するため、$\gamma = 1$ では行列 $I - P_\pi$ の対応する行がゼロ行となって特異になるからである。したがって、ゴール到達確率のような $\gamma = 1$ における量を求めるときはスイープを用いる。

14.2.3.2 方策反復法と一般化方策反復

方策評価と貪欲ステップによる方策改善（policy improvement）を組み合わせることで、Howard (1960) による別のアルゴリズムが導かれる。現在の方策を厳密に評価し、その価値に関して貪欲に行動し、これを繰り返すという手法である。貪欲ステップは、本節の冒頭で示した不等式 $\max_a Q^\pi(s,a) \geq V^\pi(s)$ をすべての状態に一斉に適用することに基づいている。

命題（方策改善）。 *π を任意の方策とし、π' を V^π に関して貪欲な方策、すなわち $\pi'(s) = \operatorname{argmax}_a \left[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s')\right]$ とする。このときすべての状態で $V^{\pi'}(s) \geq V^\pi(s)$ が成り立ち、もしすべての状態で等号が成立するならば両方策とも最適である。*

証明。 貪欲性より、1 ステップだけ π' に従いその後を π の価値で評価したものは、少なくとも 1 ステップ π に従った場合以上に良い。すなわち $\mathcal{T}^{\pi'} V^\pi = \mathcal{T} V^\pi \geq \mathcal{T}^\pi V^\pi = V^\pi$ である。作用素 $\mathcal{T}^{\pi'}$ は係数が確率であるため単調であり、すべての状態で $V \leq U$ ならば $\mathcal{T}^{\pi'} V \leq \mathcal{T}^{\pi'} U$ が成り立つ。これを繰り返し適用すると $V^\pi \leq \mathcal{T}^{\pi'} V^\pi \leq (\mathcal{T}^{\pi'})^2 V^\pi \leq \cdots$ が得られ、この列は $\mathcal{T}^{\pi'}$ の一意な不動点である $V^{\pi'}$ に収束する。もし $V^{\pi'} = V^\pi$ であれば最初の式から $\mathcal{T} V^\pi = V^\pi$ となるため、V^π は最適方程式を満たす。その唯一の解は V^* であるから、π は最適である。■

したがって、**方策反復法**（policy iteration）の各ラウンドは、いずれかの状態で価値を引き上げるか、すべての状態の価値を変えないかのどちらかであり、後者の場合はその時点で方策がすで

に最適となっている。価値は決して減少しないため、最適解に到達する前に同じ方策が再び現れることはなく、有限 MDP における決定論的な方策は有限個しかないため、方策反復法は有限回のラウンドで終了する (Puterman 1994)。終了判定には上の命題の等号成立条件、すなわち貪欲ステップがいかなる状態でも価値を改善しないという条件を用いる。代わりに貪欲方策が変化しなくなったかどうかを調べると、滑る湖のように同値が生じるケースで破綻することがある。マス 6 では**左**と**右**が同じ行動価値をもち、丸め誤差（rounding error）によってラウンドごとに `argmax` の結果が両者の間で入れ替わりうるからである。以下のループは一様方策から始め、各方策を `policy_value` で評価し、どの行動も現在の価値を丸め誤差以上には上回らなくなった時点で停止する。

```
pi, num_outer = uniform, 0
while True:
    V = mdp.policy_value(pi)                          # evaluate pi exactly
    Q = mdp.backup(V)
    num_outer += 1
    if (Q.max(axis=1) <= V + 1e-12).all():            # greedy gains nothing:
        break                                         # pi is optimal
    pi = np.eye(mdp.num_actions)[Q.argmax(axis=1)]    # improve
assert np.abs(V - V_star).max() < 1e-12
```

上のアサートは、方策反復法と価値反復法が小数点以下 12 桁まで V^* で一致することを確かめている。次のコードセルでは、それぞれが要した計算量を数える。1 回のバックアップは 1 つの状態行動対に対する角括弧項の 1 回の評価にあたるため、価値反復法の 1 スイープには $|\mathcal{S}||\mathcal{A}| = 64$ 回のバックアップというコストがかかる。

```
ok = [(mdp.backup(V_k).argmax(axis=1) == pi_star).all() for V_k in history]
assert all(ok[np.argmax(ok):])     # once optimal, the greedy policy stays so
first = mdp.backup(mdp.policy_value(uniform)).argmax(axis=1)
per_sweep = mdp.num_states * mdp.num_actions          # backups in one sweep
print(f'policy iteration: {num_outer} exact solves; the first greedy step '
      f'gives pi*: {(first == pi_star).all()}')
print(f'value iteration: certified at sweep {k_cert} '
      f'({k_cert * per_sweep:,} backups)')
print(f'its greedy policy already equals pi* from sweep {np.argmax(ok) + 1} '
      f'on ({(np.argmax(ok) + 1) * per_sweep:,} backups)')
```

```
policy iteration: 2 exact solves; the first greedy step gives pi*: True
value iteration: certified at sweep 164 (10,496 backups)
its greedy policy already equals pi* from sweep 14 on (896 backups)
```

方策反復法が要した厳密解法は 2 回だけであった。一様方策の価値に対する貪欲方策はすでに π^* となっており、2 回目のラウンドは貪欲ステップで何も改善しないことを確認しているにすぎない。一方、価値反復法は価値の精度保証を得るまでに 10,496 回のバックアップを要したが、その貪欲方策は 14 スイープ（896 回のバックアップ）の時点ですでに π^* に一致していた。方策はどの行動価値が最大かだけに依存するため、価値そのものが収束するよりずっと早く確定しうるのである。方策反復法の各ラウンドは未知数 16 個の連立一次方程式を解くため、この問題では低コストで済むが、状態数が多い問題では直接解法にかかる 3 乗オーダーのコストゆえにスイープの方が好まれる（演習問題 8 で両者のコストを比較する）。

評価と改善は、それぞれ完了するまで交互に行う必要はない。価値を V^π に近づけるステップと、方策を貪欲方策に近づけるステップを織り交ぜるあらゆる枠組みを総称して**一般化方策反復**（generalized policy iteration）と呼ぶ (Sutton and Barto 2018)。方策反復法は各評価を完了さ

せてから改善を行い、価値反復法は評価を 1 スイープ行っただけで改善に移る。経験から学習する手法は、モデルの代わりにサンプリングされた遷移から計算した、同じ 2 種類のより小さくノイズを含んだステップを進める。Figure 14.9 にそのような 4 つの枠組みを示す。

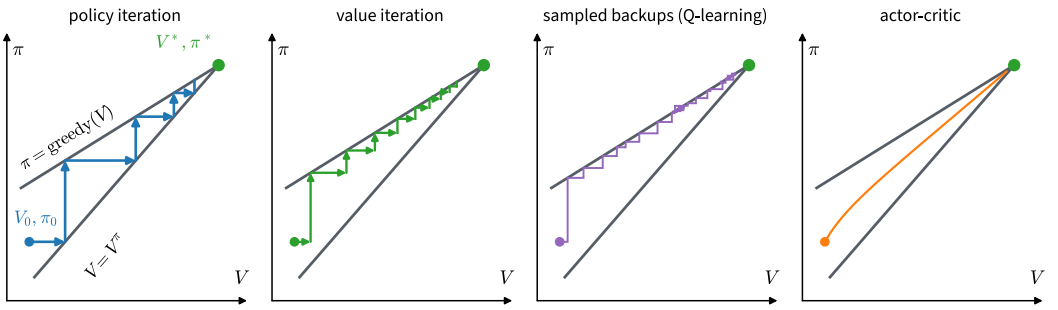


Figure 14.9: 一般化方策反復。直線 $V = V^\pi$ は方策に対して厳密な価値推定値の集合を、直線 $\pi = \text{greedy}(V)$ は価値推定値に対して貪欲な方策の集合を表し、両者が同時に成り立つのは (V^*, π^*) のみである。同じ初期点から出発して、方策反復法は改善のたびに最初の直線に到達するまで評価しきり、価値反復法はそこへ向けて 1 スイープだけ進み、標本バックアップは短くノイズを含んだステップをとり、方策とその価値推定値を同時に更新する手法は両方の座標を一度に少しずつ動かす。各軸は価値関数と方策の空間を表す模式図である。

14.2.4 滑る氷の上での最適方策

価値を見れば、π^* が最良の方策であることがわかる。では、明白な代替案である滑らない氷での最短経路と比べてどれほど優れているのか、またなぜ 4 つのマスでゴールから遠ざかる向きを指すのだろうか。この 2 つの問いにモデルから厳密に答え、さらにシミュレーションによってその結果を確認しよう。

14.2.4.1 モンテカルロ評価

モデルを使わない検証方法として、シミュレータ内で方策を実行し、得られた収益の平均をとるやり方がある。そのために、方策を観測（observation）から行動への関数 `policy(obs, rng)` として表す。方策が確率的であれば乱数生成器 `rng` から行動をサンプリングする。関数 `evaluate` は、指定したエピソード数にわたる平均割引収益を返す。

```
def evaluate(env, policy, num_episodes, gamma=1.0, rng=None):
    """Mean discounted return of an acting policy over Monte Carlo episodes.

    A policy is any function `policy(obs, rng) -> action`; a deterministic
    policy ignores `rng`, a stochastic one draws from it."""
    total = 0.0
    for _ in range(num_episodes):
        obs, done, discount = env.reset()[0], False, 1.0
        while not done:
            obs, reward, terminated, truncated, _ = env.step(policy(obs, rng))
            done = terminated or truncated
            total, discount = total + discount * reward, discount * gamma
    return total / num_episodes
```

変数 discount は現在のステップに対する γ^t を保持し、エピソードは terminated または truncated のいずれかで終了する。デフォルトの $\gamma = 1$ と、ゴールに入ったときにのみ与えられる湖の報酬のもとでは、平均収益はシミュレータの 100 ステップ制限内にゴールへ到達したエピソードの割合に一致する。

14.2.4.2 成功率

同じ確率はモデルからも計算できる。$\gamma = 1$ において、k 回のスイープによる方策評価（初期値 $V_0 = 0$）は、最初の k ステップの期待報酬を足し合わせる。報酬はゴールに入ったときにしか得られないため、k 番目の反復値の s_0 における成分は k ステップ以内にゴールへ到達する確率となり、$k = 100$ とすればシミュレータの制限と一致する。$\gamma = 1$ のモデルを lake1 として構築し、π^*、shortest、および一様方策のそれぞれについて到達確率を計算し、2,000 エピソードのモンテカルロ推定を並べて出力しよう。

```
lake1 = d2l.TabularMDP.from_gym(env, 1.0)    # gamma = 1: V(s0) = P(goal)
rng = np.random.default_rng(0)
env.reset(seed=0)
for name, p in [('slip-aware optimum', np.eye(4)[pi_star]),
                ('shortest-path policy', np.eye(4)[shortest]),
                ('uniformly random policy', uniform)]:
    exact = policy_evaluation(lake1, p, 100)[-1][0]   # within 100 steps
    mc = evaluate(env, lambda s, rng: int(rng.choice(4, p=p[s])), 2000,
                  rng=rng)
    print(f'{name:>23}: goal within 100 steps {exact:.3f} exactly, '
          f'{mc:.3f} in 2000 episodes')
```

```
     slip-aware optimum: goal within 100 steps 0.730 exactly, 0.736 in 2000 episodes
   shortest-path policy: goal within 100 steps 0.045 exactly, 0.046 in 2000 episodes
uniformly random policy: goal within 100 steps 0.014 exactly, 0.010 in 2000 episodes
```

最適方策（optimal policy）が 100 ステップ以内にゴールへ到達する確率は 0.730 であり、最短経路方策ではその約 1/16 にあたる 0.045、ランダムな歩行者では 0.014 となる。いずれのモンテカルロ推定も、厳密な値との差はサンプリングノイズで説明できる範囲に収まっている。2,000 エピソードにおける標準誤差（standard error）は、0.73 付近で約 1 パーセントポイント、0.014 付近で約 0.25 ポイントだからである。なお、制限時間によって π^* はわずかに不利を受けている。慎重な指示ゆえにエピソードが長引き、100 ステップを超えればゴールに到達できたはずの一部のエピソードが途中で打ち切られるためである。

14.2.4.3 ゴールに背を向ける指示

なぜ最短経路はこれほどひどく失敗するのだろうか。エージェントが行動する 11 個の氷のマスを 1 つずつ調べ、2 つの点を確認しよう。すなわち、π^* の指示がどちらを向いているか、そして各方策の指示によって次のステップでエージェントが穴に落ちる確率がどれほどあるかである。向きは滑らない氷での価値から読み取る。滑らない氷における遷移先のマスがより大きな価値 V_calm をもてばゴールに向かう移動、遷移先が元のマス自身であれば壁に向かう移動、それ以外はゴールから遠ざかる移動とする（穴もゴールも滑らない氷での価値が 0 であるため、そこへ入る移動も遠ざかる移動に分類される）。配列 risk[s, a] は 1 ステップで穴に落ちる確率 $\sum_{s' \text{ a hole}} P(s' \mid s, a)$ を保持する。

```
    desc = env.unwrapped.desc.flatten()
    hole, end = desc == b'H', np.isin(desc, [b'H', b'G'])
    risk = mdp.P[:, :, hole].sum(axis=-1)       # risk[s, a] = P(hole next | s, a)
    move = calm.P[np.arange(16), pi_star] @ V_calm - V_calm   # on calm ice
    for s in np.flatnonzero(~end):
        kind = ('toward the goal' if move[s] > 1e-9 else
                'into a wall' if move[s] > -1e-9 else 'away from the goal')
        print(f's={s:>2}: pi* {ARROWS[pi_star[s]]} {kind:<18}  hole risk: pi* '
              f'{risk[s, pi_star[s]]:.2f}, shortest {risk[s, shortest[s]]:.2f}, '
              f'least {risk[s].min():.2f}')
```

```
s= 0: pi* < into a wall         hole risk: pi* 0.00, shortest 0.00, least 0.00
s= 1: pi* ^ into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 2: pi* < away from the goal  hole risk: pi* 0.00, shortest 0.00, least 0.00
s= 3: pi* ^ into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 4: pi* < into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 6: pi* < away from the goal  hole risk: pi* 0.33, shortest 0.67, least 0.33
s= 8: pi* ^ away from the goal  hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 9: pi* v toward the goal     hole risk: pi* 0.00, shortest 0.00, least 0.00
s=10: pi* < away from the goal  hole risk: pi* 0.00, shortest 0.33, least 0.00
s=13: pi* > toward the goal     hole risk: pi* 0.00, shortest 0.00, least 0.00
s=14: pi* v into a wall         hole risk: pi* 0.00, shortest 0.00, least 0.00
```

最適方策がゴールへ向かって進むのは、マス 9 とマス 13 の 2 マスだけである。マス 0、1、3、4、14 の 5 マスでは、壁に向かって進むよう指示する。このような指示を出すと、本来の移動はその場にとどまる結果となり、実際に移動が生じるのは 2 方向の滑りだけになる。たとえばマス 4 では、**左**を指示するとエージェントは壁に沿って上下に滑るだけで、右にあるマス 5 の穴へ入ることは決してない。マス 14 では、**下**も**右**も 3 回中 1 回はゴールに到達し、3 回中 1 回はマス 14 にとどまる。3 つ目の結果は、**下**ならマス 13、**右**ならマス 10 への移動であり、マス 10 の方が価値が低い（先ほどのグリッドで 0.40 対 0.51）。

マス 2、6、8、10 の 4 マスでは、最適方策はゴールから遠ざかる向きを指す。マス 8 とマス 10 では、ゴールへ向かう指示はいずれも穴（マス 8 の下にある 12 や、マス 10 の隣にある 11）へ落ちる可能性がある。これに対し、マス 8 での**上**とマス 10 での**左**は穴に落ちることがなく、しかもマス 10 での**左**は 3 回中 1 回の滑りでエージェントを下方のマス 14 へと運ぶ。マス 6 では両脇に穴があるため、どの指示にも穴に落ちる危険が伴う。**左**と**右**の危険度は 1/3、**上**と**下**の危険度は 2/3 であり、π^* は最も危険の少ない同値の 2 つの指示の一方を選ぶ。`argmax` は**左**を選ぶが（**右**ならゴールへ向かう向きになる）、これは 3 回中 1 回そのまま穴 5 へ入る移動をあえて指示している。**上**や**下**では 2 方向の滑りが両方とも穴に入るのに対し、**左**なら 2 方向の滑りがどちらも安全なマスに着地するからである。マス 2 ではどの指示にも穴の危険はないが、**左**が選ばれるのは、最上段で最も価値が低くどの経路も結局マス 2 を通り直すことになる角のマス 3 へ滑る可能性がないためである。

11 個すべてのマスにわたって、最適な指示の危険度は選べる指示の中で最小となっており、マス 6 以外のすべてのマスで 0 である。一方、最短経路方策はマス 1、3、4、8、10 で 1/3 の危険を冒し、マス 6 では 2/3 の危険を冒す。ただし、これらの 1 ステップの危険度だけではエピソードがどこで終わるかはまだわからない。各マスが何回訪問されるかにも依存するからである。決定論的な方策 π に対し、P_π を π のもとでの遷移カーネルのうち終端状態の行を 0 に置き換えてそこで過程が停止するようにしたものとする。ここで調べている両方策はいずれも確率 1 でエピソードを終了させるため $I - P_\pi$ は正則であり、$s_0 = 0$ から出発したときの各状態への期待訪問回数は $(I - P_\pi)^{-1}$ の s_0 に対応する行をなす。関数 `falls` は転置した連立一次方程式を解くことでこれを求める。各マスでの期待訪問回数にそのマスでの指示の穴への危険度を掛けると、そのマス

から穴に落ちてエピソードが終わる確率が得られる。

```
def falls(mdp, pi, risk, end):
    """P(the episode falls into a hole from s), for each s, without a time
    limit: expected visits to s before the end, times the hole risk at s."""
    S, I = np.arange(mdp.num_states), np.eye(mdp.num_states)
    P_pi = mdp.P[S, pi] * ~end[:, None]               # stop at the end
    visits = np.linalg.solve(I - P_pi.T, I[0])        # starting from s0 = 0
    return visits * risk[S, pi] * ~end

for name, p in [('pi*', pi_star), ('shortest', shortest)]:
    f = falls(mdp, p, risk, end)
    print(f'{name:>8}: P(falls into a hole) {f.sum():.2f}, from ' +
          ', '.join(f'{s}: {f[s]:.2f}' for s in np.argsort(-f) if f[s] > 0))
```

```
    pi*: P(falls into a hole) 0.22, from 6: 0.22
shortest: P(falls into a hole) 0.96, from 4: 0.31, 1: 0.31, 8: 0.13, 6: 0.08, 3: 0.06, 10: 0.05
```

時間制限がなければ、最適方策が穴に落ちる確率は 0.22（そのすべてがマス 6 で生じる）であり、残りの 78% のエピソードではゴールに到達する（100 ステップの制限によってこれが先ほどの 0.730 まで下がる）。一方、最短経路方策はエピソードの 96% で穴に落ちる。落下の約 1/3 は本来の経路上のマス 4 から生じるが、さらに別の 1/3 は経路上にまったくないマス 1 から生じる。マス 0 での最初の指示である**下**が 3 回中 1 回の割合でエージェントを右のマス 1 へ滑らせ、そこで `shortest` が指示する**右**が 3 回中 1 回の割合で下のマス 5 にある穴へと滑り落ちるのである。残りの落下はマス 8、6、3、10 から生じる。最短経路方策は、本来の経路そのものよりも、滑って飛ばされた先のマスでより多く失敗しており、対する最適方策はいずれかの指示で穴を避けられるすべてのマスにおいて確実に穴を避けている。

14.2.5 報酬シェイピングと最適方策

14.1 節では、報酬が疎な湖で学習器を助けるため、ゴールに近づく移動のたびに 0.3 のボーナス（マンハッタン距離 $\ell(s)$ で判定）を与える案を取り上げ、滑らない氷での計算によってそのボーナスが堂々巡りに報酬を与えてしまうことを示した。価値反復法と厳密な方策評価を手にした今、このボーナスが滑る氷の上での最良の方策に何をもたらすかを計算でき、さらに前節で提案したポテンシャル形式が最良の方策を変化させないことも証明できる。

14.2.5.1 接近ボーナス

次の状態について平均をとると、このボーナスは期待報酬に「ゴールへ近づく確率の 0.3 倍」を加える。

$$\tilde{r}(s,a) = r(s,a) + 0.3 \sum_{s'} P(s' \mid s,a)\, \mathbf{1}\big(\ell(s') < \ell(s)\big).$$

次のコードセルでは、ボーナス 0.3、0.1、0 に対してこの報酬を構成し、報酬シェイピング（reward shaping）を施した各 MDP の最適方策を価値反復法で求めたうえで、その方策を元の報酬のもとで `policy_value` により評価する。ボーナス 0 の行が元の問題に対応する。

```
row, col = np.divmod(np.arange(16), 4)
dist = abs(row - 3) + abs(col - 3)     # l(s): grid steps to the goal at (3, 3)
```

```
closer = dist < dist[:, None, None]    # closer[s, :, s'] = 1 iff l(s') < l(s)

for bonus in [0.3, 0.1, 0.0]:
    shaped = d2l.TabularMDP(mdp.P, mdp.r + bonus * (mdp.P * closer).sum(-1),
                            gamma)
    V_shaped = value_iteration(shaped, num_iters=500)[-1]
    pi_shaped = shaped.backup(V_shaped).argmax(axis=1)
    V_true = mdp.policy_value(np.eye(mdp.num_actions)[pi_shaped])
    print(f'bonus {bonus}: at s=14 goes {ARROWS[pi_shaped[14]]}, shaped V(0) '
          f'{V_shaped[0]:.2f}, V(13) {V_shaped[13]:.2f}, V(14) '
          f'{V_shaped[14]:.2f}; true V(0) {V_true[0]:.3f}')
```

```
bonus 0.3: at s=14 goes <, shaped V(0) 2.15, V(13) 1.71, V(14) 1.52; true V(0) 0.000
bonus 0.1: at s=14 goes v, shaped V(0) 0.79, V(13) 0.87, V(14) 0.94; true V(0) 0.115
bonus 0.0: at s=14 goes v, shaped V(0) 0.18, V(13) 0.51, V(14) 0.72; true V(0) 0.180
```

ボーナスが 0.3 のとき、シェイピング後の最適方策はマス 14 で**左**を指示する。その遷移先はマス 13、10、14 のいずれかであり、決してゴールには入らない。シェイピング後の報酬のもとでの初期状態価値は 2.15 と、元の最適方策の 0.180 を大きく上回るが、決してゴールに到達しないため元の報酬のもとでの価値は 0 である。シェイピング後の価値を見れば理由がわかる。シェイピング後の報酬では、マス 13 の価値がマス 14 の価値を上回っている（1.71 対 1.52）。13 から 14 へ向かう移動でボーナスが得られ、しかも 14 からはいつでも 13 に戻って再びボーナスを稼げるからである。これは 14.1 節で見たループに具体的な価値が付いた姿にほかならない。マス 13 とマス 14 の間を往復すれば 1 周ごとにボーナスが得られ、ゴールに入って 1 度きり得られる報酬とボーナスの合計よりも価値が高くなるのである。ボーナスが 0.1 であればループの価値はゴールを下回り、シェイピング後の最適方策は再びゴールを目指すようになるが、その真の価値は 0.115 となって元の 0.180 には届かず、元とは異なる劣った方策になってしまう。一見もっともらしいボーナスが最適方策を完全に変えてしまい、ボーナスを小さくしてもなお方策は変わってしまうのである。

14.2.5.2 ポテンシャルに基づくシェイピング

14.1 節で代わりに提案したボーナスは、次のようなポテンシャルの差である。

$$\tilde{r}(s, a, s') = r(s, a) + \gamma\Phi(s') - \Phi(s), \tag{14.24}$$

ここで Φ は状態上の有界な関数である。ループを一周すると各項は割引率の分を除いて打ち消し合い、次の命題が示すように、どの方策の価値も同じ量だけ変化する。

命題（ポテンシャルに基づくシェイピング）。$0 \leq \gamma < 1$ とし、`TabularMDP` と同様に終端状態は吸収状態であるとし、$\tilde{V}^\pi$ と $\tilde{Q}^\pi$ をシェイピング後の報酬のもとでの価値とする。このとき、任意の方策 π、状態 s、行動 a に対して次が成り立つ。

$$\tilde{V}^\pi(s) = V^\pi(s) - \Phi(s), \qquad \tilde{Q}^\pi(s, a) = Q^\pi(s, a) - \Phi(s). \tag{14.25}$$

特に $\tilde{V}^* = V^* - \Phi$ および $\tilde{Q}^*(s, a) = Q^*(s, a) - \Phi(s)$ が成り立ち、シェイピング後の MDP と元の MDP は同じ最適方策をもつ (Ng et al. 1999)。

証明。任意の軌跡に沿って、最初の T ステップのシェイピング項は望遠鏡和となって相殺し合う。

$$\sum_{t=0}^{T-1} \gamma^t \big(\gamma \Phi(s_{t+1}) - \Phi(s_t)\big) = \gamma^T \Phi(s_T) - \Phi(s_0). \tag{14.26}$$

$T \to \infty$ のとき、Φ が有界かつ $\gamma < 1$ であるため第 1 項は 0 に収束する。したがって、あらゆる軌跡のシェイピング後の収益は元の収益から $\Phi(s_0)$ を引いたものになる。π のもとで s から出発する期待値をとれば第 1 の等式が得られ、(s, a) から出発する期待値をとれば第 2 の等式が得られる。各状態におけるシフト量 $\Phi(s)$ はすべての方策とすべての行動に対して同一であるため、方策間の順序も $Q^*(s, \cdot)$ を最大化する行動も変わらない。■

このようにポテンシャルはすべての価値を変化させるが、どの方策も同じだけ変化を受けるため、最適化が用いる唯一の情報である方策間の大小関係は変わらない。演習問題 7c では、湖の上で $\tilde{V}^* = V^* - \Phi$ を確かめる。Ng et al. (1999) はその逆も証明している。追加する項 $F(s, a, s')$ がポテンシャルの差でないならば、同じ状態集合と行動集合の上に、シェイピング後の MDP のいかなる最適方策も元の MDP では最適とならないような遷移カーネルと報酬が存在する。接近ボーナスはそのような項の一例であり、滑る湖はそのような MDP の一例である。

ポテンシャルが最適方策を変えないのだとすれば、なぜそれを使うのだろうか。厳密なプランニングにおいては使う理由がない。価値反復法はポテンシャルの有無にかかわらず同じ方策を見つけるからである。その目的は、サンプリングされた遷移しか観測できない学習器を助けることにある。$\Phi = -0.3\,\ell$ のようにゴールへ向かって高くなるポテンシャルを導入すると、ゴールの報酬が経路全体に分散されるため、学習器は最後の 1 手だけでなく毎回の移動で信号を受け取れるようになる。

上の証明では `TabularMDP` の 1 つの性質を利用した。すなわち、終端状態が吸収状態であるため軌跡が永遠に続き、極限で項 $\gamma^T \Phi(s_T)$ が消えるという性質である。これに対し、シミュレータは最初の終端状態に到達した時点でエピソードを終了させる。もしそれが T ステップ後に起きたとすれば、そのエピソードのシェイピング後の収益は元の収益に $\gamma^T \Phi(s_T) - \Phi(s_0)$ を加えたものになる。項 $\gamma^T \Phi(s_T)$ はエピソードがどの終端状態でいつ終わったかに依存するため、すべての終端状態で Φ が 0 でない限り、どの方策が最良かを変えてしまうおそれがある。これが 14.1 節で言葉によって述べた条件である。ポテンシャル $\Phi = -0.3\,\ell$ はゴールでは 0 だが、穴では負の値をとる。そのためシミュレータ内では、穴に落ちることに対して（しかも遅く落ちるより早く落ちる方に大きく）元の課題には含まれないペナルティを課すことになってしまう。

本節の内容はすべて遷移表 P を参照していた。すなわち、どのバックアップも厳密な確率を用いて次の状態についての平均を計算していた。これで、既知の有限 MDP におけるすべての状態とすべての指示の価値を計算し、精度保証付きで最適方策を求め、報酬の変更がそれにどう影響するかを調べられるようになった。しかし、学習するエージェントに遷移表が与えられることはめったにない。エージェントが観測できるのは、自身の行動が生み出した 1 回ずつの遷移だけである。14.3 節では、そのようなサンプルだけから価値と最適方策を学習できるかどうかを問う。

14.2.6 まとめ

動的計画法は、既知の MDP における最良の方策をその状態の価値から計算する。価値 $V^\pi(s)$ は s から π のもとで得られる期待割引収益であり、ベルマン方程式は各状態の価値を後続状態の価値と結びつける（固定した方策では行動についての平均により、最適な方策では最大値による）。$\gamma < 1$ のときベルマン作用素 $\mathcal{T}$ は γ 縮小写像であるため、価値反復法は任意の初期値から一意な V^* へと幾何級数的に収束し、スイープ間の変化量がその誤差を保証する。また方策反復法は、1 回の連立一次方程式の求解による厳密な評価と貪欲な改善とを交互に行う。滑る湖において最適方策が最短経路の約 16 倍の頻度でゴールに到達するのは、たとえゴールから遠ざかる向きであっても、すべてのマスで穴に落ちる危険が最小の指示を選ぶからである。ゴールへ近づくことに

対して支払われるボーナスは最適方策を変えうるが、ポテンシャルの差として与えられるボーナスが最適方策を変えることはない。

14.2.7 演習問題

1. [code] **連立一次方程式としての方策評価**。@eq-policy-value-solve より、$V^\pi = M_\pi r_\pi$（ただし $M_\pi = (I - \gamma P_\pi)^{-1}$）である。
   1. $I - \gamma P_\pi$ が $0 \le \gamma < 1$ に対して正則であり、$M_\pi = \sum_{t \ge 0} \gamma^t P_\pi^t$ となることを示せ。
   2. 成分 $M_\pi(s, s')$ を、s' への訪問回数（問題 4b（14.1 節）の終了する過程において s から出発して π に従うエージェントによる訪問）の観点から解釈せよ。これより M_π の各行の和が $1/(1-\gamma)$ となり、したがって $(1-\gamma) M_\pi(s, \cdot)$ が状態上の分布、すなわち π の（s から出発した）**割引状態訪問分布**になることを導け。さらに $V^\pi(s)$ を、$1/(1-\gamma)$ とこの分布のもとでの r_π の期待値との積として表せ。
   3. 滑る湖における方策 `uniform` に対して M_π を計算し、$M_\pi r_\pi$ が初期状態の価値として 0.0078（`policy_evaluation` の出力）を再現することを確かめよ。この方策のエピソードは平均して 8 ステップ未満で終わるにもかかわらず、初期状態の行の和は 20 になる。残りの重みはどこに集中しているか。
2. **停止の精度保証**。精度保証 Equation 14.21 は、観測できない誤差 $\|V_k - V^*\|_\infty$ を観測可能な変化量 $\|V_k - V_{k-1}\|_\infty$ によって上から抑える。
   1. 14.2.2.3 節の縮小写像の命題から、望遠鏡和の裾 $\sum_{j \ge k} \|V_{j+1} - V_j\|_\infty$ を上界評価することでこの式を導出せよ。与えられた許容誤差未満の誤差を保証するには、スイープ間の変化量がどれだけ小さくなければならないか。
   2. 価値反復法を始める前にスイープ数を確定させる事前上界を、γ、許容誤差、および $\|V_1 - V_0\|_\infty$ のみを用いて導出せよ。滑る湖において $V_0 = 0$、許容誤差 10^{-6} としてその上界を計算し、この湖に対する価値反復法で精度保証が要した 164 回のスイープ（14.2.2 節）と比較せよ。
3. **割引なしの湖**。14.2.2.3 節の縮小写像の命題は $\gamma < 1$ を必要とするが、FrozenLake における価値反復法は $\gamma = 1$ でも収束する。FrozenLake では穴とゴールは報酬 0 の吸収状態であり、唯一の非ゼロ報酬である 1 はゴールへ入るときに得られる。
   1. $\gamma = 1$ として命題の証明をたどれ。それでもなお何が示されるか。続いて、Equation 14.18 の作用素 $\mathcal{T}$ が $\gamma = 1$ で不動点をもつならば、必ず無限個の不動点をもつことを示せ。
   2. $V_0 = 0$ から始める $\gamma = 1$ の価値反復法が、1 で上から有界な非減少列 $V_0 \le V_1 \le V_2 \le \cdots$ を生成し、したがって収束することを示せ。$V_k(s)$ は何を表しているか。
   3. 行動が 1 つだけの 2 状態 MDP であって、V_k が $\gamma = 1$ で限りなく増大するものを構成せよ。この MDP には、小問 b で用いた FrozenLake のどの性質が欠けているか。
4. [code] **インプレースなスイープ**。●`value_iteration` は V_{k+1} の全成分を V_k から計算する。これに対し、**インプレース**なスイープは固定した順序で各状態を巡回し、$V(s)$ をその場で直ちに上書きするため、同じスイープ内で後から訪れる状態はすでに新しい価値を利用する。
   1. 状態をどの順序で巡回しても、インプレースなスイープは上限ノルムにおける γ 縮小写像であり、Equation 14.18 の $\mathcal{T}$ と同じ不動点をもつことを示せ。さらに、その反復列に対しても精度保証 Equation 14.21 がそのまま成り立つことを導け。
   2. インプレースなスイープを実装し、精度保証が 10^{-6} に達するまでに要するスイープ数を、状態の昇順と降順のそれぞれについて、滑る氷と滑らない氷の 4×4 マップおよび 8×8 マップ上で数えよ。得られたスイープ数を `value_iteration` の場合と比較せよ。
   3. なぜ滑る氷よりも滑らない氷の方が順序の影響がはるかに大きいのか。滑らない氷でのスイープ数を Figure 14.7 を用いて説明せよ。

Sutton and Barto『Reinforcement Learning: An Introduction』（第 2 版）の 4.1 節お

よび 4.5 節より改変。

5. [code] **実測される縮小率。●**14.2.2 節では、価値反復法の誤差は 1 スイープあたり約 0.92 倍で縮小し、理論保証の $\gamma = 0.95$ よりも速く減少する。スイープ 14 以降は、すべての反復値に対する貪欲方策が `pi_star` に一致する。
   1. それ以降、誤差が線形漸化式 $V_{k+1} - V^* = \gamma P_{\pi^*}(V_k - V^*)$（ただし P_{π^*} は Equation 14.23 における π^* のもとでの遷移カーネル）に従い、穴とゴールでは誤差が 0 であることを示せ。これより漸近的な縮小率がスペクトル半径 ρ（非終端状態に制限した γP_{π^*} のスペクトル半径）に一致することを導き、ρ を計算せよ。
   2. ρ を $\gamma = 0.999$ において（その割引率での最適方策を用いて）計算し、価値反復法が誤差 10^{-6} に達するまでに要するスイープ数を予測せよ。理論保証 $\gamma^k \|V_0 - V^*\|_\infty$ では $13{,}000$ 回を超えるスイープが必要となる。`value_iteration` を用いて予測を確かめよ。FrozenLake のどの性質によって ρ は γ より小さくなるのか。
   3. 14.2.2 節の規則 $\|V_k - V_{k-1}\|_\infty \le 10^{-6}$ はスイープ 128 で停止した。そこでの真の誤差はスイープ間の変化量の 11.7 倍であり、精度保証の係数 $\gamma/(1-\gamma) = 19$ を大きく下回っていた。公比 ρ で幾何級数的に縮小する誤差が係数 $\rho/(1-\rho)$ を与えることを示せ。なぜ精度保証 Equation 14.21 は ρ を γ の代わりに用いることができないのか。
6. [code] **改善量の定量化。●**14.2.3.2 節の方策改善の命題は、貪欲ステップが価値を下げないことを述べている。性能差補題 (Kakade and Langford 2002) の行列表現である次の恒等式は、その改善量を測るものである。方策 π' に対し、$\mathcal{T}^{\pi'}$ を Equation 14.22 における π' の方策評価作用素とし、$M_{\pi'}$ を問題 1 の行列とする。
   1. 任意の 2 つの方策 π と π' に対して
   $$V^{\pi'} - V^{\pi} = M_{\pi'}(\mathcal{T}^{\pi'} V^{\pi} - V^{\pi}), \tag{14.27}$$
   および $(\mathcal{T}^{\pi'} V^{\pi} - V^{\pi})(s) = \sum_a \pi'(a \mid s)\,[Q^{\pi}(s,a) - V^{\pi}(s)]$ が成り立つことを示せ。
   2. この恒等式を用いて方策改善の命題を別証明せよ。さらに、状態 s が $Q^{\pi}(s, \pi'(s)) > V^{\pi}(s)$ を満たすならば、その状態において真に改善することを示せ。
   3. π^* を π の役割に置いてこの恒等式を適用し、$V^{\pi'} \le V^*$ が確率的方策を含むあらゆる定常方策 π' に対して成り立つことを示せ。
   4. π = `uniform` および $\pi' = \pi^*$ として、初期状態においてこの恒等式を数値的に確かめよ。
7. [code] **ポテンシャルに基づくシェイピング。●**14.2.5 節の命題は、`TabularMDP` のように終端状態が吸収状態であるときにはあらゆるポテンシャルに対して成り立つが、シミュレータのように終端状態でエピソードが停止するときには終端状態で 0 になるポテンシャルに対してのみ成り立つ。
   1. 吸収状態も含め、`mdp.r` のすべての要素に定数 c を加えよ。あらゆる軌跡の割引収益が同じ量だけ変化することを示し、その変化量を求めよ。どの定数ポテンシャル Φ が $\gamma\Phi(s') - \Phi(s) = c$ を満たすか。
   2. 次に、エピソードが進行している間だけ定数を課すこととし、非終端状態においてのみ c を $r(s,a)$ に加えよ。$c \in \{-0.02, -0.01, 0.01, 0.05\}$ のそれぞれについて `value_iteration` で最適方策を計算し、状態 0 における行動、`mdp.policy_value` による元の報酬のもとでの価値、および 14.2.4 節と同様に計算した 100 ステップ以内のゴール到達確率を求めよ。c の各符号が及ぼす影響を、定数ポテンシャルと終端条件を用いて説明せよ。
   3. `shaped` の接近ボーナスをポテンシャル $\Phi(s) = -0.3\,\ell(s)$（ただし ℓ はマンハッタン距離 `dist`）に置き換え、シェイピング後の報酬を次の状態について平均すると、すべての状態において次式が得られる。

$$\tilde{r}(s,a) = r(s,a) + \sum_{s'} P(s' \mid s,a)\,[\gamma\Phi(s') - \Phi(s)]. \tag{14.28}$$

value_iteration が返す価値 $\tilde{V}$ がすべての状態で $\tilde{V}(s) = V^*(s) - \Phi(s)$ を満たすこと、およびシェイピング後の最適方策が元の報酬のもとで pi_star と同じ価値をもつことを確かめよ（**左**と**右**が同値となるマス 6 では 2 つの方策は異なりうる）。続いて、終端状態で停止するシミュレータと同様に終端状態でのシェイピング項を取り除き、得られた最適方策の元の報酬のもとでの価値を再計算せよ。最後に、終端状態で Φ を 0 に設定して同じ計算を繰り返せ。

Sutton and Barto『Reinforcement Learning: An Introduction』の演習問題 3.15 および 3.16 より改変。

8. [extended] **バックアップ予算。●●**14.2.3.2 節の方策反復法が要したのはわずか 2 ラウンド（各ラウンドで 1 回の連立一次方程式の厳密解法）であったのに対し、価値反復法は価値の精度保証を得るまでに 164 回のスイープを要した。状態数が多い問題では直接解法は高コストすぎるため、代わりにスイープによって評価を行う。コストを**バックアップ**の回数で測ろう。1 回のバックアップは 1 つの状態行動対に対する Equation 14.19 の角括弧項の 1 回の評価であるため、価値反復法の 1 スイープには $|\mathcal{S}||\mathcal{A}|$ 回のバックアップ、決定論的な方策の評価の 1 スイープには $|\mathcal{S}|$ 回のバックアップというコストがかかる。**修正方策反復法** (Puterman 1994) は、これら 2 つのアルゴリズムの間を補間する手法である。初期値を $U_0 = 0$ とし、ラウンド k ではまず価値反復法のスイープを 1 回行い（同時に貪欲方策 π_k も U_k から得られる）、続いて現在の表から出発して m 回のスイープ（Equation 14.22 による π_k の評価）を行う。Equation 14.22 の作用素 $\mathcal{T}^{\pi_k}$ を用いれば、1 ラウンドは $U_{k+1} = (\mathcal{T}^{\pi_k})^{m+1} U_k$ と書ける。$m = 0$ では価値反復法に一致し、$m \to \infty$ の極限では方策反復法になる。
   1. $V_k = \mathcal{T}^k U_0$ を価値反復法の反復列とする。$\mathcal{T}U_0 \geq U_0$ ならば $V_k \leq U_k \leq U_{k+1} \leq V^*$ がすべての k とすべての m に対して成り立つことを示し、この前提が FrozenLake で成り立つことを確かめよ。これより、同じラウンド数の後では、修正方策反復法が V^* から価値反復法より遠くなることは決してないことを導け。
   2. 修正方策反復法を実装し、各スイープ後の誤差 $\|U - V^*\|_\infty$（現在の表 U の誤差）を累積バックアップ数に対してプロットせよ。ただし $m \in \{0, 1, 5, 20, 100\}$ とし、滑る氷の 4×4 マップおよび 8×8 マップ上で $\gamma = 0.95$ とする。
   3. 大きい m に対する曲線の形状を説明せよ。なぜ貪欲ステップと貪欲ステップの間で誤差の減少が停滞するのか。
   4. $\gamma = 0.99$ として小問 b を繰り返せ。最小のバックアップ数で到達させる m（目標誤差 10^{-6} に対して）は、$\gamma \to 1$ に伴ってどのように変化するか。

14.3 時間差分学習

前節では、遷移テーブルを読み取ることで滑る湖を渡る最良の方法を見つけた。すべてのセルとすべてのコマンドについて、エージェントが滑って行き着く可能性のある 3 つの場所すべてにわたって和をとったのである。しかし、現実の湖に降り立ったエージェントはそのようなテーブルを持たない。行動を起こし、どこに行き着いたかを確かめ、得られた報酬を受け取ることしかできない。各ステップで目にするのは環境が選んだ 1 つの遷移先だけであり、起こりうる遷移先とその確率の一覧が示されることは決してない。このような 1 回ずつの抽出結果からでも、エージェントは各コマンドの良し悪しを学習し、そこから最良の方策（policy）を導き出せるだろうか。

その答えを、価値の予測については 1988 年に Richard Sutton が、行動の選択については 1989 年に Chris Watkins が見出した (Sutton 1988; Watkins 1989; Watkins and Dayan 1992)。エージェントが観測した 1 つの遷移先を計算できない総和の 1 サンプル推定値とみなし、テーブルの

要素をその推定値に向けて少しだけ動かすのである。訪問を重ねるにつれてこの更新はサンプルを平均化し、その平均が動的計画法（dynamic programming）のバックアップと一致する。こうして得られる手法である **Q 学習**（Q-learning）は、環境のモデルを必要としない。テーブルをニューラルネットワークに置き換えれば、次章で扱う深層 Q 学習になる。

サンプルからの学習には 2 つの問いが生じる。第 1 に、サンプルにはノイズが含まれるため訪問が重なるにつれてステップを縮めなければならないが、どのくらいの速さで縮めるべきか。第 2 に、エージェントが一度も試さないコマンドの要素は決して更新されないため、テーブルでの評価が低いコマンドも試し続けなければならない。本節では 1 つ目の問いに答え、2 つ目の問いには小さな割合でランダムに行動するという最も単純な規則で対処する。その規則が伴うコストについては 14.4 節で詳しく調べる。最後に本節の締めくくりとして、同じ遷移データをもう 1 つの方法で利用し、遷移テーブルそのものを推定してその推定モデル上で計画を立てる。

```
import gymnasium as gym
```

14.3.1 サンプルによるバックアップ

価値反復法（value iteration）が解いた方程式から出発し、そのどの部分にモデルが必要なのかを確かめよう。

14.3.1.1 ベルマン作用素から 1 サンプルへ

14.2 節の価値反復法は、ベルマン最適バックアップという 1 つの操作を繰り返すものであった。状態価値（state value）ではなく行動価値（action value）に対して書き下すと、ベルマン作用素（Bellman operator）$\mathcal{T}$ はテーブル Q を次のテーブルへ写す。

$$(\mathcal{T}Q)(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a'), \qquad \text{for all } s \in \mathcal{S},\ a \in \mathcal{A}. \tag{14.29}$$

$\mathcal{T}Q$ の各要素は、即時報酬の期待値に、エージェントが行き着いた先から Q に関して貪欲に行動したときの割引価値を加えたものである。その不動点（fixed point）は Q^* であり、どのテーブルから始めても反復適用によって Q^* へ収束する。これは $\mathcal{T}$ が 2 つのテーブル間の最大差を係数 γ 倍に縮めるためであり、14.2 節の縮小議論を行動価値に適用した結果である。

モデルがバックアップに現れるのは、次状態 s' にわたる平均という 1 か所だけである。平均はサンプルから推定できる。エージェントが状態 s にいて a を指示し、報酬 r を受け取って s' に到達したとする。このとき $r + \gamma \max_{a'} Q(s',a')$ という値は確率的に変動する。湖の環境では、報酬もまたエージェントがどこに行き着くかに依存する。次状態と報酬にわたって平均をとれば、この値は $(\mathcal{T}Q)(s,a)$ に等しい。したがって、1 回の遷移はバックアップの 1 要素に対する不偏な 1 サンプル推定値となる。

モデルフリー制御が学習対象を Q とし、V としない理由はここにある。状態価値のバックアップ $V(s) \leftarrow \max_a [r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s')]$ では、最大値演算が s' にわたる平均の外側にある。1 回の遷移からわかるのは 1 つの行動がもたらした結果だけであり、他の行動の角括弧内の値については何もわからないため、それらの最大値を推定することはできない。一方、Q の形式では行動ごとに独立した要素があり、平均はその要素の内側に収まっているため、最大値演算はテーブルがすでに保持している数値に対してのみ作用する。さらにこの形式のおかげで、貪欲方策（greedy policy）$\operatorname{argmax}_a Q(s,a)$ もやはりモデルなしでテーブルの 1 行を参照するだけで求まる。

Q 学習は、訪問した要素を**ターゲット**と呼ばれるこのサンプルに向けて割合 α だけ動かす。

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a')\big). \tag{14.30}$$

ステップサイズ $\alpha \in (0,1]$ はどれだけ動かすかを表す。$\alpha = 1$ なら要素をターゲットでそのまま置き換え、小さな α なら多数のターゲットを平均化する。ターゲットが次状態におけるテーブル自身の推定値から作られるため、この更新は**ブートストラップ**すると呼ばれる。式を整理すると、変化量は次の 1 つの数値に比例する。

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a), \qquad Q(s,a) \leftarrow Q(s,a) + \alpha\,\delta. \tag{14.31}$$

この数値 δ は**時間差分誤差**（temporal-difference error）である (Sutton 1988)。すなわち、1 ステップ後に判明したその対の価値の見積もりから、ステップ前にテーブルが示していた値を引いたものである。Figure 14.6 のパネル (d) にこの置き換えを示す。厳密なバックアップはすべての次状態にわたって和をとるが、サンプルによるバックアップは観測された単一の遷移をたどり、次状態での最大値演算はそのまま残す。

ブートストラップするかどうかは設計上の選択である。**モンテカルロ**推定はエピソードが終わるまで待ち、実際に得られた収益（return）を $Q(s_t,a_t)$ のターゲットとして用いる。このターゲットはエピソードを生成した方策の価値に対して不偏であるが、それ以降のすべての遷移に伴うランダム性を抱え込む。これに対し、**時間差分**（temporal-difference、TD）ターゲットは 1 ステップ先だけを見て、残りはテーブルの値を信用する。テーブルが不正確なうちはバイアスを持つものの、分散は通常はるかに小さい (Sutton 1988; Sutton and Barto 2018)。Q 学習はこの 1 ステップのターゲットに最大値演算を加えることで、データを収集する方策の価値ではなく Q^{*} を推定する。

14.3.1.2 1 遷移の手計算

滑る湖において、すべて 0 のテーブルから出発し、$\gamma = 0.95$、そして計算しやすいように $\alpha = 0.5$ とする。エージェントが初めてゴールに到達するまでは、ゴール以外の報酬がすべて 0 であり、ブートストラップ元となる要素もすべて 0 であるため、どのターゲットも 0 になる。最初の成功がセル 14 から生じたとしよう。ここでは**右**のコマンドを出すと、ゴールであるセル 15 へ確率 $1/3$ で滑り込む。ゴールは終端状態（terminal state）なのでターゲットは報酬のみとなり、更新は次のようになる。

$$Q(14,\text{right}) \leftarrow 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5.$$

テーブルの他の要素は何も変わらない。その後、エージェントがセル 13 に立ち、**右**を指示して 14 へ滑り込んだとする。このときのターゲットは $0 + 0.95 \cdot \max_{a'} Q(14,a') = 0.95 \cdot 0.5 = 0.475$ となるため、$Q(13,\text{right}) \leftarrow 0.5 \cdot 0.475 \approx 0.24$ と更新される。セル 14 での更新より前であれば、13 からの同じ遷移が起きてもエージェントは何も学べなかったはずである。このように価値は、エージェントがたまたまたどった遷移に沿って、訪問ごとに 1 セルずつゴールから逆方向へ伝播していく。価値反復法では、すべての遷移に沿って一度に、1 スイープにつき 1 セルずつ価値を伝播させていた。

ターゲットにはノイズがある。次にエージェントが 14 で**右**を指示し、まだ要素が 0 のままである上のセル 10 へ滑ったとすると、ターゲットは 0 となり $Q(14,\text{right}) \leftarrow 0.25$ へ下がる。この要素はターゲット 1 と 0 の間で揺れ動き、さらに後にはセル 10 とセル 14 からブートストラップされた値の間で揺れ動く。これらのターゲットの平均だけが本来のバックアップであり、その平

均を形成するためにステップをどれほどの速さで縮めるべきかが、後述するステップサイズの主題である。

14.3.1.3 最小二乗法と二重サンプリング

なぜこのようなノイズだらけのステップが正しい解へたどり着くのだろうか。テーブルを固定し、次状態と報酬にわたって時間差分誤差の平均をとってみよう。

$$\mathbb{E}[\delta \mid s, a] = (\mathcal{T}Q)(s,a) - Q(s,a). \tag{14.32}$$

期待更新式 $Q(s,a) \leftarrow Q(s,a) + \alpha\big((\mathcal{T}Q)(s,a) - Q(s,a)\big)$ は、その 1 要素に対する緩和された価値反復法のステップになっている。つまり Q 学習とは、サンプリングされた要素を 1 つずつ実行する価値反復法であり、各ステップに平均 0 のノイズによる摂動が加わったものである。

回帰モデルを当てはめた経験のある読者なら、テーブルをターゲットに当てはめる自然な方法は最小二乗法（least squares）ではないかと疑問に思うだろう。記録された N 個の遷移 (s_i, a_i, r_i, s_i') が与えられたとき、二乗時間差分誤差を勾配降下法（gradient descent）で最小化することも考えられる。

$$\ell(Q) = \frac{1}{N}\sum_{i=1}^{N}\Big(Q(s_i,a_i) - r_i - \gamma \max_{a'} Q(s_i', a')\Big)^2. \tag{14.33}$$

1 つの項の勾配は、$Q(s_i,a_i)$ を通る成分と、ターゲット内部の最大値を通る成分の 2 つからなる。更新式 Equation 14.31 はターゲットを固定することで前者を残し、後者を捨てる。そのためこの更新は**半勾配**（semi-gradient）ステップと呼ばれ、ℓ に対する勾配降下法ではない。

第 2 の成分を捨てることこそが、Q 学習を正しく機能させる鍵である。データには各対 (s,a) が頻度 $d(s,a)$（データ収集方策の訪問分布）で含まれており、サンプルが無限に多くあるとしよう。(s,a) で条件付けると、各二乗項の期待値は平均の二乗とターゲットの分散に分解される。

$$\begin{aligned} L(Q) = \mathbb{E}_{(s,a)\sim d}\Big[\big(Q(s,a) - (\mathcal{T}Q)(s,a)\big)^2\Big] \\ + \mathbb{E}_{(s,a)\sim d}\Big[\mathrm{Var}\big(r + \gamma \max_{a'} Q(s',a') \;\big|\; s,a\big)\Big]. \end{aligned} \tag{14.34}$$

第 1 項は二乗ベルマン残差であり、Q^* において 0 になる。第 2 項は次状態と報酬に関するターゲットの分散であり、遷移が確率的である限り Q に依存する。L の最小化は残差を小さくすることとターゲットの分散を抑えることのトレードオフになるため、その最小点（minimizer）は Q^* になるとは限らない。3 状態の例を見れば、どの方向に歪むかがわかる。状態 s から唯一の行動をとると、それぞれ確率 $1/2$ で u_1（報酬 1 を与えて終端）または u_2（報酬 0 を与えて終端）へ遷移するとする。このとき $Q^*(u_1) = 1$、$Q^*(u_2) = 0$、$Q^*(s) = \gamma/2$ である。一方、s におけるターゲットの分散は $\gamma^2(Q(u_1) - Q(u_2))^2/4$ となるため、L の最小点はこの分散を下げようとして 2 つの遷移先の価値を互いに引き寄せ、1 と 0 から遠ざけてしまう。どこまで引き寄せられるかは演習問題 4 で計算する。

残差勾配法（residual-gradient methods）は二乗残差の全勾配に沿って降下するが、同じ困難に直面する (Baird 1995)。その勾配の不偏推定量を作るには、同じ対からの独立な 2 つの遷移先（ターゲット用と導関数用）が必要になるが、1 本の軌跡（trajectory）を生きるエージェントが観測できるのは 1 つだけである。これが**二重サンプリング問題**（double-sampling problem）である。遷移と報酬が決定論的であれば分散の項は消え、2 つのアプローチは一致する。ターゲッ

トを固定するという考え方は、ニューラルネットワークを用いる設定では、ターゲット計算用にゆっくり更新される第 2 のネットワークのコピー、すなわち 15.4 節のターゲットネットワークとして再び登場する。

14.3.1.4 ステップサイズと収束

先ほどの手計算の例では、1 つの要素がノイズを含むターゲットの列 $y_1, y_2, \dots$ を次々に受け取っていた。これらを統合する最も素直な方法は平均をとることである。ステップサイズを $\alpha_n = 1/n$（その対への n 回目の訪問時）とおくと、更新式は $Q_n = Q_{n-1} + (y_n - Q_{n-1})/n = \big((n-1)Q_{n-1} + y_n\big)/n$ となり、帰納法により次式を得る。

$$Q_n = \frac{1}{n}\sum_{k=1}^{n} y_k,$$

これは初期値が何であれ、ターゲットの標本平均（sample mean）にほかならない。もしターゲットが一定の平均を持つ独立な抽出値であれば、大数の法則（law of large numbers）によりこの要素はその平均へ収束する。Robbins と Monro は、減少するステップサイズが一般にこれと同じ効果を持つための条件を与えた (Robbins and Monro 1951)。

$$\sum_{n=1}^{\infty} \alpha_n = \infty, \qquad \sum_{n=1}^{\infty} \alpha_n^2 < \infty. \tag{14.35}$$

第 1 の条件はステップが任意の距離まで移動できるようにするものであり、これによって初期値は忘れ去られる。第 2 の条件は累積するノイズを有限に抑える。標本平均に対応する $\alpha_n = 1/n$ は両方を満たす。定数のステップサイズは第 2 の条件に反するため、要素は平均の周りで揺らぎ続け、α が小さいときその分散は α のオーダーになる。

命題（表形式の Q 学習の収束）。状態集合と行動集合が有限であり、報酬が有界かつ $\gamma < 1$ であるとする。すべての対 (s,a) が無限回更新され、そのステップサイズが Robbins–Monro 条件を満たすならば、Q は確率 1 で Q^* に収束する (Watkins and Dayan 1992; Jaakkola et al. 1994; Tsitsiklis 1994)。

平均化の議論だけではこの命題は証明できない。ターゲットが学習途中のテーブルから計算されるため、一定の平均を持たないからである。残差勾配降下法でも期待ステップはある不動点で 0 になるが、それは誤った不動点である。Q 学習を際立たせているのは縮小性である。現在の誤差がどのようなものであれ、期待ステップはその最大誤差を縮め、さらにステップサイズの条件によって各ステップ周りのノイズが平均化されて消えていく。

どのコマンドも指示どおりの方向へ進む滑らない氷の上では、各対の次状態は 1 つに定まり、ターゲットに遷移ノイズは生じない。このとき定数 α を用いた更新は緩和バックアップ $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(\mathcal{T}Q)(s,a)$ となり、全要素に適用すれば縮小率 $(1-\alpha)+\alpha\gamma < 1$ で縮小する。したがって $\alpha = 0.9$ のような大きなステップでも安全である。

ただし、この条件が扱うのは極限であり、有限の計算予算ではない。標本平均はすべてのターゲットに同じ重みを与えるため、テーブルの大部分がまだ 0 だった初期のターゲット（湖の環境では小さすぎる値になる）にも同じ重みがかかってしまう。$\alpha_n = 1/n$ を用いると、Q 学習が必要とする更新回数は $1/(1-\gamma)$（ここでは 20）に対して指数関数的に増大しうるが、$\alpha_n = 1/n^{\omega}$（ただし $1/2 < \omega < 1$）とすれば多項式オーダーの増大にとどまる (Even-Dar and Mansour 2003)。以下では、初期値 0 をもう 1 つのターゲットとして数えた標本平均にあたる $\alpha_n = 1/(1+n)$ を試す。本節でデフォルトとする $\alpha_n = 1/(1+0.1\,n)$ も両方の条件を満たし、そのステップサイ

ズは漸近的に標本平均の 10 倍になる。このスケジュールは後から得られるターゲットほど重く評価し、k 番目のターゲットにはおよそ k^9 に比例する重みを与えるため（演習問題 5）、推定値は改善されていくターゲットに追随できる。

14.3.1.5 終端のマスク処理

エピソード型タスクでは、ターゲットに 1 つ補正が必要になる。穴やゴールといった終端状態にはその先の未来がないため、継続価値は 0 であり、ターゲットには即時報酬のみが含まれる。

$$Q(s,a) \leftarrow Q(s,a) \\ + \alpha\Big(r + \gamma\,(1 - \mathbf{1}(s' \text{ terminal}))\,\max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)\Big).$$

このマスクには `terminated` を用い、`truncated` は用いない。14.1 節で説明したように、時間制限による打ち切りは本来なら過程が続くはずの状態で記録を止めるだけなので、その継続価値は 0 ではない。打ち切りを終端として扱ってしまうと、長いエピソードの最後のセルはどれも価値がないとテーブルに学習させてしまう。コード上では、この補正は `gamma * (1 - terminated) * Q[s_next].max()` という 1 つの因数で表される。

14.3.2 滑る湖における Q 学習

ここからは 14.1 節の滑る湖に対して $\gamma = 0.95$ でこの更新を実行し、14.2 節の厳密解と照らし合わせて結果を評価する。学習器は `reset` と `step` のみを通じて環境とやり取りする。モデル `mdp`、最適価値 `V_star`、最適行動価値 `Q_star`（V^* の 1 回のバックアップ）、および最適方策（optimal policy）`pi_star` は、結果の評価にのみ用いる。

```
gamma, num_episodes, num_seeds = 0.95, 4000, 5
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
Q_star = mdp.backup(V_star)
pi_star = Q_star.argmax(axis=1)
```

14.3.2.1 行動方策

Q 学習には、学習中に行動を選ぶための規則である**行動方策**（behavior policy）π_e が必要である。現在のテーブルに対して貪欲に行動するだけでは、2 つの理由で失敗する。第 1 に、すべて 0 のテーブルではどの行も 4 つすべてが同値になり、`np.argmax` は最初のインデックスを返すため、貪欲なエージェントは常に**左**を指示して決してゴールを見つけられない。第 2 に、現時点で要素の値が低い行動は一度も試されないため、その要素は更新されず、過小評価が永続してしまう。最も単純な打開策は、小さな確率 ϵ でランダムに行動することである。

$$\pi_e(a \mid s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & a = \operatorname{argmax}_{a'} Q(s,a'), \\ \epsilon/|\mathcal{A}| & \text{otherwise.} \end{cases} \tag{14.36}$$

これが ϵ **貪欲**（epsilon-greedy）方策である。確率 ϵ で探索（exploration）し、それ以外のときは argmax における同値を一様ランダムに解消しつつ貪欲行動をとる。次の関数は一様乱数を 1 つ引いてどちらにするかを決め、ランダムな行動または最大点（maximizer）の集合からランダムに選んだ要素を返す。

```
def epsilon_greedy(q, epsilon, rng):
    """Explore with probability epsilon, else act greedily on the values q."""
    if rng.random() < epsilon:
        return int(rng.integers(len(q)))
    # Ties are coin flips: np.argmax would always pick action 0 on a zero
    # table, and an agent that only ever moves left never finds the goal.
    return int(rng.choice(np.flatnonzero(q == q.max())))
```

何も知らないテーブルは大いに探索すべきであり、ほぼ正しいテーブルは大半のステップで活用（exploitation）すべきである。そこで、訓練が進むにつれて ϵ を減少させる。`linear_schedule` は開始値、終了値、ステップ数を受け取り、ステップ数を引数として線形補間したのち終了値を維持する関数を返す。

```
def linear_schedule(start, end, num_steps):
    """step -> value, interpolated from start to end, then held at end."""
    return lambda step: end + (start - end) * max(0.0, 1.0 - step / num_steps)
```

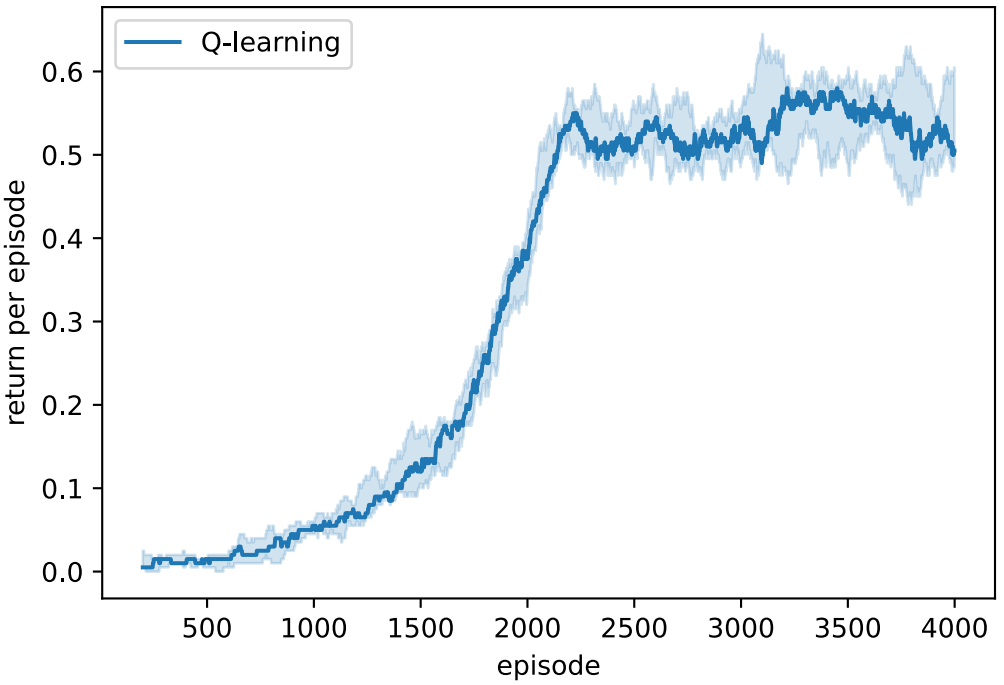
14.3.2.2 学習ループ

`q_learning` は終端マスク付きの更新式 Equation 14.30 を実行する。テーブル `Q` と訪問回数のテーブル `visits` を受け取り、両方をインプレースで書き換える。訪問回数は関数 `alpha(n)` を通じて各対のステップサイズを定め、実行後にはどの対が試されたかの記録にもなる。探索率（exploration rate）はエピソードを引数とする関数 `epsilon(k)` であり、デフォルトでは訓練の前半で 1 から 0.05 まで減少したのち 0.05 に保たれる。ループ本体は行動を選び、1 回の遷移を観測し、1 つの要素を $\alpha\delta$ だけ変化させ、各エピソードの収益を順に返す。

```
def q_learning(seed, Q, visits, env, num_episodes, gamma,
               alpha=lambda n: 1 / (1 + 0.1 * n), epsilon=None):
    """Tabular Q-learning; updates Q in place, yields each episode's return.
    `alpha(n)` is the step size at the n-th visit of a pair; `epsilon(k)` is
    the exploration rate in episode k (default: 1 to 0.05 over the first
    half of the episodes, then held)."""
    rng = np.random.default_rng(seed)
    if epsilon is None:
        epsilon = linear_schedule(1.0, 0.05, num_episodes // 2)
    env.reset(seed=seed)
    for episode in range(num_episodes):
        s, done, ret = env.reset()[0], False, 0.0
        while not done:
            a = epsilon_greedy(Q[s], epsilon(episode), rng)
            s_next, r, terminated, truncated, _ = env.step(a)
            visits[s, a] += 1
            delta = r + gamma * (1 - terminated) * Q[s_next].max() - Q[s, a]
            Q[s, a] += alpha(visits[s, a]) * delta
            s, done, ret = s_next, terminated or truncated, ret + r
        yield ret
```

テーブルに必要なのはインデックス演算だけなので、本節のコードセルはすべて素の NumPy で書かれている。シードごとに 1 つのテーブルを 4,000 エピソード訓練し、各エピソードの収益をプロットする。

```
Q = np.zeros((num_seeds, mdp.num_states, mdp.num_actions))
visits = np.zeros_like(Q)
returns = np.array([list(q_learning(k, Q[k], visits[k], env, num_episodes,
                                    gamma)) for k in range(num_seeds)])
d2l.plot_curves({'Q-learning': returns}, xlabel='episode',
                ylabel='return per episode', smooth=200)
```



曲線の各点はそこまでの直近 200 エピソードを平均しているため、曲線はエピソード 200 から始まる。帯はシード間の幅を表す。成功がまれなうちは収益は 0 付近にとどまり、ゴールから価値が逆伝播するにつれて上昇し、やがて 0.5 付近で横ばいになる。これはエピソード 2,000 で探索率が下限に達した後の領域である。

14.3.2.3 学習されたテーブルと厳密解

0.5 付近という収益は良い結果なのだろうか。14.2 節の最適方策は、100 ステップの制限時間内に約 73% の確率でゴールに到達する。この最適方策は速く到達するほど有利になる割引収益を最大化するため、割引を無視した方策のほうが成功率は 1 ポイントほど高くなる。さらに、訓練曲線が測っているのは 20 ステップに 1 回の割合でランダムに行動し続ける行動方策であり、テーブルそのものを評価しているわけではない。

ここではモデルが既知であるため、テーブルを厳密に評価できる。$\gamma = 1$ とし、ゴールでのみ報酬 1 が与えられるとすると、開始状態の価値はゴールへの到達確率に一致する。すべて 0 の状態から 100 スイープ実行すると、14.2 節の `policy_evaluation` は 100 ステップ以内のゴール到達確率を計算する。`success` は状態ごとに 1 行の行動確率で与えられた方策に対してこの方策評価（policy evaluation）をラップする関数である。マスク `live` はエージェントが行動するセルを表し、`greedy` は各テーブルの貪欲方策を保持する。次のコードセルでは、各テーブルの価値誤差と行動価値誤差、Q^* のもとで貪欲行動が取り逃す最大の価値、厳密な成功率、π^* が確率 0.05 でラ

ンダムに行動したときの成功率、および 1 回の実行が要したステップ数を表示する。

```
lake1 = d2l.TabularMDP.from_gym(env, 1.0)    # gamma = 1: V(s0) = P(goal)
success = lambda pi: d2l.policy_evaluation(lake1, pi, 100)[-1][0]
live = ~np.isin(env.unwrapped.desc.flatten(), [b'H', b'G'])  # agent acts
greedy = np.eye(mdp.num_actions)[Q.argmax(axis=-1)]    # (seed, s, a)
optimal = np.eye(mdp.num_actions)[pi_star]
taxed = 0.95 * optimal + 0.05 / mdp.num_actions        # pi* at epsilon = 0.05
v_err = np.abs(Q.max(axis=-1) - V_star).max(axis=-1)   # one entry per seed
q_err = np.abs(Q - Q_star)[:, live].max(axis=(1, 2))
regret = (V_star - (greedy * Q_star).sum(-1)).max(-1)  # what greedy gives up
wins = [success(g) for g in greedy]
for name, per_seed in [('max_s |V_Qhat(s) - V*(s)|', v_err),
                       ('max_(s,a) |Qhat - Q*|', q_err),
                       ('max_s V*(s) - Q*(s, greedy)', regret),
                       ('greedy success in 100 steps', wins)]:
    print(f'{name:>27}: {np.round(per_seed, 3)}')
steps = np.median(visits.sum(axis=(1, 2)))
print(f'success of pi* {success(optimal):.3f}, of pi* at epsilon = 0.05 '
      f'{success(taxed):.3f}; median environment steps {steps:,.0f}')
```

```
  max_s |V_Qhat(s) - V*(s)|: [0.019 0.012 0.021 0.008 0.006]
      max_(s,a) |Qhat - Q*|: [0.131 0.079 0.113 0.087 0.075]
max_s V*(s) - Q*(s, greedy): [0. 0. 0. 0. 0.]
greedy success in 100 steps: [0.73 0.73 0.73 0.73 0.73]
success of pi* 0.730, of pi* at epsilon = 0.05 0.537; median environment steps 95,569
```

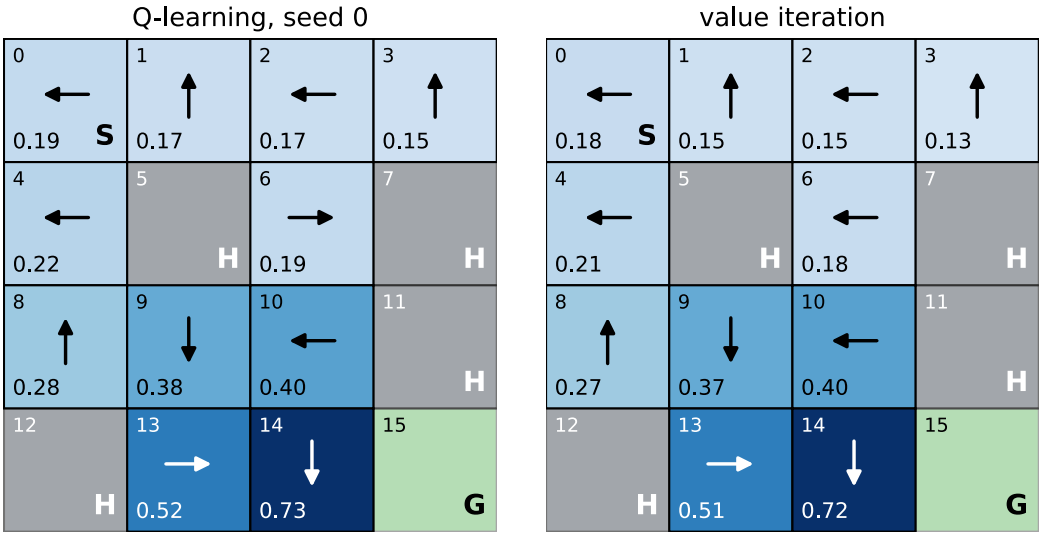
3 行目の値はすべてのシードで 0 になっている。すなわち、学習された貪欲行動が Q^* のもとで価値を損なうセルは 1 つもなく、どの貪欲方策も同値の選び方を除いて最適であり、厳密に 73%（π^* と同じ成功率）で成功している。ただしこれは典型的な実行結果であり、この予算での保証ではない。およそ 8 回に 1 回の実行では、$4{,}000$ エピソード後でも一部の貪欲行動が準最適のまま残るが、たいてい損失はごくわずかである（成功率は 71%〜74%）。まれに起こる最悪のケースでは、エージェントが開始状態で依然として**右**を選好し、成功率が半分未満にとどまる。開始状態の上位 2 つの行動は 0.01 未満しか Q^* の値が違わず、その実行では推定値が低く出たうえ、貪欲規則のもとでは探索のときにしかその行動が試されないため、低い推定値の修正が遅れるからである。価値誤差はシード全体にわたって約 0.03 以内に V^* との差が収まっている。一方、行動価値誤差はその数倍大きく、約 0.05〜0.17 である。貪欲方策が選ばない行動の要素は主にエージェントが探索するときにしか更新されないため、平均化されるターゲットの数が少ないからである。

20 ステップに 1 回ランダムに行動するよう強いられると、π^* でさえ成功率は約 54% にとどまる。訓練曲線が 0.5 付近で頭打ちになるプラトーは、この探索のコストによるものであり、テーブルの欠陥ではない。

1 回の実行は典型的に約 $96{,}000$ 環境ステップを使用する。価値反復法は V^* を、164 スイープ（各スイープ 64 回のバックアップ）、すなわち約 $10{,}500$ 回のバックアップで確定させたが、この 2 つの回数は異なるものを測っている。1 回のバックアップは遷移テーブルの 1 行全体を読み取るのに対し、1 ステップが明かすのはサンプリングされた 1 つの遷移先だけである。演習問題 6 では、必要な情報量の違いを明示したうえで両者を比較する。

最初のシードで学習された価値と矢印を、厳密解と並べて次に示す。

```
d2l.show_grid(env.unwrapped.desc, np.stack([Q[0].max(-1), V_star]),
              np.stack([Q[0].argmax(-1), pi_star]),
              titles=['Q-learning, seed 0', 'value iteration'])
```



学習された矢印は 14.2 節の滑りを考慮した戦略を再現しているが、例外はセル 6 であり、そこでは Q 学習が**右**を指示し、π^* が**左**を指示する。セル 6 は穴である 5 と 7 の間に位置し、この 2 つのコマンドは Q^* のもとで等しい価値を持つため、不一致は厳密な同値に対する解消の仕方の違いにすぎない。価値の精度はエージェントが頻繁に訪れるセルで最も高く、めったに訪れない右上の隅では粗くなっている。

14.3.2.4 ステップサイズスケジュールの比較

滑らない氷の上ではターゲットに遷移ノイズが含まれないため、前述の縮小議論により定数ステップ $\alpha = 0.9$ が許容される。このときエピソード予算の 1/8 でほぼ常に十分であり、ここでは最終テーブルの貪欲方策がすべてのエピソードでゴールに到達する。

```
calm = gym.make('FrozenLake-v1', is_slippery=False)
table = np.zeros((mdp.num_states, mdp.num_actions))
calm_visits = np.zeros_like(table)
list(q_learning(0, table, calm_visits, calm, 512, gamma,  # fills table
                alpha=lambda n: 0.9))
print(f'calm ice, 512 episodes ({calm_visits.sum():,.0f} steps), alpha = 0.9: '
      f'greedy policy succeeds '
      f'{d2l.evaluate(calm, lambda s, rng: int(table[s].argmax()), 100):.0%}')
```

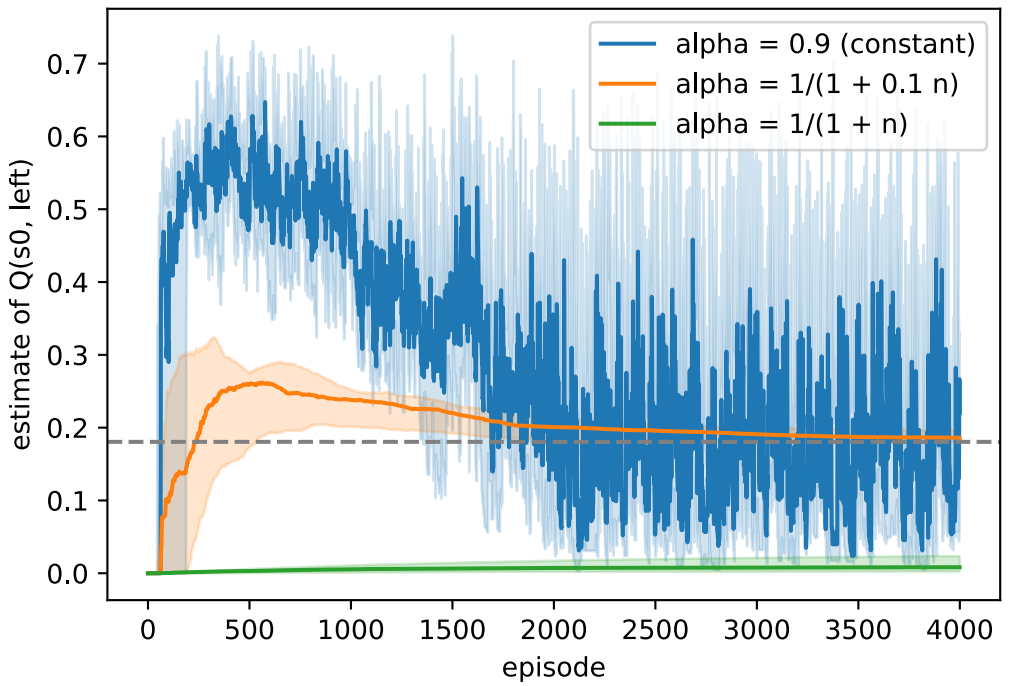
```
calm ice, 512 episodes (3,658 steps), alpha = 0.9: greedy policy succeeds 100%
```

滑らない氷での実行は約 3,500 環境ステップで済むのに対し、滑る氷での実行は約 96,000 ステップを要する。まれに失敗する実行では、エージェントがゴールに到達した回数が少なすぎて開始状態まで価値が逆伝播しておらず、これは 14.4 節で扱う探索の失敗にあたる。コストの大半を決めているのは学習規則ではなく遷移ノイズである。滑る氷の上ではステップサイズが重要になる。そこで同じシード上で 3 つのスケジュールを、1 つの要素 $Q(s_0, \text{left})$（その厳密値 0.180 は

π^* のもとでの開始状態の価値である）を追跡して比較する。プロットにはシード全体の中央値（median）と、最小値から最大値までの帯を示す。

```
a0 = int(pi_star[0])          # FrozenLake's actions: left, down, right, up
trace = {}
for name, alpha in [('0.9 (constant)', lambda n: 0.9),
                    ('1/(1 + 0.1 n)', lambda n: 1 / (1 + 0.1 * n)),
                    ('1/(1 + n)', lambda n: 1 / (1 + n))]:
    runs = []
    for seed in range(num_seeds):
        Qa = np.zeros_like(Q[0])  # updated in place: read after each episode
        runs.append([Qa[0, a0] for _ in q_learning(
            seed, Qa, np.zeros_like(Qa), env, num_episodes, gamma, alpha)])
    trace[f'alpha = {name}'] = np.array(runs)
    print(f'alpha = {name:>14}: final estimates {np.round(runs, 3)[:, -1]}')
print(f'peak of each 1/(1 + 0.1 n) run:  '
      f'{np.round(trace["alpha = 1/(1 + 0.1 n)"].max(axis=1), 3)}')
d2l.plot_curves(trace, xlabel='episode', ylabel='estimate of Q(s0, left)',
                reference=Q_star[0, a0])
```

```
alpha = 0.9 (constant): final estimates [0.056 0.221 0.206 0.265 0.236]
alpha =  1/(1 + 0.1 n): final estimates [0.19  0.186 0.195 0.184 0.18 ]
alpha =      1/(1 + n): final estimates [0.003 0.024 0.008 0.009 0.002]
peak of each 1/(1 + 0.1 n) run:  [0.228 0.291 0.262 0.324 0.205]
```



定数ステップは決して落ち着かない。各ターゲットが推定値を誤差の 90% 分だけ動かすため、4,000 エピソード後の最終推定値は 0.180 の周りに 0.05 付近から 0.25 超まで広く散らばる。デフォルトのスケジュールは、典型的な実行では厳密値から約 0.02 以内に収束して終わる。スケジ

ュール $1/(1+n)$ も Robbins–Monro 条件を満たすが、この予算ではどの実行も厳密値の 1/6 にあたる 0.03 未満にとどまる。すべて 0 のテーブルから計算された初期のターゲットが、そのままの重みを保ち続けるからである。これこそ Even-Dar と Mansour が予測した遅い収束速度である。条件が保証するのは極限であり、有限の予算に適したスケジュールを選んでくれるわけではない。

14.3.2.5 推定とデータ収集の間のフィードバック

行動方策は現在の推定値に依存するため、推定値に誤差があると次にどのデータが収集されるかが変わる。$Q(s,a)$ が大きすぎれば、ϵ 貪欲方策は a をより頻繁に選び、推定値を引き下げるようなターゲットを持つ遷移を集める。逆に $Q(s,a)$ が小さすぎれば、その行動はエージェントが探索するときにしか試されず、上の実行結果が示したように誤差の修正は遅くなる。したがって過大評価は自己修正されるが過小評価はそうならず、先ほど推定値が低かった実行で最初の行動が変わるまでに長い時間がかかったのもそのためである。

更新の不動点では、行動方策が訪問するすべての対において期待補正量が 0 になり、そこでは $Q(s,a) = (\mathcal{T}Q)(s,a)$ が成り立つ。しかしこれはまだ最適方程式ではない。$\mathcal{T}Q$ の内部にある最大値演算は、一度も訪問されず制約を受けない要素にも依然としてわたっているからである。テーブルが訪問された対の上で方程式を満たしていても全体としては誤っていることがありうるため、収束定理ではすべての対が試されることが必要になる。固定された遷移ログから学習するエージェントはこのフィードバックを完全に失うが、その場合については次章で改めて扱う。

14.3.3 オフポリシー学習

ターゲットに最大値演算が含まれることで、Q 学習はある方策に従いながら別の方策について学習する。以下の 2 つの項では、その性質とそれに伴う統計的バイアスを扱う。

14.3.3.1 オフポリシーターゲットと SARSA

Equation 14.31 の $\max_{a'}$ は、行動方策が s' で次にどの行動をとるかにかかわらず、貪欲な継続を評価する。したがって Q 学習は**オフポリシー**である。すなわち、データは探索を行う方策 π_e から得られる一方で、貪欲方策の価値を推定する。これに対するオンポリシーの代替手法は、行動方策が実際に選んだ行動 a'（状態 s' において）からブートストラップする。

$$\delta = r + \gamma\, Q(s', a') - Q(s, a). \tag{14.37}$$

この手法は、各更新が用いる組 (s, a, r, s', a') にちなんで **SARSA** と呼ばれる (Rummery and Niranjan 1994; Sutton and Barto 2018)。固定された行動方策に対して、SARSA の更新の期待値は π_e のベルマン期待方程式（Bellman expectation equation）を不動点に持つため、SARSA はランダムな行動も含めた探索方策の価値 Q^{π_e} を学習する。滑る湖において $\epsilon = 0.05$ を保ち続ける SARSA エージェントは、自身のランダムなステップでときおり穴に落ちるため、穴の隣のセルの価値を π^* がそこで達成する価値より低く見積もることになる。これに対し Q 学習は、粘り強く探索を続けながら、上で見たように π^* と一致する貪欲方策を導き出せる。同じ性質により、貪欲方策が必要とする対をデータが網羅している限り、過去の経験や別のエージェント、あるいは固定されたログからでも学習できる。ただしテーブルをニューラルネットワークに置き換えると、この柔軟性がかえって学習を不安定にしうることを次章で見る。

14.3.3.2 最大化バイアス

ノイズを含む推定値の最大値は、それぞれの平均の最大値を上回りやすい。最大値演算が、たまたま誤差が正になった要素を選び出すからである。最大値関数は凸関数であるため

$\mathbb{E}[\max_{a'} \hat{Q}(s', a')] \geq \max_{a'} \mathbb{E}[\hat{Q}(s', a')]$ が成り立ち、ノイズを含む行からブートストラップするすべてのターゲットはこの**最大化バイアス**（maximization bias）を受け継ぐ。先ほどのステップサイズごとの推移にもそれが現れている。デフォルトのスケジュールでは、$Q(s_0, \text{left})$ の推定値はいったん 0.180 を超えて上昇してから落ち着く。ノイズがなければこのような挙動は起こりえない。すべて 0 のテーブルから出発し、報酬が決して負にならない設定では、厳密なバックアップはいずれも価値を Q^* に向けて引き上げることしかできないため、価値反復法は下から 0.180 に近づく。それにもかかわらず、中央値の推移は約 0.22 以上の水準にエピソード 500 からエピソード $1{,}500$ までとどまっている。ノイズが最大値演算を通ることで推定値を押し上げたのである。表示された各実行のピークは約 0.2 から 0.33 の間にあり、12 個のシードにわたって大半の実行が同様に 0.01 の数倍から 0.1 を超える幅でオーバーシュートする。推定値のノイズがさらに大きい定数ステップではオーバーシュートもいっそう大きくなり、最初の 1,000 エピソードの大部分で約 0.5 に達する。

表形式の推定値が収束するにつれてこのバイアスは縮小するが、ノイズが消えない関数近似（function approximation）のもとでは残り続けることがある。ダブル Q 学習（Double Q-learning）は 2 つの推定器を保持し、一方で最大値を与える行動を選び、もう一方でその価値を評価することで上向きのバイアスを取り除く（ただし推定値が低めに出ることはある）(Hasselt 2010)。演習問題 7 では、このバイアスがエージェントの判断を誤らせるタスク上でダブル Q 学習を構築する。

14.3.4 学習されたモデルにおける計画

Q 学習は各遷移を 1 度しか使わない。1 つの要素を動かすとそのサンプルを捨ててしまうため、次にゴールの価値が変化しても、過去の遷移が残っていてその変化を逆方向へ伝えてくれることはない。これに対し、同じ遷移を数え上げておくこともできる。$N(s, a, s')$ を状態 s で行動 a をとって s' へ移った観測遷移回数とし、$N(s, a) = \sum_{s'} N(s, a, s')$ とおく。この回数から、次の推定モデルが得られる。

$$\hat{P}(s' \mid s, a) = \frac{N(s, a, s')}{N(s, a)},$$
$$\hat{r}(s, a) = \frac{1}{N(s, a)} \sum_{\text{visits of } (s,a)} r,$$

すなわち、各次状態の経験頻度と各対の平均報酬である。推定値をあたかも厳密な真のモデルであるかのように扱って $(\hat{P}, \hat{r})$ 上で価値反復法を行う手法は、**確実性等価**（certainty equivalence）と呼ばれる。これにより、モデルなしの問題は 14.2 節の問題へと還元される。

`Counts` は環境をラップし、すべてのステップをそのまま通しつつ $N(s, a, s')$ と報酬の合計を記録する。`q_learning` は `reset` と `step` のみを通じて環境に触れるため、このラッパ上でも変更なしにそのまま動作し、ラッパなしのときとまったく同じテーブルを学習する。`learned_model` はカウントを割り算して $\hat{P}$ と $\hat{r}$ を求め、`d2l.TabularMDP` として返す。一度も試されなかった対の行は 0 のまま残るため、プランナはそれらを価値のない行き止まりとして扱う。

```
class Counts(gym.Wrapper):
    """Pass every step through and count it: N[s, a, s'] and summed rewards."""
    def __init__(self, env):
        super().__init__(env)
        n_s, n_a = env.observation_space.n, env.action_space.n
        self.N, self.R = np.zeros((n_s, n_a, n_s)), np.zeros((n_s, n_a))
```

```
    def reset(self, **kwargs):
        self.s, info = self.env.reset(**kwargs)
        return self.s, info

    def step(self, a):
        s_next, r, terminated, truncated, info = self.env.step(a)
        self.N[self.s, a, s_next] += 1
        self.R[self.s, a] += r
        self.s = s_next
        return s_next, r, terminated, truncated, info

def learned_model(counts, gamma):
    """Certainty equivalence: the empirical kernel and mean rewards."""
    n = np.maximum(counts.N.sum(axis=-1), 1)   # unvisited pairs: zero rows
    return d2l.TabularMDP(counts.N / n[..., None], counts.R / n, gamma)
```

モデルはデータをどれほど有効に活用できるだろうか。Q 学習に 500 エピソードの短い予算と 2,000 エピソードの長い予算を与え、それぞれ独立した探索スケジュールで最後まで実行する。各実行で得られた遷移データ上で、Q 学習が学習したテーブルと、推定モデルの行動価値（価値反復法による解を 1 回バックアップしたもの）という 2 つのテーブルを比較する。`grade` はテーブルの `V_star` に対する上限ノルム（sup-norm）価値誤差と、`success` で計算した貪欲方策の厳密な成功率を返す。

```
def grade(Q, V_star, success):   # sup-norm error, success of greedy policy
    return (np.abs(Q.max(axis=-1) - V_star).max(),
            success(np.eye(Q.shape[-1])[Q.argmax(axis=-1)]))

for budget in [500, 2000]:
    rows = []
    for seed in range(num_seeds):
        counts, Qs = Counts(env), np.zeros((mdp.num_states, mdp.num_actions))
        list(q_learning(seed, Qs, np.zeros_like(Qs), counts, budget,
                        gamma))                    # fills Qs and the counts
        model = learned_model(counts, gamma)
        Q_model = model.backup(d2l.value_iteration(model, 500)[-1])
        rows.append([counts.N.sum(), *grade(Qs, V_star, success),
                     *grade(Q_model, V_star, success)])
    n, err_q, win_q, err_m, win_m = np.median(rows, axis=0)
    print(f'{budget} episodes, median {n:,.0f} steps '
          f'(medians over {num_seeds} seeds)')
    print(f'  Q-learning:        max |V - V*| {err_q:.3f}, success {win_q:.3f}')
    print(f'  planning in model: max |V - V*| {err_m:.3f}, success {win_m:.3f}')
```

```
500 episodes, median 9,984 steps (medians over 5 seeds)
  Q-learning:        max |V - V*| 0.084, success 0.486
  planning in model: max |V - V*| 0.096, success 0.639
2000 episodes, median 47,282 steps (medians over 5 seeds)
  Q-learning:        max |V - V*| 0.023, success 0.730
  planning in model: max |V - V*| 0.011, success 0.730
```

約 10,000 ステップという少ないデータ量では、計画によって得た貪欲方策のほうが優れている。Q 学習の貪欲方策がゴールに到達するのはおよそ 2 回に 1 回であるのに対し、計画に基づく貪欲

方策は典型的に 3 回に 2 回ほど到達し、10 回中 9 回ほどの実行で同等以上の成績を収めた。この予算での最悪ケースの価値誤差は、めったに訪問されない少数のセルによって決まる。この誤差もたいていはモデルのほうが小さいが常にそうとは限らず、ここに表示した 5 回の実行ではたまたま Q 学習のほうが小さくなっている。約 $47{,}000$ ステップになると、大半の実行で両方の貪欲方策が最適になり、計画による価値の精度は約 2 倍高くなる。ほぼすべてのシードで計画のほうが優れており、2 倍のエピソード数をかけた Q 学習のテーブルと同程度の精度に達していた。理由は先ほどの手計算の遷移が示したとおりである。Q 学習は訪問ごとに 1 遷移ずつ価値を逆伝播させるうえ、その遷移が得られた時点のまだ不正確なテーブルから計算したターゲットを使う。これに対しモデルはすべての遷移を保持しており、価値反復法の数百回のスイープそれぞれが、記録された全遷移を通してゴールの報酬を伝播させる。演習問題 1 では、2 状態の過程で同じ効果を確認する。

もっとも、モデルには固有のコストがある。モデルは $|\mathcal{S}|^2|\mathcal{A}|$ 個のカウントを保存する必要があり、ここでは約 1,000 個の数値で済むものの大きな状態空間では到底扱いきれず、さらに新しいデータのバッチが得られるたびにモデルを解き直さなければならない。またプランナは推定モデルを完全に信用するため、一度も試されなかった対は無価値に見え、1 度だけ試されて運よく良い結果が出た対は実態以上に良く見えてしまう。15.6 節では、カウントの代わりに回帰を用いて Pendulum 上でこのようなモデルを当てはめ、そのモデル内で計画を行う。

こうしてエージェントは、サンプルだけから最適行動価値を学習してそれに基づき行動することも、同じサンプルをモデルに変えて計画を立てることもできるようになった。どちらの方法もデータをどう収集するかに支えられているが、ここで用いた ϵ 貪欲規則は大雑把なものである。20 ステップに 1 回ランダムに行動するよう強いられると π^* でさえ成功率は約 54%（本来の 73% ではなく）に落ちるうえ、貪欲なテーブル参照には最大化の対象となる行動集合が有限である必要がある。14.4 節では、別の規則によって損失を抑えつつ同等の学習ができるかを問い、さらに本章の後半では方策そのものを学習対象とする。

14.3.5 まとめ

Q 学習は、ベルマンバックアップの期待値を観測された 1 回の遷移で置き換え、訪問した要素をステップサイズ α で時間差分誤差 δ に沿って動かす。期待ステップは緩和された価値反復法のステップであり、ベルマン作用素が縮小写像であるため、すべての対が無限回試されステップサイズが Robbins–Monro 条件を満たすとき、表形式の Q 学習は Q^* に収束する。これは半勾配法である。遷移が確率的な場合、二乗時間差分誤差に対する勾配降下法は別の点へ収束してしまい、これが二重サンプリング問題であった。滑る湖では典型的な実行で最適な貪欲テーブルが得られ、そこに到達する速さはステップサイズスケジュールが決める。またターゲット内の最大値演算は Q 学習をオフポリシーにする一方で、推定値にノイズがあるうちは上向きのバイアスを生じさせる。同じ遷移を数え上げて推定モデル上で計画を行えば、モデルの保存と求解のコストと引き換えに、同じデータからより正確な価値が得られる。

14.3.6 演習問題

1. **バッチ TD 対モンテカルロ**。マルコフ報酬過程とは行動を持たない MDP であり、各状態が報酬と次状態に関する固定された分布を持つ。そのような過程の 1 つとして、2 つの非終端状態 x と y を持ち、$\gamma = 1$ であるものを考える。あるバッチに 8 つのエピソードが記録されているとする。1 つは x から y へ報酬 0 で遷移したのち報酬 0 で終端し、6 つは y から始まって報酬 1 で終端し、1 つは y から始まって報酬 0 で終端する。TD(0) は Equation 14.31 から最大値演算を除いた状態価値の形式 $V(s) \leftarrow V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$ である。

    1. このバッチから $V(x)$ と $V(y)$ のモンテカルロ推定値を計算し、さらに小さなステップサイズでこのバッチを何度も繰り返し再生したときに TD(0) が収束する値を求め

よ。
2. 観測された 8 つの収益に対する二乗誤差が小さいのは、2 組の推定値のどちらか。
3. 回数を数えることでこの過程の遷移確率と期待報酬を推定し、そのベルマン方程式を厳密に解け。得られた解は 2 組の推定値のどちらを再現するか。
4. 今後 x から始まるエピソードの収益をより良く予測すると期待されるのはどちらの組か。また、それは過程に関するどのような仮定のもとでか。

Richard S. Sutton および Andrew G. Barto の『Reinforcement Learning: An Introduction』Example 6.4 を改変。

2. **終端として扱われた打ち切り。**状態が 1 つ、行動が 1 つで、毎ステップ報酬 1 を与え、決して終端しないタスクを考える。時間制限によって各エピソードは H ステップ後に打ち切られるものとする（`FrozenLake-v1` が 100 ステップ後に打ち切るのと同様である）。誤った実装が、ブートストラップのマスクに `terminated or truncated` を（`terminated` ではなく）用いたとする。

   1. 減衰するステップサイズを用いると、推定値は 1 エピソードの H 回の遷移にわたる期待更新量が 0 になる値へ収束する。この値を γ と H の関数として求め、正しい価値 $1/(1-\gamma)$ と $\gamma = 0.95$、$H = 100$ において比較せよ。この誤った実装は実質的にどの割引率を使っていることになるか。
   2. 今度は逆に時間制限がタスク自体の一部であり、収益が本当に H ステップ後に終わるものとする。単一の数値ではこの状態の正しい価値を表せないことを示せ。問題 5（14.1 節）と同様に、残りステップ数 h を状態に加えよ。各状態 (s, h) の価値を求め、この場合にはマスク `terminated or truncated` が正しいことを示せ。

3. **貪欲な SARSA と Q 学習。**SARSA は Equation 14.31 の最大値を、行動方策が選んだ行動 a'（状態 s' において）の価値で Equation 14.37 のように置き換える。ある実装では、a' を選んだ後に $Q(s, a)$ を更新し、次のステップでその a' を実行する。両手法とも現在のテーブルに対して貪欲に行動を選び、同値の解消も同じ固定された規則で行うとする。このとき SARSA と Q 学習は同じ行動選択と同じ更新を行う同一のアルゴリズムになるか。もし異なるなら、次のステップで両手法が異なる行動を選ぶような滑らない湖における遷移とテーブルの値を構成せよ。

Richard S. Sutton および Andrew G. Barto の『Reinforcement Learning: An Introduction』Exercise 6.12 を改変。

4. [code] **二重サンプリングの反例。**●各状態につき行動が 1 つだけの 3 状態 MDP を構築せよ。状態 s では、行動は報酬 0 を与え、u_1 または u_2 へそれぞれ確率 $1/2$ で遷移する。u_1 では報酬 1 を与えて終端し、u_2 では報酬 0 を与えて終端する。3 つの要素を $Q(s)$、$Q(u_1)$、$Q(u_2)$ と書く。

   1. サンプル目的関数 Equation 14.33 の 1 項をテーブルに関して微分せよ。非ゼロの導関数を持つのはどの要素であり、半勾配更新 Equation 14.31 が捨てるのはそのうちのどの導関数か。
   2. 3 つの要素に対する Q^* を書き下せ。
   3. d を 3 つの対の上の一様分布として母集団目的関数 Equation 14.34 を明示的に書き下し、γ の関数として閉形式で最小化せよ。最小点は Q^* に比べてどの要素をどの方向へ歪めるか。
   4. 3 つの対を同じ頻度で訪問し、減衰するステップサイズを用いて、半勾配更新と (a) の全導関数に対する確率的勾配降下法の両方を実行せよ。それぞれが (b) と (c) で予測した点へ収束することを確かめよ。
   5. 環境を状態 s にリセットでき、s を訪問するたびに独立な 2 つの次状態が得られるとする。一方をターゲットに使い、もう一方を (a) の導関数に使え。得られる確率的勾

配が二乗ベルマン残差の勾配に対する不偏推定量になることを示し、Q^* へ収束することをシミュレーションで確かめよ。

5. [code] **有限予算下でのステップサイズ。**●本節では、定数のステップサイズをスケジュール $\alpha = 1/(1 + c\,n)$（$c = 0.1$ および $c = 1$、ただし n は今回を含めたその要素への訪問回数）と比較した。解析のため、1 つの要素が受け取るターゲットは平均 m、分散 σ^2 の独立な抽出値として扱え。

   1. 定数 α について、多数回の更新後における推定値の分散を求めよ。それを $\alpha = 0.9$ において $Q(s_0, \text{left})$ に対して評価せよ。ただしターゲットは $r + \gamma V^*(s')$（s' は本節の `mdp.P[0, a0]` から抽出し、`V_star` も本節のものを使用）とし、得られた標準偏差を、本節で表示された厳密値 0.180 の周りでの定数ステップ最終推定値のばらつきと比較せよ。この要素では解析のどの仮定が破れており、残りのばらつきはどこから生じているか。
   2. $c = 1$ のとき、n 回の訪問後の推定値を初期値と n 個のターゲットの重み付き和として書き下せ。0 による初期化にはどの重みがかかるか。
   3. 一般の c について、k 番目のターゲットが n 回の訪問後に持つ重みを積の形で書き下し、その積の対数を積分と比較することで、重みが $k^{1/c-1}$ のオーダーで（k の関数として、大きな $k \le n$ に対して）増大することを示せ。$c = 0.1$ は初期の不正確なテーブルから計算されたターゲットをどのように扱うか。
   4. 滑る湖において $c \in \{0.01, 0.03, 0.1, 0.3, 1\}$ を各 5 シードでスイープし、横軸を c として、最終誤差 $|\widehat{Q}(s_0, \text{left}) - Q^*(s_0, \text{left})|$ と全要素にわたる上限ノルム誤差 $\|\widehat{Q} - Q^*\|_\infty$ の両方をプロットせよ。曲線の形状を (a) と (c) を用いて説明せよ。2 つの誤差指標は同じ c を選ぶか。

6. [code] **サンプル対バックアップ。**●`Q_star` を正解として用いよ。

   1. 滑る湖での `q_learning` の各エピソード後に $\|\widehat{Q} - Q^*\|_\infty$ を記録し、台帳 `visits` が数える環境ステップ数に対してプロットせよ。さらに同じ軸上に、Q_k を `mdp.backup`（`d2l.value_iteration` の k 回目の反復に対するもの）とおき、1 スイープあたり 64 回のベルマンバックアップ数に対して価値反復法の誤差をプロットせよ。Q 学習が 4000 エピソード後に到達する誤差に価値反復法が達するには、何回のバックアップが必要か。
   2. `q_learning` を `Counts(env)` 上で実行し、100、400、1000、4000 エピソード後に本節と同様に `learned_model` を `d2l.value_iteration` で解け。同じ環境ステップ数における推定モデルの行動価値の誤差をプロットに追加せよ。4000 エピソード後の Q 学習の誤差にモデルが到達するには何環境ステップ必要であり、モデルを 1 回解くごとに何回のバックアップを要するか。
   3. 価値反復法はモデルのバックアップを消費し、Q 学習は環境ステップを消費し、(b) の推定モデルは両方を消費する。1 環境ステップが b 回のバックアップと同じ計算コストを要するとする。測定結果を用いて、b のどの範囲で 3 つの手法のそれぞれが 4000 エピソード後の Q 学習の誤差に最小の総コストで到達するかを求めよ。遷移カーネルを読み取れないエージェントにも利用可能なまま残るのはどの手法か。

7. [code] **2 状態タスクにおける最大化バイアス。**●すべてのエピソードは状態 s_0 から始まり、そこには 2 つの行動がある。**右**は報酬 0 で終端し、**左**は状態 s_1 へ報酬 0 で遷移する。s_1 では m 個の行動のそれぞれが $\mathcal{N}(-0.1, 1)$ から抽出された報酬を与えて終端するため、**左**を選ぶのは誤りである。$\gamma = 1$、定数ステップサイズ $\alpha = 0.1$、初期値 0、および $\epsilon = 0.1$ の `epsilon_greedy` を用いよ。

   1. $m = 10$ に対して表形式の Q 学習を実装し、s_0 で**左**を選ぶ実行の割合（1000 回の実行・各 300 エピソードにわたる平均）をエピソード番号に対してプロットせよ。ϵ 貪

欲エージェントが Q^* に基づいて行動した場合、この割合はいくらになるか。

2. ダブル Q 学習 (Hasselt 2010) は 2 つのテーブル Q_1 と Q_2 を保持し、ϵ 貪欲に Q_1+Q_2 に基づいて行動し、各遷移で公平なコインを投げる。表が出れば次式で更新し、

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha(r + \gamma\, Q_2(s', \text{argmax}_{a'} Q_1(s',a')) - Q_1(s,a)), \tag{14.38}$$

裏が出れば 2 つのテーブルを入れ替えた同じ更新を適用する。この曲線をプロットに追加せよ。

3. 左を通じて到達しうる報酬はすべて負の平均を持つ。それにもかかわらず Q 学習による $Q(s_0, \text{left})$ の推定値が正になる仕組みを説明せよ。$m = 1$ と $m = 100$ に対して Q 学習の曲線がどう変化するかを予測し、実際に確認せよ。$m = 100$ における効果のうち、どの部分がノイズを含む推定値に由来し、どの部分が一度も試されていない行動に由来するか。

Richard S. Sutton および Andrew G. Barto の『Reinforcement Learning: An Introduction』Example 6.7 を改変。

8. [code] **崖歩きにおける SARSA と Q 学習。**●`CliffWalking-v1` では、エージェントは 4×12 のグリッドを左下の隅から右下の隅まで横断する。毎ステップ -1 のコスト（報酬）がかかり、下端に沿った崖へ踏み込むと -100 がかかって開始位置へ戻される。$\gamma = 1$、定数ステップサイズ $\alpha = 0.5$、定数 $\epsilon = 0.1$ を用いよ。

   1. `sarsa` を `q_learning` のコピーとして作成せよ。ただし、更新前に次の行動 a' を `d2l.epsilon_greedy` で選び、それをターゲット Equation 14.37 に使い、次のステップでその行動を実行するようにせよ。両手法を 10 シードで 500 エピソード訓練し、1 エピソードあたりの収益をプロットせよ。
   2. 各最終テーブルの貪欲方策を開始位置からたどれ。各手法はどの経路を学習し、その長さはいくらか。学習中により多くの報酬を集めるのはどちらの手法であり、2 つの問いの答えが異なるのはなぜか。
   3. 訓練中に ϵ を 0 まで減少させよ。学習される 2 つの経路はどうなり、極限で各手法はどの価値を学習するか。

Richard S. Sutton および Andrew G. Barto の『Reinforcement Learning: An Introduction』Example 6.6 を改変。

14.4 探索

お気に入りのレストランに足を運べば満足のいく食事ができるが、新しい店に入ればもっと良い食事に出会えるかもしれないし、期待外れに終わるかもしれない。しかし、お気に入りの店にばかり通い続けていては、それを確かめる機会は永遠に訪れない。選んでみて初めてその価値がわかる選択肢のなかから繰り返し選択する場面では、いつでも同じ問いが生じる。新しいものをどのくらいの頻度で試すべきか、そしていつ試すのをやめてよいのかという問いである。

William Thompson は 1933 年に医学の文脈でこの問題を提起した。これまでの患者の経過を踏まえたとき、次の患者には 2 つの治療法のどちらを施すべきかという問題である (Thompson 1933)。Herbert Robbins は 1952 年に、毎回どちらからサンプリングするかを選びながら 2 つの母集団から繰り返しサンプリングする問題としてこれを定式化し直した (Robbins 1952)。スロットマシンの俗称である片腕バンディットにちなんで、この問題はまもなく **2 腕バンディット**（two-armed bandit）と呼ばれるようになり、選択肢が 2 つより多い場合は**多腕バンディット問題**（multi-armed bandit）と呼ばれる。1985 年には Tze Leung Lai と Robbins が、学習のコストは試行回数とともに増大せざるを得ないものの、その試行回数の対数と同程度に小さく抑えられることを示した (Lai and Robbins 1985)。

14.3 節の Q 学習も同じ問いに直面し、素朴な方法でこれに答えていた。Q 学習が学習できるのは表のうち実際に試した要素だけであり、表で低く評価されている指示も試し続けるために、確率 ϵ でランダムに行動した。この規則には目に見えるコストがある。20 ステップに 1 回の割合でランダムな行動を強いられると、滑りやすい湖における最適方策（optimal policy）の成功率は 73% ではなく約 54% に落ちてしまう。これは試す量として適切なのだろうか。別の規則を使えば、取りこぼしをより少なく抑えつつ同程度のことを学習できるのではないだろうか。

この問いは、検証器（verifier）を用いて訓練される言語モデルにおいても再び重要になる。モデルはプロンプトに対する複数の回答をサンプリングし、判定器がそれらにスコアをつけ、モデルはより高いスコアを得た回答の方向へと移動する。もしすべての回答が同じスコアであれば向かうべき方向は存在せず、より良い回答を見つけ出す手段はモデル自身のランダム性だけになる。

まず、このトレードオフのコストを正確に測定できる単一状態の問題、すなわちバンディットにおいて検討する。続いて、未試行の対を試すのに長い行動の系列が必要になり得るマルコフ決定過程（MDP）に戻り、最後に 14.1 節の検証器タスクを扱う。

```
import gymnasium as gym
```

14.4.1 多腕バンディット問題

バンディットとは、状態を 1 つだけ持つ MDP である。各ラウンドでエージェントは複数の腕から 1 本を引いてランダムな報酬を受け取る。どの行動をとっても次に起こることは変化せず、報酬の遅延もないため、残る問いは「どの腕が最良か」という 1 点のみである。エージェントは実際に腕を引くことでしかこの問いに答えられず、劣った腕を 1 回引くことは、その 1 回分を最良の腕に費やせなかったことを意味する。

14.4.1.1 腕とリグレット

ここではベルヌーイ報酬を持つ 10 本の腕を用いる。腕 a は 1 を確率 μ_a で返し、それ以外のときは 0 を返す。最良の腕である 3 番目の腕は $\mu^* = \max_a \mu_a = 0.90$ であり、2 番目に良い腕は 0.60 である。エージェントは μ_a を知らない。$n(a)$ 回だけ腕 a を引いた時点で、エージェントは経験平均（empirical mean）$\hat{\mu}(a)$、すなわちその試行のうち 1 を返した割合を知る。**バンディット規則**（bandit rule）は、これらの集計値から次に引く腕を決める。関数 `bandit_run` は 1 つの規則を `num_pulls` ラウンド実行し、すべての乱数を乱数生成器 `rng` から引き出す。また、すべての腕について集計値 `wins` と `count` を保持し、それらを規則に渡す。

```
arms = np.array([0.50, 0.42, 0.90, 0.25, 0.55, 0.38, 0.60, 0.32, 0.50, 0.45])

def bandit_run(arms, rule, num_pulls, rng):
    """One state, one step: pull an arm, log the shortfall, update tallies.
    Returns the cumulative regret after each pull."""
    wins, count = np.zeros(len(arms)), np.zeros(len(arms))
    gap = np.empty(num_pulls)
    for t in range(num_pulls):
        a = rule(wins, count, t, rng)
        count[a] += 1
        wins[a] += float(rng.random() < arms[a])
        gap[t] = arms.max() - arms[a]
    return np.cumsum(gap)
```

この関数は、各試行の報酬ではなく不足分を記録する。成功率は方策（policy）の良さを表すが、学習器については学習にどれだけのコストがかかったかを知りたい。腕 a_u をラウンド u で引くと、期待値において最良の腕を $\mu^* - \mu_{a_u}$ だけ下回り、t 回引いた後の**リグレット**はこれらのギャップを足し合わせたものになる。

$$\mathrm{Regret}(t) = \sum_{u=1}^{t} \left(\mu^* - \mu_{a_u}\right) = \sum_{a} \Delta_a \, n_t(a), \qquad \Delta_a = \mu^* - \mu_a. \tag{14.39}$$

2 番目の式は試行を腕ごとにまとめたものであり、各腕のギャップ Δ_a にその腕を引いた回数 $n_t(a)$ を掛けている。リグレットとは、最初から最良の腕を知っていたエージェントと比べて失った量である。平均 0.50 の 1 本の腕をずっと引き続ける規則は、1 回あたり 0.40、合計では 800（2000 回引いた時点）を失い、そのリグレットは t に対して線形に増大する。規則のリグレットが t より緩やかに増大するならば、その規則が劣った腕を引く頻度は次第に減っていき、1 回あたりの平均リグレットは 0 に収束する。

14.4.1.2 貪欲規則とイプシロン貪欲規則

14.3 節の規則はここでもそのまま機能する。`d2l.epsilon_greedy(q, epsilon, rng)` は価値推定値の 1 行（ここでは 10 個の経験平均）を受け取る。確率 ϵ で一様ランダムに腕を 1 本選び、それ以外のときは推定値が最大の腕を選ぶ（同値の場合はランダムに選ぶ）。1 度も引かれていない腕の経験平均は 0 である。$\epsilon = 0$ のとき、この規則は**貪欲**（greedy）である。固定された ϵ は永遠に同じ割合で探索を続けるのに対し、**アニーリング**規則は時間とともに ϵ を下げていく。ここでは $\epsilon_t = \min(1, 20/t)$ をラウンド t で用いる。これは最初の 20 ラウンドではランダムに引き、その後は次第に頻度を下げていく。このスケジュールはラウンド数のみに依存し、エージェントが学習した内容には一切依存しない。

これらの規則はいずれも、価値推定値の 1 行を行動上の分布へと変換する。Figure 14.10 に、湖での 4 つの移動に対応する 4 つの行動価値の行に対する例を示す。貪欲規則は最大の推定値にすべての確率を割り当て、ϵ 貪欲規則は確率の割合 ϵ をすべての行動に一様に分配する。3 つ目の規則である**ボルツマン探索**（Boltzmann exploration）は、各行動の確率を $\exp(\hat{Q}(s,a)/\beta)$ に比例させる。これは 3.1 節のソフトマックスにソフトマックス温度 β を入れたものであり、β が大きければ一様分布に近づき、小さければ貪欲規則に近づく。ϵ 貪欲規則とは異なり、明らかに悪い行動よりも最良に近い行動をより頻繁に試す。

関数 `eps_rule` はスケジュール $t \mapsto \epsilon_t$ を、経験平均に `d2l.epsilon_greedy` を適用するバンディット規則へと変換する。関数 `regret_runs` は 20 回の独立した実行で規則を動かし、そのリグレット曲線を返す。異なる規則が互いに独立な実行になるよう、各実行はストリーム番号と実行インデックスから独自の乱数生成器のシードを設定する。関数 `report` は実行全体にわたる最終リグレットの中央値と四分位数を出力する。少数の実行が平均を大きく左右し得るため、ここでは中央値を報告する。

```
def eps_rule(schedule):
    """Epsilon-greedy on the empirical means, with epsilon = schedule(t)."""
    return lambda wins, count, t, rng: d2l.epsilon_greedy(
        wins / np.maximum(count, 1), schedule(t), rng)

def regret_runs(arms, rule, stream, num_runs=20, num_pulls=2000):
    """Regret curves of `num_runs` independent runs, shape [run, pull]."""
    return np.array([bandit_run(arms, rule, num_pulls,
                                np.random.default_rng([2026, stream, s]))
```

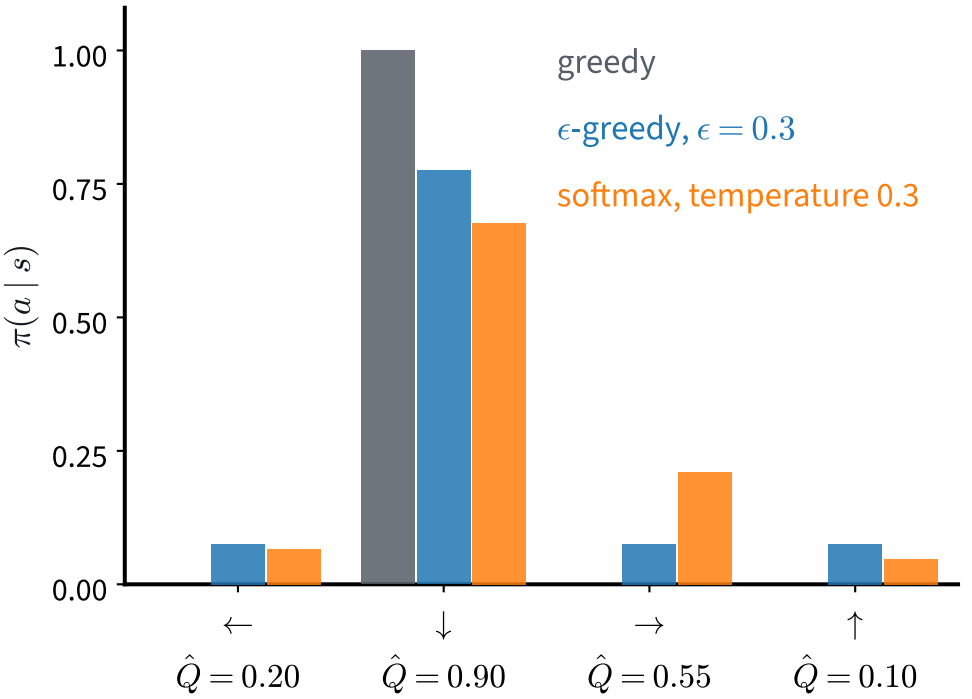


Figure 14.10: 湖での 4 つの移動に対する行動価値の 1 行 $\hat{Q}(s, \cdot) = (0.20, 0.90, 0.55, 0.10)$ に適用した 3 つの探索規則。貪欲規則は最大の値にすべての確率を置く。ϵ 貪欲規則は $\epsilon = 0.3$ のとき各行動に 0.075 を与え、残りを最大の値に割り当てる。温度 0.3 のソフトマックスは行動をその値によって順序づける。

```
                    for s in range(num_runs)])

def report(name, r):
    q25, q50, q75 = np.percentile(r[:, -1], [25, 50, 75])
    print(f'{name:>16}: regret after {r.shape[1]} pulls: median {q50:3.0f}'
          f' (quartiles {q25:.0f} to {q75:.0f})')

rules = {'greedy': eps_rule(lambda t: 0.0),
         'epsilon = 0.1': eps_rule(lambda t: 0.1),
         'epsilon = 20/t': eps_rule(lambda t: min(1.0, 20 / (t + 1)))}
regret = {name: regret_runs(arms, rule, i)
          for i, (name, rule) in enumerate(rules.items())}
for name, r in regret.items():
    report(name, r)
```

```
        greedy: regret after 2000 pulls: median 800 (quartiles 700 to 980)
  epsilon = 0.1: regret after 2000 pulls: median 106 (quartiles 92 to 132)
 epsilon = 20/t: regret after 2000 pulls: median  46 (quartiles 44 to 53)
```

貪欲規則の結果は最悪であり、その理由は正確に導き出せる。すべての経験平均は 0 から始まるため、最初の試行ではランダムな腕が引かれる。それが 0 を返せば、その腕の平均は 0 のままであり、10 本すべての腕が同値のまま並ぶ。もし 1 を返せば、その平均は正になり、その後も永遠に正のままである。1 度の成功によって成功割合は 0 より大きく保たれる一方、他のすべての腕は依然として 0 のままだからである。したがって貪欲規則は、どれか 1 本が報酬を返すまでランダムに腕を引き、それ以降は他の腕を決して引かない。これらのランダムな試行のそれぞれにおいて、腕 a は確率 $1/10$ で選ばれて 1 を確率 μ_a で返すため、最終的に落ち着く腕が a となる確率は $\mu_a / \sum_b \mu_b$ である。最良の腕がこのくじに当たるのは確率 $0.90/4.87$、すなわち約 5 回の実行に 1 回の割合である。腕 a に落ち着いた実行はほぼすべての試行で Δ_a を失い、合計で約 $2000\,\Delta_a$ を失うため、期待リグレットは次のようになる。

$$\mathbb{E}\big[\mathrm{Regret}(2000)\big] \approx 2000 \sum_a \frac{\mu_a}{\sum_b \mu_b} \Delta_a \approx 707.$$

1 回の実行で失う量は $2000\,\Delta_a$ のいずれかの値をとり、0（約 5 回の実行に 1 回）から 1300 までに分布する。800 前後となる中央値は、平均 0.50 の腕に落ち着いた実行に対応する。

$\epsilon = 0.1$ の固定規則はほぼすべての実行で最良の腕を見つけるが、探索を決してやめない。ランダムな試行は最良の腕を平均ギャップ $0.90 - 0.487 = 0.41$ だけ下回る（ここで 0.487 は 10 個の μ_a の平均である）。したがって探索だけで 1 回あたり $0.1 \times 0.41 \approx 0.04$、すなわち約 83（リグレットの中央値 106 のうち）のコストがかかり、このコストは縮小しない。固定された ϵ のリグレットは t に対して線形に増大する。

アニーリング規則がランダムに腕を引く回数は約 112 回（2000 ラウンド中）であり、最初に 20 回、その後に約 $20 \ln(2000/20) \approx 92$ 回である。1 回あたり 0.41 として、そのコストは 46 となり、リグレットの中央値である 46 と一致する。リグレットのほぼすべては探索に費やされており、その合間の貪欲な試行は最良の腕に向けられている。ラウンド t までのランダムな試行回数は $20 \ln t$ のように増大するため、リグレットも同様に増大する。すなわち、ϵ_t を $1/t$ に比例させると対数リグレットが得られる。ただし定数が重要である。定数が小さすぎると、最良の腕と 2 番目に良い腕を見分ける前に探索をやめてしまう。Auer et al. (2002) の解析によれば、m 本の腕を持ち最小ギャップが $\Delta_{\min}$ である任意のバンディットにおいて、$\epsilon_t = \min(1, c\,m/(\Delta_{\min}^2 t))$（任意の $c > 5$）とすれば対数リグレットが保証される。今回の腕では $\Delta_{\min} = 0.30$ であるため、

これはここで用いた $20/t$ のほぼ 30 倍にあたる。ここでの定数は今回の腕に合わせて選んだものであり、この理論保証の対象外である。

14.4.2 楽観主義と事後サンプリング

ϵ 貪欲規則は盲目的に探索する。ランダムな試行は 2 番目に良い腕と同じ確率で最悪の腕に向かってしまい、100 回引かれた腕も 2 回しか引かれていない腕と同じ確率で再び試される。より優れた規則は、平均がまだ不確実であり最良である可能性が残っている腕へと探索を振り向ける。意思決定を変え得るのはそのような試行だけだからである。

14.4.2.1 信頼上限

これを実現する 1 つの方法が楽観主義である。すなわち、各腕をデータが許容する範囲で最も良いものであるかのように扱う。未試行の腕はどのような値もあり得るため、まず試される。何度も引かれた腕は精度の高い推定値を持つため、その推定値が高い場合に限り再び引かれる。**信頼上限**（upper confidence bound、UCB）規則は、各経験平均にその不確実性を測るボーナスを加え、その和が最大となる腕を引く (Auer et al. 2002)。

$$a_t = \operatorname*{argmax}_a \left[\hat{\mu}(a) + \kappa \sqrt{\log t \,/\, n(a)} \right]. \tag{14.40}$$

ボーナスは $1/\sqrt{n(a)}$（$n(a)$ 個の報酬の平均における誤差が縮小する速さ）に従って縮小し、また t とともに緩やかに増大するため、腕 a が引かれていない間もどの腕も永久に見捨てられることはない。ある腕が引かれるのは、その平均が高いからか、あるいはその平均が不確実だからである。$n(a) = 0$ ではボーナスが定義されないため、規則を開始する前にすべての腕を 1 回ずつ引く。Figure 14.11 に、ボーナスと 1 回の実行におけるインデックスを示す。

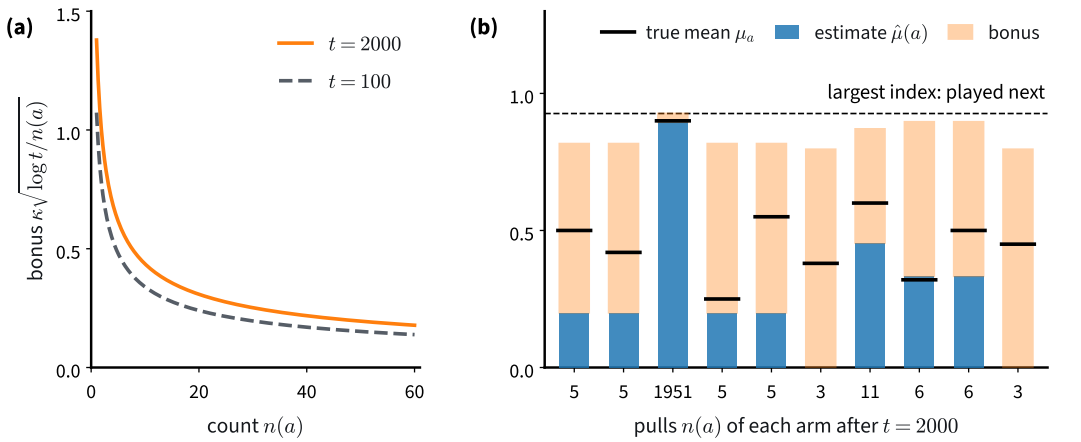


Figure 14.11: $\kappa = 0.5$ における不確実性のもとでの楽観主義。(a) ボーナス $\kappa\sqrt{\log t/n(a)}$ は腕の試行回数 $n(a)$ とともに縮小し、時刻 t とともに緩やかに増大する。(b) 10 本の腕に対して 2,000 回引いた後の UCB 規則の 1 回の実行。各棒は腕の推定値 $\hat{\mu}(a)$ の上にボーナスを積み上げたもので、その腕の試行回数が添えられており、黒い印は真の平均を表す。ほぼすべての試行はインデックスが最大の最良の腕に集まっているが、ほとんど試されていない腕もそれに近いインデックスを保っているため、$\log t$ が増大するにつれてそれぞれ再び試される。

ボーナスの大きさはヘフディングの不等式（Hoeffding's inequality）に由来する (Hoeffding 1963)。平均 $\hat{\mu}_n$（n 個の独立な $[0,1]$ 内の報酬の平均）が真の平均 μ を h より大きく上回る確率は高々 e^{-2nh^2} であり、下回る側についても同様である。$h = \kappa\sqrt{\log t/n}$ とおくと、真の平均がインデックスを上回る確率は高々 $t^{-2\kappa^2}$ となる（固定された回数 n に対して）。κ を大きくするとこのような失敗はまれになるが、悪い腕により多くの試行を費やすことになる。$\kappa = \sqrt{2}$ のとき、この規則は Auer et al. (2002) の UCB1 となり、そのリグレットは t に対して対数的に増大することが $[0,1]$ 内の報酬を持つあらゆるバンディットで成り立つ。同じことはすべての $\kappa > 1$ についても成り立ち (Bubeck and Cesa-Bianchi 2012)、演習問題 2 でこの上界を導出する。

14.4.2.2 トンプソンサンプリング

1933 年の Thompson の規則は、ボーナスを信念に置き換える。各腕についてその平均に関する事後分布（posterior distribution）を保持し、各事後分布からサンプルを 1 つずつ引き、サンプル値が最大となった腕を引く。報酬が $\{0,1\}$ の値をとり、$[0,1]$ 上の一様な事前分布（prior）を置く場合、w 回の成功（n 回の試行中）の後の事後分布はベータ分布 $\text{Beta}(1+w, 1+n-w)$ である。3 回の成功（4 回の試行中）を収めた腕の事後分布は $\text{Beta}(4,2)$ であり、平均は $2/3$ だが幅は広い。未試行の腕は一様な $\text{Beta}(1,1)$ を持ち、そのサンプル値が $\text{Beta}(4,2)$ からのサンプル値を上回る確率は $1/3$ である。Figure 14.12 にそのような 1 ラウンドを示す。このラウンドでは、未試行の腕が 18 回の成功（20 回の試行中）を収めた腕さえも上回っている。試行が積み重なるにつれて事後分布は狭まり、各腕はその腕が最良である事後確率（posterior probability）に従って引かれるようになる。トンプソンサンプリングもまた対数リグレットを達成し (Agrawal and Goyal 2012)、しかも選ぶべき定数を持たない。

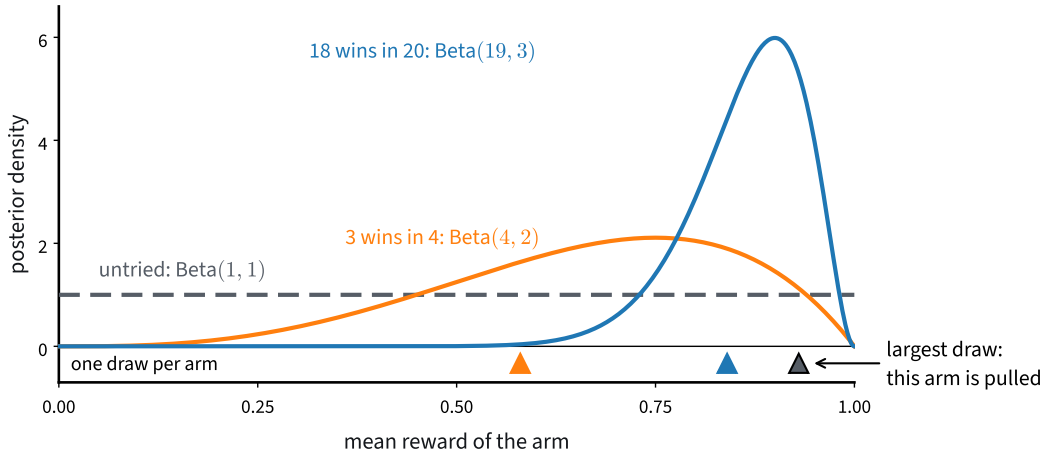


Figure 14.12: 3 本の腕（未試行の腕、3 回の成功を 4 回の試行で得た腕、18 回の成功を 20 回中で得た腕）に対するトンプソンサンプリングの 1 ラウンド。曲線は各腕の平均に対するベータ事後分布であり、腕が多く引かれるほど幅が狭くなる。規則は各事後分布から値を 1 つずつサンプリングし（三角形）、最大のサンプル値を持つ腕を引く。このラウンドでは未試行の腕が選ばれている。

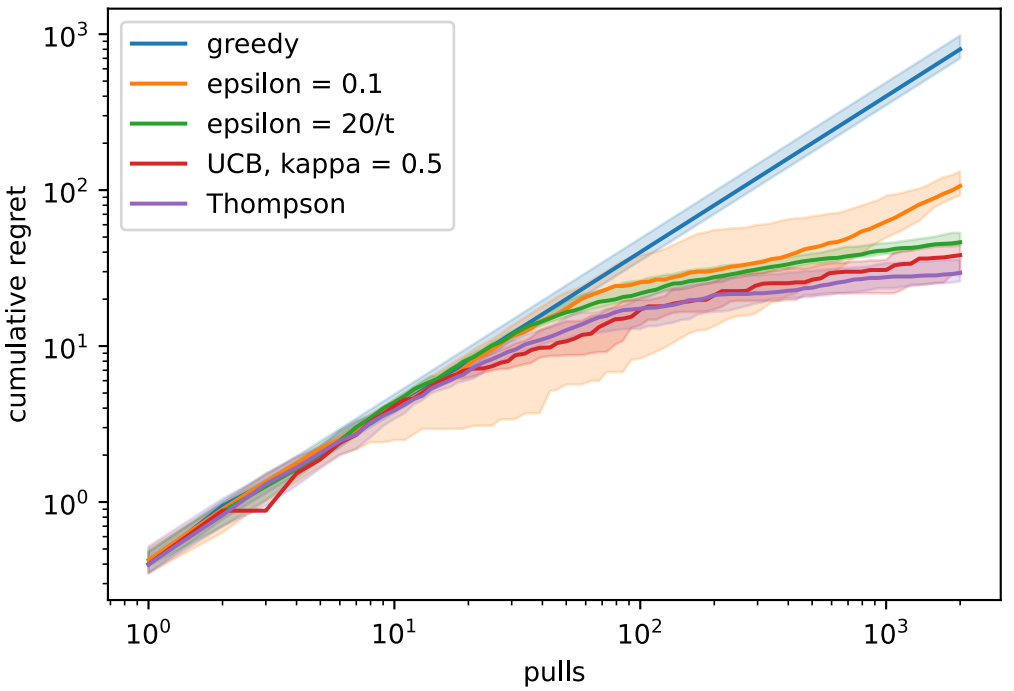
14.4.2.3 UCB とトンプソンサンプリングの比較

関数 `ucb(kappa)` は係数 κ を持つ UCB 規則を返し、`thompson` はベータ事後分布からサンプリングする。ここでは $\kappa = 0.5$（今回の腕に合わせて選んだ値であり、以下で検証する）の UCB とトンプソンサンプリングを実行し、5 つすべての規則をプロットする。`d2l.plot_curves` は実行全体に

わたる中央値と四分位数間の帯を描画する（ここでは対数軸上で等間隔にとった 100 個の試行時点において描画する）。

```
def ucb(kappa):
    """The UCB rule with coefficient kappa: each arm once, then the largest
    index mean + kappa * sqrt(log t / n)."""
    def rule(wins, count, t, rng):
        if (count == 0).any():          # play each arm once before pricing any
            return int(np.argmax(count == 0))
        bonus = kappa * np.sqrt(np.log(t + 1) / count)  # t + 1: round number
        return int(np.argmax(wins / count + bonus))
    return rule

def thompson(wins, count, t, rng):
    """Draw each arm's mean from its Beta posterior; play the largest draw."""
    return int(rng.beta(1 + wins, 1 + count - wins).argmax())

for i, (name, rule) in enumerate([('UCB, kappa = 0.5', ucb(0.5)),
                                  ('Thompson', thompson)], 3):
    regret[name] = regret_runs(arms, rule, i)
    report(name, regret[name])
pulls = np.unique(np.geomspace(1, 2000, 100).astype(int))  # even on a log axis
d2l.plot_curves({name: r[:, pulls - 1] for name, r in regret.items()},
                xlabel='pulls', ylabel='cumulative regret', x=pulls,
                xscale='log', yscale='log', band='iqr')
```

```
UCB, kappa = 0.5: regret after 2000 pulls: median  38 (quartiles 29 to 46)
        Thompson: regret after 2000 pulls: median  30 (quartiles 26 to 36)
```



どちらの規則も最終的なリグレットの中央値は約 30 から 40 となり、固定 ϵ の約 1/3 で、アニーリング規則よりも低い。両者は互いに近く、その差は実行間のばらつきよりも小さい。両対数軸上では、線形に増大するリグレットは傾き 1 の直線になる。貪欲規則と固定 ϵ はその傾きに近づく一方、UCB、トンプソンサンプリング、アニーリング規則はそこから折れ曲がって離れていく。2000 回の試行だけでは漸近的なレートを確定できないが、この曲がり方こそが対数的成長の現れである。

14.4.2.4 UCB の係数

$\kappa = 0.5$ という値は、理論が対象とする範囲 $\kappa > 1$ を下回っている。係数はどの程度影響するのだろうか。次のセルでは 3 つの係数のそれぞれについて 40 回の実行を試み、リグレットの中央値と最小値・最大値を出力する。

```
for j, kappa in enumerate([0.15, 0.5, np.sqrt(2)]):
    r = regret_runs(arms, ucb(kappa), 10 + j, num_runs=40)[:, -1]
    print(f'kappa = {kappa:.2f}: regret after 2000 pulls over 40 runs: median '
          f'{np.median(r):3.0f}, min {r.min():3.0f}, max {r.max():3.0f}')
```

```
kappa = 0.15: regret after 2000 pulls over 40 runs: median   8, min   4, max 709
kappa = 0.50: regret after 2000 pulls over 40 runs: median  38, min  21, max  59
kappa = 1.41: regret after 2000 pulls over 40 runs: median 204, min 173, max 241
```

2000 回の試行時点では、係数が結果を左右する。UCB1 ($\kappa = \sqrt{2}$) は 200 付近で終わり、$\kappa = 0.5$ のリグレットの 5〜6 倍になる。$\kappa = 0.15$ では中央値は約 10 と最小になるが、最大値が大きくなる。これほどボーナスが小さいと、最良の腕が最初の数回で不運にも失敗した実行では、劣った腕に落ち着いて 1 回あたり 0.3 以上を失い続けることがある。これは約 10 回の実行に 1 回の割合で起こり、中央値はこうした実行を隠してしまうが、平均は隠さない。

ホライズンが十分に長くなれば、理論値の κ が勝つと思うかもしれない。しかし今回の腕ではそうならない。10^5 回以上の試行（演習問題 4 で再現する）でも、$\kappa = \sqrt{2}$ のリグレットは $\kappa = 0.5$ の 6〜8 倍のままであり、どちらも $\log t$ のように増大する。この倍率はボーナスから説明できる。UCB は劣った腕 a を、そのボーナス $\kappa\sqrt{\log t / n(a)}$ がおよそ Δ_a に縮小するまで、すなわち約 $\kappa^2 \log t / \Delta_a^2$ 回引き続けるため、長い実行では 2 つのリグレットの比は $(\sqrt{2}/0.5)^2 = 8$ に近づく。理論が保証しているのは別のことである。$\kappa = \sqrt{2}$ は $[0, 1]$ 内の報酬を持つあらゆるバンディットで対数リグレットを達成するが、$\kappa = 0.5$ にはそのような保証がない。ヘフディングの不等式は最悪ケース、すなわち分散 $1/4$ の腕に対しても成り立たなければならない。ここでの最良の腕の分散は $0.9 \times 0.1 = 0.09$ であるため、その経験平均は不等式が想定するよりも正確であり、この問題ではより小さなボーナスで十分なのである。トンプソンサンプリングは、調整すべき定数なしに調整済みの UCB 規則に匹敵する。

14.4.2.5 リグレットの下界

リグレットはどこまで低く抑えられるのだろうか。Lai と Robbins は、ある族に属するすべてのバンディットでうまく機能する任意の規則は、t 回の試行のうちに各劣った腕 a を少なくともおよそ $\ln t / D(\mu_a \,\|\, \mu^*)$ 回は引かなければならないことを証明した (Lai and Robbins 1985)。ここで、次の式を考える。

$$D(p \,\|\, q) = p \ln \frac{p}{q} + (1 - p) \ln \frac{1 - p}{1 - q} \tag{14.41}$$

これは平均 p と q を持つベルヌーイ分布間の相対エントロピー（relative entropy）である。2 つの平均が近いほどこの値は小さくなるため、最良の腕に匹敵するほど良い腕は見分けにくく、より多くの回数引かなければならない。平均 0.60 の腕では $D(0.60 \,\|\, 0.90) = 0.31$ であり、約 $\ln 2000/0.31 \approx 24$ 回（ラウンド 2000 までに）引かなければならない。Equation 14.39 と合わせると、そのような規則のリグレットはいずれも少なくとも $\ln t$ のように増大する。UCB とトンプソンサンプリングはこのレートで増大するため、長い実行では定数倍を超えてこれらを上回る規則は存在しない。一方、リグレットが線形に増大する固定 ϵ はこのレートに到達しない。

14.4.3 MDP における探索

バンディットでは、どのラウンドでもすべての腕を引くことができる。しかし MDP では、状態行動対（state-action pair）はその状態からでなければ試すことができず、その状態に到達するには特定の行動の系列が必要になる場合がある。ランダムな行動が与えられた系列を生み出す確率は、その長さに対して幾何級数的に低下する。行動が 2 つの場合、長さ k の特定の行動系列が生じる確率は 2^{-k} となる。

14.4.3.1 状態のチェーン

チェーンを考えればこの現象を厳密に捉えられる。一列に並んだ N 個の状態があり、状態 0 から始まるものとする。行動 1 は 1 つ先の状態へ進み、行動 0 は開始状態へと戻り、最後の状態に入ると報酬 1 を得てエピソードが終了する（エピソードは $2N$ ステップで打ち切られる）。成功するには、エージェントは行動 1 を $N-1$ ステップ連続で選ばなければならない。1 度でも行動 0 を選ぶと開始状態に戻されてしまうからである（Figure 14.13）。

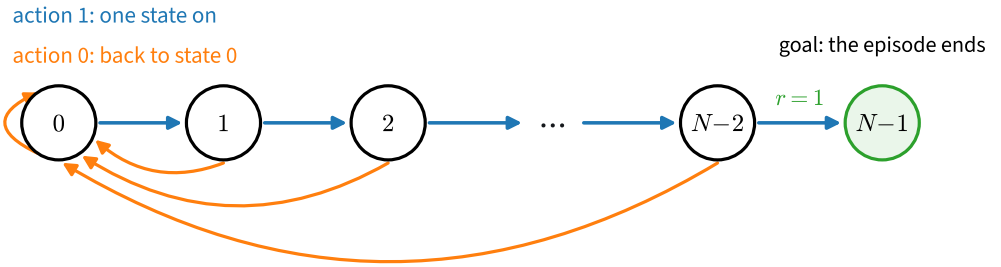


Figure 14.13: N 個の状態を持つチェーン。行動 1（青）は 1 つ先の状態へ進み、行動 0（オレンジ）はどこからでも状態 0 に戻り、唯一の報酬は最後の状態に入ったときの 1 であり、そこでエピソードが終了する。

関数 `chain` はこの MDP を `d2l.TabularMDP(P, r, gamma)`（14.1 節の配列による表現）として構築する。ここで `P[s, a]` は次の状態の分布であり、`r[s, a]` は期待報酬である。関数 `success_random` は、一様ランダムな行動が `horizon` ステップ以内に最後の状態へ到達する厳密な確率を計算する。この関数は 2 つの行動にわたって遷移カーネルを平均して $P_\pi(s' \mid s) = \frac{1}{2}\sum_a P(s' \mid s, a)$ とし、状態 0 からの状態分布を P_π によって 1 ステップずつ伝播させる。最後の状態は吸収状態（absorbing state）であるため、`horizon` ステップ後のその状態の確率が到達済みの確率となる。

```
def chain(N, gamma):
    """N states in a row: action 1 moves one state on, action 0 returns to
    the start; entering the last state pays 1 and ends the episode."""
    P, r = np.zeros((N, 2, N)), np.zeros((N, 2))
```

```
    P[:, 0, 0] = 1
    P[np.arange(N - 1), 1, np.arange(1, N)] = 1
    P[N - 1] = 0
    P[N - 1, :, N - 1] = 1              # the last state is absorbing
    r[N - 2, 1] = 1
    return d2l.TabularMDP(P, r, gamma)

def success_random(mdp, horizon):
    """Exact probability that uniformly random actions, started in state 0,
    reach the last state within `horizon` steps."""
    P_pi, d = mdp.P.mean(axis=1), np.eye(mdp.num_states)[0]
    for _ in range(horizon):
        d = d @ P_pi
    return d[-1]
```

`d2l.q_learning(seed, Q, visits, env, num_episodes, gamma, alpha, epsilon)`（14.3 節）は、表 `Q` を ϵ 貪欲な行動で収集した遷移からインプレースで更新し、`visits` に訪問回数を数え、各エピソードの収益（return）を生成する。この関数は `env.reset` と `env.step` を通じてのみ環境とやり取りし、クラス `ModelEnv` が `TabularMDP` にこれら 2 つのメソッドを与える。`reset` はエージェントを状態 0 に置き、`step` は `P[s, a]` から次の状態をサンプリングして、終端状態（terminal state）での終端と `max_steps` ステップ後の打ち切りを報告する。

```
class ModelEnv:
    """Gymnasium's reset/step on a TabularMDP: start in state 0, draw s'
    from P[s, a], end in a terminal state or after max_steps steps."""
    def __init__(self, mdp, terminal, max_steps):
        self.mdp, self.terminal, self.max_steps = mdp, terminal, max_steps
        self.rng = np.random.default_rng(0)

    def reset(self, seed=None):
        if seed is not None:
            self.rng = np.random.default_rng(seed)
        self.s, self.t = 0, 0
        return self.s, {}

    def step(self, a):
        s, self.t = self.s, self.t + 1
        self.s = int(self.rng.choice(self.mdp.num_states, p=self.mdp.P[s, a]))
        return (self.s, self.mdp.r[s, a], self.s in self.terminal,
                self.t >= self.max_steps, {})
```

ここでの学習器は 14.3 節の Q 学習に 1 つだけ変更を加えたものである。表のすべての要素をあらゆる収益の上界である 1 から開始し、エージェントは $\epsilon = 0$ で貪欲に行動する。これが**楽観的初期化**（optimistic initialization）であり、UCB の背後にある考え方と表を組み合わせたもの、すなわち未試行の対をあり得る最良の価値に見せる手法である。比較のため、次のセルではランダムな行動が最初に成功するまでの期待エピソード数（`success_random` で求めた確率の逆数）も出力する。ゼロの表から始める Q 学習は、どの行もすべて同値でランダムに選ばれるため、ϵ の値にかかわらず最初の成功まではこれとまったく同じように振る舞う（演習問題 1）。

```
def first_success(env, gamma, seed, max_episodes=100):
    """Episodes until greedy Q-learning from an optimistic table first
    reaches the goal (inf if it never does within max_episodes)."""
    # Optimistic: every entry starts at 1, an upper bound on every return
    Q = np.ones((env.mdp.num_states, env.mdp.num_actions))
    returns = d2l.q_learning(seed, Q, np.zeros_like(Q), env, max_episodes,
                             gamma, epsilon=lambda episode: 0.0)
    return next((k + 1 for k, ret in enumerate(returns) if ret > 0), np.inf)

for N in (4, 8, 12, 16, 20):
    mdp = chain(N, gamma=0.95)
    env = ModelEnv(mdp, terminal={N - 1}, max_steps=2 * N)
    first = [first_success(env, mdp.gamma, seed) for seed in range(20)]
    print(f'N = {N:2d}: episodes to the first success: random '
          f'{1 / success_random(mdp, 2 * N):5.0f} (mean), optimistic '
          f'{np.median(first):2.0f} (median, max {max(first):.0f})')
```

```
N =  4: episodes to the first success: random     2 (mean), optimistic  2 (median, max 2)
N =  8: episodes to the first success: random    23 (mean), optimistic  3 (median, max 4)
N = 12: episodes to the first success: random   273 (mean), optimistic  4 (median, max 5)
N = 16: episodes to the first success: random  3449 (mean), optimistic  6 (median, max 7)
N = 20: episodes to the first success: random 45590 (mean), optimistic  8 (median, max 10)
```

ランダムな行動では状態が 1 つ増えるごとに約 2 倍のエピソード数が必要になり、23（$N = 8$ のとき）、約 3,400（$N = 16$ のとき）、約 46,000（$N = 20$ のとき）となる。楽観的な学習器はわずかな回数で済み、約 8 回（$N = 20$ のとき）と、ほぼ N に比例して増える程度にとどまる。楽観主義が機能するのは、試した対に試行済みの印がつくからである。報酬を与えない対を 1 度試すと、その要素はターゲット $r + \gamma \max_{a'} Q(s', a')$（高々 $\gamma < 1$）に向かって移動するため、依然として 1 のままであるすべての未試行の要素を下回る。したがって貪欲規則はまだ試していない対を優先し、学習器は偶然に頼るのではなく体系的にチェーンをたどっていく。エピソード数は指数関数ではなく N の多項式オーダーで増大する。

14.4.3.2 カウントに基づくボーナス

楽観的初期化は、1 度目の試行で価値が下がるまでその対を探索する。確率的な MDP では 1 度の試行からわかることはほとんどなく、価値を知るには同じ対を繰り返し訪れる必要がある。**カウントに基づくボーナス**（count-based bonus）は、UCB 規則を MDP へと引き継ぐものである。各対の報酬に、訪問回数とともに $1/\sqrt{n(s,a)}$ のように縮小するボーナスを加え、価値バックアップ（value backup）によってそのボーナスをそこへ至る手前の状態へと伝播させることで、エージェントがあまり知らない対へ向かう経路をプランニングするようにする。モデルベース区間推定（model-based interval estimation）は学習されたモデルを用いてこれを実現し (Strehl and Littman 2008)、このようなボーナスを伴う Q 学習のリグレットは（MDP のサイズに関する多項式因子を除けば）ステップ数の平方根のように増大する (Jin et al. 2018)。表の代わりにネットワークを用いる場合、同じ状態を 2 度訪れることはまれであるため、実用的なエージェントでは訪問回数の代わりに学習された新規性の尺度（たとえば、状態の固定されたランダム関数を予測するよう訓練されたネットワークの誤差など）を用いる (Burda et al. 2019)。

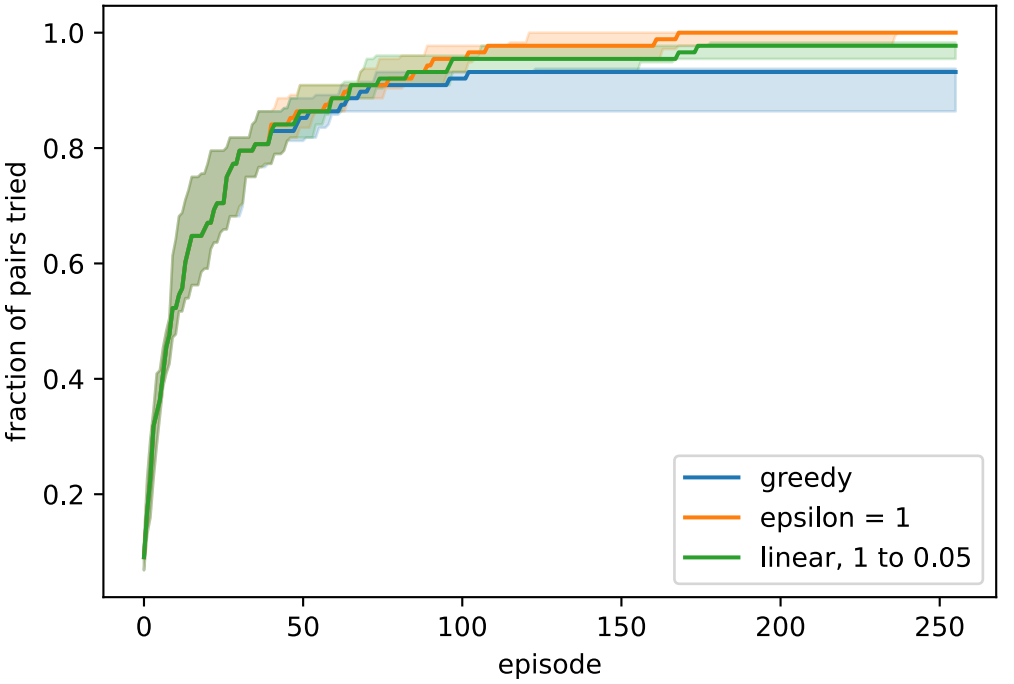
14.4.3.3 湖でのカバレッジ

湖のタスクにこうした手法は必要だろうか。14.3 節では、すべての移動が指示どおりに進む穏やかな湖での Q 学習において、ステップサイズ $\alpha = 0.9$ で 512 エピソードを実行し、得られた貪欲な表がこの湖を解いた。ここではその実験を 256 エピソードとし、3 つのスケジュールで繰

り返す。貪欲（$\epsilon = 0$）、常にランダム（$\epsilon = 1$）、そして `q_learning` のデフォルトである `d2l.linear_schedule(1.0, 0.05, 128)`（訓練の前半にわたって ϵ を 1 から 0.05 まで線形に下げる）の 3 つである。20 回の実行のそれぞれにおいて、次のセルは各エピソード後に 44 個の対（行動を選べる 11 マス、各 4 つの行動）のうち少なくとも 1 度試された割合を記録する。さらに、ゴールに到達した訓練エピソードの割合と、最終的な表の貪欲方策（greedy policy）がゴールに到達するかどうかも記録する（後者はこの決定論的な湖において `d2l.evaluate`（14.2 節）により 1 エピソードで検証する）。

```
def coverage_run(env, live, epsilon, seed, num_episodes=256, gamma=0.95):
    """One run of Q-learning with step size 0.9: the fraction of live pairs
    tried after each episode, the fraction of episodes won, and whether the
    final greedy policy reaches the goal."""
    Q = np.zeros((env.observation_space.n, env.action_space.n))
    visits, tried, returns = np.zeros_like(Q), [], []
    for ret in d2l.q_learning(seed, Q, visits, env, num_episodes, gamma,
                              alpha=lambda n: 0.9, epsilon=epsilon):
        tried.append((visits[live] > 0).mean())  # visits is updated in place
        returns.append(ret)
    solved = d2l.evaluate(env, lambda s, rng: int(Q[s].argmax()), 1)
    return tried, np.mean(returns), solved

calm = gym.make('FrozenLake-v1', is_slippery=False)
live = np.array([c not in b'HG' for c in calm.unwrapped.desc.flatten()])
schedules = {'greedy': lambda k: 0.0,
             'epsilon = 1': lambda k: 1.0,
             'linear, 1 to 0.05': d2l.linear_schedule(1.0, 0.05, 128)}
num_seeds, coverage = 20, {}
for name, epsilon in schedules.items():
    tried, won, solved = zip(*[coverage_run(calm, live, epsilon, seed)
                               for seed in range(num_seeds)])
    coverage[name] = np.array(tried)                  # (seed, episode)
    print(f'{name:>17}: pairs tried {np.median(coverage[name][:, -1]):.0%}, '
          f'episodes won {np.median(won):.0%} (medians); greedy policy '
          f'solves {np.sum(solved):.0f}/{num_seeds}')
d2l.plot_curves(coverage, xlabel='episode', ylabel='fraction of pairs tried',
                band='iqr')
```

```
          greedy: pairs tried 93%, episodes won 60% (medians); greedy policy solves 19/20
     epsilon = 1: pairs tried 100%, episodes won 2% (medians); greedy policy solves 19/20
linear, 1 to 0.05: pairs tried 98%, episodes won 58% (medians); greedy policy solves 19/20
```



穏やかな湖では、同値の場合のランダムな選択だけで貪欲規則に探索が生じる。10 個中約 9 個の対を試し、最終的な貪欲な表は他の 2 つのスケジュールの表と同様に、ほぼすべての実行で湖を解く。$\epsilon = 1$ では、エージェントはすべての対を試すものの、訓練エピソードのほとんどでゴールに到達しない。良い表を学習することと、学習中に報酬を集めることとは異なる目標であり、ϵ はこの 2 つの間でトレードオフをとる。3 つのスケジュールすべてを制約しているのは最初の成功である。それまではすべての要素が 0 であり、どのスケジュールもランダムに行動するため、同じシードであれば最初の成功はどのスケジュールでも同じエピソードで訪れる。ゴールは開始地点から 6 手の距離にあり、ランダムウォークでも数十エピソード以内に見つけられる。この湖は小さすぎてチェーンのような困難が現れないのである。一方、本節の冒頭で触れた滑りやすい氷の湖では、ある対を 1 度訪れただけではその対についてほとんど何もわからず、すべての対を 1 度ずつ試すだけでは不十分である（演習問題 1）。

14.4.4 検証器タスクにおける探索

14.1 節の検証器タスクには、4 つの算数のプロンプト x、それらすべてで共有される 9 個の候補応答 y、そして $r(x, y) = 1$（y がプロンプトの正解である場合）または 0（それ以外の場合）を返す検証器がある。参照方策（reference policy）π_{ref} はプロンプトごとに 1 行の選好度に対するソフトマックスであり、8 個の通常の候補に確率を均等に配分する。9 番目の候補（あらゆる答えを並べて保険をかけた回答であり、完全一致の検証器には決して受理されない）にはほとんど確率を与えない。$2 + 2$ と 3×4 の 2 つのプロンプトは候補のなかに正解を持つが、8×8 と 9×9 には一致する候補がない。サンプリングされた応答が正解となる確率は、プロンプト全体の平均で 0.062 である。

各ラウンドでプロンプトが到着し、1 つの応答にスコアがつけられ、エピソードが終了する。プロンプトはエージェントが観測できるものの影響を与えられない文脈であり、応答が腕にあたる。観測された文脈によって最良の腕が変わるバンディットは**文脈付きバンディット**（contextual bandit）と呼ばれる。エージェントはプロンプトごとに 1 つのバンディットに直面し、モデルを

用いることで学習した内容をプロンプト間で共有できる。

言語モデルは、このようなタスクをここまでの規則とは異なる方法で探索する。14.8 節で導出する一般的な訓練手法では、現在の方策からプロンプトに対する K 個の応答のグループをサンプリングしてスコアをつけ、グループ内の平均を上回るスコアを得た応答の確率を引き上げる一方、平均を下回る応答の確率を引き下げる。すべての応答が同じスコアになるグループでは、平均を上回るものも下回るものもないため、何も変化しない。方策があるプロンプトに確率 $p = \pi(\text{correct} \mid x)$ で正解するとき、K 個の独立な応答からなるグループですべてのスコアが同一になる確率は次のようになる。

$$(1-p)^K + p^K, \tag{14.42}$$

これは K 個すべてが不正解となる確率と、すべてが正解となる確率の和である。次のセルではプロンプトごとに 1,000 個のグループをサンプリングし、スコアがすべて同一になるグループの割合をこの式と比較する。比較は参照方策と、回答可能な 2 つのプロンプトをほぼ解いた方策（訓練された方策ではなく、各正解の選好度を 5 だけ引き上げて手作業で作ったもの）の双方で行う。配列 `R[x, y]` はすべての対に対する検証器のスコアを保持する。

```
task = d2l.VerifierTask()
R, rng = task.R, np.random.default_rng(0)     # R[x, y] = r(x, y)
for name, pi in [('reference', task.pi_ref),
                 ('solved', task.policy(task.theta_ref + 5 * R))]:
    p = (pi * R).sum(axis=1)            # chance that one response is right
    for K in (2, 8, 32):
        same = []                       # per prompt: all K rewards equal
        for x in range(len(task.prompts)):
            ys = rng.choice(len(task.resp), (1000, K), p=pi[x])
            same.append((R[x, ys] == R[x, ys[:, :1]]).all(axis=1).mean())
        print(f'{name:>9}, K = {K:2d}: identical {np.round(same, 2)}, '
              f'mean {np.mean(same):.2f}, '
              f'exact {np.mean((1 - p) ** K + p ** K):.2f}')
```

```
reference, K =  2: identical [0.78 0.78 1.   1.  ], mean 0.89, exact 0.89
reference, K =  8: identical [0.35 0.33 1.   1.  ], mean 0.67, exact 0.67
reference, K = 32: identical [0.01 0.02 1.   1.  ], mean 0.51, exact 0.51
   solved, K =  2: identical [0.89 0.93 1.   1.  ], mean 0.96, exact 0.96
   solved, K =  8: identical [0.67 0.7  1.   1.  ], mean 0.84, exact 0.85
   solved, K = 32: identical [0.21 0.21 1.   1.  ], mean 0.60, exact 0.61
```

サンプリングされた割合は式と一致する。どの K においても全グループの半分は同一になる。正解候補を持たない 2 つのプロンプトではすべての応答が失敗し、いくらサンプリングしても方策が生成できない答えは見つからないからである。回答可能な 2 つのプロンプトでは参照方策が確率 $p \approx 0.125$ で正解し、グループが大きいほど混在しやすくなる。サイズ 2 のグループでは約 78% が同一になり、サイズ 8 では 1/3、サイズ 32 ではほぼゼロになる。プロンプトがほぼ解けるようになるとグループは再び同一になりやすくなり、約 2/3 が $K = 8$ で同一となる。今度はほぼすべての応答が正解になるためであり、そのプロンプトでの学習は鈍化する。

この学習器は自身のサンプリングを通じてのみ探索する。グループが少なくとも 1 つの正解応答を含む確率 $1-(1-p)^K$ は、コードや推論のモデルを評価するのに用いられる **pass@K** スコアである (M. Chen et al. 2021)。方策がたいてい失敗するプロンプトにおいては、この値はグループから何かを学習できる確率にほぼ等しい。より高いソフトマックス温度でサンプリングすれば、より多くの応答に確率が広がり、方策がほぼ解いたプロンプトにおいてスコアが混在するグルー

プが増えるが、その代償として不正解も増える。正解の候補が存在しないプロンプトでは、これも助けにならない。

本節のすべての規則は、行動を実際に試すことでどの行動が良いかを学習するため、その試行には劣った腕を引くこと、ランダムウォークのエピソードを重ねること、あるいは何も教えてくれないグループを引くことといったコストが伴う。チェーンでは未試行の対に到達すること自体が難しく、検証器タスクでは方策が決して生成しない正解は永遠に見つからない。良い行動を手本として示すエキスパートがいればエージェントはこれらすべてを免れることができ、14.5 節ではそのような実演からの学習を扱う。

14.4.5 まとめ

自身の行動から学習するエージェントは、一見劣って見える行動も本当にそうであるかを確かめるために試さなければならず、リグレットは最初から最良の行動を知っていたエージェントと比べてこれにかかるコストを測る。10 本腕のバンディットでは、貪欲規則は最初に報酬を出した腕に落ち着いてしまい、固定された ϵ は永遠に 1 回あたり同じ量を失い続ける。ϵ を $1/t$ のように減衰させる規則、UCB、およびトンプソンサンプリングは、リグレットの増大を試行回数の対数程度に抑える。Lai と Robbins は、これが達成可能な最良のレートであることを示した。与えられた問題における UCB のリグレットはその係数によって決まり、理論上の係数はあらゆるバンディットに対して安全である反面、今回の問題では遅くなる。MDP では、ランダムな行動が必要な行動系列を見つけ出す確率はその長さに対して幾何級数的に低下するが、楽観的な価値を与えればエージェントは自ら未試行の対を探しに行く。検証器がサンプリングされた応答のグループにスコアをつける場合、すべて同じスコアになったグループは何の信号も持たず、方策自身のサンプリングが唯一の探索手段となる。

14.4.6 演習問題

1. [code] **探索スケジュール**。●●ゼロで初期化した表において、`epsilon_greedy` は同値の場合にすべて一様ランダムに選び、FrozenLake の報酬はすべて非負である。
    1. 最初の成功エピソードまでは表のすべての要素が厳密に 0 のままであり、それまではどの ϵ でも同じ振る舞いが生じることを示せ。このことから、ϵ を選ぶ基準としての最初の成功エピソードについて何が言えるか。
    2. 最初の成功の後、どの要素が正になるか。1 度正になった要素はその後も正のままであること、および貪欲規則（$\epsilon = 0$）は正の要素を含む行のゼロ要素を二度と試さないことを示せ。本節のカバレッジの実験では、この規則が穏やかな湖で 10 個中約 9 個の対を試し、そこで成功するよう学習することが示されている。滑りやすい氷の上でも同様に良い方策を学習するかどうか予測せよ。
    3. オプション引数 `epsilon`（`q_learning` の引数）はエピソードのインデックスから探索率への関数であり、`linear_schedule` が返す関数などがこれにあたる。滑りやすい湖において、4000 エピソード（14.3 節の設定）とし、ϵ を 0、0.1、0.5、1 に固定した場合、およびスケジュール `linear_schedule(1.0, 0.05, num_episodes // 2)`（`q_learning` が 14.3 節で用いているもの）の場合について、それぞれ 5 つのシードで実行せよ。各設定について、平均訓練収益、`d2l.evaluate` による 1000 エピソードにわたる最終的な貪欲方策の成功率、試された対の割合、および $\max_s \left| \max_a \hat{Q}(s,a) - V^*(s) \right|$ を報告せよ。予測は正しかったか。
    4. 各設定を訓練収益および貪欲方策の成功率によって順位づけせよ。2 つのランキングはどこで食い違い、その食い違いは何によって説明されるか。
2. **UCB の対数リグレット**。●●$\kappa = \sqrt{2}$ のとき、規則 Equation 14.40 は Auer et al. (2002) の UCB1 であり、その定理 1 は t 回引いた後の期待リグレット（$[0,1]$ 内の報酬を持つ腕に

対するもの）を次式で上から抑える。

$$\sum_{a:\,\Delta_a>0} \frac{8\ln t}{\Delta_a} + \left(1+\frac{\pi^2}{3}\right)\sum_a \Delta_a, \qquad \Delta_a = \mu^* - \mu_a. \tag{14.43}$$

この上界を導出せよ。用いる道具はヘフディングの不等式である。すなわち、平均 $\hat{\mu}_n$（n 個の独立な $[0,1]$ 内の報酬の平均であり、真の平均を μ とする）は次を満たし、

$$P(\hat{\mu}_n \geq \mu + h) \leq e^{-2nh^2}, \tag{14.44}$$

下側の裾についても同様の不等式が成り立つ。

1. 準最適な腕 a を固定し、$n_0 = \lceil 8\ln t/\Delta_a^2 \rceil$ とおく。腕 a がステップ $u \leq t$ において $n(a) \geq n_0$ のもとで引かれたならば、最良の腕のインデックスが μ^* 以下にあるか、あるいは腕 a の経験平均が μ_a をそのボーナスより大きく上回っているかのいずれかであることを示せ。
2. 両方の腕の試行回数を固定したとき、ステップ u におけるこれら 2 つの事象それぞれの確率を u^{-4} で上から抑えよ。回数に関するユニオンバウンドをとり、u について和をとることで、$\mathbb{E}[n(a)] \leq n_0 + \pi^2/3$ が t 回の試行後に成り立つことを示せ。
3. 小問 (b) とリグレットの分解 Equation 14.39 を組み合わせて定理を得よ。`arms` に対して $t = 2000$ でこの上界を評価し、本節の係数の比較実験で出力される中央値リグレット（200 付近、$\kappa = \sqrt{2}$ のとき）と比較せよ。証明のどのステップで最も大きく評価が緩んでいるか。
4. 一般の係数 κ について同じ議論を繰り返せ。どのような κ に対して小問 (b) のステップに関する和が収束するか。また、その答えは本節で選んだ $\kappa = 0.5$ について何を示唆するか。（回数に関するユニオンバウンドは緩い。ピーリング法による議論 (Bubeck and Cesa-Bianchi 2012) を用いればすべての $\kappa > 1$ に到達できる。）

3. **Thompson の事後分布。**●トンプソンサンプリングは、各腕の平均を $\text{Beta}(1+w, 1+n-w)$（w 回の成功が n 回の試行で得られた後）からサンプリングする。

1. これが $[0,1]$ 上の一様事前分布のもとでのベルヌーイ平均の事後分布であることを示せ。
2. 腕 A が 3 回の成功を 4 回の試行で収めており、腕 B は未試行であるとする。トンプソンサンプリングが B を引く確率は $1/3$ であることを示せ。より一般に、未試行の腕と比べるとき、未試行の腕を引く確率はもう一方の腕の事後平均を 1 から引いた値になることを示せ。
3. データが与えられたとき、トンプソンサンプリングは各腕を、その腕が最大の平均を持つ事後確率に等しい確率で引くことを示せ。
4. [code] 一様事前分布を、腕が悪いと予想する $\text{Beta}(1,9)$ と、腕が良いと予想する $\text{Beta}(9,1)$ にそれぞれ置き換えよ。2000 回引いた後のリグレットの中央値はどのように変化するか。2 つの事前分布のうち、楽観的初期化のように振る舞うのはどちらか。また、それはなぜか。

4. [code] **長いホライズン。**●●`bandit_run` は 1 度に 1 回の実行しか動かさない。

1. 多数の実行を同時に動かす版を作成せよ。すなわち、UCB とトンプソンサンプリングについて、`wins` と `count` を形状 (runs, arms) の配列として保持し、すべての実行の腕を 1 ステップで選ぶようにせよ。2000 回引いた時点でのその中央値が、実行間のばらつきの範囲内で本節の値と一致することを確かめよ。
2. 200 回の実行を、$\kappa \in \{0.15, 0.3, 0.5, \sqrt{2}\}$ の UCB およびトンプソンサンプリングについて 10^5 回の試行まで行い、リグレットの中央値を両対数軸上にプロットせよ。$\kappa = \sqrt{2}$ は $\kappa = 0.5$ に追いつくか。

3. $\kappa = 0.15$ について、中央値に加えてリグレットの平均もプロットせよ。なぜ両者は異なるレートで増大するのか。また、この規則を 1 度だけ実行するユーザーにとってのリスクを表すのは 2 つのうちどちらか。

5. [extended] **深い探索**。●●本節のチェーンを考える。状態は $0, \ldots, N-1$、開始状態は 0、行動 0 は状態 0 に戻り、行動 1 は状態を 1 つ進め、状態 $N-1$ に到達すると報酬 1 を得て終了し、エピソードは $2N$ ステップで打ち切られる。
   1. 一様ランダム方策が 1 エピソード内で成功する確率は高々 $2N \cdot 2^{-(N-1)}$ であり、したがって最初の成功までの期待エピソード数は少なくとも $2^{N-2}/N$ であることを示し、この下界をチェーンのセルが出力する厳密値と比較せよ。小問 (a)（演習問題 1）により、ゼロの表から始める ϵ 貪欲 Q 学習は最初の成功まで一様ランダムに振る舞うため、この下界は ϵ の値にかかわらず成り立つ。
   2. $N = 4$ のチェーンにおいて、楽観的な貪欲学習器の最初の数エピソードの推移をたどれ。各エピソードはどの要素を引き下げるか。そのパターンを用いて、必要となるエピソード数が N に対して高々多項式オーダーでしか増大しないことを論じよ。
   3. 両方の学習器を、100 ステップで打ち切られる `gym.make('FrozenLake-v1', map_name='8x8', is_slippery=True)` に移せ。楽観的な貪欲学習器と、スケジュール `linear_schedule(1.0, 0.05, num_episodes // 2)` のもとでのゼロ初期化学習器を、4000 エピソードおよび 20000 エピソードのそれぞれで各 5 シードずつ比較せよ。`d2l.evaluate` による貪欲方策の成功率、使用した環境ステップ数、および基準として価値反復法で計算した最適方策の成功率を報告せよ。確率的なマップでも楽観主義はその優位性を保つか。また、そのコストは何か。

   この課題には CPU 時間で数分かかる。

6. **何も教えてくれないグループ**。●方策が確率 p で正解するプロンプトにおいて、K 個の独立な応答からなるグループがすべて同一のスコアになる確率は $(1-p)^K + p^K$ である。
   1. 固定された $K \geq 2$ に対し、スコアが混在するグループが最も生じやすくなる p はいくらか。また、そのときの確率はいくらか。
   2. 小さい p に対し、グループに混在が生じる確率を少なくとも $1/2$ にする最小の K がおよそ $\ln 2/p$ であることを示せ。検証器タスクの参照方策（$p \approx 0.125$）および $p = 0.01$ についてこの値を計算せよ。
   3. ある訓練セットのプロンプトが持つ成功確率 p が $[0, 1]$ 上に一様に分布しているとする。サイズ $K = 8$ のグループのうちどれだけの割合が同一になり、どのプロンプトがそれらを生み出すか。
   4. `VerifierTask` の正解候補を持たない 2 つのプロンプトでは、どのような K やサンプリング温度に対してもすべてのグループが同一になる。これらのプロンプトで学習できるようにするための変更（タスクまたは学習器に対するもの）を 2 つ挙げ、それぞれが検証器単独では得られない何を必要とするかを述べよ。

14.5 デモンストレーションからの学習

「基礎」パートのすべてのモデルは、画像とそのクラス、文とその次の単語といったラベル付きの例から学習した。前節では、誰もラベルを与えてくれず、悪い行動を試しながら良い行動を見つけなければならないときに、エージェントがどれほど多くを失うかを見た。しかし多くの場合、誰かが手本を示すことができる。熟練した運転者が何を見て何をするか、すなわち各瞬間のカメラ画像と、それに応じて選ばれた操舵角を記録するとしよう。Fashion-MNIST で分類器を訓練したのと同じように、画像を入力、操舵角をラベルとして、これらの対から運転方策を訓練できるだろうか。

Dean Pomerleau は 1980 年代後半にカーネギーメロン大学でそのようなネットワークを構築した。1 つの隠れ層を持つネットワーク ALVINN は、道路の画像を操舵方向へ写像してバンを道路に沿って走らせた (Pomerleau 1989)。その後のバージョンでは、人間が運転する様子を観察することで、わずか数分で運転を学習した (Pomerleau 1991)。この手法は現在、**行動クローニング**（behavior cloning）と呼ばれている。環境のモデルも報酬も必要とせず、選択を記録できるエキスパートさえいればよい。これは今日でもエキスパートから学習する最も一般的な方法である。たとえば、厳選された応答で言語モデルをファインチューニングすることは、そこまでのテキストを状態、次のトークンを行動とする行動クローニングにほかならない。

Pomerleau はその弱点も見抜いていた。運転者がバンを車線の中央に保ち続けたため、記録には路肩の景色も、車線へ戻るための操舵の例も含まれていなかったのである。ひとたびネットワークが車線から逸れると、一度も見たことのない画像に直面することになったため、Pomerleau は視点をずらした道路画像を合成し、バンを元の車線へ戻す操舵角をラベルとして付与しなければならなかった。分類器は訓練時と同様の入力でテストされるが、方策は次に目にする入力を自ら選び出す。本節では、この違いが湖の環境でどれほどのコストをもたらすかを測定し、タスクが長くなるにつれてそのコストがどのように増大し得るかを示した上で、学習者が自力で到達した状態にエキスパートがラベルを付けるようデータの集め方を変える。

```
import gymnasium as gym
```

14.5.1 行動クローニング

デモンストレーションとは、エキスパートから記録された軌跡である。行動クローニングは、エキスパートが訪問した状態とそこで取った行動だけを保持し、報酬も遷移モデルも使わない。湖の環境では、エキスパートは 14.2 節で価値反復法が計算する最適方策 π^* であり、したがって本節では π^* が最適方策とエキスパートの両方を表す。ここでは、どのコマンドもエージェントを意図した方向へ移動させる滑らない氷を用いる。このときエキスパートは、毎エピソードでスタートからゴールまで同じ 6 手の経路をたどり、その訪問状態は Pomerleau の記録における車線中央のように、湖を貫く細いチューブをなす。湖のモデルは既知であるため、あらゆる成功率を厳密に計算し、サンプリングノイズなしで学習者とエキスパートを比較できる。

14.5.1.1 分類への帰着

次のセルでは、滑らない湖を `TabularMDP`（配列 P と r、および 14.1 節の割引率 γ）として構築し、`d2l.value_iteration` で解いた上で、V^* にベルマン作用素を適用した `mdp.backup` から各状態の最良の行動を選ぶことでエキスパートを読み取る。関数 `demonstrations` は環境内でエキスパートをロールアウトし、観測した状態と取った行動だけを記録する。

```
gamma = 0.95
env = gym.make('FrozenLake-v1', is_slippery=False)
mdp = d2l.TabularMDP.from_gym(env, gamma)
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
pi_star = mdp.backup(V_star).argmax(axis=1)

def demonstrations(env, expert, num_episodes):
    """Roll the expert; record only what it saw and what it did."""
    states, actions = [], []
    for _ in range(num_episodes):
        s, done = env.reset()[0], False
        while not done:
```

```
            states.append(s)
            actions.append(int(expert[s]))
            s, _, terminated, truncated, _ = env.step(actions[-1])
            done = terminated or truncated
    return np.array(states), np.array(actions)

demo_s, demo_a = demonstrations(env, pi_star, 3)
print(f'{demo_s.size} state-action pairs from 3 expert episodes, '
      f'covering states {np.unique(demo_s)}')
```

```
18 state-action pairs from 3 expert episodes, covering states [ 0  4  8  9 13 14]
```

滑らない氷の上ではどのエキスパートのエピソードも**下**、**下**、**右**、**下**、**右**、**右**とまったく同じになるため、3 エピソードからは同じ 6 個の対が 3 回繰り返された計 18 個のラベル付き対が得られる。状態 0 と状態 9 では**下**と**右**が同じ最短手数になり、同点の場合は**下**が選ばれる。エキスパートは氷のセル 1、2、3、6、10 を一度も訪問しない。

行動クローニングは、3.1 節が画像とラベルを扱ったのと同じようにこれらの対を扱う。すなわち、状態を入力、エキスパートの行動をクラスとし、条件付き分布 $\pi_\theta(a \mid s)$（パラメータ θ）を、対数尤度 $\sum_i \log \pi_\theta(a_i \mid s_i)$ の最大化、言い換えれば交差エントロピーの最小化によって当てはめる。必要なのは、パラメータを持つ方策と、それに対して勾配ステップを実行する仕組みだけである。

14.5.1.2 方策コンテナ

`Policy` は、状態を各行動につき 1 つのスコアへ写像するネットワーク `policy`（これにより $\pi_\theta(\cdot \mid s)$ は s におけるスコアのソフトマックスとなる）と、そのパラメータ用の Adam オプティマイザ `opt_pi` を保持する。`log_prob` は状態のバッチとそこで取られた行動に対して、行動クローニングが最大化する量である $\log \pi_\theta(a \mid s)$ を返す。

```
class seeded:
    """In a `with seeded(seed):` block, PyTorch's CPU generator starts from
    seed; on exit it resumes its previous state, so the seed affects only
    the draws inside the block. seed=None draws from the global stream."""
    def __init__(self, seed):
        self.seed = seed

    def __enter__(self):
        if self.seed is not None:
            self.state = torch.get_rng_state()
            torch.default_generator.manual_seed(self.seed)

    def __exit__(self, *exc):
        if self.seed is not None:
            torch.set_rng_state(self.state)

class Policy(nn.Module):
    """A policy network and its optimizer; pi(.|s) is the softmax of the
    network's outputs at s."""
    def __init__(self, policy, lr=1e-2):
        super().__init__()
        self.policy = policy
```

```
        self.opt_pi = torch.optim.Adam(policy.parameters(), lr=lr)

    def log_prob(self, obs, act):
        """log pi(a|s) for a batch of states and the actions taken there."""
        return torch.log_softmax(self.policy(obs), dim=-1) \
                    .gather(-1, act[:, None]).squeeze(-1)

    @classmethod
    def tabular(cls, num_states, num_actions, lr=0.1, seed=None):
        """One preference theta_{s,a} per state-action pair: an embedding."""
        with seeded(seed):
            policy = nn.Embedding(num_states, num_actions)
        nn.init.zeros_(policy.weight)
        return cls(policy, lr)
```

コンストラクタ `tabular` はソフトマックス回帰モデルである。埋め込みテーブルが状態行動対ごとに 1 つの選好 $\theta_{s,a}$ を保持し、整数で表された状態がその行を指定し、ソフトマックスによって $\pi_\theta(\cdot \mid s)$ が得られる。ゼロ初期化により、初期方策は一様分布になる。

引数 `seed` は、このコンストラクタおよび以降のコンストラクタが構築するネットワークのランダム初期化を固定する。`seeded` は PyTorch の CPU 乱数生成器を `seed` から開始させる処理を `with` ブロック内で行い、終了後にその乱数生成器の状態を復元するため、シードがコンストラクタ外の乱数抽出に影響することはない。

環境は NumPy の整数や配列をやり取りするため、3 つのメソッドがその境界で型変換を担う。`act` は呼び出し側から渡された NumPy の乱数生成器を用いて $\pi_\theta(\cdot \mid s)$ から行動をサンプリングするため、環境へ伝わる乱数性はすべて単一の明示的な発生源に集約される。このメソッドは、サンプリングしたエピソードの収益を平均する 14.2 節の `d2l.evaluate` の行動プロトコルに従う。`act_greedy` は最も確率の高い行動を返し、`log_prob_np` は対数確率を NumPy 配列として返す。

```
@d2l.add_to_class(Policy)
def act(self, obs, rng):
    """Sample an action; numpy in, int out, the acting protocol of evaluate."""
    with torch.inference_mode():
        probs = torch.softmax(self.policy(torch.as_tensor(obs)), -1).numpy()
    return int(rng.choice(len(probs), p=probs))

@d2l.add_to_class(Policy)
def act_greedy(self, obs, rng=None):
    """The most probable action; the same protocol as act."""
    with torch.inference_mode():
        return int(self.policy(torch.as_tensor(obs)).argmax())

@d2l.add_to_class(Policy)
def log_prob_np(self, obs, act):
    """log pi(a|s) for numpy batches of states and actions, as numpy."""
    with torch.inference_mode():
        return self.log_prob(torch.as_tensor(obs),
                             torch.as_tensor(act)).numpy()
```

これら 3 つのメソッドは、計算結果を一切微分しないため、`torch.inference_mode` の下で実行される。

状態集合が小さい方策に対しては、本章で行う厳密な計算のために、probs がテーブル全体 $\pi_\theta(a \mid s)$ を配列として学習者から読み出す。

```
@d2l.add_to_class(Policy)
def probs(self, num_states, num_actions):
    """pi(a|s) for every pair, as a (num_states, num_actions) numpy array."""
    s, a = np.divmod(np.arange(num_states * num_actions), num_actions)
    return np.exp(self.log_prob_np(s, a)).reshape(num_states, num_actions)
```

14.5.1.3 クローンの当てはめ

クローンの当てはめとは、$\frac{1}{n}\sum_i w_i \log \pi_\theta(a_i \mid s_i)$ においてすべての重み w_i を 1 とした勾配上昇法（gradient ascent）である。policy_step は、状態・行動・重みのバッチに対して、この目的関数に関するオプティマイザの更新を 1 ステップ実行する。

```
def policy_step(ac, obs, act, weight):
    """One ascent step on (1/n) sum_t w_t log pi(a_t|s_t) over a batch of
    n states and actions; the weights w arrive as numpy, that is, as data."""
    w = torch.as_tensor(weight)
    loss = -(w * ac.log_prob(torch.as_tensor(obs),
                             torch.as_tensor(act))).mean()
    ac.opt_pi.zero_grad()
    loss.backward()
    ac.opt_pi.step()
    return loss.item()
```

重みは NumPy 配列、すなわちデータとして渡されるため、そこに勾配は流れない。行動クローニングでは、重みはすべて 1 である。

当てはめはいつまで続けるべきだろうか。収束するまで訓練すれば、状態ごとに 1 行を持つテーブルはデモンストレーションされた 6 個の行動を完全にコピーし、滑らない氷の上ではクローンが毎回エキスパートの経路をたどることになる。しかし現実の分類器は厳密ではない。よく似た入力、限られた表現能力、ノイズを含むラベルといった要因により、訓練分布上にもある程度の誤差が残る。クローンにそのような誤差を持たせるため、clone は単位重みで policy_step を呼び出し、対全体で平均したときに π_θ がデモンストレーションされた行動と不一致になる確率が高々 $\varepsilon = 0.2$ になった時点で停止する。この停止規則が ε を固定するため、本節のすべての比較では当てはめ誤差が一定に保たれる。

```
def clone(mdp, states, actions, eps=0.2):
    """Behavior cloning: cross-entropy fit of pi(a|s) to the expert's
    choices (policy_step with every weight equal to one), stopped once pi
    disagrees with the demonstrated actions with probability at most eps."""
    learner = Policy.tabular(mdp.num_states, mdp.num_actions)
    weight = np.ones(len(states), np.float32)
    while 1 - np.exp(learner.log_prob_np(states, actions)).mean() > eps:
        policy_step(learner, states, actions, weight)
    return learner

bc = clone(mdp, demo_s, demo_a)
pi_bc = bc.probs(mdp.num_states, mdp.num_actions)
```

```
for s in (9, 10):
    print(f'clone pi(.|s={s}): {np.round(pi_bc[s], 3)}')
```

```
clone pi(.|s=9): [0.061 0.818 0.061 0.061]
clone pi(.|s=10): [0.25 0.25 0.25 0.25]
```

Adam を 14 ステップ実行すると、不一致率は 0.182 になる。デモンストレーションされた各状態は同じ回数だけ現れて同じ勾配を受け取るため、状態 9 の行が示すように、6 つの行はいずれもエキスパートの行動に対する確率が 0.818、残り 3 つの行動に対する確率がそれぞれ 0.061 となって終了する。状態 10 は一度もデモンストレーションされていない。その行は勾配を受け取らず一様分布のままであるため、そこではクローンは 4 回に 3 回の割合でエキスパートと不一致になる。データを繰り返しても当てはめの結果は変わらない。6 個の対のコピー 3 つにわたる平均損失は 1 つのコピーにわたる平均と同じであるため、エキスパートの 1 エピソードから作ったクローンも 100 エピソードから作ったクローンも同じテーブルになる。

14.5.2 分布シフト

14.5.2.1 訓練と運用

分類器の性能保証は、訓練データの生成元である分布に対するものである。行動クローニングはエキスパートが訪問する状態で訓練される一方、運用時には自身の行動が生み出す状態に直面する。ここではクローンを 2 通りの方法で運用する。最も確率の高い行動を選ぶ貪欲な方法と、言語モデルが次のトークンをサンプリングするように π_θ から `act` でサンプリングする方法である。

湖のモデルは既知であるため、方策の状態分布はサンプリングではなく厳密に計算できる。$d_t^\pi(s)$ を、方策 π が状態 s に t ステップ後にいる確率とする。確率質量は最初すべてスタート状態に置かれ、1 ステップ進むごとに方策とカーネルを通して移動する。

$$d_{t+1}^\pi(s') = \sum_s d_t^\pi(s) \sum_a \pi(a \mid s)\, P(s' \mid s, a). \tag{14.45}$$

次のステップにおける s' の確率質量は、方策が選ぶ行動のもとで各状態からそこへ送り込まれる質量の和である。`visitation` は湖の 100 ステップという時間制限にわたってこの漸化式を実行する。ゴールと穴はエピソードを終了させて質量を保持するため、100 ステップ後にゴールにある質量が成功確率となる。サンプリングクローンについては、`d2l.evaluate` によるモンテカルロ実行でこの厳密値を検証する。

```
def visitation(mdp, pi, absorbing=None, T=100):
    """d_0, ..., d_T: the state distributions of the policy pi[s, a] from the
    start state, propagated exactly through P for FrozenLake's 100-step
    limit. Mass that enters an `absorbing` state stays there."""
    P_pi = np.einsum('sa,sat->st', pi, mdp.P)
    if absorbing is not None:
        P_pi[absorbing] = np.eye(mdp.num_states)[absorbing]
    d = [np.eye(mdp.num_states)[0]]
    for _ in range(T):
        d.append(d[-1] @ P_pi)
    return np.array(d)

desc = env.unwrapped.desc.ravel()
goal, holes = desc == b'G', desc == b'H'
```

```
expert = np.eye(mdp.num_actions)[pi_star]               # one-hot rows
greedy = np.eye(mdp.num_actions)[pi_bc.argmax(axis=1)]
for name, pi in [('expert', expert), ('greedy clone', greedy),
                 ('sampling clone', pi_bc)]:
    d_T = visitation(mdp, pi)[-1]
    print(f'{name:>14}: reaches the goal {d_T[goal].sum():.3f}, '
          f'falls into a hole {d_T[holes].sum():.3f}')
rate = d2l.evaluate(env, bc.act, num_episodes=1000,
                    rng=np.random.default_rng(0))
print(f'sampling clone, 1000 episodes of bc.act: reaches the goal {rate:.3f}')
```

```
       expert: reaches the goal 1.000, falls into a hole 0.000
  greedy clone: reaches the goal 1.000, falls into a hole 0.000
sampling clone: reaches the goal 0.660, falls into a hole 0.340
sampling clone, 1000 episodes of bc.act: reaches the goal 0.656
```

エキスパートは毎回ゴールに到達し、貪欲クローンも同様である。デモンストレーションされた各行で最も確率の高い行動はエキスパートの行動と一致するため、貪欲クローンはエキスパートの経路をそのまま歩くからである。サンプリングクローンはエピソードの 66% でゴールに到達し、残りでは穴に落ちる。サンプリングした 1,000 エピソードの結果もノイズの範囲内でこれと一致する。滑らない氷の上ではクローンの選択以外にランダムな要素は存在しないため、経路から外れる一歩一歩はすべてクローン自身の誤った行動によるものである。

悪いのは分類器なのだろうか。次のセルでは、クローンがエキスパートとどれだけの頻度で不一致になるかを 3 つの状態集合で測定する。すなわち、訓練対、新たに生成したエキスパートの 100 エピソードの状態、そしてクローン自身が訪問する状態（訪問頻度で重み付けしたもの）である。続いて、エキスパートの 1、10、100 エピソードからクローンを作成する。

```
disagree = 1 - pi_bc[np.arange(mdp.num_states), pi_star]   # per state
fresh_s, _ = demonstrations(env, pi_star, 100)
occupancy = (visitation(mdp, pi_bc)[:-1] * ~(goal | holes)).sum(axis=0)
print('disagreement with the expert')
print(f'  on the training pairs:        {disagree[demo_s].mean():.3f}')
print(f'  on 100 fresh expert episodes: {disagree[fresh_s].mean():.3f}')
print(f"  on the clone's own states:    "
      f'{occupancy @ disagree / occupancy.sum():.3f}')
for n in [1, 10, 100]:
    pi_n = clone(mdp, *demonstrations(env, pi_star, n)).probs(
        mdp.num_states, mdp.num_actions)
    print(f'clone of {n:>3} expert episodes: reaches the goal '
          f'{visitation(mdp, pi_n)[-1][goal].sum():.3f}')
```

```
disagreement with the expert
  on the training pairs:        0.182
  on 100 fresh expert episodes: 0.182
  on the clone's own states:    0.216
clone of   1 expert episodes: reaches the goal 0.660
clone of  10 expert episodes: reaches the goal 0.660
clone of 100 expert episodes: reaches the goal 0.660
```

新たなエキスパートエピソード上でのクローンの誤り率は訓練対と同じ 18% であり、汎化ギャップは存在しない。デモンストレーションがエキスパートの分布を網羅しているからである。ここではエキスパートのデータを増やしても役に立たない。新しい状態は一切加わらず、同じ 6 個の

対の繰り返しから作ったクローンはどれも同じテーブルになり、成功率も 66% のままである。一方、クローン自身の訪問状態では誤り率が 22% へと上昇する。データが存在せず確率 $3/4$ で不一致となるセルが含まれるためである。方策がテストされる状態の分布が移動しており、それを動かしたのは方策自身の行動にほかならない。これが**分布シフト**（distribution shift）である。6 ステップのタスクではこのシフトは小さく、それによって生じる損失の大きさは 14.5.3.2 節で測定する。

14.5.2.2 カバレッジの欠落

カバレッジの欠落（missing coverage）は、これとは異なる失敗である。滑りやすい氷の上ではエキスパート自身も滑るため、訪問する状態の集合はランダムになり、少数のエピソードでは正しい行動が重要となる状態を取りこぼすことがある。次のセルでは、滑りやすい湖とその最適方策を構築し、各 n についてエキスパートの n エピソードからなるデータセットを 200 個ずつ抽出し、各データセットの貪欲クローンを厳密に評価する。このクローンはデモンストレーションされたすべての状態でエキスパートの行動を取り、それ以外の状態では行が一様分布のままであるため同点時の規則により行動 0 を選ぶ。セルでは 600 個のクローンを実際に訓練する代わりに、このテーブルを直接書き下している。

```
ice = gym.make('FrozenLake-v1', is_slippery=True)
mdp_ice = d2l.TabularMDP.from_gym(ice, gamma)
V_ice = d2l.value_iteration(mdp_ice, num_iters=1000)[-1]
pi_ice = mdp_ice.backup(V_ice).argmax(axis=1)
one_hot = np.eye(mdp_ice.num_actions)
expert_ice = visitation(mdp_ice, one_hot[pi_ice])[-1][goal].sum()
ice.reset(seed=5)
print(f'share of 200 datasets whose greedy clone is below half the expert '
      f'({expert_ice:.3f}):')
for n in [1, 3, 10]:
    rates = []
    for _ in range(200):
        seen = np.isin(np.arange(mdp_ice.num_states),
                       demonstrations(ice, pi_ice, n)[0])
        # the greedy clone: the expert's action where it has data; elsewhere
        # its row is untouched and uniform, and the tie goes to action 0
        tab = np.where(seen, pi_ice, 0)
        rates.append(visitation(mdp_ice, one_hot[tab])[-1][goal].sum())
    failed = np.mean(np.array(rates) < expert_ice / 2)
    print(f'  expert episodes per dataset {n:>2}: {failed:.1%}')
```

```
share of 200 datasets whose greedy clone is below half the expert (0.730):
  expert episodes per dataset  1: 32.0%
  expert episodes per dataset  3: 1.5%
  expert episodes per dataset 10: 0.0%
```

1 エピソードの場合、約 3 つに 1 つのデータセットでクローンの成功率がエキスパートの半分を下回る。3 エピソードでは 100 個中 1〜2 個になり、10 エピソードでは 200 個中 0 個になる。カバレッジの欠落はエキスパートのデータが蓄積するにつれて消滅する。しかし分布シフトは消えない。クローンが自身のミスによって到達する状態は、エキスパートが一度も訪問しない状態だからである。

14.5.3 複合誤差

14.5.3.1 二次および線形の誤差増大

湖におけるエキスパートの経路は 6 手である。タスクがさらに長くなると、1 ステップあたりの誤りがもたらすコストはどのように増大するだろうか。教師あり学習が抑えるのは、1 回の意思決定における誤り確率である。逐次的なタスクでは、序盤の誤りが以降のすべての状態を変化させ、以降の報酬をすべて失わせかねない。デモンストレーションはエキスパートの状態分布 $d_t^{\pi^*}$ から状態をサンプリングするため、行動クローニングがクローンの誤り率を制御できるのはその分布上だけであり、他の場所では制御できない。

命題（複合誤差）（compounding error）：**T ステップからなり各ステップの報酬が $[0,1]$ に収まるエピソード型タスクを考え、方策 $\hat{\pi}$ は各ステップ t において、エキスパートと不一致になる確率が高々 ε であるとする（ただし状態はエキスパートの分布 $d_t^{\pi^*}$ から抽出されるものとする）。(i) $\hat{\pi}$ の期待収益がエキスパートの期待収益を下回る不足分は高々 $\varepsilon T(T+1)/2 \le \varepsilon T^2$ であり、$\varepsilon T \le 1$ である限り不足分がこのオーダーに達するタスクが存在する。(ii) これに対し、$\hat{\pi}$ がエキスパートと不一致になる確率が高々 ε であるのが自身の状態分布 $d_t^{\hat{\pi}}$ から状態を抽出した場合であり、かつ 1 回の誤った行動の後にエキスパートの振る舞いに従ったときのコストが高々 u であるならば、不足分は高々 $u\varepsilon T$ である** (Ross and Bagnell 2010)。

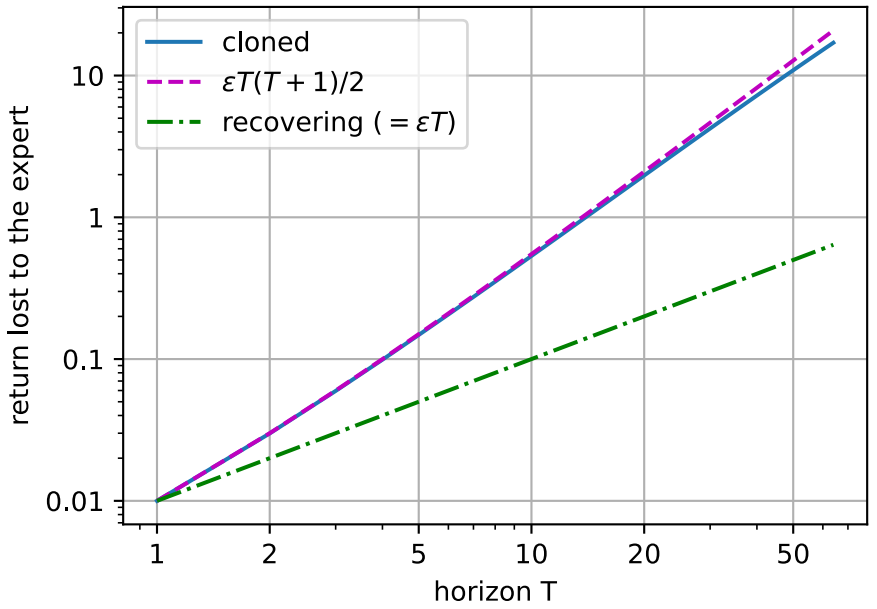
証明： (i) $\hat{\pi}$ とエキスパートを同じ初期状態・同じ環境の乱数性のもとで実行すると、$\hat{\pi}$ がエキスパートの取らない行動を最初に取るまで両者の軌跡は一致する。ステップ t におけるエキスパートの状態は $d_t^{\pi^*}$ から抽出され、最初の不一致がステップ t で生じるのは $\hat{\pi}$ がその状態で不一致になる場合に限られるため、その確率は高々 ε である。それ以降、$\hat{\pi}$ が失い得る報酬は高々ステップ $t, \dots, T$ の分、すなわち個数にして $T-t+1$ 個である。t について和をとると、不足分は高々 $\sum_{t=1}^{T} \varepsilon(T-t+1) = \varepsilon T(T+1)/2$ となる。(ii) π_k を、$\hat{\pi}$ に最初の k ステップだけ従い、その後はエキスパートに従う方策とする。すると π_0 はエキスパート、π_T は $\hat{\pi}$ となる。隣り合う 2 つのハイブリッド方策 π_{k-1} と π_k は、ステップ k における行動（$d_k^{\hat{\pi}}$ から抽出された状態で取られる行動）だけが異なる。その行動は高々確率 ε でエキスパートと不一致になり、不一致の後にエキスパートの振る舞いに従ったときのコストは高々 u である。エキスパートの収益から $\hat{\pi}$ の収益を引いた差は、隣り合うハイブリッド方策間の T 個の差の総和に等しく、したがって高々 $u\varepsilon T$ である。■

後半の (ii) は、Kakade and Langford (2002) による望遠鏡和（telescoping）の議論である。ε が T ステップにわたる 1 ステップあたり誤り率の最大値ではなく平均値であったとしても、(i) の和は依然として高々 εT^2 に収まる。(i) が最初のミスに対してそれ以降のすべてを損失として計上するのは、一度ミスをした後の方策は誤り率を何も制御できない状態に入るからである。(ii) は ε を測る場所を方策自身の訪問状態へと移すが、これが有用なのは u が小さいとき、すなわち 1 回の誤った行動からの回復をタスクが許す場合に限られる。1 回のミスが致命的であれば、u は残りのホライズンと同じ大きさになり、(ii) も再びオーダー εT^2 の評価を与える。

湖のホライズンは固定されているため、代わりに T に伴う増大を T ステップのチェーン上で観察しよう。エキスパートは毎ステップ右へ移動し、1 ステップごとに報酬 1 を獲得する。2 種類の模倣者は、毎ステップ独立に確率 ε でエキスパートと不一致になる。**クローン型**の模倣者にとってミスは永続的である。一度も示されたことのない状態へとチェーンから外れ、それ以降は何も得られないため、ステップ t の報酬を獲得する確率は $(1-\varepsilon)^t$ となる。一方、**回復型**の模倣者は 1 ステップ後にチェーンへ復帰するため、1 回のミスのコストはちょうど報酬 1 個分、すなわち $u=1$ とした (ii) に相当し、ステップ t の報酬を確率 $1-\varepsilon$ で獲得する。次のセルでは、$\varepsilon = 0.01$ とし、T を 64 まで変化させて失われる収益を厳密に計算する。

```
eps, T_max = 0.01, 64
T = np.arange(1, T_max + 1)
# probability of earning the reward of step t: the cloned imitator only if
# it has agreed at every step up to t, the recovering one if it agrees at t
earn_bc = np.cumprod(np.full(T_max, 1 - eps))
earn_rec = np.full(T_max, 1 - eps)
gap_bc, gap_rec = T - earn_bc.cumsum(), T - earn_rec.cumsum()
d2l.plot([T] * 3, [gap_bc, eps * T * (T + 1) / 2, gap_rec],
         xlabel='horizon T', ylabel='return lost to the expert',
         legend=['cloned', '$\\varepsilon T(T+1)/2$',
                 'recovering ($=\\varepsilon T$)'],
         xscale='log', yscale='log', figsize=(5, 3.5))
print(f'lost return at T={T_max}: cloned {gap_bc[-1]:.2f} '
      f'(eps T(T+1)/2 = {eps * T_max * (T_max + 1) / 2:.2f}), '
      f'recovering {gap_rec[-1]:.2f} (eps T = {eps * T_max:.2f})')
```

```
lost return at T=64: cloned 17.03 (eps T(T+1)/2 = 20.80), recovering 0.64 (eps T = 0.64)
```



両対数軸で見ると、クローン型の模倣者は二次の基準線 $\varepsilon T(T+1)/2$ の近くに留まり（$T = 10$ で差は 3% 以内、$T = 30$ で 10% 以内）、εT が大きくなるにつれて基準線より緩やかに下へ離れていく。最初のミスは 1 度しか起こり得ないからである。最初の不一致がステップ t で生じる確率は $\varepsilon(1-\varepsilon)^{t-1}$ であり、上界が計上する ε より小さい。(i) が要請するとおり、この曲線が基準線を上回ることは決してない。回復型の模倣者が失う収益はちょうど εT である。$T = 64$ では、クローン型の模倣者が約 17 の報酬を失うのに対し、回復型は 0.64 にとどまり、損失は約 27 分の 1 に抑えられる。演習問題 1 では、両方の曲線を閉形式で導出する。

14.5.3.2 クローンの損失の分解

湖の環境に戻ろう。クローンの 34 ポイントの損失のうち、どれだけが分布シフトに起因するのだろうか。クローンが失敗する仕方は 2 通りある。1 つは、訓練された 6 つの状態のいずれかで誤っ

た行動を取り、エキスパートの経路から直接穴に落ちる場合である。これはエキスパートの分布上での誤り ε であり、より良い当てはめによってしか取り除けない。もう 1 つは、経路を外れてデータのないセルへ入り、そこから失敗する場合である。こちらがシフトによる損失である。次のセルでは、経路外のセルを吸収状態にすることで visitation によりクローンが経路を外れる頻度を測り、経路上に留まるエピソードと経路を外れるエピソードのそれぞれがゴールに到達する頻度を比較する。最後の行では、経路上と経路外のすべてのセルでクローンと同じ 18% の誤り率を持つ学習者を評価する。

```
path = np.isin(np.arange(mdp.num_states), demo_s)    # the expert's cells
off_path = ~path & ~(goal | holes)
d_T = visitation(mdp, pi_bc, absorbing=off_path)[-1]
left = d_T[off_path].sum()
stay_goal = d_T[goal].sum()
clone_goal = visitation(mdp, pi_bc)[-1][goal].sum()
print(f'the clone leaves the expert path in {left:.3f} of its episodes, '
      f'entering cells {np.flatnonzero(d_T * off_path)}')
print(f'it reaches the goal in {stay_goal / (1 - left):.3f} of the episodes '
      f'that stay on the path, {(clone_goal - stay_goal) / left:.3f} of '
      f'those that leave it')
# the clone's demonstrated rows, with their error, at every cell
p_agree = pi_bc[demo_s[0], demo_a[0]]
labeled = np.where(expert == 1, p_agree, (1 - p_agree) / 3)
print(f'a learner with the same error at every cell reaches the goal '
      f'{visitation(mdp, labeled)[-1][goal].sum():.3f}')
```

```
the clone leaves the expert path in 0.172 of its episodes, entering cells [ 1 10]
it reaches the goal in 0.715 of the episodes that stay on the path, 0.394 of those that leave it
a learner with the same error at every cell reaches the goal 0.735
```

クローンはエピソードの 17% で経路を外れるが、その出口は常に状態 0 での誤った行動によるセル 1 への進入か、状態 9 または 14 からのセル 10 への進入である（Figure 14.14）。経路上に留まるエピソードは 72% の確率でゴールに到達し、経路を外れるエピソードは 39% の確率でゴールに到達する。損失もこれに応じて分解される。34 ポイントのうち、24 ポイントは経路そのものからの転落（0.828×0.285）であり、10 ポイントは経路を外れた後の失敗（0.172×0.606）である。すべてのセルでクローンと同じ誤り率を持つ学習者は 73.5% の確率でゴールに到達するため、経路外のラベルによって取り戻せるのは 34 ポイントのうち約 7.5 ポイントであり、10 ポイントには届かない。ラベル付きのセルでもなお 18% の確率で誤るからである。この湖では、分布シフトはクローンの損失の約 5 分の 1 を占める。残りは経路上および経路外における 18% の誤りそのものであり、データをどのように変えても取り除けない。経路がさらに長くなると、クローンははるかに頻繁に経路を外れるようになる。演習問題 4 では、8×8 の湖でそれが損失の 2 つの内訳にどう影響するかを調べる。

14.5.4 対話的なデータ収集

14.5.4.1 データセット集約

デモンストレーションを増やしてもシフトは解消できない。クローンが失敗するセルをエキスパートは一度も訪問しないからである。学習者が自らそこを訪問し、エキスパートがそこにラベルを付けなければならない。データセット集約（dataset aggregation）の略である **DAgger** は、まさにこれを行う手法である (Ross et al. 2011)。すなわち、現在の学習者を実行し、学習者が訪問した各状態でエキスパートならどの行動を取ったかを尋ね、それらの対をデータセットに追加して再び当てはめる（Figure 14.15）。モデル、損失、オプティマイザ、停止規則はすべてそのまま

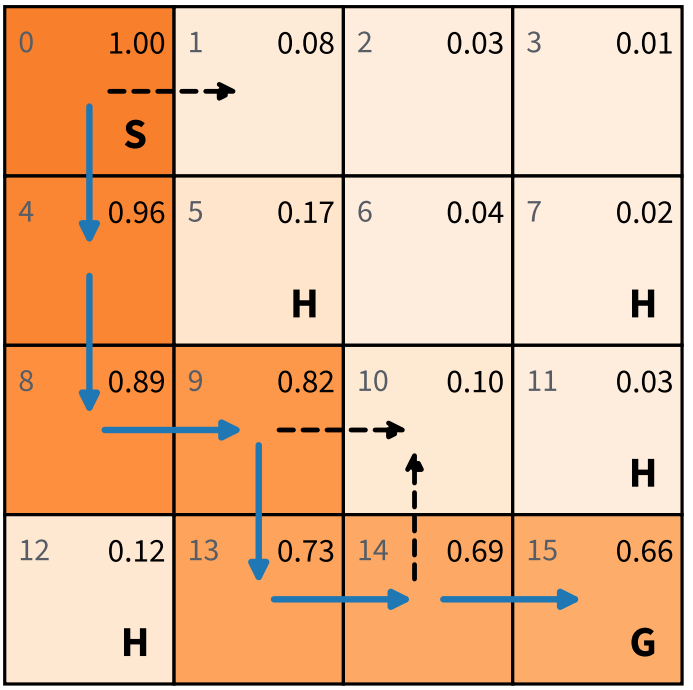


Figure 14.14: 滑らない湖における本節のクローンの移動先。青い矢印はエキスパートの 6 手の経路を示す。各セルの数値は、行動をサンプリングしエキスパートの状態で確率 $\varepsilon = 0.18$ でエキスパートと不一致になる運用時のクローンが、100 ステップの制限時間内にそのセルへ入る確率を表し、色が濃いセルほど頻繁に訪問される。破線の矢印は、クローンが経路を外れてデータがなく一様に行動するセルへと出る出口（0 から 1、および 9 または 14 から 10）である。クローンは確率 0.66 でゴールに到達し、残りの大半は経路脇の穴である 5 と 12 に吸い込まれる。

であり、データだけが変わる。

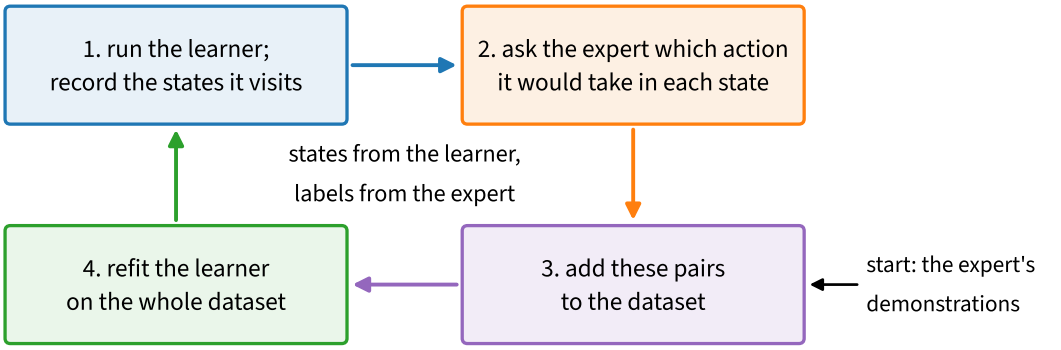


Figure 14.15: DAgger の 1 ラウンド。学習者が状態を選び、エキスパートがラベルを与える。すなわち、学習者を実行し、学習者が訪問した各状態でエキスパートが取るべき行動を指定し、それらの対をデータセットに加えて学習者を再び当てはめる。最初のデータセットはエキスパート自身のデモンストレーションである。

`relabel` は運用時と同じく行動をサンプリングする `act` で学習者をロールアウトし、訪問した状態にエキスパートの行動でラベルを付ける。

```
def relabel(env, expert, learner, num_episodes, rng):
    """Roll the learner, sampling its actions; label its states with the
    expert's actions."""
    visited = []
    for _ in range(num_episodes):
        s, done = env.reset()[0], False
        while not done:
            visited.append(s)
            s, _, terminated, truncated, _ = env.step(learner.act(s, rng))
            done = terminated or truncated
    visited = np.array(visited)
    return visited, expert[visited]
```

`dagger` は 3 エピソード分のデモンストレーションから出発し、各ラウンドで学習者の 10 エピソードを収集する処理を 4 ラウンド実行し、毎ラウンド後に `clone` で再び当てはめて各学習者を厳密に評価する。オンライン付録「ユーティリティ関数とクラス」の `d2l.run_seeds` でこれを 10 個のシードにわたって実行し、結果を積み重ねる。

```
def dagger(seed, env, mdp, expert, demo_s, demo_a, num_rounds=4,
           num_episodes=10):
    """Yield (labels, distinct states, success) for behavior cloning and
    each DAgger round after it."""
    rng = np.random.default_rng(seed)
    goal = env.unwrapped.desc.ravel() == b'G'
    S, A = demo_s, demo_a                       # the aggregated dataset
    for k in range(num_rounds + 1):
        learner = clone(mdp, S, A)
```

```
            pi = learner.probs(mdp.num_states, mdp.num_actions)
            yield S.size, np.unique(S).size, visitation(mdp, pi)[-1][goal].sum()
            if k < num_rounds:            # the learner visits, the expert labels
                new_s, new_a = relabel(env, expert, learner, num_episodes, rng)
                S, A = np.concatenate([S, new_s]), np.concatenate([A, new_a])

    runs = d2l.run_seeds(dagger, num_seeds=10, env=env, mdp=mdp, expert=pi_star,
                         demo_s=demo_s, demo_a=demo_a)   # (seed, round, quantity)
    for k, (n, m, rate) in enumerate(np.median(runs, axis=0)):
        print(f'round {k}: {n:>3.0f} labels at {m:>2.0f} states, success '
              f'{rate:.3f} (seeds: {runs[:, k, 2].min():.3f} to '
              f'{runs[:, k, 2].max():.3f}; medians over 10 seeds)')
```

```
round 0:  18 labels at  6 states, success 0.660 (seeds: 0.660 to 0.660; medians over 10 seeds)
round 1:  84 labels at  8 states, success 0.710 (seeds: 0.649 to 0.735; medians over 10 seeds)
round 2: 152 labels at  9 states, success 0.717 (seeds: 0.652 to 0.735; medians over 10 seeds)
round 3: 218 labels at 11 states, success 0.735 (seeds: 0.710 to 0.735; medians over 10 seeds)
round 4: 283 labels at 11 states, success 0.735 (seeds: 0.717 to 0.735; medians over 10 seeds)
```

ラウンド 0 は成功率 0.66 の行動クローニングである。各ラウンドでは約 65 個のラベルが追加され、その大半は経路上の 6 つの状態に対するものだが、ときおり経路外のセルも加わる。4 ラウンド（数百ラベル）のうちに、10 シードにわたる中央値はシードの選び方に応じて 0.70〜0.735 に達し、すべてのセルでクローンと同じ誤り率を持つ学習者の 0.735 に一致するかその近くまで到達する。また、どのシードでも第 4 ラウンドの学習者は行動クローニングを上回る。エキスパートのラベルだけをどれだけ増やしても、行動クローニングは 0.66 のままである。DAgger は経路外の状態に由来していた損失を取り戻す。エキスパートとの間に残る差は当てはめ自体の 18% の誤りであり、DAgger はこれには手を加えない。なお、学習者が 0.735 に到達するのは近似ではなく厳密な一致である。Adam は各パラメータの更新ステップをそのパラメータ自身の勾配の大きさで割るが、ラベル付きの行の勾配は「その行のラベル数 × ラベル付きの全行に共通する因子」となるため、何回ラベル付けされたかにかかわらず、ラベル付きのすべての行が同じ 0.818 で終了するからである。したがって学習者はどのセルにラベルがあるかだけで一意に決まり、到達し得るすべてのセルにラベルが付いた時点で厳密に 0.735 の学習者となる。

ラウンドごとに単調に改善するとは限らない。シードによってはあるラウンドで成功率がわずかに下がり、最悪で 0.649 になることがあり、その原因は演習問題 5 で突き止める。Ross et al. (2011) の理論保証は、学習者の系列のうち最良のもの、あるいはその系列からランダムに 1 つ選んだものに対する保証であり、最後の学習者に対するものではない。この保証は、多数ラウンドのデータ収集、蓄積されたデータセット上で訓練されるノーリグレット学習者（no-regret learner）、そして元の解析ではデータ収集時におけるエキスパートと学習者の行動の混合を仮定している。さらに、学習者がどのような状態に到達してもラベルを付けられるよう、随時呼び出せるエキスパートも必要とする。過去のすべてのラウンドのデータを保持し続けることで、当てはめた方策が直近の軌跡のみに依存してしまうのを防いでいる。

14.5.4.2 行動クローニングとしての教師ありファインチューニング

14.1 節のトークンマルコフ決定過程（MDP）のもとでは、言語モデルの教師ありファインチューニング（supervised fine-tuning）は行動クローニングの一例となる。状態はコンテキスト（プロンプトとそこまでに書かれたトークン列）、行動は次のトークン、遷移は確率 1 でそのトークンを末尾に追加する操作であり、厳選された応答がエキスパートのデモンストレーションにあたる。ファインチューニングの交差エントロピーは、各応答のトークンにわたって和をとった行動クローニングの負の対数尤度である。ホライズン T は応答の長さであり、u は 1 トークン誤った後にモデルがうまく続けた場合のコストを表す。すなわち、風変わりだが無害な単語であれば小さく、

後続のすべてのステップが依存する計算の序盤で数字を 1 つ誤れば大きくなる。運用時にはモデルは自身のトークンをサンプリングしてそれを条件とするため、応答が長くなるにつれてコンテキストは厳選された応答の分布からドリフトしていく。

言語モデルに対する DAgger には、モデル自身が生成したプレフィックスにラベルを付けるエキスパート、すなわち生徒モデルが生成した各コンテキストに対して次にどのトークンが来るべきかを指定する存在が必要である。人間が何百万ものプレフィックスに対してこれを行うことはできないが、より大規模なモデルなら可能である。オンポリシー蒸留（on-policy distillation）は、教師モデルの次トークン分布をラベルとして、生徒自身のサンプル上で生徒を訓練する手法であり (Agarwal et al. 2024)、教師を随時呼び出せるエキスパートとした DAgger にほかならない。また、行動クローニングは固定されたデータセットのみが利用可能なオフライン強化学習（offline reinforcement learning）における標準的な最初のベースラインでもあり、15.7 節ではオフライン手法を行動クローニングと比較する。

行動クローニングは単なる分類器だけでエキスパートの記録された選択を方策へと変え、DAgger はデータだけを変えることで、学習者自身の訪問状態に起因する損失を取り除く。ただし、どちらもエキスパートを必要とする。デモンストレーションを行うエキスパートであり、DAgger の場合はさらに学習者が何を行ってもその場でラベルを付けられる随時待機型のエキスパートである。エキスパートが利用できない場合にも、2 つの道が残されている。環境の規則が既知で微分可能であれば、そこから良い行動を計算でき、14.6 節では連続的な状態と行動に対してこれを行う。環境の規則すら未知であっても、重み $w_i = 1$ を「その行動がどれほどうまくいったか」を表す数値に置き換えれば、クローニングの更新則はそのまま適用できる。14.7 節では、その重みを報酬から導出する。

14.5.5 まとめ

行動クローニングはソフトマックス回帰と同様に交差エントロピーによってエキスパートの状態行動対に方策 $\pi_\theta(a \mid s)$ を当てはめる手法であり、遷移モデルも報酬も必要としない。各当てはめステップは、単位重みを用いた対数確率に対する勾配ステップである。この当てはめはエキスパートの状態分布上での方策の誤りを制御するが、運用時の方策は自身の行動が生み出す状態を訪問する。エキスパートの状態における 1 ステップあたりの誤り率 ε は、失われる収益が最大 $\varepsilon T(T+1)/2$（T ステップあたり）に達し得る一方、方策自身の状態における同じ誤り率であれば、損失は高々 $u\varepsilon T$（1 回のミスのコストが高々 u のとき）に収まる。滑らない湖の 6 手の経路では、エキスパートのデータを増やしてもクローンは変化せず、エキスパートが一度も訪問しない状態へのシフトが損失の約 5 分の 1 を占める。DAgger は学習者自身の訪問状態にエキスパートの行動でラベルを付け、データだけを変えることでその損失を回復する。

14.5.6 演習問題

1. **閉形式で表すチェーン**。14.5.3.1 節のチェーン実験では、模倣者は確率 ε で各ステップ $t = 1, \dots, T$ において独立にエキスパートと不一致になる。クローン型の模倣者がステップ t の報酬を獲得するのは、t までのすべてのステップで一致した場合に限られる。回復型の模倣者は、ステップ t で一致すれば常にその報酬を獲得する。
    1. クローン型の模倣者がエキスパートに対して失う期待収益 $G(T)$ を閉形式で導出し、回復型の模倣者が εT を失うことを示せ。両者を $\varepsilon = 0.01$ および $T = 64$ に対して出力された値と比較せよ。
    2. $G(T)$ を ε について 2 次まで展開せよ。1 次の項が命題の上界 $\varepsilon T(T+1)/2$ になることを示し、2 次の項を求めよ。その符号から、クローンの曲線が基準線に対してどちら側に位置するかについて何が言えるか。$\varepsilon = 0.01$ のとき、1 次の項が $G(T)$ の 10% 以内に収まるのはどのホライズンまでか。

3. $G(T)$ は $\varepsilon T \gg 1$ になるとどのように増大するか。また、したがって二次の領域が成り立つには εT にどのような条件が必要か。プロットの右端にあたる $\varepsilon = 0.01$ および $T = 64$ における 2 つの損失の比を計算し、2 つの基準線の比 $(T+1)/2$ と比較せよ。

2. **行動クローニングとしてのファインチューニング**。14.5.4.2 節では、コンテキストを状態、次のトークンを行動、厳選された応答をデモンストレーションとみなすことで、言語モデルの教師ありファインチューニングを行動クローニングと同一視した。
   1. 14.5.3.1 節の命題は、すべてのステップで同じ誤り率 ε を仮定している。ファインチューニングされたモデルは、厳選された応答に沿っては小さな一定の率でしか誤らないにもかかわらず、自身が生成した応答に沿っては位置が進むにつれて増大する率で誤ることがある。その理由を説明し、命題のどちらの主張がこのモデルを記述するかを述べよ。
   2. 件数の問題を別にすれば、言語モデルに対する DAgger のラベル供給源として人間が不向きなのはなぜか。オンポリシー蒸留は、生徒の各プレフィックスに対して 1 つのトークンではなく教師の次トークン分布全体をラベルとして付与する。得られる損失を書き下し、教師が 1 つのトークンに全確率を置くときに行動クローニングの損失に帰着することを示せ。

3. [code] **カバレッジに必要なデモンストレーション**。●14.5.2 節のカバレッジのセルでは、滑りやすい湖 `mdp_ice` とその最適方策 `pi_ice` を構築している。決定論的方策 `pi` がゴールに到達する確率は、`visitation(mdp_ice, one_hot[pi])` の後にゴールにある確率質量である。
   1. 各非終端状態 s について、基本的には `pi_ice` に従うが s でのみ行動 0 を取る方策の成功確率を計算せよ。この変更によって成功確率が `pi_ice` の半分未満に下がるとき、s を重要状態と呼ぶ。どの状態が重要状態か。
   2. q_s を、エキスパートの 1 エピソードが s を訪問する確率とする。q_s を、引数 `absorbing`（`visitation` の引数）を用いて厳密に計算せよ。すなわち、s を吸収状態にし、100 ステップ後に s にある確率質量を読み取れ。$q_s = 0$ となる非終端状態はどれか。また、それは任意の量のエキスパートデータに対して何を意味するか。3 エピソードからなるデータセットが状態 13 を一度も訪問しない確率はいくらか。
   3. 重要状態への訪問を互いに独立とみなして、それらの q_s からエキスパートの $n \in \{1, 3, 10\}$ エピソードがいずれかの重要状態をデモンストレーションしないまま残す確率を予測し、その予測をセルが出力する割合と比較せよ。この予測は高すぎるか、それとも低すぎるか。また、それは異なる重要状態への訪問について何を物語っているか。

4. [code] **より長い経路での DAgger**。●4×4 の滑らない湖では、分布シフトがクローンの損失に占める割合は約 5 分の 1 である。最適経路に 14 手を要する 8×8 の滑らない湖 `gym.make('FrozenLake-v1', map_name='8x8', is_slippery=False)` で同じ実験を繰り返せ。すなわち、`env`、`mdp`、`pi_star`、`demo_s`、`demo_a` を再構築し、本節のセルを再実行せよ。
   1. サンプリングクローンの成功確率、エキスパートの経路を外れる確率、およびすべてのセルでクローンと同じ誤り率を持つ学習者の成功確率を計算せよ。クローンの損失のうち、経路を外れた後に生じる割合はどれだけか。また、経路外のラベルによって回復できる割合はどれだけか。前者が経路の長さとともに増大する一方で、後者がそうならないのはなぜか。
   2. 10 個のシードで DAgger を実行し、エキスパートへのクエリ数に対する成功率を、少なくとも同数のエキスパートラベルを用いた行動クローニングと並べてプロットせよ。
   3. 表形式の方策が必要とするラベルは、状態ごとに高々 1 つである。各ラウンド後に DAgger がラベルを付けた異なる状態の個数を数え、ラベルのない状態でのみエキスパートに尋ねた場合に必要となるクエリ数を見積もれ。

5. **DAgger の理論保証**。●Ross et al. (2011) の解析では、データ収集方策 π_i（反復 i における方策）は各ステップで確率 β_i でエキスパートの行動を取り、それ以外では現在の学習者 $\hat{\pi}_i$ の

行動を取る。次の学習者 $\hat{\pi}_{i+1}$ は、clone による再当てはめと同様に、反復 $1, \dots, i$ の集約データ上の損失を最小化する。$d_\pi = \frac{1}{T}\sum_{t=1}^{T} d_t^\pi$ を π の平均状態分布とし、$c(s,\pi) \in [0, c_{\max}]$ を π の状態 s における分類損失（$\mathbf{1}(\pi(s) \neq \pi^*(s))$ など）とし、$c_i(\pi) = \mathbb{E}_{s\sim d_{\pi_i}}[c(s,\pi)]$ とおく。

1. $\|d_{\pi_i} - d_{\hat{\pi}_i}\|_1 \leq 2\min(1, T\beta_i)$ を示せ。
2. $\varepsilon_N = \min_\pi \frac{1}{N}\sum_{i=1}^{N} c_i(\pi)$ を事後的に最良な方策の損失とし、学習者の平均リグレットが高々 δ_N であると仮定する。

$$\frac{1}{N}\sum_{i=1}^{N} c_i(\hat{\pi}_i) - \varepsilon_N \leq \delta_N. \tag{14.46}$$

このとき、系列中のある学習者 $\hat{\pi}_j$ が次式を満たすことを示せ。

$$\mathbb{E}_{s\sim d_{\hat{\pi}_j}}[c(s,\hat{\pi}_j)] \leq \varepsilon_N + \delta_N + \frac{2c_{\max}}{N}\sum_{i=1}^{N}\min(1, T\beta_i). \tag{14.47}$$

同論文の定理 4.1 の証明では、最後の和を簡約する前にこの上界を導出している。

3. 14.5.4.1 節のループでは $\beta_1 = 1$、かつ $\beta_i = 0$（$i \geq 2$）を用いている。この上界をこのスケジュールと 0-1 損失に特殊化し、14.5.3.1 節の命題の (ii) と組み合わせよ。N が T に対してどのように増大すれば、ε_N や δ_N を含まない項を有界に保てるか。
4. 出力された実行結果の一部のシードでは、第 1 ラウンドの学習者が行動クローニングより悪化している。状態 1 のみにラベルを付けると滑らない湖のクローンの厳密な成功率が下がることを示し、状態 1 におけるエキスパートの行動と、ラベル付与前後における状態 1 および 2 の学習者の行からその低下を説明せよ。系列の中から最良の学習者を特定するには何が必要になるか。

6. **デモンストレーションの背後にある報酬。**●本文ではエキスパートの行動から学習した。これに対し、**逆強化学習**はデモンストレーションが最適となるような報酬を推定し、その報酬のもとでプランニングを行う。
   1. 14.2.5 節の望遠鏡和の議論により、ポテンシャルに基づく報酬シェイピング $\tilde{r}(s,a,s') = r(s,a) + \gamma\Phi(s') - \Phi(s)$（$r$ に対するもので、終端状態で $\Phi = 0$ を満たすもの）はいずれもエキスパートの最適性を保つ。ゼロ報酬のもとでもエキスパートが最適になることを示せ。これら 2 つの事実は、デモンストレーションから r を復元することについて何を意味するか。
   2. 最大エントロピー逆強化学習 (Ziebart et al. 2008) は、初期状態が固定された決定論的エピソード型タスクの軌跡を実行可能な軌跡全体の上で $P_\theta(\tau) \propto \exp\left(\sum_t r_\theta(s_t,a_t)\right)$ とモデル化し、デモンストレーションの尤度を最大化することで θ を当てはめる。ゴールに到達するエキスパートに対して、この基準がゼロ報酬を棄却するのはなぜか。実行可能なすべての軌跡が終端状態で終わると仮定する。このとき、この基準が r とその $\gamma = 1$ の報酬シェイピングを区別できないのはなぜか。また、時間制限によって打ち切られた軌跡に対しては、この議論のどの段階が成り立たなくなるか。

14.6 最適制御、LQR、モデル予測制御

動的計画法（14.2 節）は、16 個の状態価値からなる表を反復走査し、すべてのマスでベルマンの最適性方程式が成り立つまで更新することで滑りやすい湖の問題を解いた。それ以降に扱った Q 学習から行動クローニングに至るすべての手法も、状態ごとに表の 1 行を保持してきた。では、

14.1 節の車両の例のように状態が位置・角度・速度からなる連続ベクトル $\mathbf{x}_t \in \mathbb{R}^d$ であり、行動が連続的な制御入力 $\mathbf{u}_t \in \mathbb{R}^m$ として既知のダイナミクス $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t)$ を通じて作用する場合はどうなるだろうか。グリッドはスケールしない。d 個の座標それぞれを 100 個のビンに分割すると 100^d 個のセルが生じるが、これこそ Bellman の**次元の呪い**（curse of dimensionality）である (Bellman 1957b)。

制御理論（control theory）は、深層強化学習が登場するはるか以前に、Kalman (1960) による代数的観察を出発点として連続状態・連続行動空間におけるこの問題を解決していた。遷移ダイナミクスが状態と制御入力に関して線形であり、1 ステップのコストがその両方に関して二次であるとき、ベルマンの後ろ向き動的計画法はすべてのステップで価値関数の二次形式を保つ。100^d 個の要素をもつ表の代わりに、厳密な価値関数 $V_t(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x}$ は、$d \times d$ の対称行列 $\mathbf{P}_t$ によって $d(d+1)/2$ 個の独立な数値だけでパラメータ化される。ベルマンバックアップは**リッカチ漸化式**（Riccati recursion）に帰着し、最適方策は線形状態フィードバック則 $\mathbf{u}_t^\ast = -\mathbf{K}_t \mathbf{x}_t$ となる。これが**線形二次レギュレータ**（linear-quadratic regulator、LQR）である (Kalman 1960; Anderson and Moore 2007)。

現実の物理システムの多くは非線形であり、アクチュエータには飽和がある。2 つのアイデアが LQR を非線形連続制御へと拡張し、現代の軌跡最適化およびロボティクスの骨格をなしている (Tedrake 2023; Bertsekas 2025)。第 1 に、**反復 LQR**（iterative LQR、iLQR）(Li and Todorov 2004) は、候補となる軌跡の周りで非線形ダイナミクスの線形化とコストの二次近似を繰り返し、後ろ向きパスで時変 LQR 問題を解いて前向きパスで軌跡を改善する。第 2 に、**モデル予測制御**（model predictive control、MPC）(Rawlings et al. 2017) は、有限ホライズンの軌跡最適化を閉ループ方策へと変換する。すなわち、各ステップ t において現在の観測状態 $\mathbf{x}_t$ から移動する先読みホライズン（receding lookahead horizon）にわたって制御問題を解き直し、最初の制御入力 $\mathbf{u}_{t|t}^\ast$ のみを適用して、ステップ $t+1$ で再計画する。本節では LQR、iLQR、および移動ホライズン MPC を展開し、局所線形フィードバックで十分な領域と非線形なエネルギー注入（energy pumping）が必要となる領域を明らかにしたうえで、MPC が 15.6 節における学習済みニューラルダイナミクスを用いた計画の土台をいかに築くかを示す。

14.6.1 線形二次制御

14.6.1.1 状態空間ダイナミクスと二次コスト

制御理論の記法では、連続状態を $\mathbf{x}_t \in \mathbb{R}^d$、行動（または制御入力）を $\mathbf{u}_t \in \mathbb{R}^m$ と書く。強化学習が報酬 $r(\mathbf{x}_t, \mathbf{u}_t)$ を最大化するのに対し、最適制御（optimal control）はステージコスト $c(\mathbf{x}_t, \mathbf{u}_t) = -r(\mathbf{x}_t, \mathbf{u}_t)$ を最小化する。離散時間線形システムは次のように時間発展する。

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t, \tag{14.48}$$

ここで、$\mathbf{A} \in \mathbb{R}^{d \times d}$ は状態遷移行列、$\mathbf{B} \in \mathbb{R}^{d \times m}$ は制御入力行列である（38.1 節の線形常微分方程式の離散時間版に相当する）。目標状態は原点 $\mathbf{x} = \mathbf{0}$ である。H ステップの有限ホライズンにわたり、次の総二次コストを最小化する。

$$J(\mathbf{x}_0, \mathbf{u}_{0:H-1}) = \sum_{t=0}^{H-1} \left(\mathbf{x}_t^\top \mathbf{Q} \mathbf{x}_t + \mathbf{u}_t^\top \mathbf{R} \mathbf{u}_t \right) + \mathbf{x}_H^\top \mathbf{Q}_H \mathbf{x}_H, \tag{14.49}$$

ここで、$\mathbf{Q} \succeq \mathbf{0}$ と $\mathbf{Q}_H \succeq \mathbf{0}$ は $d \times d$ の対称半正定値な状態コスト行列であり、$\mathbf{R} \succ \mathbf{0}$ は $m \times m$ の対称正定値な制御コスト行列である。$\mathbf{R} \succ \mathbf{0}$ を課すことで、大きな制御入力には常に正のペナルティが伴うため、$\mathbf{u}_t$ に関する 1 ステップ最小化は一意な有限解をもつ。

14.6.1.2 厳密な連続動的計画法とリッカチ漸化式

$V_t(\mathbf{x})$ を状態 $\mathbf{x}$、ステップ $t \in \{0, \dots, H\}$ からの最適残りコスト（cost-to-go）とする。終端ステップ $t = H$ では残りの制御入力はなく、$V_H(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_H \mathbf{x}$（ただし $\mathbf{P}_H = \mathbf{Q}_H$）となる。$t+1$ から t へと後ろ向きに進むと、有限ホライズン版のベルマンの最適性方程式（14.2 節）は次の式を与える。

$$V_t(\mathbf{x}) = \min_{\mathbf{u} \in \mathbb{R}^m} \Big\{ \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{u}^\top \mathbf{R} \mathbf{u} + V_{t+1}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}) \Big\}. \tag{14.50}$$

後ろ向き帰納法により、ある対称半正定値行列について $V_{t+1}(\mathbf{x}') = \mathbf{x}'^\top \mathbf{P}_{t+1} \mathbf{x}'$（$\mathbf{P}_{t+1} \succeq \mathbf{0}$）が成り立つと仮定しよう。$\mathbf{x}' = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$ を代入すると、括弧内の行動価値関数 $Q_t(\mathbf{x}, \mathbf{u})$ は $\mathbf{u}$ に関する狭義凸二次関数に展開される。

$$\begin{aligned} Q_t(\mathbf{x}, \mathbf{u}) &= \mathbf{x}^\top (\mathbf{Q} + \mathbf{A}^\top \mathbf{P}_{t+1} \mathbf{A}) \mathbf{x} + 2 \mathbf{u}^\top \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{A} \mathbf{x} \\ &\quad + \mathbf{u}^\top (\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{B}) \mathbf{u}. \end{aligned}$$

$\mathbf{u}$ に関する勾配をゼロとおくと、次のようになる。

$$\nabla_{\mathbf{u}} Q_t(\mathbf{x}, \mathbf{u}) = 2(\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{B}) \mathbf{u} + 2 \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{A} \mathbf{x} = \mathbf{0}, \tag{14.51}$$

これより、一意な最適制御が閉形式で得られる。

$$\mathbf{u}_t^* = -\mathbf{K}_t \mathbf{x}_t, \qquad \mathbf{K}_t = (\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{A}. \tag{14.52}$$

$\mathbf{u} = -\mathbf{K}_t \mathbf{x}$ を $Q_t(\mathbf{x}, \mathbf{u})$ に戻して代入すると、$V_t(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x}$ が純粋な二次形式のままであることが確認でき、$\mathbf{P}_t$ は後ろ向きの**離散時間リッカチ漸化式**（discrete-time Riccati recursion）に従う。

$$\begin{aligned} \mathbf{P}_t = \mathbf{Q} &+ \mathbf{A}^\top \mathbf{P}_{t+1} \mathbf{A} \\ &- \mathbf{A}^\top \mathbf{P}_{t+1} \mathbf{B} (\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{A}. \end{aligned} \tag{14.53}$$

この漸化式は $\mathbf{P}_H = \mathbf{Q}_H$ から後ろ向きに 1 回計算するだけでよく、その際に状態を知る必要はない。したがってシステムの実行コストは低い。各ステップで制御器は $\mathbf{x}_t$ を計測し、保存しておいたゲインを掛けるだけでよい（Figure 14.16）。

Equation 14.52 と Equation 14.53 には、特筆すべき 3 つの性質がある。

1. **次元の呪いがないこと：**各ベルマンバックアップは、無限の行動空間 $\mathbb{R}^m$ とすべての状態 $\mathbb{R}^d$ にわたる最小化を、$O(d^3 + m^3)$ 回の $\mathbf{P}_{t+1}$ に対する行列演算に置き換える。
2. **線形フィードバックゲイン：**最適方策 $\pi_t^*(\mathbf{x}) = -\mathbf{K}_t \mathbf{x}$ は状態に関して線形であり、ゲイン行列 $\mathbf{K}_t \in \mathbb{R}^{m \times d}$ は $(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R})$ のみに依存し、初期状態 $\mathbf{x}_0$ には依存しない。
3. **確実性等価（certainty equivalence）：**平均ゼロの独立ノイズ $\mathbf{w}_t$（共分散 Σ）が遷移に加わっても（$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t$）、交差項 $\mathbb{E}[\mathbf{w}_t^\top \mathbf{P}_{t+1}(\mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t)]$ は消え、価値関数に定数 $\operatorname{tr}(\mathbf{P}_{t+1}\Sigma)$ が加わるだけで最適フィードバック行列 $\mathbf{K}_t$ は変化しない（演習問題 2）。

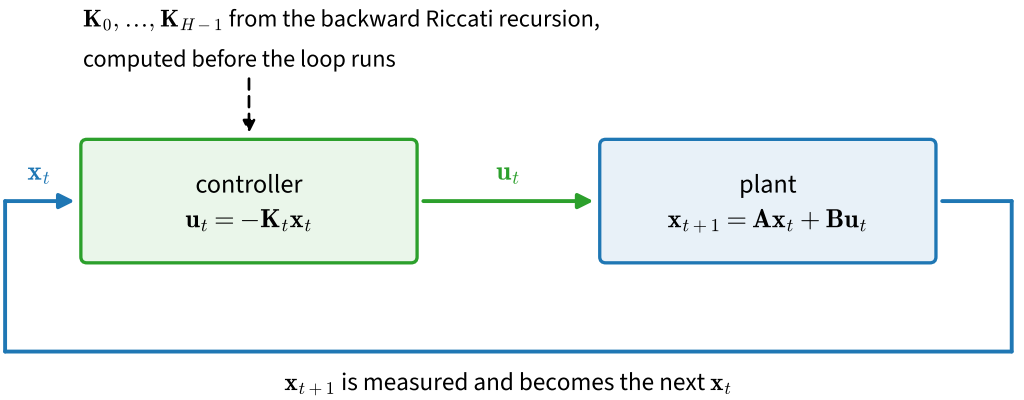


Figure 14.16: 閉ループでの LQR。制御器は計測された状態 $\mathbf{x}_t$ にゲイン $-\mathbf{K}_t$ を掛け、システム（制御用語でいう**プラント**）は次の状態で応答し、その状態が再び計測される。すべてのゲインはループ開始前に後ろ向きリッカチ漸化式によって計算される。

ホライズン $H \to \infty$ の極限では（標準的な可制御性（controllability）および可観測性（observability）の条件のもとで）、$\mathbf{P}_t$ は**離散代数リッカチ方程式**（discrete algebraic Riccati equation、DARE）の定常解 $\mathbf{P}_\infty$ に収束し、時不変制御器 $\mathbf{u}_t = -\mathbf{K}_\infty \mathbf{x}_t$ は閉ループダイナミクス $\mathbf{x}_{t+1} = (\mathbf{A} - \mathbf{B}\mathbf{K}_\infty)\mathbf{x}_t$ を安定化して、$\mathbf{A} - \mathbf{B}\mathbf{K}_\infty$ のすべての固有値を単位円の厳密な内側、すなわち $\rho(\mathbf{A} - \mathbf{B}\mathbf{K}_\infty) < 1$ に配置する。

14.6.1.3 二重積分器

`riccati_lqr` を 14.1 節の状態拡張の例で扱った車両でテストしよう。これは直線上を動く単位質量の質点であり、状態は $\mathbf{x}_t = (p_t, v_t)^\top$（位置と速度）、制御入力は u_t（加速度）、タイムステップは $\Delta t = 0.1$ である。Newton の運動法則より、$p_{t+1} = p_t + v_t \Delta t + \tfrac{1}{2} u_t \Delta t^2$ および $v_{t+1} = v_t + u_t \Delta t$ が成り立つ。位置と速度のずれに $\mathbf{Q} = \mathrm{diag}(1.0, 0.1)$ でペナルティを課し、制御入力の重み $r_u \in \{0.01, 0.10, 1.00\}$ を $\mathbf{R} = [r_u]$ において変化させる。

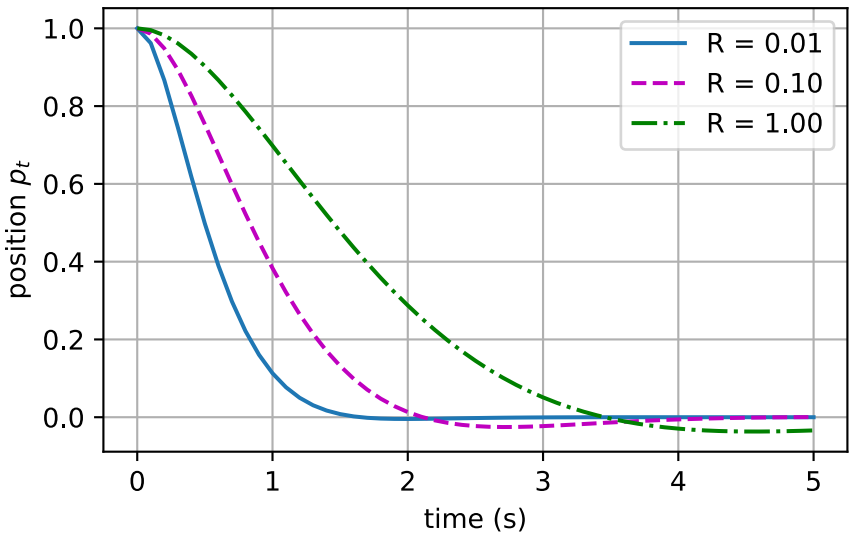
```
def riccati_lqr(A, B, Q, R, H=100):
    """Backward Riccati recursion for horizon H; returns (Ps, Ks) from t=0..H."""
    P = Q.copy()
    Ps, Ks = [P], []
    for _ in range(H):
        BtP = B.T @ P
        K = np.linalg.solve(R + BtP @ B, BtP @ A)
        P = Q + A.T @ P @ (A - B @ K)
        Ks.append(K)
        Ps.append(P)
    return Ps[::-1], Ks[::-1]

dt = 0.1
A_di = np.array([[1.0, dt], [0.0, 1.0]])
B_di = np.array([[0.5 * dt**2], [dt]])
Q_di = np.diag([1.0, 0.1])
```

```
for ru in (0.01, 0.10, 1.00):
    Ps, Ks = riccati_lqr(A_di, B_di, Q_di, np.array([[ru]]), H=100)
    K_inf, P_inf = Ks[0][0], Ps[0]
    rho = np.abs(np.linalg.eigvals(A_di - B_di @ Ks[0])).max()
    print(f'R={ru:4.2f}  K_inf=[{K_inf[0]:5.3f}, {K_inf[1]:5.3f}]  '
          f'P_00={P_inf[0, 0]:5.2f}  rho(A - B K)={rho:.3f}')
```

```
R=0.01  K_inf=[7.613, 4.585]  P_00= 6.02  rho(A - B K)=0.761
R=0.10  K_inf=[2.762, 2.508]  P_00= 9.08  rho(A - B K)=0.874
R=1.00  K_inf=[0.930, 1.395]  P_00=15.00  rho(A - B K)=0.930
```

また、閉ループの位置軌跡 p_t（初期状態 $\mathbf{x}_0 = (1, 0)^\top$）を 3 つの制御入力ペナルティのもとでプロットできる。

```
trajs, steps_di = [], np.arange(51) * dt
for ru in (0.01, 0.10, 1.00):
    _, Ks = riccati_lqr(A_di, B_di, Q_di, np.array([[ru]]), H=100)
    x, xs_di = np.array([1.0, 0.0]), [1.0]
    for _ in range(50):
        x = (A_di - B_di @ Ks[0]) @ x
        xs_di.append(x[0])
    trajs.append(xs_di)
d2l.plot(steps_di, trajs, 'time (s)', 'position $p_t$',
         legend=['R = 0.01', 'R = 0.10', 'R = 1.00'], figsize=(5, 3))
```



定常フィードバック則 $u_t^* = -k_p p_t - k_v v_t$ の 2 つの成分に注目しよう。動的計画法によって、位置誤差 p_t に対してゲイン k_p で押し戻し、速度 v_t をゲイン k_v で減衰させる**比例・微分**（proportional-derivative、PD）制御器が自動的に導出されている。制御入力が安価なとき（$R = 0.01$）、LQR は積極的なゲイン $\mathbf{K}_\infty = [7.613, 4.585]$ を選び、状態ノルムを 1 ステップあたり $\rho(\mathbf{A} - \mathbf{B}\mathbf{K}_\infty) = 0.761$ の比率で縮小させる。一方、制御入力が高価なとき（$R = 1.00$）は、穏やかな加速度 $\mathbf{K}_\infty = [0.930, 1.395]$ を用い、縮小率は $\rho = 0.930$、累積コスト $V_0(\mathbf{x}_0) = \mathbf{x}_0^\top \mathbf{P}_\infty \mathbf{x}_0$ はより大きくなる（$P_{00} = 15.00$ 対 6.02）。

14.6.2 平衡点周りでの非線形システムの線形化

14.6.2.1 Pendulum-v1 の解析的線形化

ここで、本章の残りの部分および第 15 章を通じて用いる物理ベンチマークである倒立振子（inverted pendulum、Pendulum-v1）を考えよう。質量 $m = 1$、長さ $l = 1$ の剛体棒の一端が重力 $g = 10$ のもとで軸支されており、有界なモータトルク $u_t \in [-2, 2]$ によって駆動される。連続状態は $\mathbf{x}_t = (\vartheta_t, \dot{\vartheta}_t)^\top$ であり、ϑ_t は鉛直上向きから測った角度（$\vartheta = 0$ が直立、$\vartheta = \pi$ が真下への懸垂）、$\dot{\vartheta}_t \in [-8, 8]$ は角速度である。水平位置（$\vartheta = \pi/2$）における最大重力トルクは $mgl/2 = 5 > 2$ であるため、このモータは**劣駆動**（underactuated）であり、棒を最下点から頂上まで一息に持ち上げることはできない。シミュレータの 1 ステップ $\Delta t = 0.05\,\mathrm{s}$ にわたる半陰的オイラー法（semi-implicit Euler）の遷移 $\mathbf{x}_{t+1} = f(\mathbf{x}_t, u_t)$ は次の式で与えられる。

$$\begin{aligned} \dot{\vartheta}_{t+1} &= \dot{\vartheta}_t + \Big(\frac{3g}{2l} \sin \vartheta_t + \frac{3}{m l^2} u_t \Big) \Delta t, \\ \vartheta_{t+1} &= \vartheta_t + \dot{\vartheta}_{t+1} \Delta t, \end{aligned} \tag{14.54}$$

ただし、角速度は $[-8, 8]$ にクリッピングされる。各ステップでは二次ステージコスト $c(\vartheta_t, \dot{\vartheta}_t, u_t) = \vartheta_t^2 + 0.1\,\dot{\vartheta}_t^2 + 0.001\,u_t^2$ が発生し（ここで ϑ_t は $[-\pi, \pi]$ に折り返され、$\mathbf{Q} = \mathrm{diag}(1.0, 0.1)$ および $\mathbf{R} = [[0.001]]$ に対応する）、等価な表現として報酬 $r_t = -c(\vartheta_t, \dot{\vartheta}_t, u_t)$ が得られる。標準的なベンチマークのエピソードは 200 ステップ（10 s）実行される。無制御の振子のスコアが $-1{,}200$ から $-1{,}500$ 程度であるのに対し、棒を振り上げ（swing-up）て直立状態に保つ制御器は、ベンチマークのランダムな初期状態にわたる平均で約 -200 以上のスコアを達成する。本節では既知の運動方程式 Equation 14.54 を用いて直接計画を立てるが、14.9 節、15.5 節、および 15.6 節では、物理法則を知らないエージェントが行動を選択したりダイナミクスモデルを学習したりする場面でこの同じ振子ベンチマークに立ち返る。

直立姿勢 $\mathbf{x}^\ast = (0, 0)^\top$ とゼロトルク $u^\ast = 0$ の組は不動点 $f(\mathbf{x}^\ast, u^\ast) = \mathbf{x}^\ast$ である。任意の基準対 $(\bar{\mathbf{x}}, \bar{u})$ の近傍では、1 次のテイラー展開によって非線形遷移を偏差 $\delta\mathbf{x}_t = \mathbf{x}_t - \bar{\mathbf{x}}$ と $\delta u_t = u_t - \bar{u}$ に関する局所線形モデルで近似できる。

$$\begin{aligned} f(\mathbf{x}_t, u_t) - f(\bar{\mathbf{x}}, \bar{u}) &\approx \mathbf{A}(\bar{\mathbf{x}}, \bar{u})\,\delta\mathbf{x}_t + \mathbf{B}(\bar{\mathbf{x}}, \bar{u})\,\delta u_t, \\ \mathbf{A} = \frac{\partial f}{\partial \mathbf{x}}(\bar{\mathbf{x}}, \bar{u}), &\qquad \mathbf{B} = \frac{\partial f}{\partial u}(\bar{\mathbf{x}}, \bar{u}). \end{aligned} \tag{14.55}$$

Equation 14.54 を $(\vartheta_t, \dot{\vartheta}_t)$ および u_t で微分すると、$\partial \dot{\vartheta}_{t+1} / \partial \vartheta_t = \frac{3g}{2l} \cos(\vartheta_t) \Delta t$ および $\partial \dot{\vartheta}_{t+1} / \partial u_t = \frac{3}{m l^2} \Delta t$ が得られる。

```
def pendulum_jacobians(th, dt=0.05, g=10.0):
    """Exact Jacobians (A, B) of Pendulum-v1 at angle th."""
    dthdot_dth = 1.5 * g * np.cos(th) * dt
    dthdot_du = 3.0 * dt
    A = np.array([[1.0 + dthdot_dth * dt, dt], [dthdot_dth, 1.0]])
    B = np.array([[dthdot_du * dt], [dthdot_du]])
    return A, B

A_p, B_p = pendulum_jacobians(0.0)
Q_p, R_p = np.diag([1.0, 0.1]), np.array([[0.001]])
Ps_p, Ks_p = riccati_lqr(A_p, B_p, Q_p, R_p, H=200)
```

```
    K_lqr = Ks_p[0][0]
    eigs_open = np.linalg.eigvals(A_p).real
    eigs_closed = np.linalg.eigvals(A_p - B_p @ Ks_p[0]).real
    print(f'A at upright:\n{A_p}')
    print(f'open-loop eigenvalues:   {eigs_open[0]:.4f}, {eigs_open[1]:.4f}')
    print(f'LQR gain K_inf:          [{K_lqr[0]:.3f}, {K_lqr[1]:.3f}]')
    print(f'closed-loop eigenvalues: {eigs_closed[0]:.4f}, {eigs_closed[1]:.4f}')
```

```
A at upright:
[[1.0375 0.05  ]
 [0.75   1.    ]]
open-loop eigenvalues:   1.2133, 0.8242
LQR gain K_inf:          [19.693, 5.263]
closed-loop eigenvalues: 0.8537, 0.2467
```

直立不動点において、開ループのヤコビ行列 $\mathbf{A}$ は不安定な固有値 $\lambda_1 = 1.2133 > 1$ をもつ。そのため、わずかな傾きであっても 1 ステップあたり 21% 超の割合で指数関数的に増大する（38.1 節の相図と同様である）。`Pendulum-v1` のコスト行列 $\mathbf{Q} = \mathrm{diag}(1, 0.1)$ と $\mathbf{R} = [[0.001]]$ を用いてリッカチ漸化式を解くと、フィードバックゲイン $\mathbf{K}_\infty = [19.693, 5.263]$ が得られ、閉ループの固有値は両方とも単位円の内側（$\lambda_1 = 0.8537, \lambda_2 = 0.2467$）に引き込まれる。

14.6.2.2 トルク飽和と吸引領域

モータトルクが $[-2, 2]$ にクリッピングされているとき、この線形制御器は頂上からどこまで離れた振子を捕捉できるだろうか。振子を引き下ろす重力トルクは $\frac{1}{2}mgl\sin\vartheta = 5\sin\vartheta$ であるのに対し、モータが供給できるトルクは高々 $|u| = 2$ である。$|\vartheta| > \arcsin(2/5) \approx 0.412\,\mathrm{rad}$（$23.6^\circ$）を超えると、重力がモータの最大保持トルクを上回る。クリッピング付き LQR 方策 $u_t = \mathrm{clip}(-\mathbf{K}_\infty \mathbf{x}_t, -2, 2)$ を、完全な非線形振子に対して静止状態の 5 つの初期角度 ϑ_0 からテストしてみよう。

```
    def wrap(th):
        return ((th + np.pi) % (2 * np.pi)) - np.pi

    def pendulum_step(x, u, dt=0.05, g=10.0):
        """One step of nonlinear Pendulum-v1 dynamics in (theta, thetadot)."""
        th, thdot = x
        u_c = float(np.clip(u, -2.0, 2.0))
        newthdot = float(np.clip(thdot + (1.5 * g * np.sin(th) + 3.0 * u_c) * dt,
                                 -8.0, 8.0))
        newth = th + newthdot * dt
        cost = wrap(th)**2 + 0.1 * thdot**2 + 0.001 * u_c**2
        return np.array([newth, newthdot]), cost

    for th0 in (0.20, 0.40, 0.50, 0.60, np.pi):
        x, total_cost = np.array([th0, 0.0]), 0.0
        for _ in range(200):
            u = -(K_lqr[0] * wrap(x[0]) + K_lqr[1] * x[1])
            x, c = pendulum_step(x, u)
            total_cost += c
        print(f'th0={th0:4.2f} rad: 200-step return={-total_cost:8.2f}, '
              f'final theta={wrap(x[0]):+6.3f} rad')
```

```
th0=0.20 rad: 200-step return=   -0.32, final theta=+0.000 rad
th0=0.40 rad: 200-step return=   -2.97, final theta=+0.000 rad
th0=0.50 rad: 200-step return= -770.94, final theta=+2.592 rad
th0=0.60 rad: 200-step return= -546.37, final theta=+0.001 rad
th0=3.14 rad: 200-step return=-1471.86, final theta=-2.249 rad
```

静的保持限界の内側（$\vartheta_0 = 0.20$ および 0.40 rad）では、線形 LQR はほぼゼロのコストで振子をそのまま直立状態（$\vartheta_{200} = 0.000$）へと導く（収益は -0.32 と -2.97）。このしきい値をわずかに超えると（$\vartheta_0 = 0.50$ rad）、最大の負トルク $u = -2$ でも落下を食い止められず、200 ステップ後も棒は最下点を通過して揺れ続けている（収益 -770.94）。しきい値の先では、結果はもはや ϑ_0 に対して単調ではない。0.60 rad から始めた場合も棒は同様に落下するが、さらに数回揺れたあと、たまたまクリッピング付き制御器が捕捉できるほど十分に遅い速度で上がってくる（収益 -546.37）。これは制御ではなく偶然である。最下点（$\vartheta_0 = \pi$）からは、線形制御器の収益は -1471.86 となり、一度も頂上に到達しない。モータが劣駆動（$u_{\max} = 2 < 5$）であるため、$\vartheta_0 = \pi$ からの振り上げには**エネルギー注入**が必要になる。すなわち、一方向に押し、振子が最下点を通って振り戻る際にトルクを反転させて運動エネルギーと位置エネルギーを蓄え、棒が頂上付近の捕捉領域まで登り切って初めて直立安定化へ切り替える必要がある。

14.6.3 反復 LQR による非線形軌跡最適化

14.6.3.1 軌跡に沿った線形化

$(\mathbf{0}, 0)$ での単一の線形化は、非線形な $\sin(\vartheta)$ の曲率（$\vartheta_0 = \pi$ からのスイングに沿ったもの）について何の情報ももたない。そこで、**反復線形二次制御**（iterative linear-quadratic regulation、iLQR）(Li and Todorov 2004) は、微分動的計画法（differential dynamic programming）(Mayne 1966) のガウス・ニュートン型変種であり、ノミナル軌跡全体の周りで線形化する。

ハードクリッピングによる折れ目を避けてトルク制約 $u_t \in [-2, 2]$ を滑らかに満たすため、$u_t = 2\tanh(v_t)$ とおき、無制約の制御変数 $v_t \in \mathbb{R}$ でパラメータ化する。任意のノミナル制御系列 $\bar{v}_{0:H-1}$ と、その前向きロールアウト $\bar{\mathbf{x}}_{0:H}$（真の非線形ダイナミクス $\mathbf{x}_{t+1} = f(\mathbf{x}_t, v_t)$ に従うもの。なお以下の振り上げでは速度上限に達しないため速度クリッピングは省く）が与えられたとき、iLQR の各反復は次の 2 つのパスを実行する。

1. **$(\bar{\mathbf{x}}_{0:H}, \bar{v}_{0:H-1})$ に沿った後ろ向きリッカチパス：**各ステップ $t = H-1, \dots, 0$ において局所ヤコビ行列 $\mathbf{A}_t = \frac{\partial f}{\partial \mathbf{x}}(\bar{\mathbf{x}}_t, \bar{v}_t)$ と $\mathbf{B}_t = \frac{\partial f}{\partial v}(\bar{\mathbf{x}}_t, \bar{v}_t)$ を計算し、行動価値の摂動 $\delta Q_t(\delta\mathbf{x}_t, \delta v_t)$ の 2 次テイラー近似を、ステージコストの勾配 $(\ell_{\mathbf{x},t}, \ell_{v,t})$、ヘッセ行列 $(\mathbf{L}_{\mathbf{xx},t}, \ell_{vv,t})$、および次ステップの価値勾配 $\mathbf{v}_{\mathbf{x},t+1}$ とヘッセ行列 $\mathbf{V}_{\mathbf{xx},t+1}$ から構成する。

$$
\begin{aligned}
\mathbf{q}_{\mathbf{x},t} &= \ell_{\mathbf{x},t} + \mathbf{A}_t^{\top}\mathbf{v}_{\mathbf{x},t+1}, \qquad q_{v,t} = \ell_{v,t} + \mathbf{B}_t^{\top}\mathbf{v}_{\mathbf{x},t+1},\\
\mathbf{Q}_{\mathbf{xx},t} &= \mathbf{L}_{\mathbf{xx},t} + \mathbf{A}_t^{\top}\mathbf{V}_{\mathbf{xx},t+1}\mathbf{A}_t, \qquad \mathbf{q}_{v\mathbf{x},t} = \mathbf{B}_t^{\top}\mathbf{V}_{\mathbf{xx},t+1}\mathbf{A}_t,\\
q_{vv,t} &= \ell_{vv,t} + \mu + \mathbf{B}_t^{\top}\mathbf{V}_{\mathbf{xx},t+1}\mathbf{B}_t,
\end{aligned}
$$

   ここで、$\mu > 0$ はレーベンバーグ・マーカート減衰項である。δv_t に関して最小化すると、開ループのフィードフォワード補正量 $k_t = -q_{v,t}/q_{vv,t}$ と局所線形フィードバックゲイン $\mathbf{K}_t = -\mathbf{q}_{v\mathbf{x},t}/q_{vv,t}$ が得られる。

2. **バックトラッキング直線探索を伴う前向きロールアウト：**線形モデルは $(\bar{\mathbf{x}}_t, \bar{v}_t)$ の近傍でのみ正確であるため、ステップサイズ $\alpha \in \{1, 1/2, 1/4, \dots\}$ を用いて**真の非線形ダイナミクス**をロールアウトする。

$$
\begin{aligned}
v_t^{\text{new}} &= \bar{v}_t + \alpha\, k_t + \mathbf{K}_t(\mathbf{x}_t^{\text{new}} - \bar{\mathbf{x}}_t),\\
\mathbf{x}_{t+1}^{\text{new}} &= f(\mathbf{x}_t^{\text{new}}, v_t^{\text{new}}),
\end{aligned}
\tag{14.56}
$$

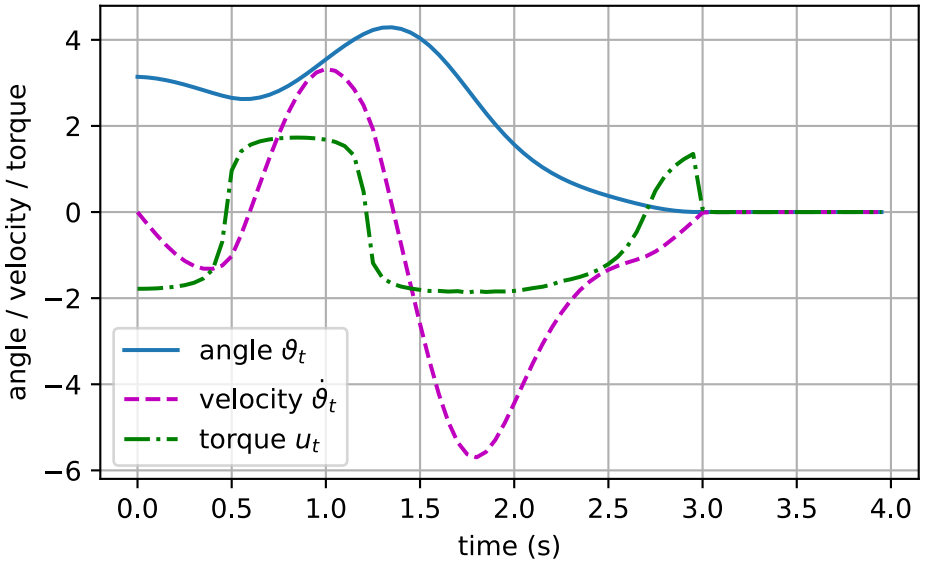
そして、真の非線形軌跡コストを減少させる最初の α を採用する。

軌跡コストの設計における 1 つの工夫は示唆に富む。$\vartheta_0 = \pi$ からエネルギーを注入するには、振子はまず反対側の $\vartheta > \pi$ へと振れてから、勢いよく $\vartheta = 0$ へ向かって振り戻る必要がある。もし重い二次ペナルティ ϑ_t^2 を序盤のすべてのステップ t に課すと、局所勾配法は 0 から離れる方向へ振ることを拒んでしまう。バックスイング中に ϑ_t^2 が増加するからである。そこで、スイング段階では状態コストの重みを軽くし（$q_{\text{run}} = 0.02$、$t < H - 20$）、終盤の捕捉区間では重くする（$q_T = 100$、$t \geq H - 20$）ことで、iLQR は序盤のステップを使ってエネルギーを注入し、終盤のステップで棒を $(\vartheta, \dot{\vartheta}) = (0, 0)$ に固定できる。

```
def step_tanh(x, v, dt=0.05, g=10.0):
    """Nonlinear step and analytical Jacobians (A_t, B_t) for u = 2 tanh(v)."""
    th, thdot = x[0], x[1]
    th_v = np.tanh(v)
    u = 2.0 * th_v
    du_dv = 2.0 * (1.0 - th_v**2)
    newthdot = thdot + (1.5 * g * np.sin(th) + 3.0 * u) * dt
    newth = th + newthdot * dt
    dthdot_dth = 1.5 * g * np.cos(th) * dt
    dthdot_dv = 3.0 * dt * du_dv
    A = np.array([[1.0 + dthdot_dth * dt, dt], [dthdot_dth, 1.0]])
    B = np.array([[dthdot_dv * dt], [dthdot_dv]])
    return np.array([newth, newthdot]), A, B, u

def ilqr_swingup(x0, H=80, iters=35, q_run=0.02, r_v=0.02, q_T=100.0,
                 vs_init=None):
    """iLQR trajectory optimization from x0 over horizon H."""
    vs = (np.array(vs_init, dtype=float) if vs_init is not None
          else np.array([0.2 if t < 15 else 0.0 for t in range(H)]))
    Q_base = np.diag([1.0, 0.1])
    for _ in range(iters):
        xs, As, Bs, us, cost = [np.array(x0, dtype=float)], [], [], [], 0.0
        for t in range(H):
            x_next, A, B, u = step_tanh(xs[-1], vs[t])
            As.append(A); Bs.append(B); us.append(u)
            w = q_T if t >= H - 20 else q_run
            cost += w * (xs[-1] @ Q_base @ xs[-1]) + r_v * vs[t]**2
            xs.append(x_next)
        cost += q_T * (xs[-1] @ Q_base @ xs[-1])

        Vx = 2.0 * q_T * (Q_base @ xs[-1])
        Vxx = 2.0 * q_T * Q_base
        ks, Ks = np.zeros(H), np.zeros((H, 2))
        for t in range(H - 1, -1, -1):
            w = q_T if t >= H - 20 else q_run
            A, B = As[t], Bs[t]
            Qx = 2.0 * w * (Q_base @ xs[t]) + A.T @ Vx
            Qv = 2.0 * r_v * vs[t] + (B.T @ Vx).item()
            Qxx = 2.0 * w * Q_base + A.T @ Vxx @ A
```

```
            Qvv = 2.0 * r_v + 1e-2 + (B.T @ Vxx @ B).item()
            Qvx = (B.T @ Vxx @ A).ravel()
            ks[t], Ks[t] = -Qv / Qvv, -Qvx / Qvv
            Vx = Qx + Qvx * ks[t]
            Vxx = Qxx + np.outer(Qvx, Ks[t])

        for alpha in (1.0, 0.5, 0.25, 0.1, 0.03):
            x_new, vs_new, c_new = np.array(x0, dtype=float), np.zeros(H), 0.0
            for t in range(H):
                v_t = vs[t] + alpha * ks[t] + Ks[t] @ (x_new - xs[t])
                vs_new[t] = v_t
                w = q_T if t >= H - 20 else q_run
                c_new += w * (x_new @ Q_base @ x_new) + r_v * v_t**2
                x_new, _, _, _ = step_tanh(x_new, v_t)
            c_new += q_T * (x_new @ Q_base @ x_new)
            if c_new < cost:
                vs = vs_new
                break
    return np.array(xs), np.array(us), vs, Ks, cost

xs_nom, us_nom, vs_nom, Ks_nom, _ = ilqr_swingup([np.pi, 0.0], H=80, iters=35)
for t in range(0, 81, 10):
    u_t = us_nom[t] if t < 80 else 0.0
    print(f't={t:2d}  theta={xs_nom[t, 0]:7.4f}  '
          f'thetadot={xs_nom[t, 1]:7.4f}  u={u_t:+6.3f}')
d2l.plot(np.arange(80) * 0.05, [xs_nom[:-1, 0], xs_nom[:-1, 1], us_nom],
         'time (s)', 'angle / velocity / torque',
         legend=[r'angle $\vartheta_t$', r'velocity $\dot{\vartheta}_t$',
                 r'torque $u_t$'], figsize=(5.5, 3.2))
```

```
t= 0  theta= 3.1416  thetadot= 0.0000  u=-1.782
t=10  theta= 2.6515  thetadot=-1.0260  u=+0.965
t=20  theta= 3.5453  thetadot= 3.3225  u=+1.681
t=30  theta= 4.0383  thetadot=-2.6013  u=-1.807
t=40  theta= 1.5668  thetadot=-4.4402  u=-1.832
t=50  theta= 0.3746  thetadot=-1.3417  u=-1.209
t=60  theta= 0.0009  thetadot=-0.0132  u=+0.066
t=70  theta= 0.0002  thetadot=-0.0006  u=-0.000
t=80  theta= 0.0001  thetadot=-0.0000  u=+0.000
```



iLQR が 35 回の反復によって最下点 $(\pi, 0)$ から発見した軌跡を見てみよう。

- **ステップ 0〜10（初期のバックスイング）：**左方向へ押し（$u_0 = -1.782$）、$\vartheta_{10} = 2.652$ rad まで振る。
- **ステップ 10〜30（エネルギー注入）：**トルクを反転させ（$u_{10} = +0.965 \rightarrow u_{20} = +1.681$）、最下点を通過して反対側の $\vartheta_{30} = 4.038$ rad まで振子を駆動し、重力位置エネルギーを蓄える。
- **ステップ 30〜60（振り抜きと捕捉）：**再びトルクを反転させ（$u_{30} = -1.807 \rightarrow u_{40} = -1.832$）、最大 5.7 rad/s で最下点を通過し、$\dot{\vartheta}_{40} = -4.440$ rad/s の速度を保ったまま水平位置（$\vartheta_{40} = 1.567$ rad）を越えて直立の捕捉領域（$\vartheta_{50} = 0.375$ rad）へと駆け上がり、滑らかに制動して頂上で静止する（$\vartheta_{60} = 0.0009$ rad、$\vartheta_{80} = 0.0001$ rad、$u_{80} = 0.000$）。

この計画は局所最適解であり、最良の振り上げだと誤解してはならない。iLQR の各反復は与えられた軌跡を改善するものであるため、結果は初期制御系列に依存する。`ilqr_swingup` は最初の 15 ステップにわたる小さな押し（$\bar{v}_t = 0.2$）とそれ以降のゼロ入力から出発し、固定された減衰係数 μ を用いている。この初期の押しは重要であり、$\bar{v} = 0$ から始めると、同じ 35 回の反復では棒を頂上で静止させることができない。初期推定値へのこの依存性は以下で再び現れ、制御器が衝撃から回復できるかどうかを左右することになる。

14.6.4 移動ホライズンモデル予測制御

14.6.4.1 開ループ計画、局所フィードバックチューブ、オンライン再計画

最適軌跡 $(\bar{\mathbf{x}}_{0:H}, \bar{u}_{0:H-1})$ は、それだけではまだ頑健な制御器ではない。この計画を実行するには次の 3 通りの方法がある。

1. **開ループ実行：**事前計算された系列 $\bar{u}_t$ を、実際の状態 $\mathbf{x}_t$ を見ずにそのまま適用する。直立姿勢は不安定な鞍点であるため、わずかな外乱や数値ドリフトでも軌跡はコースから外れてしまう。
2. **時変 LQR（time-varying LQR、TV-LQR）追従：**iLQR の最後の後ろ向きパスで計算された局所フィードバック行列 $\mathbf{K}_t$ を用い、$v_t = \bar{v}_t + \mathbf{K}_t(\mathbf{x}_t - \bar{\mathbf{x}}_t)$ を適用する。これによりオンライン計画コストなしでノミナル軌跡の周りに局所線形フィードバックチューブが形成

されるが、強い衝撃によって $\mathbf{x}_t$ がチューブの外へ押し出され、新たな振り上げのためのエネルギー注入が必要な領域に入ると破綻する。

3. **移動ホライズンモデル予測制御（MPC）**：各タイムステップ t において真の状態 $\mathbf{x}_t$ を観測し、$\mathbf{x}_t$ を始点とする先読みウィンドウ上で有限ホライズン制御問題を解き直し（前ステップの計画を 1 ステップずらしてウォームスタートする）、**最初の制御入力のみ** $u^*_{t|t}$ を適用して、これを $t+1$ で繰り返す (Rawlings et al. 2017)。

ここで実装する `mpc_action` では、3 番目の方式における 2 つの細部を定めている。第 1 はウィンドウの長さである。シフトされた計画は直前の計画より 1 ステップ短くなるため、ウィンドウはノミナル計画の 80 ステップから始まり、それとともに短縮していく。残り `H_min` $= 25$ ステップになった時点で計画の末尾をゼロで埋め、以降はウィンドウをその長さに保ったまま移動させる。第 2 はウォームスタートが失敗したときの対処である。iLQR は局所手法であるため、ウォームスタートされた計画は、元の軌跡がもはや頂上へ到達しなくなった場合でも古い軌跡の近くにとどまってしまう。再最適化した計画の終端が直立位置から 0.2 rad を超えて離れている場合、`mpc_action` はそれを破棄し、`H_mpc` $= 60$ ステップ（3 s）にわたる新たな振り上げを、`ilqr_swingup` がデフォルトで用いる初期の押しから計画し直す。

3 つの制御器すべてを、$T = 120$ ステップにわたり初期状態 $\mathbf{x}_0 = (\pi, 0)$ から比較しよう。振子がちょうど水平位置を越えて直立の捕捉領域へ登っていくステップにおいて、外部から角速度インパルス $\Delta\dot{\vartheta} \in \{0.0, +0.5, +2.2\}$ rad/s がステップ $t = 45$ で加わるものとする。

```
T = 120
xs_pad = np.vstack([xs_nom, np.zeros((T - 80, 2))])
vs_pad = np.concatenate([vs_nom, np.zeros(T - 80)])
us_pad = np.concatenate([us_nom, np.zeros(T - 80)])
Ks_pad = np.vstack([Ks_nom, np.tile(Ks_nom[-1], (T - 80, 1))])

def mpc_action(x, vs_warm, H_mpc=60, H_min=25):
    """One MPC step: re-optimize the shifted plan (at least H_min steps) from x;
    if it no longer ends at the top, plan afresh over H_mpc steps."""
    H = max(H_min, len(vs_warm))
    warm = np.concatenate([vs_warm, np.zeros(H)])[:H]
    xs_w, us_w, vs_w, _, _ = ilqr_swingup(x, H=H, iters=18, vs_init=warm)
    if abs(xs_w[-1, 0]) > 0.2:   # warm plan missed the top: re-seed
        _, us_s, vs_s, _, _ = ilqr_swingup(x, H=H_mpc, iters=25)
        return us_s[0], vs_s[0], vs_s[1:]
    return us_w[0], vs_w[0], vs_w[1:]

for impulse in (0.0, 0.5, 2.2):
    x_ol, x_tv, x_mpc = [np.array([np.pi, 0.0]) for _ in range(3)]
    ret_ol = ret_tv = ret_mpc = 0.0
    th_ol, th_tv, th_mpc = [], [], []
    vs_warm = vs_nom.copy()
    for t in range(T):
        if t == 45:
            for x in (x_ol, x_tv, x_mpc):
                x[1] += impulse
        th_ol.append(wrap(x_ol[0]))
        th_tv.append(wrap(x_tv[0]))
        th_mpc.append(wrap(x_mpc[0]))
        # 1. Open-loop
        ret_ol -= wrap(x_ol[0])**2 + 0.1 * x_ol[1]**2 + 0.001 * us_pad[t]**2
```

```
            x_ol, _, _, _ = step_tanh(x_ol, vs_pad[t])
            # 2. Time-varying LQR tracking
            v_tv = vs_pad[t] + Ks_pad[t] @ (x_tv - xs_pad[t])
            u_tv = 2.0 * np.tanh(v_tv)
            ret_tv -= wrap(x_tv[0])**2 + 0.1 * x_tv[1]**2 + 0.001 * u_tv**2
            x_tv, _, _, _ = step_tanh(x_tv, v_tv)
            # 3. Receding-horizon MPC
            u_m, v_m, vs_warm = mpc_action(x_mpc, vs_warm)
            ret_mpc -= wrap(x_mpc[0])**2 + 0.1 * x_mpc[1]**2 + 0.001 * u_m**2
            x_mpc, _, _, _ = step_tanh(x_mpc, v_m)
        print(f'impulse={impulse:+.1f} rad/s at t=45:')
        print(f'  Open-loop:  final theta={wrap(x_ol[0]):+6.3f}, return={ret_ol:6.1f}')
        print(f'  TV-LQR:     final theta={wrap(x_tv[0]):+6.3f}, return={ret_tv:6.1f}')
        print(f'  MPC (iLQR): final theta={wrap(x_mpc[0]):+6.3f}, return={ret_mpc:6.1f}')
    d2l.plot(np.arange(T) * 0.05, [th_ol, th_tv, th_mpc],
             'time (s)', r'wrapped angle $\vartheta_t$ (rad)',
             legend=['Open-loop', 'TV-LQR', 'MPC (iLQR)'], figsize=(5.5, 3.2))
```

```
impulse=+0.0 rad/s at t=45:
  Open-loop:  final theta=+0.075, return=-333.6
  TV-LQR:     final theta=+0.000, return=-333.7
  MPC (iLQR): final theta=+0.000, return=-331.6
impulse=+0.5 rad/s at t=45:
  Open-loop:  final theta=-2.939, return=-502.3
  TV-LQR:     final theta=-0.001, return=-333.9
  MPC (iLQR): final theta=+0.000, return=-331.7
impulse=+2.2 rad/s at t=45:
  Open-loop:  final theta=+2.941, return=-621.1
  TV-LQR:     final theta=+1.959, return=-797.9
  MPC (iLQR): final theta=-0.000, return=-558.2
```

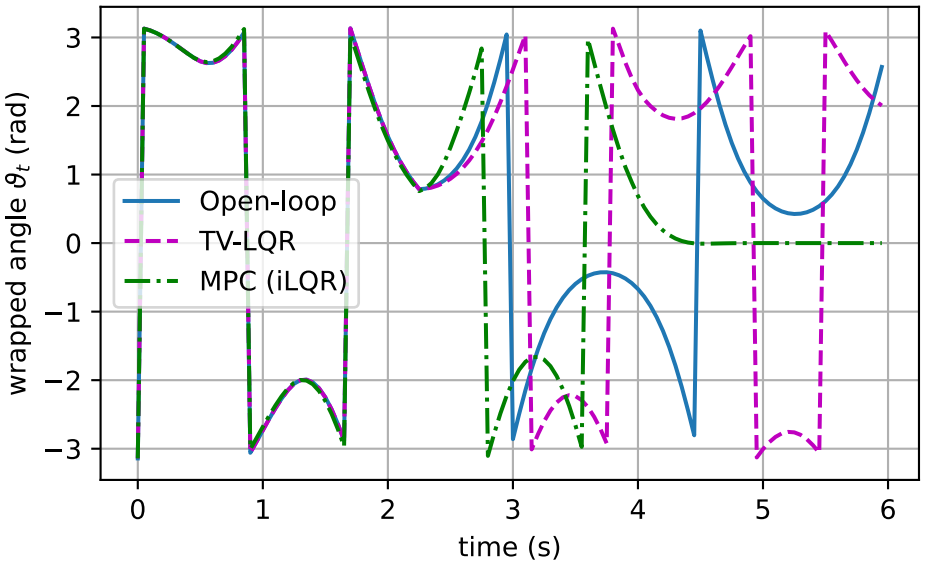


Figure 14.17 に本節の 3 つの段階を並べて示す。直立平衡点周りの局所 LQR の価値等高線とト

ルク飽和境界、$(\vartheta, \dot{\vartheta})$ 位相平面における iLQR のエネルギー注入軌跡、そして強いインパルス $\Delta\dot{\vartheta} = +2.2$ rad/s（$t = 45$）のもとでの外乱抑制（disturbance rejection）軌跡である。

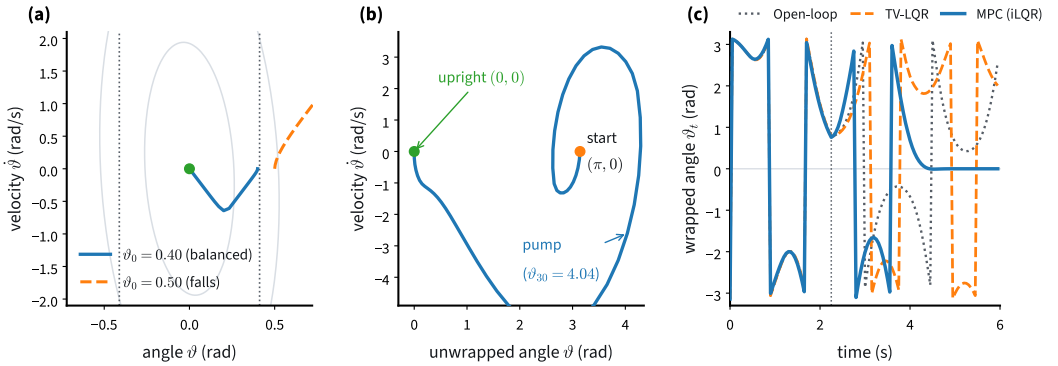


Figure 14.17: 局所 LQR から移動ホライズン MPC までの最適制御。(a) 直立 LQR の残りコスト等高線 $V_\infty(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_\infty \mathbf{x}$ と静的保持境界 $|\vartheta| \le \arcsin(2/5) \approx 0.412$ rad（点線）。$\vartheta_0 = 0.40$ rad からの出発では原点に収束する（青）のに対し、$\vartheta_0 = 0.50$ rad ではモータトルク $|u| \le 2$ が飽和して落下する（オレンジ破線）。(b) 80 ステップの iLQR 振り上げ軌跡（$(\vartheta, \dot{\vartheta})$ 位相平面）。$(\pi, 0)$ から $\vartheta_{30} = 4.04$ rad までエネルギーを注入したのち、最下点を振り抜けて直立平衡点 $(0, 0)$ に至る。(c) 折り返し角度 ϑ_t の時間変化（インパルス $\Delta\dot{\vartheta} = +2.2$ rad/s が $t = 45$ すなわち 2.25 s の縦点線で加わった場合）。開ループと TV-LQR は失敗するが、移動ホライズン MPC は新たな振り上げを計画して $\vartheta = 0$ に戻る。

この振子ベンチマークにおける 3 段階の外乱レベルは、3 つの制御器の特性を鮮明に描き分ける。

- **外乱なし（$\Delta\dot{\vartheta} = 0.0$）：**3 つとも振子を振り上げ、120 ステップにわたって -332 から -334 の収益を得る。これがベンチマークの目安として挙げた -200 を下回るのは 2 つの理由による。第 1 に、ベンチマークがランダムな初期状態にわたって平均するのに対し、ここでは最悪の状態である最下点から出発していること。第 2 に、iLQR は収益を採点する振子の報酬そのものではなく、スイング中の状態重みを軽くした独自の軌跡コストを最小化していることである。$t = 80$ 以降の最後の 40 ステップでは、開ループは不安定な鞍点で $\vartheta_{120} = +0.075$ rad へとドリフトし始めるが、TV-LQR と MPC は $\vartheta_{120} = 0.000$ rad を維持する。
- **中程度の外乱（$\Delta\dot{\vartheta} = +0.5$ rad/s）：**開ループ実行は事前計算されたスケジュールから弾き出されて再び落下し、最終的に $\vartheta_{120} = -2.939$ rad（収益 -502.3）に終わる。一方、TV-LQR（$\vartheta_{120} = -0.001$ rad、収益 -333.9）と MPC（$\vartheta_{120} = +0.000$ rad、収益 -331.7）はどちらも衝撃を吸収し、棒を直立状態に固定する。
- **強い外乱（$\Delta\dot{\vartheta} = +2.2$ rad/s）：**衝撃によって上昇が食い止められ、振子は最下点に向かって落下し始める。古いノミナル軌跡周りの TV-LQR の線形フィードバックはトルクが飽和して破綻する（$\vartheta_{120} = +1.959$ rad、収益 -797.9）。回復には古い軌跡の線形近傍には存在しない新たな振り上げのエネルギー注入が必要だからである。MPC は振子を直立平衡点へと復帰させる（$\vartheta_{120} = 0.000$ rad、収益 -558.2）が、これは再シード（re-seed）規則のおかげである。衝撃が加わったステップでウォームスタート計画が頂上を外すため、`mpc_action` は外乱後の状態から新たな振り上げを計画し、以降はそれに従う。ウォームスタート計画は古い軌跡の近くにとどまるため、ウォームスタートからの再計画だけでは回復できない。

14.6.4.2 古典的制御から強化学習へ

Table 14.2 は古典的最適制御と強化学習の対応関係をまとめた対訳表であり、本章の以降の節および第 15 章が LQR と iLQR の各仮定をどのように緩和していくかを示している。

Table 14.2: 古典的最適制御と強化学習の架け橋。

概念	古典的最適制御	強化学習
変数	状態 $\mathbf{x}_t \in \mathbb{R}^d$、制御入力 $\mathbf{u}_t \in \mathbb{R}^m$	状態 $s_t \in \mathcal{S}$、行動 $a_t \in \mathcal{A}$
目的関数	累積コスト $\sum_t c(\mathbf{x}_t, \mathbf{u}_t)$ の最小化	期待割引収益 $\mathbb{E}[\sum_t \gamma^t r(s_t, a_t)]$ の最大化
価値関数	残りコスト $V_t(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x}$（LQR）	状態価値 $V^\pi(s)$ または行動価値 $Q^\pi(s,a)$
ベルマンバックアップ	リッカチ漸化式 Equation 14.53	ベルマン期待値・最適作用素（14.2 節）
既知の非線形物理系 f	勾配ベースの軌跡最適化（iLQR・DDP）＋ MPC	木探索またはサンプリングベースの MPC（15.6 節）
未知ダイナミクス、学習モデル $\hat{f}_\phi$	システム同定＋確実性等価 MPC	ランダムシューティング・CEM によるモデルベース強化学習（15.6 節）
未知ダイナミクス、モデルなし	モデルフリー適応制御	方策勾配（14.7 節、14.9 節）、PPO（15.2 節）、DQN（15.4 節）、SAC（15.5 節）

ダイナミクス f が未知であり、しかも遷移モデルを当てはめずに行動しなければならない場合、後ろ向きリッカチパスに必要なヤコビ行列 $\mathbf{A}_t = \partial f / \partial \mathbf{x}$ と $\mathbf{B}_t = \partial f / \partial \mathbf{u}$ は計算できない。14.7 節では、確率的方策 $\pi_\theta(a \mid s)$ が自身のサンプリングした行動の対数確率を代わりに微分することで、これらの未知の導関数をいかに迂回するかを示す。

14.6.5 まとめ

線形ダイナミクス $\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t$ と二次コスト $\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{u}^\top \mathbf{R}\mathbf{u}$ に対して、ベルマンの後ろ向き動的計画法は連続空間における厳密な閉形式解をもつ。すなわち、残りコストは二次形式 $V_t(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x}$ となり、$\mathbf{P}_t$ は後ろ向きリッカチ漸化式に従い、最適方策は線形フィードバック制御器 $\mathbf{u}_t^* = -\mathbf{K}_t \mathbf{x}_t$（LQR）で与えられる。非線形システムを不安定不動点の周りで解析的ヤコビ行列によって線形化すると、アクチュエータ制限下の捕捉領域内にある状態は安定化できるが、その外側で劣駆動の振り上げによるエネルギー注入が必要になると破綻する。反復 LQR（iLQR）はノミナル軌跡の周りでダイナミクスの線形化とコストの二次近似を繰り返し、後ろ向きリッカチパスと前向き非線形ロールアウトによって多段階のエネルギー注入軌跡を発見する。開ループ計画の実行は外乱のもとで破綻し、固定軌跡周りの時変 LQR フィードバックは小さな摂動しか扱えないが、移動ホライズンモデル予測制御（MPC）は各ステップで現在の状態から有限ホライズン問題を解き直すため、大きな外乱によって全く新しい軌跡が必要になった場合でも回復できる。

14.6.6 演習問題

1. **リッカチバックアップにおける平方完成。**$\mathbf{S}_{t+1} = \mathbf{R} + \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{B}$ および $\mathbf{K}_t = \mathbf{S}_{t+1}^{-1} \mathbf{B}^\top \mathbf{P}_{t+1} \mathbf{A}$ とおく。
   1. 積を展開することで、任意の $\mathbf{x} \in \mathbb{R}^d$ と $\mathbf{u} \in \mathbb{R}^m$ に対して

$$Q_t(\mathbf{x}, \mathbf{u}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x} + (\mathbf{u} + \mathbf{K}_t \mathbf{x})^\top \mathbf{S}_{t+1} (\mathbf{u} + \mathbf{K}_t \mathbf{x}),$$

   が成り立つことを示せ（ここで $\mathbf{P}_t$ は Equation 14.53 で与えられる）。

   2. $\mathbf{R} \succ \mathbf{0}$ と $\mathbf{P}_{t+1} \succeq \mathbf{0}$ を用いて、$\mathbf{S}_{t+1} \succ \mathbf{0}$ であること、$\mathbf{u}_t^* = -\mathbf{K}_t\mathbf{x}$ が $Q_t(\mathbf{x},\mathbf{u})$ の一意な大域的最小点であること、および $\mathbf{P}_t \succeq \mathbf{0}$ であることを導け。
   3. Joseph の安定化形式
   $$\mathbf{P}_t = \mathbf{Q} + \mathbf{K}_t^\top \mathbf{R}\mathbf{K}_t + (\mathbf{A}-\mathbf{B}\mathbf{K}_t)^\top \mathbf{P}_{t+1}(\mathbf{A}-\mathbf{B}\mathbf{K}_t), \tag{14.57}$$
   が成り立つことを確かめ、$\mathbf{K}_t$ を任意のフィードバック行列 $\mathbf{K}$ に置き換えても右辺のすべての項が明らかに半正定値となる理由を説明せよ。
2. **加法的ガウスノイズのもとでの確実性等価**。線形ダイナミクス Equation 14.48 が、平均ゼロで独立なノイズベクトル $\mathbf{w}_0,\dots,\mathbf{w}_{H-1}$（共分散 $\mathbb{E}[\mathbf{w}_t\mathbf{w}_t^\top] = \Sigma$）によって摂動され、$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t$ となると仮定する。
   1. 後ろ向き帰納法により、最適残りコストが $V_t(\mathbf{x}) = \mathbf{x}^\top\mathbf{P}_t\mathbf{x} + c_t$ となることを証明せよ。ここで、$\mathbf{P}_t$ は決定論的な場合と全く同じリッカチ漸化式 Equation 14.53 に従い、$c_t = \sum_{k=t+1}^{H} \operatorname{tr}(\mathbf{P}_k\Sigma)$ である。
   2. 最適フィードバック方策 $\mathbf{u}_t^* = -\mathbf{K}_t\mathbf{x}_t$ がノイズの共分散 Σ に依存しないことを導け。もしノイズが乗法的であり、$\mathbf{x}_{t+1} = (\mathbf{A}+\tilde{\mathbf{A}}_t)\mathbf{x}_t + (\mathbf{B}+\tilde{\mathbf{B}}_t)\mathbf{u}_t$ である場合、この確実性等価の性質はどこで破綻するか。
3. [code] **可制御性と無限ホライズン極限**。線形システム $(\mathbf{A},\mathbf{B})$（状態数 d）は、可制御性行列 $\mathcal{C} = [\mathbf{B}, \mathbf{A}\mathbf{B}, \dots, \mathbf{A}^{d-1}\mathbf{B}]$ が行フルランク d をもつとき**可制御**であるという。
   1. $\mathbf{x}_0 = \mathbf{0}$ から出発したとき、d ステップ後に到達する状態が $\mathbf{x}_d = \mathcal{C}\,[\mathbf{u}_{d-1}^\top, \mathbf{u}_{d-2}^\top, \dots, \mathbf{u}_0^\top]^\top$ に等しくなることを示し、$\mathcal{C}$ が行フルランクをもつことが $\mathbb{R}^d$ 内の任意の目標状態に d ステップで到達するための必要十分条件であることを示せ。
   2. $\operatorname{rank}(\mathcal{C})$ と $\det(\mathcal{C})$ を、二重積分器 $(\mathbf{A}_{\mathrm{di}},\mathbf{B}_{\mathrm{di}})$ および線形化した直立振子 $(\mathbf{A}_p,\mathbf{B}_p)$ に対して計算せよ。
   3. 第 1 座標に制御が作用しない（$\mathbf{B}_{\mathrm{un}} = [0,0]^\top$ に相当する）非結合な不安定システム $\mathbf{A} = \operatorname{diag}(1.2, 0.8)$, $\mathbf{B} = [0,1]^\top$ を設定し、`riccati_lqr` を $H \in \{10, 50, 100\}$ に対して実行せよ。$\mathbf{P}_0$ のどの成分が $H \to \infty$ で発散するか。また、それはなぜか。
4. [code] **エネルギー整形と軌跡最適化の比較**。古典的な非線形制御では、振子の全力学的エネルギー $E(\vartheta,\dot\vartheta) = \tfrac{1}{6}ml^2\dot\vartheta^2 + \tfrac{1}{2}mgl\cos\vartheta$ を制御して振り上げることが多い。この値は $E^* = \tfrac{1}{2}mgl = 5$（直立静止平衡点 $(\vartheta=0,\dot\vartheta=0)$）および -5（最下点 $(\vartheta=\pi,\dot\vartheta=0)$）に等しい (Tedrake 2023)。
   1. 連続時間の摩擦のない軌跡（$\ddot\vartheta = \frac{3g}{2l}\sin\vartheta + \frac{3}{ml^2}u$）に沿ったエネルギーの時間変化率が $\dot E = u\dot\vartheta$ となることを示し、トルク $u = -k_E(E-E^*)\dot\vartheta$（$k_E > 0$）を加えると $(E-E^*)^2$ が $\dot\vartheta \neq 0$ である限り単調に減少することを示せ。
   2. `pendulum_step` 上でハイブリッド制御器を実装せよ。すなわち、クリッピング付きエネルギー注入 $u = \mathrm{clip}(-k_E(E-E^*)\dot\vartheta, -2, 2)$（微小な初期キック $u_0 = 2$ を伴う）を $|\mathrm{wrap}(\vartheta)| > 0.35$ のときに適用し、直立 LQR 制御器 $u = \mathrm{clip}(-\mathbf{K}_\infty\mathbf{x}, -2, 2)$ へ $|\mathrm{wrap}(\vartheta)| \le 0.35$ となった時点で切り替えるものとする。200 ステップの $(\pi, 0)$ からの収益を iLQR の収益と比較せよ。
5. [code] **MPC におけるホライズン長と終端コスト**。`mpc_action` はシフトされた計画を少なくとも `H_min` $= 25$ ステップにわたって再最適化し、その計画が頂上を外したときには `H_mpc` $= 60$ ステップ（3 s）にわたる新たな振り上げを計画する。
   1. 本節の $\Delta\dot\vartheta = +2.2$ の実験を `H_mpc` $\in \{15, 25, 40, 60\}$ で繰り返せ。どのホライズンで MPC は振子を頂上へ戻せるか。それらをノミナルな振り上げが頂上に到達するのに要するステップ数と比較し、短いホライズンではプランナが最下点付近に閉じ込められる理由を説明せよ。
   2. 終端コスト行列 $q_T\mathbf{Q}_{\mathrm{base}}$（`ilqr_swingup` 内）を、無限ホライズンの DARE 行列 $\mathbf{P}_\infty$（直立平衡点で `riccati_lqr` により計算したもの）に置き換えよ。すでに直立の捕捉領域

内（$|\vartheta_0| \le 0.35$）にある初期状態に対して、終端コスト $\mathbf{x}_H^\top \mathbf{P}_\infty \mathbf{x}_H$ をもつ移動ホライズン LQR 制御器は、**厳密な**無限ホライズン最適制御 $\mathbf{u}_t^* = -\mathbf{K}_\infty \mathbf{x}_t$ を、先読みがわずか 1 ステップ（$H = 1$）のときでさえ生成することを示せ。

6. **Pontryagin の共状態と価値勾配。** $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t)$ を滑らかな決定論的システムとし、$J = \sum_{t=0}^{H-1} c(\mathbf{x}_t, \mathbf{u}_t) + c_H(\mathbf{x}_H)$ とおく。ラグランジュ乗数（**共状態**または**随伴変数**とよばれる）$\lambda_1, \dots, \lambda_H \in \mathbb{R}^d$ を、制約 $\mathbf{x}_{t+1} - f(\mathbf{x}_t, \mathbf{u}_t) = \mathbf{0}$ に対して導入する。
   1. ラグランジアン $\mathcal{L} = J + \sum_{t=0}^{H-1} \lambda_{t+1}^\top (f(\mathbf{x}_t, \mathbf{u}_t) - \mathbf{x}_{t+1})$ を $\mathbf{x}_t$ と $\mathbf{u}_t$ で微分し、後ろ向き随伴方程式

$$\lambda_H = \nabla_{\mathbf{x}} c_H(\mathbf{x}_H), \qquad \lambda_t = \nabla_{\mathbf{x}} c(\mathbf{x}_t, \mathbf{u}_t) + \mathbf{A}_t^\top \lambda_{t+1}, \tag{14.58}$$

   および勾配 $\nabla_{\mathbf{u}_t} J = \nabla_{\mathbf{u}} c(\mathbf{x}_t, \mathbf{u}_t) + \mathbf{B}_t^\top \lambda_{t+1}$ を導出せよ (Pontryagin et al. 1962)。
   2. この後ろ向き随伴漸化式を、1 次のベクトル $\mathbf{v}_{\mathbf{x},t}$ および $q_{v,t}$（`ilqr_swingup` の後ろ向きパスにおけるもの）、ならびに 8.5 節の通時的誤差逆伝播法と比較せよ。iLQR の 2 次のヘッセ行列の漸化式 $\mathbf{V}_{\mathbf{xx},t}$ は、1 次の通時的誤差逆伝播法に対して何を付け加えているか。

14.7 方策勾配

14.5 節では、デモンストレーションされた各行動の対数確率（log probability）に対して、すべての行動の重みを 1 とした上昇ステップを実行することで、方策（policy）がエキスパートから学習した。また 14.6 節では、プランナが既知の遷移方程式 $f(\mathbf{x}_t, \mathbf{u}_t)$ を微分して後ろ向きリッカチゲインを計算した。ではここで、エキスパートも環境の既知で微分可能なモデルも存在せず、エージェント自身の試行と各試行が得た報酬（reward）だけがある状況を考えてみよう。自然な推測は、クローニングの更新式をそのまま残して重みだけを変えることである。すなわち、エージェントがとったすべての行動の対数確率を、その行動が属するエピソードの収益（return）で重み付けして引き上げ、報われたエピソードの行動ほど起こりやすくなるようにする。この推測には理論的な根拠があるのだろうか。方策のパラメータ θ が動いたとき、期待収益はどのように変化するのだろうか。

Ronald Williams は 1992 年にこの問いに答えた (Williams 1992)。推測した更新の平均は期待収益の勾配（gradient）に一致するため、この推測は確率的勾配上昇法（stochastic gradient ascent）そのものになっており、Williams はこのアルゴリズムを REINFORCE と名付けた。エピソードの収益はエージェントが知らない環境の遷移確率に依存するにもかかわらず、この推定に必要なのはサンプリングされたエピソードと方策自身の対数確率だけである。方策を関数 $\pi_\theta(a \mid s)$ として書き下すことは、14.3 節の価値テーブルが抱えていた制約も取り除く。テーブル上で貪欲に行動するには有限個の行動リストにわたる最大値が必要になるが、パラメータ化された方策であればトルクを出力することも、トークンの語彙（vocabulary）上の分布を出力することもできる。同じ恒等式を言語モデル（language model）の次トークン分布に適用すれば、完成した回答に与えられる単一の報酬から推論モデルを訓練できる。

本節では、方策勾配（policy gradient）を持つ最小の問題からこの推定量を導出し、すべての勾配を厳密に計算できる 14.1 節の検証器タスク（verifier task）および湖の環境で実行する。この推定量は平均的には正しいが、更新のたびに新しいエピソードを必要とし、小さなバッチでは大きなノイズが残る。

```
import gymnasium as gym
```

14.7.1 方策のパラメータ化

勾配上昇法には、パラメータに対して滑らかに変化する方策が必要である。決定論的な方策はargmax であり、区分的に定数となる。そのため、パラメータをわずかに動かしても何も変わらないか、行動が不連続に切り替わるかのどちらかになる。一方、確率的な方策の確率は任意の微小量だけ変化できる。

14.7.1.1 ソフトマックス選好

`Policy.tabular`（14.5 節）は、各状態行動対（state-action pair）に対して**選好**（preference）と呼ばれる 1 つのパラメータ $\theta_{s,a}$ を保持する。行動全体にわたるソフトマックスにより、状態 s における選好が確率へと変換される。

$$\pi_\theta(a \mid s) = \frac{e^{\theta_{s,a}}}{\sum_{a'} e^{\theta_{s,a'}}}. \tag{14.59}$$

これは状態のインデックスを入力とする 3.1 節のソフトマックス回帰（softmax regression）であり、行動クローニング（behavior cloning）が当てはめたものと同じ方策である。ゼロテーブルは一様方策（一様分布に従う方策）を与える。$Q(s,a)$ とは異なり、選好は収益の推定値ではない。ある状態におけるすべての選好に同じ定数を加えても、方策は変化しない。

14.7.1.2 スコア関数

ソフトマックスは容易に微分できる。$\log \pi_\theta(a \mid s) = \theta_{s,a} - \log \sum_{a''} e^{\theta_{s,a''}}$ であるから、同じ状態における選好 $\theta_{s,a'}$ に関する導関数（derivative）は、任意の行動 a' に対して次式で与えられる。

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta_{s,a'}} = \mathbf{1}(a' = a) - \pi_\theta(a' \mid s), \tag{14.60}$$

また、他のすべての状態における選好に関する導関数は 0 である。ベクトル $\nabla_\theta \log \pi_\theta(a \mid s)$ は**スコア関数**（score function）と呼ばれる。言葉で言えば、a の対数確率を高めることは、その行動自身の選好を引き上げるとともに、s におけるすべての選好をそれぞれの確率に比例して引き下げることを意味する。e_a で行動 a の指示ベクトルを表すと、s におけるスコアの行は $e_a - \pi_\theta(\cdot \mid s)$ となる。その成分の和は 0 であるため、スコアは方策を変化させない唯一の方向、すなわち s におけるすべての選好に定数を加える方向と直交しており、その方向の成分を持たない。ランダムなテーブルにおいて、この公式を `Policy.log_prob` の自動微分（automatic differentiation）と照らし合わせて確認しよう。

```
rng = np.random.default_rng(0)
check = d2l.Policy.tabular(16, 4)
with torch.no_grad():   # move to a generic table, off the uniform start
    check.policy.weight.copy_(torch.as_tensor(rng.standard_normal((16, 4))))
check.log_prob(torch.tensor([6]), torch.tensor([2])).backward()
probs = np.exp(check.log_prob_np(np.repeat(6, 4), np.arange(4)))
hand = np.zeros((16, 4), dtype=np.float32)
hand[6], hand[6, 2] = -probs, 1 - probs[2]
print(bool(torch.allclose(check.policy.weight.grad, torch.as_tensor(hand),
                          atol=1e-6)))
```

```
True
```

ライブラリの `log_prob` の状態 6・行動 2 における勾配は Equation 14.60 と一致し、テーブルの他のすべての行では 0 になっている。

14.7.2 方策勾配

期待収益の勾配はどのような形をしており、エージェント自身のエピソードから推定できるのだろうか。その答えは状態が 1 つの場合に最も明快に現れ、軌跡（trajectory）に対しても同じ形を保つ。

14.7.2.1 最小の事例

1 つの状態と、選好 θ_1 および θ_2 を持つ 2 つの行動を考えよう。行動 1 は報酬 1 を獲得し、行動 2 は 0 を獲得する。$p = \pi_\theta(1)$ とおく。期待報酬（expected reward）は $J(\theta) = p$ であり、Equation 14.60 より θ_1 に関するその導関数は $p(1-p)$ となる。ここで、p を直接計算することはできず、行動を試すことしかできないとしよう。それでもなお、この導関数は方策自身の行動にわたる平均として書き表せる。各項に $\pi_\theta(a)$ を掛けて割ると、次式が得られる。

$$\begin{aligned}\frac{\partial J}{\partial \theta_1} &= \sum_a r(a)\,\frac{\partial \pi_\theta(a)}{\partial \theta_1} = \sum_a \pi_\theta(a)\,r(a)\,\frac{\partial \log \pi_\theta(a)}{\partial \theta_1} \\ &= \mathbb{E}_{a\sim\pi_\theta}\left[r(a)\,\frac{\partial \log \pi_\theta(a)}{\partial \theta_1}\right].\end{aligned}$$

したがって、サンプリングされた 1 つの行動について報酬にそのスコアを掛けたものは、この導関数の推定値になる。確率 p でサンプルは行動 1 となり、その報酬は 1、スコアは $1-p$ である。確率 $1-p$ で報酬は 0 となり、推定値も 0 になる。平均は $p(1-p)$ であり、導関数そのものに一致する。二次モーメント（second moment）は $p(1-p)^2$ であるため、標準偏差は $(1-p)\sqrt{p(1-p)}$ となる。これと平均との比は次の値になる。

$$\sqrt{\frac{1-p}{p}} \tag{14.61}$$

すなわち、標準偏差は平均にこの比を掛けた大きさになる。報酬をもたらす行動がまれなとき、この推定値は平均としては正しいものの、ほぼすべての単一の抽出では役に立たない。たいていは 0 になり、ごくたまに約 $1/p$ 倍も大きすぎる値になるからである。独立な n 回の抽出を平均すると、この比は $\sqrt{n}$ で割られる。

湖の環境も、学習開始時点では同じ構造を持つ。一様方策がゴールに到達するのはエピソードの約 1.4% であり（14.2 節）、これは滑らない氷でも滑る氷でも変わらない。一様にランダムな指示は一様にランダムな移動になるからである。ゴールを逃したエピソードは何も獲得しない。$p = 0.014$ では上の比は 8.4 になり、16 エピソードからなるバッチに成功が 1 つも含まれない確率は $(1-0.014)^{16} \approx 0.80$ となる。つまり、5 バッチのうち 4 バッチは推定値 0 を返す。本節の残りでは、2 つの行動を軌跡に置き換えて同じ計算を実行する。

14.7.2.2 軌跡上の最適化問題

湖の環境におけるエピソードは軌跡 $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$ であり、最初の終端状態（terminal state）s_T で終わるため、その長さ T は確率的に決まる。エージェントが各行動を選

び、環境が次の各状態を選ぶため、軌跡の確率は方策の因子、遷移カーネル（transition kernel）の因子、および初期分布 μ_0（14.1 節）の積となる。

$$P(\tau;\theta) = \mu_0(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t \mid s_t)\,P(s_{t+1} \mid s_t, a_t). \tag{14.62}$$

θ に依存するのは方策の因子だけである。最適化の目的関数は、$R(\tau)=\sum_{t=0}^{T-1}\gamma^t r_t$ とおいたときのこれらの軌跡にわたる期待収益（期待値）である。

$$J(\theta) = \mathbb{E}_{\tau\sim P(\cdot;\theta)}\big[R(\tau)\big] = \sum_{\tau} R(\tau)\,P(\tau;\theta). \tag{14.63}$$

湖の環境は固定された初期状態を持つため、$J(\theta)=V^{\pi_\theta}(s_0)$ と 14.2 節の記法で書ける。検証器タスクでは、μ_0 は 4 つのプロンプト上の一様分布である。

14.7.2.3 対数微分トリック

最小の事例での手順がそのまま当てはまる。固定された軌跡の収益 $R(\tau)$ は θ に依存せず、その確率だけが依存する。勾配を和の中に入れ、$P(\tau;\theta)$ を掛けて割ると、次式が得られる。

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{\tau} R(\tau)\,\nabla_\theta P(\tau;\theta) = \sum_{\tau} P(\tau;\theta)\,R(\tau)\,\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}\\ &= \sum_{\tau} P(\tau;\theta)\,R(\tau)\,\nabla_\theta \log P(\tau;\theta),\end{aligned}$$

すなわち、次のようになる。

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim P(\cdot;\theta)}\Big[R(\tau)\,\nabla_\theta \log P(\tau;\theta)\Big]. \tag{14.64}$$

恒等式 $\nabla \log P = \nabla P / P$ は**対数微分トリック**（log-derivative trick）と呼ばれる。これにより、勾配は現在の方策が生成する軌跡にわたる平均へと戻され、サンプリングによって求められるようになる。エピソードは任意のステップ数だけ続きうるため、軌跡にわたる和は無限和である。それでも勾配を和の中に入れる操作は正当化される。ソフトマックス方策のもとではどの状態も数ステップ以内に終端状態へ到達する正の確率を持つため、長いエピソードは指数関数的にまれであり、各ステップが加えるスコアの長さも有界だからである。

14.7.2.4 遷移確率の相殺

勾配の式にはまだ $\log P(\tau;\theta)$ が含まれており、$P(\tau;\theta)$ にはエージェントが知らない遷移カーネルが含まれている。しかし、積 Equation 14.62 の対数は和になり、微分すると方策の項だけが残る。

$$\nabla_\theta \log P(\tau;\theta) = \nabla_\theta \Big[\log \mu_0(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t \mid s_t) + \sum_{t=0}^{T-1} \log P(s_{t+1} \mid s_t, a_t) \Big]$$
$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

Figure 14.18 に因子の連鎖を示す。初期分布とカーネルは θ に依存しないため、それらの対数の勾配は 0 になる。どの軌跡がサンプリングされるかを決めるのは依然としてカーネルであるが、推定量がカーネルの値を評価することは一切ない。これがこの手法を**モデルフリー**にする理由であり、同時にこの導出が必要とする仮定、すなわちダイナミクスが方策のパラメータに依存してはならないという条件を明確にしている（演習問題 1 ではこの仮定を外す）。

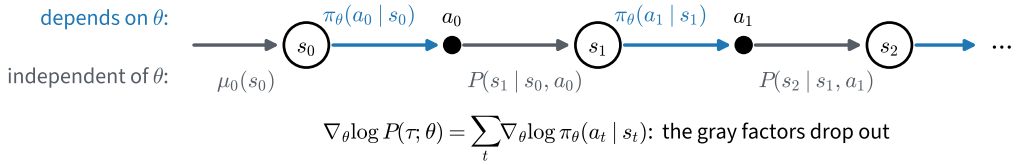


Figure 14.18: 因子の連鎖として表した軌跡の確率 $P(\tau;\theta) = \mu_0(s_0) \prod_t \pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t)$。方策の因子（青、連鎖の上側）だけが θ に依存する。初期分布と遷移カーネル（灰色、下側）は依存しないため、それらの対数の勾配は 0 となり、軌跡のスコアは方策のスコアの和となって P を知らずに計算できる。

14.7.2.5 REINFORCE 推定量

スコアの和を Equation 14.64 に代入し、期待値を現在の方策からサンプリングした n 本の軌跡 $\tau_1, \dots, \tau_n$ にわたる平均に置き換えると、次式が得られる。

$$\widehat{u} = \frac{1}{n} \sum_{i=1}^{n} R(\tau_i) \sum_{t=0}^{T_i-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.65}$$

各軌跡は、自らがとったすべての行動の対数確率を、その収益に比例する量だけ引き上げる。これは冒頭で述べた推測そのものであり、各項の期待値は $\nabla_\theta J(\theta)$ に等しいため、$\widehat{u}$ は勾配の不偏推定値（統計的バイアスのない推定値）となる。REINFORCE (Williams 1992) は上昇ステップ $\theta \leftarrow \theta + \eta\, \widehat{u}$ をステップサイズ（step size）η で実行し、続いて更新後の方策から新しいバッチをサンプリングする。湖の環境ではすべての収益が非負であるため、サンプリングされた行動はすべて引き上げられる。サンプリングされなかった行動はスコアの $-\pi_\theta(\cdot \mid s)$ の部分を通じて確率を失い、収益 0 のエピソードは何も変化させない。

Figure 14.19 は、4 行動のソフトマックスでは曲線として描けない連続行動の 1 ステップ問題について、1 回の更新ステップを図示したものである。方策はガウス分布 $\mathcal{N}(a;\mu,1)$ であり、平均に関するそのスコアは $a - \mu$ となる。1 エピソードは単一の行動 a であるから、その収益 $R(a)$ はその行動の報酬に一致し、更新ステップは $\mu \leftarrow \mu + \eta \frac{1}{n} \sum_i R(a_i)(a_i - \mu)$ となる。すなわち、平均は最も多くの報酬を得たサンプル行動の方向へ移動する。12 個のサンプルによって平均は 0.32 へ移動するが、厳密な勾配であれば 0.52 へ移動するところである。

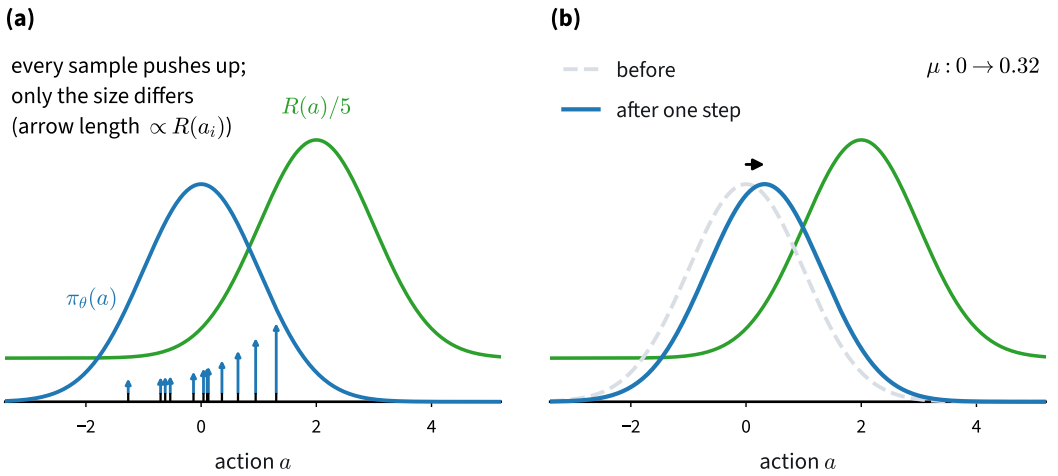


Figure 14.19: 1 ステップ問題における REINFORCE の 1 ステップ。方策は $\pi_\theta(a) = \mathcal{N}(a; \mu, 1)$（ただし $\mu = 0$）、報酬は $R(a) = 0.4 + 2e^{-(a-2)^2/2}$ であるため、期待報酬は $J = 0.92$、厳密な勾配は $\mathrm{d}J/\mathrm{d}\mu = 0.52$ である。(a) サンプリングされた 12 個の行動はそれぞれ、自身の報酬で重み付けされて対数確率を引き上げる。すべての報酬が正であるため、矢印はすべて上を向く。(b) ステップ $\mu \leftarrow \mu + \eta \frac{1}{12}\sum_i R(a_i)(a_i - \mu)$（$\eta = 1$）は平均を 0 から 0.32 へ、報酬の高い行動の方向へ移動させる。厳密な勾配であれば 0.52 へ移動していたはずである。

14.7.3 共通例題における REINFORCE

検証器タスクは 1 エピソードあたり 1 ステップしかなく状態遷移も持たないため、あらゆる量を厳密に計算できる。一方、湖の環境は複数ステップにわたるエピソードと、更新処理が決して参照しない遷移カーネルを持つ。

14.7.3.1 検証器タスク

14.1 節の検証器タスクには、4 つの算数のプロンプト x と、それらすべてで共有される 9 個の応答候補 y がある。方策はプロンプトごとに選好の行を 1 つ持つテーブル θ であり、$\pi_\theta(y \mid x)$ は第 x 行のソフトマックスで与えられ、検証器は正解に対して $r(x, y) = 1$、それ以外に対して 0 を与える。軌跡はプロンプトに対する単一の応答であり、その確率は $\mu_0(x)\,\pi_\theta(y \mid x)$ でカーネルは一切含まれない。第 x 行（θ の行）はプロンプト x の期待スコア $J_x(\theta) = \sum_y \pi_\theta(y \mid x)\, r(x, y)$ のみに影響し、状態をプロンプトに置き換えて Equation 14.60 を適用すれば、その厳密な勾配は 9 個の項の和となる。

$$\nabla_{\theta_x} J_x(\theta) = \sum_y \pi_\theta(y \mid x)\, r(x, y)\, (e_y - \pi_\theta(\cdot \mid x)). \tag{14.66}$$

REINFORCE は、この和をサンプリングした K 個の応答にわたる平均に置き換える。`verifier_step` は各プロンプトに対して K 個の応答をサンプリングし、関数 `weighting` でそれらの報酬を重みに変換し、θ の行を `lr` $= \eta$ に K 個のスコアの重み付き平均を掛けた分だけ動かす。`weighting = lambda r: r` とすれば重みは報酬そのものになり、更新ステップは Equation 14.65 と一致する（14.8 節では別の重み付けを渡す）。判定器は引数 `reward` として渡されるため、同じステップを用いて異なる検証器に対して訓練できる。

```
def verifier_step(task, theta, K, reward, rng, weighting, lr=2.0):
    """One REINFORCE step on a VerifierTask: K responses per prompt, each
    score e_y - pi(.|x) (eq_softmax_score) weighted by weighting(rewards of
    the prompt's K responses), averaged over the K, one ascent step."""
    pi, g = task.policy(theta), np.zeros_like(theta)
    eye = np.eye(len(task.resp))
    for x in range(len(task.prompts)):         # one group of K per prompt
        ys = rng.choice(len(task.resp), K, p=pi[x])
        w = weighting(np.array([reward(x, y) for y in ys]))
        g[x] = (eye[ys] - pi[x]).T @ w / K
    return theta + lr * g
```

完全一致の検証器は正解の応答 y^* にしか報酬を与えないため、プロンプト x に対する推定値は $\frac{c}{K}\left(e_{y^*} - \pi_\theta(\cdot \mid x)\right)$ となる。ここで c は K 個のうち正解であった応答の個数である。個数 c は成功確率 $p = \pi_\theta(y^* \mid x)$ の二項分布に従うため、この推定値の平均は厳密な勾配である $p\left(e_{y^*} - \pi_\theta(\cdot \mid x)\right)$ に一致し、相対標準偏差は $\sqrt{(1-p)/(Kp)}$、すなわち最小の事例における比を $\sqrt{K}$ で割った値になる。参照テーブルでは回答可能な 2 つのプロンプトにおいて $p \approx 1/8$ であるため、$K = 16$ のグループによる推定値は勾配の長さの約 0.66 倍だけずれると予測される。

次のコードセルでこの予測を確かめよう。参照テーブルにおいて 10,000 個の推定値を抽出し（`lr=1.0` とすると更新後の値から θ を引いた差が推定値になる）、さらに $\eta = 2$ で 1 ステップ進める。

```
task = d2l.VerifierTask()
theta0, pi0, R = task.theta_ref, task.policy(task.theta_ref), task.R
exact = pi0 * R - pi0 * (pi0 * R).sum(axis=1, keepdims=True)  # nine terms
rng = np.random.default_rng(0)
U = np.stack([verifier_step(task, theta0, 16, task.verify, rng, lambda r: r,
                            lr=1.0) - theta0 for _ in range(10_000)])
err = lambda u: np.linalg.norm(u - exact) / np.linalg.norm(exact)
print(f'one K=16 estimate is off by '
      f'{np.sqrt(np.mean([err(u) ** 2 for u in U])):.2f} of the gradient (RMS)')
print('the mean of M of them: ' + ', '.join(
    f'M={M}: {err(U[:M].mean(axis=0)):.3f}' for M in (10, 100, 1000, 10_000)))
print(f'groups without a success, per prompt: '
      f'{np.round((U == 0).all(axis=-1).mean(axis=0), 2)}')
after = [task.success(task.policy(theta0 + 2.0 * u)) for u in U]
print(f'success {task.success(pi0):.3f}; one step with lr = 2 gives '
      f'{task.success(task.policy(theta0 + 2.0 * exact)):.3f} (exact gradient),'
      f' median {np.median(after):.3f} (sampled)')
```

```
one K=16 estimate is off by 0.66 of the gradient (RMS)
the mean of M of them: M=10: 0.126, M=100: 0.061, M=1000: 0.047, M=10000: 0.009
groups without a success, per prompt: [0.12 0.12 1.   1.  ]
success 0.062; one step with lr = 2 gives 0.077 (exact gradient), median 0.077 (sampled)
```

予測どおり、1 回の推定値は勾配の長さの 0.66 倍だけずれている。M 個の推定値の平均の誤差は M が増えるにつれて平均的に $0.66/\sqrt{M}$ に従って下がり続け、下げ止まりは見られない。すなわち、この推定量は不偏である。各推定値は回答可能なプロンプトごとに 1 つの二項カウントに帰着するため、1 回の平均値はそのレートよりかなり小さく出ることもあれば、最大で約 2 倍大きく出ることもある。回答可能な各プロンプトでは、グループの約 12% ($(7/8)^{16} \approx 0.12$) が

正解の応答を 1 つも含まず、何も寄与しない。どの候補も検証器を通過しない 8*8 と 9*9 では、すべてのグループが空になる。どの応答も解けないプロンプトに対しては、いくらサンプリングを重ねても勾配は一切生じない。$\eta = 2$ による厳密な 1 ステップは成功率を 0.062 から 0.077 へと引き上げ、サンプリングによるステップも中央値（median）では同程度の改善を示す。このタスクでは重みが付くスコアが正解の応答のものだけであるため、サンプリングによるステップが成功率を下げることは決してない。

14.7.3.2 湖

湖の環境では 1 エピソードが複数ステップからなり、遷移カーネルが関与する。ここではこれまでの節で用いた滑る氷ではなく、すべての指示が意図どおりにエージェントを移動させる滑らない湖を用いる。その理由は、1 つのエピソードが 1 つの行動についてどれだけの情報を持つかにある。一様な初期方策では、どちらのマップでも同じランダムウォークとなり、成功確率も同じ約 1.4% になる。しかし滑る氷の上では、ある指示はそれ自身の方向または直交する 2 方向のいずれかへそれぞれ確率 $1/3$ でエージェントを動かすため、指示が制御できるのは反対方向へは進まないという一点だけである。その結果、一様な初期方策において滑る氷での厳密な勾配の長さは滑らない氷の 3 分の 1 になる一方、1 エピソード推定値のノイズはほぼ変わらないため、同じ精度を得るには 9 倍大きなバッチが必要になる（演習問題 3 でこの係数を導出する）。

方策が改善するにつれて、この差はさらに広がる。滑る氷の上での成功エピソードは 6 ステップではなく数十ステップ続くため、1 回の成功がはるかに多くの行動の間で分け合われ、しかも γ^{T-1} で割り引かれるからである。それでも REINFORCE は滑る氷の上でもゆっくりとながら学習する（演習問題 6）。

このマップに対する Gymnasium のデフォルトの 100 ステップ上限は外しておく。目的関数 Equation 14.63 は完全なエピソード全体にわたって定義されており、途中で打ち切られたエピソードの報酬和はその収益ではないからである（14.1 節）。ソフトマックス方策はすべての行動に正の確率を与えるため、このマップではすべてのエピソードが確率 1 で終了し、10,000 ステップという上限はここでの実行が生み出すどのエピソードよりもはるかに大きい安全策にすぎない。

```
gamma = 0.95
env = gym.make('FrozenLake-v1', is_slippery=False, max_episode_steps=10_000)
```

`Batch` は複数エピソードの遷移をフラットな配列に格納し、あわせて各エピソードの最終ステップの 1 つ先のインデックスを保持する。`episode_returns` は各エピソードに対して $R(\tau)$ を計算し、`per_step` はエピソードごとに 1 つの値をそのエピソードの全ステップにわたって繰り返すことで、収益を各行動の重みへと変換する。このコンテナは次の状態と、**終端**（terminated）のみを表し**打ち切り**（truncated）は決して含めないフラグ `term` も記録する。14.1 節の規則により、途中で打ち切られたエピソードは終端状態に到達していないからである。ここでの湖のエピソードはすべて終端に達するため、このフラグが意味を持つのはエピソードが打ち切られるようになってからである。

```
class Batch:
    """A flat batch of transitions, plus the episode boundaries."""
    def __init__(self, obs, act, rew, next_obs, term, ep_ends):
        self.obs, self.act, self.rew = obs, act, rew
        self.next_obs, self.term = next_obs, term
        self.ep_ends = ep_ends    # one past the last step of each episode

    def __len__(self):
```

```
        return len(self.rew)

    def episodes(self):
        """Yield one slice per episode."""
        start = 0
        for end in self.ep_ends:
            yield slice(start, end)
            start = end

    def episode_returns(self, gamma=1.0):
        """R(tau), the discounted return, one number per episode."""
        return np.array([(gamma ** np.arange(ep.stop - ep.start)
                          * self.rew[ep]).sum() for ep in self.episodes()])

    def per_step(self, values):
        """Repeat one value per episode over that episode's steps."""
        return np.repeat(values, np.diff(self.ep_ends, prepend=0))
```

`rollout` はバッチにデータを詰める関数である。関数 `policy(obs, rng) -> action`（14.2 節の `evaluate` と同じ行動プロトコル）として書かれた任意の方策で完全なエピソードを実行し、各ステップの状態、行動、報酬、次の状態、および終端フラグを追加していく。エージェントの乱数性はすべて NumPy の乱数生成器 `rng` に由来するため、シードを固定すればサンプリングされるすべてのエピソードが確定する。

```
def rollout(env, policy, num_episodes, rng):
    """Collect complete episodes from `policy(obs, rng) -> action` as a
    Batch. `term` records `terminated`, never `truncated`: a time-limit cut
    is not a terminal state. All sampling runs through the numpy `rng`."""
    cols, ep_ends = [[] for _ in range(5)], []
    for _ in range(num_episodes):
        obs, done = env.reset()[0], False
        while not done:
            act = policy(obs, rng)
            next_obs, reward, terminated, truncated, _ = env.step(act)
            done = terminated or truncated
            for col, val in zip(cols, (obs, act, reward, next_obs,
                                       float(terminated))):
                col.append(val)
            obs = next_obs
        ep_ends.append(len(cols[0]))
    obs, act, rew, next_obs, term = (np.asarray(c) for c in cols)
    return Batch(obs, act, rew.astype(np.float32), next_obs,
                 term.astype(np.float32), np.asarray(ep_ends))
```

訓練ループでは 14.5 節の `policy_step` を再利用する。これは $\frac{1}{N}\sum_t w_t \log \pi_\theta(a_t \mid s_t)$ に対して、バッチの N ステップにわたりオプティマイザを 1 ステップ進める関数である。行動クローニングでは $w_t = 1$ を渡したが、REINFORCE ではステップ t を含むエピソードの収益を渡す。内部的には `policy_step` はこの和に負号を付けた**代理損失**（surrogate loss）を最小化するが、その損失の値自体は方策の性能について何も語らない。重要なのは現在の θ におけるその勾配、すなわち推定値だけである。

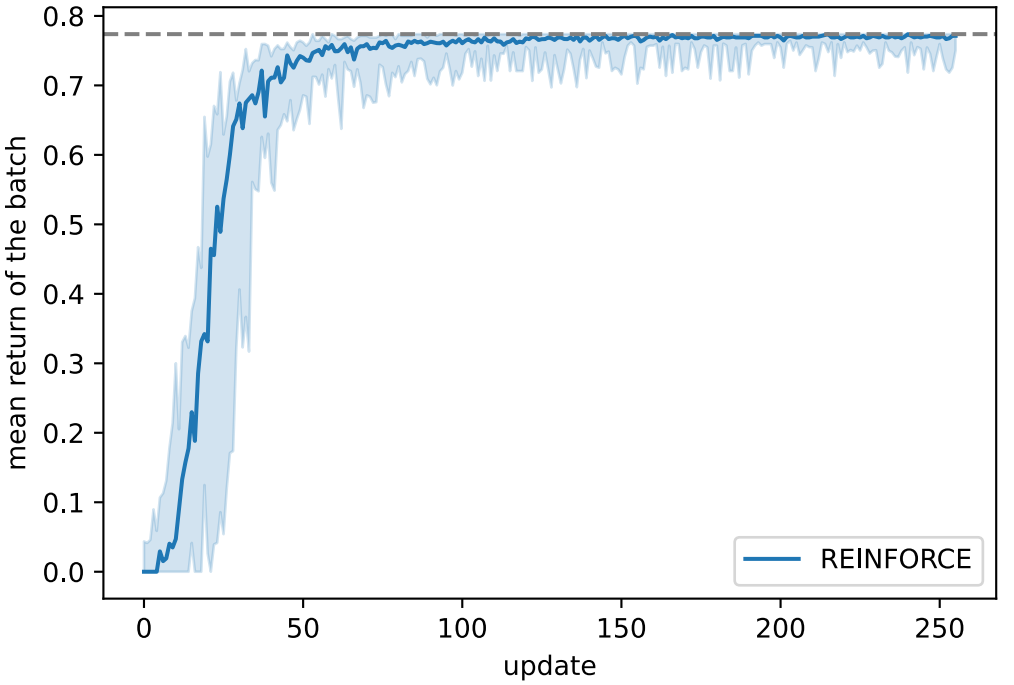
Equation 14.65 とは細部が 2 点異なる。第 1 に、`policy_step` が割るのはステップ数 N でありエ

ピソード数 n ではないため、推定値はバッチの平均エピソード長（バッチごとに変動する係数）で割られることになる。第 2 に、オプティマイザは `Policy.tabular` の学習率（learning rate）0.1 を持つ Adam であり、素の更新ステップ $\theta \leftarrow \theta + \eta\hat{u}$ ではない。素の勾配上昇法は、ある行動の確率が小さいところでは極端に遅くなる。Equation 14.60 により、その選好の勾配がその確率を因数として持つからである。これに対し、Adam は各座標をほぼ学習率の大きさだけ動かす（演習問題 6 で両者を比較する）。以下のループは新しいバッチを収集し、各ステップをそのエピソードの収益で重み付けして 1 ステップ更新し、バッチの平均収益とステップ数を返す。

```
def train_reinforce_tabular(ac, env, gamma, seed, num_updates=256,
                            batch_episodes=16):
    """REINFORCE: a fresh batch from the current policy, every step of a
    trajectory weighted by that trajectory's return, one ascent step.
    Yields the batch's mean return and its number of steps."""
    rng = np.random.default_rng(seed)      # one stream for all sampling
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = rollout(env, ac.act, batch_episodes, rng)
        R = batch.episode_returns(gamma)
        d2l.policy_step(ac, batch.obs, batch.act, batch.per_step(R))
        yield float(R.mean()), len(batch)
```

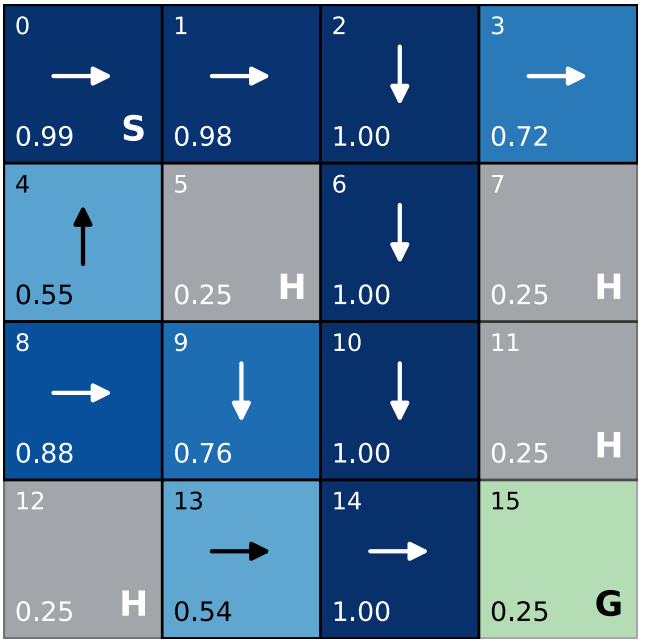
それぞれ異なるシードを持つ 12 個のエージェントを一様方策から訓練し、各訓練バッチの平均収益（$J(\theta)$ の推定値）を、6 ステップ経路の上限である $\gamma^5 \approx 0.774$ とともにプロットする。

```
agents = [d2l.Policy.tabular(16, 4) for _ in range(12)]  # uniform policies
runs = np.array([list(train_reinforce_tabular(ac, env, gamma, seed))
                 for seed, ac in enumerate(agents)])     # (seed, update, 2)
curves, steps = runs[..., 0], runs[..., 1].astype(int)
d2l.plot_curves({'REINFORCE': curves}, xlabel='update',
                ylabel='mean return of the batch', reference=gamma ** 5)
```



図は 12 回の実行の中央値と、最も遅い実行から最も速い実行までの帯を示している。最小の事例から予測されたとおり、初期のバッチの大半は成功を 1 つも含まず推定値が 0 になるため、中央値は平坦に始まる。ひとたびバッチ内で成功が出ると、そのときの行動が起こりやすくなり、成功の頻度が増して平均収益は急速に上昇する。すべての実行は数十回の更新のうちに破線の上限へ近づくが、方策が依然として確率的であり時折最短経路から外れるため、上限よりわずかに低い水準にとどまる。次のコードセルでは、1 番目のエージェントが学習した方策について、各マスで最も確率の高い行動とその確率を表示する。

```
probs = agents[0].probs(16, 4)
d2l.show_grid(env.unwrapped.desc, probs.max(axis=1), probs.argmax(axis=1))
```



方策はゴールへ至る 3 本の最短経路のうち、たまたま最初に成功したエピソードが通った 1 本に集中している。穴とゴールでは一度も行動がとられないため、一様分布のままである。選ばれた経路から外れたマスには、初期の成功エピソードがそこを通過した際に獲得した選好がほぼそのまま残っている。それらのマスが変化するのはエピソードがたまたま迷い込んだときだけだが、経路が固まるにつれてそれはまれになる。訪問と訪問の間は Adam のモメンタム（momentum）が直前の変化を数十回の更新にわたって引き継ぐが、その後は動かなくなる。このループのどこにも遷移カーネルを評価する処理はない。`rollout` はシミュレータがサンプリングした次の状態を記録しただけであり、更新処理が読み取ったのは状態・行動・収益のみである。これこそが 14.7.2.4 節の相殺が実際に働いている姿であり、カーネルを持たない検証器タスクでは見ることのできなかった現象である。

14.7.4 方策勾配定理

REINFORCE は、ある行動がとられるより前に届いた報酬も含め、軌跡全体の収益によってすべての行動のスコアを重み付けする。しかし、行動がすでに獲得された過去の報酬に影響を与えることはできないため、それらの報酬は平均的には何も寄与しないはずである。方策勾配定理（policy gradient theorem）(Sutton et al. 2000) はこの事実を厳密に定式化し、勾配を軌跡全体ではなく方策が訪問する状態にわたる平均として書き直す。この定理を述べるため、エピソード終了後もマルコフ連鎖は報酬 0 で終端状態にとどまり続けるものとし、s_t がすべての t に対して定義されるようにする。$d_t^{\pi_\theta}(s) = \Pr(s_t = s)$ を s_t が π_θ のもとで従う分布とし、さらに次式をおく。

$$d_\gamma^{\pi_\theta}(s) = (1-\gamma)\sum_{t \geq 0} \gamma^t \, d_t^{\pi_\theta}(s) \tag{14.67}$$

これは時間に関するその割引平均であり、**割引状態占有分布**（discounted state-occupancy distribution）と呼ばれる。係数 $1-\gamma$ によって重み γ^t の総和が 1 になるため、$d_\gamma^{\pi_\theta}$ は状態上の確率分布（probability distribution）となる。

14.7.4.1 主張と証明

命題（方策勾配定理）。 **目的関数 $J(\theta) = \mathbb{E}_{s_0 \sim \mu_0}[V^{\pi_\theta}(s_0)]$ に対して、次式が成り立つ。**

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t \geq 0} \gamma^t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, Q^{\pi_\theta}(s_t, a_t)\Big] \\ &= \frac{1}{1-\gamma}\, \mathbb{E}_{s \sim d_\gamma^{\pi_\theta},\, a \sim \pi_\theta(\cdot \mid s)}\Big[\nabla_\theta \log \pi_\theta(a \mid s) \, Q^{\pi_\theta}(s, a)\Big] \end{aligned} \tag{14.68}$$

証明。 Equation 14.64 と相殺により、$\nabla_\theta J = \mathbb{E}\big[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, R(\tau)\big]$ である。各 t において、収益を t より前の報酬と t 以降の報酬に分割する。t より前の報酬はエージェントが s_t に立った時点で確定している一方、a_t はこれから抽出される段階にあり、a_t のスコアは s_t が与えられたもとで平均が 0 になる。実際、$\sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) = \sum_a \nabla_\theta \pi_\theta(a \mid s) = \nabla_\theta 1 = 0$ である。したがって、過去の報酬の寄与は 0 となる。t 以降の報酬については (s_t, a_t) で条件付けると、マルコフ性（Markov property）によりそれらの割引和の期待値は $\gamma^t Q^{\pi_\theta}(s_t, a_t)$ となる。これより第 1 の形式が得られる。第 2 の形式を得るには、t にわたる和を訪問した状態ごとにまとめればよい。任意の関数 f に対して $\mathbb{E}\big[\sum_t \gamma^t f(s_t, a_t)\big] = \sum_s \sum_t \gamma^t d_t^{\pi_\theta}(s)\, \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s)} f(s, a)$ が成り立つため、ここに $d_\gamma^{\pi_\theta}$ の定義を代入すればよい。終端状態では $Q^{\pi_\theta} = 0$ であるため、エピソード終了後のステップは何も寄与しない。■

14.7.4.2 帰結

この定理からは 3 つの帰結が導かれる。第 1 に、時刻 t におけるスコアが必要とするのは t 以降の報酬だけである。証明の中で過去の報酬は平均 0 のノイズとして取り除かれたため、それらを落とした推定量は同じ平均を持ち、通常はノイズも小さくなる。第 2 に、勾配は方策が訪問する状態にわたる平均であり、どれだけ頻繁に、かつどれだけ早い時点で訪問するかによって重み付けされる。そのため、方策がめったに到達しない状態はわずかな勾配しか受け取らず、一度も訪問しなかったバッチではその行の推定値が厳密に 0 になる。先ほど学習された経路から外れたマスがほとんど変化しなかった理由はここにある。第 3 に、多くの実装は第 1 の形式から因子 γ^t を落とす。これは $d_\gamma^{\pi_\theta}$ を割引なしの状態訪問に置き換え、遅い時点で訪問される状態の相対的な重みを変えるものであり、一般にはいかなる目的関数の勾配にもならない更新をもたらす (Nota and Thomas 2020)。演習問題 7 でその違いを詳しく調べる。

14.7.5 コストと限界

14.7.5.1 オンポリシーデータの要件

推定量 Equation 14.65 は、**現在の**方策から抽出された軌跡を平均する。1 回更新するとパラメータが変化するため、すでに収集した軌跡はもはや $P(\tau;\theta)$ に従わない。この要件を持つ手法は**オンポリシー**と呼ばれ、更新のたびに新しいバッチを収集する。これに対し、Q 学習は**オフポリシー**である。その更新は、任意の行動方策（behavior policy）が生成した遷移から貪欲方策（greedy policy）について学習する（14.3 節）。REINFORCE が使用するデータ量を環境のステップ数で数えてみよう。

```
hit = (curves >= 0.7).argmax(axis=1) + 1   # batches until the mean is 0.7
to_hit = np.array([s[:k].sum() for s, k in zip(steps, hit)])
for name, x in [('batches until a batch mean reaches 0.7', hit),
                ('environment steps until then', to_hit),
                ('environment steps of the full run', steps.sum(axis=1))]:
```

```
        print(f'{name:>38}: median {np.median(x):>6,.0f}, '
              f'range {x.min():,} to {x.max():,}')
```

```
batches until a batch mean reaches 0.7: median     32, range 25 to 49
          environment steps until then: median  4,313, range 3,210 to 6,191
     environment steps of the full run: median 27,049, range 26,058 to 28,383
```

バッチ平均が初めて 0.7（上限の約 90%）に達するのは 25〜50 回の更新後であり、環境のステップ数にして 3,000〜7,000 ステップに相当する。256 回の更新を行う実行全体では約 27,000 ステップを消費するが、これは方策の改善が止まった後も各更新が新しいバッチを収集し続けるためである。同じマップにおいて、$\alpha = 0.9$ の表形式の Q 学習はほぼすべての実行でゴールに到達する貪欲テーブルを発見し、実行全体に要したのは約 3,500 ステップであった（14.3 節）。1 つのバッチを複数回の更新に再利用するのが自然な対策に思えるが、補正なしでは理論的に破綻する（演習問題 4）。

14.7.5.2 不偏性と分散

湖の環境もモデルが既知であるため、検証器タスクと同様に厳密な勾配と照らし合わせて検証できる。14.2 節では、方策の価値は連立一次方程式（linear system）$V^{\pi} = r_{\pi} + \gamma P_{\pi} V^{\pi}$ の解として求まった。ここで $P_{\pi}(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s, a)$ および $r_{\pi}(s) = \sum_a \pi(a \mid s)\, r(s, a)$ である。したがって、目的関数は次式で表される。

$$J(\theta) = V^{\pi_\theta}(s_0) = \left[\left(I - \gamma P_{\pi_\theta} \right)^{-1} r_{\pi_\theta} \right]_{s_0}. \tag{14.69}$$

P_{π_θ} と r_{π_θ} のすべての成分は θ の滑らかな関数であるため、連立一次方程式の求解を通した自動微分によって厳密な $\nabla_\theta J$ が得られる。`TabularMDP.from_gym` では穴とゴールが報酬 0 の吸収状態（absorbing state）になっているため、この求解にはエピソードの終端も織り込まれている。シード 3 の実行において上昇が始まっている 16 回更新後の方策を `probe` として固定しよう。

```
probe = d2l.Policy.tabular(16, 4)
list(train_reinforce_tabular(probe, env, gamma, 3, num_updates=16))  # train
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

`policy_objective` は選好テーブル、カーネル `P[s, a, s']`、および報酬 `r[s, a]` から $J(\theta)$ を Equation 14.69 によって計算し、フレームワークの `grad` が連立一次方程式の求解を通してそれを微分する。このプローブにおける J とその厳密な勾配を計算しよう。

```
def policy_objective(theta, P, r, gamma):
    """J(theta) = V(s0) of the softmax policy, a differentiable linear solve."""
    pi = torch.softmax(theta, -1)
    P_pi = torch.einsum('sa,sat->st', pi, P)
    return torch.linalg.solve(torch.eye(len(P)) - gamma * P_pi,
                              (pi * r).sum(-1))[0]

P, r = torch.as_tensor(mdp.P).float(), torch.as_tensor(mdp.r).float()
theta = probe.policy.weight.detach()
g_exact = torch.func.grad(policy_objective)(theta, P, r, gamma).numpy()
print(f'J(theta) after 16 updates = '
      f'{float(policy_objective(theta, P, r, gamma)):.3f}, '
```

```
        f'against the ceiling gamma^5 = {gamma ** 5:.3f}')
```

```
J(theta) after 16 updates = 0.313, against the ceiling gamma^5 = 0.774
```

プローブは一様方策と上限のちょうど中間あたりに位置している。ここで θ を固定し、推定量を何度も抽出してみよう。`tabular_score` はバッチの重み付きスコア $w_t\,(e_{a_t} - \pi_\theta(\cdot \mid s_t))$ を Equation 14.60 に従ってそれぞれの状態の行へ足し合わせる。`reinforce_estimate` は `verifier_step` と同様に、これを用いて Equation 14.65 の 1 回の抽出を計算する。すなわち、各ステップをそのエピソードの収益で重み付けし、和を n で割る。各バッチサイズ（batch size）n について 50 個の推定値を抽出し、$\|\nabla_\theta J\|$ に対する相対値として 3 つの数値を出力する。1 回の抽出の二乗平均平方根誤差（root-mean-square error）、50 個の平均の誤差、そして不偏推定量（unbiased estimator）がその平均に残すはずの誤差（1 つ目の数値を $\sqrt{50}$ で割った値）である。

```
def tabular_score(obs, act, w, pi):
    """sum_t w_t (e_{a_t} - pi(.|s_t)), each term added into row s_t: the
    weighted softmax score (eq_softmax_score) of a batch, the shape of pi."""
    u = np.zeros(pi.shape)
    np.add.at(u, obs, w[:, None] * (np.eye(pi.shape[1])[act] - pi[obs]))
    return u

def reinforce_estimate(env, probe, gamma, n, rng):
    """One draw of eq_reinforce at the frozen tabular policy `probe`: each
    step adds its trajectory's return times its score to row s; the sum is
    divided by n."""
    pi = probe.probs(env.observation_space.n, env.action_space.n)
    batch = rollout(env, probe.act, n, rng)
    w = batch.per_step(batch.episode_returns(gamma))
    return tabular_score(batch.obs, batch.act, w, pi).ravel() / n

rng, g = np.random.default_rng(4), g_exact.ravel()
for n in [4, 16, 64]:
    U = np.stack([reinforce_estimate(env, probe, gamma, n, rng)
                  for _ in range(50)])
    zero = int((U == 0).all(axis=1).sum())
    one = np.sqrt(((U - g) ** 2).sum(axis=1).mean()) / np.linalg.norm(g)
    mean = np.linalg.norm(U.mean(axis=0) - g) / np.linalg.norm(g)
    print(f'n={n:>2}: {zero:>2}/50 zero; one draw off by {one:.2f}, the mean '
          f'of 50 by {mean:.2f} (if unbiased: {one / np.sqrt(50):.2f})')
```

```
n= 4:  3/50 zero; one draw off by 3.85, the mean of 50 by 0.48 (if unbiased: 0.54)
n=16:  0/50 zero; one draw off by 1.78, the mean of 50 by 0.24 (if unbiased: 0.25)
n=64:  0/50 zero; one draw off by 0.96, the mean of 50 by 0.10 (if unbiased: 0.14)
```

$n = 4$ では、単一の抽出は推定対象である勾配の長さの 3 倍以上も外れており、バッチ内のどのエピソードもゴールに到達しなかったために 0 になる抽出も現れうる。n を 4 倍にするたびに誤差は半分になり、最小の事例で見た $1/\sqrt{n}$ の法則に従っている。どの n においても 50 回の抽出の平均の誤差は不偏推定量から予測される値と同程度であり、このテストでは統計的バイアスは検出されない。演習問題 5 では抽出回数をさらに大幅に増やし、小さなバイアスでも現れるようにして検証する。

14.7.5.3 プラトーと収束

目的関数は θ に関して凹関数ではない。すでに最小の事例においても、$J = p$ は $\theta_1 - \theta_2$ のロジスティック関数であり、両端で平坦になる。非凹関数に対する勾配上昇法は、一般には性能の低い停留点（stationary point）で止まりうる。しかし、表形式のソフトマックス方策ではそうならない。準最適な停留点は存在せず、初期分布がすべての状態に正の確率を与えるならば、十分に小さいステップサイズによる厳密な勾配上昇法は最適方策（optimal policy）へと収束（convergence）する (A. Agarwal et al. 2021; Mei et al. 2020)。それでも収束が遅くなることはある。ある状態でより良い行動の確率が小さいと、その選好の勾配がその確率を因数として持つため、勾配上昇はプラトーの上を這うようにしか進まず、厳密なソフトマックス勾配上昇法が状態数に対して指数関数的なステップ数を要するマルコフ決定過程（MDP）も存在する (Li et al. 2021)。

固定ステップサイズによるサンプル更新は、2 つ目のリスクをもたらす。たまたま運の良かった少数のバッチによってある状態の方策が悪い行動にほぼ決定論的に偏ってしまうと、その選択を覆すための勾配が極端に小さくなり、実行が長いあいだその行動に固着してしまうのである (Mei et al. 2021)。滑らない湖では 12 回の実行すべてが上限から数パーセント以内の範囲に収まったが、滑る氷の上では同じ手順でもシードごとに異なる水準に落ち着き、最適値を大きく下回るものも現れる。演習問題 6 でその理由を突き止める。

これで、モデルもエキスパートも価値テーブルも使わず、方策自身のエピソードとその報酬だけから方策を改善できるようになった。しかも検証器タスクにおいては、同じ更新式がすでに言語モデルの事後学習（post-training）と同じ形状をしている。すなわち、応答が軌跡であり、報酬はその最後に届く。この推定値は平均的には正しいが、プローブ方策において 1 バッチあたり 4 エピソードとした場合、単一の抽出は勾配の長さの 3 倍以上も外れ、失敗したエピソードは何も寄与しない。続く 14.8 節では、平均を動かさずにそのノイズをどれだけ取り除けるかを問う。

14.7.6 まとめ

方策勾配法は、パラメータ化された方策 π_θ の期待収益 $J(\theta)$ に対する勾配上昇法によって、方策を直接改善する。対数微分トリックにより $\nabla_\theta J$ は方策自身の軌跡にわたる「収益と軌跡のスコアの積」の平均として書け、さらに遷移カーネルがスコアから消えるため、モデルなしでサンプリングされたエピソードから勾配を推定できる。REINFORCE はすべての行動のスコアをそのエピソードの収益で重み付けする手法であり、行動クローニングの更新における単位重みを収益に置き換えたものに相当する。検証器タスクと滑らない湖において、この推定値は平均的には厳密な勾配と一致するが、湖では小さなバッチの推定値は勾配の長さの数倍も外れ、誤差は $1/\sqrt{n}$ の速さでしか縮まらない。方策勾配定理は、行動より前に得られた報酬がノイズとしてしか寄与しないことを示している。また、各更新には現在の方策から収集した新しいバッチが必要である。

14.7.7 演習問題

1. **パラメータ依存のダイナミクス**。14.7.2.4 節の導出では、遷移確率が θ に依存しないと仮定している。しかし推薦システムにおいて、あるアイテムを頻繁に目にするユーザーほどそれをクリックしなくなる場合、この仮定は破られる。
    1. ダイナミクスが $P_\theta(s_{t+1} \mid s_t, a_t)$ であるとする。$\nabla_\theta \log P(\tau;\theta)$ の導出をやり直し、もはや消えなくなる項を示せ。その項を計算するには、エージェントは何を知っている必要があるか。
    2. 1 ステップのモデルにおいて、推薦システムはアイテム 1 を確率 $\pi_\theta(1) = \sigma(\theta)$ で提示し、それ以外ではアイテム 0 を提示するものとする（ここで σ はロジスティック関数である）。提示されたアイテム 1 は確率 $1 - \sigma(\theta)$ でクリックされ、アイテム 0 は決してクリックされず、クリックされると報酬 1 が得られる。$J(\theta)$ と $\mathrm{d}J/\mathrm{d}\theta$ を計算せよ。この導関数を、上の項を落とした REINFORCE 推定値 Equation 14.65 の期待値

と比較せよ。素の勾配上昇法は、これら 2 つのそれぞれを用いたときどこに収束するか。

2. [code] **疎な報酬**。本文の最小の事例では、1 つの状態と 2 つの行動があり、行動 1 の報酬が 1、行動 2 の報酬が 0、かつ $p = \pi_\theta(1)$ であり、1 サンプル推定値は平均の $\sqrt{(1-p)/p}$ 倍の標準偏差を持つ。湖の環境においてこれに相当する量は、一様方策のエピソードがゴールに到達する確率 $p_0 \approx 0.014$ であり、これは 14.2.4 節において $\gamma = 1$ での方策評価によって計算されている。

   1. 成功を 1 つも含まないバッチでは推定値が 0 となり、このとき `policy_step` はゼロテーブルを変化させないため、最初の成功が出るまで方策は一様なまま保たれる。最初の非ゼロ推定値が出るまでの更新回数（その回を含む）はどの分布に従い、16 エピソードのバッチではその平均はいくらか。12 回の訓練実行のそれぞれについてこの回数を記録し、理論予測および訓練曲線の平坦な立ち上がり部分と比較せよ。
   2. 独立な n 個の 1 サンプル推定値の平均が、その平均の $\sqrt{(1-p)/(np)}$ 倍の標準偏差を持つことを示せ。$p = p_0$ において標準偏差を平均と等しくするには、1 バッチあたり何エピソード必要か。訓練によって成功確率が上昇するにつれて、この答えはどのように変化するか。

3. [code] **テーブルにおける方策勾配定理**。●表形式のソフトマックス方策に対して、$\rho_\gamma(s) = \sum_{t \geq 0} \gamma^t d_t^{\pi_\theta}(s)$ で s への割引訪問回数を表し、$d_\gamma^{\pi_\theta} = (1-\gamma)\rho_\gamma$ が Equation 14.68 において成り立つようにする。

   1. Equation 14.68 の状態占有形式と Equation 14.60 を組み合わせ、次式が成り立つことを示せ。
   $$\frac{\partial J}{\partial \theta_{s,a'}} = \rho_\gamma(s)\, \pi_\theta(a' \mid s)\big(Q^{\pi_\theta}(s,a') - V^{\pi_\theta}(s)\big).$$
   2. 演習問題 1b（14.2 節）より、$\rho_\gamma^\top$ は $(I - \gamma P_{\pi_\theta})^{-1}$ のうち s_0 に対応する行である。ρ_γ、V^{π_θ}、Q^{π_θ} を `probe` のパラメータにおいて `mdp.P` と `mdp.r` を用いた連立一次方程式の求解により計算し、上の式を組み立てて結果を `g_exact` と比較せよ。
   3. 滑る氷の上では、指示 a はエージェントを方向 a またはそれに直交する 2 方向のいずれかへそれぞれ確率 $1/3$ で移動させる。一様方策のもとでは両方のマップが同じマルコフ連鎖を誘導し、したがって同じ ρ_γ と V を持つこと、さらに滑る氷での $Q(s,a)$ が $Q_{\text{slip}}(s,a) - V(s) = -\frac{1}{3}\big(Q_{\text{calm}}(s,\bar{a}) - V(s)\big)$（ただし $\bar{a}$ は a と反対向きの指示）を満たすことを示せ。これより、$\theta = 0$ における厳密な勾配の長さは滑る氷の上では 3 分の 1 になることを結論づけ、両方の遷移テーブル上で `policy_objective` を用いて確認せよ。

4. [code] **バッチの再利用**。●`train_reinforce_tabular` を修正し、何の補正も行わずに各バッチ上で `policy_step` を k 回（$k \in \{1, 2, 5, 20\}$）呼び出すようにせよ。そのバッチを収集したときのパラメータを θ_{old} と書く。

   1. 滑らない湖において、バッチの平均収益と、k 回目のステップ後のバッチ全体にわたる比 $\pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ の平均および最大値を記録せよ（両方策の評価には `log_prob_np` を用いる）。各 k において、バッチの平均収益が 0.7 に達するまでに環境のステップ数はどれだけ必要か。また、上の比はどれほど大きくなるか。
   2. k が大きくなるにつれて、更新ステップは固定されたバッチ上の代理目的関数 $\sum_t w_t \log \pi_\theta(a_t \mid s_t)$ の最大点に近づく（ここで w_t はステップ t を含むエピソードの収益である）。これを最大化する表形式方策を閉形式で求め、それを用いて小問 a の結果を説明せよ。

3. 滑るマップ（`is_slippery=True`）において、$k = 1$ と $k = 20$ を最終的な $J(\theta)$（滑る氷の遷移テーブルから Equation 14.69 で計算したもの）によって比較せよ。各実行は数秒で終わる。小問 b の閉形式を用いて、滑らない湖との違いを説明せよ。

5. [code] **不偏性の検証。**●本文の 50 回の抽出によるテストでは、50 回の抽出のノイズより小さい統計的バイアスを見抜くことはできない。より鋭敏なテストは M 個の推定値を平均することである。不偏推定量であれば M が増えるにつれて平均の誤差は縮み続けるが、バイアスのある推定量ではバイアスの大きさで下げ止まる。

   1. `probe` のパラメータにおいて、$M = 25{,}600$ 個の推定値を `reinforce_estimate`$(n = 1)$ で抽出せよ。$\|$`g_exact`$\|$ に対する累積平均の相対誤差を、両対数軸上で M に対してプロットせよ。曲線の傾きを説明せよ。
   2. `max_episode_steps` を 20 および 10 に下げて再構築した環境を `reinforce_estimate` に渡し、`rollout` が一部のエピソードを打ち切りによって終了させる状態で同じ測定を繰り返せ。それぞれの曲線はどの水準で下げ止まるか。打ち切られた推定値は、どの目的関数の勾配に対する不偏推定値になっているか。

6. **ソフトマックスのプラトー。**●1 つの状態と 2 つの行動があり、報酬が 1 と 0、初期選好が $\theta = (0, L)$（ただし $L > 0$）であって、より良い行動の初期確率が $1/(1 + e^{L})$ となる場合を考える。これは 14.7.5.3 節のプラトーを最小の形で取り出したものである。

   1. J が θ に $x = \theta_1 - \theta_2$ を通じてのみ依存し、$J = \sigma(x)$（σ はロジスティック関数）と書けることを示せ。これより、J が凹関数でないことを結論づけよ。
   2. ステップサイズ η による素の勾配上昇法の 1 ステップが、x を $2\eta\,\sigma(x)(1-\sigma(x))$ だけ増加させることを示せ。これを微分方程式として扱い、$\pi_\theta(1) = 1/2$ に $x = -L$ から到達するまでのステップ数が約 $e^{L}/(2\eta)$（ただし L は十分大きいとする）になることを示せ。
   3. 同じ厳密な勾配を Adam に入力すると、勾配の符号が一定であるあいだ、各座標は 1 ステップあたりほぼ学習率 η だけ動く。必要なステップ数を見積もり、小問 b の答えと $L = 10$、$\eta = 0.1$ において比較せよ。この比較は、`Policy.tabular` で Adam を用いることについて何を示唆しているか。
   4. [code] 滑るマップ上で複数のシードを用いて `train_reinforce_tabular` を 1,024 回更新し、最終的な $J(\theta)$ を Equation 14.69 で計算せよ。$V^{*}(s_0)$ を大きく下回って終了した実行の最終方策は、どの状態で、どの行動に対してほぼ決定論的になっているか。そこでの $\|\nabla_\theta J\|$ と一様な初期方策での値を計算せよ。最終的な θ は停留点か。その点および $\theta = 0$ から厳密な勾配上昇法を実行し、観察される挙動を小問 b および c と関連づけよ。

7. [code] **省略された割引因子。**●●多くの実装は、因子 γ^{t} を Equation 14.68 の軌跡形式から省略しており、これにより次の更新ベクトル場が得られる。

$$F(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t \ge 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\Big]. \tag{14.70}$$

   Nota and Thomas (2020) は、F がいかなる関数の勾配にもならない場合があることを証明し、さらに F に沿った厳密な更新が方策を最悪の方策へと向かわせるマルコフ決定過程（MDP）を構成した。以下の各小問では、まず 1 つ目の事実を FrozenLake 上で確認し、続いて 2 つ目の出発点として F に沿ったステップが J を引き下げる小さな MDP を構成する。

   1. 演習問題 3a にならい、次式を示せ。

$$F_{s,a'}(\theta) = \rho_1(s)\,\pi_\theta(a' \mid s)\big(Q^{\pi_\theta}(s, a') - V^{\pi_\theta}(s)\big), \tag{14.71}$$

ここで $\rho_1(s) = \sum_{t \geq 0} \Pr(s_t = s,\, t < T)$ は、エピソードが終了するまでの s への期待訪問回数である。方策の遷移行列のうち終端状態に属する行を 0 に設定した連立一次方程式を解くことで、ρ_1 を計算せよ。

2. 連続微分可能な勾配場のヤコビ行列はヘッセ行列であるため対称になる。自動微分によってゼロテーブルおよび probe のパラメータにおけるヤコビ行列 DF を計算し、フロベニウスノルムによる $\|DF - DF^{\top}\| / \|DF\|$ を求めよ。対照実験として、$\nabla_\theta J$ に対しても同じ確認を実行せよ。

3. 同じ 2 点において、F と $\nabla_\theta J$ のなす角のコサインを測定せよ。それにもかかわらず実際の実装が F を受け入れている実用上の理由を 1 つ挙げよ。

4. 表形式のソフトマックス方策では $F(\theta)^{\top} \nabla_\theta J(\theta) \geq 0$ がすべての θ で成り立ち、したがって FrozenLake 上では F に沿った微小ステップが決して J を下げないことを示せ。次に、順次に訪問される系列上の 2 つの状態の方策を単一のパラメータが制御するような小さなエピソード型 MDP と、ある θ で $F(\theta)\, \mathrm{d}J/\mathrm{d}\theta < 0$ となる組を構成せよ。

14.8 ベースラインと分散低減

前節では、サンプリングされた各行動の対数確率を、その軌跡（trajectory）の収益（return）に比例して引き上げることで方策（policy）を訓練した。

$$\hat{u} = \frac{1}{n} \sum_{i=1}^{n} R(\tau_i) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.72}$$

平均的には、この推定値は期待収益 $J(\theta)$ の勾配に一致する。しかし、1 回の抽出ではそうならない。1 回の更新あたり 4 エピソードを用いると、訓練途中の方策における単一の推定値は勾配の長さの 3 倍以上ずれるのが常であり、成功を 1 つも含まないバッチでは更新がまったく行われなかった。その原因の大部分は重みにある。穏やかな湖ではすべての収益が 0 か γ^{T-1} のいずれかであるため、各行動は無視されるか報酬を与えられるかのどちらかになる。成功したエピソードでの遠回りは確率が引き上げられ、失敗したエピソードで穴に踏み込む一歩は決して引き下げられない。

Williams は REINFORCE を導入した 1992 年の論文でこの問題を見抜き、そこで解決策を提案した。重み付けの前に、基準となる水準すなわち**ベースライン**を収益から差し引くというものである (Williams 1992)。統計学ではこれを制御変量（control variate）と呼ぶ。平均が既知の量を差し引けば、推定値の平均を保ったまま分散（variance）を下げることができる。本節では、方策自身の行動のスコアは平均が 0 になるという 1 つの事実を出発点として、どの量を差し引いてよいのか、そして厳密な勾配を基準に測ったときそれぞれがどれだけ分散を取り除くのかを明らかにする。次章のアクター・クリティック法や、言語モデルの事後学習（post-training）に使われるグループベースの更新は、この推定量に別のベースラインを組み合わせたものにほかならない。

```
import gymnasium as gym
```

穏やかな湖、$\gamma = 0.95$、および 14.7 節のソフトマックステーブルを引き続き用いる。穏やかな湖ではすべての行動が目に見えて収益に影響するが、滑りやすい氷の上では成功の大半が運に左右され、収益から個々の行動についてわかることはほとんどない（14.7 節で確認したように、一様な初期方策では同じ精度を得るために約 9 倍大きなバッチが必要になる）。各更新には分散の違いが現れるよう 4 エピソードだけを使い、各訓練の比較では 20 個のシード（最大のステップサイ

ズでは 100 個）を実行する。エピソードは 200 ステップで打ち切る。それほど長く歩き続けるのはループに陥った方策だけであり、その価値（value）は 0 に近いため、この打ち切りによって重みが目立って変わることはない。打ち切りを行わなければ、以下で用いる大きなステップサイズでは数千ステップに及ぶエピソードが生じてしまう。

```
gamma, eta_v = 0.95, 0.1              # discount; value-table step size
num_updates, batch_episodes = 150, 4  # small batches, to expose variance
num_seeds = 20                        # twenty seeds per arm
num_seeds_large = 100                 # at the large step size, eta = 16
env = gym.make('FrozenLake-v1', is_slippery=False, max_episode_steps=200)
```

14.8.1 ゼロ平均恒等式

本節で扱う手法はいずれもスコアの重みを変更するが、平均は変えずに保たなければならない。14.7 節の方策勾配定理（policy gradient theorem）の証明ですでに用いた 1 つの恒等式がこれを保証する。ここでは、微分可能なあらゆる方策に対してこの恒等式を述べる。

14.8.1.1 ゼロ平均補題

ソフトマックステーブルでは、行動 a の状態 s におけるスコアは、状態 s の選好（preference）に対応する 1 つのブロックだけが非ゼロとなる。

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta_{s,a'}} = \mathbf{1}(a' = a) - \pi_\theta(a' \mid s). \tag{14.73}$$

s における方策自身の行動について平均をとると、指示関数の平均は $\pi_\theta(a' \mid s)$ となり、第 2 項と打ち消し合う。この相殺は、微分可能なすべての方策で成り立つ。

補題：すべての状態 s に対して、$\sum_{a \in \mathcal{A}} \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s) = 0$ が成り立つ。

証明：$\sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) = \sum_a \nabla_\theta \pi_\theta(a \mid s) = \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0.$ ■

言葉で言えば、どの状態においても、サンプリングされた行動のスコアの平均は 0 である。ある行動が得る確率質量は、他の行動が失うからである。

14.8.1.2 条件付き恒等式

実用上役立つ帰結はもう少し強い。c を、エージェントが状態 s_t に立った時点ですでに定まっている任意の量（軌跡の前半で得られた報酬、定数、あるいは s_t 自身の関数など）とする。軌跡のプレフィックス $(s_0, a_0, r_0, \dots, s_t)$ で条件付けると、値 c は確定している一方で行動 a_t はまだ確率的であり、さらに方策は s_t のみを参照するため、内側の期待値は先ほどの補題が扱う形そのものになる。

$$\begin{aligned}\mathbb{E}\big[c\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\big] &= \mathbb{E}\Big[c\, \underbrace{\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \mid s_0, a_0, \dots, s_t\big]}_{=\,0 \text{ by the lemma}}\Big] \\ &= 0.\end{aligned}$$

したがって、REINFORCE 推定量に含まれる任意のスコアにそのような量を掛けても、あるいはその重みからそのような量を差し引いても、平均は変わらない。条件はプレフィックスに関するものである。すなわち、a_t やそれ以降に起きる事柄に依存する量は対象外である。

14.8.2 残りの累積報酬、ベースライン、制御変量

この恒等式により、ステップ t の重みに対する 2 つの操作が正当化される。a_t より前に確定している項を落とすことと、a_t より前に確定している量を差し引くことである。どちらも平均を動かさない。そのいずれかが分散を減らすかどうかは、また別の問題である。

14.8.2.1 残りの累積報酬と因果関係

REINFORCE 推定量では、ステップ t のスコアに軌跡全体の収益 $R(\tau_i)$ が掛けられており、その中には行動 a_t^i が抽出されるより前に得られた報酬も含まれている。それらの報酬はプレフィックスによって確定しているため、条件付き恒等式により、ステップ t のスコアとの積の平均は 0 になる。それらを取り除くと、次式が残る。

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim P(\cdot;\theta)}\Big[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T-1} \gamma^{t'} r_{t'}\Big]. \tag{14.74}$$

これが**因果関係**（causality）である。行動は未来に影響を与えうるが、過去には影響しない。内側の和は $\gamma^t \widehat{G}_t$ に等しく、右の因子は次式で与えられる。

$$\widehat{G}_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} \tag{14.75}$$

これはステップ t からの**残りの累積報酬**（reward-to-go）、すなわちエピソードの残り部分の割引収益である。素の推定量は T 個のスコアの和に T ステップにわたる和である収益を掛けるため、毎ステップ報酬が得られる場合にはエピソード長とともに分散が急速に増大する。これに対し、過去の報酬を取り除けば、各スコアに掛ける和が短くなる。もっとも、平均 0 の項を取り除くこと自体が必ずしも分散を下げるとは限らない。取り除いた項が残りの部分と相関（correlation）している場合があるからであり、以下の測定では両方の結果が現れる。

本書の実装も含め、実際の実装ではさらに因子 γ^t も落とし、ステップ t のスコアを $\widehat{G}_t$ のみで重み付けする。

$$\hat{u} = \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1} \widehat{G}_t^i \, \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.76}$$

γ^t を落とすことは、先ほどの恒等式からは正当化されない。状態への遅い時点での訪問を早い時点での訪問と同じ重みで数えることになるため、$\gamma < 1$ のとき Equation 14.76 の平均は $\nabla_\theta J$ から傾く。Thomas (2014) はアクター・クリティックアルゴリズムにおけるこの統計的バイアス（estimator bias）を指摘し、Nota and Thomas (2020) はこの更新がいかなる目的関数の勾配にもなっていないことを示した。以下で調べる方策では、この傾きは約 8 度（コサインで 0.991）であり、平均の長さは約 1.4 倍になる（演習問題 8）。本節では、この推定量を基礎とする諸手法と共通のこの慣例を踏襲しつつ、両方の版を測定する。

すべてのステップの残りの累積報酬は、最終ステップの後ろを 0 として始めるエピソードごとの 1 回の逆方向走査 $\widehat{G}_t = r_t + \gamma \widehat{G}_{t+1}$ から求まる。`backward_scan` は、任意のベクトルと係数に対してこの漸化式を 14.7 節の `Batch.episodes` が返す各エピソードにわたり実行する（後の節では別の量を別の係数で走査する）。そして `reward_to_go` は、これを係数 γ で報酬に適用する。

```
@d2l.add_to_class(d2l.Batch)
def backward_scan(self, x, factor):
    """y_t = x_t + factor * y_{t+1}, restarted at every episode boundary."""
    y = np.zeros_like(x)
    for ep in self.episodes():
        running = 0.0
        for t in reversed(range(ep.start, ep.stop)):
            running = x[t] + factor * running
            y[t] = running
    return y

@d2l.add_to_class(d2l.Batch)
def reward_to_go(self, gamma):
    """G_t: the discounted return of the rest of its episode, by one scan."""
    return self.backward_scan(self.rew, gamma)
```

14.8.2.2 ベースライン

2 つ目の操作は減算である。任意の量 b は、s_t までの軌跡のプレフィックスによって定まる限り（定数であっても現在の状態の関数 $b(s_t)$ であっても）、残りの累積報酬から差し引くことができる。

$$\widehat{u} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \left(\widehat{G}_t^i - b(s_t^i)\right) \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.77}$$

このような b を**ベースライン**と呼ぶ。追加される項は $b(s_t)$ とスコアの積であり、条件付き恒等式によってその平均は 0 になるため、Equation 14.77 と Equation 14.76 の平均は一致する。

この湖ではどの残りの累積報酬も負にならないため、ベースラインがなければ、サンプリングされた行動が引き下げられることは決してない。典型的な残りの累積報酬に近い値を差し引けば、通常より良い結果となった続きの展開には正の重みが、悪かった展開には負の重みが与えられる。では、平均が変わらないのに、なぜ学習が変化しうるのか。それは次項で計算する分散を通じてである。また、重みの最良の予測値である $Q(s_t, a_t)$ の推定値を差し引かないのはなぜか。それは a_t に依存するため恒等式が適用できなくなり、平均がずれてしまうからである（演習問題 1）。

14.8.2.3 制御変量

ベースラインの減算は、モンテカルロ推定における一般的な手法の 1 つの例である。$\mathbb{E}[X]$ を求めたいとし、各サンプルには期待値が既知で相関を持つ量 Y も含まれているとする。任意のスカラー c に対して、次式を定義する。

$$X_c = X - c\left(Y - \mathbb{E}[Y]\right). \tag{14.78}$$

$\mathbb{E}[Y - \mathbb{E}[Y]] = 0$ であるため、推定量 X_c の期待値は X の期待値とすべての c について一致する。その分散は次式で与えられる。

$$\mathrm{Var}(X_c) = \mathrm{Var}(X) - 2c\,\mathrm{Cov}(X, Y) + c^2\,\mathrm{Var}(Y), \tag{14.79}$$

これは c の二次関数であり、次式で最小となる。

$$c^* = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(Y)}. \tag{14.80}$$

c^* を元の式に代入すると、分散は $(1 - \mathrm{corr}^2)\,\mathrm{Var}(X)$ となる。ここで corr は X と Y の相関である。相関が強ければ大幅な分散低減（variance reduction）が可能になる。$\mathrm{corr} = 0.9$ では分散は約 5 分の 1 に減るが、$\mathrm{corr} = 0$ では変わらない。この量 Y を X に対する**制御変量**と呼ぶ。

方策勾配（policy gradient）では、$X = \hat{G}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ および $Y = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ とおく。補題により $\mathbb{E}[Y \mid s_t] = 0$ であり、$c = b(s_t)$ とすれば Equation 14.77 が復元される。すなわち、ベースラインとはスコアから構成された制御変量である。最良の係数は状態ごとに個別に計算した c^* であり、ベクトル値のスコアに対しては積を内積（inner product）に置き換えることで、演習問題 2 の最適ベースライン b^* が得られる。Greensmith et al. (2004) は、方策勾配に対してこの解析を展開している。

最小の事例を見ると、その効果の大きさがわかる。Figure 14.19 の 1 ステップ問題を取り上げよう。1 つの行動 a を正規分布 $\mathcal{N}(a; \mu, 1)$(ただし $\mu = 0$)から抽出し、報酬を $R(a) = 0.4 + 2e^{-(a-2)^2/2}$ とする。期待報酬（expected reward）は $J = \mathbb{E}[R] = 0.92$、μ に関するスコアは $a - \mu = a$、厳密な勾配は $\mathrm{d}J/\mathrm{d}\mu = 0.52$ である。1 サンプルによる推定値 $R(a)\,a$ はこの平均を持つが、標準偏差は 1.34 と推定対象の量の 2.5 倍を超える。平均報酬を差し引いた $(R(a) - 0.92)\,a$ では、標準偏差が 0.69 に下がる。最良の定数は $b^* = \mathbb{E}[R(a)\,a^2]/\mathbb{E}[a^2] = 1.18$ であり、平均報酬よりも大きくなる。これは、0 から離れた行動ほど大きなスコアを持ち、さらにこの問題では大きな報酬も持つためである。このとき、ベースラインなしの分散のうち $1 - \mathrm{corr}^2 = 0.23$ だけが残る（Figure 14.20）。

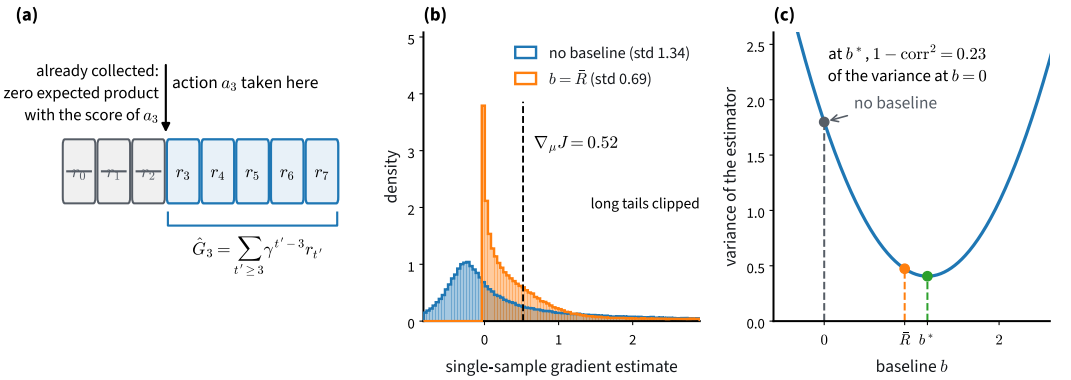


Figure 14.20: スコア関数推定量のノイズを抑える 2 つの方法。(a) 複数ステップのエピソードでは、ある行動より前に得られた報酬とその行動のスコアとの積の期待値は 0 になるため、残りの累積報酬 $\hat{G}_3$ だけが残る。(b) Figure 14.19 の 1 ステップ問題では、平均報酬を差し引くことで 1 サンプル推定値の標準偏差が 1.34 から 0.69 に下がる。両ヒストグラムとも 1 パーセンタイルと 95 パーセンタイルでクリップしている。(c) 定数ベースライン b の関数としての分散は放物線を描き、$b^* = 1.18$ で最小となり、元の分散の 0.23（$b = 0$ における値に対する比）だけが残る。

14.8.2.4 アドバンテージと学習されたベースライン

この湖では、1 つの定数をすべての状態に適合させることはできない。ゴールの隣のマスは、スタート地点よりも期待される残りの累積報酬がはるかに高いからである。状態に依存する自然なベースラインは、その状態の価値である。14.2 節より、$V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s,a)$ は行動の価値に関する方策の平均である。ある行動の**アドバンテージ**は、その行動がこの平均よりどれだけ優れているかを測る量であり、次式で定義される。

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s), \qquad \mathbb{E}_{a\sim\pi(\cdot\mid s)}\big[A^\pi(s,a)\big] = 0. \tag{14.81}$$

いま述べた関係から、その平均は 0 になる。正のアドバンテージは、状態 s における方策の平均を上回る行動を表す。$\mathbb{E}[\hat{G}_t \mid s_t, a_t] = Q^{\pi_\theta}(s_t,a_t)$ であるため、重み $\hat{G}_t - V^{\pi_\theta}(s_t)$ は選択された行動のアドバンテージのサンプルとなり、この更新はその状態の平均を上回る行動の確率を引き上げ、残りを引き下げる。

V^{π_θ} は未知だが、同じバッチから推定できる。テーブル $\hat{V}(s)$ を保持し、各バッチの後に訪問したすべての状態の項目を、そこで観測された残りの累積報酬に向けて次のように動かす。

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \eta_V(\hat{G}_t - \hat{V}(s_t)), \tag{14.82}$$

ここで η_V はステップサイズである。これがベースライン付き REINFORCE である (Williams 1992)。このテーブルは、完了したエピソードの残りの累積報酬、すなわちモンテカルロ収益（Monte Carlo return）に対する回帰によって当てはめられる。1 ステップ先の $\hat{V}$ 自身の予測値からターゲットを構成すると、第 15 章のアクター・クリティック法が得られる。そのアドバンテージ推定（advantage estimate）は、ブートストラップを一切行わない場合に $\hat{G}_t - \hat{V}(s_t)$ へと帰着する。

14.8.3 中心化、スケーリング、正規化

実務では、各バッチの重みを標準化したり、グループ内のサンプル同士を比較したりすることが多く、またどの実装でも損失の和を何で割るかを決めなければならない。それぞれの選択は、推定量そのものを変えるか、そのスケールだけを変えるかのいずれかであるが、この 2 つは混同しやすい。両者を見分けるには、厳密な勾配がわかっている固定された方策が必要になる。

まず、`probe` を 14.7 節とまったく同じように、`d2l.train_reinforce_tabular` と同じ 16 エピソードずつの 16 回の更新で再構築する。その価値 $J(\theta) = V^{\pi_\theta}(s_0)$ は、14.2 節の連立一次方程式（linear system）$V = r_\pi + \gamma P_\pi V$ の解の第 1 成分である。ここで $P_\pi(s' \mid s) = \sum_a \pi_\theta(a \mid s)\, P(s' \mid s,a)$ および $r_\pi(s) = \sum_a \pi_\theta(a \mid s)\, r(s,a)$ である。`probe.probs` はプローブから π_θ を読み出し、`mdp.policy_value(pi_np)` はこの方程式系を NumPy で解いて、すべての状態における V^{π_θ} すなわち厳密な状態ベースラインを返す。

```
mdp = d2l.TabularMDP.from_gym(env, gamma)
probe = d2l.Policy.tabular(16, 4)        # the probe of sec_policygradient
list(d2l.train_reinforce_tabular(probe, env, gamma, 3, num_updates=16))
pi_np = probe.probs(16, 4)
V_pi = mdp.policy_value(pi_np)           # the exact state baseline
```

`exact_gradient` は、同じ方程式系をフレームワークの配列で記述した 14.7 節の `d2l.policy_objective` を、プローブの選好テーブルにおいて微分する。報酬テーブル `r` を引数にとるため、後ほど変更を加えた報酬に対してもそのまま使える。

```
theta_probe, P = probe.policy.weight.detach(), torch.as_tensor(mdp.P).float()

def exact_gradient(r):
    """grad J at the probe for the rewards r[s, a], flattened."""
    return torch.func.grad(d2l.policy_objective)(
        theta_probe, P, torch.as_tensor(r).float(), gamma).numpy().ravel()

g_exact = exact_gradient(mdp.r)
print(f'J(theta) = {V_pi[0]:.3f}, |grad J| = {np.linalg.norm(g_exact):.3f}')
```

```
J(theta) = 0.313, |grad J| = 0.129
```

出力された $J(\theta) = 0.313$（達成可能な最大値は $\gamma^5 \approx 0.77$）は、14.7 節におけるプローブの価値である。その厳密な勾配が、以下のすべての測定の基準となる。

14.8.3.1 バッチ中心化と分散スケーリング

よく使われる変種として、各バッチ内で残りの累積報酬の値を標準化する手法がある。現在のバッチ内のすべての $\hat{G}_t^i$ を集め、その平均 $\bar{R}$ と標準偏差 σ を計算して、次の量を用いる。

$$\tilde{G}_t^i = \frac{\hat{G}_t^i - \bar{R}}{\sigma + 10^{-8}} \tag{14.83}$$

これを $\hat{G}_t^i$ の代わりに使う。ここで定数 10^{-8} はゼロ除算を防ぐためのものである。$\bar{R}$ を差し引く操作はベースラインであり、サンプル間の相対的な重みを変え、推定値の方向も変えうる。ただし、$\bar{R}$ は同じバッチから計算されるため、サンプリングされた行動に弱く依存する。その結果生じる統計的バイアスについては、軌跡ごとの重みの場合を次項で、ここで用いたステップ単位の重みの場合を演習問題 3 で計算する。

$\sigma + 10^{-8}$ で割る操作は、バッチ全体の推定値に 1 つの正の数を掛けるだけなので、1 つのバッチ内ではベースラインとしてではなく、報酬のばらつきに応じて変化する適応的ステップサイズ（adaptive step size）として働く。一方、バッチをまたいで見るとこの係数は確率的でありバッチと相関するため、期待更新量も傾く。演習問題 4 では、プロンプトごとに異なる σ がもたらす関連する効果を導出する。この湖では $0 \le \hat{G}_t \le 1$ から $\sigma \le 1/2$ が従うため、固定されたステップサイズのもとでは、正規化は中心化の少なくとも 2 倍の長さのステップをとる（以下の測定では約 3 倍となる）。`standardize` は Equation 14.83 を実装したものである。

```
def standardize(x, eps=1e-8):
    """Center a NumPy batch of weights and divide by its standard deviation
    (NumPy's std, which divides by the batch size)."""
    return (x - x.mean()) / (x.std() + eps)
```

14.8.3.2 1 つ抜きベースライン

バッチ平均には、差し引く対象であるサンプル自身が含まれている。1 つ抜き（leave-one-out）の中心化は、各軌跡に対して**他の**軌跡の平均をベースラインとして与えることで、この自己依存性を取り除く。

$$b_i = \frac{1}{n-1}\sum_{j \neq i} R(\tau_j).$$

各軌跡は独立に抽出されるため、b_i は軌跡 i と独立であり、条件付きゼロ平均恒等式が補正なしでそのまま成り立つ。すなわち、この推定量はどのバッチサイズでも不偏である。その重みは、$\bar{R}$ による中心化の重みを定数倍したものになる。

$$R_i - b_i = \frac{n}{n-1}(R_i - \bar{R}), \tag{14.84}$$

したがって、軌跡ごとの重みに対する単純な中心化は、推定量の平均を $(n-1)/n$ 倍（$n = 2$ では半分）にする。全列挙できるほど小さな問題（順に訪問する 2 つの状態、各状態で 2 つの行動、テーブルから与えられる報酬）で、これら両方の主張を確かめよう。`leave_one_out` は 1 バッチ分の重み $R_i - b_i$ を計算する。

```
def leave_one_out(R):
    """R_i - b_i, with b_i the mean of the other n - 1 returns: n/(n-1)
    times the centered returns."""
    return (R - R.mean()) * len(R) / (len(R) - 1)
```

次のセルでは、起こりうる 4 つの軌跡から厳密な勾配 `g` を計算し、さらにサイズ 2 の順序付きバッチ全 16 通りにわたる両推定量の厳密な平均を求める。

```
rng = np.random.default_rng(0)
th = rng.standard_normal((2, 2))              # a generic two-state table
pi2 = np.exp(th) / np.exp(th).sum(1, keepdims=True)
r2 = np.array([[0.3, 1.0], [0.6, 0.1]])       # r[s, a]; s0 -> s1 -> done
trajs = [(a0, a1) for a0 in range(2) for a1 in range(2)]
p = np.array([pi2[0, a0] * pi2[1, a1] for a0, a1 in trajs])
R = np.array([r2[0, a0] + r2[1, a1] for a0, a1 in trajs])
S = np.zeros((4, 2, 2))                       # the score of each trajectory
for i, (a0, a1) in enumerate(trajs):
    S[i] = -pi2
    S[i, 0, a0] += 1
    S[i, 1, a1] += 1
g = (p[:, None, None] * R[:, None, None] * S).sum(0)    # exact gradient
u_loo = u_cen = 0.0
for i in range(4):
    for j in range(4):                        # every batch of n = 2
        w, wc = leave_one_out(R[[i, j]]), R[[i, j]] - R[[i, j]].mean()
        u_loo += p[i] * p[j] * (w[0] * S[i] + w[1] * S[j]) / 2
        u_cen += p[i] * p[j] * (wc[0] * S[i] + wc[1] * S[j]) / 2
print(f'E[leave-one-out] equals the exact gradient: {np.allclose(u_loo, g)}')
print(f'E[centered] equals (n-1)/n of it:           {np.allclose(u_cen, g / 2)}')
```

```
E[leave-one-out] equals the exact gradient: True
E[centered] equals (n-1)/n of it:           True
```

どちらの恒等式も機械精度で成り立つ。Kool et al. (2019) は入力あたり複数のサンプルを用いる REINFORCE に対して 1 つ抜きベースラインを提案しており、これは RLOO という名称で言語モ

デルの事後学習に使われている (Ahmadian et al. 2024)。すなわち、1 つのプロンプトに対して n 個の応答をサンプリングし、それぞれの応答を、自身の報酬から残りの $n-1$ 個の平均報酬を引いた値で重み付けする。

14.8.3.3 バッチとしてのグループ

この湖ではすべてのエピソードが同じマスから始まるため、1 つのバッチ平均を全エピソードで共有できる。14.1 節の検証器タスク（verifier task）はそうではない。4 つの算術プロンプト、共通の 9 個の応答候補、そして正解に対して報酬 1 を与える検証器からなり、参照方策の成功確率は 0.062、また 2 つのプロンプトには正解候補が存在しないため、どの方策も 0.5 を超えることはできない。各プロンプトが固有の成功率を持つ独立した初期状態であるため、ベースラインはプロンプトごとに用意しなければならない。14.7 節の `verifier_step` は、各プロンプトに対して K 個の応答をサンプリングし、スコア $\mathbf{e}_y - \pi(\cdot \mid x)$（各応答 y に対するもの）を K 個の報酬の関数で重み付けする。1 つのプロンプトに対する K 個の応答、すなわち**グループ**がベースラインを与える。

報酬が 0 と 1 の場合、グループ平均による中心化は単純な形をとる。正解が k 個（全応答数 K のうち）であれば、グループ平均は k/K となるため、正解の応答は重み $1-k/K$ を、不正解の応答は重み $-k/K$ を受け取る。ベースラインなしでは不正解の応答の重みは 0 であったが、ここでは引き下げられるようになり、しかもそのプロンプトで方策がすでに高い頻度で成功しているほど強く引き下げられる。グループの標準偏差 $\sqrt{\hat{p}(1-\hat{p})}$（ただし $\hat{p} = k/K$）で割ると、2 つの重みは次のようになる。

$$+\sqrt{\frac{1-\hat{p}}{\hat{p}}} \quad \text{and} \quad -\sqrt{\frac{\hat{p}}{1-\hat{p}}}. \tag{14.85}$$

$K = 16$ で正解が 2 個のとき、各正解の応答は約 $+2.65$、各不正解の応答は約 -0.38 を受け取る。つまり、難しいプロンプトにおけるまれな成功は強く押し上げられる。全員不正解または全員正解のように、すべての応答が同じスコアをとるグループでは比較対象がなく、すべての応答の重みが 0 になる（これがどれほどの頻度で起きるかは 14.4 節で数えた）。

グループ内における応答 j の標準化された重みは次式で与えられる。

$$\hat{A}_j = \frac{r_j - \bar{R}}{\sigma + 10^{-8}}, \tag{14.86}$$

ここで、$\bar{R}$ と σ はグループ内の K 個の報酬の平均と標準偏差である。これはプロンプトを初期状態、グループをバッチとみなした Equation 14.83 の軌跡単位の形であり、ステップごとの残りの累積報酬ではなく、応答ごとの 1 つの報酬を標準化する。これが Group Relative Policy Optimization（GRPO）の重みであり (Shao et al. 2024)、`group_step` は `verifier_step` の重み付けに `standardize` を用いたものである。

```
def group_step(task, theta, K, reward, rng, lr=2.0):
    """verifier_step with the group-normalized weight (GRPO): the rewards
    of each prompt's K responses, standardized within the group."""
    return d2l.verifier_step(task, theta, K, reward, rng, standardize, lr)
```

グループ平均にはその応答自身の報酬が含まれるため、1 つ抜きの計算により、グループ中心化された更新の期待値は勾配の $(K-1)/K$ 倍になる。σ による除算を除いたサイズ 2 のグルー

プについて、これを厳密に確かめよう。次のセルでは、すべての応答のスコア（Sv）、9 項の和 $\sum_y \pi(y \mid x)\, r(x,y)\, (\mathbf{e}_y - \pi(\cdot \mid x))$ として求まる各プロンプトの成功率の厳密な勾配（gv）、および応答のすべての順序対にわたる中心化更新の厳密な平均（u）を計算する。

```
task = d2l.VerifierTask()                  # the instance of sec_mdp
K, pi, Rv = 2, task.policy(task.theta_ref), task.R
Sv = np.eye(len(task.resp))[None, :, :] - pi[:, None, :]  # eq_softmax_score
gv = np.einsum('xy,xy,xyv->xv', pi, Rv, Sv)          # the exact gradient
u = np.zeros_like(gv)
for y1, y2 in np.ndindex(len(task.resp), len(task.resp)):  # groups of K = 2
    w = (Rv[:, y1] - Rv[:, y2]) / 2                  # reward minus group mean
    u += (pi[:, y1] * pi[:, y2] * w)[:, None] * (Sv[:, y1] - Sv[:, y2]) / K
print(f'E of the group-centered update is (K-1)/K of the gradient: '
      f'{np.allclose(u, gv * (K - 1) / K)}')
print(f'rescaled by K/(K-1), leave-one-out, it is exact: '
      f'{np.allclose(u * K / (K - 1), gv)}')
```

```
E of the group-centered update is (K-1)/K of the gradient: True
rescaled by K/(K-1), leave-one-out, it is exact: True
```

この統計的バイアスはどれほど影響するだろうか。$K = 1$ では係数が 0 になる。グループ平均が応答自身の報酬と一致し、すべての重みが 0 になって、勾配が 0 でないにもかかわらず方策は一切動かない。$K \geq 2$ では、この係数はステップを縮小させるだけである。以下のスイープ実験では、予算をプロンプトあたり採点済み応答 6,400 個に固定するため、グループが小さいほど多くのステップを踏む（3,200 ステップが $K = 2$ の場合、200 ステップが $K = 32$ の場合である）。

```
budget = 6400                    # scored responses per prompt, matched
for K in (1, 2, 4, 8, 32):
    rng, theta = np.random.default_rng(1), task.theta_ref.copy()
    for _ in range(budget // K):
        theta = group_step(task, theta, K, task.verify, rng)
    print(f'K = {K:2d}: success of a sampled response '
          f'{task.success(task.policy(theta)):.3f}')
```

```
K =  1: success of a sampled response 0.062
K =  2: success of a sampled response 0.500
K =  4: success of a sampled response 0.500
K =  8: success of a sampled response 0.500
K = 32: success of a sampled response 0.500
```

予測どおり、$K = 1$ では方策は参照スコア 0.062 のままであり、$K \geq 2$ では試した 10 個のシードすべてにおいて、同じ予算内で上限の 0.5 に到達する。本来なら学習された価値テーブルが担うはずのプロンプトごとの比較を、グループが提供しているのである。なお、プロンプトごとに σ で割ることには、この問題例には現れないさらなる効果がある。更新がどのプロンプトを優先するかが変わるのである（演習問題 4）。

14.8.3.4 損失の除数

Equation 14.76 の二重和はエピソードとステップにわたっており、実装ではこれを何らかの数で割る必要がある。エピソード数 n で割ればその式そのものになる。総ステップ数で割ると、各バッチがその平均エピソード長の逆数で再スケーリングされるが、この長さは方策の改善に伴って変化する。各エピソードをそれぞれの長さで割ると、エピソード間の重み付けが変わり、方向も

変化しうる。固定された定数で割る場合はスケールだけが変わる。長さの異なる 4 エピソードからなる 1 つのバッチで、これら 4 通りすべてを計算してみよう。`agg` は、バッチ `b` の方策 `pi` のもとでのスコアを 14.7 節の `d2l.tabular_score` を用いて合計する際、各スコアに残りの累積報酬とステップごとの係数 `scale` の積を重みとして掛ける。

```
b = d2l.rollout(env, probe.act, 4, np.random.default_rng(6))
G, T = b.reward_to_go(gamma), np.diff(b.ep_ends, prepend=0)
print(f'episode lengths {T}, successes {int(b.rew.sum())}')

def agg(b, G, pi, scale):  # sum_t scale_t G_t score_t
    return d2l.tabular_score(b.obs, b.act, G * scale, pi).ravel()

grads = {'episodes': agg(b, G, pi_np, np.full(len(b), 1 / len(T))),
         'own length': agg(b, G, pi_np, np.repeat(1 / T, T) / len(T)),
         'total steps': agg(b, G, pi_np, np.full(len(b), 1 / len(b))),
         'a constant': agg(b, G, pi_np, np.full(len(b), 1 / 32))}
for k, u in grads.items():
    cos = u @ grads['episodes'] / (np.linalg.norm(u)
                                   * np.linalg.norm(grads['episodes']))
    print(f'{k:>12}: |grad| = {np.linalg.norm(u):.3f}, '
          f'cos to episodes = {cos:.3f}')
```

```
episode lengths [ 8 20  8 11], successes 3
    episodes: |grad| = 0.868, cos to episodes = 1.000
  own length: |grad| = 0.082, cos to episodes = 0.937
 total steps: |grad| = 0.074, cos to episodes = 1.000
  a constant: |grad| = 0.109, cos to episodes = 1.000
```

4 つの勾配のうち 3 つは互いに平行で、大きさには 1 桁の開きがある。エピソードごとの長さで割る版だけは、他から方向が傾いている。14.5 節の `policy_step` は 3 番目の選択肢である総ステップ数で割るため、以下の訓練ループでは 1 番目の選択肢を復元するよう重みを再スケーリングする。言語モデルの訓練でも、応答をエピソード、トークンをステップとみなせば同じ選択に直面する。GRPO は各応答をそれぞれの長さで割り、DAPO は総トークン数で割り (Q. Yu et al. 2025)、Dr. GRPO はすべてのバッチを同じ係数で再スケーリングする唯一の選択肢である定数で割る (Z. Liu et al. 2025)。

14.8.4 5 つの推定量の比較

これで、ステップ t におけるスコアに対する 5 通りの重み付けが出そろった。このうちどれが分散を減らし、その効果はどれほどだろうか。まず厳密な勾配がわかっている固定方策で答えを確かめ、続いて訓練の中で調べる。

14.8.4.1 固定方策における推定量

5 通りの重み付けは次のとおりである。

1. **軌跡の収益** $R(\tau)$：本節の冒頭に示した不偏な REINFORCE 推定量である。
2. **残りの累積報酬** $\hat{G}_t$：Equation 14.76 のとおり、因子 γ^t を落としたものである。
3. **中心化**した残りの累積報酬 $\hat{G}_t - \bar{R}$：$\bar{R}$ はバッチ内の全ステップにわたる平均である。
4. **正規化**した残りの累積報酬：Equation 14.83 である。
5. **価値ベースライン** $\hat{G}_t - V(s_t)$：固定方策では、ここでは計算可能な連立一次方程式の解から得られる厳密な V^{π_θ} を用い、訓練では Equation 14.82 の学習されたテーブル $\hat{V}$ を用い

る。

6 行目の**因果的**は、ステップ t のスコアを因果関係の議論から得られた不偏推定量（unbiased estimator）である $\gamma^t \hat{G}_t$ で重み付けするものであり、過去の報酬を落とす操作と γ^t を落とす操作を切り分ける。

プローブを固定したまま、4 エピソードからなるバッチを 2,000 個抽出し、それぞれについてすべての推定量を計算する。`estimate` は `d2l.tabular_score` を用いてバッチの重み付きスコアを合計し、Equation 14.76 と同様にエピソード数で割る。`discounted` は各残りの累積報酬に γ^t（t はそのエピソードの先頭から数える）を掛ける。`weightings` は各名称を、バッチ・残りの累積報酬・価値テーブルの関数に対応付ける（`per_step` は各エピソードの収益をそのステップ全体に繰り返す）。`stats` は出力される 3 つの列を計算する。`|mean|/|grad J|` と `cos` は推定量の平均のスケールと方向、すなわち抽出された推定値の平均の $\nabla_\theta J$ に対する相対的な長さと、両者のなす角のコサインを表す。`noise` 列は相対分散であり、抽出値の共分散のトレースを、その推定量自身の平均の長さの二乗で割った値である。各推定量自身の平均を基準に測るのは、正規化された重みが中心化された重みを σ で割ったものであり、再スケーリングされた推定量がスケールの違いによって不利に評価されるべきではないからである。

```
def estimate(b, w, pi):
    """One draw of the estimator: weighted scores, averaged over episodes."""
    return d2l.tabular_score(b.obs, b.act, w, pi).ravel() / len(b.ep_ends)

def discounted(b, G, gamma):
    """gamma^t G_t: reward-to-go with the factor gamma^t of the gradient kept."""
    t = np.concatenate([np.arange(ep.stop - ep.start) for ep in b.episodes()])
    return gamma ** t * G

weightings = {                  # V: exact V^pi here, the learned table later
    'return': lambda b, G, V: b.per_step(b.episode_returns(gamma)),
    'causal': lambda b, G, V: discounted(b, G, gamma),
    'reward-to-go': lambda b, G, V: G,
    'centered': lambda b, G, V: G - G.mean(),
    'normalized': lambda b, G, V: standardize(G),
    'value baseline': lambda b, G, V: G - V[b.obs]}

def stats(u, g):
    """The scale and direction of the mean of the draws u against g, and
    the variance of the draws relative to their own mean."""
    m = u.mean(axis=0)
    size_m, size_g = np.linalg.norm(m), np.linalg.norm(g)
    return (size_m / size_g, m @ g / (size_m * size_g),
            ((u - m) ** 2).sum(axis=1).mean() / (m @ m))
```

この湖では報酬がゴールでの 1 回しかないため、落とすべき過去の報酬が存在しない。因果関係が働く様子を見るため、次のセルでは非終端マスからの移動ごとに 0.1 のコストを課して測定を繰り返す。記録されたすべてのステップはそのような移動であるため、バッチの各報酬から 0.1 を引けば新しい報酬のもとでの同じエピソードが得られる。`costly` は同じカーネルとコスト付き報酬を持つ MDP であり、厳密な勾配と V^π を与える。

```
step_cost = 0.1                 # charged for every move from a live cell
live = ~np.isin(env.unwrapped.desc.ravel(), [b'H', b'G'])
costly = d2l.TabularMDP(mdp.P, mdp.r - step_cost * live[:, None], gamma)
```

```
    cases = {'lake': (0.0, g_exact, V_pi),
             'step cost': (step_cost, exact_gradient(costly.r),
                           costly.policy_value(pi_np))}
    rng = np.random.default_rng(4)
    batches = [d2l.rollout(env, probe.act, 4, rng) for _ in range(2000)]
    for case, (c, g, V) in cases.items():
        bs = [d2l.Batch(b.obs, b.act, b.rew - c, b.next_obs, b.term, b.ep_ends)
              for b in batches]
        for k, fn in weightings.items():
            u = np.stack([estimate(b, fn(b, b.reward_to_go(gamma), V), pi_np)
                          for b in bs])
            scale, cos, noise = stats(u, g)
            print(f'{case:>9} {k:>14}: |mean|/|grad J| = {scale:.2f}, '
                  f'cos = {cos:.3f}, noise = {noise:5.1f}')
```

```
     lake         return: |mean|/|grad J| = 0.99, cos = 0.998, noise =  13.6
     lake         causal: |mean|/|grad J| = 0.99, cos = 0.998, noise =  13.6
     lake   reward-to-go: |mean|/|grad J| = 1.38, cos = 0.987, noise =  12.4
     lake       centered: |mean|/|grad J| = 1.11, cos = 0.986, noise =   7.6
     lake     normalized: |mean|/|grad J| = 3.28, cos = 0.982, noise =   7.5
     lake value baseline: |mean|/|grad J| = 1.38, cos = 0.990, noise =   5.7
step cost         return: |mean|/|grad J| = 0.98, cos = 0.996, noise =  19.5
step cost         causal: |mean|/|grad J| = 1.00, cos = 0.999, noise =   5.8
step cost   reward-to-go: |mean|/|grad J| = 1.29, cos = 0.989, noise =   5.4
step cost       centered: |mean|/|grad J| = 1.07, cos = 0.985, noise =   5.6
step cost     normalized: |mean|/|grad J| = 2.12, cos = 0.980, noise =   5.8
step cost value baseline: |mean|/|grad J| = 1.28, cos = 0.989, noise =   3.0
```

通常の湖の設定では、軌跡の収益の行と因果的の行は抽出ごとに完全に一致する。成功エピソードの末尾に報酬が 1 つあるだけなので、どのステップでも $\gamma^t \hat{G}_t = \gamma^{T-1}$ が収益そのものになるからである。残りの累積報酬がこれらと異なるのは落とされた γ^t の分だけであり、平均は約 1.4 倍長くなり、コサインはわずかに低下する。不偏推定量を $2{,}000$ 回の抽出にわたって平均するとコサインは 0.997 付近になるため、この列に現れているのは実質的にこの傾きだけである。ノイズの列には、別のシードで抽出を繰り返しても保たれる順序が現れている。価値ベースラインが最もノイズが小さく、中心化と正規化が互いに一致してこれに続き、収益と残りの累積報酬が最もノイズが大きい。中心化は残りの累積報酬のノイズを約 40% 取り除き、価値ベースラインは約半分を取り除く。また中心化は平均の長さを残りの累積報酬の約 0.8 倍に縮めるが、これは軌跡ごとの中心化における係数 $(n-1)/n = 3/4$ に近い。σ で割ると平均の長さは約 3 倍に伸びる一方、ステップサイズとして当然であるように、相対ノイズは中心化と同じ値にとどまる。

移動ごとのコストがある設定では、過去の報酬を落とすことで収益のノイズがここでは約 3 分の 1 に減っている。しかし、常にそうなるとは限らない。落とされた項は平均こそ 0 だが残りの項と相関しており、$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y)$ は $\mathrm{Var}(X)$ より小さくなりうる。実際、移動ごとのコストを 0.01 にすると、過去の報酬を落とすことでノイズは約 6 分の 1 増加する（演習問題 8）。他の行の順序もこの方策に固有のものであり、一様方策では中心化は役に立たない（演習問題 5）。

14.8.4.2 訓練におけるステップサイズと分散

訓練では多数のステップが積み重なり、ステップサイズが関わってくる。以下の各アームは同じシードを用い、`policy_step` に渡す重みだけが異なる。Adam は座標ごとに再スケーリングして調査対象のスケールの違いを部分的に打ち消してしまうため、ここではステップサイズ η の素の SGD で

更新する。価値ベースラインのアームしか使わないものの、すべての実行において Equation 14.82 により $\hat{V}$ を更新する。`entropy` は、nat 単位のエントロピー $-\sum_a \pi(a \mid s) \log \pi(a \mid s)$ をバッチ内の状態にわたって平均するため、各状態はエージェントが訪問した回数だけ重み付けされる。

```
def entropy(theta, obs):
    """Entropy of pi(.|s) in nats, averaged over the states the batch visited."""
    logp = theta - np.logaddexp.reduce(theta, axis=1, keepdims=True)
    return float(-(np.exp(logp) * logp).sum(axis=1)[obs].mean())
```

`train` は、`weightings` のいずれか 1 つとステップサイズ `eta` を用いた素の SGD による REINFORCE である。残りの引数は環境および本節の冒頭で固定した設定である。`policy_step` はバッチの N ステップにわたって平均をとる一方、固定方策の表では n エピソードにわたって平均をとっていた。そこで、すべての重みに N/n を掛けて前者の平均を後者に変換し、各アームがその表で測定した推定量をそのまま実行するようにする。この係数がなければ実効的なステップは $\eta\, n/N$ となり、エピソードが短くなるにつれて大きくなってしまう。MDP を `exact` として与えると、`train` は学習されたテーブルを、これから更新される方策の厳密な V^{π_θ}（固定方策のときと同様に毎回の更新前に新たに解く）で置き換える。このアームを**厳密ベースライン**（exact baseline）と呼ぶ。各更新では、バッチの成功率、パラメータステップの長さ、およびバッチに沿ったエントロピーを記録する。

`table` は方策から選好テーブルをコピーし、`d2l.plain_sgd`（オンライン付録「ユーティリティ関数とクラス」）は方策の Adam オプティマイザを素の SGD に置き換え、`d2l.run_seeds`（オンライン付録「ユーティリティ関数とクラス」）はシードごとに `train` を 1 回呼び出して結果を積み重ねる。

```
def table(ac):
    """The policy's preference table, copied out to numpy."""
    return ac.policy.weight.detach().numpy().copy()
```

これらの補助関数を用いて、`train` は素の SGD による REINFORCE ループを `num_updates` バッチにわたって実行する。各ステップの重みは `weighting(batch, G, V)` の出力に `len(batch) / batch_episodes` を掛けてスケーリングし、価値テーブル `V` を Equation 14.82 によって更新し（`exact` が与えられた場合は V^{π} を厳密に解く）、平均収益、パラメータステップのノルム、方策エントロピーを記録する。

```
def train(seed, weighting, eta, env, gamma, num_updates, batch_episodes,
          eta_v, exact=None):
    """REINFORCE by plain SGD; the arms differ in one line, the weight. With
    an MDP as `exact`, V is the exact V^pi of the current policy."""
    S, A = env.observation_space.n, env.action_space.n
    rng, V = np.random.default_rng(seed), np.zeros(S, np.float32)
    ac = d2l.Policy.tabular(S, A, seed=seed)
    ac.opt_pi = d2l.plain_sgd(ac.policy, eta)
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = d2l.rollout(env, ac.act, batch_episodes, rng)
        G, before = batch.reward_to_go(gamma), table(ac)
        if exact is not None:               # V^pi of the policy that acted
            V = exact.policy_value(ac.probs(S, A))
        w = weighting(batch, G, V) * len(batch) / batch_episodes
        d2l.policy_step(ac, batch.obs, batch.act, w)  # sum over steps / n
```

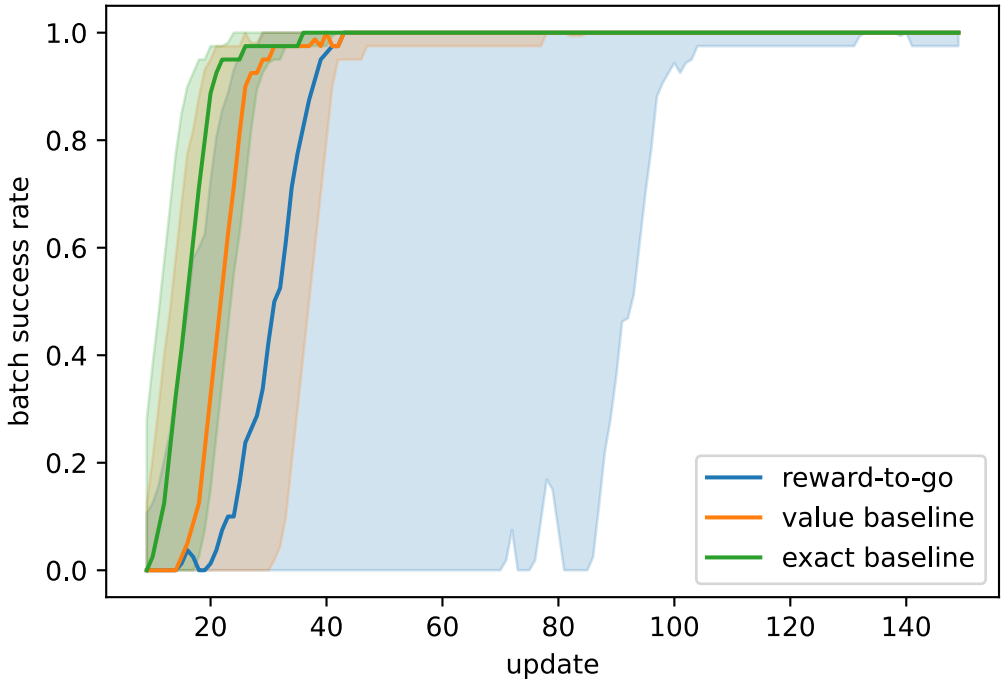
```
        for s, g in zip(batch.obs, G):     # eq_value_baseline, every arm
            V[s] += eta_v * (g - V[s])
        yield (float(batch.episode_returns().mean()),
               float(np.linalg.norm(table(ac) - before)),
               entropy(before, batch.obs))
```

各重み付けはスケールが異なるため、共通の η を用いるとステップの長さが異なり、ステップの長さだけでも学習の速さは変わる。平均ステップ長をそろえても、この問題は解消されない。ステップの長さにはノイズが含まれるうえ、ベースラインを持つアームのステップは方策が成功し始めると縮小するのに対し、ベースラインを持たないアームはすでに選んでいる行動を引き上げ続けるため、平均でそろえたアーム同士でも学習中のステップ長は依然として異なるからである。純粋な比較を行うには、代わりに期待更新量を固定する。厳密な V^{π_θ} はバッチを抽出する前に確定しており、過去のバッチのみに当てはめられた学習済みの $\hat{V}$ も同様である。したがって条件付き恒等式により、残りの累積報酬、価値ベースライン、厳密ベースラインはあらゆる θ において同じ期待更新量を持ち、同一の η のもとではノイズのみが異なる。そこで、6 つのアームすべてを $\eta = 2$ で実行し（`shared`）、さらにこれら 3 つのアームを $\eta = 16$ で `num_seeds_large` 個のシードにわたって再実行する（`large`）。`setup` は全実行で共通の設定をまとめた辞書であり、各アームの実行結果は形状 (seeds, updates, 3) の配列に積み重ねられる。

```
variants = ['return', 'reward-to-go', 'centered', 'normalized',
            'value baseline']
arms = {v: dict(weighting=weightings[v]) for v in variants}
arms['exact baseline'] = dict(weighting=weightings['value baseline'],
                              exact=mdp)
setup = dict(env=env, gamma=gamma, num_updates=num_updates,
             batch_episodes=batch_episodes, eta_v=eta_v)
shared = {v: d2l.run_seeds(train, num_seeds, eta=2.0, **kw, **setup)
          for v, kw in arms.items()}
large = {v: d2l.run_seeds(train, num_seeds_large, eta=16.0, **arms[v],
                          **setup)
         for v in ('reward-to-go', 'value baseline', 'exact baseline')}
```

共通の平均を持つ 3 つのアームの $\eta = 16$ における成功率を、10 回の更新ウィンドウで平滑化して次に示す。各帯はシードの中央値の周りの四分位範囲を表している。

```
d2l.plot_curves({v: r[:, :, 0] for v, r in large.items()}, xlabel='update',
                ylabel='batch success rate', smooth=10, band='iqr')
```



`to90` は、直近 10 回の更新にわたる成功率が 90% に達する最初の更新を求める（許容誤差は 10 回中 9 回の丸め誤差を吸収するためのものである）。次のセルでは、両方のステップサイズについて、シードにわたる中央値と最速・最遅のシード、150 回の更新以内に一度も 90% に達しない実行の数、および 90% に達する前の更新にわたる平均ステップ長を出力する。

```
    def to90(curve):
        """Last update of the first 10-update window whose mean success is 0.9."""
        hit = np.convolve(curve, np.ones(10) / 10, 'valid') >= 0.9 - 1e-9
        return hit.argmax() + 9 if hit.any() else len(curve)

    for name, runs in (('eta = 2', shared), ('eta = 16', large)):
        print(name)
        for v, r in runs.items():
            reach = np.array([to90(c) for c in r[:, :, 0]])
            climb = np.mean([c[:t, 1].mean() for c, t in zip(r, reach)])
            print(f'{v:>14}: to 90%: median {np.median(reach):5.1f} '
                  f'[{reach.min():3d}, {reach.max():3d}], never '
                  f'{(reach == num_updates).sum():2d}, |step| {climb:.2f}')
```

```
eta = 2
        return: to 90%: median  63.0 [ 42, 108], never  0, |step| 0.43
  reward-to-go: to 90%: median  51.0 [ 29, 109], never  0, |step| 0.53
      centered: to 90%: median  58.5 [ 39, 119], never  0, |step| 0.32
    normalized: to 90%: median  32.0 [ 15,  85], never  0, |step| 0.93
value baseline: to 90%: median  52.0 [ 38,  97], never  0, |step| 0.37
exact baseline: to 90%: median  50.5 [ 33,  96], never  0, |step| 0.40
eta = 16
  reward-to-go: to 90%: median  35.0 [ 11, 150], never 15, |step| 1.46
```

```
value baseline: to 90%: median  26.0 [ 11, 150], never  5, |step| 1.30
exact baseline: to 90%: median  20.5 [ 10,  41], never  0, |step| 1.68
```

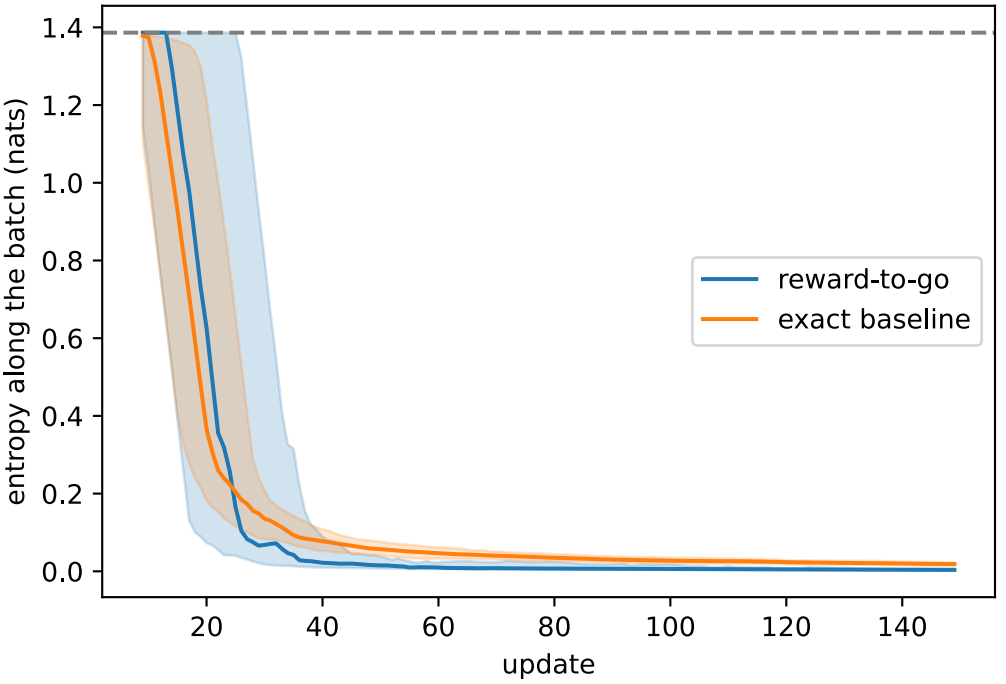
共通のステップサイズでは、順序はステップサイズ効果によるものとなる。正規化は最も少ない更新回数（50〜70 回に対して約 30 回）で済むが、90% に到達する前のステップは中心化の約 3 倍、残りの累積報酬の 1.5 倍以上の長さがある。中心化はステップを縮小させるため、素の収益とほぼ同じ速さにとどまり、自身が中心化する重みである残りの累積報酬よりも遅い。共通の平均を持つ 3 つのアームはこのステップサイズでは区別がつかず、いずれも約 50 回の更新を要する。

$\eta = 16$ では 3 つに差がつく。3 つとも $\eta = 2$ のときより速く学習するが、最も恩恵を受けるのは厳密ベースラインであり、その中央値は 50 回から約 20 回の更新へと下がる（残りの累積報酬は 51 回から約 35 回）。さらに大きな違いは失敗例に現れる。厳密ベースラインの実行では失敗が 1 つもないのに対し、残りの累積報酬では約 7 回に 1 回の実行が一度も 90% に到達しない。そうした実行のほぼすべてが停滞している。ノイズの乗った方向に長いステップを踏んだ結果、制限時間までループするか穴に落ちる経路上で方策がほぼ決定論的になり、以降のすべてのエピソードの収益が 0、すべての重みが 0 となって、方策が二度と動かなくなるのである。学習されたテーブルは両者の中間に位置する（中央値は 26 回の更新で、20 回に 1 回の実行が 90% に到達しない）。このステップサイズでは、$\hat{V}$ が追随できる以上に速く方策が変化するためである。ステップ長の列を見ると、この改善がステップの伸長によるものではないことがわかる。90% に到達する前、厳密ベースラインの 1 更新あたりの移動量は、残りの累積報酬より最大でも約 6 分の 1 大きいだけである。小さなステップを数多く踏めばノイズは平均化されて消えるため、分散が低くても小さなステップが良くなるわけではない。分散の低減は、より長いステップを使えるようにするのである（演習問題 6）。

$\eta = 2$ ではどのアームでも最遅のシードは最速のシードの 2〜6 倍の更新回数を要し、$\eta = 16$ では厳密ベースライン以外の実行のばらつきがさらに広がる。そのため、20 シードでは近接した中央値の順序が入れ替わりうる。アームあたり 200 シードを用いた別の実験で、そのうち 20 シードを無作為に抽出すると、$\eta = 2$ における残りの累積報酬と厳密ベースラインの中央値は約 7 回に 1 回の割合で 6 回超の差がつく。一方、$\eta = 16$ では 50 回の抽出のうち約 49 回で厳密ベースラインの方が低い中央値となるものの、残りの累積報酬のシードが 1 つ挙動を変えるだけで 20 シードの中央値が 15 回以上動きうる。そこで、$\eta = 16$ の各アームでは 100 シードを実行している。本節では中央値と範囲を報告し、主張に依拠する前により多くのシードで検証する (Henderson et al. 2018; R. Agarwal et al. 2021)。

エージェントが訪問する状態に沿って、方策はどれほど決定論的になるだろうか。

```
d2l.plot_curves({v: large[v][:, :, 2] for v in ('reward-to-go',
                                                'exact baseline')},
                xlabel='update', ylabel='entropy along the batch (nats)',
                reference=np.log(4), smooth=10, band='iqr')
```



$\eta = 16$ では、両アームの中央値は最初の 20〜40 回の更新のうちに、一様方策の $\ln 4 \approx 1.39$ nat からバッチに沿って 0.1 nat 未満へと低下し、通常は残りの累積報酬の方が先に低下する。停滞した実行も成功した実行と同じくらい決定論的であり、繰り返す経路上では 0 nat 付近になる。なお、全 16 状態にわたって平均をとると 0.4 nat を上回ったままになる。4 つの穴とゴールでは行動が一切とられず、$\ln 4$ のまま残るためである。

以上で、どの重みが使用可能か、それぞれがどれだけ分散を取り除くか、そしてより優れた推定量とより長いステップとをどう見分けるかが明らかになった。本節の推定量はすべて、16 行の選好と 16 個の $\widehat{V}$ の要素からなるテーブルを参照していた。しかし、レール上の台車は 4 つの実数で記述され、同じ状態を二度と訪問することはない。14.9 節では、これら両方のテーブルをネットワークに置き換える。

14.8.5 まとめ

方策自身の行動のスコアは平均が 0 であり、その行動より前に確定している任意の量との積も同様に平均が 0 となる。これにより、期待勾配を変えることなく過去の報酬を落とし、ベースラインを差し引くことができる。ベースラインとは制御変量であり、重みをサンプリングされたアドバンテージへと変える状態価値 V^{π} は、測定した方策においてノイズの約半分を取り除いた。各軌跡の収益をバッチ平均やグループ平均で中心化すると期待勾配に $(n-1)/n$ が掛かるが、これは 1 つ抜きによって解消される。ステップごとに中心化する場合は、別の係数で縮小する。ばらつきで割る操作はステップサイズを変え、二値報酬のもとでは GRPO のグループ正規化重みが失敗例を引き下げる（サイズ 1 のグループでは学習できない）。共通のステップサイズを用いると、最も長いステップをとる重み付けが有利になる。同一のステップサイズかつ共通の平均で比較すると、厳密な価値ベースラインと残りの累積報酬は小さなステップでは同じ速さで学習するが、大きなステップではベースラインの方が速く学習し、停滞もしない。

14.8.6 演習問題

1. **許容されるベースライン。**条件付きゼロ平均恒等式により、エージェントが s_t に立った時点ですでに定まっている任意の量を $\hat{G}_t$ から差し引くことができる。以下の各候補について、Equation 14.77 においてそれを差し引いたときに期待勾配が不変に保たれるかどうかを判定し、その条件を用いて理由を示せ。

   1. 時間インデックス t
   2. 和 $r_0 + \cdots + r_{t-1}$（ステップ t より前に得られた報酬の合計）
   3. エピソードに残っているステップ数 $T - t$
   4. 次の状態 s_{t+1} の関数
   5. 行動価値の推定値 $\hat{Q}(s_t, a_t)$

   最後の候補で $\hat{Q} = Q^{\pi_\theta}$ としたとき、減算後のステップ t の期待寄与を計算せよ。この結果は、行動に依存するベースラインについて何を示しているか。

2. **最適定数ベースライン。**状態 s を固定し、サンプリングされた行動のスコアを $\psi = \nabla_\theta \log \pi_\theta(a \mid s)$ と書き、1 ステップ推定量 $(\hat{G} - b)\,\psi$（b は定数）を考える。その平均は b に依存しないため、ノイズを共分散のトレースで測る。

$$\mathbb{E}\big[\|(\hat{G} - b)\,\psi\|^2\big] - \big\|\mathbb{E}[\hat{G}\,\psi]\big\|^2. \tag{14.87}$$

   1. このトレースを b について最小化し、最小点が次式で与えられることを示せ。

$$b^* = \frac{\mathbb{E}\big[\|\psi\|^2\,\hat{G}\big]}{\mathbb{E}\big[\|\psi\|^2\big]}. \tag{14.88}$$

      他の任意の b はトレースをちょうど $\mathbb{E}\big[\|\psi\|^2\big]\,(b - b^*)^2$ だけ増やすこと、および積を内積に置き換えたときに b^* が Equation 14.78 の係数 c^*（ただし $X = \hat{G}\,\psi$、$Y = \psi$）に一致することを示せ。

   2. Figure 14.20 の 1 ステップ問題では、行動は $a \sim \mathcal{N}(\mu, 1)$（$\mu = 0$）であるため、$\mu$ に関するスコアは a となる。b^* を $R(a) = 0.4 + 2e^{-(a-2)^2/2}$ に対して計算し、値 $b^* = 1.18$ を確認せよ。なお、標準正規分布の密度関数と $e^{-(a-2)^2/2}$ の積は別の正規分布の密度関数に比例する。なぜ b^* は $\mathbb{E}[R] = 0.92$ を上回るのか。

3. [code] **バッチ中心化の統計的バイアス。**●14.8.3.2 節では、軌跡ごとのバッチ中心化が期待勾配を縮小させる係数 $(n-1)/n$ を、1 つ抜きの恒等式 $R_i - b_i = \frac{n}{n-1}(R_i - \bar{R})$ から導いた。この議論は軌跡ごとの重みにしか適用できない。

   1. 1 つ抜きベースラインを使わずに、この係数を直接導出せよ。中心化された推定値を $\frac{1}{n}\sum_{i=1}^{n}\big(R(\tau_i) - \bar{R}\big)\,\mathbf{S}_i$ と書き（ここで $\mathbf{S}_i$ は軌跡 i のスコアの和、$\bar{R}$ はバッチの平均収益である）、$\mathbb{E}[\bar{R}\,\mathbf{S}_i]$ を n 個の項に展開して、軌跡間の独立性を用いよ。非ゼロの期待値を持つのはどの項か。
   2. 一方、`train` の中心化アームは、バッチ内のすべての残りの累積報酬の値の平均をステップごとに差し引く。この重み付けへと全列挙を拡張せよ。この全列挙では割引を用いないため、2 状態問題の各軌跡は $\hat{G}_0 = r_0 + r_1$ および $\hat{G}_1 = r_1$ を持ち、`S[i]` の第 s 行は状態 s でとられたステップのスコアとなる。$n \in \{2, 4, 8\}$ に対して推定値の期待値を計算し、ノルムの比とコサインを通じて `g` と比較せよ。

3. この比較結果を説明せよ。2 つのステップのそれぞれについて、そのステップのスコアとの積の期待値が非ゼロになるのは、バッチ平均に含まれるどの残りの累積報酬の値か。

4. σ **正規化目的関数。**●GRPO は各グループの中心化された報酬をその標準偏差 σ で割るが、この選択については Z. Liu et al. (2025) が解析している。プロンプト x が二値報酬（正解の応答で $r = 1$、それ以外で $r = 0$）を持ち、現在の方策がこのプロンプトで成功する確率を p_x とする。グループが十分に大きく、$\bar{R}$ と σ がそれぞれの母集団の値 p_x と $\sqrt{p_x(1-p_x)}$ に等しいとする。

   1. σ を $p_x = 0.5$ および $p_x = 0.9$ のそれぞれについて計算せよ。この除算によって、2 つ目のプロンプトの更新は 1 つ目のプロンプトに対して何倍に増幅されるか。

   2. y をサンプリングされた応答とする。次式を示せ。

$$\mathbb{E}\big[(r - p_x)\,\nabla_\theta \log \pi_\theta(y \mid x)\big] = \nabla_\theta\, p_x. \tag{14.89}$$

   これより、σ 正規化更新が $\sum_x h(p_x)$（ただし関数 h は $h'(p) = 1/\sqrt{p(1-p)}$ を満たす）を上昇させることを結論づけ、h を求めよ。

   3. h を正規化しない更新の目的関数 $\sum_x p_x$ と比較せよ。それと比べて重みが増すのはどのプロンプトか。訓練セットの大半のプロンプトがほぼ常に解けるかほぼ決して解けないかのどちらかであるとき、これは実務者が望む重み付けといえるか。

   4. K 個の応答からなる有限のグループでは、K 個の報酬がすべて一致したときにプロンプトは勾配に寄与しない。$p_x = 0.9$ かつ $K = 8$ のときにこの事象が起きる確率を計算し、これが小さな σ を持つプロンプトの増幅をどのように制限するか説明せよ。

5. [code] **方策にわたる分散の順序。**●14.8.4.1 節の固定方策の比較では、16 回の更新後の 1 つの方策 `probe`（$J(\theta) = 0.313$）を調べた。プローブの更新回数 $k \in \{0, 8, 32, 64\}$ の後に到達する各方策でも同じ比較を繰り返せ。すなわち、`probe` を訓練するループの反復回数を変更し、`theta_probe`、`pi_np`、`g_exact`、`V_pi` を再計算してから 2,000 個のバッチを抽出せよ。

   1. 5 つの推定量のノイズを k に対して表にまとめよ。16 回の更新で見られた順序はすべての方策で保たれるか。中心化が役に立たなくなるのはどの方策であり、それはなぜか。
   2. $J(\theta)$ が最大値 $\gamma^5 \approx 0.77$ に近づくにつれて、ベースラインを持たない 2 つの推定量はどのように振る舞うか。また、それはなぜか。
   3. $k = 0$ において、成功を 1 つも含まないバッチの個数を数え、そのようなバッチが 5 つの推定量のそれぞれにどのように寄与するか説明せよ。

6. [code] **ステップサイズの交絡。**●共通のステップサイズでは、`train` の正規化アームが最も少ない更新回数で成功率 90% に到達したが、同時に最も長いステップもとっていた。$\sigma > 0$ であるバッチでは、正規化された重み Equation 14.83 は、中心化された重みをステップサイズ $\eta/(\sigma + 10^{-8})$ で SGD 更新したものと一致する。

   1. 正規化アームにおいてバッチごとに σ を記録し、成功を含まないバッチを除いたうえで、$1/(\sigma + 10^{-8})$ のシードにわたる中央値を更新インデックスに対してプロットせよ。方策が改善するにつれて、このステップサイズの乗数はどのように変化するか。正規化アームを η を実行全体の $1/(\sigma + 10^{-8})$ の平均で割ったステップサイズで動かして両アームをそろえようとする試みに対し、このことは何を意味するか。
   2. 残りの累積報酬、価値ベースライン、厳密ベースラインの比較を $\eta \in \{4, 8, 32\}$ で繰り返し、各アームについて η に対する 90% 到達までの更新回数の中央値と、一度も到達しない実行の割合をプロットせよ。各アームにとって最良のステップサイズはどれ

か。推定量のノイズが下がるにつれて最良のステップサイズが大きくなる理由を説明せよ。

3. 代わりにステップ長をそろえると、ノイズの大きいアームにはより小さい η が与えられる。`shared` の 5 つのアームのそれぞれについて、最初の 40 回の更新にわたる平均ステップ長が約 0.6 になるステップサイズを選び、各実行が 90% に到達する前と後とで平均ステップ長を別々に記録せよ。学習中により長いステップをとるのはどのアームか。また、平均ステップ長をそろえた比較でも依然として分散とステップサイズが混ざり合ってしまうのはなぜか。

7. [code] **最適状態ベースライン**。●●演習問題 2 では、1 つの状態における分散最小の定数ベースラインを導出した。これを状態ごとに適用すると、1 ステップ分の項の分散を最小化する Greensmith et al. (2004) の最適ベースライン関数が得られる。本問では、表形式のソフトマックス方策についてこれを具体的に求め、エピソード単位の推定値 `estimate` に対しても最適となる条件を調べる。

   1. ソフトマックスのスコアから次式を示せ。

$$\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2 = 1 - 2\pi_\theta(a \mid s) + \sum_{a'} \pi_\theta(a' \mid s)^2. \tag{14.90}$$

   さらに $\mathbb{E}[\widehat{G}_t \mid s_t = s, a_t = a] = Q^{\pi_\theta}(s,a)$ を用いて、最適ベースライン $b^*(s)$ を $\pi_\theta(\cdot \mid s)$ と $Q^{\pi_\theta}(s,\cdot)$ で表せ。

   2. 1 エピソードの推定値は、その全ステップの重み付きスコアの和である。表形式のパラメータ化では異なる状態における 2 つのステップのスコアが直交し、その結果、推定値の二乗ノルムがステップごとの項と同一状態への複数回の訪問間の交差項とに分解されることを示せ。エピソードに関するどのような条件のもとで、$b^*(s)$ は `estimate` の共分散のトレースを厳密に最小化するか。FrozenLake はその条件を満たすか。

   3. `probe` において $b^*(s)$ を計算し（Q^{π_θ} は `mdp.backup(V_pi)` から得る）、重み付け `G - b_star[b.obs]` を 14.8.4.1 節の `weightings` に `lambda b, G, V: G - b_star[b.obs]` として追加せよ。`probe` およびプローブの 64 回の更新後の方策において、そのノイズを厳密ベースラインのノイズと比較せよ。

   4. $b^*(s)$ が $V^{\pi_\theta}(s)$ と最も大きく異なるのはどの状態か。重み $\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2$ を用いて、ほぼ決定論的な状態がなぜ $b^*(s)$ をまれにしか選ばれない行動の行動価値の側へ引き寄せるのかを説明し、2 つのベースラインの差を演習問題 2a で求めた過剰分散と関連づけよ。

8. [code] **残りの累積報酬の分解**。●Equation 14.76 の残りの累積報酬の重みは、軌跡の収益に対して 2 つの変更を加えている。各行動より前に得られた報酬を落とすこと（これは条件付き恒等式で許容される）と、因子 γ^t を落とすこと（これは許容されない）である。

   1. $\rho(s) = \mathbb{E}\big[\sum_t \mathbf{1}(s_t = s)\big]$ を 1 エピソードにおける s への期待訪問回数とする。演習問題 7a と同様に、Equation 14.76 の平均は $\sum_s \rho(s) \sum_a \nabla_\theta \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s,a)$ である一方、方策勾配は各状態を $\sum_t \gamma^t \Pr(s_t = s)$ で重み付けする。両者が一致するのはどの γ のときか。
   2. `probe` において、穴でもゴールでもない 11 個のマス上で ρ を $\rho = \mathbf{e}_{s_0} + P_\pi^\top \rho$ を解くことで計算し、`mdp.backup(V_pi)` から Q^{π_θ} を得て、Equation 14.76 の平均を厳密に評価せよ。その `g_exact` とのコサインおよび `g_exact` に対する相対的な長さを、固定方策の表の残りの累積報酬の行と比較せよ。
   3. 過去の報酬を落とすことは、平均 0 の項 Y を推定値 $X + Y$ から取り除くことに相当する。$\mathrm{Var}(X) < \mathrm{Var}(X+Y)$ となるのは $\mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y) > 0$ のとき、か

つそのときに限ることを示せ。移動ごとのコストを 0.01（0.1 の代わり）とした場合、およびプローブの 64 回の更新後に到達する方策において、収益の行と因果的の行の比較を繰り返せ。因果関係は常にノイズを減らすか。それぞれのケースを、過去の報酬と残りの累積報酬との共分散の符号によって説明せよ。

14.9 強化学習における関数近似

既知の運動方程式から計算した 14.6 節のプランナを除き、ここまで学習してきた量はすべてテーブルに格納されていた。湖のマスに対応する 16 行や、検証器タスクのプロンプトごとの 1 行である。ここで、上部にヒンジで取り付けられたポールのバランスをとる台車を思い描いてみよう。その状態は台車の位置と速度、およびポールの角度と角速度という 4 つの実数であり、1,000 エピソードを実行してもエージェントがまったく同じ 4 つの数値を 2 度目にすることはない。このような状態上のテーブルでは訪問のたびに新しい行が必要になり、1 回の訪問で学習したことは次の訪問について何も語らない。ポールがある速度で左に傾いているときにエージェントが学習したことを、わずかに異なる速度で左に傾いた瞬間にも引き継げるよう、似た状態に似た値を与える関数が求められる。

この問題は、この分野そのものと同じくらい古い。Bellman は状態変数の数に伴うテーブルの肥大化を**次元の呪い**（curse of dimensionality）と名付け (Bellman 1957b)、1959 年にはすでに Arthur Samuel のチェッカープログラムが盤面をテーブルから参照する代わりに特徴量の重み付き和でスコア付けしていた (Samuel 1959)。テーブルをパラメータ化された関数で置き換えることを**関数近似**（function approximation）と呼ぶ。この関数としてニューラルネットワークを用いることで、強化学習（reinforcement learning）はバックギャモン (Tesauro 1995)、Atari ゲーム (Mnih et al. 2015)、そして囲碁へと広がった (Silver et al. 2016)。

方策勾配（policy gradient）にとってこの一歩は小さい。14.7 節および 14.8 節の導出では、どこでもテーブルという形式を使っていなかったからである。REINFORCE が必要とするのはスコア $\nabla_\theta \log \pi_\theta(a \mid s)$ のみであり、ネットワークもテーブルの行とまったく同様にこれを供給できる。変わるのは、1 回の更新があらゆる状態における推定値を動かすようになる点である。この共有によって連続状態での学習が可能になる一方で、学習が破綻する新たな要因も生まれる。本節では、ポール付きの台車と連続トルクで駆動される振り子という 2 つの小規模な制御タスクでこの一歩を踏み出し、1 回の更新がどこまで広がるかを測定したうえで、こうして得られるエージェントになお何ができないのかを確認して締めくくる。

```
import gymnasium as gym
```

14.9.1 テーブルに代わるネットワーク

前節までのテーブルは、方策（policy）と、ベースラインとして用いた価値推定の 2 か所でネットワークに置き換わる。いずれのネットワークを書く前にも、まずはネットワークがその自然な拡張となる形でテーブルを捉え直しておこう。

14.9.1.1 ワンホット状態上の線形ネットワークとしてのテーブル

`Policy.tabular`（14.5 節）は選好を 16×4 の埋め込みテーブル θ に格納しており、状態 s における選好はそのテーブルの第 s 行である。第 s 行は $\theta^\top \mathbf{e}_s$ に等しい。ここで $\mathbf{e}_s$ は状態 s のワンホットベクトルである。したがって、テーブルとは入力特徴量が固定されたワンホットベクトルである線形ネットワークにほかならない。連続状態に対する古典的な対処法も同じ形式をとる。状態空間を複数の箱に分割し、$\mathbf{e}_s$ を s が属する箱のワンホットベクトルとするのである。後述する台車とポールの課題に対する初期の学習型制御器では、人手で設計した分割による 162 個の箱が使

われていた (Barto et al. 1983)。1 つの隠れ層を持つネットワークはこの形式を保ったまま、固定された特徴量を学習される特徴に置き換える。すなわち、選好は $\mathbf{W}_2 \tanh(\mathbf{W}_1\mathbf{s} + \mathbf{b}_1) + \mathbf{b}_2$ となり、ここで $\mathbf{s}$ は実数値の観測ベクトルそのものである。

これら 2 つの形式は、1 回の更新が何を引き起こすかという点で異なる。価値推定 $\hat{V}_w$（パラメータ w）に対し、サイズ η_V の勾配ステップを $\frac{1}{2}\big(y - \hat{V}_w(s_0)\big)^2$ 上で 1 回とって、単一の状態 s_0 における推定値をターゲット y に近づけるとする。η_V の一次までで、任意の状態 s における推定値の変化量は次式で与えられる。

$$\Delta\hat{V}(s) = \eta_V\big(y - \hat{V}(s_0)\big)\, k(s, s_0), \qquad k(s, s_0) = \nabla_w \hat{V}(s)^\top \nabla_w \hat{V}(s_0). \tag{14.91}$$

したがって、s における変化量は、s_0 で求められた変化量に 2 つの状態における勾配の重なり具合を掛けたものになる。テーブルの場合、$\nabla_w \hat{V}(s) = \mathbf{e}_s$ であるため、$k(s, s_0)$ は 1（$s = s_0$ のとき）または 0（それ以外のとき）となる。すなわち、1 つの状態での勾配ステップは他のすべての要素を元のまま保つ。（Policy のデフォルトオプティマイザである Adam は、蓄積されたモメンタムを通じて現在のバッチで勾配が 0 である要素も動かし得るため、この記述はあらゆるオプティマイザのステップではなく勾配について成り立つ。）ネットワークの場合、k は 2 つの状態の滑らかな関数であり、そのネットワークの**接カーネル**（tangent kernel）と呼ばれる (Jacot et al. 2018)。このとき、s_0 でのステップは似た状態における推定値も一緒に動かす。演習問題 7 では Equation 14.91 を導出し、14.9.4.1 節では訓練するネットワークについて k の形状を測定する。

14.9.1.2 方策ネットワークと価値ネットワーク

Policy コンテナ（14.5 節）は、方策ネットワークとそのオプティマイザを保持する。14.8 節では、学習されたベースラインは別個の NumPy テーブルであり、各バッチの後に $\hat{V}(s_t)$ をステップサイズ η_V で、訪問したすべての状態で観測された残りの累積報酬（reward-to-go）$\hat{G}_t$ に向かって動かしていた。ここではベースラインもネットワークになり、コンテナの内部に移る。ActorCritic は Policy を継承し、価値モジュール value とそれ専用の Adam オプティマイザ opt_v を追加する。V は損失が微分する形式であるフレームワーク配列として価値を返し、value_np は Policy の境界メソッドと同様に NumPy 配列を受け取って返す。この名前は 2 つの構成要素に由来しており、方策はしばしば**アクター**、価値推定は**クリティック**と呼ばれる。tabular コンストラクタは両者をゼロ初期化されたテーブルとして構築するため、本節末尾の測定で行うように、どちらのネットワークの代わりにもテーブルを差し込める。

```
class ActorCritic(d2l.Policy):
    """A policy and a value function, each with its own optimizer."""
    def __init__(self, policy, value, lr=1e-2):
        super().__init__(policy, lr)
        self.value = value
        self.opt_v = torch.optim.Adam(value.parameters(), lr=lr)

    def V(self, obs):
        """V(s) for a batch of states, as a tensor."""
        return self.value(obs).squeeze(-1)

    def value_np(self, obs):
        """V(s) for a numpy batch of states, as numpy."""
        with torch.inference_mode():
            return self.V(torch.as_tensor(obs)).numpy()
```

```
    @classmethod
    def tabular(cls, num_states, num_actions, lr=0.1, seed=None):
        """Preference and value tables: two zero-initialized embeddings."""
        with d2l.seeded(seed):
            policy, value = (nn.Embedding(num_states, num_actions),
                             nn.Embedding(num_states, 1))
        nn.init.zeros_(policy.weight), nn.init.zeros_(value.weight)
        return cls(policy, value, lr)
```

コンストラクタ `mlp` はテーブルの代わりに、それぞれ `tanh_mlp` で構築したネットワークを与える。すなわち、`obs_dim` 個の入力と行動ごとに 1 つの選好を持つ方策ネットワークと、同じ形状で 1 つの出力を持つ価値ネットワークである。いずれも、この程度の小規模なタスクでよく使われる幅である 64 ユニットの隠れ層を 1 つ持ち、活性化関数にはオンポリシー手法の大規模な調査で方策ネットワークに最も有効であると報告された tanh を用いる (Andrychowicz et al. 2021)。14.7 節のソフトマックス $\pi_\theta(a \mid s) \propto \exp(\text{preference}_a)$ は、テーブルの行に対してと同様にネットワークの出力に作用する。スコアには隠れ層も含まれるようになり、自動微分が追加の連鎖律を適用する。

```
def tanh_mlp(in_dim, hidden, out_dim):
    """A network with one tanh hidden layer."""
    return nn.Sequential(nn.Linear(in_dim, hidden), nn.Tanh(),
                         nn.Linear(hidden, out_dim))

@d2l.add_to_class(ActorCritic)
@classmethod
def mlp(cls, obs_dim, num_actions, hidden=64, lr=1e-2, seed=None):
    """The same container with the tables replaced by one-hidden-layer nets."""
    with d2l.seeded(seed):
        policy, value = (tanh_mlp(obs_dim, hidden, num_actions),
                         tanh_mlp(obs_dim, hidden, 1))
    return cls(policy, value, lr)
```

14.9.1.3 価値回帰

テーブルの場合、学習されたベースラインの更新は 1 つの要素を観測された 1 つのターゲットに向かって動かしていた。ネットワークの場合、それに対応するのは回帰（regression）、すなわち $\hat{V}(s_t)$ とバッチ全体のターゲットとの平均二乗誤差（mean squared error）に対するオプティマイザのステップである。`fit_value` は、固定されたターゲットベクトルに向かってこのようなステップを `num_steps` 回、価値ネットワーク `ac.value`（上記の `ActorCritic` が持つもの）上で実行する。

ベースラインにはバッチあたり何ステップ必要だろうか。以下の両タスクにおいてターゲットは大きさが 100 程度の割引収益であるのに対し、Adam の 1 ステップは各パラメータをおよそ学習率 0.01 だけ動かす。バッチあたり 1 ステップでは、ベースラインはターゲットに追いつかない。後述する振り子タスクでは、ターゲットの分散のうちベースラインが説明する割合 $1 - \mathrm{Var}(\hat{G} - \hat{V})/\mathrm{Var}(\hat{G})$ は、大半のシードで実行終了時までに 0.4 から 0.8 の間までしか上がらず、さらに低いままにとどまるシードもある。重み $\hat{G}_t - \hat{V}(s_t)$ の残りの部分は、本来なら良いベースラインが取り除くはずのノイズである。バッチあたり 40 ステップとすればこの割合は約 0.99 まで高まるため、以下の訓練ループでは 40 を用いる。

```
def fit_value(ac, obs, target, num_steps=1):
    """Regress the value head on a fixed target: eq_value_baseline for nets."""
    obs, target = torch.as_tensor(obs), torch.as_tensor(target)
    for _ in range(num_steps):
        loss = ((ac.V(obs) - target) ** 2).mean()
        ac.opt_v.zero_grad()
        loss.backward()
        ac.opt_v.step()
    return loss.item()
```

14.9.1.4 時間制限

本節の両タスクは時計によってエピソードを止める。CartPole は 500 ステップ後、Pendulum は 200 ステップ後である。14.1 節でその帰結を確認したとおり、時間制限で打ち切られたエピソードの報酬和はその収益ではない。過程は本来その後も続いていたはずだからである。残りの累積報酬はこの打ち切りを終端として扱ってしまう。上限までポールのバランスを保ち続けた CartPole のエピソードを考えてみよう。打ち切り付近では残りの累積報酬が 1 に向かって縮む一方、価値ネットワークには他の場面と同じくバランスのとれたポールが見えているため、エピソード序盤の直立状態とほぼ同じ価値を予測する。すると終盤ステップの重み $\hat{G}_t - \hat{V}(s_t)$ は強い負の値となり、更新はポールを立て続けていたまさにその行動の確率を引き下げてしまう。$\gamma = 0.99$、$T = 500$ とし、価値ネットワークが永遠にバランスを保ち続ける価値である $1/(1-\gamma) = 100$ を予測するとしよう。$t = 490$ における残りの累積報酬は $\hat{G}_t = (1 - 0.99^{10})/0.01 \approx 9.6$ であり、重みは -90 近くになる。演習問題 1 では、エピソード全体にわたってこの重みを計算する。

その補正は、打ち切りによって隠された将来の価値を足し戻すことである (Pardo et al. 2018)。これは時間制限をタスクの一部ではなく訓練上の便宜として扱うことを意味する。もし時間制限がタスクの一部であれば、残り時間を状態に含めるべきである（14.1 節）。あるエピソードがステップ $t = 0, \ldots, T-1$ を持ち、s_T を最後の行動が到達した状態とし、term $\in \{0, 1\}$ でその状態が終端状態かどうかを記録するとする。このとき、ステップ t のターゲットは次式となる。

$$y_t = \hat{G}_t + (1 - \text{term})\, \gamma^{T-t}\, \hat{V}(s_T). \tag{14.92}$$

終端したエピソードには何も加算されない。打ち切られたエピソードでは、記録が止まった状態からそのまま続いていたかのように、すべてのステップが記録停止時の状態の割引価値を受け取る。上の $t = 490$ の例では補正によって $0.99^{10} \cdot 100 \approx 90.4$ が加算されるため、$y_t \approx 100$ となって重みは約 0 に戻る。`Batch`（14.7 節）はこれに必要な情報をすでに記録している。すなわち、`next_obs` は各ステップに対する s_{t+1} を保持し、`term` は打ち切りではなく終端のみを記録する（14.1 節の規則）。`truncation_bootstrap` は、各エピソードの最終ステップでのみ $\gamma(1-\text{term})\hat{V}(s_T)$ を保持しそれ以外が 0 であるベクトルを構築し、逆方向スキャン $x_t + \gamma x_{t+1} + \gamma^2 x_{t+2} + \cdots$（14.8 節）をその上で実行することで、その価値に係数 γ^{T-t} を掛けて手前のすべてのステップへと伝播させる。

```
@d2l.add_to_class(d2l.Batch)
def truncation_bootstrap(self, gamma, V):
    """gamma^(T-t) (1 - term_T) V(s_T) for every step t of each episode:
    the discounted value of the future that a time-limit cut did not see.
    V maps a batch of numpy states to numpy values."""
    tail = np.zeros_like(self.rew)
    last = self.ep_ends - 1
```

```
        tail[last] = gamma * (1 - self.term[last]) * V(self.next_obs[last])
        return self.backward_scan(tail, gamma)
```

14.9.2 CartPole における REINFORCE

方策ネットワーク、価値ネットワーク、およびターゲットがそろったところで、訓練ループを組み立て、ポール付きの台車で実行してみよう。

14.9.2.1 訓練ループ

`train_reinforce` は、任意の方策オブジェクトに対して学習されたベースライン付きの REINFORCE を実行する関数であり、`make_agent(seed)` がエージェントを構築し、`env_name` が環境を指定する。各更新では `rollout` で 8 エピソードを収集し、ターゲット y_t を計算し、$y_t - \hat{V}(s_t)$ から計算した重みで方策ステップを 1 回とった後、40 回の回帰ステップで $\hat{V}$ をターゲットに当てはめる。ネットワークを用いる点以外にも、14.8 節の学習されたベースラインによる実行とは 4 つの違いがある。オプティマイザは素の SGD ではなく、各 `Policy` が保持する Adam である。重みは `d2l.standardize` を通してバッチ平均を引き、バッチ標準偏差で割る。ベースラインは要素ごとではなく回帰によって当てはめる。そして打ち切られたエピソードにはブートストラップを施す。

Adam のもとでは、すべての重みを 1 つの正の数で割ってもステップはほとんど変わらない。Adam は各座標をそのスケールの移動推定値で割るからである（演習問題 5 でこれを厳密に示す）。それでも正規化には 2 つの働きがある。中心化はバッチ自身から計算される第 2 のベースラインとして機能する。除算は、実行が進むにつれて収益が増大する間も重みのスケールを一定に保つ。このドリフトには、減衰の遅い Adam の移動平均では遅れてしか追従できないからである。

結果として得られる更新は、目的関数 $J(\theta)$ の勾配ではない。バッチ内のステップ数を N、正規化された重みを $\tilde{w}_t$ とすると、方策ステップは次の式に従う。

$$\hat{g} = \frac{1}{N} \sum_t \tilde{w}_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t), \qquad \tilde{w} = \mathrm{standardize}\big(y - \hat{V}(s)\big). \tag{14.93}$$

4 つの要因が $\hat{g}$ を $\nabla_\theta J$ から隔てている。因子 γ^t が落とされているため、終盤ステップのスコアも序盤ステップと同じ重みで扱われる。これは 14.8 節の残りの累積報酬推定量であり、一般にはいかなる目的関数の勾配でもない (Nota and Thomas 2020)。打ち切られたエピソードのターゲット y_t は $\hat{V}(s_T)$ を含むため、打ち切り時点におけるクリティックの誤差が統計的バイアスとして重みに入り込む。中心化にはバッチ自身の平均を用いるが、これはサンプリングされた行動にわずかに依存する。N およびバッチの標準偏差で割ることで各バッチはそのデータ自身によって再スケーリングされ、その後に Adam がほぼ一定の大きさのステップをとる。この組み合わせに直接適用できる確率近似（stochastic approximation）の定理は存在しないため、このループが機能するという根拠は収束証明ではなく実行結果から得られる。

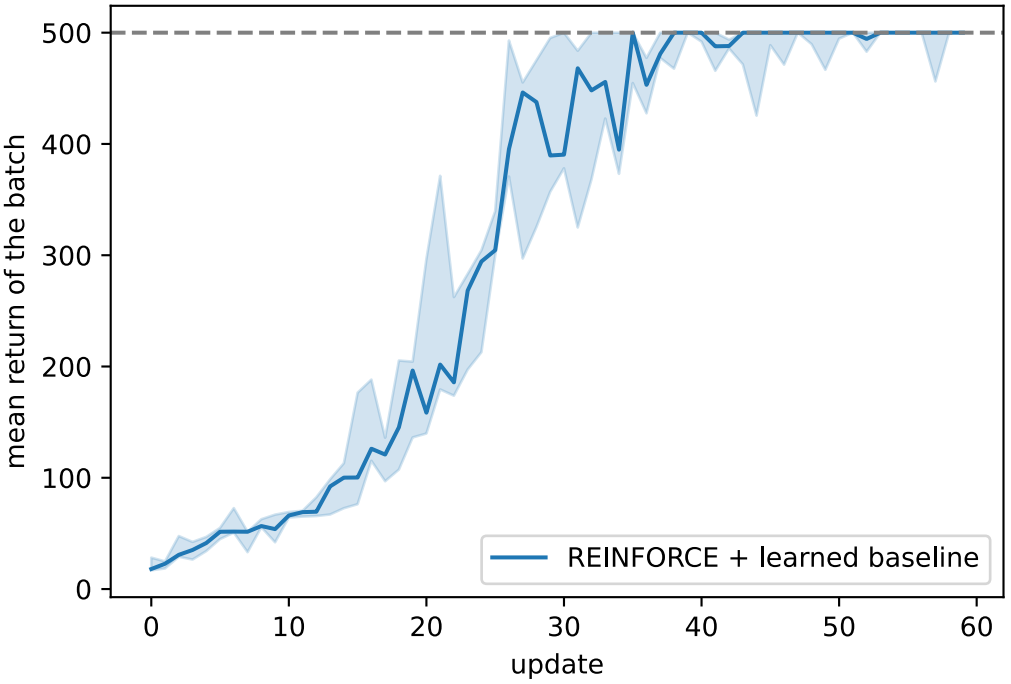
```
def train_reinforce(seed, make_agent, env_name, gamma=0.99, num_updates=60,
                    batch_episodes=8, value_steps=40):
    """REINFORCE with a learned baseline for the agent `make_agent` builds."""
    rng, env = np.random.default_rng(seed), gym.make(env_name)
    ac = make_agent(seed)
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = d2l.rollout(env, ac.act, batch_episodes, rng)
```

```
        G = (batch.reward_to_go(gamma)
             + batch.truncation_bootstrap(gamma, ac.value_np))   # time limits
        w = d2l.standardize(G - ac.value_np(batch.obs))
        L = d2l.policy_step(ac, batch.obs, batch.act, w)
        fit_value(ac, batch.obs, G, num_steps=value_steps)  # fit V-hat well
        yield float(batch.episode_returns().mean()), L
```

14.9.2.2 CartPole

CartPole は古典的なポールバランシング問題である (Barto et al. 1983)。状態は冒頭で述べた 4 つの数値であり、2 つの行動は台車を左または右に押す。ポールが立っているステップごとに報酬 1 を得る。ポールが大きく傾きすぎるか台車がレールから外れるとエピソードは終端し、500 ステップに達すると打ち切られるため、達成可能な最大の収益は 500 である。湖での実行と比べると 3 つの設定が変わる。コンストラクタは `ActorCritic.mlp(4, 2)`（`Policy.tabular(16, 4)` の代わり）、環境は `'CartPole-v1'`、割引率は 0.99 であり、これによって実効ホライズン $1/(1-\gamma)$（14.1 節）は 20 ステップから 100 ステップに延びる。`d2l.run_seeds`（オンライン付録「ユーティリティ関数とクラス」）は訓練ジェネレータをシードごとに 1 回実行し、生成された値を積み重ねる。ここでは 3 つのシードを 60 回の更新にわたって訓練する。

```
def cartpole_agent(seed):
    return ActorCritic.mlp(4, 2, seed=seed)

runs = d2l.run_seeds(train_reinforce, 3, make_agent=cartpole_agent,
                     env_name='CartPole-v1')
d2l.plot_curves({'REINFORCE + learned baseline': runs[:, :, 0]},
                xlabel='update', ylabel='mean return of the batch',
                reference=500)
```



未訓練の方策はポールを約 20 ステップの間バランスさせる。どのシードでもバッチ平均は 50 回目の更新前後で 500 ステップの上限に到達し、個々のバッチがそれを下回ることはあるものの、おおむねその水準にとどまる。このタスクでは成功したエピソードのみが打ち切られるため、今回の実行では時間制限の補正を加えても平均収益はほとんど変わらなかったが、補正なしでは実行終盤に方策が上限から外れる頻度が増えた。

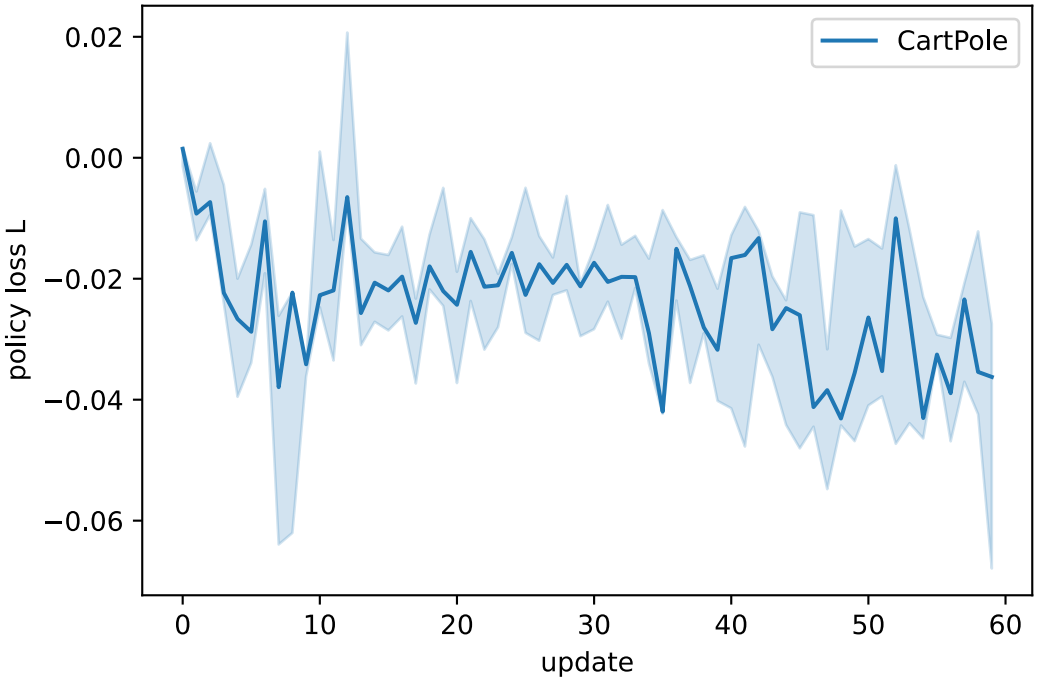
14.9.2.3 代理損失

`train_reinforce` は各更新でもう 1 つの数値、すなわち代理損失（surrogate loss）の値 `L` を返す。これは `policy_step` が最小化する量である（14.7 節）。深層学習フレームワークは更新をスカラー損失によって表現するため、$\hat{g}$ を明示的に組み立てる代わりに、`policy_step` は次式を最小化する。

$$L(\theta) = -\frac{1}{N}\sum_{t} \tilde{w}_t \, \log \pi_\theta(a_t \mid s_t), \tag{14.94}$$

ここで重み $\tilde{w}_t$ は固定されている。重みは NumPy 配列として渡され、計算グラフを持たない。L の勾配は $-\hat{g}$ であるため、L に対する降下ステップは $\hat{g}$ に沿ったステップとなる。しかし、L の値は方策の性能がどれほど良いかを推定するものではない。L は現在のパラメータにおいて正しい導関数を持つように作られているだけでそれ以上の意味はないため、L の減少が収益の増加を意味するとは限らない。CartPole の実行ではその両方を記録している。

```
d2l.plot_curves({'CartPole': runs[:, :, 1]}, xlabel='update',
                ylabel='policy loss L')
```



収益が 20 から 500 へと上昇する間も、L は大きさが 0.1 未満の小さな値にとどまり、初回以降のほぼすべての更新で負となる。正規化された重みは平均 0、標準偏差 1 を持つため、L はバッチ内の対数確率の標準偏差に重みとの相関を掛けて符号を反転させたものに等しい（演習問題 4）。正の重みを持つ行動がすでに平均を上回る確率を持っているときにこの値は負になり、その大きさは収益について何も語らない。訓練の進行を追うには、収益と、方策がどれだけ動いたかのような更新自体を表す量を観察すべきである。

14.9.3 連続行動

ソフトマックスには有限個の行動リストが必要である。しかし多くの制御問題では、トルクや操舵角のように実数値の行動を扱う。14.7 節におけるスコア関数（score function）の導出では、行動集合の有限性を一度も使っておらず、必要なのは $\log \pi_\theta(a \mid s)$ が θ について微分可能であることだけである。したがって、実数値の行動上の確率密度がこの性質を持つ方策であれば、まったく同じループで訓練できる。

14.9.3.1 ガウス方策

14.6 節の振り子（`Pendulum-v1`）に戻ろう。振り子はモーター付きの支点から吊り下げられており、行動はトルク $u \in [-2, 2]$ である。直立位置からの振り子の角度を ϑ と書き、θ は引き続き方策パラメータを表すものとする。14.6 節のプランナは運動方程式を知っており、それを微分していた。方策勾配エージェントは運動方程式を一切知らず、目にするのは観測 $(\cos\vartheta, \sin\vartheta, \dot\vartheta)$ と、各ステップにおける報酬 $-(\vartheta^2 + 0.1\,\dot\vartheta^2 + 0.001\,u^2)$（ただし ϑ は $[-\pi, \pi]$ に折り返される）のみである。したがって、振り子が真上で静止しているときにコストが最小となる。各エピソードはランダムな角度から始まり、200 ステップ後に打ち切られる。制御しない振り子が得る収益は -1200 付近であり、振り子を振り上げてバランスを保つ制御器は約 -200 以上の収益を得る。

ここでは、ネットワークが平均行動を出力するガウス方策（Gaussian policy）$\pi_\theta(a \mid s) =$

$\mathcal{N}(a; \mu_\theta(s), \sigma^2)$ を用いる。GaussianHead がそのネットワークである。**トランク**が隠れ特徴を計算し、デフォルトでは 1 層の tanh 層であるが、trunk 引数によって hidden 個の出力を持つ任意のモジュールに置き換えられる。線形層がその特徴を平均 $\mu_\theta(s)$ に写像し、zero_mean=True を指定するとその重みとバイアスは 0 から始まる（以下の Pendulum の実行ではこれを用いる）。対数標準偏差 $\log\sigma$ はデフォルトでは 1 つの学習されるベクトルであり、したがって σ はどの状態でも同じになる。state_std=True を指定すると、特徴上の第 2 の線形層（$[-5, 2]$ にクランプされる）となり、σ が状態に依存するようになる。演習問題 2 ではこのオプションを用いる。$\log\sigma$ は方策モジュールに属するため、方策ステップは平均と合わせてこれも更新する。

```
class GaussianHead(nn.Module):
    """A Gaussian over actions: a trunk of hidden features, a linear mean,
    and a log standard deviation that is one learned vector or, with
    `state_std=True`, a second linear head clamped to [-5, 2]. The default
    trunk is one tanh layer; any module with `hidden` outputs will do.
    `zero_mean=True` sets the mean layer to zero, so that mu = 0 at first."""
    def __init__(self, obs_dim, act_dim, hidden, state_std=False, trunk=None,
                 zero_mean=False):
        super().__init__()
        self.trunk = trunk if trunk is not None else nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.Tanh())
        self.mean = nn.Linear(hidden, act_dim)
        if zero_mean:
            nn.init.zeros_(self.mean.weight), nn.init.zeros_(self.mean.bias)
        self.state_std = state_std
        self.log_std = (nn.Linear(hidden, act_dim) if state_std
                        else nn.Parameter(torch.zeros(act_dim)))

    def forward(self, obs):
        h = self.trunk(obs)
        if self.state_std:
            return self.mean(h), self.log_std(h).clamp(-5, 2).exp()
        return self.mean(h), self.log_std.exp()
```

GaussianPolicy は ActorCritic を継承し、GaussianHead と tanh_mlp による価値ネットワーク（mlp と同様のもの）を構築したうえで、分布に依存する 3 つのメソッド log_prob、act、act_greedy をオーバーライドする。rollout、policy_step、fit_value、train_reinforce はこのインターフェースのみを用いるため、一切の変更を必要としない。

```
class GaussianPolicy(ActorCritic):
    """The same interface over a Normal instead of a softmax; nothing that
    consumes the interface changes. `head_args` go to GaussianHead."""
    def __init__(self, obs_dim, act_dim, hidden=64, lr=1e-2, seed=None,
                 **head_args):
        with d2l.seeded(seed):
            policy = GaussianHead(obs_dim, act_dim, hidden, **head_args)
            value = tanh_mlp(obs_dim, hidden, 1)
        super().__init__(policy, value, lr)

    def log_prob(self, obs, act):
        """log pi(a|s): the Normal density, summed over action dimensions."""
        mean, std = self.policy(obs)
```

```
        return torch.distributions.Normal(mean, std).log_prob(act).sum(-1)

    def act(self, obs, rng):
        """Sample mu(s) + sigma(s) z, z ~ N(0, I) from the numpy rng."""
        with torch.inference_mode():
            mean, std = self.policy(torch.as_tensor(obs))
        return mean.numpy() + std.numpy() * rng.standard_normal(
            mean.shape, dtype=np.float32)

    def act_greedy(self, obs, rng=None):
        """The mean action mu(s)."""
        with torch.inference_mode():
            return self.policy(torch.as_tensor(obs))[0].numpy()
```

環境はサンプリングされた各トルクを $[-2, 2]$ にクリップするため、act は非有界なガウス分布からのサンプルをそのまま返してよい。log_prob はクリップ前のサンプルをスコア付けするが、演習問題 2 で推定量が不偏に保たれる理由を示す。更新は log_prob をその勾配（スコア）を通じてのみ用いる。$z = (a - \mu)/\sigma$ とおくと、ガウス分布のスコアは次式となる。

$$\frac{\partial \log \pi}{\partial \mu} = \frac{z}{\sigma}, \qquad \frac{\partial \log \pi}{\partial \log \sigma} = z^2 - 1. \tag{14.95}$$

正の重みのもとでは、平均を上回るサンプルは μ を引き上げ、どちらの方向であれ平均から離れたサンプルは σ を引き上げ、平均に近いサンプルは σ を引き下げる。log_prob をガウス密度の式と、またその勾配をこれら 2 つの式と、$\sigma = 0.5$ およびランダムな 5 個の状態と行動において照合してみよう。平均層のバイアスは μ に導関数 1 で入るため、その勾配は μ に関するスコアを 5 個のサンプルにわたって合計したものになる。

```
sigma = 0.5
gp = GaussianPolicy(3, 1, seed=1)
gp.policy.log_std.data.fill_(np.log(sigma))
rng = np.random.default_rng(1)
obs = rng.standard_normal((5, 3)).astype(np.float32)
act = rng.standard_normal((5, 1)).astype(np.float32)
z = (act - np.stack([gp.act_greedy(o) for o in obs])) / sigma
hand = -0.5 * z ** 2 - np.log(sigma) - 0.5 * np.log(2 * np.pi)
print('density:', np.allclose(gp.log_prob_np(obs, act), hand.sum(-1), atol=1e-5))
gp.log_prob(torch.as_tensor(obs), torch.as_tensor(act)).sum().backward()
g_mu, g_ls = gp.policy.mean.bias.grad.numpy(), gp.policy.log_std.grad.numpy()
print('score in mu:', np.allclose(g_mu, (z / sigma).sum(0), atol=1e-4),
      '| score in log sigma:', np.allclose(g_ls, (z ** 2 - 1).sum(0), atol=1e-4))
```

```
density: True
score in mu: True | score in log sigma: True
```

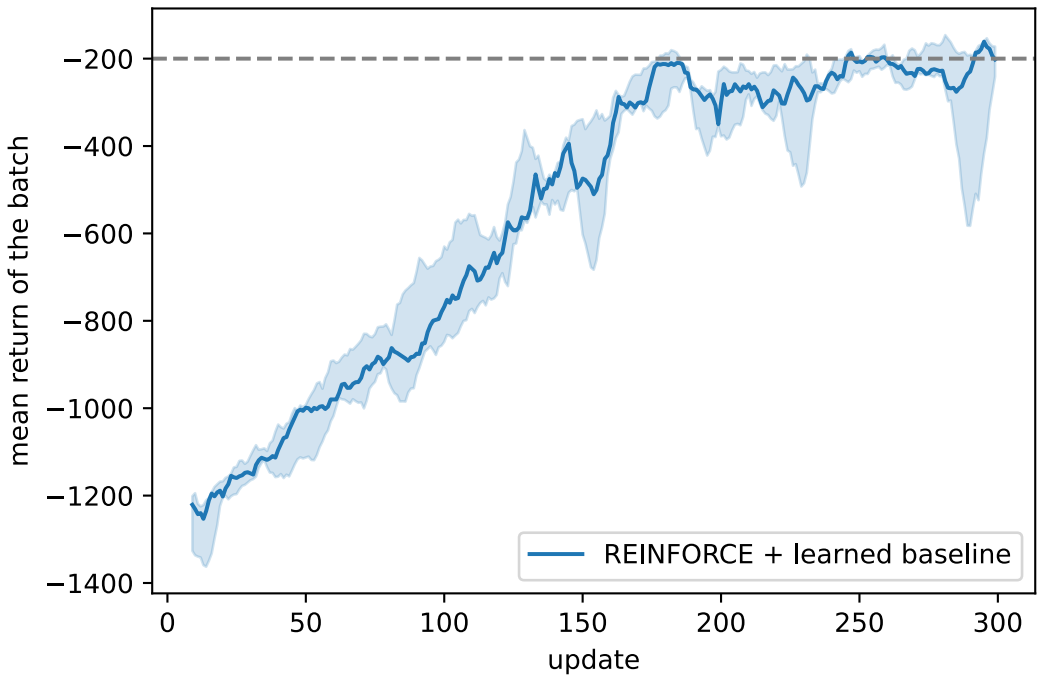
3 つの確認はいずれも True を出力する。すなわち、log_prob はガウス密度であり、その勾配は更新が用いるスコアとなっている。

14.9.3.2 Pendulum

曲線には割引なしの 200 ステップ収益を表示するが、更新では実効ホライズン 20 ステップに相当する $\gamma = 0.95$ で割引を行う。ここでの割引率はタスクの一部ではなく設計上の選択である。トルクは 100 ステップ後のコストよりも直後の数十ステップにはるかに強く影響し、ホライズンを短くすることで、報告される目的関数に対する多少の統計的バイアスと引き換えに分散の小さいターゲットが得られる。今回の実行では $\gamma = 0.9$ でもほぼ同等に訓練できた。

`pendulum_agent` は平均層の重みとバイアスを 0 に設定し（`zero_mean=True`）、他の層の乱数抽出にかかわらずすべてのシードが $\mu = 0$、$\sigma = 1$ から始まるようにする。平均が $[-2, 2]$ の外にあると学習は停滞してしまう。環境がほぼすべてのサンプルを同じ境界値にクリップするため、どのサンプルが引かれたかに報酬が依存しなくなり、行動に関する情報をまったく持たない重みがスコアに掛けられることになるからである。ここで用いる層の初期化では初期平均は境界の十分内側に収まるが、より大きな重みを持つ初期化では一部のシードで外に出てしまい、そうしたシードは停滞する。方策の最終層の重みを非常に小さくするか 0 にすることは、オンポリシー手法における標準的な推奨事項である (Andrychowicz et al. 2021)。方策はバランスを保つ前にまず振り子を振り上げることを学習しなければならないため、8 エピソードからなる更新を 300 回実行して訓練する。

```
def pendulum_agent(seed):   # every seed starts at mu = 0
    return GaussianPolicy(3, 1, seed=seed, zero_mean=True)

runs_p = d2l.run_seeds(train_reinforce, 3, make_agent=pendulum_agent,
                       env_name='Pendulum-v1', gamma=0.95, num_updates=300)
d2l.plot_curves({'REINFORCE + learned baseline': runs_p[:, :, 0]},
                xlabel='update', ylabel='mean return of the batch',
                reference=-200, smooth=10)
```



収益は -1200 付近から始まる。すべての実行が振り子を振り上げてバランスを保つことを学習する。10 回の更新にわたる平均は、約 150〜300 回の更新（すなわち数十万環境ステップ）の後に -200 を上回り、最良値はおおむね -150 から -200 の間に収まる。その後の更新もほとんどはその水準付近にとどまるが、数回の更新にわたって数百ほど後退することもある。実行全体を通じて σ は初期値の約半分まで低下するが、これは良い行動の近傍で演習問題 2 が予測する方向と一致する。ここでは時間制限の補正が重要になる。Pendulum のエピソードはすべて打ち切られるため、補正なしでは同じ予算内で -200 に到達したシードは今回の実行では約半数にとどまった。

これほど小さなタスクに対して、数十万ステップというのは膨大な経験量である。1,600 ステップからなる各バッチは 1 回の更新に使われただけで破棄され、重みは各行動の価値の学習された推定値ではなくエピソード全体の収益から計算されている。ソフトアクタークリティック（soft actor-critic、15.5 節）は、同じタスクにおいて約 5,000〜11,000 ステップ以内に -200 へ到達する。1 サンプルからなぜそれほど多くを学習できるのかという問いを、第 15 章で取り上げる。

14.9.3.3 スコア関数勾配とパスワイズ勾配

連続行動では、期待報酬（expected reward）を微分する第 2 の方法も可能になる。単一の状態、ガウス方策 $a \sim \mathcal{N}(\mu, \sigma^2)$、および既知の微分可能な報酬 $Q(a)$ を考えよう。$\mathbb{E}[Q(a)]$ の μ に関する勾配には、互いに等価な 2 つの形式がある。

$$\nabla_\mu \mathbb{E}[Q(a)] = \mathbb{E}\left[Q(a)\,\frac{a-\mu}{\sigma^2}\right] = \mathbb{E}[Q'(\mu+\sigma z)], \qquad z \sim \mathcal{N}(0,1). \tag{14.96}$$

1 つ目の等号は、本章を通じて用いてきたスコア関数の恒等式である。2 つ目の等号は $a = \mu + \sigma z$ を代入することで μ を分布から Q の内部へと移し、サンプルを通して微分する。これは**パスワイズ**勾配、すなわち再パラメータ化（reparameterization）勾配であり (Kingma and Welling 2014)、Figure 14.21 に各推定量の勾配がどこを流れるかを示す。既知の Q のもとでは両者とも不偏である。$Q(a) = -(a-1)^2$ において $\mu = 0$、$\sigma = 0.5$ としたとき（厳密な勾配は 2）、両者のノイズを比較しよう。

```
def Q(a):                    # a stand-in critic, differentiable in the action
    return -(a - 1.0) ** 2

sigma, N = 0.5, 100000
a = sigma * np.random.default_rng(2).standard_normal(N, dtype=np.float32)
g_score = Q(a) * a / sigma ** 2              # score-function samples, mu = 0
at = torch.as_tensor(a).requires_grad_(True)  # pathwise: grad through a
Q(at).sum().backward()
g_path = at.grad.numpy()                     # per-sample dQ/da (da/dmu = 1)
print(f'score:    mean {g_score.mean():.2f}, variance {g_score.var():.1f}')
print(f'pathwise: mean {g_path.mean():.2f}, variance {g_path.var():.2f}')
print(f'score variance if Q gains a constant +10: '
      f'{((Q(a) + 10) * a / sigma ** 2).var():.0f}; pathwise is unchanged')
```

```
score:    mean 1.98, variance 21.5
pathwise: mean 2.00, variance 1.00
score variance if Q gains a constant +10: 281; pathwise is unchanged
```

どちらの標本平均も 2 に近い。1 サンプルあたりの厳密な分散はスコア関数で 21.75、パスワイズ推定量で 1.0 と約 20 倍の開きがあり、さらに定数 10 を Q に加えるとスコア関数の分散は

281.75 に跳ね上がる。Q がスコアに掛け合わされるためである。パスワイズ推定量は Q' のみに依存するため、定数を加えても変化しない。演習問題 6 でこれらの分散を導出するが、その比は Q、σ、およびオフセットに依存する。低い分散の代償は、行動の微分可能な関数として Q にアクセスする必要があることである。実際には Q は学習されたクリティック Q_w であり、その行動勾配 $\partial Q_w/\partial a$ の誤差が方策勾配の統計的バイアスとなる。

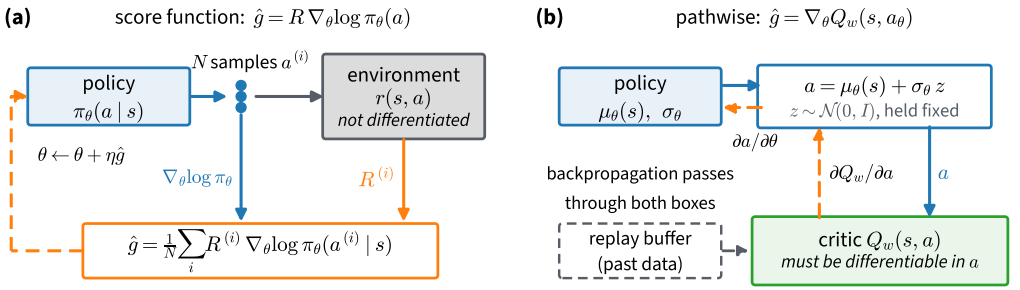


Figure 14.21: 同じ勾配に対する 2 つの推定量。(a) スコア関数推定量は返された報酬を通じてのみ環境を用いるため、環境は微分可能である必要がない。(b) パスワイズ推定量はサンプルを $a = \mu_\theta(s) + \sigma_\theta z$（ここでは単一の学習される σ_θ を持つ）と書き、$\partial Q_w/\partial a$ と $\partial a/\partial\theta$ を通して微分する。クリティック Q_w は行動について微分可能でなければならない。これはリプレイされたデータ上の回帰によって訓練でき、$\partial Q_w/\partial a$ における誤差は勾配の統計的バイアスとなる。

パスワイズ形式はまた、連続制御が行動に関する最大化を回避する手段でもある。価値ベースの手法は $\max_a Q(s,a)$ を計算して行動を選ぶが、これは湖の上では 4 つの行動からの参照で済むものの、$a \in \mathbb{R}^d$ に対しては毎ステップの最適化問題となる。決定論的方策勾配（deterministic policy gradient）は代わりに、学習されたクリティックを通じた $\partial Q_w/\partial a$ に従って、価値の高い行動を出力するようアクターネットワークを訓練する (Silver et al. 2014)。

14.9.4 状態間の汎化

冒頭では、似た状態に似た値を与える関数を求めた。Equation 14.91 によれば、1 回の勾配ステップはカーネル k に従って状態間に広がる。ここでは、CartPole の訓練に用いたものと同じ形状の、新規構築した価値ネットワークについてその広がりを測定する。

14.9.4.1 状態間の結合のプローブ

このプローブは、$\hat{V}$ を $[-1,1]^4$ 内の 256 個のランダムな状態のうちの 1 つにおいて、現在の推定値より 1 単位高いターゲットへ向かう 1 回の勾配ステップによって引き上げ、全 256 個の状態における変化を記録する。続いて、16 要素のテーブルに対しても同じ操作を行う。各プローブでは Adam オプティマイザ `opt_v` を素の SGD（`d2l.plain_sgd`、オンライン付録「ユーティリティ関数とクラス」）に置き換え、この 1 ステップが Equation 14.91 に従うようにする。Adam のままでは、各パラメータがその移動スケールによって再スケーリングされてしまうからである。`fit_value` は二乗誤差の半分と比べて勾配が 2 倍になる平均二乗誤差を最小化するため、テーブルにおけるステップサイズ 0.25 はその要素を 0.5 だけ動かす。次のコードセルでは、残りの 255 個のプローブ状態を動かした状態からの距離順に並べ替える。

```
net_probe = ActorCritic.mlp(4, 2, seed=0)
tab_probe = ActorCritic.tabular(16, 4, seed=0)
for ac, lr in ((net_probe, 0.02), (tab_probe, 0.25)):  # one plain step
    ac.opt_v = d2l.plain_sgd(ac.value, lr)
S = np.random.default_rng(0).uniform(-1, 1, (256, 4)).astype(np.float32)
before = net_probe.value_np(S)
fit_value(net_probe, S[:1], before[:1] + 1.0)      # raise ONE state
dV = net_probe.value_np(S) - before
order = 1 + np.argsort(np.linalg.norm(S[1:] - S[0], axis=1))  # nearest first
before_t = tab_probe.value_np(np.arange(16))
fit_value(tab_probe, np.arange(1), before_t[:1] + 1.0)
dV_t = tab_probe.value_np(np.arange(16)) - before_t
print(f'network: nudged state {dV[0]:+.2f}; 10 nearest probes '
      f'{dV[order[:10]].mean():+.2f}, 10 farthest {dV[order[-10:]].mean():+.2f}; '
      f'{(dV[1:] < 0).sum()} of 255 moved down')
print(f'table:   nudged entry {dV_t[0]:+.2f}; '
      f'largest move among the other fifteen: {np.abs(dV_t[1:]).max():.6f}')
```

```
network: nudged state +0.49; 10 nearest probes +0.43, 10 farthest -0.06; 15 of 255 moved down
table:   nudged entry +0.50; largest move among the other fifteen: 0.000000
```

動かした状態は上昇し、それに最も近い 10 個のプローブもその変化量の 80% 超というほぼ同等の幅で上昇する。変化量は距離とともに減衰し、最も遠いプローブでは 0 付近かわずかに負となり、逆方向へ動くプローブも現れる。一方、テーブルは動かした要素を $+0.50$ 動かすだけで、他の要素は一切動かさない。最も遠いプローブがどれだけ下がるか、また何個のプローブが下がるかはネットワークのランダム初期化（random initialization）に依存するが、全体の形状は変わらない。Figure 14.22 は 1 次元の状態上で同じ対比を描いている。

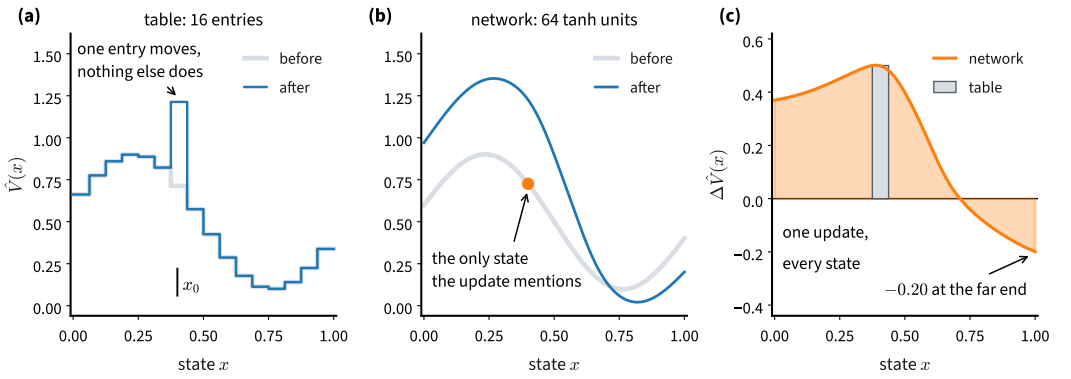


Figure 14.22: 1 回の価値更新が影響を及ぼす範囲。1 次元の状態上の価値推定を 16 要素のテーブルと 64 個の tanh ユニットを持つネットワークとして当てはめ、それぞれ印を付けた状態 x_0 において現在の推定値より 1 単位高いターゲットへ向けて 1 回の勾配ステップをとる（ステップサイズ η_V はいずれも x_0 が $+0.50$ だけ動くように選ぶ）。(a) テーブルは 1 つの要素のみを動かし、他は一切動かさない。(b) ネットワークはあらゆる状態の推定値が x_0 の推定値とパラメータを共有しているため、曲線全体を動かす。(c) 2 つの変化量を並べた比較。一次近似では、ネットワークの変化量は $\eta_V \left(y - \hat{V}(x_0)\right) k(x, x_0)$（$y$ はターゲット）であり、ここで k は x と x_0 におけるパラメータ勾配の内積である。すなわち、x_0 の周囲に正の隆起が生じ、$x \approx 0.7$ を超えると負に転じて遠端では -0.20 に達する。

正の隆起は汎化（generalization）である。1 つの状態での経験が似た状態における推定値に情報を伝えるため、ネットワークは状態空間を人手で分割しなくても CartPole を学習できる。負の遠方領域は干渉（interference）である。状態空間のある領域に当てはめると、バッチに含まれない領域の推定値まで変化してしまう。バランスをうまく保つ CartPole の方策はほぼ直立した状態のバッチばかりを収集するため、それに当てはめるとバッチが一度も訪問しない大きなポール角での推定値が動いてしまう（演習問題 7 でこれを詳しく追う）。こうした変化により、各更新がそのバッチ自体にはよく適合していても、訓練曲線が一時的に落ち込むことがある。

14.9.4.2 モンテカルロエージェントの限界

本節の価値ネットワークは、ターゲットに自身の推定値 $\hat{V}(s_T)$ が含まれる時間制限の打ち切り時点を除き、観測された収益に当てはめられる。打ち切りから離れた箇所では、その当てはめは通常の回帰である。毎ステップで**ブートストラップ**し、次の状態における推定値をすべてのターゲットに組み込む手法では、この性質が失われる。ターゲットが更新のたびに動き、更新はもはや固定された目的関数の勾配ではなくなり、非線形近似器や別の方策からのデータを用いると反復が発散し得る (Tsitsiklis and Van Roy 1997)。

ここまでで、REINFORCE 推定量、当てはめたベースライン、および時間制限の補正を用いて、連続状態と連続行動の上で方策ネットワークを訓練できるようになった。それでもこのエージェントにはまだ 3 つの限界がある。ターゲットがエピソード全体の収益であるため、すべてのエピソードが終わるまで待たなければならない（15.1 節ではより早く学習するためにブートストラップを行う）。各バッチを 1 ステップだけ使って破棄してしまう（15.2 節では制御された数ステップにわたってバッチを再利用し、15.4 節では過去の遷移をリプレイする）。そして 1 回の更新が方策をどれだけ動かすかを制限していないが、大きく動かしすぎると現在の収益も次に収集されるデータも損なわれ得る。第 15 章ではこれら 3 つすべてを取り上げ、あわせてここでの数十万ステップとソフトアクタークリティックの 1 万ステップとの間にある隔たりを埋めていく。

14.9.5 手法の見取り図

本章と次章では多くの手法を扱う。それぞれの手法は、何を学習するのか、そしてどのデータで更新を駆動できるのかという 2 つの問いによって位置づけられる。Figure 14.23 はこの 2 つの問いに沿って各手法を配置したものである。

Table 14.3 に、両章のアルゴリズムを、それぞれが何を推定するか、どのデータを利用できるか、およびどこで導入されるかとともに一覧にする。

Table 14.3: これら 2 章のアルゴリズムの見取り図。「〜で言及」は本書で説明するが実装はしない手法を示す。

アルゴリズム	推定対象	更新を駆動できるデータ	掲載箇所
価値反復法、方策反復法	ベルマン作用素のスイープによる厳密な V^*、方策評価による V^π	なし（カーネル P と報酬 r を使用）	14.2 節
行動クローニング	デモンストレーション上の交差エントロピーによる $\pi(a \mid s)$	固定されたエキスパートデータセット	14.5 節
DAgger	上と同じ当てはめ	必要に応じてエキスパートが再ラベル付けした学習者自身の状態	14.5 節

アルゴリズム	推定対象	更新を駆動できるデータ	掲載箇所
LQR、iLQR、移動ホライズン MPC	リッカチ漸化式による厳密な二次残りコスト $V_t(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_t \mathbf{x}$ と線形フィードバック $\mathbf{u}_t^* = -\mathbf{K}_t \mathbf{x}_t$	なし（既知のダイナミクス $f(\mathbf{x}, \mathbf{u})$ と二次コストを微分）	14.6 節
Q 学習（表形式）	サンプルによるバックアップからの Q^*	任意の行動方策の遷移（オフポリシー）	14.3 節
SARSA	探索を含めた行動方策自身の Q	自身の遷移（オンポリシー）	14.3 節
確実性等価計画	サンプルから集計したモデルの Q^*	任意の行動方策の遷移（カウント集計）	14.3 節
UCB、トンプソンサンプリング	腕の平均とその不確実性	自身の試行（バンディット）	14.4 節
REINFORCE	スコア関数による $\nabla_\theta J$	現在の方策からの新規の軌跡のみ	14.7 節
ベースライン付き REINFORCE、RLOO	より低分散な同じ勾配	現在の方策からの新規の軌跡	14.8 節
GRPO	グループ正規化されたアドバンテージ（価値ネットワークなし）	プロンプトごとの新規グループ（各グループを数回のクリップ付きステップで再利用）	重みは 14.8 節、機構は 15.8 節
ネットワーク上の学習されたベースライン付き REINFORCE	ネットワークとしての π_θ と $\widehat{V}$	現在の方策からの新規の軌跡	14.9 節
アクター・クリティック（A2C）	π_θ とブートストラップされたクリティック	新規の（現在に近い）軌跡	15.1 節
PPO	クリップされた確率比のもとでの π_θ	数ステップ再利用される 1 バッチ	15.2 節
KL 正則化方策最適化（RLHF の背後にある目的関数）	報酬によって参照方策から傾斜させた方策	学習された報酬でスコア付けされたロールアウト	15.3 節（大規模な RLHF はより詳しく扱う「言語モデル」パートに委ねる）
DQN、Double DQN	ネットワークとしての Q	過去の経験のリプレイバッファ	15.4 節
Rainbow	DQN とその検証済み構成要素	リプレイバッファ	15.9 節で言及
DDPG、TD3	クリティックとそれを最大化するよう訓練された決定論的アクター	リプレイバッファ（オフポリシー）	15.5 節で言及
SAC	ツインソフトクリティックと、報酬＋エントロピーを最大化する確率的スクワッシュアクター	リプレイバッファ（オフポリシー）	15.5 節
学習されたモデル内でのモデル予測制御、方策へ蒸留された探索	回帰で当てはめたダイナミクスと報酬のモデル、およびその内部で探索された計画	モデルへ再当てはめされるエージェント自身の蓄積経験	15.6 節
Dyna、MBPO	オフポリシー学習器のために追加の遷移を想像するのに用いる同じモデル	リプレイバッファ＋モデルロールアウト	15.6 節で言及
悲観主義を伴うオフライン Q 学習、CQL	データが薄い領域にペナルティを課した Q	固定されたログデータセット（相互作用は一切なし）	15.7 節

アルゴリズム	推定対象	更新を駆動できるデータ	掲載箇所
Decision Transformer	収益条件付き系列モデル	固定されたログデータセット	15.7 節で言及
MuZero、Dreamer	計画や想像を行うための学習された環境モデル	モデルを通じてリプレイされる自身の相互作用	15.9 節で言及
DPO	選好から直接求める正則化最適解	固定された選好データセット	15.8 節（大規模な選好最適化はより詳しく扱う「言語モデル」パートに委ねる）

14.9.6 まとめ

連続状態上のテーブルでは訪問のたびに行が必要になるため、方策と価値推定はネットワークへと置き換わる。テーブルとは、ワンホット状態上の線形ネットワークという特別な場合にほかならない。方策勾配の導出に必要なのはスコアのみであるため、学習されたベースライン付きの REINFORCE は 3 つの実践的な追加要素を伴ってそのままネットワークへ引き継がれる。すなわち、ベースラインがターゲットに追従できるよう十分な回数の回帰ステップで当てはめること、重みを正規化すること、そして時間制限で打ち切られたエピソードを最終状態の価値でブートストラップすることである。このループが実行する更新は J の不偏勾配ではなくステップごとの代理損失に基づくものであり、その損失の値は性能について何も語らない。ガウス方策はこのループを連続トルクへと拡張し、それによって Pendulum のすべての実行が数十万ステップの後に振り子を振り上げてバランスを保つようになる。1 回の更新はあらゆる状態における推定値を動かし、似た状態を引き上げる一方で離れた状態を引き下げる。このエージェントはエピソード全体の終了を待ち、各バッチを 1 度だけ使い、ステップ幅も制限しない。まさにそこから第 15 章が始まる。

14.9.7 演習問題

1. **時間制限における割引**。`train_reinforce` は各バッチ内のエピソードの割引なし平均収益を返す一方、更新では各ステップを $\gamma = 0.99$ での残りの累積報酬によって重み付けする。CartPole の各ステップは報酬 1 を獲得し、環境は 500 ステップに達したエピソードをすべて打ち切る。
    1. 長さ T のエピソードの割引収益が $(1-\gamma^T)/(1-\gamma)$ であることを示し、$T \in \{20, 100, 200, 500\}$ に対してその値を求めよ。これより、割引なし収益と割引収益がエピソードを同じ順序に順位付けすることを結論づけよ。エピソードが 200 ステップから 500 ステップに延びたとき、それぞれは何倍に増えるか。
    2. 時間制限の補正なしの場合について、上限に達したエピソードの残りの累積報酬 $\hat{G}_t$ を $t \in \{0, 400, 450, 490\}$ において計算せよ。価値ネットワークは状態のみを入力とし、成功したエピソードの終盤の状態は序盤の状態とよく似ている。成功したエピソードの最終付近のステップの重みはどちらの符号をとりやすく、方策の更新はそれらのステップでとられた行動にどのような影響を与えるか。
    3. `train_reinforce` はこれらのターゲットに `batch.truncation_bootstrap(gamma, ac.value_np)` を加算する。ポールが立っているすべての状態で $\hat{V}$ が $1/(1-\gamma)$ を予測すると仮定せよ。このとき、打ち切られたエピソードのすべてのステップにおけるターゲット y_t が、上限までの残りステップ数によらず $1/(1-\gamma)$ となることを示せ。ポールが倒れて終わるエピソードのターゲットが依然として終端に向かって縮むのはなぜか。また、それが正しいのはなぜか。
    4. Pendulum のエピソードはすべて打ち切られるのに対し、CartPole では成功したエピソードのみが打ち切られる。この事実を用いて、補正を取り除くと CartPole よりも

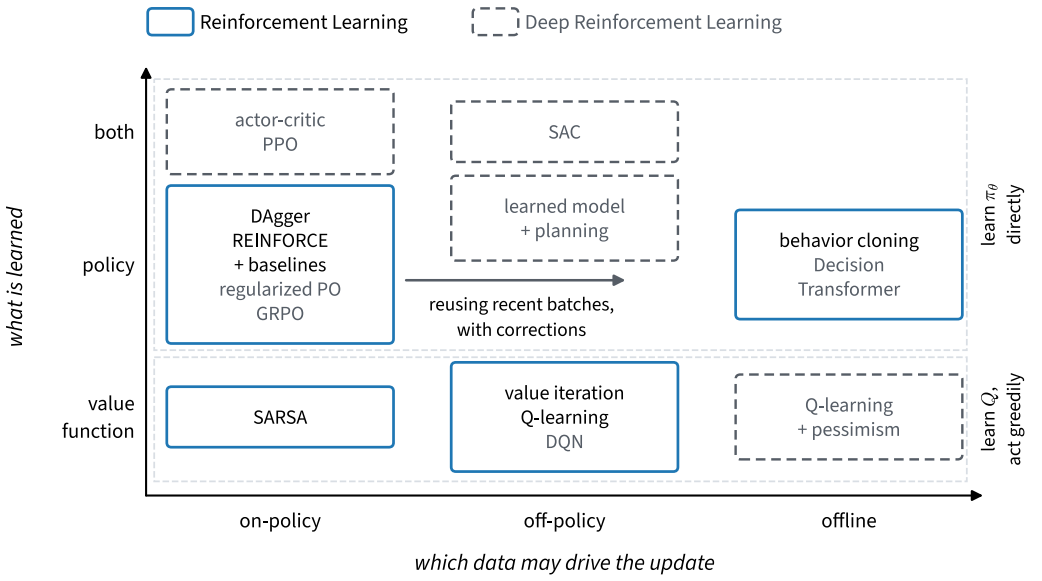


Figure 14.23: 何を学習するか、およびどのデータで更新を駆動できるかによって整理した、本章と第 15 章の各手法。実線の枠は本章の手法、破線の枠と灰色の名称は第 15 章の手法を表す。行動クローニングは固定されたデモンストレーションの集合から学習するのに対し、DAgger は学習者自身の状態にエキスパートがラベルを付け直したものから学習するため、オンポリシーの列に置かれる。価値反復法は特定の方策からのデータではなく既知のカーネルを更新に用いるため、オフポリシーの列に置かれる。オンポリシーの方策枠から出る矢印は直近バッチの再利用を表しており、収集時点から方策が変化しているため補正が必要になる。オフポリシーの列にある学習されたモデルはエージェント自身の蓄積された経験に当てはめられ、エージェントはその内部で計画を行う（15.6 節）。

Pendulum の学習がはるかに遅くなる理由を説明せよ。

2. [code] **ガウス分布のスコア**。GaussianHead は 1 次元の行動を平均 μ と対数標準偏差 $\ell = \log\sigma$（コード上の log_std）によってパラメータ化する。$z = (a-\mu)/\sigma$（a はサンプリングされた行動）とおく。
   1. $\partial \log\pi/\partial\mu = z/\sigma$ および $\partial\log\pi/\partial\ell = z^2 - 1$ であること、さらにその両方が方策のもとで平均 0 を持つことを示せ。
   2. Q を行動の二次関数とし、その二階導関数を Q'' とする。$\mathbb{E}\big[Q(a)\,(z^2-1)\big] = \sigma^2 Q''$ となることを、$\mathbb{E}[z^4] = 3$ を用いて示せ。方策勾配の期待ステップは σ を、Q の最大値の近傍においてどちらの方向へ動かすか。このことは、平均が最良の行動へ近づいた方策の探索について何を意味するか。
   3. Pendulum はサンプリングされた各トルクを $[-2, 2]$ にクリップする一方、log_prob はクリップ前のサンプルをスコア付けする。クリップを伴う環境の収益に対して、スコア関数推定量が不偏に保たれる理由を説明せよ。
   4. pendulum_agent のエージェントを GaussianPolicy(3, 1, state_std=True) で構築して σ が状態に依存するようにし、Pendulum の実行と同様に train_reinforce で 3 シードを訓練せよ。25 回の更新ごとに、σ を垂下状態 $(\cos\vartheta, \sin\vartheta, \dot\vartheta) = (-1, 0, 0)$ と直立状態 $(1, 0, 0)$ で記録せよ。σ が先に縮小するのはどちらの状態であり、それは小問 b とどのように整合するか。収益曲線を runs_p と比較せよ。

   Richard S. Sutton および Andrew G. Barto 著 Reinforcement Learning: An Introduction の演習問題 13.4 を改変。

3. [code] **不連続な報酬**。@eq-score-vs-pathwise の 2 つ目の等号は Q を通して微分する。$\mu = 0$、$\sigma = 0.5$（g_score と g_path の計算と同じ設定）とし、しきい値報酬として $Q(a) = 1$（$a > 1$ のとき）、$Q(a) = 0$（それ以外のとき）をとる。
   1. 厳密な勾配 $\nabla_\mu \mathbb{E}[Q(a)]$ を計算せよ。パスワイズ推定量は各サンプルに対して何を返すか。2 つ目の等号の導出のどの段階が破綻するか。
   2. その計算における Q を滑らかな階段関数 $Q_k(a) = 1/\big(1 + e^{-k(a-1)}\big)$（$k \in \{1, 10, 100, 1000\}$）に置き換え、g_score と g_path の平均および分散を記録せよ。サンプルは単精度であるため、$k \geq 100$ では最小付近のサンプルで因子 $e^{-k(a-1)}$ がオーバーフローし、式をそのまま書くと導関数がそこで NaN になる。数値的に安定なシグモイドを用いて Q_k を実装せよ。両者の分散を k に対してプロットせよ。大きい k において分散が低いのはどちらの推定量か。g_path の分散が k に（Q_k' の高さと幅を通じて）どう依存するか、また g_score の分散が k に（Q_k を通じて）どう依存するか説明せよ。
   3. Pendulum はサンプリングされた各トルクを $[-2, 2]$ にクリップする。クリップされたトルクを通じたパスワイズ推定量は、この区間の外のサンプルに対して何を返すか。演習問題 2c のスコア関数推定量がこれらのサンプルを無駄にしないのはなぜか。

4. **代理損失の値**。policy_step は、現在の方策が収集したばかりのバッチ上で代理損失 Equation 14.94 を評価する。d2l.standardize による重みは、平均 0、標準偏差 1 を（バッチの N ステップにわたって）持つ。
   1. $L(\theta) = -s\,c$ であることを示せ。ここで s はバッチにわたる $\log\pi_\theta(a_t \mid s_t)$ の標準偏差であり、c は重みとこれらの対数確率との相関係数である。これより $|L(\theta)| \leq s$ を結論づけよ。
   2. 行動次元が 1 である GaussianPolicy について、$s \approx 1/\sqrt{2}$ が任意の平均ネットワークと任意の σ に対して成り立つことを示せ。このことは、Pendulum の実行における方策損失について何を意味するか。

3. CartPole において方策がほぼ決定論的になるにつれて、s はどのように振る舞うか。このことは、訓練終盤における L の値について何を意味するか。

5. [code] **Adam のもとでの正規化**。●素の SGD のもとでは、重みをその標準偏差で割ると実効学習率が変化する（演習問題 6、14.8 節）。`train_reinforce` は代わりに Adam で更新する。各座標に対するステップは $-\eta\,\hat{m}_t/(\sqrt{\hat{v}_t}+\epsilon_0)$ であり、ここで $\hat{m}_t$ と $\hat{v}_t$ は勾配 g_t と g_t^2 のバイアス補正済み移動平均（減衰率は $\beta_1 = 0.9$ および $\beta_2 = 0.999$）である。

   1. $\epsilon_0 = 0$ のとき、すべての勾配に同じ定数 $c > 0$ を掛けても Adam の各ステップが不変であることを示せ。

   2. 次式を示せ。

$$\hat{v}_t = \frac{1-\beta_2}{1-\beta_2^t}\sum_{i=1}^{t}\beta_2^{t-i}g_i^2.$$

      Pendulum の実行長である $t = 300$ において、最も古い二乗勾配と最も新しい二乗勾配の重みは何倍異なるか。ある座標の勾配が一定の符号を保ったまま、300 回の更新の実行の途中でスケールが 10 分の 1 に低下したとする。実行終了時のステップは開始時のステップと比べてどの程度の倍率だけ小さくなるか。

   3. `d2l.standardize` の中心化は `train_reinforce` 内で維持したまま除算を取り除き、重みを `A - A.mean()`（ただし `A = G - ac.value_np(batch.obs)`）として、2 つの変種がスケールのみ異なるようにせよ。各バッチの重みの標準偏差を記録し、両変種をそれぞれ 3 シードずつ CartPole と Pendulum で訓練せよ。各タスクにおいて重みのスケールはどのように推移するか。小問 b の結果から、除算を取り除くことで収益曲線がより大きく変化すると予測されるのはどちらのタスクか。

6. **スコア関数推定量とパスワイズ推定量の分散**。●サンプル `g_score` と `g_path` は、Equation 14.96 の 2 つの形式を $Q(a) = -(a-1)^2$、$\mu = 0$ において推定する。$\mathbb{E}[z^4] = 3$ および $\mathbb{E}[z^6] = 15$（$z \sim \mathcal{N}(0,1)$ に対して）を用いよ。

   1. 1 サンプルあたりの分散が、スコア関数推定量では $15\sigma^2 + 14 + 1/\sigma^2$、パスワイズ推定量では $4\sigma^2$ となることを示せ。$\sigma = 0.5$ において両者を評価し、出力された分散と比較せよ。
   2. 定数 c を Q に加えることは、ベースライン $b = -c$ を差し引くことと同じである。スコア a/σ^2 を b^* の公式（演習問題 2、14.8 節）に代入して、$b^* = -(1+3\sigma^2)$ となること、およびスコア関数推定量の最小分散が $6\sigma^2 + 8$ となることを示せ。$c = 10$ に対して出力された分散を導出した式と照合せよ。
   3. $\sigma \to 0$ とする。最適ベースラインがある場合とない場合とで、2 つの分散はそれぞれどのように振る舞うか。スコア関数の分散の極限 8 を、Q の μ の周りでのテイラー展開を通じて説明せよ。決定論的アクターに対して利用可能なまま保たれるのは 2 つの推定量のうちどちらか。

7. [code] **接カーネル**。●`dV` の測定では、$\hat{V}$ を 1 つのプローブ状態において `fit_value` の素の SGD 1 ステップによって引き上げる。価値ネットワークのパラメータを w と書き、その接カーネルを

$$k(s,s') = \nabla_w \hat{V}(s)^\top \nabla_w \hat{V}(s') \tag{14.97}$$

   と書く（Equation 14.91 と同様）。

1. サイズ η の勾配ステップを $\frac{1}{2}\big(y - \hat{V}(s_0)\big)^2$ 上で 1 回とると、s における推定値が $\eta\,\big(y - \hat{V}(s_0)\big)\,k(s_0, s) + O(\eta^2)$ だけ変化することを示せ。`fit_value` は代わりに $\big(y - \hat{V}(s_0)\big)^2$ を最小化するが、この予測はどう変わるか。
2. 新規に構築した `net_probe` について、勾配 $\nabla_w \hat{V}$ を 256 個のプローブ状態 `S` において計算し、カーネル値 $k(s_0, s)$ を s_0 = `S[0]` に対して計算せよ。Adam を使わずに $\eta = 10^{-3}$ で $y = \hat{V}(s_0) + 1$ に向かう素の勾配ステップを 1 回とり、すべてのプローブにおける実測の変化量を予測と比較せよ。
3. $k(s_0, s)/k(s_0, s_0)$ を距離 $\|s - s_0\|$ に対してプロットせよ。負のカーネル値を持つプローブは何個あり、その個数はコードセルが下方向に動いたと報告したプローブの個数と比べてどうか。
4. バランスをうまく保つ CartPole の方策は、大半が垂直に近い状態からなるバッチを収集する。このカーネルを用いて、そのようなバッチへの当てはめが、バッチには一度も含まれない大きなポール角における価値推定と方策をどう変化させるか説明せよ。また、その変化によってそれらの角度における方策が悪化し得るのはなぜか。

Chapter 15

深層強化学習

第 14 章の最後のエージェントは、数十万ステップをかけて振り子を振り上げた。シミュレータならそれも安価に済む。しかし、ロボットアームであれば先に摩耗してしまうだろうし、自身の回答で訓練される言語モデルでは、回答の各バッチを 1 回の更新ごとに使い捨てるわけにはいかない。そのエージェントは毎回エピソードが終わるまで待ち、各バッチを 1 度しか使わず、1 回の更新で方策（policy）がどれだけ変化するかも一切制限していなかった。1 つのサンプルからどれだけのことを学べるのか、そしてそこからさらに多くを学ばせようとするとどのような問題が生じるのだろうか。

これには 4 つのアイデアが答えを与えるが、それぞれ固有の誤差を伴う。エージェントはエピソードの終了を待つ代わりに、残りの区間に対する学習された推定値を信頼できる (Samuel 1959; Sutton 1988)。ただし、学習が浅いうちはその推定値に統計的バイアスがある。過去の方策によるデータを重み付けし直すか再生して再利用することもできる (Lin 1992)。しかし、データは古くなりターゲットも移動する。方策の振る舞いの変化量によって各更新を抑えることもできる (Kakade and Langford 2002; Schulman et al. 2017)。もっとも、その上界が制限するのは各ステップの幅であり、ステップを重ねた到達先ではない。環境のモデルを学習してその内部でプランニングすることもできる (Sutton 1990)。その場合、モデルの誤差を引き継ぐうえ、ステップごとに計算が必要になる。さらに、学習されたターゲット、状態間で共有されるネットワーク、そして別の方策のデータが組み合わさると、表形式では決して起きない価値の発散が生じうる。制御器、ゲームプレイヤ、そして事後学習された言語モデルは、いずれもこれらのアイデアから構築されている。

本章では、これらのアイデアを 3 つの段階に分けて展開する。まずはオンポリシーおよび正則化付きの方策最適化から始める。15.1 節ではエピソード終了までの待機をブートストラップされたクリティックと多ステップのアドバンテージに置き換え、15.2 節では確率比クリッピング（probability-ratio clipping）によって保護された複数回の更新にわたり各ロールアウトを再利用し、15.3 節では報酬自体が比較データから当てはめられる場合に方策を参照分布へつなぎ留める。次に、オフポリシーの再生とプランニングに進む。15.4 節では凍結したターゲットのコピーと Double DQN によって、再生された遷移上でのニューラル Q 学習を安定化させ、15.5 節ではツインクリティックと最大エントロピー（maximum entropy）の連続行動アクターを組み合わせて数千ステップで振り子を振り上げ、15.6 節では数百ステップから物理法則を学習してモデル内部でプランニングする。最後に、オンライン探索（exploration）や遷移の不確実性がない設定を扱う。15.7 節では固定されたログから保守的に学習し、15.8 節では本章の手法群をトークンレベルの言語モデルの事後学習へと特化させる。

離散行動の各手法は環境ステップ数を共通軸として CartPole で一貫して評価し、ソフトアクタ

ークリティックと学習されたモデルは Pendulum で扱い、検証器タスクと 2 つの小さなグリッドワールドは厳密な閉形式の検証を与える。すべての例は PyTorch と JAX の双方で、CPU 上で数分以内に実行できる。Figure 14.23 と Table 14.3 は両章のすべての手法を位置づけ、15.9 節は本章の範囲を超える発展的トピックを概観する。なお、大規模な選好に基づく事後学習については「言語モデル」パートで扱う。

15.1 アクター・クリティック法と複数ステップ収益

14.9 節のエージェントはポールのバランスを取ることを学習するが、その学習は遅い。各更新では 8 エピソード全体、すなわち最大 4,000 ステップものバランス維持が終わるのを待ち、そのうえでようやく各行動に対し、エピソードの残りの部分が期待よりうまくいったかを問いかける。一方、14.3 節の Q 学習エージェントにはそのような忍耐は不要であった。遷移のたびに、エピソードの残りの代わりにその先の展開に対する自身の推定値を用いて、テーブルの 1 要素を書き換えていたからである。方策勾配エージェントもこれと同じ方法で学習できるだろうか。

結論から言えば可能であり、そのために必要な部品はすでに手元にある。14.9 節で学習されたベースライン $\hat{V}$ は、ある状態に続く報酬の割引和である残りの累積報酬（reward-to-go）を予測するように訓練されている。これがエピソードの残りを予測するのであれば、エージェントはその終了を待つ必要がない。1 ステップ分の報酬に次の状態での予測値を加えたものを、サンプリングされた末尾の代わりに使えるからである。このとき、方策は**アクター**となり、価値推定はその**クリティック**となる。Barto et al. (1983) はまさにこのカートポール問題に対して最初のこのような組を構築し、同論文における「適応的クリティック要素（adaptive critic element）」がクリティックの名前の由来となった。

この置き換えは、将来のランダム性とクリティックの誤差とを引き換えにするものである。本節ではこのトレードオフを詳しく調べる。すなわち、単一の遷移からアクターに何が伝わるのか、サンプリングされた末尾をどれだけ残すべきか、そしてクリティックが取り除く分散よりもクリティック自身の誤差のほうが大きく響く CartPole において、このトレードオフがどのような結果をもたらすのかを明らかにする。本章の残りで扱う手法の大半は何らかのクリティックを備えており、ここで構築する 1 ステップ目標と複数ステップ目標を幾何重みで混合した推定量は、PPO で用いられているものである。

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

タスクとエージェントは 14.9 節のものを引き継ぐ。CartPole の状態はカートの位置・速度とポールの角度・角速度という 4 つの数値からなり、2 つの行動はカートを左または右に押す。ポールが倒れずに立っているステップごとに報酬 1 が与えられ、エピソードは 500 ステップで打ち切られるため、可能な最大の収益は 500 である。割引率は $\gamma = 0.99$ とする。エージェントは `ActorCritic.mlp` であり、それぞれ 64 ユニットの隠れ層を 1 つ持つ方策ネットワークと価値ネットワークから構成され、各ネットワークが独自の Adam オプティマイザを持つ。以下の主な訓練では、1 回あたり 8 エピソードからなる更新を 150 回、環境ステップ数にして最大 600,000 ステップ実行する。新しいハイパーパラメータである `critic_steps` はバッチごとにクリティックが反復する回帰パスの回数を設定し、`grad_clip` はアクターの勾配のノルムに上限を設ける。これらはいずれも訓練ループとあわせて説明する。

```
gamma, num_updates, batch_episodes = 0.99, 150, 8
num_seeds, critic_steps, grad_clip = 3, 20, 0.5
def cartpole_agent(seed):
    return d2l.ActorCritic.mlp(4, 2, seed=seed)
```

15.1.1 時間差分アドバンテージ

15.1.1.1 残りの累積報酬のブートストラップ

クリティックが何を置き換えられるかを確認するため、残りの累積報酬を 1 ステップ分書き出してみよう。

$$\hat{G}_t = r_t + \gamma\, \hat{G}_{t+1}. \tag{15.1}$$

学習されたベースラインは残りの累積報酬に向かって回帰されるため、$\hat{V}(s_{t+1})$ はまさに $\hat{G}_{t+1}$ がサンプリングする量を予測している。そこでサンプルが得られるのを待つ代わりに、予測値を用いて $\hat{G}_t \approx r_t + \gamma\hat{V}(s_{t+1})$ と置き換える。ステップ t におけるスコアの重みは $\hat{G}_t - \hat{V}(s_t)$ であったが、この置き換えによって次のようになる。

$$\delta_t = r_t + \gamma\, \hat{V}(s_{t+1}) - \hat{V}(s_t), \tag{15.2}$$

ただし、$\hat{V}(s_{t+1}) = 0$（s_{t+1} が終端状態のとき）とする。この**時間差分誤差**（temporal-difference error、TD 誤差）は、同じ収益に対する 2 つの予測、すなわち s_t での予測と、実際の報酬を 1 つ得た 1 ステップ後の予測とを比較している。

14.3 節の TD 誤差も、1 点を除けばこれと同じ形をしていた。Q 学習は最適価値を推定するため、最良の継続価値 $\max_{a'} \hat{Q}(s', a')$ でブートストラップするが、ここでのブートストラップは同節の SARSA の目標と同様に、現在の方策自身の継続価値を用いる。終端マスクには `terminated` を用い、`truncated` は用いない。時間制限はタスク自体を終了させずに観測を打ち切るだけなので、打ち切り時点における状態の価値は 0 ではないからである（14.1 節）。`td_target` は、状態を価値に写す任意の関数から、目標 $r_t + \gamma\hat{V}(s_{t+1})$ を `Batch`（`d2l.rollout` が記録する遷移とエピソード境界の平坦なレコード、14.7 節）内のすべての遷移に対して計算する。戻り値は NumPy 配列であるため、クリティックの勾配がこの目標を通って流れることはない。

```
@d2l.add_to_class(d2l.Batch)
def td_target(self, bootstrap, gamma):
    """r_t + gamma (1 - terminated) V(s'), by a numpy bootstrap."""
    return self.rew + gamma * (1 - self.term) * bootstrap(self.next_obs)
```

14.8 節の残りの累積報酬推定量と同様に、アクターの更新では、エピソード後半のステップを前半に対して割り引く因子 γ^t を省略する。

15.1.1.2 1 遷移の手計算

3 つの数値を見れば、この新しい重みが何を意味するかがわかる。クリティックがすべての状態で 100、すなわちポールを永久に倒さず維持したときの価値 $1/(1-\gamma)$（$\gamma = 0.99$ の場合）を予測しているとしよう。ポールが立ったままのステップでは $\delta_t = 1 + 0.99 \cdot 100 - 100 = 0$ となり、そのステップは予測どおりに進んだため、とった行動は推奨も抑制もされない。一方、ポールが倒

れるステップではエピソードが終了してブートストラップがマスクされ、$\delta_t = 1 - 100 = -99$ となる。エピソードを終わらせた行動の確率は強く押し下げられるのである。

初期化されたばかりのクリティックは、あらゆる状態で 0 付近の価値を予測する。すると、行動によってポールが立ったままでも倒れても、すべてのステップで $\delta_t \approx 1$ となる。とられた行動はどれも同じ大きさだけ押し上げられるため、クリティックが両者を区別できるようになるまでアクターはほとんど学習できない。

15.1.1.3 正確なクリティックが与えるアドバンテージ

完全なクリティックがあれば、TD 誤差はモンテカルロ重みと同じ量を推定する。いま仮に $\hat{V} = V^\pi$ であるとしよう。行動価値は $Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^\pi(s')$ を満たす。これは 14.2 節のベルマン方程式を V^π を用いて書き表したものである。したがって、Equation 15.2 を次の状態について平均すると次式が得られる。

$$\begin{aligned}\mathbb{E}\big[\delta_t \mid s_t, a_t\big] &= r(s_t,a_t) + \gamma \sum_{s'} P(s' \mid s_t,a_t)\, V^\pi(s') - V^\pi(s_t) \\ &= Q^\pi(s_t,a_t) - V^\pi(s_t).\end{aligned}$$

右辺はアドバンテージ $A^\pi(s_t,a_t)$ であり、とった行動がその状態における方策の平均よりもどれだけ優れているかを表す。14.8 節の学習されたベースラインは、これを $\hat{G}_t - \hat{V}(s_t)$ によって推定していた。モンテカルロ重みがエピソードの残り全体を必要としたのに対し、ここでは 1 つの報酬と 2 回のクリティック評価を伴う 1 回の遷移だけで、アドバンテージの不偏サンプルが得られる。

訓練中の $\hat{V}$ は V^π と等しくないため、δ_t はアドバンテージのバイアスを持つ推定値となる。14.8 節のゼロ平均恒等式によれば、行動の前に確定している任意の量は、勾配の平均を変えることなく重みから差し引くことができる。現在の状態のみに依存する $\hat{V}(s_t)$ にはこの性質が当てはまる。しかし、次の状態を通じて行動に依存する $\hat{V}(s_{t+1})$ には当てはまらないため、行動の遷移先の状態におけるクリティックの誤差が統計的バイアスとして更新に入り込む（演習問題 1）。このバイアスは、γ^t の省略によってすでに生じている傾きに上乗せされるものである（14.8 節）。

クリティック自身の更新は、サンプリングによる方策評価（policy evaluation）である。14.2 節における方策評価は、V^π を不動点に持つベルマン期待値バックアップを適用していた。$\hat{V}(s_t)$ を $r_t + \gamma \hat{V}(s_{t+1})$ に向かって回帰させることは、ちょうど Q 学習が最適性バックアップをサンプルで置き換えたのと同様に、その期待値を 1 回のサンプリングされた遷移で置き換えることにあたる。クリティックが現在の方策の価値を推定し、アクターがその推定値に対して方策を改善するため、アクター・クリティックは 14.2 節の一般化方策反復法（generalized policy iteration）の近似形となっている。回帰目標が $\hat{V}$ 自身を含んでいるため、勾配は予測 $\hat{V}(s_t)$ についてのみ計算し、目標を通しては計算しない。このような更新は**半勾配**（semi-gradient）ステップと呼ばれ、その期待更新量は固定されたいかなる目的関数の勾配とも一致しない。

15.1.1.4 チェーンの例

ブートストラップはどれだけの分散を取り除き、どれだけのバイアスをつけ加えるのだろうか。小さなチェーンを用いれば、その両方に閉じた形で答えられる。10 個の状態 $0, 1, \ldots, 9$ を考えよう。状態 s からエージェントは常に $s+1$ へ移動し、状態 9 からの遷移でエピソードが終了して報酬 1 が与えられる。各報酬には標準偏差 $\sigma = 0.15$ の独立なガウスノイズが加わり、割引率は $\gamma = 0.97$ とする。クリティックは $e(s) = 0.5\,(1 - s/9)^2$ だけの誤差を持ち、始点では大きく

外れているが終点では誤差がない。ゼロ初期化からの TD 学習では、終端で得られる報酬が後半の状態にまず伝わるため、このような形状の誤差が生じやすい。

n 個の報酬をサンプリングしてから状態 n でクリティックからブートストラップする目標は、割り引かれたその状態でのクリティックの誤差と、n 個の報酬のノイズとを受け継ぐ。そのバイアスと分散は、次の式で与えられる。

$$\text{bias} = \gamma^n e(n), \qquad \text{variance} = \sigma^2 \frac{1-\gamma^{2n}}{1-\gamma^2}, \tag{15.3}$$

ただし上式は $n \le 9$ の場合であり、モンテカルロ目標である $n = 10$ にはバイアスがない。$n = 1$ ではバイアスが 0.38、分散が 0.02 であり、$n = 10$ ではバイアスが 0、分散が 0.17 となる。バイアスの二乗と分散の和である平均二乗誤差は、0.17（$n = 1$）から 0.10（$n = 4$）まで下がったのち、再び 0.17（$n = 10$）まで上昇する。

最良の深さが両端の間にあるのは、深い目標ほど価値がより正確にわかっている状態からブートストラップするためである。もしクリティックがすべての状態で一様に 0.5 だけ外れていれば、ブートストラップを行う深さの範囲で誤差は単調に変化するはずである（演習問題 2）。Figure 15.1 に、この目標の族と実測された曲線を示す。

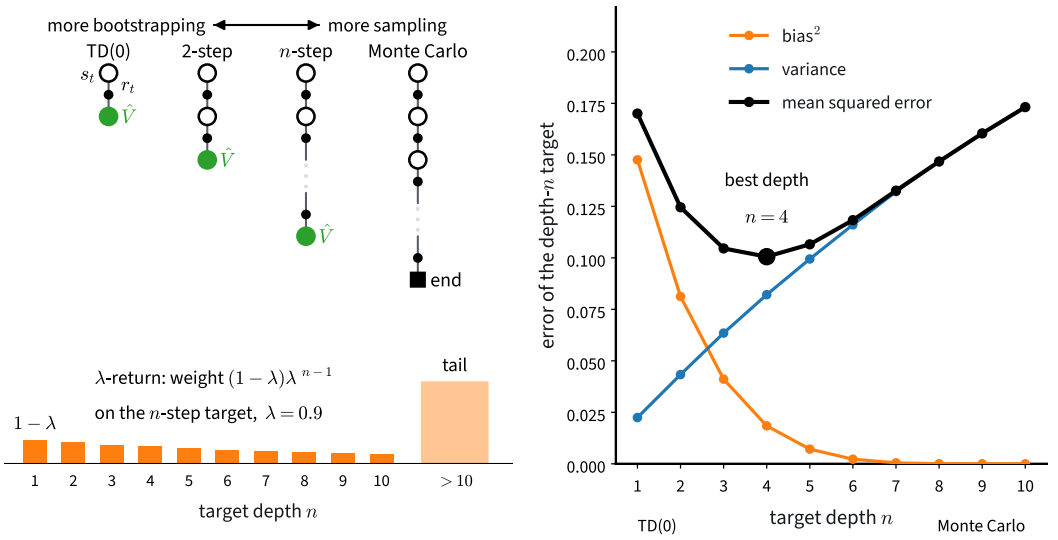


Figure 15.1: ブートストラップ目標の族。左：深さ 1、2、n、および終端までのバックアップ線図。緑のノードは目標がサンプリングを打ち切ってクリティックを信頼する状態を表し、重み $(1-\lambda)\lambda^{n-1}$（本節の λ 収益が $\lambda = 0.9$ のときに各目標へ割り当てる重み）を添えている。右：本文の 10 状態チェーンにおける深さ n の目標のバイアス、分散、平均二乗誤差（20,000 回のロールアウトで測定、最小誤差は $n = 4$）。

15.1.2 アクター・クリティックの更新

15.1.2.1 1 つの数値からの 2 つの更新

これで両方の学習器が同じスカラーから更新されるようになる。クリティックのパラメータを w、クリティックを $\hat{V}_w$ と書こう。遷移 (s_t, a_t, r_t, s_{t+1}) の後、2 つの更新式は次のようになる。

$$w \leftarrow w + \alpha\, \delta_t\, \nabla_w \hat{V}_w(s_t), \qquad \theta \leftarrow \theta + \eta\, \delta_t\, \nabla_\theta \log \pi_\theta(a_t \mid s_t), \tag{15.4}$$

ここで、クリティックのステップサイズは 14.3 節の TD 学習と同様に α とし、アクターのステップサイズは 14.7 節の方策勾配と同様に η とする。クリティックの更新は 1 ステップ目標に向かう半勾配ステップであり、アクターの更新は 14.8 節の方策勾配ステップにおいて重みを δ_t としたものである。たった 1 つの数値が両方の役割を果たす。すなわち、δ_t はクリティックにはその予測がどれだけ外れていたかを伝え、アクターには直前にとった行動がクリティックの予想より良い結果になったか悪い結果になったかを伝える（Figure 15.2）。δ_t の計算には 1 回の遷移しか要らないため、Q 学習がテーブルを更新したのと同じように、エージェントが行動しながら毎ステップ Equation 15.4 を適用でき、学習を始める前に何かが終了するのを待つ必要はない。

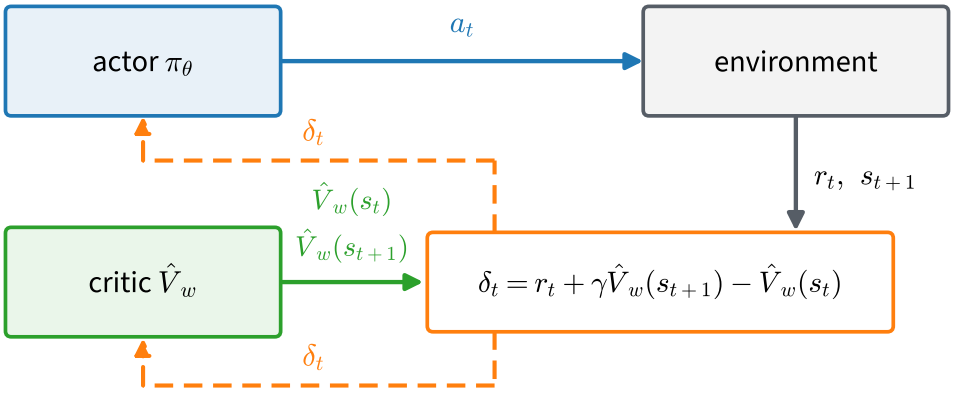


Figure 15.2: アクター・クリティックのループ。1 回の遷移から 1 つの TD 誤差が生じ、クリティックの予測を修正するとともにアクターのステップを重み付けする。

クリティックの目標は両方のネットワークに伴って移動する。アクターのステップはクリティックが予測すべき価値を変化させ、クリティックのステップは自身の目標を変化させる。もしアクターがクリティックの追従できる速さを超えて動けば、アクターは古くなったクリティックの誤差で重み付けされることになる。Konda and Tsitsiklis (2000) は、線形クリティックと減衰するステップサイズのもとで、クリティックのステップサイズがアクターよりもゆっくり縮小し、クリティックが現在の方策に追いつき続けられるときにアクター・クリティックが収束することを証明した。以下のループでは、バッチごとにクリティックへ複数回の回帰パスを与えることで同じ効果を得る。

15.1.2.2 バッチ化されたループ

以下のコードでは、遷移のたびに Equation 15.4 を適用するのではなく、14.9 節の 8 エピソードからなるバッチを維持したうえで各バッチに対する処理を変更する。まずクリティックが、バッチの 1 ステップ目標に向かって `critic_steps` 回の回帰パスを実行し、各パスの前に現在のクリティックから目標を再計算する。回帰器自身の予測を含む目標に向かって回帰することを**適合 TD**（fitted TD）と呼び、目標を再計算することでそれは移動目標となる。続いてアクターが、再適合

されたクリティックの TD 誤差で重み付けされた 1 ステップを実行する。本節ではこのループをバッチ型 1 ステップ・アクター・クリティックと呼ぶ。これに最も近い著名な手法は Mnih et al. (2016) のアドバンテージ・アクター・クリティックであり、同じスコアを n ステップのアドバンテージで重み付けし、さらにエントロピーボーナスを加える。元の手法である A3C は多数のワーカーを非同期に実行していたが、現在の実装で用いられている同期版の **A2C** は、多数の環境から並列に収集したロールアウトごとに 1 回の同時勾配ステップを実行する。

アクターの重みには、14.9 節のモンテカルロ重みのようにバッチ内で正規化したものではなく、生の TD 誤差をそのまま用いる。これは Adam がステップをスケーリングする仕組みのためである。Adam は各勾配座標を過去の二乗の移動平均の平方根で割るが、この移動平均はデフォルトの減衰率では約 1,000 ステップ分を記憶し、今回の 150 回の更新よりも長い。そのため、最初のうちは重みのスケールによらずステップの大きさがほぼ同じになり、後のステップは勾配が初期の大きさより小さくなったときにのみ縮小する。クリティックが改善するにつれて、TD 誤差の標準偏差は訓練初期の約 3 から上限付近での 0.1 以下へと低下し、その時点では大半がクリティック自身のノイズとなるが、それに伴って Adam のステップも実行全体で約 8 分の 1 に縮小する。もし TD 誤差を正規化してしまうと、勾配が初期の大きさに保たれ、そのノイズが定める方向へ Adam が大きなステップを取り続けてしまう（演習問題 6）。

訓練の初期には生の重みが、ひいてはアクターの勾配が、正規化した重みを用いる場合よりも大きくなる。そこで一般的な値である 0.5 で勾配のノルムに上限を設ける。Adam のもとでは、この上限は 1 ステップの大きさを制限するのではなく、初期の大きなバッチが Adam の移動平均にどれだけ強く寄与するかを制限する。`policy_step_clip` は、バッチ内の行動の重み付き対数確率に対して 1 回の勾配ステップをとる 14.5 節の `policy_step` にこの上限を加えたものであり、上限適用前に測定したノルムを返す。

```
def actor_loss(ac, obs, act, weight):
    """-(1/n) sum_t w_t log pi(a_t|s_t), the loss of d2l.policy_step."""
    w, obs, act = (torch.as_tensor(x) for x in (weight, obs, act))
    return -(w * ac.log_prob(obs, act)).mean()

def policy_step_clip(ac, obs, act, weight, clip=grad_clip):
    """d2l.policy_step with the actor's gradient norm capped at `clip`;
    returns the norm measured before the cap."""
    loss = actor_loss(ac, obs, act, weight)
    ac.opt_pi.zero_grad()
    loss.backward()
    gnorm = nn.utils.clip_grad_norm_(ac.policy.parameters(), clip)
    ac.opt_pi.step()
    return float(gnorm)
```

損失のみを計算する関数が `actor_loss` であり、本節末尾のプローブで再利用する。

訓練ループを実装するのが `train` である。各更新では `d2l.rollout` でバッチを収集し、関数 `weight(ac, b)` を呼び出してバッチ `b` の各ステップにおけるアクターの重みを求め、その重みで上限付きのアクターステップを 1 回実行したうえで、これまでの環境ステップ数、バッチの平均収益、上限適用前のアクターの勾配ノルムを順に生成する。この重み関数の中でクリティックの適合も進めるため、本節のすべての推定量に対してループ自体は変わらず、重み関数だけを入れ替えればよい。アクター・クリティックの重みは `td_weight` である。`d2l.fit_value`（14.9 節）がクリティックの予測と目標ベクトルの二乗誤差に対して Adam の 1 ステップを実行し、`ac.value_np` が状態のバッチ上でクリティックを評価し、クリティックの反復パスが終わった後の重みは再適合されたクリティックの生の TD 誤差となる。

```
def train(seed, ac, weight, num_updates=num_updates,
          batch_episodes=batch_episodes):
    """Batched policy gradient with the per-step weights weight(ac, b);
    yields (env steps, mean return of the batch, actor grad norm)."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    steps = 0
    for _ in range(num_updates):
        b = d2l.rollout(env, ac.act, batch_episodes, rng)
        gnorm = policy_step_clip(ac, b.obs, b.act, weight(ac, b))
        steps += len(b)
        yield steps, float(b.episode_returns().mean()), gnorm

def td_weight(ac, b):
    """Actor-critic: refit the critic toward its one-step target, then
    weight each step by the raw TD error of the refitted critic."""
    for _ in range(critic_steps):   # fresh target, one pass, repeat
        d2l.fit_value(ac, b.obs, b.td_target(ac.value_np, gamma))
    return b.td_target(ac.value_np, gamma) - ac.value_np(b.obs)
```

アクターの重みを計算する前にクリティックを適合させるというのは 1 つの設計選択である。A2C や大半の GAE 実装では、バッチ収集時点のクリティックを用いて TD 誤差を計算し、その後で両方のネットワークを更新する。A2C のようにロールアウトあたりのクリティックの更新が 1 ステップであれば、この順序はほとんど影響しない。しかしバッチあたり 20 パスとなると話は別である。再適合されたクリティックはバッチ自身の状態において V^{π} にはるかに近づくため、クリティックに全面的に依存しているここでのアクターの重みに、クリティックの誤差が乗りにくくなるからである（演習問題 3 でこれら 2 つの順序を比較する）。

毎回新しいバッチを集めることは、もう 1 つの面でもクリティックを助けている。各バッチは現在の方策とそれが訪れる状態から得られるため、現在の振る舞いにとって問題となるクリティックの誤差があれば、次のバッチで関連する遷移に出会いやすい。もっとも、このフィードバックは収束を保証するものではなく、非線形クリティックを用いた半勾配 TD は自方策のデータ上であっても発散しうる (Tsitsiklis and Van Roy 1997)。さらに過去の方策から再生された遷移で訓練すると、同じ更新が関数近似、ブートストラップ、オフポリシーデータという Sutton and Barto (2018) の**死の三要素**（deadly triad）を兼ね備えることになり、発散が頻繁に起こるようになる。この場合は 15.4 節で扱う。

15.1.2.3 CartPole におけるブートストラップ重みとモンテカルロ重み

ブートストラップされた重みは、置き換え元のモンテカルロ重みよりも速く学習するだろうか。比較基準とするのは 14.9 節のループである学習されたベースライン付き REINFORCE であり、重み以外のすべてのコードを両条件で共有できるよう、2 つ目の重み関数 `mc_weight` として記述する。重み以外の違いをなくすため、さらに 2 つの細部を揃える。すなわち、アクターのステップは `policy_step_clip` を通し、クリティックはバッチごとに同じ `critic_steps` 回の回帰パスを（ここでは固定されたモンテカルロ目標 $\hat{G}_t$ に向かって）実行する。14.9 節と同様に、`truncation_bootstrap` が時間制限での打ち切り時点におけるクリティックの割引価値を残りの累積報酬に加え、`d2l.standardize` がバッチ内で重みを標準化し、重みはクリティックのパスより前に計算する。アクターとクリティックは別々のネットワークであるため、この順序によってアクターのステップが変わることはない。

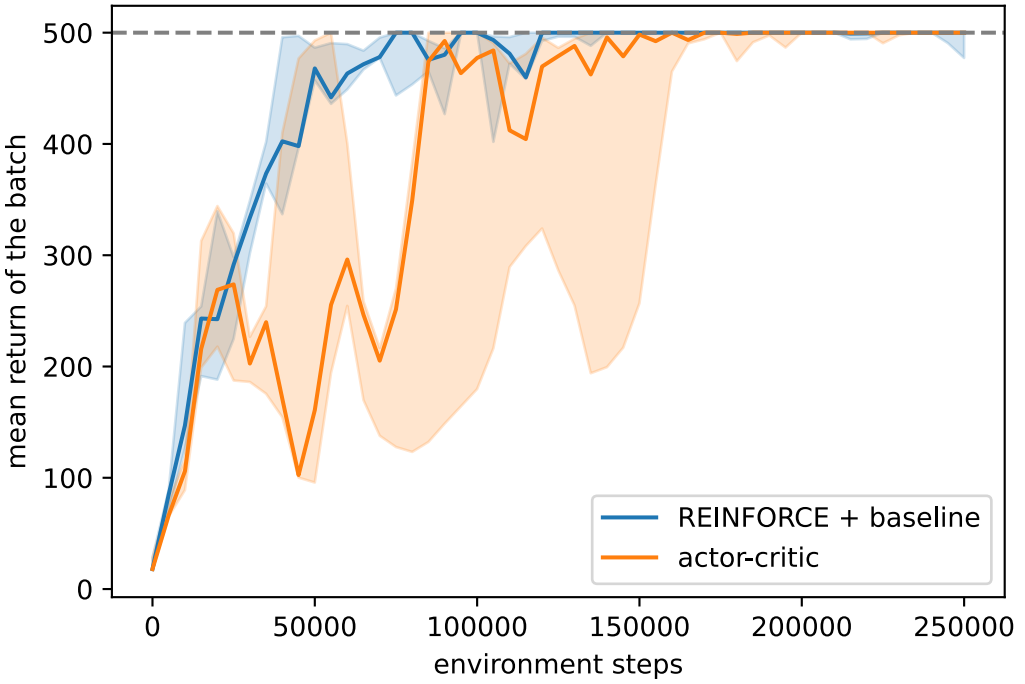
```
def mc_weight(ac, b):
    """REINFORCE with a learned baseline: the standardized Monte Carlo
    advantage G - V, after which the critic fits G."""
    G = (b.reward_to_go(gamma)
         + b.truncation_bootstrap(gamma, ac.value_np))   # time limits
    w = d2l.standardize(G - ac.value_np(b.obs))
    d2l.fit_value(ac, b.obs, G, num_steps=critic_steps)
    return w
```

それぞれの条件を 3 つのシードで訓練し、本章の以降でもコストの尺度として用いる環境ステップ数に対してバッチの平均収益をプロットする。`d2l.plot_curves`（オンライン付録「ユーティリティ関数とクラス」）はシード間の中央値を、最小シードから最大シードまでの帯とともに描画する。更新あたりのステップ数はシードごとに異なるため、まず `d2l.on_grid` で各実行結果を共通のステップ数のグリッド上に補間する。

```
runs = {'REINFORCE + baseline': np.array(
    [list(train(s, cartpole_agent(s), mc_weight)) for s in range(num_seeds)])}
```

次に、`'actor-critic'` 条件を `td_weight` を用いて同じ 3 つのシードで訓練し、両条件を共通のステップ数 `grid` 上へ `d2l.on_grid` で補間して、シード間でのバッチ収益の中央値と最小〜最大の帯をプロットする。

```
runs['actor-critic'] = np.array(
    [list(train(s, cartpole_agent(s), td_weight)) for s in range(num_seeds)])
grid = np.arange(0, 250_001, 5_000)
d2l.plot_curves({name: d2l.on_grid(r, grid, 1) for name, r in runs.items()},
                xlabel='environment steps', ylabel='mean return of the batch',
                x=grid, reference=500)
```



プロットをシードごとに読み解くため、直近 10 バッチの平均収益が初めて 450 に達するまでに各実行が要したステップ数と、アクターの勾配に対する上限が作動した頻度とを集計する。`steps_to` は実行結果の行から最初の該当ステップを見つける関数であり、15.5 節でも再び使用する。

```
def steps_to(run, level, window):
    """Env steps at which the mean return of the last `window` rows of
    run = [(env step, return, ...), ...] first reaches `level`; inf if it
    never does."""
    mean = np.convolve(run[:, 1], np.ones(window) / window, 'valid')
    hit = np.flatnonzero(mean >= level)
    return run[hit[0] + window - 1, 0] if len(hit) else np.inf
```

`steps_to(run, 450, 10)` を `runs` 内の各シードに適用し、10 バッチにわたって平均収益 450 を維持するまでに各条件が何千環境ステップを要したかを、アクターの勾配ノルムが `grad_clip` を超えた更新の割合とあわせて集計する。

```
for name, r in runs.items():
    k = [f'{steps_to(run, 450, 10) / 1000:.0f}' for run in r]
    print(f'{name:>20}: 450 after {", ".join(k)} thousand steps; '
          f'cap active {(r[:, :, 2] > grad_clip).mean():.0%}')
```

```
REINFORCE + baseline: 450 after 79, 64, 77 thousand steps; cap active 0%
        actor-critic: 450 after 113, 187, 112 thousand steps; cap active 29%
```

ここに示した 3 シードのうち少なくとも 2 シードでは、モンテカルロ重みのほうが先に上限付近へ到達する。3 シードは小さなサンプルであるが、同じバッチあたり 20 回のクリティックパスで 40 シードにわたって実験したところ、少なくとも 35 シードで REINFORCE が先に 450 へ到達し、必要な環境ステップ数も概して約 6 分の 1 から 3 分の 1 少なかった。また、上限を大きく下回っ

て終了した実行は REINFORCE には 1 つもなかったのに対し、アクター・クリティックでは 2 つあった。勾配の上限はアクター・クリティックの更新の 5 分の 1 から 5 分の 2 で作動するが、正規化された重みによって勾配が小さく保たれる REINFORCE では一度も作動しない。

CartPole では、ブートストラップによって実行が安定するわけでもない。報酬は毎ステップ決定論的な $+1$ であり、時間制限でのブートストラップが打ち切られた各エピソードの長い末尾をすでにクリティックの価値で置き換えているため、モンテカルロ重みのノイズは最悪の場合よりもはるかに小さい。残るのはクリティックのバイアスであり、それはクリティックがまだ未熟な段階、すなわちアクター・クリティックが遅れをとる段階で最も大きくなる。未訓練のクリティックが一様に $\delta_t \approx 1$ を出すのがその極端な例である。

15.1.2.4 共通バッチ上での 2 つの重み

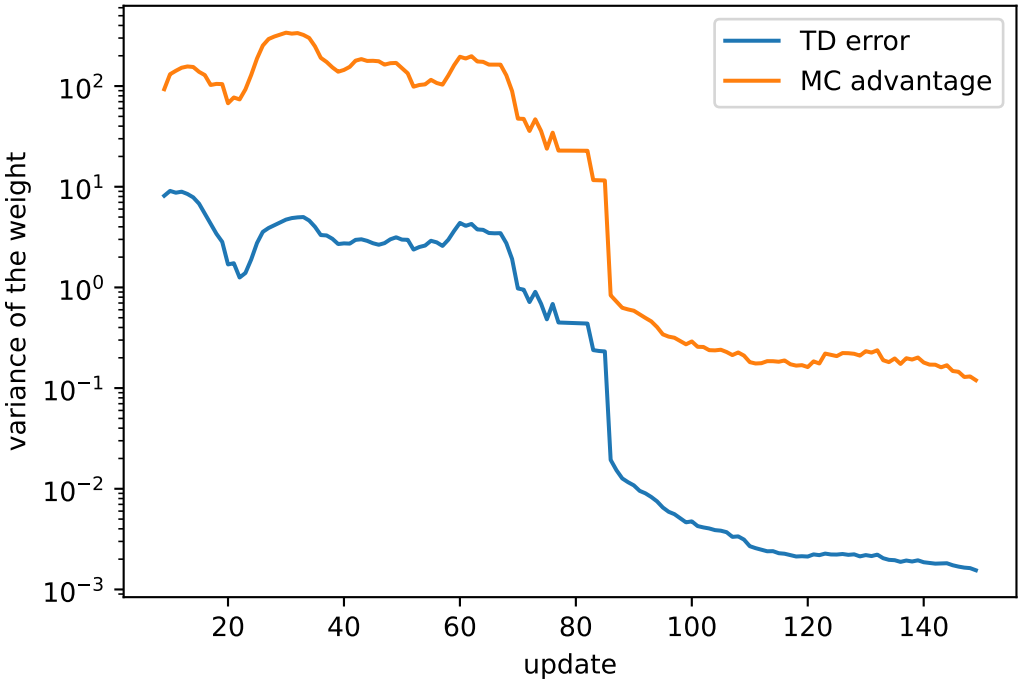
ここでブートストラップが学習を速めないのだとしたら、何を変えているのだろうか。同じバッチ上で両方の重みを測定してみよう。下のセルでは、1 つのシードについて `td_weight` を呼び出す重み関数でアクター・クリティック条件を再実行し、各更新において TD 誤差の標準偏差と、モンテカルロアドバンテージ `A_mc`（標準化前の `mc_weight` の重み）の標準偏差もあわせて記録する。

```
trace = []   # per update: std of the TD error, std of the MC advantage

def traced_td_weight(ac, b):
    """td_weight, recording the spread of both weights on its batch."""
    delta = td_weight(ac, b)
    A_mc = (b.reward_to_go(gamma)   # G of mc_weight, minus V
            + b.truncation_bootstrap(gamma, ac.value_np) - ac.value_np(b.obs))
    trace.append((delta.std(), A_mc.std()))
    return delta

for _ in train(0, cartpole_agent(0), traced_td_weight):
    pass
trace = np.array(trace)
```

`trace` から、最初の 10 回と最後の 50 回の更新における TD 誤差の標準偏差の中央値をモンテカルロアドバンテージのものと比較し、さらに分散比の中央値を計算して、バッチごとの両者の分散を対数スケールでプロットする。

```
for name, rows in (('first 10', trace[:10]), ('last 50', trace[-50:])):
    print(f'{name:>8} updates: std of the TD error {np.median(rows[:, 0]):.2f}'
          f', of the MC advantage {np.median(rows[:, 1]):.2f}')
print(f'median variance ratio, MC over TD: '
      f'{np.median((trace[:, 1] / trace[:, 0]) ** 2):.0f}x')
d2l.plot_curves({'TD error': trace[:, 0] ** 2, 'MC advantage': trace[:, 1] ** 2},
                xlabel='update', ylabel='variance of the weight', smooth=10,
                yscale='log')
```

```
first 10 updates: std of the TD error 2.69, of the MC advantage 7.69
 last 50 updates: std of the TD error 0.04, of the MC advantage 0.41
median variance ratio, MC over TD: 56x
```



同じ遷移上で比べると、最初の数回の更新においてモンテカルロアドバンテージは TD 誤差の 2〜3 倍の標準偏差、すなわち 5〜10 倍の分散を持ち、実行全体では TD 誤差の数十倍の分散を持つ（プロットの縦軸は対数スケールである）。TD 誤差は 1 つの報酬と 1 回の遷移のみに依存するが、モンテカルロアドバンテージはエピソードに残されたすべての報酬と遷移に依存する。ほぼすべてのエピソードが 500 ステップの打ち切りに達するようになると、両者の分散は急激に低下する。TD 誤差の分散が下がるのはその時点でクリティックが正確になっているためであり、モンテカルロアドバンテージの分散が下がるのは打ち切られた各エピソードの末尾をブートストラップがクリティックの価値で置き換えるためである。

分散の削減効果自体は確かにあるものの、CartPole において学習の律速となっているのは分散ではない。REINFORCE は更新ごとに 8 エピソードを平均して重みを正規化しており、十分に適合したベースラインがあればそれで事足りる。これに対し、クリティックのバイアスはクリティックが追いつくまでの間、毎回の更新を遅らせてしまう。

ブートストラップの本質的な強みは構造面にある。TD 誤差は 1 回の遷移しか必要としないため、エピソードが終わる前に更新できることにある。継続タスクではエピソード終了時の収益がそもそも訪れず、1 エピソードが数千ステップ続くタスクでは更新あたり 8 エピソードを集めるのに数万ステップを要する。そのような場面ではクリティックが唯一の選択肢となる（演習問題 5 で完全オンライン版を構築する）。これら両極端の間には調整ダイヤルが存在する。1 ステップではなく数ステップ後にブートストラップすれば、分散削減の大半を維持しつつ、クリティックのバイアスの影響を抑えられるのである。

15.1.3 複数ステップ収益と GAE

15.1.3.1 n ステップ収益

1 ステップ目標とモンテカルロ収益は、ある 1 つの族の両端にあたる。サンプリングされた n 個の報酬を含めてからクリティックでブートストラップすると、次のようになる。

$$\hat{G}_t^{(n)} = r_t + \gamma\, r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n\, \hat{V}(s_{t+n}), \tag{15.5}$$

そして $\hat{G}_t^{(n)} - \hat{V}(s_t)$ をアドバンテージ推定値として用いる。$n = 1$ のときこれは δ_t に一致し、n がエピソードの終端に達するときモンテカルロアドバンテージとなる。ステップを 1 つ増やすごとに、クリティックの誤差はさらに因子 γ だけ割り引かれてより先の状態で評価される一方、サンプリングされた報酬と遷移がもう 1 つ加わる。これはチェーンの例で見たトレードオフそのものであり、そこでは $n = 4$ が両者の均衡点であった。

15.1.3.2 λ 収益と逐次相殺の恒等式

1 つの深さだけを選ぶ代わりに、計算可能な形になる幾何重みを用いてすべての深さを平均することができる。

$$\hat{A}_t^{\mathrm{GAE}} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} (\hat{G}_t^{(n)} - \hat{V}(s_t)), \qquad \lambda \in [0, 1]. \tag{15.6}$$

重み $(1-\lambda)\lambda^{n-1}$ の総和は 1 である。$\lambda = 0$ ではすべての重みが 1 ステップ推定値に置かれる。終了するエピソードでは、終端を超えるすべての深さがモンテカルロアドバンテージに等しくなるため、$\lambda \to 1$ とするとこの混合はモンテカルロアドバンテージに収束する。このように書くと、すべての深さを個別に計算する必要があるように見えるが、混合式が簡約されるためその必要はない。

命題。この混合は TD 誤差の割引和に等しい。

$$\hat{A}_t^{\mathrm{GAE}} = \sum_{l=0}^{\infty} (\gamma\lambda)^l\, \delta_{t+l}. \tag{15.7}$$

証明。各深さの式は逐次相殺によって TD 誤差の和となる。すなわち $\sum_{l=0}^{n-1} \gamma^l \delta_{t+l} = \sum_{l=0}^{n-1} \gamma^l r_{t+l} + \gamma^n \hat{V}(s_{t+n}) - \hat{V}(s_t) = \hat{G}_t^{(n)} - \hat{V}(s_t)$ であり、内側の $\hat{V}$ の項は対になって打ち消し合う。これを Equation 15.6 に代入して 2 つの和の順序を入れ替えると、組 (n, l)（ただし $l < n$）からの寄与は $(1-\lambda)\lambda^{n-1}\gamma^l\delta_{t+l}$ であるため、$\hat{A}_t^{\mathrm{GAE}} = \sum_{l \ge 0} \gamma^l \delta_{t+l}\, (1-\lambda) \sum_{n \ge l+1} \lambda^{n-1}$ となる。内側の等比級数の和は λ^l である。■

この恒等式により、すべての深さにわたる平均がエピソード長に対して線形な 1 回の逆方向パスへと変換され、実際の実装でもそのように計算される (Schulman et al. 2016)。エピソード的タスクでは終端で $\hat{V} = 0$ となり後続の報酬もないため和は終端で止まり、終端より先の δ はすべて 0 となることから無限和は有限和と一致する。

15.1.3.3 逆方向スキャンとしての GAE

和 Equation 15.7 は、そのまま**一般化アドバンテージ推定**（generalized advantage estimation、GAE）のアルゴリズムになっている。すなわち、TD 誤差を計算し、エピソード境界ごとにリセットしながら因子 $\gamma\lambda$ で逆方向に累積すればよい。残りの累積報酬を漸化式 $\hat{G}_t = r_t + \gamma \hat{G}_{t+1}$ で計算する 14.8 節のメソッド `Batch.backward_scan` は、すでにこの漸化式を実装しており、変わるのは入力と割引因子だけである。

```
@d2l.add_to_class(d2l.Batch)
def gae(self, value_fn, gamma, lam):
    """GAE(gamma, lam): the reward-to-go scan, run on the TD errors."""
    delta = self.td_target(value_fn, gamma) - value_fn(self.obs)
    return self.backward_scan(delta, gamma * lam)
```

両端点はバッチ上で検証できる恒等式となっている。$\lambda = 0$ では、このスキャンは TD 誤差そのものを返す。$\lambda = 1$ では、終了するエピソードにおいて逐次相殺が終端まで進み、残りの累積報酬からベースラインを引いたもの、すなわち 14.8 節のモンテカルロアドバンテージを返す。

```
env = gym.make('CartPole-v1')
env.reset(seed=0)
fresh = cartpole_agent(0)
b = d2l.rollout(env, fresh.act, 8, np.random.default_rng(0))
V = fresh.value_np(b.obs)
delta = b.td_target(fresh.value_np, gamma) - V
assert np.allclose(b.gae(fresh.value_np, gamma, 0.0), delta)
assert np.allclose(b.gae(fresh.value_np, gamma, 1.0),
                   b.reward_to_go(gamma) - V, atol=1e-4)
print('lambda = 0 is the TD error; lambda = 1 is the Monte Carlo advantage')
```

```
lambda = 0 is the TD error; lambda = 1 is the Monte Carlo advantage
```

$\lambda = 1$ の恒等式が成り立つには、エピソードが終了する必要がある。時間制限で打ち切られたエピソードでは、`gae` が最後の TD 誤差を通じて欠けた将来をブートストラップするのに対し、`reward_to_go` は `truncation_bootstrap` で補われるまで将来を空として扱う。上の未訓練の方策は打ち切りよりずっと前にポールを倒すため、アサーションが成立している。この同じ逆方向の和にはオンライン版の TD(λ) があり、クリティックのパラメータごとに減衰する**適格度トレース**（eligibility trace）を保持することで、エピソードを保存せずにストリームから学習するエージェントに適している (Sutton 1988; Elsayed et al. 2024)。

15.1.4 λ 収益のバイアスと分散

15.1.4.1 λ のスイープ

どの λ が最もよく訓練できるだろうか。このダイヤルを動かすには、新しい重みと新しいクリティック目標が必要になる。アクターは GAE アドバンテージで重み付けし、クリティックは λ 収益、すなわちアドバンテージにその基準となった価値を加えたもの（`lambda_return` で計算する）に向かって適合させる。$\lambda = 0$ のとき、これは `td_weight` の 1 ステップ目標に一致する。15.2 節でも同じ方法でクリティックを適合させる。

```
@d2l.add_to_class(d2l.Batch)
def lambda_return(self, value_fn, gamma, lam):
    """The critic's target for GAE: the advantage plus the value it is
    measured from."""
    return self.gae(value_fn, gamma, lam) + value_fn(self.obs)
```

`gae_weight(lam)` はダイヤルの 1 つの設定に対する重み関数を返し、`gae_run` はそれを用いて 1 つのシードを訓練する。`td_weight` とは異なり、ここでは重みを標準化する。$\lambda = 1$ 付近では重みが数十の分散を持つモンテカルロアドバンテージのスケールになるため、生の重みのままだと大きな λ では上限に達し、小さな λ では達しないという不均衡が生じるからである。訓練予算は 4

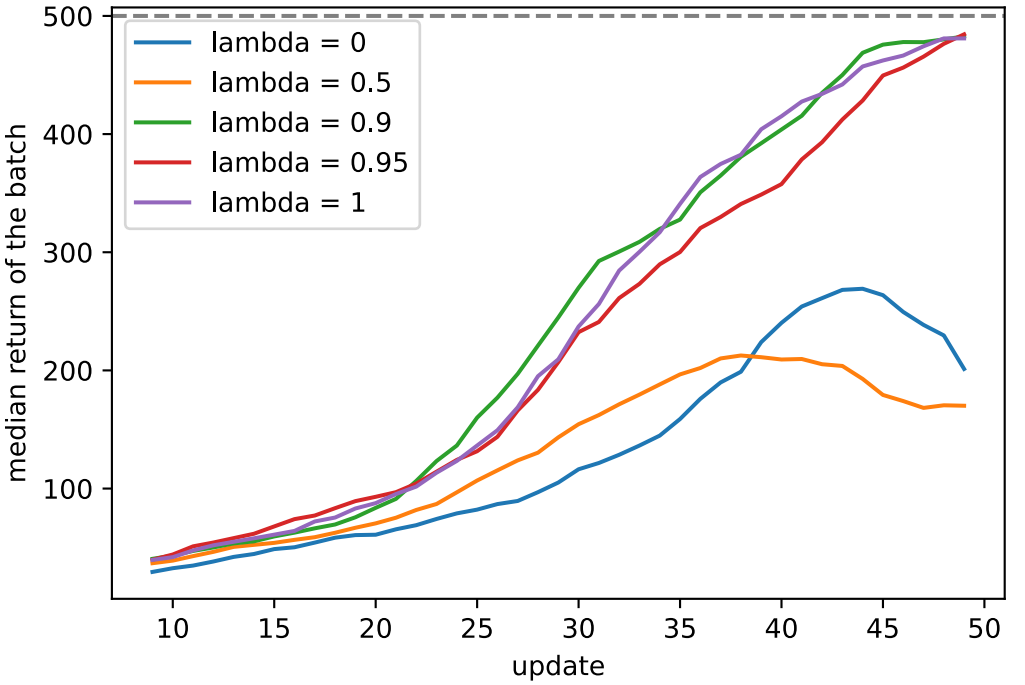
エピソードずつの更新 50 回と短く設定し、クリティックがまだ学習途上にあるエージェントをどの推定量が助けるかを比較する。

```
lams = [0.0, 0.5, 0.9, 0.95, 1.0]

def gae_weight(lam):
    """The weight of GAE(gamma, lam), standardized so that every lam steps
    alike; the critic first fits the lambda-return."""
    def weight(ac, b):
        for _ in range(critic_steps):
            d2l.fit_value(ac, b.obs, b.lambda_return(ac.value_np, gamma, lam))
        return d2l.standardize(b.gae(ac.value_np, gamma, lam))
    return weight

def gae_run(seed, lam):
    """The returns of one short run: 50 updates of four episodes."""
    return [ret for _, ret, _ in train(seed, cartpole_agent(seed),
                                       gae_weight(lam), num_updates=50,
                                       batch_episodes=4)]

sweep = {lam: d2l.run_seeds(gae_run, 5, lam=lam) for lam in lams}
```

sweep 内の各 λ について、20 更新移動平均による収益の最良値をシードごとに（シード間の中央値とともに）出力し、50 回の更新にわたるバッチ収益の中央値をプロットする。

```
for lam, r in sweep.items():
    best = np.array([np.convolve(c, np.ones(20) / 20, 'valid').max()
                     for c in r])
    print(f'lambda = {lam:4}: best 20-update window, '
          f'median {np.median(best):5.1f}, seeds {np.round(np.sort(best))}')
d2l.plot_curves({f'lambda = {lam:g}': np.median(r, axis=0)
                 for lam, r in sweep.items()},
                xlabel='update', ylabel='median return of the batch',
                smooth=10, reference=500)
```

```
lambda =  0.0: best 20-update window, median 222.5, seeds [138. 198. 223. 283. 328.]
lambda =  0.5: best 20-update window, median 203.3, seeds [175. 179. 203. 218. 321.]
lambda =  0.9: best 20-update window, median 419.5, seeds [387. 400. 420. 454. 486.]
lambda = 0.95: best 20-update window, median 406.9, seeds [327. 367. 407. 444. 482.]
lambda =  1.0: best 20-update window, median 449.7, seeds [293. 385. 450. 464. 479.]
```



シード間の中央値で見ると、$\lambda = 0$ と $\lambda = 0.5$ では 20 更新ウィンドウの最良値が約 150〜250 と上限を大きく下回るのに対し、$\lambda \geq 0.9$ ではその約 2 倍の高さに達する。個々のシードはこの境界をまたいで重なり合うため、ここでは中央値で比較している。短い訓練予算では訓練の大半の期間にわたってクリティックが不正確であり、クリティックへの依存が最も少ない推定量が最も速く学習する。これは先ほどの CartPole の比較を反対側から見たものにほかならない。すなわち CartPole では、未熟なクリティックが取り除く分散よりも、そのバイアスによる損失のほうが大きいのである。

15.1.4.2 固定方策プローブ

訓練曲線には、推定量の質だけでなく最適化や変化するデータの影響も混ざり合っている。推定量単独の性質を調べるため、途中まで訓練したエージェントを固定し、その固定されたパラメータのもとで多数のバッチを抽出して、各 λ による方策勾配の推定値を基準値と比較する。基準値には、全抽出にわたる $\lambda = 1$ の推定値の平均を用いる。エピソードが時間制限より前に終了する場合、これは環境のモデルなしに不偏なモンテカルロ勾配を近似する（打ち切られるエピソードについては演習問題 4 で扱う）。`grad_vec` は、1 バッチ分の重み付き対数確率の勾配を平坦なベクトルとして返す。

```
def grad_vec(ac, b, w):
    """The actor's gradient on batch b with weights w, as one flat vector."""
    g = torch.autograd.grad(actor_loss(ac, b.obs, b.act, w),
                            list(ac.policy.parameters()))
    return np.concatenate([x.numpy().ravel() for x in g])
```

次に、`freeze` は初期化したばかりのエージェントを `td_weight` で `warmup=40` 回のアクター・クリティック更新（通常の実行の約 4 分の 1）だけ訓練し、クリティックがまだ不正確な状態における方策勾配の推定値を調べられるようにする。

```
def freeze(seed, warmup=40):
    """An agent after `warmup` actor-critic updates: its critic still
    wrong."""
    ac = cartpole_agent(seed)
    for _ in train(seed, ac, td_weight, num_updates=warmup):
        pass
    return ac
```

probe は固定されたエージェントのもとで 4 エピソードからなるバッチを 200 個抽出し、各 λ について平均推定値のバイアス（基準値の長さに対する相対値）と単一推定値の分散（基準値の二乗長に対する相対値）を返す。バイアスの二乗と分散の和が、1 回の抽出における期待二乗誤差となる。

```
def probe(ac, seed, num_draws=200):
    """Relative bias and variance of the batch gradient for every lambda,
    against the mean of the lambda = 1 estimates."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    draws = {lam: [] for lam in lams}
    for _ in range(num_draws):
        b = d2l.rollout(env, ac.act, 4, rng)
        for lam in lams:
            draws[lam].append(grad_vec(ac, b, b.gae(ac.value_np, gamma, lam)))
    u = {lam: np.stack(d) for lam, d in draws.items()}
    u_ref = u[1.0].mean(axis=0)   # the many-rollout reference
    bias = [np.linalg.norm(u[l].mean(0) - u_ref) for l in lams]
    var = [((u[l] - u[l].mean(0)) ** 2).sum(1).mean() for l in lams]
    return (np.array(bias) / np.linalg.norm(u_ref),
            np.array(var) / (u_ref ** 2).sum())
```

1 つの固定エージェントは起こりうる多数の結果からの 1 サンプルにすぎず、あるエージェントにとって最良の λ はその個別のクリティックがどれだけ外れているかに依存する。そこで 8 個のエージェントを固定し、λ ごとにバイアス、分散、および同じエージェントの $\lambda = 1$ における誤差で割った誤差比のエージェント間中央値を出力する。

```
stats = np.array([probe(freeze(s), s) for s in range(8)])  # agent, stat, lam
err = stats[:, 0] ** 2 + stats[:, 1]
ratio = err / err[:, -1:]
for i, lam in enumerate(lams):
    print(f'lambda = {lam:4}: bias {np.median(stats[:, 0, i]):4.2f}, '
          f'variance {np.median(stats[:, 1, i]):4.2f}, '
          f'error / error at 1: {np.median(ratio[:, i]):4.2f}')
hits = np.isin(np.array(lams)[err.argmin(axis=1)], [0.9, 0.95]).sum()
print(f'{hits} of {len(stats)} agents have their smallest error at lambda = 0.9 '
      f'or 0.95')
```

```
lambda =  0.0: bias 1.19, variance 0.04, error / error at 1: 1.91
lambda =  0.5: bias 1.13, variance 0.05, error / error at 1: 1.61
lambda =  0.9: bias 0.84, variance 0.17, error / error at 1: 0.83
lambda = 0.95: bias 0.53, variance 0.32, error / error at 1: 0.61
lambda =  1.0: bias 0.00, variance 0.98, error / error at 1: 1.00
```

```
7 of 8 agents have their smallest error at lambda = 0.9 or 0.95
```

8 個の固定エージェント全体で見ると、誤差の中央値は $\lambda = 0.9$ または 0.95 で最小となり、$\lambda = 1$ における誤差のおよそ 2 分の 1 から 5 分の 3 になる。また $\lambda = 0$ における誤差よりも小さくなるが、そのマージンは実行ごとに大きくばらつく。$\lambda = 0$ から $\lambda = 1$ にかけてバイアスは 0 まで低下する一方、分散は 10 倍以上に増大し、その増大の大半は遠い将来の TD 誤差にかかる重み $(\gamma\lambda)^l$ が減衰しなくなる最後の 1 段階で生じる。個々のエージェントにはばらつきがあり、大半は 0.9 または 0.95 で最小誤差をとるが、残りのエージェントではすでに正確なクリティックに対して以下の命題が予測するとおり、より小さな λ が勝る。$\lambda = 1$ の行は自身の平均を基準値としているため、構成上バイアスが 0 となる。PPO の実装で広く使われるデフォルト値は $\lambda = 0.95$ である (Schulman et al. 2017)。

15.1.4.3 内部に最適点が生じる理由

なぜ最良の λ が両端の厳密に内側に位置するのだろうか。完全なクリティックを考えると、その対比から答えが明らかになる。

命題。クリティックが正確（$\widehat{V} = V^\pi$）であり、エピソードに時間制限がないとする。このとき任意の λ は $A^\pi(s_t, a_t)$ の不偏推定値を与え、さらに次式が成り立つ。

$$\mathbb{E}\big[(\widehat{A}_t^{\mathrm{GAE}})^2 \mid s_t, a_t\big] = \sum_{l \geq 0} (\gamma\lambda)^{2l}\, \mathbb{E}\big[\delta_{t+l}^2 \mid s_t, a_t\big],$$

したがって、推定値の誤差は $\lambda = 0$ で最小となる。

証明。クリティックが正確なとき、任意の状態における TD 誤差はその状態が与えられたもとで平均 0 となる。すなわち、前節の恒等式を行動について平均することにより $\mathbb{E}[\delta_{t+l} \mid s_{t+l}] = \sum_a \pi(a \mid s_{t+l})\, Q^\pi(s_{t+l}, a) - V^\pi(s_{t+l}) = 0$ が成り立つ。マルコフ性により、これは s_{t+l} までの全履歴（$l \geq 1$ のときは s_t、a_t、およびそれ以前のすべての TD 誤差を含む）が与えられたもとでも同様に成り立つ。ゆえに Equation 15.7 における $l \geq 1$ のすべての項は (s_t, a_t) が与えられたもとで平均 0 となり、推定値の平均は $\mathbb{E}[\delta_t \mid s_t, a_t] = A^\pi(s_t, a_t)$ となる。$k < l$ に対する交差項 $\mathbb{E}[\delta_{t+k}\, \delta_{t+l} \mid s_t, a_t]$ も同じ理由で消えるため、二乗和だけが残る。各二乗項は非負であり、その重み $(\gamma\lambda)^{2l}$ は λ とともに増大する一方、平均は変化しないため、平均二乗誤差は λ とともに増大する。■

正確なクリティックのもとでは、サンプリングされた末尾はノイズを加えるだけであり、1 ステップ TD 誤差が最良となる。したがって、内部の最適点はクリティックの誤差を映し出している。クリティックが不正確だからこそ内部に最適点が存在するのであり、クリティックが改善するにつれて最適点は $\lambda = 0$ へと移動するはずである（この点は演習問題 7 で固定方策のもとで検証する）。この意味で、$\lambda = 0.95$ というデフォルト値は、訓練中のクリティックが通常どの程度正確であるかを表す経験則といえる。

15.1.4.4 クリティックかグループベースラインか

学習されたクリティックは、アドバンテージを推定する 1 つの方法である。14.8 節ではもう 1 つの方法として、同じ初期状態からの複数のサンプルを互いに比較するグループベースラインを扱った。どちらを使うべきかは、本節で見たトレードオフから導かれる。報酬が毎ステップ得られ、エピソードが長く、状態の評価が安価な場面ではクリティックのほうが優れた推定量となる。学習に伴って誤差が縮小し、エピソードが終わる前に更新できるからである。一方、言語モデルがプロンプトに回答して検証器がその回答を採点する場合のように、各エピソードが単一の報酬で終わりサンプルが高価な場面では、グループベースラインのほうが優れた推定量となる。クリティックを使うなら各応答のあらゆるプレフィックスから最終評点を予測せねばならず、それ自体が

難しい回帰問題になるのに対し、同じプロンプトに対する他の応答を使えば 2 つ目のネットワークなしにベースラインが得られる。14.8 節で重みを導出した GRPO は、この選択をとっている。

実用されるアクター・クリティックの実装でも、エピソード全体を最後まで収集することはしない。多数の環境を並列に一定ステップ数だけ進め、エピソードが終了したかどうかにかかわらずそこで各セグメントを区切って更新する。この区切りは時間制限と同様に扱われる。すなわち、未終了セグメントの最後の TD 誤差は、`td_target` が打ち切りエピソードを処理するのと同じように区切り時点でのクリティックの価値からブートストラップし、`gae` の逆方向スキャンも各区切りでリセットされる。固定長のセグメントを用いることで、更新の間隔がエピソード長に依存しなくなり、エピソード単位のバッチでは活かされていなかったブートストラップの構造的な利点が発揮される。

以上で、単一の遷移から方策を更新し、サンプリングされた末尾をどれだけ残すかを選べるようになった。1 ステップなら TD 誤差、すべての深さの幾何重み付き混合なら λ 収益、そしてその混合を 1 回の逆方向パスで計算する GAE である。CartPole では、訓練の大部分においてクリティックが未熟であるため、末尾の大半を残す λ が 0.95 付近の選択が最良となる。もっとも、14.9 節からの制約がまだ 2 つ残っている。各バッチが 1 回のアクターステップに使われただけで破棄されることと、その 1 ステップが方策をどれだけ大きく動かすかに何の制限もないことである。15.2 節ではこの両方に取り組む。

15.1.5 まとめ

アクター・クリティック法は、サンプリングされた収益の代わりに、学習されたクリティックの TD 誤差によって方策勾配を重み付けする。正確なクリティックのもとでは TD 誤差は 1 遷移からなるアドバンテージの不偏サンプルとなるが、学習されたクリティックのもとではサンプリングされた末尾の分散をクリティックのバイアスと引き換えにし、クリティック自身の更新はサンプリングされた方策評価の半勾配ステップとなる。CartPole においてクリティックの計算予算を揃えると、未熟なクリティックが取り除く分散よりもそのバイアスの悪影響のほうが大きいため、モンテカルロ重みのほうが少ない環境ステップ数で上限に達する。ブートストラップの本質的な強みは、エピソードの終了を待たずに更新できる点にある。n ステップ収益と λ 収益はこれら 2 つの重みの間を補間し、GAE は λ 収益を TD 誤差の逆方向スキャンとして計算する。固定された方策のもとでは誤差は $\lambda = 0.9$〜0.95 付近で最小となるが、正確なクリティックがあれば $\lambda = 0$ で最小となるはずであり、内部の最適点はクリティックの誤差の大きさを表している。

15.1.6 演習問題

1. **ブートストラップされた重みのバイアス。** 14.8 節の条件付きゼロ平均恒等式によれば、現在の状態の任意の関数 $b(s_t)$ は、勾配にバイアスを生じさせることなくスコア $\psi_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ の重みから差し引くことができる。クリティックの誤差を $e(s) = \widehat{V}(s) - V^\pi(s)$ と書き、両方の価値が 0 である終端状態では $e = 0$ とする。
   1. Equation 15.2 の TD 誤差が $\delta_t = \delta_t^\pi + \gamma\, e(s_{t+1}) - e(s_t)$ と分解できることを示せ。ここで δ_t^π は正確なクリティック V^π の TD 誤差である。上の恒等式は 2 つの誤差項のうち一方を取り除く。もう一方の項については、恒等式の証明のどの段階が成り立たなくなるか。
   2. ステップ t の項 $\mathbb{E}[\delta_t \psi_t]$ が $\mathbb{E}[A^\pi(s_t, a_t)\, \psi_t]$ に対して持つバイアスは $\gamma\, \mathbb{E}[\bar{e}(s_t, a_t)\, \psi_t]$（ただし $\bar{e}(s, a) = \mathbb{E}[e(s_{t+1}) \mid s_t = s, a_t = a]$）であることを示せ。$e$ に関するどのような条件のもとでこのバイアスは消えるか。そのために $\widehat{V} = V^\pi$ は必要か。
   3. クリティックがすべての非終端状態を同じ定数 $c > 0$ だけ過大評価しているとする。一方の行動でエピソードが終了し、もう一方の行動では終了しない状態において、こ

のバイアスはどちらの行動を有利にするか。また $c < 0$ の場合はどう変わるか。毎ステップ 1 の報酬を与える CartPole では、初期化直後のクリティックは 0 付近の価値を予測する一方、V^{π} は正である。このモデルによれば、訓練初期のバイアスは方策をどの方向へ押しやると予測されるか。

2. [code] **最適なブートストラップ深さ**。本節のチェーンの例、すなわち $\sigma = 0.15$、$\gamma = 0.97$、クリティック誤差 $e(s) = 0.5\,(1 - s/9)^2$ を考える。
   1. バイアス $\gamma^n e(n)$ と分散 $\sigma^2(1-\gamma^{2n})/(1-\gamma^2)$ を、深さ n の目標（状態 0 から出発する場合）に対して導出せよ。さらにすべての深さについて 20,000 回のロールアウトをシミュレーションし、実測した平均二乗誤差を式と比較して最適点が $n = 4$ であることを確認せよ。
   2. クリティックの誤差をすべての非終端状態で定数 $e(s) = 0.5$ に置き換えよ。$n \le 9$ に対して平均二乗誤差が

$$\frac{\sigma^2}{1-\gamma^2} + \gamma^{2n}\Big(0.25 - \frac{\sigma^2}{1-\gamma^2}\Big), \tag{15.8}$$

   となり、任意の σ についてブートストラップする深さの範囲で単調であることを示せ。モンテカルロ目標 $n = 10$ を含めたときも単調なままであるか。このようなクリティックに対して最適となりうる深さはどれか、また $\sigma = 0.15$ で最適な深さはどれか。
   3. 先細りの誤差に戻し、$\sigma = 0.04$ および $\sigma = 0.3$ における最適な深さを求めよ。最適点が移動する方向を説明せよ。
3. [code] **クリティックのパス回数と更新の順序**。 `td_weight` はバッチごとに `critic_steps` 回の回帰パスをクリティックに与え、各パスの前に `td_target` を再計算する。
   1. `critic_steps` を $\{1, 5, 20\}$ としてそれぞれ 2 シードずつアクター・クリティック条件を再実行し、曲線をプロットせよ。パス回数は学習の速さと上限 500 付近での振る舞いをどのように変えるか。
   2. 代わりにバッチごとに 1 回だけ目標を計算し、`d2l.fit_value(ac, b.obs, target, num_steps=critic_steps)` で適合させよ。同じ 2 シードを 20 パスで訓練せよ。状態間の汎化を無視したとき、固定目標を用いた 1 バッチの間に、ある報酬が $\hat{V}$ に及ぼす影響はエピソードに沿って何ステップ後方まで伝わりうるか。また、各パスの前に目標を再計算する場合には高々何ステップまで伝わりうるか。これら 2 つの答えを用いて、20 パスにおける固定目標と再計算目標の曲線の違いを説明せよ。
   3. `delta` を計算する行をクリティックのパスより上に移動し、バッチ収集時点のクリティックの TD 誤差でアクターを重み付けするようにせよ。それぞれの順序で 10 シードずつ訓練し、最後の 10 バッチの平均が 300 を下回る実行の数を数えよ。1 パスの場合よりも 20 パスの場合でこの順序が重要になるのはなぜか。
4. λ **収益の漸化式**。$G_t^{\lambda} = \hat{A}_t^{\mathrm{GAE}} + \hat{V}(s_t)$ を、`gae_weight` がクリティックを適合させる目標とする。
   1. Equation 15.7 から、エピソード内で $\hat{A}_t^{\mathrm{GAE}} = \delta_t + \gamma\lambda\,\hat{A}_{t+1}^{\mathrm{GAE}}$ が成り立つことを示し、エピソードの最後ではない非終端ステップにおいて次式が成り立つことを導け。

$$G_t^{\lambda} = r_t + \gamma\big((1-\lambda)\,\hat{V}(s_{t+1}) + \lambda\,G_{t+1}^{\lambda}\big). \tag{15.9}$$

   終了するエピソードの最後のステップ、および時間制限で打ち切られるエピソードの最後のステップでは、この漸化式はそれぞれどのようになるか。
   2. エピソードが T ステップ後に打ち切られ、その最後の `next_obs` を s_T とする。このエピソードのステップ t において、`b.gae(value_fn, gamma, 1.0)` は `b.reward_to_go(gamma) - value_fn(b.obs)` をちょうど $\gamma^{T-t}\,\hat{V}(s_T)$ だけ上回ることを示せ。一般の λ について、G_t^{λ} の $\hat{V}(s_T)$ に関する導関数を計算せよ。$\lambda < 1$ のとき、打ち切り時点におけるクリティックの価値は打ち切られたエピソードのどのステップに影響するか。

3. δ_{t+l} が Equation 15.7 において持つ重みは $(\gamma\lambda)^l$ である。半減期 τ（この重みが $l = 0$ での値の半分まで低下するステップ数）を導出し、$\gamma = 0.99$ のもとで $\lambda \in \{0.5, 0.9, 0.95, 1\}$ に対してその値を求めよ。固定方策プローブが出力する分散は $\lambda = 0.95$ と $\lambda = 1$ の間で最も大きく増大する。これを半減期と関連づけて説明せよ。

Richard S. Sutton および Andrew G. Barto 著『Reinforcement Learning: An Introduction』の演習問題 12.1 および 12.2 を改変。

5. [extended] **完全オンライン更新。**●Equation 15.4 の更新は、バッチを一切使わずに遷移のたびに適用できる。すなわち、終了時にマスクされた単一の目標 $r_t + \gamma \hat{V}(s_{t+1})$ に向かうクリティックの 1 ステップと、`td_weight` と同様に δ_t そのもので重み付けされたアクターの 1 ステップを実行する。
   1. 長さ 1 の配列に `d2l.fit_value` と `policy_step_clip` を適用してこの変種を実装せよ。`train` によるアクター・クリティック 1 回分の環境ステップ数を数え、オンライン版にも同じステップ数を与えよ。`ActorCritic.mlp` のデフォルトの学習率と、その 10 分の 1 の学習率（`lr` として `cartpole_agent` 内に渡す）とでそれぞれ 3 シードずつ訓練し、バッチ化されたループの曲線と比較せよ。
   2. このとき連続する更新は 1 つのエピソードの連続する状態から生じる。何が問題になるか。リプレイバッファは過去の遷移を保存し、そこから一様に抽出したミニバッチで訓練する。これはそれらの失敗のうちどれに対処できるか。また、アクターのスコア関数による更新を再生された遷移上でそのまま訓練できないのはなぜか。
   3. 並列環境を用いれば、リプレイバッファなしにデータの相関を解消できる (Mnih et al. 2016)。8 個の CartPole 環境を同期して 1 ステップずつ進め、各環境から 1 つずつ集めた 8 遷移のスライスごとに更新せよ。これは先ほどの失敗のうちどれを解消し、どれを残すか。
6. [code] **勾配ノルムと上限。**●`policy_step_clip` が返すノルムは $\hat{\mathbf{g}} = \frac{1}{N}\sum_t \hat{A}_t \psi_t$ のノルムである。ここで $\hat{A}_t$ はアクターの重み（`mc_weight` の正規化されたモンテカルロアドバンテージ、および `td_weight` の生の TD 誤差）であり、$\psi_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ である。`runs` の最後の列には各更新におけるこのノルムが記録されている。
   1. `runs` 内の 2 条件についてノルムの中央値を比較せよ。N 個の項を独立同分布として扱い、次式を示せ。

$$\mathbb{E}[\|\hat{\mathbf{g}}\|^2] = \left\|\mathbb{E}[\hat{A}_t \psi_t]\right\|^2 + \frac{1}{N}\Big(\mathbb{E}[\|\hat{A}_t \psi_t\|^2] - \left\|\mathbb{E}[\hat{A}_t \psi_t]\right\|^2\Big). \tag{15.10}$$

   重みをシグナル $\mathbb{E}[\hat{A}_t \mid s_t, a_t] - \mathbb{E}[\hat{A}_t \mid s_t]$ と残余とに分解し、$\mathbb{E}[\hat{A}_t \psi_t]$ に寄与するのはシグナルのみであることを示せ。単位分散の重みにおいて残余が大きいと、ノルムはどうなるか。
   2. エージェント `freeze(0)` において、`d2l.rollout` で 8 エピソードからなるバッチを 50 個抽出せよ。各バッチについて、`grad_vec(ac, b, d2l.standardize(w))` のノルムを TD 誤差とモンテカルロアドバンテージ `b.reward_to_go(gamma) - ac.value_np(b.obs)` のそれぞれに対して計算し、さらにバッチ内で各重みベクトルをランダムに並べ替えた後にもう一度計算せよ。両方の重みを正規化することで、`td_weight` の生の TD 誤差が持つスケールの違いは取り除かれる。ランダムな並べ替えは何を取り除くか。各推定量のノルムは、並べ替え後のノルムと比べてどうなっているか。
   3. アクター・クリティック条件を変更し、バッチごとに `delta` の標準偏差（`td_weight` 内）と、1 ステップによるアクターのパラメータ変化量のノルム（`train` 内）を記録せよ。元のコードのまま、`clip=float('inf')` とした場合、重みを `d2l.standardize(delta)` とした場合の 3 つの規則のもとで 10 シードずつ訓練せよ。各規則のもとで最初の 10 回と最後の 50 回の更新におけるパラメータ変化量を比較し、移動平均が 475 に達した後に 300 を下回る実行の数を数えよ。記録した標準偏差と Adam の座標ごとのスケー

リングを用いて、クリティックのノイズへと縮小していく重みを正規化すると訓練終盤で悪影響が出うる理由、および Adam のもとで上限が何を変えるかを説明せよ。

7. [code] **GAE のバイアス。**●演習問題 1 では、重み δ_t のバイアスをクリティックの誤差 $e = \hat{V} - V^{\pi}$ を用いて求めた。本問ではこの結果を任意の λ へ拡張する。終了するエピソードを考え、終了時およびそれ以降では $e = 0$ とする。
   1. 次式が成り立つことを示せ。

$$\hat{A}_t^{\mathrm{GAE}} = \sum_{l \geq 0} (\gamma\lambda)^l \, \delta_{t+l}^{\pi} - e(s_t) + (1-\lambda) \sum_{k \geq 1} \gamma^k \lambda^{k-1} \, e(s_{t+k}), \tag{15.11}$$

      ここで δ^{π} は V^{π} の TD 誤差である。内部の最適点に関する命題とあわせて、勾配推定値のステップ t の項のバイアスが $\mathbf{b}(\lambda) = (1-\lambda) \sum_{k \geq 1} \gamma^k \lambda^{k-1} \, \mathbf{c}_k$（ただし $\mathbf{c}_k = \mathbb{E}[e(s_{t+k}) \, \psi_t]$）となることを導け。
   2. $\mathbf{c}_k = q^{k-1} \mathbf{c}_1$（固定ベクトル $\mathbf{c}_1$ およびある $0 \leq q \leq 1$ に対して）と仮定する。すなわち、後の状態におけるクリティック誤差に対する行動の影響が幾何級数的に減衰するものとする。$\|\mathbf{b}(\lambda)\| / \|\mathbf{b}(0)\| = (1-\lambda)/(1-\gamma\lambda q)$ を示し、`lams` 内の各値に対して $\gamma = 0.99$ のもとで $q = 1$ と $q = 0.9$ の場合にこの比を計算せよ。固定方策プローブの 8 個のエージェントそれぞれについて、各 λ におけるバイアスを $\lambda = 0$ におけるバイアスで割り、予測された比と比較せよ。どちらの q がよく当てはまるか。もし比が 1 を超えるなら、それはベクトル列 $\mathbf{c}_k$ について何を意味するか。
   3. `freeze(0)` の方策を固定したままクリティックのみを適合させ続けよ。すなわち、アクターのステップを行わずにクリティックのパスのみをさらに 50 バッチ実行し、プローブを繰り返せ。固定された方策のもとでクリティックをより長く適合させるにつれて、相対バイアスおよび 1 回抽出誤差を最小にする λ はどのように移動するか。実測された最適点は、正確なクリティックに対して命題が予測する $\lambda = 0$ に向かって移動するか。

15.2 信頼領域と近接方策最適化

14.9 節と 15.1 節のエージェントは、8 エピソードを収集して 1 回の勾配ステップを実行すると、そのバッチを破棄していた。エージェントがポールをうまくバランスさせられるようになると、この 8 エピソードは 4,000 ステップ分の経験に相当するが、そのすべてがたった 1 回の更新にしか使われない。しかも、そのステップには歯止めがない。大きな更新によって方策が大きく変化してしまうのを防ぐ仕組みはなく、次のバッチはその更新が生み出した方策でそのまま収集されることになる。1 つのバッチに対して複数回のステップを実行することはできるだろうか。また、どこまで更新を進めてよいのだろうか。

その答えは、パラメータに関するある事実から始まる。パラメータに同じだけの変化を加えても、出発点がどこにあるかによって、方策の振る舞いがほとんど変わらないこともあれば、一変してしまうこともある。つまり、パラメータ空間におけるステップサイズは、本来重要な量である振る舞いの変化を制御しない。2015 年、John Schulman らは、まさにその振る舞いの変化を制御対象に据えた。信頼領域方策最適化（trust region policy optimization、TRPO）は、そのようなステップがどれだけの改善を保証するかを示す定理に導かれ、訪問した状態における新旧の行動分布間のダイバージェンスを抑える (Schulman et al. 2015)。その 2 年後、近接方策最適化（proximal policy optimization、PPO）は、この制約を任意の勾配オプティマイザで実行できるクリップ付き目的関数に置き換えた (Schulman et al. 2017)。PPO は、シミュレーション上のロボットから人間のフィードバックに基づく言語モデルのファインチューニングに至るまで、標準的な方策勾配法となった (Ouyang et al. 2022)。

本節ではこの両方を構築する。まず 2 行動の方策を用いて、なぜパラメータ間の距離が不適切な

量なのかを示す。次に測度変換（change of measure）によって、古い方策で収集したバッチから新しい方策を評価できるようにする。この評価は厳密であるものの、エピソード長に伴って分散が増大するため、ステップごとに 1 つの比だけを残すことで再利用を現実的なものにする。続いて短い補題により、その近似が何を落としているかを明らかにする。TRPO はその誤差を抑え、PPO はそれをクリップする。最後に CartPole 上で、バッチの再利用によって環境ステップ数をどれだけ節約できるか、クリップがないと何が起きるか、そして方策の訓練中にどの数値を監視すべきかを確かめる。

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

`torch.set_num_threads(1)` は、これらのネットワークにおける小さな行列積を PyTorch が多数のコアに分散させるのを防ぐ。分散させるとかえって処理が遅くなるためである。

15.2.1 パラメータ空間と方策空間

更新するのは方策のパラメータであるから、パラメータ更新が方策そのものに何をもたらすかを問うべきである。単一のパラメータ θ と 2 つの行動を持つ方策族がその答えを与えてくれる。この例は、方策最適化に関する Joshua Achiam の講義に由来する (Achiam 2017)。次の方策を考えよう。

$$\pi_\theta(a) = \begin{cases} \sigma(\theta) & a = 1 \\ 1 - \sigma(\theta) & a = 2, \end{cases} \qquad \text{where } \sigma(\theta) = \frac{1}{1 + e^{-\theta}}. \tag{15.12}$$

異なる 2 つの出発点から、同じ大きさ $\Delta\theta = 2$ のパラメータ更新を 2 つ考えてみよう。

$$\pi_{\theta=0}(a{=}1) = \sigma(0) = 0.50 \xrightarrow{\Delta\theta=2} \sigma(2) = 0.88,$$
$$\pi_{\theta=6}(a{=}1) = \sigma(6) = 0.9975 \xrightarrow{\Delta\theta=2} \sigma(8) = 0.9997.$$

最初の更新は、エージェントを 2 つの行動に対して無差別な状態から、一方を強く選好する状態へと変化させる。2 つ目の更新では、エージェントの観察者が検出できるような変化は何も生じない。パラメータから方策への写像の傾きを見ればその理由は明らかである。$\sigma'(\theta) = \sigma(\theta)(1 - \sigma(\theta))$ は、$\theta = 0$ では 0.25 であるのに対し、$\theta = 6$ では約 0.0025 にすぎない。Figure 15.3 に両方の更新を示す。

この違いにより、オンポリシー訓練は不安定になる。$\theta = 0$ 付近でノイズの大きい大幅な更新が生じると、方策は飽和領域へと押し込まれうる。そこでは頻繁に選ばれる行動のスコアがほぼゼロになり（$\nabla_\theta \log \pi_\theta(a{=}1) = 1 - \sigma(\theta) \approx 0.0025$、$\theta = 6$ のとき）、もう一方の行動は -1 近くのスコアを持つものの、400 ステップに 1 回しかサンプリングされない。その結果、以降のオンポリシーのバッチには、この更新を覆すだけの材料がほとんど含まれなくなる。パラメータステップのユークリッド長を制限しても、これを防ぐことはできない。同じ長さであっても、$\theta = 6$ では無害である一方、$\theta = 0$ では劇的な変化をもたらすからである。

したがって、更新ではパラメータ間の距離ではなく方策間の距離を制限すべきである。そのような制限のもとでは、オプティマイザは方策がほとんど変化しない領域で長いパラメータステップを取ることができ、さらに以下で見るように、同じバッチに対して複数回のステップを実行できるようになる。

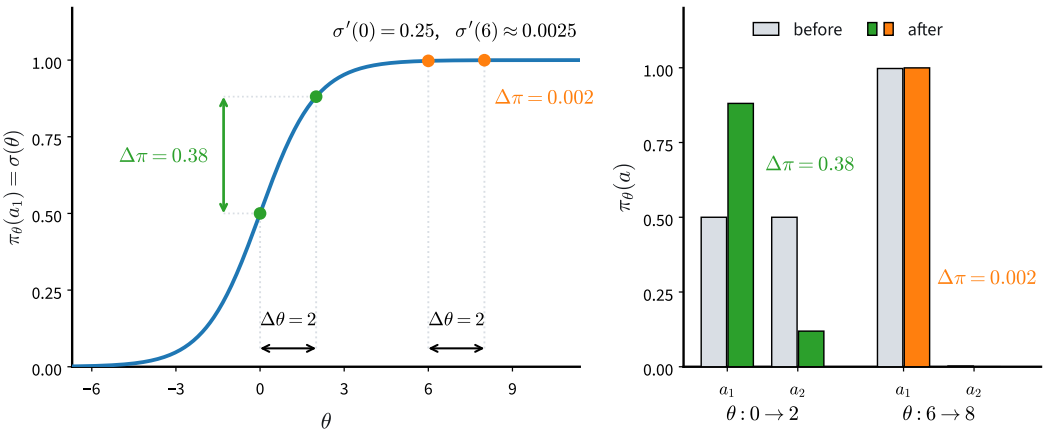


Figure 15.3: 同じ大きさ $\Delta\theta = 2$ の 2 つのパラメータ更新。左：両方の更新および各出発点における傾きを添えた写像 $\pi_\theta(a{=}1) = \sigma(\theta)$。右：各更新の前後における 2 つの行動分布。

15.2.2 重要度サンプリング

15.2.2.1 測度変換

14.7 節以降に登場したすべての推定量は現在の方策からサンプリングされた軌跡を用いていたため、バッチは 1 回の勾配ステップの後には使えなくなっていた。では代わりに、バッチが以前の方策 $\pi_{\theta_{\text{old}}}$ によって収集されたものだとしたらどうだろうか。そのバッチからでも、更新後の方策 π_θ がどれだけうまく振る舞うかを知ることができるだろうか。

重要度サンプリング（importance sampling）がその厳密な答えを与える。軌跡の任意の関数 f に対して、次が成り立つ。

$$\begin{aligned}\mathbb{E}_{\tau\sim P(\cdot;\theta)}\big[f(\tau)\big] &= \sum_\tau P(\tau;\theta)\, f(\tau) = \sum_\tau P(\tau;\theta_{\text{old}})\, \frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, f(\tau) \\ &= \mathbb{E}_{\tau\sim P(\cdot;\theta_{\text{old}})}\Big[\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, f(\tau)\Big],\end{aligned} \tag{15.13}$$

この等式は、新しい方策が生成しうるすべての軌跡が古い方策のもとで正の確率を持つ限り成り立つ。ソフトマックス方策はどの行動にも確率ゼロを割り当てないため、この条件を満たす。収益を表す $f = R$ に対してこの恒等式を適用すると、目的関数は**古い**方策のもとでの期待値として書き直せる。

$$J(\theta) = \mathbb{E}_{\tau\sim P(\cdot;\theta)}\big[R(\tau)\big] = \mathbb{E}_{\tau\sim P(\cdot;\theta_{\text{old}})}\Big[\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, R(\tau)\Big]. \tag{15.14}$$

このように、古い方策から得られたデータの各軌跡にその尤度比（likelihood ratio）で重み付けすれば、新しい方策の収益を統計的バイアスなしに推定できる。14.7 節で見たとおり、軌跡の確率は開始状態の確率、方策の行動確率、および環境の遷移確率の積である。この比では方策以外の因子がすべて約分されるため、重みの計算に環境のモデルは不要である。

$$\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})} = \prod_{t=0}^{T-1} \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}. \tag{15.15}$$

15.2.2.2 爆発する積

この推定量は不偏であるが、その分散はエピソードの長さに伴って増大する。状態を無視し、$\pi_{\theta_{\text{old}}} = (0.5, 0.5)$ でデータを収集して $\pi_\theta = (0.6, 0.4)$ で評価される方策を考えよう。Equation 15.15 の各因子は等確率で 1.2 または 0.8 を取るため、その平均は 1、二乗平均は $(1.44 + 0.64)/2 = 1.04$ である。異なるステップの因子は互いに独立であるため、軌跡の重み W_T（ステップ数 T のエピソードに対する重み）は平均 1 を持ち、その分散は次のようになる。

$$\text{Var}[W_T] = 1.04^T - 1,$$

この値は、$T = 10$ では約 0.5、$T = 100$ では約 50、CartPole の 500 ステップでは約 3×10^8 に達する。したがって、1 回の意思決定あたりデータ収集時の方策と 10 パーセントポイント異なる方策を、CartPole のエピソード全体から評価することはできない。大きな重みを持つごく少数の軌跡が推定値を決めてしまうからである。

15.2.2.3 ステップごとの代理目的関数

ステップ t における報酬はステップ t までに取った行動のみに依存するため、その報酬に対する厳密な補正に必要なのは比の積 $\rho_0 \cdots \rho_t$ だけである。ここで、各比は次で定義される。

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}. \tag{15.16}$$

それ以前の比の積 $\rho_0 \cdots \rho_{t-1}$ は状態に対する補正であり、s_t の分布を古い方策のものから新しい方策のものへと変換する。これを落とし、s_t における行動に対する ρ_t だけを残せば、分散が t に伴って増大しない重みが得られる。このたった 1 つの近似により、古い方策の状態において新しい方策の行動を評価することになる。軌跡 i のステップ t におけるアドバンテージ推定値 $\hat{A}_t^i$ を用いると、得られる**代理目的関数**（surrogate objective）は、n 本の軌跡からなるバッチの全 $N = \sum_i T_i$ ステップにわたる平均となる。

$$\hat{L}(\theta) = \frac{1}{N} \sum_{i=1}^{n} \sum_{t=0}^{T_i - 1} \rho_t^i(\theta)\, \hat{A}_t^i. \tag{15.17}$$

コードで実際に計算するのは軌跡ごとではなくステップごとの平均であり、これによって以下で導入するダイバージェンス制約も状態にわたる平均となる。$\theta = \theta_{\text{old}}$ ではすべての比が 1 に等しく、$\nabla_\theta \rho_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ が成り立つため、そこでの $\hat{L}$ の勾配は 14.8 節の推定量と同じ方向を向く。すなわち、それ以降のすべての実装と同様に因子 γ^t を落としたうえで、各スコアをそのアドバンテージで重み付けしたものである。アドバンテージを平均 0 に標準化すれば、厳密に $\hat{L}(\theta_{\text{old}}) = 0$ となる。θ_{old} から離れるにつれて代理目的関数の精度は低下するため、次の問いはそれをどこまで信頼できるかである。

15.2.3 信頼領域

15.2.3.1 性能差補題

代理目的関数は、古い方策の状態において新しい方策の行動を評価する。この置き換えによって何が変わるかを確かめるため、2 つの収益を厳密に比較しよう。古い方策のアドバンテージを $A^{\text{old}}(s,a) = Q^{\pi_{\theta_{\text{old}}}}(s,a) - V^{\pi_{\theta_{\text{old}}}}(s)$ と書く。これは、その後に古い方策に従って行動するとき、行動 a が状態 s における古い方策の平均よりどれだけ優れているかを表す。次の補題は、終端状態（terminal state）で終了するエピソード、および $\gamma < 1$ の継続タスクに対して成り立つ。本節の CartPole の実行では、14.9 節と同様に 500 ステップの時間制限でブートストラップするため、最適化する目的関数は継続タスクのものとなる。

命題（性能差補題、Kakade and Langford (2002)）。

$$J(\theta) - J(\theta_{\text{old}}) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t \ge 0} \gamma^t \, A^{\text{old}}(s_t, a_t)\Big]. \tag{15.18}$$

証明。V で $V^{\pi_{\theta_{\text{old}}}}$ を表すことにする。任意の軌跡に沿った和 $\sum_t \gamma^t (r_t + \gamma V(s_{t+1}) - V(s_t))$ は望遠鏡和となる。途中の V の項は対になって打ち消し合い、最後の項は終端状態で 0 になるか、$t \to \infty$（$\gamma < 1$ のとき）で消失する。残るのは $R(\tau) - V(s_0)$ である。ここで $\tau \sim P(\cdot;\theta)$ のもとで期待値を取る。(s_t, a_t) が与えられたとき、次の状態に関する $r_t + \gamma V(s_{t+1})$ の期待値は、古い方策のベルマン方程式により $Q^{\pi_{\theta_{\text{old}}}}(s_t,a_t)$ となるため、ステップ t の項は $\gamma^t \, \mathbb{E}\big[A^{\text{old}}(s_t,a_t)\big]$ となる。一方、$\mathbb{E}[R(\tau)] = J(\theta)$ であり、さらに $\mathbb{E}[V(s_0)] = J(\theta_{\text{old}})$ も成り立つ。両方策とも s_0 を同じ初期分布から引くからである。■

この補題は、新しい方策による改善量が、新しい方策自身の軌跡に沿って古い方策のアドバンテージを足し合わせたものに等しいことを示している。この和の項を状態ごとにまとめると、代理目的関数との比較が厳密になる。$d^{\pi}_{\gamma}(s) = (1-\gamma)\sum_{t \ge 0} \gamma^t \, \Pr(s_t = s)$ を 14.7 節の割引状態占有分布（discounted state-occupancy distribution）、すなわちステップ t に γ^t の重みを付けたもとで方策が訪問する状態の分布とする。このとき、次が成り立つ。

$$J(\theta) - J(\theta_{\text{old}}) = \frac{1}{1-\gamma} \sum_{s} d^{\pi_\theta}_{\gamma}(s) \sum_{a} \pi_\theta(a \mid s) \, A^{\text{old}}(s,a).$$

代理目的関数は、この和を新しい方策の状態占有分布ではなく古い方策の状態占有分布 $d^{\pi_{\theta_{\text{old}}}}_{\gamma}$ を用いて計算しており、この置き換えが唯一の誤差である。この誤差は状態分布がどれだけ動くかに依存し、それはパラメータがどれだけ動くかではなく、方策がどれだけ変化するかに依存する。

15.2.3.2 単調改善下界

直前の式の状態占有分布を置き換えることで、母集団版の代理目的関数が次のように定義される。

$$\bar{L}(\theta) = \frac{1}{1-\gamma} \sum_{s} d^{\pi_{\theta_{\text{old}}}}_{\gamma}(s) \sum_{a} \pi_\theta(a \mid s) \, A^{\text{old}}(s,a), \tag{15.19}$$

$\hat{L}$ は、この式から因子 γ^t を除いた標本推定量にあたる。古い方策のアドバンテージはどの状態でも平均すると 0 になるため、$\bar{L}(\theta_{\text{old}}) = 0$ である。では、θ がどこまで動くと $\bar{L}$ は改善量を予測できなくなるのだろうか。その答えは、あらゆる状態における行動分布の最大の変化量を用いた下界として与えられる。

命題（Kakade and Langford (2002); Schulman et al. (2015)）。$\gamma < 1$ **とし、**$A_{\max} = \max_{s,a} |A^{\text{old}}(s,a)|$ **とおくと、次が成り立つ。**

$$\begin{aligned} J(\theta) \;\geq\; & J(\theta_{\text{old}}) + \bar{L}(\theta) \\ & - \frac{4\gamma A_{\max}}{(1-\gamma)^2} \max_{s} D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s) \,\|\, \pi_{\theta}(\cdot \mid s)\big). \end{aligned} \tag{15.20}$$

代理目的関数は、あらゆる状態における新旧の行動分布間のカルバック・ライブラー・ダイバージェンスの最大値に比例するペナルティの範囲内で正しい。演習問題 6 では、行動をカップリングして 2 つの方策を並走させることでこの下界を導出する。$\theta = \theta_{\text{old}}$ において下界の両辺はともに $J(\theta_{\text{old}})$ に等しいため、右辺を引き上げるステップは真の収益も必ず引き上げる。すなわち、この下界を最大化すれば単調改善が得られる。ただし、この定数はそのまま使うには大きすぎる。$\gamma = 0.99$ では $A_{\max}$ の $4 \cdot 0.99 / 0.01^2 \approx 40{,}000$ 倍にもなるため、この下界が保証できるステップはごく小さなものにとどまる。

TRPO はこの下界の形状を保ちつつ、その構成要素をバッチから推定できる量に置き換える。すなわち、バッチ内の状態にわたる平均ダイバージェンスを予算 Δ_{KL} 以下に抑えるという制約のもとで、標本代理目的関数を最大化する。

$$\begin{aligned} & \max_{\theta} \; \hat{L}(\theta) \\ & \text{subject to } \frac{1}{N} \sum_{i,t} D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s_t^i) \,\|\, \pi_{\theta}(\cdot \mid s_t^i)\big) \leq \Delta_{\text{KL}}, \end{aligned} \tag{15.21}$$

ここでは、全状態にわたる最大値の代わりに平均を用い、保守的な係数の代わりに選んだ予算（典型的には $\Delta_{\text{KL}} = 0.01$）を用いている。TRPO はこの問題を二次手法で解くが、これについては演習問題 7 で本節の構成要素から構築する。Figure 15.4 に、なぜこの制約を方策空間に置く必要があるかを示す。

15.2.3.3 クリップ付き目的関数

PPO は、制約付きの二次最適化問題を、上限が組み込まれた一次の目的関数に置き換える。PPO は比そのものをクリップするのではなく、比が 1 の周りの半幅 ϵ_{clip} のバンドを超えた時点で、そのサンプルからそれ以上の利得を与えないようにする。

$$\begin{aligned} L^{\text{CLIP}}(\theta) = \frac{1}{N} \sum_{i,t} \min \Big(& \rho_t^i(\theta)\, \hat{A}_t^i, \\ & \text{clip}\big(\rho_t^i(\theta),\, 1-\epsilon_{\text{clip}},\, 1+\epsilon_{\text{clip}}\big)\, \hat{A}_t^i \Big). \end{aligned} \tag{15.22}$$

1 つのサンプルについて考えよう。$\hat{A}_t > 0$ の場合、その寄与は ρ_t とともに増加するが、それは $\rho_t = 1 + \epsilon_{\text{clip}}$ までであり、そこを超えるとクリップされた項のほうが小さくなるため勾配は 0 になる。$\hat{A}_t < 0$ の場合のしきい値は $1 - \epsilon_{\text{clip}}$ である。通常の $\epsilon_{\text{clip}} = 0.2$ を用い、古い方策が確率 0.5 で選んだ行動を考えると、正のアドバンテージを持つサンプルは新しい方策がその行動に確率 0.6 を与えた時点で更新への上積みをやめ、負のアドバンテージを持つサンプルは確率が 0.4 まで下がった時点で上積みをやめる。最小値を取ることで、この制限は片側のみに働く。すなわち、目的関数を悪化させる方向の変化は常に反映される一方、望ましい方向へのそれ以上の移動は何も加えない。Figure 15.5 に、アドバンテージの正負両方について 1 サンプルの寄与を示す。

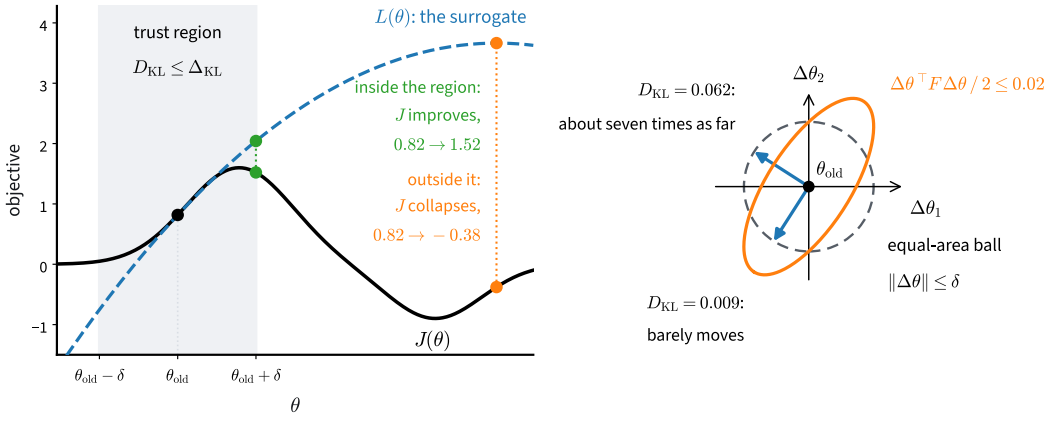


Figure 15.4: 方策更新の制限。左：代理目的関数 L は真の目的関数 J に θ_{old} で接しているが、方策が大きく変化すると不正確になる。L の制約なし最大点は J を 0.82 から -0.38 へと下げてしまうのに対し、網掛けの信頼領域 $D_{\mathrm{KL}} \le \Delta_{\mathrm{KL}}$（1 パラメータの場合は区間 $\theta_{\mathrm{old}} \pm \delta$）内の最良点は 1.52 へと引き上げる。右：3 行動のソフトマックス方策において、ダイバージェンス制約の二次近似 $\frac{1}{2}\Delta\theta^\top F \Delta\theta \le 0.02$（$F$ は θ_{old} におけるダイバージェンスのヘッセ行列）はパラメータ空間内の楕円となる。ユークリッド長が等しい 2 つのパラメータステップが、それぞれ 0.009 と 0.062 のダイバージェンスだけ方策を変化させており、同じパラメータ距離でも一方の方向では他方の約 7 倍大きく方策が動く。

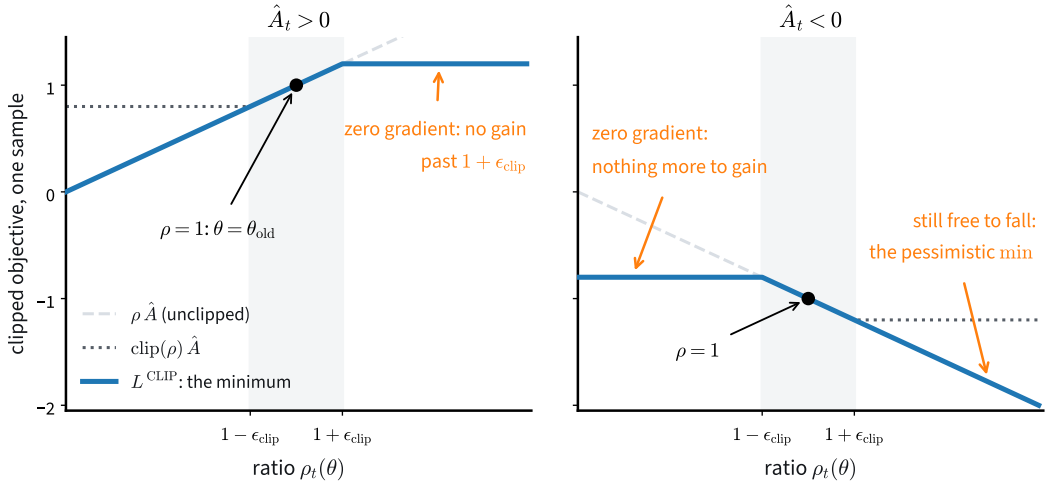


Figure 15.5: クリップ付き目的関数 Equation 15.22 に対する 1 サンプルの寄与を比 ρ_t に対してプロットしたもの。正のアドバンテージでは $1 + \epsilon_{\mathrm{clip}}$ を超えると増加が止まり、負のアドバンテージでは $1 - \epsilon_{\mathrm{clip}}$ を下回ると増加が止まる。1 を挟んだ反対側では最小値演算によりクリップなしの項が残るため、目的関数を下げる変化が隠されることはない。

比そのものがクリップされるわけではないため、2 つの理由から比は依然としてバンドの外へ出ることがある。第 1 に、すべてのサンプルがネットワークのパラメータを共有しているため、あるサンプルのためのステップが他のサンプルの比も動かしてしまうこと、第 2 に、あるサンプルの勾配が 0 になった後も Adam のモメンタムが数ステップにわたってパラメータを動かし続けることである。Engstrom et al. (2020) は、PPO の比が実際にバンドの外へ出ること、また更新あたりのダイバージェンスがいかなる一定の予算以下にも抑えられていないことを実測した。したがって、クリップは制約というより誘因の設計であり、Equation 15.20 とは異なり改善を保証するものではない。それでも、個々の行動確率の際限ない変化を促すことなく、1 つのバッチに対して複数パスの最適化を実行することを可能にする。

15.2.3.4 エントロピーボーナス

多くの実装では、バッチ内の状態における行動分布の平均エントロピーに重み `entropy_coef` を掛けた小さな**エントロピーボーナス**を加える。このボーナスは、2 行動の例で見たように頻繁に選ばれる行動のスコアが小さくなって回復が遅れる 0 や 1 の付近から、確率を遠ざける働きをする。K 個の行動上の分布のエントロピーは $\log K$ から一様分布とのダイバージェンスを引いたものに等しいため、このボーナスは一様方策からの距離に対するペナルティとなっている。

15.2.4 PPO の実践

15.2.4.1 アドバンテージ推定

Equation 15.22 の $\hat{A}$ には、どのようなアドバンテージ推定を用いてもよい。ここでは 15.1 節の一般化アドバンテージ推定（generalized advantage estimation、GAE）を用いる。`Batch.gae` は、クリティックの TD 誤差 $\delta_t = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)$ を重み $(\gamma\lambda)^k$ で各エピソードを逆向きに足し合わせるものであり、λ が各アドバンテージのどれだけ先まで見通すかを定める。同節では、この推定の誤差は $\lambda = 0.9$ から 0.95 付近で最小になることが示されたが、0.95 は PPO の原論文の設定でもある。アドバンテージは、クリティックをそのバッチに当てはめる前の時点のままで計算する。これは GAE の原論文の順序に従ったものである (Schulman et al. 2016)。15.1 節の `td_weight` では逆の順序を用いたが、それはアクターの重みがクリティックに全面的に依存する 1 ステップ TD 誤差だったからである。$\lambda = 0.95$ では、アドバンテージの大半はバッチ自身のサンプリングされた報酬からなり、クリティックへの依存は小さいため、この順序によって本節の結果が変わることはなく、原論文に従うことにする。

15.2.4.2 実装

CartPole の実験設定は 15.1 節のものを引き継ぐ。エージェントは 14.9 節の `ActorCritic.mlp(4, 2)` であり、それぞれ 64 個の tanh ユニットからなる 1 つの隠れ層と、学習率 10^{-2} の独自の Adam オプティマイザを持つ方策ネットワークと価値ネットワークで構成される。各更新では 8 エピソードからなるバッチを収集し、クリティックはその上で `critic_steps = 20` 回の回帰パスを実行する。新しく加わるハイパーパラメータは、各バッチに対するアクターのパス数 `num_epochs = 20`、バンドの半幅 `epsilon_clip = 0.2`、およびエントロピー重み `entropy_coef = 0.01` の 3 つである。20 パスという設定は RL Baselines3 Zoo における CartPole の設定でもあるが (Raffin et al. 2021)、そちらの学習率は本節の 10 分の 1 にあたる 10^{-3} である。本節の学習率では 1 パスごとに方策が大きく動くため、訓練が高速になる一方でクリップが不可欠となる設定になっており、クリップを外すとどうなるかは後のアブレーション実験で確かめる。

```
gamma, lam, num_updates, batch_episodes = 0.99, 0.95, 60, 8
num_seeds, num_epochs, critic_steps = 8, 20, 20
epsilon_clip, entropy_coef = 0.2, 0.01
```

```
def cartpole_agent(seed):
    return d2l.ActorCritic.mlp(4, 2, seed=seed)
```

サンプルごとの目的関数は、更新処理に関数として渡す。clipped_surrogate は Equation 15.22 であり、項 $\rho_t \hat{A}_t$ のみからなる plain_surrogate は後のアブレーション実験で用いる。いずれも引数としてバンドの半幅 ϵ_{clip} を受け取るが、実際にそれを使うのは前者のみである。

```
def clipped_surrogate(rho, adv, epsilon):
    """PPO's per-sample objective eq_ppo_clip: min(rho A, clip(rho) A)."""
    return torch.minimum(rho * adv,
                         rho.clamp(1 - epsilon, 1 + epsilon) * adv)

def plain_surrogate(rho, adv, epsilon):
    """The unclipped per-sample objective rho A."""
    return rho * adv
```

ppo_epochs は、バッチの状態と行動に加え、再利用の前に固定した 2 つの量、すなわちアドバンテージと収集方策の対数確率 logp_old を受け取る。そして、符号を反転した平均代理目的関数からエントロピーボーナスを引いた損失に対して、num_epochs ステップだけアクターのオプティマイザ opt_pi を進める。エントロピーの計算には各状態における分布全体が必要になるため、更新処理では全行動にわたる対数ソフトマックスを一度計算し、そこから $\log \pi_\theta(a_t \mid s_t)$ を読み取る。各パスの前には診断指標を 1 行記録する。その内容は、バンド外に出た比の割合、$\log \pi_{\theta_{\text{old}}} - \log \pi_\theta$ の標本平均、および平均エントロピーである。2 つ目の数値は**近似 KL**（approximate KL）と呼ばれ、古い方策のもとでの期待値は訪問した状態におけるダイバージェンス $D_{\text{KL}}(\pi_{\theta_{\text{old}}} \| \pi_\theta)$ に一致する（もっとも、有限のバッチ上では負の値になることもある）。この行はパスの前に記録されるため、最初の行は常に 0、0、そして古い方策のエントロピーとなる。

```
def ppo_epochs(ac, obs, act, adv, logp_old, epsilon, num_epochs,
               entropy_coef=0.01, surrogate=clipped_surrogate):
    """num_epochs passes on the mean surrogate plus an entropy bonus, on one
    frozen batch of states and actions; returns [num_epochs, 3] numpy
    diagnostics, recorded before each pass: fraction of ratios outside the
    band, approximate KL, mean policy entropy."""
    obs, act, adv, logp_old = (torch.as_tensor(x)
                               for x in (obs, act, adv, logp_old))
    diag = []
    for _ in range(num_epochs):
        logp_all = torch.log_softmax(ac.policy(obs), dim=-1)
        logp = logp_all.gather(-1, act[:, None]).squeeze(-1)
        rho = torch.exp(logp - logp_old)
        entropy = -(logp_all.exp() * logp_all).sum(-1).mean()
        loss = -surrogate(rho, adv, epsilon).mean() - entropy_coef * entropy
        ac.opt_pi.zero_grad()
        loss.backward()
        ac.opt_pi.step()
        diag.append(torch.stack([((rho - 1).abs() > epsilon).float().mean(),
                                 (logp_old - logp).mean(), entropy]).detach())
    return torch.stack(diag).numpy()
```

訓練ループは 15.1 節のアクター・クリティックのループにおいて、単一のアクターステップを

ppo_epochs に置き換え、クリティックのパスの前にアドバンテージを計算するようにしたものである。まず rollout が 8 エピソードを実行して Batch に格納する。続いてループは Batch.gae と現在のクリティックを用いてアドバンテージを計算し、standardize によってバッチ全体で標準化する。また、logp_old を log_prob_np で読み取る。この関数は与えられた状態と行動に対する $\log \pi(a \mid s)$ を NumPy 配列として返す。次にクリティックが、15.1 節の Batch.lambda_return と 14.9 節の fit_value を用いて λ 収益のターゲットに向けた回帰パスを実行し、最後にアクターが自身のパスを実行する。各更新は、そこまでに費やした環境ステップ数とバッチの平均収益を順に返す。各バッチの診断指標は呼び出し元から渡されたリスト diag に追加され、CLIPPED、KL、ENTROPY がその 3 つの列を表す。

```
CLIPPED, KL, ENTROPY = 0, 1, 2   # the columns of ppo_epochs' diagnostics

def train_ppo(seed, ac, diag, surrogate=clipped_surrogate,
              num_epochs=num_epochs):
    """PPO on CartPole: freeze the advantages and the collecting policy's
    log-probs, then take num_epochs surrogate passes on each batch. Appends
    each batch's diagnostics to diag; yields (env steps, mean return)."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    steps = 0
    for _ in range(num_updates):
        b = d2l.rollout(env, ac.act, batch_episodes, rng)
        adv = d2l.standardize(b.gae(ac.value_np, gamma, lam))  # pre-fit critic
        logp_old = ac.log_prob_np(b.obs, b.act)
        for _ in range(critic_steps):   # fresh lambda-return target, per pass
            d2l.fit_value(ac, b.obs, b.lambda_return(ac.value_np, gamma, lam))
        diag.append(ppo_epochs(ac, b.obs, b.act, adv, logp_old, epsilon_clip,
                               num_epochs, entropy_coef, surrogate))
        steps += len(b)
        yield steps, float(b.episode_returns().mean())
```

15.2.4.3 バッチの再利用と環境ステップ数

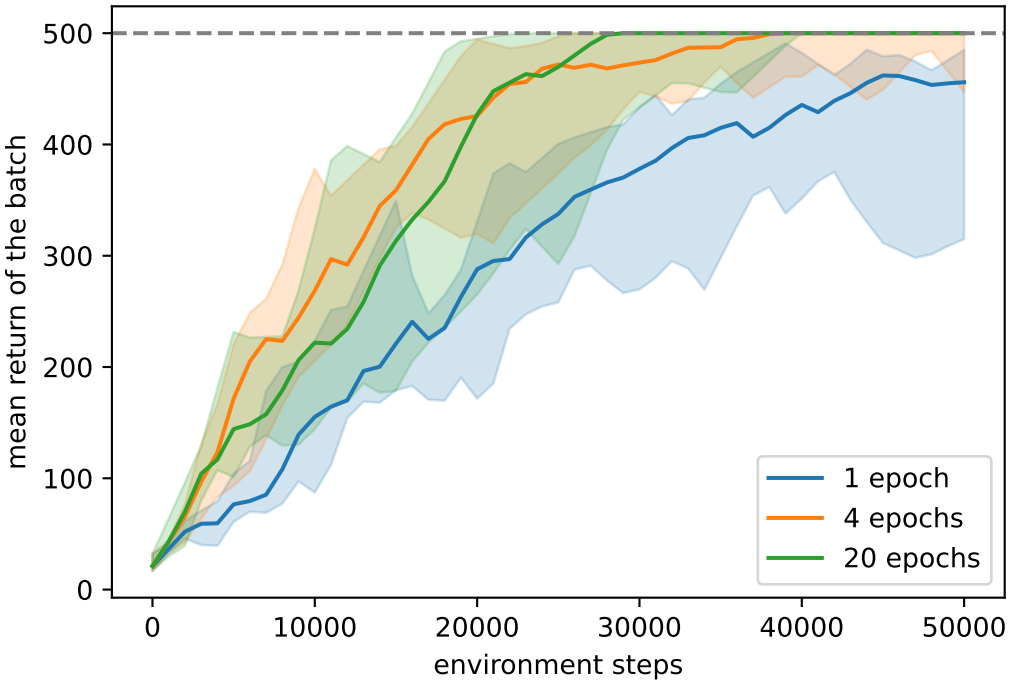
バッチの再利用によって環境ステップ数はどれだけ節約できるだろうか。バッチあたり 1、4、20 パスのクリップ付き PPO をそれぞれ 8 個のシードで訓練し、環境ステップ数が 50,000 に達した時点で各実行を打ち切って、そこまでに費やしたステップ数に対してバッチの平均収益をプロットする。1 パスの場合はすべての比が 1 に等しくクリップは一度も働かないため、この条件は 15.1 節の λ スイープに近く、エントロピーボーナスがあり、勾配の上限がなく、クリティックのパスの前にアドバンテージを計算する点だけが異なる。同節と同様に、d2l.on_grid を用いて各実行を共通のステップグリッド上に揃える。これらの実行では診断指標は不要なため、それぞれに空のリストを渡す。

```
budget = 50_000
grid = np.arange(0, budget + 1, 1000)
reuse = {}
for e in (1, 4, 20):
    runs_e = []
    for s in range(num_seeds):
        run = []
        for row in train_ppo(s, cartpole_agent(s), [], num_epochs=e):
```

```
            run.append(row)                    # (env steps, mean return)
            if row[0] >= budget:
                break
        runs_e.append(np.array(run))
    reuse[f'{e} epoch' + 's' * (e > 1)] = d2l.on_grid(runs_e, grid, 1)
```

reuse に格納された補間済みの収益曲線を grid に対してプロットすると、バッチの再利用によって追加の勾配パスと引き換えに環境との相互作用をどれだけ減らせるかがわかる。

```
d2l.plot_curves(reuse, xlabel='environment steps',
                ylabel='mean return of the batch', x=grid, reference=500)
```



各バッチを 4 エポック再利用すると、1 エポックの場合に必要な環境ステップ数（50,000 以上）の約半分にあたる約 25,000 ステップ、更新回数にしても約半分で上限に到達する（50,000 ステップの予算内では、1 エポックの中央値は上限のわずかに手前にとどまる）。さらに 20 エポックまで増やしてもそれに比べれば変化は小さく、クリップ付きの実行が崩壊することもない。この改善はバッチそのものに由来する。数千ステップからなるバッチには 1 回の勾配ステップでは抽出しきれない情報が含まれており、クリップがあるおかげでオプティマイザはどの行動確率も動かしすぎることなく複数回のステップを実行できるのである。

15.2.4.4 クリッピングのアブレーション

改善の大半を得るには 4 パスで十分であるが、本節の学習率で 20 パスを実行すると、1 つのバッチ内で方策は大きく動く。この領域でクリップは重要だろうか。同じ 8 個のシード上でクリップを有効にした 20 エポックの PPO を訓練し、後の検証用に訓練済みエージェントを、次の項のために全診断指標を保存しておく。

```
arms = {'clipped (PPO)': clipped_surrogate, 'no clip': plain_surrogate}
agents = {arm: [cartpole_agent(s) for s in range(num_seeds)] for arm in arms}
runs, diag = {}, {arm: [] for arm in arms}
runs['clipped (PPO)'] = np.array(
    [list(train_ppo(s, agents['clipped (PPO)'][s], diag['clipped (PPO)']))
     for s in range(num_seeds)])
```

続いて対照実験として、代理目的関数をクリップなしにする以外はすべての行が同一の実行を走らせる。その後、各条件の診断指標を（シード、更新、エポック、列）の形状に変形する。

```
runs['no clip'] = np.array(
    [list(train_ppo(s, agents['no clip'][s], diag['no clip'],
                    surrogate=plain_surrogate))
     for s in range(num_seeds)])
diag = {arm: np.reshape(d, (num_seeds, num_updates, num_epochs, 3))
        for arm, d in diag.items()}
```

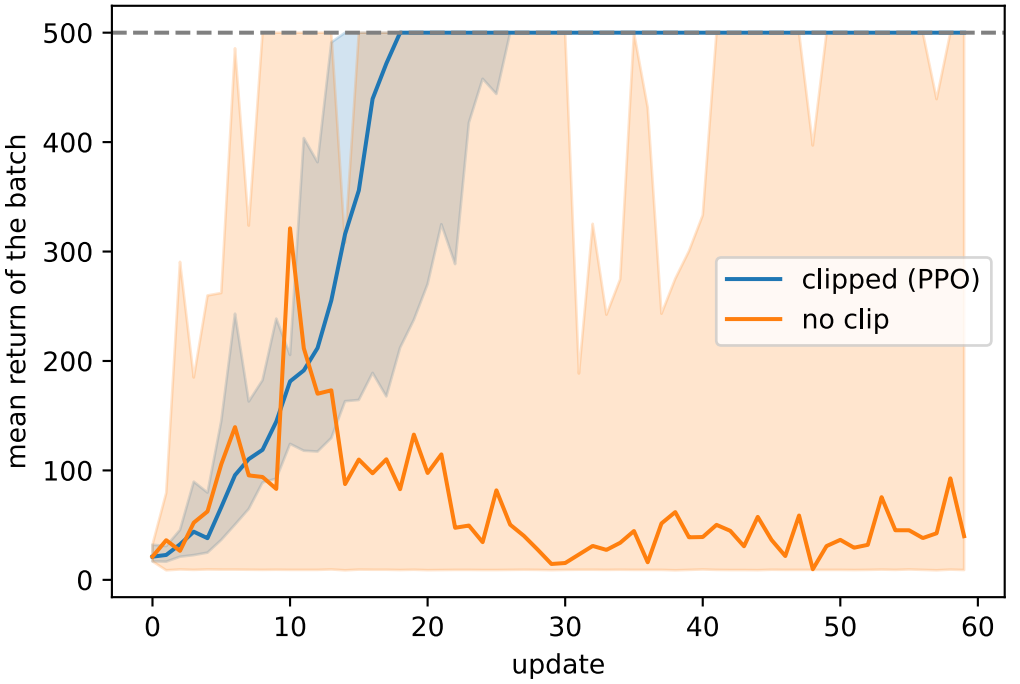
各実行を 2 つの基準で要約しよう。バッチ収益が最初に 475 に到達した後、最大値の半分である 250 を一度も下回らなければ、その実行はタスクを**維持**したとみなす。一方、最後の 10 回の更新における収益の平均が 100 を下回った場合、その実行は**破綻**に終わったとみなす。

```
def holds(ret):   # reaches 475, then never falls below 250 again
    hit = np.flatnonzero(ret >= 475)
    return len(hit) > 0 and ret[hit[0]:].min() >= 250

for name, r in runs.items():
    held = sum(holds(ret) for ret in r[:, :, 1])
    dead = r[:, -10:, 1].mean(axis=1) < 100
    print(f'{name:>13}: {held} of {num_seeds} seeds hold the task, '
          f'{int(dead.sum())} end dead')
```

```
clipped (PPO): 8 of 8 seeds hold the task, 0 end dead
      no clip: 0 of 8 seeds hold the task, 4 end dead
```

全更新にわたるバッチ収益の軌跡 `r[:, :, 1]` をプロットすると、`'clipped (PPO)'` の安定したプラトーと `'no clip'` の崩壊が鮮明に対比される。

```
d2l.plot_curves({name: r[:, :, 1] for name, r in runs.items()},
                xlabel='update', ylabel='mean return of the batch',
                reference=500)
```



20 エポック、学習率 10^{-2} の設定では、クリップ付きのシードはすべて上限に到達してそのままタスクを維持する。一方、クリップなしではタスクを維持できるシードはほとんどなく、一部は破綻に終わる。ノートブック外で 64 個のシードにわたって検証すると、タスクを維持できる実行は 20 回に 1 回程度にすぎず、約半数は破綻に終わる。しかもその大半は、毎ステップ同じ方向にカートを押し続けたときのエピソード長にあたる約 9 の収益に張り付く。8 個のシードではこれらの割合を安定して示すには少なすぎるため、上で出力される破綻した実行の数が半数を大きく下回ることもある。クリップなしのパスを繰り返すと方策は飽和領域へと押し込まれ、2 行動の例で見たように選ばれた行動の対数確率の勾配が小さくなる。すると次のバッチはその飽和した方策によって収集されるため、それを覆すような行動がほとんど含まれなくなるのである。PPO の原論文におけるアブレーション実験でも同じ結果が得られており、クリップもペナルティも持たない変種は検証されたすべての手法の中で最悪であった (Schulman et al. 2017)。クリップを必要としている原因はこの大きな学習率にある。アクターの学習率を 10 分の 1 の 10^{-3} に下げれば、クリップなし PPO でも 10 個のシードすべてで崩壊しなくなり、クリップ付き PPO と同じ速さで上限に到達する。さらに 3×10^{-4} ではクリップはほとんど働かない。ただし、どちらの学習率も本節の設定より遅く、本節の約 24,000 ステップに対して、上限到達までにそれぞれ約 35,000〜40,000 ステップおよび約 85,000〜90,000 ステップを要する。大きな学習率が訓練を高速にし、クリップがその大きな学習率を安全にしているのである。

クリップはどのくらいの頻度で働くのだろうか。各更新ではエポックごとにすべてのサンプルを 1 回ずつ検査し、最初のエポックはすべてのサンプルで $\rho_t = 1$ から始まる。エポックごとの全検査のうちバンド外に出た割合と、各バッチの最終エポックにおける同じ割合を出力してみよう。

```
for name, d in diag.items():
    print(f'{name:>13}: ratio checks outside the band: '
          f'{d[..., CLIPPED].mean():.1%} across all epochs')
    print(f'{"":>13}  {d[:, :, -1, CLIPPED].mean():.1%} at the last epoch '
          f'of each batch')
```

```
clipped (PPO): ratio checks outside the band: 5.3% across all epochs
               5.0% at the last epoch of each batch
      no clip: ratio checks outside the band: 17.6% across all epochs
               25.1% at the last epoch of each batch
```

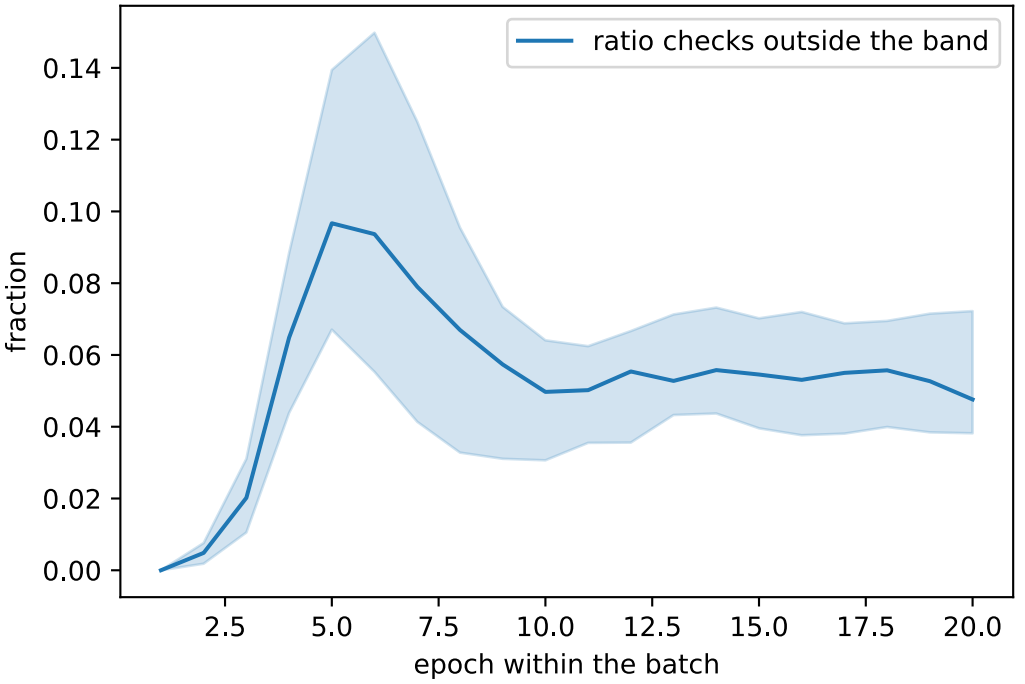
クリップ付きの実行では、どちらの割合も 20 回の検査につき 1 回前後である。クリップなしの対照実験では、比がバンド外に出る頻度はその数倍にのぼり、しかも平均よりバッチの最終エポックでのほうが高くなる。パスを重ねるたびに比がさらに離れていくのを止める仕組みがないからである。

15.2.4.5 訓練の診断指標

14.9 節で見たように、方策勾配の損失の値は学習の進捗について何も語らない。収益が 20 から 500 へと上昇するあいだも、損失は 0 付近にとどまり続ける。監視すべきは更新そのものの測定値であり、`diag` にはすでにすべての実行・シード・更新・エポックにわたる測定値が格納されている。まずは更新方向に平均を取り、1 バッチ内の 20 エポックの様子を見てみよう。

```
clipped = diag['clipped (PPO)']
d2l.plot_curves({'ratio checks outside the band':
                 clipped[..., CLIPPED].mean(1)},
                xlabel='epoch within the batch', ylabel='fraction',
                x=np.arange(1, num_epochs + 1))
kl = np.median(clipped[..., KL].mean(1), axis=0)
print(f'approximate KL: peak {kl.max():.4f} at epoch {kl.argmax() + 1}, '
      f'{kl[-1]:.4f} at epoch {num_epochs}')
for name, d in diag.items():
    print(f'{name:>13}: approximate KL at the last epoch, median '
          f'{np.median(d[:, :, -1, KL]):.3f}')
```

```
approximate KL: peak 0.0085 at epoch 6, 0.0053 at epoch 20
clipped (PPO): approximate KL at the last epoch, median 0.003
      no clip: approximate KL at the last epoch, median 0.139
```

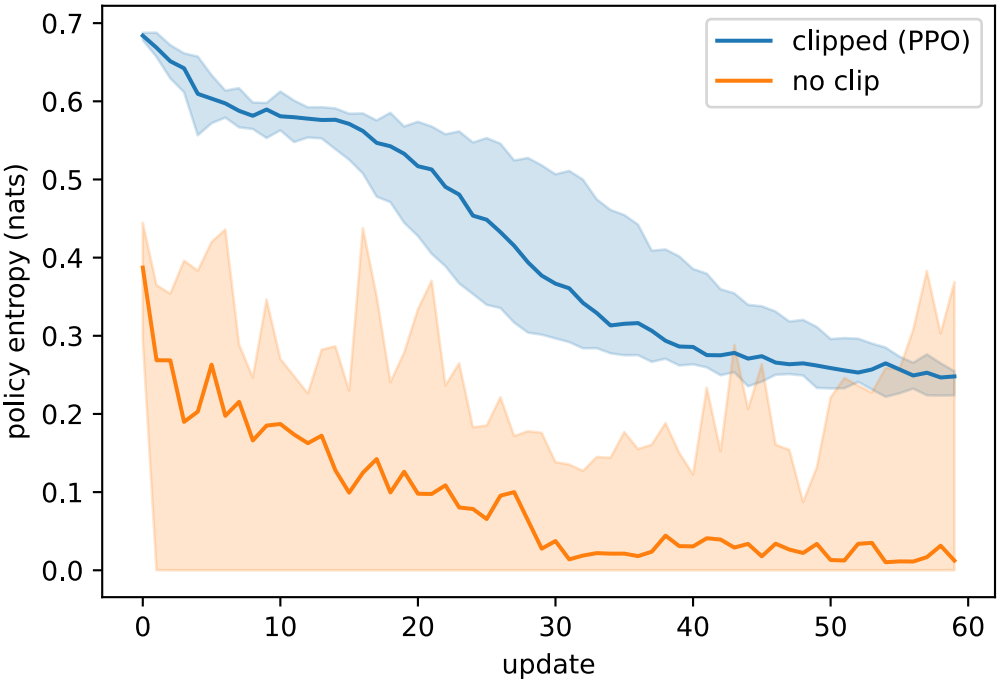


バッチ内では、すべての比が 1 に等しい最初のエポックにおいて、バンド外の割合は厳密に 0 である。その後 5〜6 エポック目あたりでピークに達し、20 エポック目までにはピークの約半分へと低下する。この低下は片側のみの最小値演算に由来する。悪化する側へバンドを出たサンプルはクリップなしの項の勾配を保つため比がバンド内へ引き戻されるのに対し、改善する側へバンドを超えたサンプルは勾配が 0 になってその場にとどまるからである。近似 KL も同じ形状を描き、値は約 10 分の 1 になる（上のセルがそのピークを出力する）。クリップがない場合、ダイバージェンスはパスごとに増大し続け、バッチの最終エポックではクリップ付きの実行の数十倍に達するのが普通である。

更新全体にわたってクリップ付き方策のエントロピーを追うと、方策がどれだけ決定論的に近づいたかがわかる。

```
d2l.plot_curves({name: d[..., ENTROPY].mean(-1) for name, d in diag.items()},
                xlabel='update', ylabel='policy entropy (nats)')
print(f'clipped: entropy {clipped[:, :5, :, ENTROPY].mean():.2f} over the '
      f'first five updates, {clipped[:, -5:, :, ENTROPY].mean():.2f} over the '
      f'last five')
```

```
clipped: entropy 0.65 over the first five updates, 0.25 over the last five
```



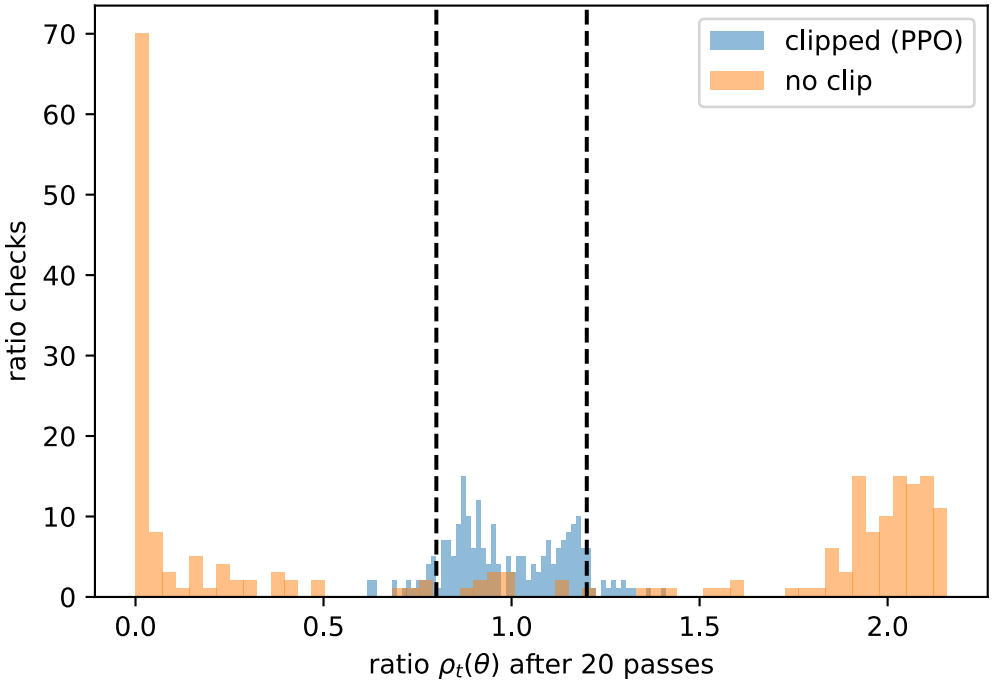
クリップ付きの実行のエントロピーは約 0.65 nats から約 0.25 nats へと低下する。学習が進むにつれて方策の選択はより確定的になり、エントロピーボーナスはそれを止めずに緩やかにする。一方、クリップなしの実行ではエントロピーがはるかに急速に失われており、これこそが破綻する実行を終焉させる飽和の正体である。

要約された割合だけでは、比がどのように分布しているかは見えない。そこで、バッチ内のドリフトが最大となる訓練開始時点の同一な初期エージェントを 2 つ用意し、固定した同一のアドバンテージと対数確率を持つ同じバッチを両方に与えて、一方ではクリップあり、もう一方ではクリップなしで 20 パスを実行し、各パスの後にすべての比を読み取る。3 つ目の同一なエージェント `collector` はバッチを収集した後はそのまま保持し、すべての比における古い方策とする。

```
probe = {arm: cartpole_agent(2) for arm in arms}   # twins
collector = cartpole_agent(2)                      # a third twin, kept fresh
env = gym.make('CartPole-v1')
env.reset(seed=2)
batch = d2l.rollout(env, collector.act, batch_episodes,
                    np.random.default_rng(2))
adv = d2l.standardize(batch.gae(collector.value_np, gamma, lam))
logp_old = collector.log_prob_np(batch.obs, batch.act)
rhos = {}
for arm, surrogate in arms.items():
    R = []
    for _ in range(num_epochs):   # one pass at a time, ratios read after
        ppo_epochs(probe[arm], batch.obs, batch.act, adv, logp_old,
                   epsilon_clip, 1, entropy_coef, surrogate)
        R.append(np.exp(probe[arm].log_prob_np(batch.obs, batch.act)
                        - logp_old))
    rhos[arm] = np.array(R)
```

d2l.plot_hist（オンライン付録「ユーティリティ関数とクラス」）を用いて最終パス後の 2 組の比のヒストグラムを重ねて描画し、バンドの両端を破線で示す。

```
d2l.plot_hist({arm: r[-1] for arm, r in rhos.items()},
              xlabel=rf'ratio $\rho_t(\theta)$ after {num_epochs} passes',
              ylabel='ratio checks',
              vlines=(1 - epsilon_clip, 1 + epsilon_clip))
```

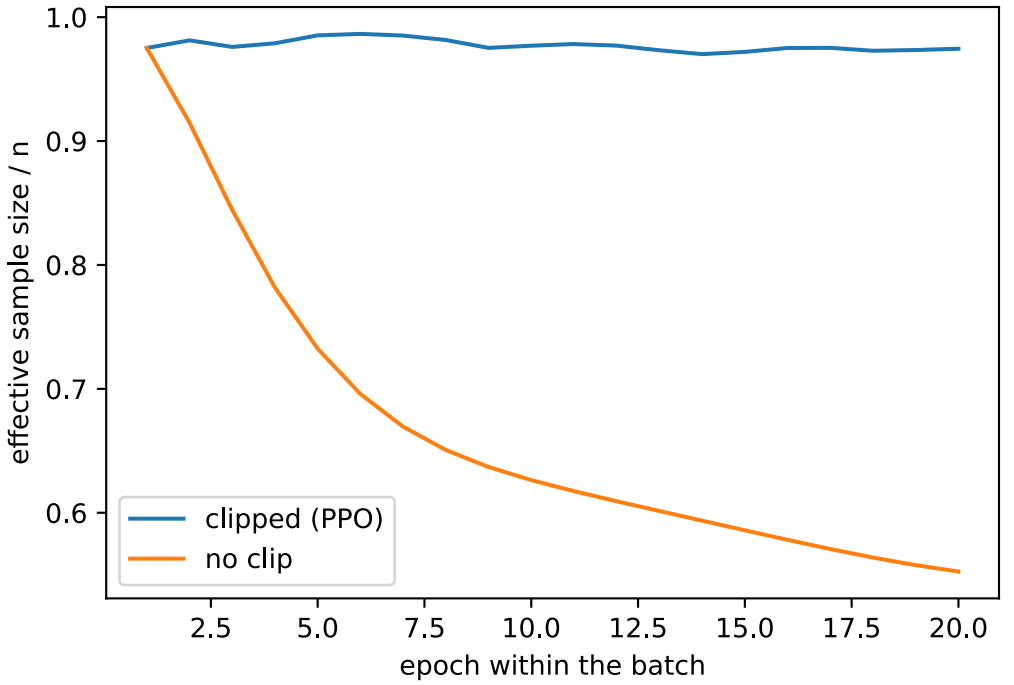


20 パスの後でも、クリップ付きエージェントの比は 1 の近くにまとまっており、破線で示されたバンドの端を超えるものは少数にとどまる。一方、クリップなしの比は 0 の近くに積み上がり、残りの大半はバンドを大きく超えて 2〜3 付近まで広がっている。2 や 3 という比は、初期方策が確率 2 分の 1 や 3 分の 1 で選んだ行動を新しい方策がほぼ常に選ぶようになった場合の $1/\pi_{\theta_{\text{old}}}(a)$ に相当し、0 付近の比はもう一方の行動に対応する。このように重み付けし直されたバッチには、どれだけの価値が残っているのだろうか。n 個の重要度重みの**有効標本サイズ**（effective sample size）$N_{\text{eff}} = 1/\sum_j \bar{w}_j^2$ がこの問いに答える。ここで $\bar{w}_j$ は、ステップ j の比をバッチ内のすべての比の和で割った値である。この量はすべての重みが等しいときに n に等しくなり、単一の重みが支配するときに 1 となる。

```
ess = {}
for arm, r in rhos.items():
    w = r / r.sum(axis=1, keepdims=True)   # normalized weights per epoch
    ess[arm] = 1 / (w ** 2).sum(axis=1) / r.shape[1]
d2l.plot_curves(ess, xlabel='epoch within the batch',
                ylabel='effective sample size / n',
                x=np.arange(1, num_epochs + 1))
print(f'after {num_epochs} epochs the batch is worth '
```

```
        + ' vs '.join(f'{v[-1]:.0%} ({k})' for k, v in ess.items())
        + f' of its {len(batch)} steps')
```

```
after 20 epochs the batch is worth 97% (clipped (PPO)) vs 55% (no clip) of its 226 steps
```



クリップありでは、20 エポック目まで重みはほぼ均等に保たれる。クリップなしでは、有効標本サイズはクリップありの条件を大きく下回り、バッチの約半分まで落ち込む。この値は飽和によって説明できる。新しい方策がある行動 $a^*(s)$ を確実に選ぶようになったとしよう。このとき s におけるサンプルは、重み $1/\pi_{\theta_{\text{old}}}(a^* \mid s)$ を持ち（そのサンプルの行動が a^* であった場合であり、これは確率 $\pi_{\theta_{\text{old}}}(a^* \mid s)$ で起きる）、それ以外では重み 0 となる。重みの平均は 1、重みの二乗平均は $1/\pi_{\theta_{\text{old}}}(a^* \mid s)$ であるため、バッチが大きくなるにつれて N_{eff}/n は訪問した状態にわたる $\pi_{\theta_{\text{old}}}(a^* \mid s)$ の調和平均に近づく。2 つの行動に対してほぼ一様な初期方策では、その値は約 2 分の 1 である。これを確かめるには、サンプリングされた行動だけでなくすべての行動の確率が必要になる。`action_probs` はそれらを NumPy 配列として返し、未訓練の双子エージェント `collector` が収集方策にあたる。

```
def action_probs(ac, obs):
    """pi(.|s) for a numpy batch of states, as numpy."""
    with torch.inference_mode():
        return torch.softmax(ac.policy(torch.as_tensor(obs)), -1).numpy()
```

```
p_old = action_probs(collector, batch.obs)     # the collecting policy
p_new = action_probs(probe['no clip'], batch.obs)
a_star = p_new.argmax(axis=1)                  # the unclipped choice
p_star = p_old[np.arange(len(batch)), a_star]
```

```
print(f'unclipped policy: median probability of its choice '
      f'{np.median(p_new.max(axis=1)):.3f}')
print(f'harmonic mean of the old probability of that choice: '
      f'{1 / (1 / p_star).mean():.2f}')
```

```
unclipped policy: median probability of its choice 0.996
harmonic mean of the old probability of that choice: 0.49
```

クリップなしの方策はほぼ決定論的になっており、調和平均はクリップなし条件の有効標本サイズとの差が約 0.1 以内に収まっている（方策が完全には決定論的でないため、調和平均のほうがわずかに小さい）。有効標本サイズは重みがどれだけ均等に散らばっているかを測るものであり、アドバンテージや新しい方策が訪問するはずの状態は無視するため、ダイバージェンスやクリップ率を置き換えるのではなく補完する指標となる。

最後に、サンプリングのノイズを取り除くため最も確率の高い行動を貪欲に選ぶようにすると、訓練済みのクリップ付きエージェントは時間制限までポールをバランスさせ続けるはずである。

```
ac, env = agents['clipped (PPO)'][0], gym.make('CartPole-v1')
env.reset(seed=2)
score = d2l.evaluate(env, ac.act_greedy, num_episodes=100)
print(f'greedy mean return over 100 episodes: {score:.0f}')
```

```
greedy mean return over 100 episodes: 500
```

15.2.4.6 ベクトル化された収集とミニバッチ

ここまでのループでは、1 つの環境でエピソードを最後まで実行し、フルバッチのパスを取っていた。`cleanrl/ppo.py` (Huang, Rousslan F. J. Dossa, et al. 2022) のような実用実装では、環境の N 個のコピーを並列に固定ホライズン T だけ進め、$N \times T$ の長方形配列をなす遷移データを生成して、その端でまだ続いているすべてのエピソードを打ち切る。打ち切り箇所では、14.9 節が CartPole の時間制限で用いたのと同じブートストラップによって、記録されていない将来の価値をクリティックが補う。訓練済みエージェントを用い、8 個の環境を 32 ステップ進めてそのような長方形データを 1 つ収集してみよう。

```
envs = gym.vector.SyncVectorEnv([lambda: gym.make('CartPole-v1')] * 8)
obs, _ = envs.reset(seed=0)
rng, steps = np.random.default_rng(3), []
for _ in range(32):                       # the fixed 32 x 8 rectangle
    u = rng.random((len(obs), 1))         # inverse-CDF sampling, per env
    act = (u > action_probs(ac, obs).cumsum(1)[:, :-1]).sum(1)
    obs, rew, term, trunc, _ = envs.step(act)
    steps.append((rew, term | trunc))
rew, done = map(np.array, zip(*steps))
print(f'a fixed {rew.shape} rectangle of steps, '
      f'{int(done.sum())} episode boundaries inside it')
print(f'V at the cut, the value of the unrecorded future: '
      f'{ac.value_np(obs).round(0)}')
```

```
a fixed (32, 8) rectangle of steps, 0 episode boundaries inside it
V at the cut, the value of the unrecorded future: [100. 100. 100. 100. 100. 100. 100. 100.]
```

訓練済みエージェントは 32 ステップ以内には倒れないため、8 本の軌跡はすべて打ち切り箇所で終わり、その価値は永続的にバランスを保った場合の割引価値である $1/(1-\gamma) = 100$ に近い値となる。ブートストラップがなければ、GAE は各打ち切りを転倒として扱い、その直前の行動に大きな負のアドバンテージを与えてしまうだろう。また、Gymnasium のベクトル環境は終了したエピソードを自動的に再開する。エピソード終了直後のステップは報酬 0 とともに新しいエピソードの最初の観測を返し、渡された行動を無視するため、実装ではそのステップをマスクしてアドバンテージと損失の計算から除外する。最後に、実用コードはこの長方形データをシャッフルしてミニバッチに分割し、その上で数エポック（典型的には 4 エポック）を実行する。これにより各比が検査される回数は 20 回ではなく 4 回になり、ステップが小さくなることで必要なメモリが減り、勾配ノイズも加わる。演習問題 5 でこのループを構築する。

ベクトル化された収集は、1 サンプルあたりのコストの意味も変える。本章で環境ステップ数を数えているのは、ロボットアームの実機ではすべてのステップが現実の動作だからである。しかし GPU 上で動作する物理シミュレータは、一度に数千体のロボットのコピーを進めることができるため (Makoviychuk et al. 2021)、数千環境分の幅を持つ長方形データが 1 秒未満で得られ、重要になるコストは実経過時間へと変わる。非常に大きなバッチによって少数回の各更新が正確になるため、PPO はこの領域に適している。方策が実機へ転移するようにシミュレータのパラメータをランダム化する**ドメインランダマイゼーション** (Tobin et al. 2017) と組み合わせることで、脚式ロボットは数分の訓練で歩行を学習する (Rudin et al. 2022)。次節以降の手法のほうがはるかに少ないステップ数で済むにもかかわらず、この分野で今なお PPO が標準であり続けている理由はここにある。次節以降の手法が優位に立つのは、ステップが実機上の現実の動作である場合である。

PPO によってエージェントは各バッチに対して複数回のステップを実行できるようになり、CartPole では 4 エポックで 1 エポックの半分の環境ステップ数で上限に到達する。そのクリップは各ステップが行動確率をどれだけ動かすかを制限するが、制限しているのは経路であって到達点ではない。クリップ付きの実行でもエントロピーが依然として 0.65 nats から 0.25 nats へと低下するのは、目的関数があくまで小さなエントロピーボーナスを伴う収益であり、その最適解がほぼ決定論的だからである。15.3 節では代わりに目的関数そのものを変更し、方策を固定された参照方策の近くに保つことで、経路だけでなく最適解自体を動かす。

15.2.5 まとめ

数千環境ステップからなるオンポリシーのバッチは 1 回の勾配ステップ以上に活用されるべきであるが、パラメータ空間における同じ大きさのステップは方策をまったく異なる大きさで変化させうる。重要度サンプリングは古いデータ上で新しい方策を厳密に評価するものの、その分散はエピソード長に対して指数関数的に増大する。そこでステップごとに 1 つの比だけを残すと、その増大が取り除かれる代わりに、古い方策の状態において新しい方策の行動を評価することになる。性能差補題はこの置き換えこそが代理目的関数の唯一の誤差であることを示しており、TRPO は新旧の行動分布間のダイバージェンスに制約を課すことでこの誤差を抑える。これに対し PPO は、比が 1 の周りのバンドを出た時点で目的関数への各サンプルの寄与をクリップすることで、数エポックにわたる各バッチの再利用を可能にする。CartPole では 4 エポックで 1 エポックの約半分の環境ステップ数で上限に到達し、大きな学習率でもクリップが方策の崩壊を防ぐ。クリップ率、近似 KL、エントロピー、および比の有効標本サイズは、各更新が方策をどれだけ動かしたかを示す指標となる。

15.2.6 演習問題

1. **1 つのロジットに対するステップ制限。**●Figure 15.3 の 2 行動方策 $\pi_\theta(a{=}1) = \sigma(\theta)$ と、更新 $\theta \to \theta + \Delta$ を制限する 3 つの方法（ダイバージェンスに対する信頼領域、PPO のクリップ、および $|\Delta|$ の上限）を考える。

1. $D_{\mathrm{KL}}(\pi_\theta \| \pi_{\theta+\Delta})$ を Δ の 2 次まで展開し、$\frac{1}{2}\sigma'(\theta)\,\Delta^2$ に等しいことを示せ。図に描かれた $\Delta = 2$ の 2 つの更新（$\theta = 0$ からの更新と $\theta = 6$ からの更新）について厳密なダイバージェンスを計算せよ。$\Delta_{\mathrm{KL}} = 0.02$ の信頼領域内に収まるのはどちらか。
2. この二次近似式を用いて、この信頼領域が許容する最大の $|\Delta|$ を $\theta = 0$ および $\theta = 6$ において求めよ。$\theta = 6$ において、その大きさを持つ正負両方向の 2 つのステップに対する厳密なダイバージェンスを計算せよ。そこで二次近似式の精度が $\theta = 0$ のときよりはるかに低くなるのはなぜか。
3. 次に、正のアドバンテージを持つ 1 つのサンプルに対し、$\epsilon_{\mathrm{clip}} = 0.2$ とした Equation 15.22 のクリップによってステップを制限する。更新はサンプリングされた行動の確率を高めるため、$\Delta > 0$（サンプルが $a = 1$ のとき）または $\Delta < 0$（サンプルが $a = 2$ のとき）となる。クリップ付き目的関数が増加し続ける最大の $|\Delta|$ を、$\theta = 0$ と $\theta = 6$ のそれぞれで両方の行動について求めよ。
4. 方策が θ に対して鈍感な領域では大きなパラメータステップを許容し、敏感な領域では小さなステップのみを許容するのは、3 つの制限のうちどれか。$\theta = 6$ において、サンプリングされた行動がどちらのときにクリップは信頼領域より緩くなり、どちらのときにより厳しくなるか。

2. **爆発する比の積**。状態を無視する方策が $\pi_{\theta_{\mathrm{old}}}$ でエピソードを収集し、そのデータを用いて π_θ を評価するとする。軌跡の比 Equation 15.15 を W_T（ステップ数 T のエピソードに対する比）と書く。

   1. 古い方策のもとで $\mathbb{E}[W_T] = 1$ および $\mathrm{Var}[W_T] = (1+\chi^2)^T - 1$ が成り立つことを示せ。ここで $\chi^2 = \sum_a \big(\pi_\theta(a) - \pi_{\theta_{\mathrm{old}}}(a)\big)^2 / \pi_{\theta_{\mathrm{old}}}(a)$ である。本文の値 $1.04^T - 1$ が $\pi_{\theta_{\mathrm{old}}} = (0.5, 0.5)$ と $\pi_\theta = (0.6, 0.4)$ に対して成り立つことを確かめよ。
   2. エピソード数 n が大きくなるにつれて、重み W_T の有効標本サイズは $n/\mathbb{E}[W_T^2]$ に近づく。(a) の 2 つの方策について、この極限が 1 に等しくなる n（$T = 200$ のとき）はいくらか。その本数のエピソードからなるバッチを 1 つシミュレーションし（1 エピソード内で $a = 1$ となるステップ数は二項分布に従う）、その有効標本サイズを計算せよ。バッチから算出される有効標本サイズがその極限よりはるかに大きくなるのはなぜか。重要度重み付き推定量の分散を表しているのはこの 2 つのうちどちらか。
   3. ステップごとの代理目的関数 Equation 15.17 は、各ステップを単一の比 ρ_t で重み付けする。ここでのその重みの分散はいくらか。また、Equation 15.17 におけるどの近似によってそれは T に依存しなくなっているか。

3. **診断指標が推定している量**。●状態 s を固定し、$a \sim \pi_{\theta_{\mathrm{old}}}(\cdot \mid s)$ をサンプリングして、$\rho = \pi_\theta(a \mid s)/\pi_{\theta_{\mathrm{old}}}(a \mid s)$ とおく。

   1. ppo_epochs の近似 KL はバッチ全体で $k_1 = -\log\rho$ を平均する。$\mathbb{E}[k_1] = D_{\mathrm{KL}}(\pi_{\theta_{\mathrm{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s))$ が成り立つことを示せ。続いて $k_3 = (\rho - 1) - \log\rho$ も同じ期待値を持ち、すべてのサンプルに対して非負であることを示せ。両推定量を $\rho - 1$ の 2 次まで展開せよ。方策の変化が小さいとき、k_3 の分散が k_1 よりはるかに小さくなるのはなぜか。なお、`cleanrl/ppo.py` は両方を記録している。

   2. 問題 2b では、有効標本サイズの極限 $n/\mathbb{E}[w^2]$（独立な n 個の重み w が $\mathbb{E}[w] = 1$ を満たす場合）を用いた。大数の法則を用いてこの極限を導出せよ。続いて $\mathbb{E}[\rho^2] = 1 + \chi^2$ を示し、重み ρ の有効標本サイズを n で割った値が $1/(1+\chi^2)$ に近づくことを示せ。ここで、次とおく。

$$\chi^2 = \sum_a \frac{\big(\pi_\theta(a \mid s) - \pi_{\theta_{\mathrm{old}}}(a \mid s)\big)^2}{\pi_{\theta_{\mathrm{old}}}(a \mid s)}.$$

方策の変化について 2 次までで $\chi^2 \approx 2D_{\mathrm{KL}}$ が成り立つことを示せ。

3. バッチから状態が引かれるとき、χ^2 は訪問した状態にわたるその平均となる。χ^2 のどの値を、15.2.4.5 節の 2 つの割合 `ess[name][-1]` は意味するか。(b) の二次の関係式は、それぞれに対してどの近似 KL を予測するか。各予測を、同じ比に対する k_1 推定値（`-np.log(rhos[arm][-1])).mean()`（各 `arm` について）と比較せよ。どちらのエージェントで二次の関係式が破綻するか。方策の変化の大きさ以外に、方策の訓練に用いたまさにそのバッチ上で測定された比は、(b) のどの仮定に反しているか。

John Schulman による Approximating KL Divergence の推定量 k_1 および k_3 に基づく。

4. [code] **KL による早期停止。**●クリッピングに加えて、方策が十分に動いた時点でバッチの再利用を打ち切る実装も可能である。Spinning Up の PPO は毎パスの前に近似 KL を計算し、それが 1.5 倍の `target_kl`（デフォルト値は 0.01）を超えた時点でそのバッチに対するパスを終了する。

   1. `ppo_epochs` の呼び出し（`train_ppo` 内）を、単一パスのループ `ppo_epochs(..., 1, ...)` に置き換えよ（`rhos` を 15.2.4.5 節で収集しているループと同様にする）。各パスの前に `log_prob_np` から近似 KL を計算し、停止規則を適用せよ。各バッチに費やされたパス数を記録せよ。
   2. アブレーション実験の 8 個のシード上で、早期停止を伴うクリップなしの変種を実行せよ。この停止規則だけで崩壊を防げるか。バッチあたりのパス数は訓練を通じてどのように変化するか。
   3. クリップ付きの変種も同様に早期停止付きで実行せよ。この実行において、1 バッチでの方策の変化を制限しているのは 2 つの仕組みのうちどちらか。その答えから、クリップを信頼領域と表現することについて何が言えるか。

Joshua Achiam による Spinning Up in Deep RL の PPO 実装（`target_kl` による早期停止）に基づく。

5. [code] **標準的なレシピ。**●●15.2.4.6 節で説明したループを構築せよ。すなわち、`gym.vector.SyncVectorEnv` 内の 8 個の環境を更新ごとに 32 ステップ進め、各打ち切り箇所でのクリティックの価値を用い自動リセットのステップをマスクして除外したうえで 32×8 の長方形データに対して GAE を計算し、続いてサイズ 64 のシャッフルされたミニバッチに分割して長方形データ上で 4 エポックを実行する。各ミニバッチでは、そのミニバッチ内で標準化した独自のアドバンテージを用いて `ppo_epochs` を 1 パス実行し、エポック開始前に計算した λ 収益のターゲットに向けて `fit_value` を 1 ステップ実行する。両ネットワークの学習率は 10^{-3} とせよ。

   1. 3 個のシードで訓練し、完了した直近 8 エピソードの平均が少なくとも 475 に達するまでの環境ステップ数を記録せよ。正しい実装では約 45,000 ステップ（大半のシードで 35,000〜55,000 ステップ）を要し、約 25,000 ステップで済む 15.2.4.3 節の再利用実験における 4 エポック条件より多くなる。
   2. クリティックの学習率を本節のループと同じ 10^{-2} に引き上げ、アクターの学習率は 10^{-3} のまま保て。正しい実装では約 15,000〜20,000 ステップで 475 に到達し、再利用実験のどの条件よりも少なくなる。(a) の実行の律速となっていたのはどちらのネットワークか。ベクトル化もミニバッチも用いない本節のループを同じ 2 つの学習率で動かすと、約 35,000〜40,000 ステップを要する。短い長方形データにすることで、1 環境ステップあたりに方策が更新される頻度はどう変わるか。
   3. 自動リセットのステップに対するマスクを外せ。それでも実行は学習するか。マスクを外した版はクリティックのターゲットにどのような遷移をつけ加えるか。また、CartPole がその影響をほとんど受けないのはなぜか。

6. **単調改善下界。**●●この問題では、Equation 15.20 を Schulman et al. (2015) のカップリン

グ論法によって導出する。$\gamma < 1$ の無限ホライズンタスクを考え、$\pi = \pi_{\theta_{\text{old}}}$、$\tilde{\pi} = \pi_\theta$、$A = A^{\text{old}}$、および $\bar{A}(s) = \sum_a \tilde{\pi}(a \mid s)\, A(s,a)$ と書く。$\Delta_{\text{TV}} = \max_s D_{\text{TV}}(\pi(\cdot \mid s), \tilde{\pi}(\cdot \mid s))$ とおき、ここで全変動距離は $D_{\text{TV}}(p,q) = \frac{1}{2}\sum_a |p(a) - q(a)|$ で定義される。

1. Equation 15.18 から $J(\theta) - J(\theta_{\text{old}}) = \sum_t \gamma^t \mathbb{E}_{s_t \sim \tilde{\pi}}\big[\bar{A}(s_t)\big]$ となること、および母集団代理目的関数 Equation 15.19 が $\bar{L}(\theta) = \sum_t \gamma^t \mathbb{E}_{s_t \sim \pi}\big[\bar{A}(s_t)\big]$ と書けることを示せ。ここで $s_t \sim \pi$ は時刻 t における方策 π のもとでの状態分布を表す。

2. 全変動距離が Δ_{TV} である 2 つの分布はカップリングできる。すなわち、$(a, \tilde{a})$ の同時分布であって、周辺分布が $\pi(\cdot \mid s)$ と $\tilde{\pi}(\cdot \mid s)$ であり、すべての状態で $P(a \neq \tilde{a}) \le \Delta_{\text{TV}}$ を満たすものが存在する。$\mathbb{E}_{a \sim \pi}[A(s,a)] = 0$ を用いて $|\bar{A}(s)| \le 2\Delta_{\text{TV}} A_{\max}$ を示せ。

3. 同じ開始状態から毎ステップでカップリングした行動を用いて両方策を実行し、m_t を時刻 t より前に 2 つの行動が食い違ったステップ数とする。事象 $m_t = 0$ の上では、2 本の軌跡は時刻 t まで一致する。次を示せ。

$$\big|\mathbb{E}_{s_t \sim \tilde{\pi}}[\bar{A}(s_t)] - \mathbb{E}_{s_t \sim \pi}[\bar{A}(s_t)]\big| \le 4\Delta_{\text{TV}}\big(1 - (1 - \Delta_{\text{TV}})^t\big) A_{\max}. \tag{15.23}$$

4. t について和を取ることで次を得て、

$$\big|J(\theta) - J(\theta_{\text{old}}) - \bar{L}(\theta)\big| \le \frac{4\Delta_{\text{TV}}^2 \gamma A_{\max}}{(1-\gamma)^2}, \tag{15.24}$$

   $D_{\text{TV}}^2 \le D_{\text{KL}}$ を用いて証明を完成せよ。

5. 係数 $4\gamma/(1-\gamma)^2$ の値を $\gamma = 0.99$ において計算せよ。(b) を用いて $\bar{L}(\theta) \le 2\Delta_{\text{TV}} A_{\max}/(1-\gamma)$ を示せ。ピンスカーの不等式 $D_{\text{KL}} \ge 2D_{\text{TV}}^2$ を用い、この下界が改善を保証するのは $\Delta_{\text{TV}} < (1-\gamma)/(4\gamma)$ の場合に限られることを示せ。このとき、改善が保証されるステップの $\bar{L}(\theta)$ は $A_{\max}$ を単位として最大どれだけ大きくなれるか。Δ_{TV} に対するこの上限を、ピンスカーの不等式が許容する全変動 $\sqrt{\Delta_{\text{KL}}/2}$（ダイバージェンスが予算 $\Delta_{\text{KL}} = 0.01$ に等しい状態における値であり、これは Schulman et al. (2015) が全実験で用いた予算である）と比較せよ。

7. [extended] **構成要素から作る TRPO**。●●`ppo_epochs` の呼び出し（`train_ppo` 内）を、1 回の TRPO 更新 (Schulman et al. 2015)（制約付き問題 Equation 15.21 で $\Delta_{\text{KL}} = 0.01$ としたもの）に置き換え、クリティックとアドバンテージはそのまま残せ。`ac.policy` のパラメータを 1 本の平坦なベクトルとして扱い、収集方策の行動確率 `action_probs(ac, b.obs)` を `logp_old` とともに固定せよ。

   1. 代理目的関数 Equation 15.17 の勾配 $\mathbf{g}$（θ_{old} における値）を計算せよ。続いて、バッチ内の状態にわたる平均ダイバージェンスの勾配をもう一度微分することで、フィッシャー・ベクトル積 $F\mathbf{v}$（θ_{old} におけるそのダイバージェンスのヘッセ行列を $\mathbf{v}$ に作用させたもの）を返す関数を作成せよ。勾配の有限差分と比較して検証せよ。
   2. 共役勾配法の 10 回の反復によって $F\mathbf{x} = \mathbf{g}$ を近似的に解き、ステップを $\xi\mathbf{x}$（ただし $\xi = \sqrt{2\Delta_{\text{KL}}/\mathbf{x}^\top F\mathbf{x}}$）にスケーリングせよ。$\xi$ を、実測される平均ダイバージェンスが高々 Δ_{KL} になり、かつ代理目的関数が増加するまで半分にし続けよ。
   3. 3 個のシードで訓練し、15.2.4.4 節のクリップ付きの実行と、学習曲線、各更新の実測ダイバージェンス、および半減ステップの回数について比較せよ。各手法は 1 バッチあたり何回の勾配計算とフィッシャー・ベクトル積の計算を費やすか。

4. Engstrom et al. (2020) は、コードレベルの設計選択を揃えると PPO と TRPO が同等の性能を示すことを見出した。ここでは両方とも方策更新以外のすべての選択を共有している。比較結果はこの知見と一致するか。

15.3 KL 正則化方策最適化

第 14 章および本章でこれまでに扱った手法はすべて、誰かが報酬を書き下していることを前提としていた。凍った湖やポールを載せたカートであれば、「ゴールに到達する」「ポールを直立に保つ」といったように報酬の設計は簡単である。しかし、人間が関心を寄せる大半のタスクではそうはいかない。有益な回答、安全な車線変更、あるいは読みやすい証明にスコアをつける数式を書き下せる人などいないからである。とはいえ、人間は 2 つの試行を見比べてどちらを好むかを答えることはでき、そのような比較データに報酬を当てはめることができる。広く使われた初期のチャットモデルに有益な応答を学習させたのもこの方法であり、人間の比較データに報酬モデル（reward model）を当てはめ、それに対して方策を訓練したのである (Christiano et al. 2017; Ouyang et al. 2022)。

データに当てはめた報酬は、人手で書き下した報酬にはなかった問題を突きつける。モデルは比較データが情報を提供する領域でのみ正確であり、それ以外の領域では推測に頼る一方、オプティマイザはあらゆる領域を探索する。比較データが決してカバーしなかった領域で、たまたま当てはめた報酬が高くなる場所をオプティマイザが見つけ出すと、方策はそこへ進み、真の品質は低下してしまう。実務で定着した対策は、比較データを生成した方策からの乖離に対してコストを課すこと、すなわちその参照からの KL ダイバージェンスにペナルティを科すことである。15.2 節の信頼領域（trust region）は、各更新が直前の方策からどれだけ動けるかを制限していた。これに対し、このペナルティは目的関数の一部であるため、解に至る経路だけでなく解そのものを変化させ、しかもその解は参照分布を報酬の指数で再重み付けした閉形式を持つ。

まず真の報酬がわかっているグリッドワールドを構築し、比較データから報酬を当てはめ、オプティマイザがその隙を突く様子を観察する。次にペナルティ付きの問題を 1 回の意思決定について解き、その解を多段階へと拡張した上で、方策が参照から離れるにつれて真の品質が上昇した後に低下していく様子をたどる。

15.3.1 比較からの報酬の学習

何がうまくいかなくなるのかを見るには、真の報酬が既知であるタスクを用意し、それを学習器には隠したまま結果を採点できるようにする必要がある。小さなグリッドワールドがその好例となる。

15.3.1.1 グリッドワールドとその参照方策

この環境は決定論的な移動を持つ 4×4 のグリッドであり、Figure 15.6 に示すとおりである。エピソードは左上のセルから始まり、右下の角にあるゴールに入ると終了して $+1$ の報酬を得る。それ以外の移動には 0.04 のコストがかかり、2 行目に隣り合って並ぶ 2 つの危険セル、すなわち**危険レーン**（hazard lane）のいずれかに入る移動には 1.5 のコストがかかる。真の報酬 $r^*(s, a)$ は、行動 a によって状態 s から入るセルに対する支払いである。セルには行ごとに番号が振られており、開始位置は状態 0、危険セルは状態 5 と 6、ゴールは状態 15 である。行動 0 から 3 はそれぞれ左、下、右、上への移動を表し、壁に向かって移動するとエージェントはその場にとどまる。

比較データは、1 つの固定された方策である**参照方策**（reference policy）π_{ref} の軌跡から収集する。これを有能だが慎重な方策とするため、温度 0.25 における最適行動価値のソフトマックス $\pi_{\text{ref}}(a \mid s) \propto e^{Q^*(s,a)/0.25}$ として定義する。開始位置では、ゴールへ向かう 2 つの移動は $Q^* = 0.59$、壁に向かう 2 つの移動は 0.52 であるため、参照方策は良い移動をそれぞれ確率

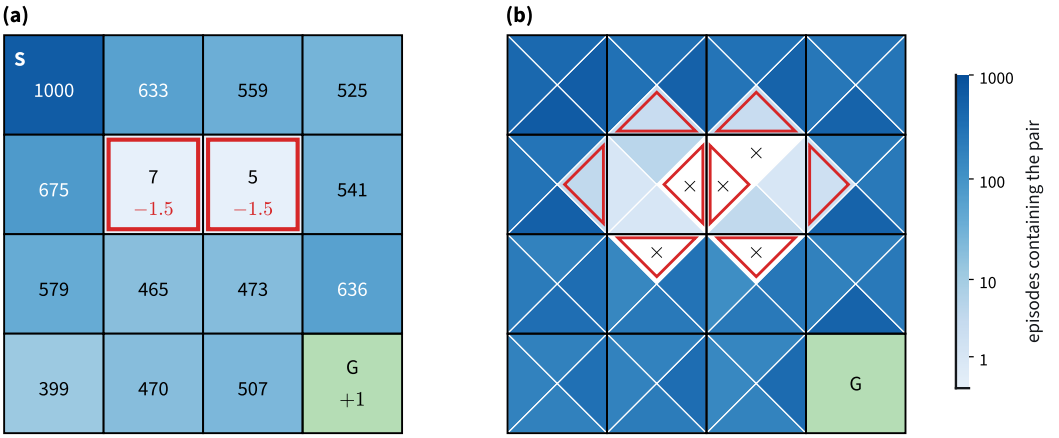


Figure 15.6: 本節のグリッドワールド。(a) エピソードは状態 0（S）から始まり、ゴールである状態 15（G、+1）で終了する。危険セルである 5 または 6 への移動には 1.5、それ以外のすべての移動には 0.04 のコストがかかる。各セルの数値は、1,000 本の参照軌跡のうち何本でそのセルが訪問されたかを示す。(b) 各状態行動対の台。各セルは左・下・右・上の移動に対応する 4 つの三角形に分割され、その対を含む軌跡の本数（対数スケール）で濃淡が付けられている。バツ印は一度も生じない対を表し、危険レーンへ入る 8 つの移動は太枠で囲まれている。

0.28 で、無駄な移動をそれぞれ確率 0.22 で選択する。危険セルへの移動は他の選択肢より価値が約 1.5 低く、ソフトマックスによって確率は約 e^{-6} になるため、参照方策がそこへ入ることはめったにない。次のコードセルでは、配列 $P[s, a, s']$ と $r^*[s, a]$ を `d2l.TabularMDP`（14.1 節）として構築し、Q^* を `d2l.value_iteration` の実行とさらに 1 回の `backup`（14.2 節）の適用によって計算して、参照方策を構成する。

```
gamma = 0.95                                  # a 4 x 4 gridworld
P = np.zeros((16, 4, 16))
for s, a in np.ndindex(16, 4):
    x, y = s % 4, s // 4
    dx, dy = [(-1, 0), (0, 1), (1, 0), (0, -1)][a]   # left, down, right, up
    P[s, a, min(3, max(0, y + dy)) * 4 + min(3, max(0, x + dx))] = 1.0
P[15], P[15, :, 15] = 0.0, 1.0                # the goal absorbs
bonus = np.full(16, -0.04)                    # every step costs a little
bonus[[5, 6]], bonus[15] = -1.5, 1.0          # a hazard lane; the goal pays
r_true = np.einsum('sat,t->sa', P, bonus)     # r*(s, a): the entered bonus
r_true[15] = 0.0
mdp = d2l.TabularMDP(P, r_true, gamma)
Q_true = mdp.backup(d2l.value_iteration(mdp, 200)[-1])
pi_ref = np.exp(Q_true / 0.25)                # competent but hedging
pi_ref /= pi_ref.sum(1, keepdims=True)
```

15.3.1.2 選好とブラッドリー・テリーモデル

1 回の比較からわかるのは、評価者が 2 つの軌跡のどちらを選好（preference）したかだけであり、どれほどの差があったかはわからない。ブラッドリー・テリーモデル（Bradley-Terry model）は、このような回答を統計モデルへと変換する (Bradley and Terry 1952)。このモデルは各候補にスカラースコアとして軌跡の収益 $r(\tau)$（報酬 r のもとでの値）を割り当て、比較をスコア差の

ノイズ付き観測として扱う。

$$P(\tau \succ \tau') = \sigma\big(r(\tau) - r(\tau')\big), \qquad \sigma(u) = \frac{1}{1 + e^{-u}}. \tag{15.25}$$

したがって、収益の等しい 2 つの軌跡はそれぞれ確率 1/2 で選好され、収益に 1 単位の差があると優れた方が 4 回中約 3 回勝つことになる。

グリッドワールドでは、収益は報酬に関して線形である。軌跡の割引訪問回数を $\mathbf{c}(\tau)$ とし、状態行動対ごとに 1 つの要素を持たせる。すなわち、要素 (s, a) は γ^t の和であり、軌跡がステップ t において行動 a を状態 s でとるたびにこの項が加わる。すると $r(\tau) = \mathbf{c}(\tau)^\top r$（ここで報酬は 64 要素に平坦化されている）となり、Equation 15.25 の当てはめは特徴量を $\mathbf{c}(\tau) - \mathbf{c}(\tau')$、重みを報酬とするロジスティック回帰となる。`sample_traj` は $\mathbf{c}(\tau)$ を π_{ref} の 1 本の軌跡（ゴールに到達するか 40 ステップ後に終了する）に対して返し、`X` はそれを 1000 本積み重ねる。ランダムな 6000 組のペアに対し、Equation 15.25 の確率（r^* のもとでの値）に従うコイントスでラベルを付け、対数尤度に対する勾配上昇法（gradient ascent）によって $\hat{r}$ を当てはめる。このとき、データが少ない対でも当てはめが定まるよう、小さな重み減衰（weight decay）0.02 を加える。`episodes` は各対が出現するエピソード数を数え、`core` は少なくとも 50 エピソードに出現する対を表し、`into` は危険レーンへ上および左右から入る 4 つの移動を並べる。

```
rng = np.random.default_rng(0)
sigmoid = lambda u: 1 / (1 + np.exp(-u))

def sample_traj(mdp, pi, rng, T=40):
    """Discounted visit counts c(tau) of one episode from state 0."""
    s, c = 0, np.zeros((16, 4))
    for t in range(T):
        a = rng.choice(4, p=pi[s])
        c[s, a] += mdp.gamma ** t             # the return is c(tau) . r
        s = mdp.P[s, a].argmax()
        if s == 15:
            break
    return c.ravel()

X = np.stack([sample_traj(mdp, pi_ref, rng) for _ in range(1000)])
R = X @ r_true.ravel()                        # true returns; the fit never sees them
i, j = rng.integers(0, 1000, (2, 6000))
y = (rng.random(6000) < sigmoid(R[i] - R[j])).astype(float)
D = X[i] - X[j]
lr, wd, num_steps = 0.5, 0.02, 4000           # ascent step, weight decay
w = np.zeros(64)
for _ in range(num_steps):                    # logistic regression on (D, y)
    w += lr * (D.T @ (y - sigmoid(D @ w)) / len(y) - wd * w)
r_hat = w.reshape(16, 4)
episodes = (X > 0).sum(0).reshape(16, 4)      # episodes containing each pair
core = episodes >= 50
into = ([1, 4, 2, 7], [1, 2, 1, 0])           # into cells 5, 6 from above, sides
print(f'never visited: {(episodes == 0).sum()} pairs; '
      f'in at least 50 episodes: {core.sum()} pairs')
print(f'rms fit error on those {core.sum()}: '
      f'{np.sqrt(((r_hat - r_true)[core] ** 2).mean()):.3f}')
print(f'moves into the hazards: in {episodes[into]} episodes, '
```

```
        f'fitted {np.round(r_hat[into], 2)}')
```

```
never visited: 9 pairs; in at least 50 episodes: 46 pairs
rms fit error on those 46: 0.193
moves into the hazards: in [3 4 3 2] episodes, fitted [-0.03 -0.05 -0.02 -0.02]
```

この当てはめは、core に含まれる 46 個の対において二乗平均平方根（RMS）誤差 0.19 を持つ。9 個の対は一度も出現しない。すべてのエピソードがすでに終了しているゴールでの 4 つの行動と、Figure 15.6 でバツ印が付いた 5 つの対である。これらの当てはめ報酬は初期値の 0 のままである。危険レーンへ上および左右から入る移動は、1,000 エピソード中それぞれ 2〜4 回しか生じず、下から入る 2 つの移動は一度も生じない。これでは少なすぎて、重みの初期値であり重み減衰が引き戻す先でもある 0 から、当てはめ報酬を大きく動かすことはできない（真のコストはいずれも 1.5 である）。これに対し、ニューラル報酬モデルであれば、良くも悪くもその帰納バイアス（inductive bias）に従って外挿することになる（演習問題 4）。

15.3.1.3 シェイピングに対する不変性

RMS 誤差 0.19 は、ステップコスト 0.04 と比べると大きく見えるが、その大部分はいかなる比較からも観測できないものである。14.1 節で導入したポテンシャルに基づくシェイピング（potential-based shaping）は、$\gamma\Phi(s') - \Phi(s)$ を s から s' への各移動の報酬に加える（ここで Φ は状態上の関数であり、ゴールにおいて $\Phi = 0$ を満たす）。そして 14.2 節では、これが最適方策を変化させないことを証明した。軌跡に沿うとこれらの項は望遠鏡和となって相殺されるため、状態 0 からゴールに到達するすべての軌跡は、収益が同じ量 $-\Phi(0)$ だけ変化する。すべての比較は状態 0 からの 2 本の軌跡を対象とするため、このシフトは $r(\tau) - r(\tau')$ において打ち消し合い、Equation 15.25 では検出できない。（10 本に 1 本の割合で 40 ステップ後に打ち切られる軌跡には末尾の項 $\gamma^{40}\Phi(s_{40})$ が残るが、小さすぎて Φ を特定するには至らない。）したがって、比較データはシェイピングの自由度を除いてしか報酬を決定できず、区別できない報酬の中では重み減衰によって重みが最小のものが選ばれる。

次のコードセルでは、最小二乗法（least squares）の意味で core 上の当てはめ誤差を最もよく説明するシェイピング項のポテンシャルを求める。F の (s, a) 行に Φ を掛けると $\gamma \sum_{s'} P(s' \mid s, a)\,\Phi(s') - \Phi(s)$ が得られ、最後の列を除くと $\Phi(15) = 0$ と固定される。

```
err = (r_hat - r_true)[core]
F = (gamma * P - np.eye(16)[:, None, :])[core][:, :15]  # shaping per unit of Phi
Phi = np.linalg.lstsq(F, err, rcond=None)[0]
print(f'rms error on core {np.sqrt((err ** 2).mean()):.3f}, after removing '
      f'the best shaping term {np.sqrt(((err - F @ Phi) ** 2).mean()):.3f}')
print(f'Phi(0) = {Phi[0]:.2f}; moves 13 -> 14 -> goal: fitted '
      f'{r_hat[13, 2]:+.2f}, {r_hat[14, 2]:+.2f}, true '
      f'{r_true[13, 2]:+.2f}, {r_true[14, 2]:+.2f}')
```

```
rms error on core 0.193, after removing the best shaping term 0.032
Phi(0) = 0.78; moves 13 -> 14 -> goal: fitted -0.08, +0.09, true -0.04, +1.00
```

シェイピング項を取り除くと RMS 誤差は 0.03 になる。つまり、シェイピングを除けば、データが存在する領域での当てはめは良好である。ポテンシャルは開始位置での約 0.8 からゴール手前での約 0.9 まで上昇し、core のどの対も触れない危険セルでは不定のまま残る。このようなポテンシャルは、ゴールへ入る移動から約 0.9 を引き、それ以外の各移動から約 $(1-\gamma) \times 0.8 = 0.04$ を引く。出力された報酬が示すのもまさにこの値であり、最後の移動の報酬は 0.09（本来は 1）となり、通常の移動のコストは 0.08（本来は 0.04）となる。どちらの報酬も状態 0 からのあらゆ

る軌跡のペアに同じ順序を付けるが、当てはめ報酬の方が重みは小さい。そしてこの当てはめ報酬のもとでは、0 付近と推定された危険セルへの移動の方が、通常の移動よりも低コストになってしまう。

言語モデルも同様の不変性に直面する。そこでの比較は、同一のプロンプトに対する 2 つの応答 y と y'（プロンプトを x とする）を対象とし、報酬 $r(x,y)+f(x)$（ここで f はプロンプトの任意の関数）はすべての比較を $r(x,y)$ とまったく同じように予測するため、学習された報酬はプロンプトごとの相対的な値としてしか識別されない（演習問題 1）。方策勾配法（policy gradient）はこれに影響されない。報酬からプロンプトの関数を引く操作はベースラインに相当し、期待勾配を変化させないからである（14.8 節）。GRPO のグループ平均もそのようなベースラインの一種だが、応答自身の報酬を含んでいる点が異なり、その節で示したとおり、期待更新量が $(K-1)/K$ 倍（K 個の応答からなるグループの場合）にスケールされる。

15.3.2 当てはめ報酬の最適化

ここで、エージェントが当てはめ報酬に対してとるのと同じ手順、すなわちそれを最大化する方策を求めることを行う。ダイナミクスが既知であれば、価値反復法（value iteration）によってその方策が求まる。`plan` は与えられた報酬に対する最適な決定論的方策をワンホットな行を持つ行列として返す。`ret` は、状態 0 からの厳密な方策の収益を 14.2 節の `policy_value` を通じて返す（これは線形ベルマン方程式 $V^\pi=(I-\gamma P_\pi)^{-1}r_\pi$ を 1 ステップで解く）。真の報酬と当てはめ報酬のそれぞれについて計画を立て、両方の方策を両方の報酬のもとで採点してみよう。

```
def plan(r):
    """Optimal deterministic policy (one-hot rows) for reward r."""
    m = d2l.TabularMDP(P, r, gamma)
    return np.eye(4)[m.backup(d2l.value_iteration(m, 200)[-1]).argmax(1)]

def ret(pi, r):
    """Exact return of policy pi from state 0 under reward r."""
    return d2l.TabularMDP(P, r, gamma).policy_value(pi)[0]

for name, r in (('true reward  ', r_true), ('fitted reward', r_hat)):
    pi = plan(r)
    print(f'plan on {name}: fitted return {ret(pi, r_hat):+.2f}, '
          f'true return {ret(pi, r_true):+.2f}')
```

```
plan on true reward  : fitted return -0.24, true return +0.59
plan on fitted reward: fitted return -0.03, true return -2.11
```

当てはめ報酬のもとでは、$\hat{r}$ 上の計画のスコアは -0.03 となり、真の最適方策の -0.24 を上回る。真の最適方策の当てはめスコアは真のスコアより 0.83 低く、シェイピングのオフセット $\Phi(0)=0.78$ に近いため、当てはめスコアは差にしか意味がない。一方、真の報酬のもとでは、2 つの計画のスコアはそれぞれ -2.11 と $+0.59$ となる。$\hat{r}$ 上の計画は $0\to1\to5\to6\to7\to11\to15$ と進み、2 つの危険セルをまっすぐ突っ切る。これは外周に沿う安全な経路と同じ最短ステップ数でありながら、$\hat{r}$ のもとではより低コストなのである。価値反復法は与えられた報酬を忠実に最適化したのであり、失敗の原因は参照方策のデータ外における報酬にある。異なる乱数シードで抽出した 24 個の選好データセットにわたって調べると、$\hat{r}$ 上の計画は 19 個で危険セルを横断し、さらに 4 個でゴールに一度も到達しない。真の最適方策が得られたのは 1 個のデータセットだけであった。

この失敗には名前が付いている。グッドハートの法則（Goodhart's law）は、指標が目標になった途端に指標としての価値を失うと述べるものである。強化学習（reinforcement learning）で

は、指定された報酬や当てはめ報酬のもとで高いスコアを出しながら本来の目的を損なう方策は、報酬を**ハック**する（報酬ハッキング）と言い、その多数の事例がまとめられている (Krakovna et al. 2020)。14.2 節で扱った接近ボーナス（最適方策を永遠に周回させてしまった例）は、人手で書いた報酬の誤りによって同じ仕組みが働いたものであった。ここでの誤りは統計的なものであり、データが手薄な場所に潜んでいる。

15.3.3 KL 正則化目的関数

当てはめ報酬が信頼に値するのは、参照方策が訪れた領域だけである。参照方策の近くにとどまる方策はデータが存在する領域にとどまるため、そこから離れることに対してコストを課す。ここで自然な尺度となるのが KL ダイバージェンスである。これは方策自身のもとでの期待値であるため、その方策自身のサンプルから推定でき、参照方策が決してとらない行動をとる方策に対しては無限大となる。有限個の行動集合上の報酬 $r(a)$、参照方策 π_{ref}、および係数 $\beta > 0$ を持つ 1 回の意思決定に対して、**KL 正則化目的関数**（KL-regularized objective）は次の式で定義される。

$$\max_{\pi} \; \mathbb{E}_{a\sim\pi}\big[r(a)\big] - \beta\, D_{\mathrm{KL}}\big(\pi \,\|\, \pi_{\text{ref}}\big). \tag{15.26}$$

第 1 項は期待報酬（expected reward）であり、第 2 項は参照方策からのダイバージェンス 1 nat ごとに報酬を β 単位だけ差し引く。係数 β はペナルティの重みであり、参照方策は固定されたままである。

15.3.3.1 1 回の意思決定に対する閉形式の最適解

命題。$\beta > 0$ に対して、目的関数 Equation 15.26 を最大化する唯一の方策は次で与えられる。

$$\pi^{\star}(a) = \frac{1}{Z}\,\pi_{\text{ref}}(a)\, e^{r(a)/\beta}, \qquad Z = \sum_{a} \pi_{\text{ref}}(a)\, e^{r(a)/\beta}, \tag{15.27}$$

また、その最大値は $\beta \log Z$ である。

証明。構成より $r(a) = \beta \log\big(\pi^{\star}(a) Z / \pi_{\text{ref}}(a)\big)$ が成り立つ。これを目的関数に代入すると、任意の方策 π に対して次を得る。

$$\begin{aligned} \mathbb{E}_{\pi}[r] - \beta D_{\mathrm{KL}}(\pi\|\pi_{\text{ref}}) &= \sum_{a} \pi(a)\, \beta \log \frac{\pi^{\star}(a)\, Z}{\pi(a)} \\ &= \beta \log Z - \beta\, D_{\mathrm{KL}}(\pi\|\pi^{\star}). \end{aligned}$$

第 37 章で証明されているギブスの不等式（Gibbs' inequality）より、残ったダイバージェンスは非負であり、$\pi = \pi^{\star}$ のとき、かつそのときに限り 0 となる。■

このように、最適解は参照方策に指数ティルト（exponential tilt）$e^{r/\beta}$ を掛けて再正規化したものとなり、その達成値は対数分配関数（log-partition function）となる。Figure 15.7 に 5 つの行動に対するこの積を示す。そこでの参照方策は意図的に報酬に関して単調でないように選ばれており、2 番目の行動における落ち込みは穏やかなティルトのもとでも残存する。すなわち、最適解は行動を報酬だけでランキングするのではなく、参照方策を再重み付けするのである。

2 つの境界ケースを補足しておく。第 1 に、最適解は参照方策の台（support）を受け継ぐ。$\pi_{\text{ref}}(a) = 0$ であれば、a に正の確率質量を置くいかなる方策もダイバージェンスが無限大とな

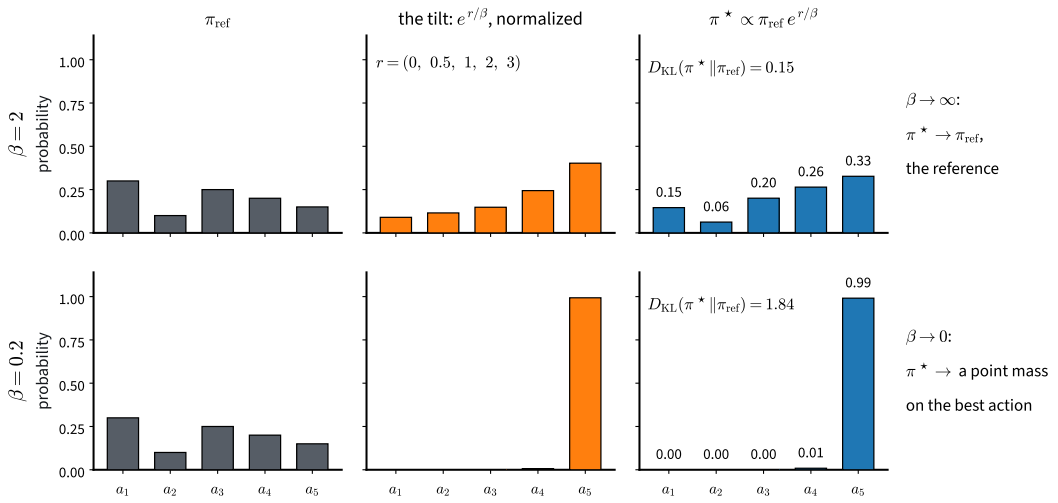


Figure 15.7: 積として表されるペナルティ付き目的関数の最適解。各行は参照方策 $\pi_{\text{ref}} = (0.30,\ 0.10,\ 0.25,\ 0.20,\ 0.15)$ にティルト $e^{r/\beta}$（報酬は $r = (0,\ 0.5,\ 1,\ 2,\ 3)$）を掛けて再正規化し、$\pi^\star$ を得る。上段は $\beta = 2$、下段は $\beta = 0.2$ を用いている。参照方策における a_2 での落ち込みは、上段では $\pi^\star$ にも残るが、下段では押し流される。余白の注記は 2 つの極限を示しており、$\beta \to \infty$ では参照方策に戻り、$\beta \to 0$ では最良の行動上の点質量となる。

り、Equation 15.27 は報酬がいくらであっても a に確率 0 を割り当てる。第 2 に、$\beta \to 0$ とすると、最適解は π_{ref} の台の中で最良の行動に集中する。最良の行動が複数同点で並ぶ場合、極限はその同点集合上で参照方策を再正規化した分布となる。

閉形式を一切知らない反復法で Equation 15.26 を解くことにより、この命題を数値的に確認しよう。`solve_kl` は指数勾配上昇法（exponentiated gradient ascent）を実行する。すなわち、現在の方策にステップサイズと勾配 $r - \beta(\log(\pi/\pi_{\text{ref}}) + 1)$ の積の指数を掛け、単体（simplex）上に再正規化する。$+1$ はすべての行動で共通であり正規化で相殺されるため、コードでは省略している。

```
p_ref = np.array([0.30, 0.10, 0.25, 0.20, 0.15])
r = np.array([0.0, 0.5, 1.0, 2.0, 3.0])

def solve_kl(r, p, beta, steps=3000, eta=0.3):
    pi = np.full_like(p, 1 / len(p))
    for _ in range(steps):                    # exponentiated gradient ascent
        pi = pi * np.exp(eta * (r - beta * np.log(pi / p)))
        pi /= pi.sum()
    return pi

for beta in (5.0, 2.0, 0.5, 0.2):
    pi = solve_kl(r, p_ref, beta)
    print(f'beta = {beta:3}: pi_beta = {np.round(pi, 3)}, '
          f'reward {pi @ r:.2f}, KL {(pi * np.log(pi / p_ref)).sum():.2f}')
```

```
beta = 5.0: pi_beta = [0.233 0.086 0.237 0.232 0.212], reward 1.38, KL 0.02
beta = 2.0: pi_beta = [0.146 0.062 0.2   0.264 0.327], reward 1.74, KL 0.15
beta = 0.5: pi_beta = [0.004 0.004 0.025 0.148 0.819], reward 2.78, KL 1.26
```

```
beta = 0.2: pi_beta = [0.    0.    0.    0.009 0.991], reward 2.99, KL 1.84
```

β が 5 から 0.2 へ下がるにつれて、期待報酬は 1.38 から 2.99 へ上昇し、ダイバージェンスは 0.02 から 1.84 nat へ増加する。Figure 15.7 の 2 つの行は、$\beta = 2$（ダイバージェンス 0.15 nat）および $\beta = 0.2$ における最適解である。ここで、収束した各反復解を Equation 15.27 と比較し、達成された目的関数値を $\beta \log Z$ と比較する。`pi_star` は閉形式を評価する際、オーバーフローを防ぐために指数の内部で最大報酬を差し引く（このシフトは正規化で打ち消し合う）。

```
def pi_star(r, p, beta):
    star = p * np.exp((r - r.max()) / beta)    # eq_kl_optimum, stabilized
    return star / star.sum()

gap = max(np.abs(solve_kl(r, p_ref, b) - pi_star(r, p_ref, b)).max()
          for b in (5.0, 2.0, 0.5, 0.2))
pi2 = solve_kl(r, p_ref, 2.0)
print(f'largest gap to the closed form over the ladder: {gap:.1e}')
print(f'objective at beta = 2: numerical '
      f'{pi2 @ r - 2.0 * (pi2 * np.log(pi2 / p_ref)).sum():.6f}, '
      f'beta log Z = {2.0 * np.log(p_ref @ np.exp(r / 2.0)):.6f}')
```

```
largest gap to the closed form over the ladder: 1.4e-16
objective at beta = 2: numerical 1.442004, beta log Z = 1.442004
```

反復解はすべての β において浮動小数点の精度まで閉形式と一致し、目的関数値も小数点以下 6 桁まで $\beta \log Z$ と一致する。

15.3.3.2 閉形式から導かれる 4 つの帰結

ペナルティがなければ、最適解は点質量となる。 Equation 15.27 において $\beta \to 0$ とすると、全確率質量が最高報酬の行動（同点がある場合は同点集合上での参照方策の分布）へと集中する。これは正則化なしの問題であり、その解は点質量（point mass）、すなわち argmax である。この主張は最適解に関するものであり、特定の訓練実行の経過について述べたものではない。15.2 節における PPO 方策のエントロピーが訓練中に低下したのは、軽くエントロピー正則化された PPO の目的関数が持つほぼ決定論的な最適解へと向かう動きと整合している。より大きな重みを持つペナルティを課せば、確率的方策が最適となる。

係数は参照方策と argmax の間を補間する。 β が非常に大きい場合と非常に小さい場合における閉形式を確認してみよう。

```
for beta in (100.0, 0.01):
    print(f'beta = {beta:6}: pi_star = {np.round(pi_star(r, p_ref, beta), 3)}')
```

```
beta =  100.0: pi_star = [0.297 0.099 0.25  0.202 0.153]
beta =   0.01: pi_star = [0. 0. 0. 0. 1.]
```

大きい β は参照方策を再現し、小さい β は貪欲（greedy）になり、その中間のあらゆる方策はある β の値に対する厳密な最適解となる。

一様な参照方策はペナルティをエントロピーボーナスに変える。 一様な参照方策に対しては $D_{\mathrm{KL}}(\pi \| \text{uniform}) = \log|\mathcal{A}| - H(\pi)$ となるため、ペナルティはエントロピーに対するボーナス $\beta H(\pi)$ と定数の和になり、Equation 15.26 は**最大エントロピー**（maximum-entropy）目的関数となる。閉形式は報酬のソフトマックスへと特殊化される。

```
boltzmann = np.exp(r / 0.5) / np.exp(r / 0.5).sum()
print(np.allclose(pi_star(r, np.full(5, 0.2), 0.5), boltzmann))
```

```
True
```

これは 14.7 節のソフトマックス方策と同じ形である。したがって、14.4 節のボルツマン探索（Boltzmann exploration）規則は、β を温度とみなせば、エントロピーボーナスを持つ 1 ステップ目的関数の厳密な最適解にほかならない。

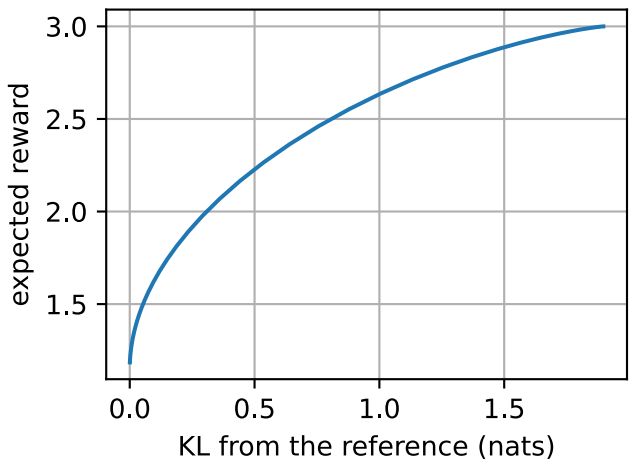
最適解は事後分布である。閉形式 Equation 15.27 はベイズの規則と同じ形をしている。すなわち、π_{ref} が事前分布（prior）、$e^{r/\beta}$ が非正規化尤度、$\pi^\star$ が事後分布（posterior）、Z が正規化定数である。推論としての制御（control-as-inference）の諸手法はこの恒等式を土台とし、報酬を「最適性」と呼ばれる事象の対数尤度、正則化方策最適化を変分推論（variational inference）として解釈する (Levine 2018; Korbak et al. 2022)。

15.3.3.3 1 回の意思決定におけるフロンティア

各 β は期待報酬と参照方策からのダイバージェンスを持つ 1 つの最適解を与える。β を掃引すると、各ダイバージェンスで達成可能な最良の報酬、すなわち**報酬・ダイバージェンスフロンティア**（reward-divergence frontier）が描かれる。

```
front = np.stack([pi_star(r, p_ref, b) for b in np.logspace(1.5, -1.5, 61)])
kl_f = (front * np.log(front / p_ref)).sum(1)
print(f'KL 0 pays {p_ref @ r:.2f}; the frontier ends at KL '
      f'{np.log(1 / p_ref[r.argmax()]):.2f}, reward {r.max():.0f}')
d2l.plot(kl_f, front @ r, 'KL from the reference (nats)', 'expected reward')
```

```
KL 0 pays 1.15; the frontier ends at KL 1.90, reward 3
```



このフロンティアは参照方策の平均報酬 1.15 から始まり、最良の行動上の点質量（ダイバージェンスは $\log(1/0.15) = 1.90$ nat）で終わる。この曲線は単調増加かつ凹であり、傾きは β（β に対する最適解での値）となるため（演習問題 5）、ダイバージェンスを 1 nat 増やすごとに得られる報酬の増分は逓減していく。Figure 15.7 の 2 つの行はこの曲線上の 2 点である。Best-of-N サンプリング（参照方策から N 個の回答を抽出し、最も報酬の高いものを残す手法）は、一切の訓練なしにこれと同じ軸に沿って移動する。その参照方策からのダイバージェンスは高々

$\log N - (N-1)/N$ であり、約 1.8 nat（$N = 16$ のとき）である (Gao et al. 2023)。グリッドワールドでも同様の曲線をたどるが、そこでは最適化対象の報酬ではなく真の報酬で採点することになる。

15.3.3.4 信頼領域とペナルティの比較

信頼領域と固定参照ペナルティはいずれも KL ダイバージェンスを用いるが、その役割は異なる。

> **KL ダイバージェンスの 2 つの用途。** **信頼領域**（15.2 節）は**直前の反復解**からのダイバージェンスを測る。各更新を制限して局所近似を保護するものであり、その参照先は更新のたびに移動する。一方、**固定参照ペナルティ**（本節）は目的関数の一部であり、最適解を Equation 15.27 へと移動させる。言語モデルの PPO 流のファインチューニングではこの両方が使われる。すなわち、安定した更新のためにサンプリング方策に対するクリップ付き代理目的関数を用い、最終的な方策を制御するために凍結した初期モデルに対する報酬ペナルティを用いる (Ouyang et al. 2022)。

閉形式を見れば、信頼領域だけを用いた場合にどこへ行き着くかがわかる。各更新が、直前の反復解からのダイバージェンスに重み $\tilde{\beta}$ を置いた信頼領域問題 $\pi_{k+1} = \operatorname{argmax}_{\pi} \mathbb{E}_{\pi}[r] - \tilde{\beta} D_{\mathrm{KL}}(\pi \| \pi_k)$ を、$\pi_0 = \pi_{\mathrm{ref}}$ から出発して厳密に解くものとする。命題より、各ステップは方策に $e^{r/\tilde{\beta}}$ を掛けるため、k ステップ後には次のようになる。

$$\pi_k(a) = \frac{\pi_{\mathrm{ref}}(a)\, e^{k\, r(a)/\tilde{\beta}}}{\sum_{a'} \pi_{\mathrm{ref}}(a')\, e^{k\, r(a')/\tilde{\beta}}}, \tag{15.28}$$

これは係数 $\tilde{\beta}/k$ を持つ閉形式にほかならない。k が大きくなるにつれてこの係数は 0 に近づき、反復解は最良の行動上の点質量へと収束する。信頼領域が定めるのは経路であって到達点ではなく、固定された参照方策だけが反復解の収束先を変えるのである（演習問題 6）。

15.3.3.5 ダイバージェンスの向き

ペナルティ $D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}})$ は 37.3.2 節の分類における**逆方向**（reverse）のダイバージェンスであり、π のもとでの期待値であるため方策自身のサンプルから推定できる。閉形式を見ると、ある行動 a の報酬が最良の行動より $\Delta(a) > 0$ だけ劣るときにどう扱われるかがわかる。最良の行動に対する相対比で見ると、$\pi^\star$ のもとでのその確率は参照確率の $e^{-\Delta(a)/\beta}$ 倍となる。一方、順方向（forward）のペナルティ $D_{\mathrm{KL}}(\pi_{\mathrm{ref}} \| \pi)$ は、参照方策がとる行動を 1 つでも切り捨てる方策に対して無限大となり、その最適解はすべての行動に対して参照確率の $\beta/\Delta(a)$ のオーダーの下限を保つ（演習問題 7）。どちらの向きでも、任意の $\beta > 0$ に対して、参照方策がとるすべての行動に正の確率が保たれる。両者の違いは、β が小さくなるときに劣った行動の確率がどれだけ速く減衰するかにあり、逆方向のペナルティでは $1/\beta$ に関して指数関数的に減衰するのに対し、順方向のペナルティでは β に関して線形にしか減衰しない。

15.3.4 ソフトバックアップ

グリッドワールドは 1 回きりの意思決定ではない。マルコフ決定過程（Markov decision process、MDP）では毎ステップでペナルティが課され、状態 0 からの目的関数は次のようになる。

$$\mathbb{E}_{\pi}\Big[\sum_{t \geq 0} \gamma^t \big(r(s_t, a_t) - \beta\, D_{\mathrm{KL}}\big(\pi(\cdot \mid s_t) \| \pi_{\mathrm{ref}}(\cdot \mid s_t)\big)\big)\Big]. \tag{15.29}$$

各状態において、方策は先ほど解いたような 1 回の意思決定問題に直面する。問題は、その意思決定をどの報酬によってティルトすべきかである。

15.3.4.1 終端の 1 ステップ前

エピソードの最後の意思決定では、行動の価値はその報酬そのものであり、先の命題がそのまま当てはまる。すなわち、そこでの最適解は参照方策を $e^{r/\beta}$ でティルトしたものであり、その状態の価値は $\beta \log Z$ である。その 1 ステップ前では、ある行動の価値はその行動の報酬と遷移先の状態の価値との和になる。その遷移先の価値とは、**そこで**の最適方策がペナルティのもとで達成する値 $\beta \log Z$ であり、そこで利用可能な最良の報酬よりも小さい。最終ステップにおける最適方策は、その最良の行動をある確率でしかとらない上、ペナルティも課されるからである。

2 ステップの例を考えると、この違いが具体的になる。Figure 15.7 の 5 行動の意思決定（$\beta = 2$）をエピソードの最終ステップとしよう。その価値は $\beta \log Z = 1.44$ であり、最良の報酬 3 よりも小さい。その 1 ステップ前で、エージェントは即座には報酬を生まないこの意思決定へ進むか、それとも確実に報酬 2 を得て終了するかのいずれかを選ぶものとし、参照方策はそれぞれを確率 1/2 で選び、割引はないものとする。次のコードセルでは、最初の意思決定をソフト価値（soft value）1.44 でティルトした場合と最良の報酬 3 でティルトした場合のそれぞれについて、2 ステップ方策のペナルティ付き目的関数を評価する。

```
beta, stop = 2.0, 2.0                          # stopping pays 2 for sure
V_next = beta * np.log(p_ref @ np.exp(r / beta))   # value of the last step
for name, q in (('soft value', V_next), ('best reward', r.max())):
    pi0 = pi_star(np.array([q, stop]), np.array([0.5, 0.5]), beta)
    J = pi0 @ [V_next, stop] - beta * (pi0 * np.log(pi0 / 0.5)).sum()
    print(f'tilt by the {name} {q:.2f}: pi(enter) = {pi0[0]:.2f}, '
          f'objective {J:.3f}')
Z0 = 0.5 * np.exp(V_next / beta) + 0.5 * np.exp(stop / beta)
print(f'beta log Z of the first decision: {beta * np.log(Z0):.3f}')
```

```
tilt by the soft value 1.44: pi(enter) = 0.43, objective 1.740
tilt by the best reward 3.00: pi(enter) = 0.62, objective 1.592
beta log Z of the first decision: 1.740
```

ソフト価値でティルトすると、最初の意思決定における最適値 $\beta \log Z$ が達成される。一方、最良の報酬でティルトすると、2 番目の意思決定を 3 と評価してしまうが、そこでのペナルティ付き方策は実際には 1.44 しか得られないため、進入しすぎて約 0.15 を失う。ティルトには、後続のあらゆる報酬**と**後続のあらゆるペナルティの両方を織り込んだ価値を用いなければならない。

15.3.4.2 ソフト価値反復法

この価値をすべての状態で書き下すと一般形が得られる。ペナルティ付き目的関数のもとでの行動価値を $Q_\beta(s, a)$（行動 a を状態 s でとったときの価値）、状態価値を $V_\beta(s)$（状態 s の価値）とする。すべての状態において、報酬の役割に $Q_\beta(s, \cdot)$ を置いて先の命題を適用すると、次式が得られる。

$$
\begin{aligned}
V_\beta(s) &= \beta \log \sum_a \pi_{\text{ref}}(a \mid s)\, e^{Q_\beta(s,a)/\beta}, \\
Q_\beta(s, a) &= r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V_\beta(s'),
\end{aligned}
\tag{15.30}
$$

そして最適方策は $\pi_\beta(a \mid s) \propto \pi_{\text{ref}}(a \mid s)\, e^{Q_\beta(s,a)/\beta}$ となる。これは 14.2 節のベルマン最適方程式において、行動に関する最大値を参照方策で重み付けした log-sum-exp、すなわち $\beta \to 0$ で参照方策の台上のハード最大値へと収束する**ソフト最大値**（soft maximum）に置き換えたものである。14.3 節で示したように、行動価値がノイズを含む推定値であるときハード最大値は上方に統計的バイアスを持つが、log-sum-exp も凸関数であるため、程度は穏やかながら同様に上方へのバイアスを持つ。参照方策が一様である場合、これは最大エントロピー強化学習におけるソフト価値反復法（soft value iteration）に一致する。

価値反復法は、ハードな方程式を $V \leftarrow \max_a \text{backup}(V)$ という掃引によって $V = 0$ から解いた。ソフト最大値を用いて同じ掃引を行えば Equation 15.30 が解ける。ハードな掃引が縮小写像であることの証明（14.2.2.3 節）で使ったのは、最大値が持つ 2 つの性質だけであった。単調であることと、すべての要素に定数を加えると最大値にもその定数が加わることである。ソフト最大値はこの両方を満たすため、ソフトな掃引もまた γ 縮小写像となる（任意の $\beta > 0$ および任意の参照方策に対して成立する。演習問題 3）。`soft_max` は行動価値の配列の各行について、数値安定性のために行の最大値を差し引いてからソフト最大値を計算し、ティルトされた方策とあわせて返す。`soft_v` は `backup` に続けて `soft_max` を適用する掃引を繰り返す。

```
def soft_max(Q, p, beta):
    """Row-wise beta log sum_a p(a) exp(Q(a) / beta), and the tilted policy."""
    m = Q.max(1, keepdims=True)                # subtracted for stability
    tilt = p * np.exp((Q - m) / beta)
    V = m[:, 0] + beta * np.log(tilt.sum(1))
    return V, tilt / tilt.sum(1, keepdims=True)

def soft_v(mdp, p, beta, num_iters=400):
    """Soft value iteration V <- soft_max(backup(V)); returns V and pi_beta."""
    V = np.zeros(mdp.num_states)
    for _ in range(num_iters):
        V, pi = soft_max(mdp.backup(V), p, beta)
    return V, pi
```

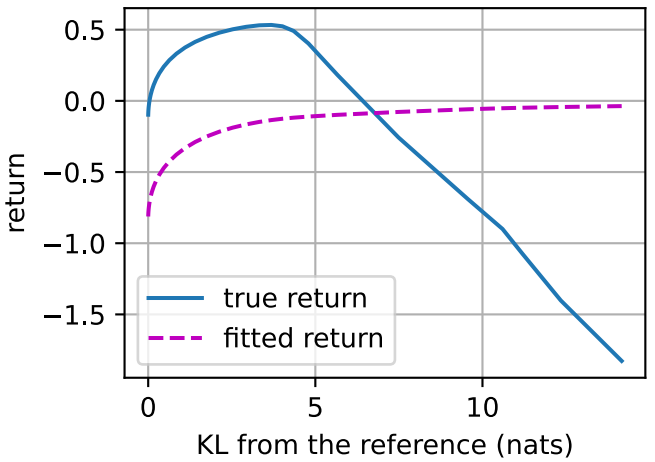
15.3.4.3 KL 予算に対する真の収益

ここでグリッドワールド上のフロンティアをたどるが、1 つ変更を加える。オプティマイザには当てはめ報酬 $\hat{r}$ を見せ、採点は真の報酬で行うのである。36 個の β の値（10 から $10^{-2.5}$ まで）のそれぞれについて、`soft_v` で厳密なペナルティ付き最適方策 π_β を $\hat{r}$ に対して計算し、その真の収益、当てはめ収益、および参照方策からのダイバージェンスを記録する。ダイバージェンスとしては、MDP の目的関数のペナルティ項である $\mathbb{E}_{\pi_\beta}\big[\sum_t \gamma^t D_{\text{KL}}(\pi_\beta(\cdot \mid s_t) \| \pi_{\text{ref}}(\cdot \mid s_t))\big]$（開始状態は状態 0）を用いる。これは、$\pi_\beta$ と π_{ref} が生成する軌跡の分布間の KL ダイバージェンス（各ステップ後に確率 $1-\gamma$ でもエピソードが終了するとした場合）に等しい（演習問題 2）。コードでは、状態ごとのダイバージェンス `kl` に、状態 0 からの各状態への期待割引訪問回数 `visits`（π_β の遷移行列を用いた 1 回の線形方程式の求解で得られる）を重みとして掛ける。あわせて、掃引全体にわたる Equation 15.30 の最大残差も記録する。

```
m_hat = d2l.TabularMDP(P, r_hat, gamma)
kls, true_ret, fit_ret, resid = [], [], [], 0.0
for beta in np.logspace(1, -2.5, 36):
    V, pi = soft_v(m_hat, pi_ref, beta)
    V_new, _ = soft_max(m_hat.backup(V), pi_ref, beta)
    resid = max(resid, np.abs(V_new - V).max())
```

```
        kl = (pi * np.log(np.where(pi > 0, pi / pi_ref, 1.0))).sum(1)
        P_pi = np.einsum('sa,sat->st', pi, P)
        visits = np.linalg.solve(np.eye(16) - gamma * P_pi.T, np.eye(16)[0])
        kls.append(visits @ kl)                    # discounted KL from state 0
        true_ret.append(ret(pi, r_true))
        fit_ret.append(ret(pi, r_hat))
    k = int(np.argmax(true_ret))
    print(f'largest soft Bellman residual: {resid:.0e}')
    print(f'reference: true return {ret(pi_ref, r_true):+.2f}; beta = 10: '
          f'KL {kls[0]:.3f}, true return {true_ret[0]:+.2f}')
    print(f'peak: true return {true_ret[k]:+.2f} at KL {kls[k]:.1f}; '
          f'smallest beta: {true_ret[-1]:+.2f} at KL {kls[-1]:.1f}')
    d2l.plot(kls, [true_ret, fit_ret], 'KL from the reference (nats)', 'return',
             legend=['true return', 'fitted return'])
```

```
largest soft Bellman residual: 0e+00
reference: true return -0.12; beta = 10: KL 0.001, true return -0.10
peak: true return +0.53 at KL 3.7; smallest beta: -1.83 at KL 14.2
```



プロット上の各点はペナルティ付き目的関数の厳密な最適解であり、最大のベルマン残差は 10^{-10} 未満である。当てはめ収益は一貫して上昇し、厳密な最適解である限り必ずそうなる。実際、$\beta_1 < \beta_2$ とし、それぞれの最適解が当てはめ収益 J_1, J_2 とダイバージェンス D_1, D_2 を持つとすると、最適性より $J_1 - \beta_1 D_1 \geq J_2 - \beta_1 D_2$ および $J_2 - \beta_2 D_2 \geq J_1 - \beta_2 D_1$ が成り立つ。この 2 式を足し合わせると $(\beta_2 - \beta_1)(D_1 - D_2) \geq 0$ となるため $D_1 \geq D_2$ が従い、これを第 1 の不等式に用いれば $J_1 \geq J_2$ が得られる。

真の収益は異なる振る舞いを見せる。参照方策の -0.12 から上昇して $+0.53$（3.7 nat の地点）に達し、真の最適値 $+0.59$ に迫るが、その後は方策が危険レーンに集中するにつれて -1.83 へと低下する。当てはめ報酬への適度な最適化は真の収益を改善するが、強く押しすぎるとデータが一度も計測しなかった移動を悪用してしまう。24 個の選好データセットにわたって調べると、ピークは数 nat の地点で約 $+0.5$ から最適値 $+0.59$ の間に位置し、そのうち 21 個ではピークの後に真の収益が低下する。残る 3 個では掃引範囲内で低下しない。1 個では当てはめ最適方策が真の最適方策と一致しており、2 個では安全な経路と危険な経路の当てはめ価値がほぼ同点となるため、ペナルティ付き最適方策が $\beta = 10^{-2.5}$ まで安全な経路を維持するからである。ピークが参照方策からどれだけ離れた位置に現れ、その後の低下がどれほど深くなるかは、比較データ

が危険セルをどれだけカバーしているかと、そのコストの大きさとに依存する。Gao et al. (2023) は、学習された報酬モデルに対して最適化しホールドアウトされた報酬モデルで採点した言語モデルにおいて、これと同じ上昇と低下を測定している。

オプティマイザに見えるのは当てはめ曲線だけであり、その曲線は単調に上昇するため、真の収益のピークはオプティマイザからは見えない。したがって実務では、訓練に使った報酬モデル以外の何か（ホールドアウトされた比較データ、人間の判定、あるいはチェッカー）に対して方策を評価することで β を選ぶ。参照方策が一様であれば同じペナルティはエントロピーボーナスとなり、15.5 節ではこれを最大化する連続行動向けのオフポリシー（off-policy）アクター・クリティックを構築する。そのクリティックは log-sum-exp バックアップを反復するのではなく、エントロピーボーナス付きの報酬のもとで自身の方策を評価するが、先の命題の証明にある恒等式により、方策がティルトされたものであれば両者は一致する。また 15.8 節では、閉形式 Equation 15.27 を第 14 章の検証器（verifier）タスクに適用し、保険をかけた回答（hedged answer）が割に合わなくなる β を予測する。ここまでの手法はすべて、現在の方策から新たに得たサンプルか既知のモデルから学習してきた。続く 15.4 節では、14.3 節の Q テーブルが行ったように、任意の方策から蓄積された遷移データからネットワークが行動価値を学習できるかを問う。

15.3.5 まとめ

比較データに当てはめた報酬は、比較データが存在する領域でのみ正確である。共通の開始状態から出発する軌跡間の比較では報酬のポテンシャルシェイピングを見分けることができず、データが一度もカバーしない状態行動対は当てはめの初期値のままである。オプティマイザはそうした隙を突く。グリッドワールドでは、当てはめ報酬上の計画は参照方策がめったに入らなかった危険セルを通り抜ける。参照方策からの KL ダイバージェンスにペナルティを科すと目的関数が変わり、その厳密な最適解は参照方策を $e^{r/\beta}$ でティルトしたものとなって、最適値は $\beta \log Z$ となる。参照方策が一様であれば、このペナルティはエントロピーボーナスになる。MDP においてティルトはソフト行動価値を用い、これは最大値の代わりに log-sum-exp を持つベルマン方程式を満たす。ペナルティの重みを掃引すると、当てはめ収益は単調に上昇する一方、真の収益は適度なダイバージェンスでピークに達した後に低下するため、重みはオプティマイザが参照しない評価によって選ばなければならない。

15.3.6 演習問題

1. **識別可能性とスケール**。ブラッドリー・テリーモデル Equation 15.25 を、同一のプロンプトに対する 2 つの応答 y と y'（プロンプトは x）の比較データに当てはめる。
   1. $r(x,y)$ を $r(x,y)+f(x)$（f はプロンプトの任意の関数）に置き換えても、このようなあらゆるデータセットの尤度が変化しないことを示せ。逆に、x に対する 2 つの応答がデータセット内で比較されているときに限り、両者を辺で結ぶグラフを考える。比較されたすべての対に同じ選好確率を割り当てる 2 つの報酬は、このグラフの各連結成分上で定数差しか持たないことを示せ。1 つのプロンプトに対するグラフが複数の連結成分を持つとき、何が識別されるか。
   2. アノテータがある一貫性パラメータのもとで $P(y \succ y') = \sigma\big(\omega\,(r^*(x,y) - r^*(x,y'))\big)$（ただし $\omega > 0$）に従ってラベル付けし、無限のデータに対して最尤法により Equation 15.25 を当てはめるものとする。この当てはめはどの報酬を復元するか。
   3. 当てはめ報酬に対する最適解 Equation 15.27（係数は β）が、r^* に対する係数 β/ω での最適解と一致することを示せ。あるアノテータ集団のラベルに当てはめた報酬モデルに対して β を調整したとする。同じ β を、より一貫性の低い集団（すなわち ω がより小さい集団）のラベルに当てはめた報酬モデルに用いた場合、どれほど強いペナルティとして働くか。

2. [code] **軌跡のダイバージェンス。** $p_\pi(\tau)$ を、方策 π が状態 0 から出発して固定ホライズンの軌跡 $\tau = (s_0, a_0, s_1, a_1, \dots, s_T)$（長さ T）を生成する確率とする。
   1. 次式を示せ。
$$D_{\mathrm{KL}}(p_\pi \| p_{\pi_{\mathrm{ref}}}) = \mathbb{E}_\pi\Big[\sum_{t=0}^{T-1} D_{\mathrm{KL}}(\pi(\cdot \mid s_t) \| \pi_{\mathrm{ref}}(\cdot \mid s_t))\Big].$$
   $p_\pi(\tau)$ のどの因子が相殺されるか。また、それゆえに遷移のモデルを持たなくても π の軌跡からこのダイバージェンスを推定できるのはなぜか。
   2. 代わりに、軌跡が行動とは独立に各ステップの後で確率 $1-\gamma$ で終了するものとする。2 つの軌跡分布間のダイバージェンスが、15.3.4.3 節の掃引で `kls` に `visits @ kl` として追加した状態ごとのダイバージェンスの割引和になることを示せ。
   3. その掃引において真の収益がピークとなる方策 π_β をとれ。$\mathbf{c}(\tau)^\top \log(\pi_\beta/\pi_{\mathrm{ref}})$（対数は要素ごとにとり $\mathbf{c}(\tau)$ と同様に平坦化する）を、`sample_traj(mdp, pi, rng, T=400)` から生成した 2000 本の軌跡にわたって平均し、ピーク時点で出力されたダイバージェンス 3.7 nat と比較せよ。軌跡ごとの値の中には負になるものがある。これがギブスの不等式と矛盾しないのはなぜか。
3. [code] **湖の上のソフトバックアップ。** 14.2.2.3 節の証明では、最大値の 1 つの性質 $|\max_a f(a) - \max_a g(a)| \le \max_a |f(a) - g(a)|$ を用いた。
   1. $m_\beta(f) = \beta \log \sum_a \pi_{\mathrm{ref}}(a)\, e^{f(a)/\beta}$ とおく。m_β が単調であり、かつ $m_\beta(f + c) = m_\beta(f) + c$ が任意の定数 c に対して成り立つことを示し、そこから m_β が最大値と同じ性質を持つことを導け。以上より、バックアップ Equation 15.30 が γ 縮小写像であることを結論せよ（任意の $\beta > 0$ および任意の参照方策に対して成立する）。
   2. `soft_v` を 14.2 節の滑りやすい湖（`d2l.TabularMDP` として構築したもの）に一様な参照方策のもとで適用せよ。$V_\beta(s_0)$ を β に対して 10^{-4} から 10^{2} までの対数グリッド上でプロットせよ。この曲線は両端において、14.2 節で計算したどの 2 つの値に近づくか。正則化目的関数を用いて、この曲線が単調になる理由を説明せよ。
   3. およそどの β において、$V_\beta(s_0)$ は 2 つの極限の中間に位置するか。この係数を、湖における最適行動価値 $Q^*(s_0, a)$ どうしの差と比較せよ。
4. [extended] **ニューラル報酬モデル。**● `r_hat` の線形当てはめではデータのない対が 0 付近にとどまるのに対し、ニューラル報酬モデルはその帰納バイアスに従って外挿する。64 個の重み `w` を、64 個の ReLU ユニットからなる隠れ層を 2 層持ち状態行動対ごとに 1 つの出力を出す多層パーセプトロンの出力に置き換え、ゴールの行は 0 に固定せよ。これを同じ `D` と `y` 上で、ブラッドリー・テリーモデルの負の対数尤度を用い、学習率 0.01 の Adam によってフルバッチで 2000 ステップ訓練せよ。
   1. 入力を状態と行動のワンホット符号としたモデルと、入力を $[0, 1]$ にスケーリングした状態の 2 つのグリッド座標および行動のワンホット符号としたモデルを、それぞれ 4 つのシードで訓練せよ。各モデルについて、`core` 上の RMS 誤差、危険セルへ入る移動 `into` の当てはめ報酬、および `plan` の真の収益を報告せよ。
   2. 各当てはめ報酬に対して 15.3.4.3 節の掃引を繰り返せ。ペナルティを必要とするのはどちらの表現か。入力のどの性質がその違いを説明するか。
5. **フロンティアの傾き。**● $u = 1/\beta$ とおき、$R(u)$ と $D(u)$ をそれぞれ期待報酬とダイバージェンス $D_{\mathrm{KL}}(\pi^\star \| \pi_{\mathrm{ref}})$（最適解 Equation 15.27 における値）とする。
   1. $R(u) = \frac{d}{du} \log Z$ および $R'(u) = \mathrm{Var}_{\pi^\star}(r)$ を示せ。次に、命題の最適値 $\beta \log Z$ を用いて $D'(u) = u\, \mathrm{Var}_{\pi^\star}(r)$ を示せ。
   2. `front` と `kl_f` が描くフロンティアは、傾きが β（点 $\pi^\star$ において）であり、r が π_{ref} の台上で定数でない限り単調増加かつ凹であることを結論せよ。
   3. $\pi^\star$ が制約付き問題 $\max_\pi \mathbb{E}_\pi[r]$（制約条件は $D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}}) \le D(u)$）の解でもあることを示せ。$\beta$ はこの制約付き問題におけるどの量に対応するか。

4. Best-of-N サンプリングは、N 個の独立標本を π_{ref} から抽出し、最も報酬の高いものを残す。参照方策のもとで報酬が連続分布を持つとき、Best-of-N 分布の参照方策からのダイバージェンスが $\log N - (N-1)/N$ であることを示せ。報酬が互いに異なり単調増加である 5 行動バンディット `p_ref`、`r` に対しては、Best-of-N は行動 a を確率 $F(a)^N - F(a-1)^N$ で選ぶ（ここで $F(a)$ は a 以下の行動に対する参照確率の和）。$N = 1, \dots, 16$ に対してその期待報酬をダイバージェンスに対して `front` および `kl_f` とともにプロットせよ。フロンティアにどれだけ近づくか。また、そのダイバージェンスは $\log N - (N-1)/N$ 以下にとどまるか。

6. **移動する参照方策。**●15.2 節の信頼領域は直前の反復解からのダイバージェンスを測る。本節では、理想化された更新 $\pi_{k+1} = \text{argmax}_\pi \; \mathbb{E}_\pi[r] - \tilde{\beta}\, D_{\text{KL}}(\pi \| \pi_k)$ を $\pi_0 = \pi_{\text{ref}}$ から開始すると argmax に収束することを示した。
   1. ここで PPO 流のファインチューニングと同様に Equation 15.26 の固定参照ペナルティを加え、π_{k+1} が次式を最大化するものとする。

$$\mathbb{E}_\pi[r] - \beta\, D_{\text{KL}}(\pi \| \pi_{\text{ref}}) - \tilde{\beta}\, D_{\text{KL}}(\pi \| \pi_k). \tag{15.31}$$

   定数差を除いて次式が成り立つことを示せ。

$$\log \pi_{k+1} = \frac{\beta \log \pi_{\text{ref}} + \tilde{\beta} \log \pi_k + r}{\beta + \tilde{\beta}}, \tag{15.32}$$

   さらに π_k の極限を求めよ。極限を決定するのはどちらの係数か。また、どのくらいの速さで極限に近づくか。
   2. 小問 a の漸化式が、適切なステップサイズを用いた `solve_kl` の更新と一致することを示し、そのステップサイズを β と $\tilde{\beta}$ で表せ。`solve_kl` が収束するステップサイズのうち、信頼領域としての解釈を持たないものはどれか。

7. [code] **順方向 KL ペナルティ。**●15.3.3.5 節では Equation 15.26 の逆方向ダイバージェンスを順方向ダイバージェンスと対比した。ペナルティを順方向ダイバージェンスに置き換え、次の問題を考える。

$$\max_\pi \; \mathbb{E}_{a\sim\pi}\big[r(a)\big] - \beta\, D_{\text{KL}}(\pi_{\text{ref}} \| \pi). \tag{15.33}$$

   1. 乗数 m を制約 $\sum_a \pi(a) = 1$ に対して導入し、最適解が $\pi(a) = \beta\, \pi_{\text{ref}}(a)/(m - r(a))$ となることを示せ（ここで $m > \max_a r(a)$ はこれらの確率の和を 1 にする唯一の値である）。
   2. 準最適な行動の報酬ギャップを $\Delta(a) = \max_{a'} r(a') - r(a) > 0$（行動を a とする）とおく。$\beta \to 0$ のとき、$\pi(a)/\pi_{\text{ref}}(a)$ は順方向ペナルティのもとでどのくらいの速さで 0 に収束するか。また、逆方向ペナルティのもとではどうか。
   3. 5 行動バンディット `p_ref`、`r` に対し、m に関する二分法によって順方向ペナルティの最適解を計算せよ。それぞれのペナルティについて、β のうち最適解の期待報酬が 2.9 となる値を求め、2 つの最適解を行動ごとに比較せよ。順方向ペナルティのもとでより大きな確率が割り当てられるのはどの行動か。また、これは 15.3.3.5 節の内容とどのように整合するか。

15.4 深層 Q ネットワーク

14.3 節では、エージェントはいかなる振る舞いから生成された遷移からでも、1 遷移ずつ行動価値（action value）のテーブルを学習した。14.9 節では、方策（policy）のテーブルをネットワークに置き換え、CartPole 上で訓練した。Q 学習（Q-learning）でも同じことをするのは、一見す

ると 1 行の変更で済むように思える。ネットワークに 2 つの行動価値を出力させ、更新則をそのまま保てばよい。方策勾配（policy gradient）では、導出のどこにもテーブルという形式を使っていなかったため、この置き換えに代償は伴わなかった。しかし Q 学習では、テーブルがターゲット $r + \gamma \max_{a'} Q(s', a')$ の内部に入っている。テーブルがネットワークになると、更新のたびに予測とともにターゲットも動き、1 回の更新が一度も訪れていない状態の価値まで変化させ、表形式の手法を収束させていた縮小写像の議論はもはや通用しなくなる。ネットワークになっても、任意の遷移から学習できるというテーブルの特権を保てるだろうか。

2013 年、DeepMind の研究グループがこの置き換えを機能させた。過去の遷移の蓄積を用いて訓練することで、ネットワークが画面のピクセルから 7 種類の Atari ゲームをプレイできるようになったのである (Mnih et al. 2013)。研究グループはこれを深層 Q ネットワーク（deep Q-network、DQN）と名付けた。2015 年までには、同じネットワークとハイパーパラメータにターゲット計算用の凍結コピーを加えることで、49 種類のゲームをプレイし、その半数以上で人間のテスターと同程度の性能を達成した (Mnih et al. 2015)。学習を安定させた 2 つの仕組み、すなわち遷移の蓄積と凍結コピーは、それ以降のほぼすべてのオフポリシー深層強化学習手法に組み込まれている。またこの 2 つは、前節までの手法では得られなかった利点もエージェントにもたらす。アクター・クリティックは各バッチを 1 回しか使わず、PPO でも数エポックしか使わなかったのに対し、DQN は保存したすべての遷移を使って何度も繰り返し訓練する。

本節ではまず、サンプリングノイズが一切ないにもかかわらず重みが際限なく増大する 7 状態の例を用いて、素朴な置き換えが失敗し得る理由を示す。次に CartPole 上で DQN を構築し、凍結コピーを取り除いてそれが何を防いでいるかを確認したうえで、最後にターゲット内の最大値が生み出すバイアスと、それを軽減する小さな変更を扱う。15.3 節ではこの最大値をソフト最大値に置き換えたが、本節ではハード最大値のままとする。

```
import copy
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

ここでも割引率 0.99 の CartPole を用いるが、方策の改善に伴ってエピソード長が変化するため、訓練予算は 50,000 環境ステップとして指定する。エージェントは 2 環境ステップごとに、保存された 128 個の遷移からなるミニバッチ上で 1 回の勾配ステップを実行する。Q ネットワークは状態を 128 ユニットの 2 つの隠れ層に通して行動ごとに 1 つの価値へ写像し、貪欲行動はその argmax とする。探索（exploration）は ϵ 貪欲とし、最初の 10,000 ステップにわたって ϵ を 1 から 0.05 まで線形に減少させる。これには `linear_schedule`（14.3 節）を用いる。`make_env` は環境を構築し、そのオプション引数 `copies` は各行動を複数回提示する（本節の最後の実験で用いるものであり、それまでは 1 である）。

```
gamma, num_env_steps, num_seeds = 0.99, 50_000, 3
buffer_size, batch_size, lr = 200_000, 128, 1e-3
train_freq, sync_every, warmup, grad_clip = 2, 250, 1_000, 10.0
epsilon = d2l.linear_schedule(1.0, 0.05, 10_000)

def make_env(copies=1):
    """CartPole; with copies > 1 each of its two actions is offered `copies`
    times (action a moves the cart as action a % 2), which leaves Q* as is."""
    env = gym.make('CartPole-v1')
    return env if copies == 1 else gym.wrappers.TransformAction(
        env, lambda a: a % 2, gym.spaces.Discrete(2 * copies))
```

次に、`make_qnet(seed, copies=1)` はシードを固定した 2 つの隠れ層を持つ多層パーセプトロン（multilayer perceptron、MLP）を構築し、4 次元の CartPole の状態を $2 \times$ copies 個の行動価値へ写像する。

```
def make_qnet(seed, copies=1):
    """A seeded Q-network with one output per action."""
    torch.manual_seed(seed)
    return nn.Sequential(nn.Linear(4, 128), nn.ReLU(),
                         nn.Linear(128, 128), nn.ReLU(),
                         nn.Linear(128, 2 * copies))
```

15.4.1 不安定性の要因

15.4.1.1 半勾配更新

14.3 節のテーブルをネットワーク Q_w（パラメータは w）に置き換えても、更新則自体は変わらず、ブートストラップされたターゲットに向かう半勾配（semi-gradient）ステップのままである。遷移 (s, a, r, s') をサンプリングし、次のターゲットを作る。

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right) \max_{a'} Q_w(s', a'), \tag{15.34}$$

そのうえで y を固定し、$Q_w(s, a)$ を y に近づけるオプティマイザのステップを 1 回実行する。指示関数は、s' でエピソードが終了するときにブートストラップを取り除く。各ステップは、固定された数値に向かう通常の回帰（regression）の 1 ステップである。しかし、この数値が w とともに動くため、ステップの列はいかなる固定された目的関数に対する降下にもなっていない。これは 14.9 節で、あらゆるブートストラップターゲットについて確認したとおりである。したがって、損失が下がっていることはネットワークが現在のターゲットに適合していることを示すにすぎず、ターゲット自体が正しいことを示すわけではない。

テーブルの場合、期待更新は価値反復法（value iteration）の緩和ステップであり、ベルマンバックアップは縮小写像となる（14.2 節）。すなわち、全状態にわたる最大誤差を係数 γ 倍に縮小する。一方、ネットワークはバックアップ後のあらゆる関数を表現できるわけではないため、各回帰は関数クラスの中でターゲットに最も近い関数を返す。この「近さ」は訓練データ上の最小二乗法（least squares）で測られ、各状態での誤差はデータがその状態を訪れる頻度に比例して重み付けされる。線形特徴量の場合、この当てはめは射影（projection）になる。バックアップが最大誤差を縮小するのに対し、当てはめは訪問頻度で重み付けされた平均誤差を縮小する。これら 2 つの尺度は一致しないため、重み付き誤差を減らす当てはめが、めったに訪れない状態での誤差をかえって拡大してしまうことがあり、次のバックアップがその誤差をそこへ至る状態のターゲットへと伝播させる。データが評価対象の方策自身から得られている場合は、2 つの尺度が十分に整合するため線形特徴量では収束する (Tsitsiklis and Van Roy 1997)。しかし別の方策からのデータを用いると、線形特徴量であっても発散し得る。

15.4.1.2 相関したデータと移動するターゲット

実用上、2 つの結合（coupling）がネットワークの場合をテーブルの場合よりも難しくする。1 つ目はデータにおける結合である。カートとポールは 1 ステップごとにわずかしか動かないため、1 つのエピソード内の連続する遷移はほぼ同一であり、到着するストリームをそのまま使って訓練すると状態空間（state space）の狭い領域が何度も連続して更新される。テーブルは訪れた要素しか変更しないため、相関（correlation）があっても学習が遅くなるだけで済む。しかしネット

ワークはステップごとに多くの状態での予測を変化させるため（Figure 14.22）、似通った更新が続くと他の領域で学習した内容を上書きしてしまうことがある。

2 つ目の結合はターゲットにおける結合である。ターゲット y は Q_w を含むため、オプティマイザのステップごとに予測と回帰先の数値の両方が変化する。15.1 節のクリティックにも同じ結合があったが、更新のたびに現在の方策による新しい遷移で再当てはめしていたため、方策が訪れる状態での誤差はすぐに表面化して修正された。

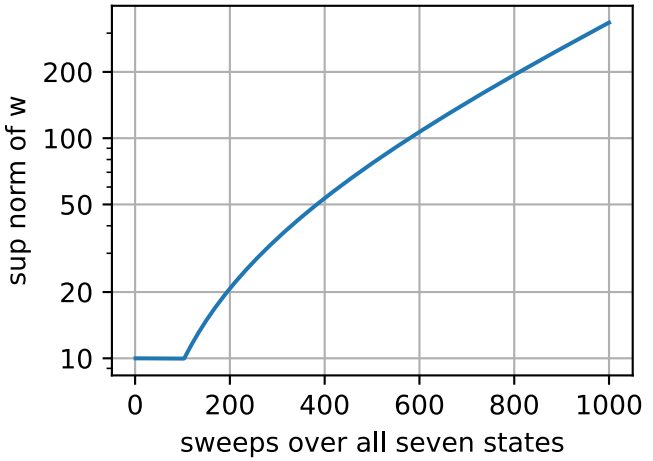
15.4.1.3 死の三要素と Baird の反例

ネットワーク版の Q 学習では、**関数近似**（function approximation）、**ブートストラップ**、そして**オフポリシーデータ**（価値を学習したい方策以外の方策によって収集されたデータ）という 3 つの要素が同時に重なる。Figure 15.8 は、本章と前章のアルゴリズムをそれらが含む要素ごとに配置したものである。2 つの要素が重なる各領域には、それぞれの条件のもとで収束する手法が存在する。表形式の Q 学習はオフポリシーデータからブートストラップし、14.3 節のステップサイズ条件のもとで収束する。最も単純な形のアクター・クリティックにおけるクリティックである線形時間差分学習は、関数近似を用いてブートストラップし、データが評価対象の方策から得られる場合に線形特徴量のもとで収束する。行動クローニング（behavior cloning）はブートストラップを行わずにオフポリシーデータへネットワークを当てはめる手法であり、固定されたラベルへの教師あり回帰にほかならない。なお、これらは十分条件（sufficient condition）であって、2 要素の組み合わせ一般に関する定理ではない。非線形な時間差分学習はオンポリシーであっても発散し得る (Tsitsiklis and Van Roy 1997)。これら 3 つの要素がすべて揃う状況は**死の三要素**（deadly triad）と呼ばれる (Sutton and Barto 2018; Hasselt et al. 2018)。

Baird の反例は、7 状態の線形問題においてこの三要素だけを取り出して示す (Baird 1995)。すべての報酬は 0 であるため、真の価値関数（value function）は 0 であり、$w = 0$ によって厳密に表現される。計算には期待時間差分更新を用いるため、サンプリングノイズは存在しない。常に状態 7 へ遷移する方策を評価しながら、7 つの状態をオフポリシー分布として一様に重み付けする。

```
w = np.ones(8)                          # eight weights for seven states
w[6] = 10.0                             # the classic starting point
Phi = np.zeros((7, 8))                  # linear values v = Phi w
Phi[:6, :6], Phi[:6, 7] = 2 * np.eye(6), 1.0   # v(i) = 2 w_i + w_8
Phi[6, 6], Phi[6, 7] = 1.0, 2.0                # v(7) = w_7 + 2 w_8
sup = []
for _ in range(1000):
    v = Phi @ w                         # every reward is 0
    delta = gamma * v[6] - v            # the next state is always state 7
    w += 0.01 * (delta[:, None] * Phi).mean(0)   # uniform state weighting
    sup.append(np.abs(w).max())
print(f'sup norm of w after 0, 500, 1000 sweeps: 10, {sup[499]:.0f}, '
      f'{sup[-1]:.0f}')
print(f'the value the weights claim for state 7: {(Phi @ w)[6]:.0f}, '
      f'true value 0')
d2l.plot(np.arange(1, 1001), sup, 'sweeps over all seven states',
         'sup norm of w', yscale='log')
```

```
sup norm of w after 0, 500, 1000 sweeps: 10, 77, 335
the value the weights claim for state 7: 677, true value 0
```



過渡期を過ぎると、重みは 1 スイープあたり一定の倍率に収束するペースで増大する。1,000 スイープ後には重みの上限ノルム（sup norm）は 10 から 335 へと増大し、真の価値が 0 であるにもかかわらず、重みが主張する状態 7 の価値は 677 になる。ここには標本誤差（sampling error）は一切関与しておらず、ステップサイズを小さくしても発散が遅くなるだけで解消はされない。真の価値関数は表現可能であるため、原因は近似誤差でもない。オフポリシーの重み付けのもとでは、期待更新に重みが増大していく方向が存在するのである (Sutton and Barto 2018)。代わりに評価対象の方策自身の訪問頻度で状態を重み付けすると発散は止まるが、その重み付けが一度も訪れない状態では到達する価値が誤った値になる（演習問題 1）。

経験再生（experience replay）とターゲットネットワークは、三要素のどれかを取り除くわけではない。時間的相関を低減しターゲットの変化を遅らせることで実用上の訓練を大幅に安定させるが、一般的な収束保証（convergence guarantee）を回復させるものではない。

15.4.2 経験再生とターゲットネットワーク

DQN はこれら 2 つの結合に対し、それぞれ 1 つずつの仕組みで対処する。Figure 15.9 に両者の組み合わせを示す。

15.4.2.1 経験再生

相関したデータへの処方箋は**経験再生バッファ**（experience replay buffer）である (Lin 1992)。すべての遷移を大きな配列に保存し、訓練用の各ミニバッチをその配列から一様に抽出する。するとミニバッチには異なる時刻に収集された遷移が混ざり合い、各遷移が多くの更新で再利用される。(s, a, r, s') に対するターゲットが a を選んだ方策に依存しないため、Q 学習ではこの再利用が許される。一方で経験再生には欠点もある。保存された遷移の大半は古い方策によって収集されたものであるため、現在の行動方策が該当する状態を再び訪れるまで、ターゲット内の誤差が多くの更新にわたって残り続ける可能性がある。

`ReplayBuffer` は、遷移の 5 つのフィールド（状態、行動、報酬、次状態、および終端を記録するフラグ `term`）を事前確保した配列に保持する。`add` は次の空きスロットに遷移を書き込み（配列が満杯になると最も古いものを上書きする）、`sample` は `batch_size` 個のスロットを一様に抽出して `Batch`（14.7 節）として返す。

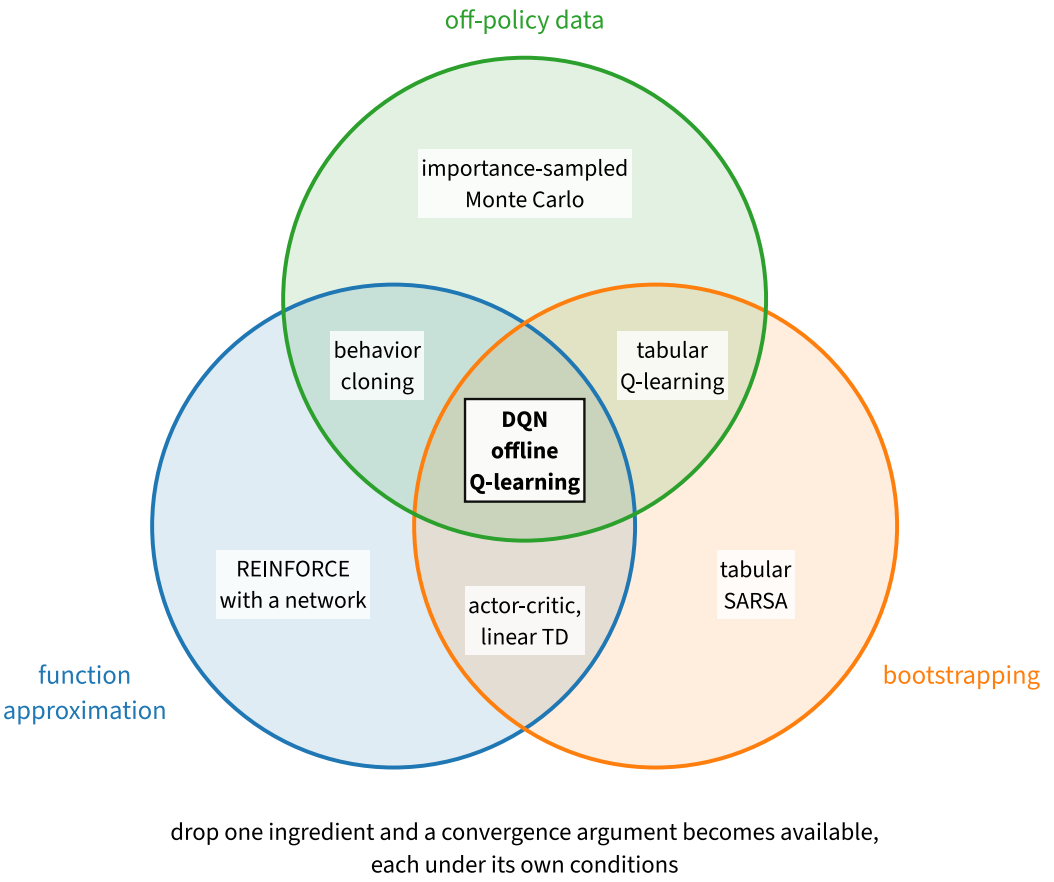


Figure 15.8: 死の三要素。各円が 1 つの要素を表し、すべての領域にこれら 2 つの章で扱ったアルゴリズムが入る。3 つの要素すべてが重なる中央に DQN とオフライン Q 学習が位置し、Baird の反例はそこで発散が起こり得ることを示す。いずれか 1 つの要素を取り除けば、それぞれのステップサイズ、カバレッジ、表現の条件のもとで収束の議論が成り立つようになる。死の三要素は警戒すべきパターンであり、必要十分条件の定理ではない。

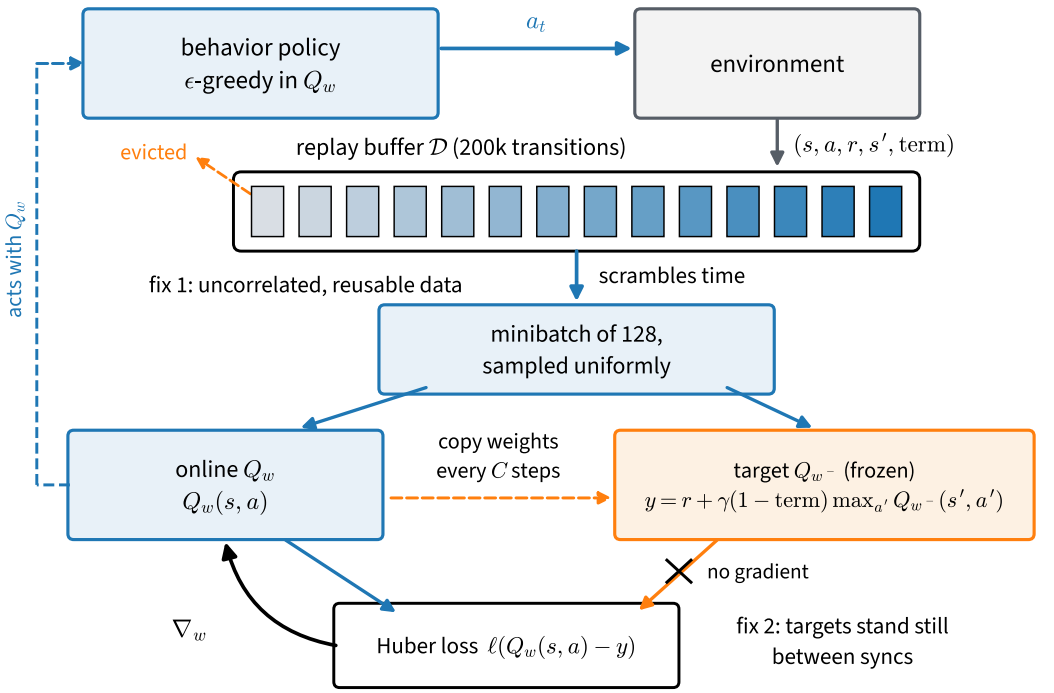


Figure 15.9: データフローとして見た深層 Q 学習。行動方策はオンラインネットワークを用いて行動し、遷移はリプレイバッファに入る。一様にサンプリングされたミニバッチが時間順をかき混ぜることで、データはほぼ独立かつ再利用可能になる。回帰目標は C ステップごとに更新されるネットワークの凍結コピーによって計算されるため、同期と同期の間、ターゲットは静止する。勾配はただ 1 つの経路を通ってオンラインネットワークへ流れ、凍結コピーからの印付きの辺には勾配が流れない。保存された各遷移はフラグ term を持ち、ターゲット内のマスク $1 - \text{term}$ は真の終端でのみブートストラップを止める。図のバッファは満杯で最も古い要素を追い出している長期実行の定常状態を描いているが、本節の実験では 50,000 ステップの予算が 200,000 スロットのリングを満たすことはないため、追い出しは生じない。

```
class ReplayBuffer:
    """A ring of transitions in preallocated numpy; sample() returns a Batch."""
    def __init__(self, capacity, obs_dim):
        self.obs = np.zeros((capacity, obs_dim), np.float32)
        self.act = np.zeros(capacity, np.int64)
        self.rew = np.zeros(capacity, np.float32)
        self.next_obs = np.zeros((capacity, obs_dim), np.float32)
        self.term = np.zeros(capacity, np.float32)
        self.capacity, self.size, self.ptr = capacity, 0, 0

    def add(self, obs, act, rew, next_obs, term):
        i = self.ptr
        self.obs[i], self.act[i], self.rew[i] = obs, act, rew
        self.next_obs[i], self.term[i] = next_obs, term
        self.ptr, self.size = (i + 1) % self.capacity, min(self.size + 1,
                                                           self.capacity)

    def __len__(self):
        return self.size

    def sample(self, batch_size, rng):
        """batch_size transitions drawn uniformly, each a one-step
        fragment: ep_ends marks every transition as its own episode."""
        i = rng.integers(self.size, size=batch_size)
        return d2l.Batch(self.obs[i], self.act[i], self.rew[i],
                         self.next_obs[i], self.term[i],
                         np.arange(1, batch_size + 1))
```

本節での容量は 200,000 遷移であり、50,000 ステップの予算より大きいため、遷移の追い出しは生じない（より小さなバッファについては演習問題 4 で調べる）。一様抽出にはエピソードの順序がないため、`sample` はすべての遷移を長さ 1 のエピソード断片として扱う。このようなバッチ上では `reward_to_go` は報酬そのものになり、また 14.9 節の `truncation_bootstrap` は $\gamma(1-\text{term})\,\widehat{V}(s')$（与えられた価値推定 $\widehat{V}$ に対する項）を加えるため、両者の和は 15.1 節の `td_target` が計算する 1 ステップターゲットになる。経験再生では 1 ステップターゲットしか残らないため、1 ステップのブートストラップと組み合わせて用いられる。n ステップターゲットには連続した遷移が必要になるため、バッファ側で順序を保持しなければならない（演習問題 6）。

15.4.2.2 ターゲットネットワーク

移動するターゲットへの処方箋は**ターゲットネットワーク**である (Mnih et al. 2015)。オンラインネットワークの凍結コピー Q_{w^-} がすべてのブートストラップを次式のように計算する。

$$y = r + \gamma\,\big(1 - \mathbf{1}(s' \text{ terminal})\big)\max_{a'} Q_{w^-}(s', a'), \tag{15.35}$$

そしてこのコピーは、$w^- \leftarrow w$ として C 環境ステップごと（ここでは 250 ステップごと）にオンライン重みと同期（synchronization）される。同期と同期の間はターゲットが静止するため、オンラインネットワークは通常の回帰問題を解くことになる。予測がターゲットへフィードバックされるのは、勾配ステップごとではなく同期ごとに 1 回だけになる。15.1 節のクリティックは各バッチが新鮮だったため最新の重みからブートストラップしていたが、経験再生はその新鮮さを失わせるため、DQN は変化の緩やかな回帰問題と引き換えに数百ステップ古いターゲットを受け入れる。

さらに 2 つの制御によって 1 ステップの大きさを抑える。フーバー損失（Huber loss）は残差（residual）が 1 未満では二次、それ以上では線形であるため、予測に関する導関数（derivative）の大きさは決して 1 を超えない。ただし全勾配（full gradient）には $\nabla_w Q_w(s,a)$ も含まれ、これはフーバー損失では抑えられないため、更新時にはパラメータ勾配のノルムを 10 でクリップする（元論文ではなく Stable-Baselines3 のデフォルト値である (Raffin et al. 2021)）。コードでは、`q_values` が勾配を記録せずに NumPy の状態上でネットワークを順伝播させ、`fit_q` が $Q_w(s,a)$ を固定されたターゲット y に近づけるクリップ付きフーバー回帰ステップを 1 回実行し、`q_step` が DQN の更新を担う。そのターゲットは、凍結コピーのブートストラップ $\max_{a'} Q_{w^-}(s',a')$ を用いた `batch.td_target` である。

```
def q_values(qnet, obs):
    """The acting forward: numpy in, numpy out, no gradient recorded."""
    with torch.inference_mode():
        return qnet(torch.as_tensor(obs)).numpy()

def fit_q(qnet, opt, batch, y):
    """One clipped Huber regression step of Q_w(s, a) toward the fixed y."""
    pred = qnet(torch.as_tensor(batch.obs)).gather(
        -1, torch.as_tensor(batch.act)[:, None]).squeeze(-1)
    loss = nn.functional.huber_loss(pred, torch.as_tensor(y))
    opt.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(qnet.parameters(), grad_clip)
    opt.step()

def q_step(qnet, target, opt, batch):
    """The DQN update: the critic's own td_target, bootstrapped by the copy."""
    fit_q(qnet, opt, batch, batch.td_target(
        lambda s: q_values(target, s).max(-1), gamma))
```

15.4.2.3 訓練ループ

フレームワーク間で異なる学習器の構成要素は `make_learner` から得られる。具体的には、凍結コピー、オプティマイザ、`update`（サンプリングされたバッチ上で更新則 `step` を 1 ステップ実行する）、および `sync`（オンライン重みを凍結コピーへコピーする）である。

```
def make_learner(qnet, step):
    """DQN's moving parts: the frozen copy, Adam, the update and the sync."""
    target = copy.deepcopy(qnet)
    opt = torch.optim.Adam(qnet.parameters(), lr=lr)
    update = lambda b: step(qnet, target, opt, b)
    sync = lambda: target.load_state_dict(qnet.state_dict())
    return update, sync
```

このループは 14.3 節の `epsilon_greedy` を用いて ϵ 貪欲に行動し（確率 ϵ でランダムな行動をとり、それ以外は argmax をとる）、すべての遷移を保存し、1,000 ステップのウォームアップ後に 2 ステップごと、サンプリングされたミニバッチ上で `update` を呼び出す。1 つのスイッチで本節の実験を切り替える。`use_target=True` では、コピーは `sync_every` ステップごとに更新される。`use_target=False` では毎ステップ後に更新されるため、ターゲットはオンラインネットワーク自身によって計算されることになり、ターゲットネットワークなしの素朴なレシピになる。引数 `step` は更新則を選択し、`copies` は各行動の複製数を選択する。各エピソードの終わりに、ジェネ

レータはステップ数、そのエピソードの収益、およびネットワークによる推定値 $\max_a Q_w(s_0, a)$（中心化された開始状態 $s_0 = \mathbf{0}$ における値）を返す。

```
def train_dqn(seed, qnet, use_target=True, step=q_step, copies=1):
    """DQN on CartPole; yields (env step, episode return, max_a Q(s0, a))."""
    rng, env = np.random.default_rng(seed), make_env(copies)
    update, sync = make_learner(qnet, step)
    buffer, s0 = ReplayBuffer(buffer_size, 4), np.zeros(4, np.float32)
    obs, ep_return = env.reset(seed=seed)[0], 0.0
    for t in range(1, num_env_steps + 1):
        a = d2l.epsilon_greedy(q_values(qnet, obs), epsilon(t), rng)
        next_obs, rew, terminated, truncated, _ = env.step(a)
        buffer.add(obs, a, rew, next_obs, float(terminated))
        obs, ep_return = next_obs, ep_return + rew
        if terminated or truncated:
            yield t, ep_return, q_values(qnet, s0).max()
            obs, ep_return = env.reset()[0], 0.0
        if len(buffer) >= warmup and t % train_freq == 0:
            update(buffer.sample(batch_size, rng))
        if t % (sync_every if use_target else 1) == 0:
            sync()
```

バッファが保存するのは `terminated` であり、`truncated` は決して保存しない。500 ステップで打ち切られた CartPole のエピソードは将来のある状態で終わっており、そこでブートストラップをマスクすると生き延びることに価値がないとネットワークに学習させてしまうことは 14.3 節で示したとおりである。誤ってマスクした場合に何が起きるかは演習問題 2 でたどる。

15.4.2.4 ターゲットネットワークのアブレーション

凍結コピーは本当に必要なのだろうか。そのスイッチ以外は同一の条件で、各アームにつき 3 つのシードを訓練する。シード間で事象がどの程度の頻度で起きるかに関する以下の記述は、各アーム 20 シードで確認したものである。訓練後のエージェントを後のセルで評価できるよう、ネットワークは訓練ループの外で構築する。`train_checked` は `train_dqn` をラップする関数である。エピソードごとの記録を収集するとともに、5,000 環境ステップの倍数を超えた最初のエピソード終了時、および予算終了時にもう 1 回、別コピーの環境上で常に $\operatorname{argmax}_a Q_w(s, a)$ をとる**貪欲方策**（greedy policy）を 10 エピソード実行する。`d2l.evaluate` は複数エピソードにわたる方策の収益を平均するモンテカルロ評価（14.2 節）であり、`greedy` は Q ネットワークをそのような方策へ変換する。これらのチェックは訓練自体には影響しない。まずは価値推定の挙動を理解したうえで、その結果を読むことにする。

```
def greedy(qnet):
    """The greedy policy of a Q-network, (obs, rng) -> argmax_a Q(obs, a)."""
    return lambda s, rng: int(q_values(qnet, s).argmax())

def train_checked(seed, qnet, every=5_000, episodes=10, **kwargs):
    """train_dqn, plus a greedy evaluation over a few episodes at the first
    episode end past every multiple of `every` steps, and one at the end."""
    env, run, checks = make_env(kwargs.get('copies', 1)), [], []
    env.reset(seed=10_000 + seed)
    for t, ret, v0 in train_dqn(seed, qnet, **kwargs):
        run.append((t, ret, v0))
```

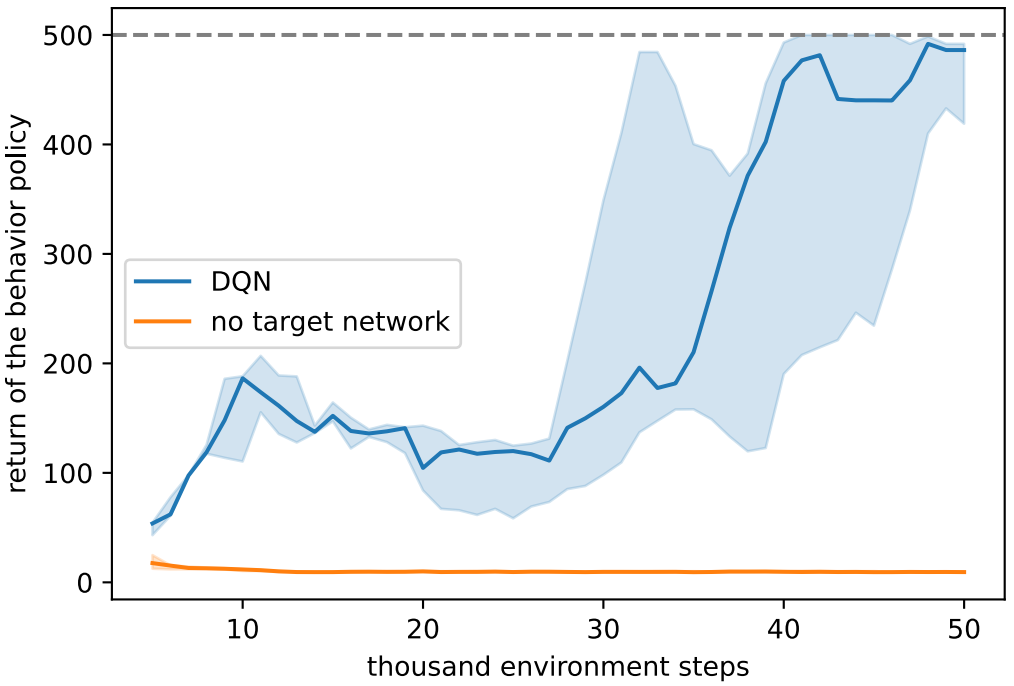
```
            if t >= every * (len(checks) + 1):
                checks.append(d2l.evaluate(env, greedy(qnet), episodes))
        checks.append(d2l.evaluate(env, greedy(qnet), episodes))
        return run, checks
```

`train_checked` を用いて、`arms = ('DQN', 'no target network')` の各アームにつき 3 つのシードを訓練し、エピソードごとの訓練記録を `runs` に、定期的な貪欲評価のスコアを `checks` に保存する。

```
arms = ('DQN', 'no target network')
qnets = {arm: [make_qnet(seed) for seed in range(num_seeds)] for arm in arms}
runs, checks = {}, {}
for arm in arms:
    out = [train_checked(seed, qnets[arm][seed], use_target=arm == 'DQN')
           for seed in range(num_seeds)]
    runs[arm], checks[arm] = [r for r, _ in out], [c for _, c in out]
```

シードごとに完了するエピソード数は異なるため、予算の単位である環境ステップ数を横軸にとってプロットする。`d2l.on_grid` は各実行のエピソードごとの記録を環境ステップの共通グリッド上に再サンプリングし、`d2l.plot_curves` はシード間の中央値（median）と最小シードから最大シードまでの帯を描画する。

```
grid = np.arange(1, num_env_steps // 1000 + 1) * 1000
d2l.plot_curves({arm: d2l.on_grid(runs[arm], grid, 1) for arm in runs},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='return of the behavior policy', smooth=5,
                reference=500)
```

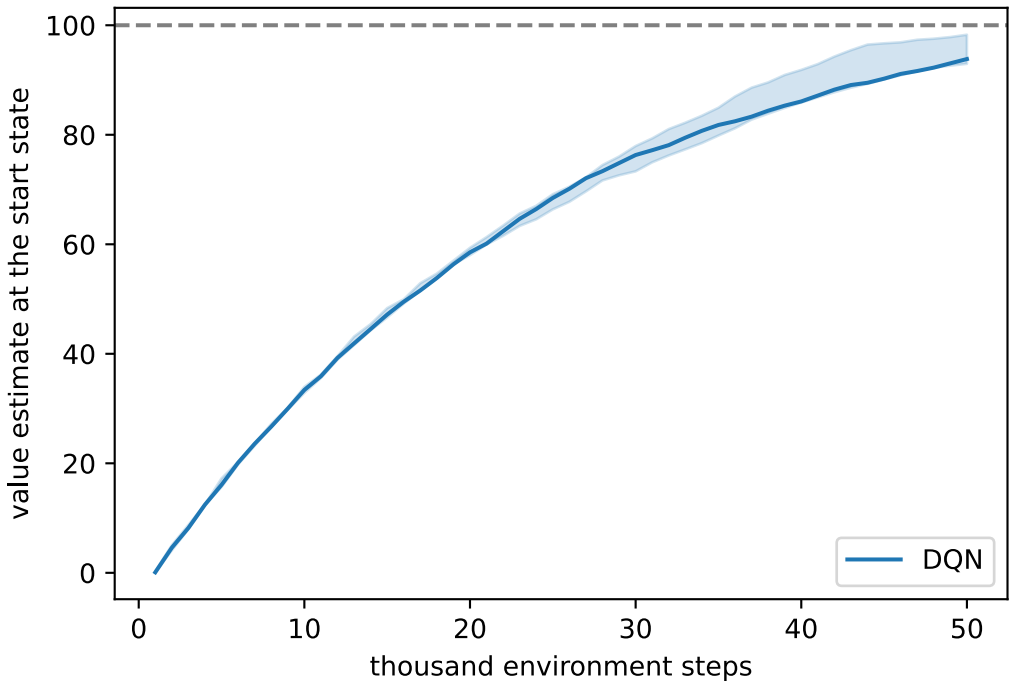


ターゲットネットワークありの場合、曲線はギザギザしたままであるものの、すべてのシードでランダム方策の収益である約 20 から数百の収益へと改善する。最終的な探索率（exploration rate）$\epsilon = 0.05$ のもとでも 20 ステップに 1 回はランダムな行動が強制されるため、行動方策（behavior policy）の収益は方策本来の性能を控えめに表している。ターゲットネットワークなしの場合、収益はランダム方策の水準を決して超えず、やがて約 9（毎ステップ同じ向きにカートを押し続けたときのエピソード長）まで落ち込む。

価値推定を見れば、何が起きたかがわかる。訓練ループは中心化された開始状態における $\max_a Q_w(s_0, a)$ を記録しており、その値がどこに落ち着くべきかはあらかじめ計算できる。バッファは `terminated` のときだけブートストラップをマスクするため、更新は 500 ステップの上限をまたいでブートストラップを続け、かつ状態には時計が含まれない。したがってネットワークは、時間上限がデータ収集だけを止めて過程自体は止めない継続タスク（continuing task）の価値を学習する。ポールを永久に倒さず保ち続ける方策は開始時点から $\sum_t \gamma^t = 1/(1-\gamma) = 100$ を獲得し、毎ステップの報酬が 1 である以上、いかなる方策もこれ以上は獲得できない。破線はこの上限を表す。

```
ceiling = 1 / (1 - gamma)
d2l.plot_curves({'DQN': d2l.on_grid(runs['DQN'], grid, 2)},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='value estimate at the start state',
                reference=ceiling)
print(f'continuing-task ceiling: {ceiling:.0f}; the no-target arm ends at ' +
      ', '.join(f'{np.asarray(r)[-1, 2]:.0e}' for r in
                runs['no target network']))
```

```
continuing-task ceiling: 100; the no-target arm ends at 1e+08, 1e+08, 1e+08
```



DQN の推定値は上限に向かって上昇していくのに対し、ターゲットネットワークなしの推定値

は 10^8 を超えて突き抜ける。暴走が任意の方向ではなく上向きに起きるのは、CartPole が毎ステップ 1 の報酬を与えるからである。エピソードを終了させないあらゆる遷移のターゲットは $1 + \gamma \max_{a'} Q_w(s', a')$、すなわちネットワーク自身の割引出力より 1 だけ大きい値になり、さらに $Q_w(s, a)$ をそこへ向かって押し上げる共有パラメータは $Q_w(s', \cdot)$ も一緒に押し上げる（s' が s に近いためである）。毎ステップでターゲットが予測の先を走り続け、この上昇を止めるものが何もない。Figure 15.10 はこのサイクルと、凍結コピーが断ち切る 1 本の辺を示している。

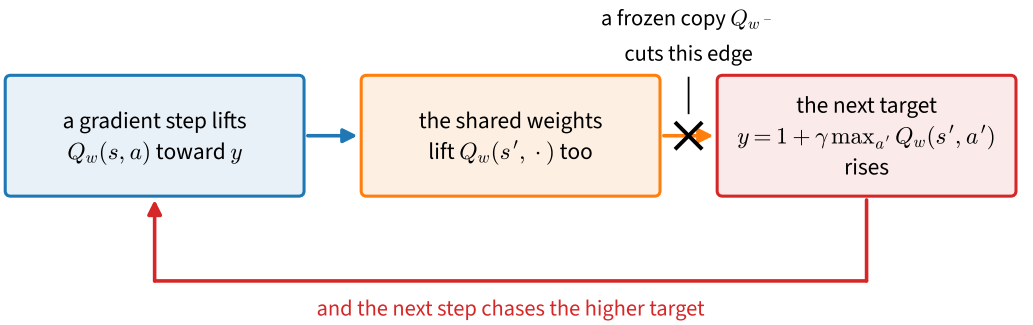


Figure 15.10: CartPole におけるターゲットネットワークなしの Q 学習。勾配ステップが $Q_w(s, a)$ をそのターゲット y に向かって押し上げると、共有重みによって $Q_w(s', \cdot)$ も押し上げられる。次のターゲットは $Q_w(s', \cdot)$ から計算されるためそれに伴って上昇し、次のステップはそのさらに高くなったターゲットを追いかける。代わりに凍結コピー Q_{w^-} がターゲットを計算することで、次の同期まで印付きの辺が断ち切られる。

凍結コピーは 125 勾配ステップの間ターゲットを静止させるため、価値は同期 1 回につき 1 段階ずつしか上昇できない。すべての価値が 0 から始まり、オンラインネットワークが 2 回の同期の間でターゲットに厳密に適合すると仮定しよう。このとき、n 回の同期の後にいかなる価値も次の値を超えることはない。

$$\sum_{k=0}^{n-1} \gamma^k = \frac{1 - \gamma^n}{1 - \gamma}, \tag{15.36}$$

これは報酬 1 が n ステップ続くときの割引収益である。最初の勾配ステップの後、予算内には約 196 回の同期が含まれるため、開始状態の価値は約 86 で上から抑えられる。プロットされた推定値はこの上界によく沿って推移しつつ、約 10,000 ステップ以降は数ポイント上回り、すべてのシードで 86 を超えて終わる。これはネットワークが汎化（generalization）するためである。ある状態の価値を押し上げるステップはその近傍の価値も押し上げるため、同期 1 回につき 1 バックアップよりも速く価値が伝播する（演習問題 5）。

ターゲットネットワークはこのフィードバックに対する修復法の 1 つであり、唯一の方法ではない。PQN (Gallici et al. 2025) は凍結コピーとリプレイバッファの両方を廃止し、すべての隠れ層の活性化値（activation）を正規化したうえで、多数の環境から並列にデータを収集する。演習問題 8 では CartPole 上でこの正規化を試す。

15.4.2.5 貪欲方策の評価

訓練中の収益曲線は、実行結果の要約としては不十分である。探索的な行動が含まれているうえ、最終値は実行がたまたまいつ止まったかに左右され、さらに曲線全体を見た後で最良値を選ぶの

はどの手法をも実際以上に良く見せてしまう。そこで、あらかじめ固定した予算ごとに貪欲方策を評価する。これが `train_checked` に記録しておいたチェック、すなわち 5,000 環境ステップごとおよび予算終了時点における 10 エピソードの貪欲収益である。

```
print('greedy return (10 episodes) every 5,000 environment steps')
for arm in arms:
    for seed, c in enumerate(checks[arm]):
        print(f'{arm:>17}, seed {seed}: ' + ' '.join(f'{x:3.0f}' for x in c))
```

```
greedy return (10 episodes) every 5,000 environment steps
              DQN, seed 0: 214 152  57 111 120 136 500 500 500 500
              DQN, seed 1: 232 150 101  93  54 163 101 335 404 103
              DQN, seed 2: 182  92 124 117 117 112 265 500 500 500
no target network, seed 0:   9   9  10   9   9   9   9   9   9  10
no target network, seed 1:  10  10   9   9  10  10   9   9   9  10
no target network, seed 2:   9   9  10  10   9   9   9   9   9   9
```

このように確認すると、DQN の貪欲方策は最初の 30,000 ステップではおおむね約 50〜250 の収益にとどまり（まれに早い段階で 500 に達することもある）、その後 35,000〜45,000 ステップの間で大半のシードが 500 に到達する。一度良い結果が出てもそれで確定するわけではない。出力結果の 1 シードがそうであるように、後から数百ポイント後退するシードもあり、20 シード中約半数でそうした低下が見られた。ターゲットネットワークなしでは、どのチェック時点でも貪欲方策が約 10 ステップを超えて持続することはなく、訓練を長く続けても救われない。また、DQN は保存された各遷移で何度も訓練するにもかかわらず、エピソード終了までポールを倒さず保つのに、バッチあたり 4 エポックの PPO が 15.2 節で要したよりも多くの環境ステップを必要とする。

貪欲収益が示すのは方策がどれだけ優れているかであり、ネットワークがその良さをどれだけ正確に把握しているかではない。Van Hasselt et al. (2016) は、価値推定を貪欲方策が実際に獲得する割引収益（まさにその推定値が予測しようとしている量）と照らし合わせて検証している。`audit` は訓練済みネットワークに対して 3 つの数値を計算する。100 エピソードにわたる貪欲収益、20 エピソードにわたる貪欲方策の割引収益（`d2l.evaluate` に `gamma` を指定）、および推定値 $\max_a Q_w(s_0, a)$ である。エピソードは中心化された状態の近くから始まるため、この割引収益は s_0 の価値を測るものになる。500 ステップの上限のもとでは、完璧な方策であっても獲得できる割引収益は $(1-\gamma^{500})/(1-\gamma) \approx 99.3$ にとどまり、継続タスクの上限よりわずかに小さい。

```
def audit(qnet, seed, copies=1):
    """Greedy return over 100 episodes, its discounted return over 20, and
    the network's own estimate max_a Q(s0, a) of the latter."""
    env = make_env(copies)
    env.reset(seed=100 + seed)
    score = d2l.evaluate(env, greedy(qnet), 100)
    env.reset(seed=100 + seed)
    disc = d2l.evaluate(env, greedy(qnet), 20, gamma=gamma)
    return score, disc, q_values(qnet, np.zeros(4, np.float32)).max()

audits = {arm: [audit(qnets[arm][seed], seed) for seed in range(num_seeds)]
          for arm in arms}
for arm in arms:
    for seed, (score, disc, v0) in enumerate(audits[arm]):
        print(f'{arm:>17}, seed {seed}: return {score:3.0f}, discounted '
```

```
                f'{disc:5.1f}, max_a Q(s0) {v0:8.3g}')
```

```
             DQN, seed 0: return 500, discounted  99.3, max_a Q(s0)     93.9
             DQN, seed 1: return  93, discounted  61.9, max_a Q(s0)     98.4
             DQN, seed 2: return 500, discounted  99.3, max_a Q(s0)     93.1
no target network, seed 0: return  10, discounted   9.0, max_a Q(s0)  1.4e+08
no target network, seed 1: return   9, discounted   9.1, max_a Q(s0) 1.47e+08
no target network, seed 2: return   9, discounted   8.9, max_a Q(s0) 1.45e+08
```

貪欲方策がポールを倒さず保てる場合、推定値は方策が獲得する割引収益から約 8 ポイント以内に収まり、たいていはそれを下回る。予算終了時点でも価値がまだ上昇途中であり、上の開始状態の価値プロットが頭打ちになっていないからである。一方、貪欲方策が失敗する場合（5 シード中約 2 シードで発生）には、推定値は方策が獲得する収益を上回り、そのような実行の約半数で 20 ポイント以上も高くなる。行動が 2 つしかない設定では、ネットワークが実際の方策の達成値以上の価値を約束してしまう兆候はこうした失敗時の実行にしか現れず、件数も少ないうえ他の誤差と混ざり合っているため効果を測りにくい。より鮮明な検証には、さらに多くの行動にわたる最大値が必要になる。

15.4.3 過大評価

15.4.3.1 数値で見る最大化バイアス

ターゲット Equation 15.35 は推定された行動価値の最大値をとる。ノイズを含む推定値に対しては $\mathbb{E}[\max_{a'} \hat{Q}(s', a')] \geq \max_{a'} \mathbb{E}[\hat{Q}(s', a')]$ が成り立ち、どの行動が最大になるかをノイズが変え得るときには狭義の不等号になる。14.3 節ではテーブル上でこの最大化バイアス（maximization bias）を実演し、15.3 節ではハード最大値が一様な参照方策を持つソフトバックアップの $\beta \to 0$ 極限であることを示した。関数近似を用いる場合、ある状態でのバイアスはそこへ至る状態群のターゲットへと流れ込む (Thrun and Schwartz 1993)。

次のシミュレーションでは、4 つの行動すべてに同じ真の価値 0 を与え、各推定値に独立な単位分散ノイズを加える。単一推定量（single estimator）は同じノイズ付き推定値を用いて行動の選択と評価の両方を行う。二重推定量（double estimator）は 1 組目のノイズ付き推定値で選択し、独立な 2 組目の推定値で評価する (Hasselt 2010)。

```
rng = np.random.default_rng(0)
est = rng.standard_normal((100_000, 4))     # four values, all truly zero
second = rng.standard_normal((100_000, 4))  # an independent second estimate
sel = est.argmax(1)
print(f'single estimator, E[max of the estimates]: {est.max(1).mean():.3f}')
print(f'select with one, evaluate with the other: '
      f'{second[np.arange(100_000), sel].mean():+.3f}')
```

```
single estimator, E[max of the estimates]: 1.031
select with one, evaluate with the other: -0.006
```

単位分散ノイズを持つ 4 つの推定値の最大値には、約 1 単位の上向きバイアスが生じる。選択と評価に独立な推定値を用いれば、どの行動が選ばれたとしても評価側のノイズの平均は 0 であるため、測定されるバイアスは 0 付近になる。Figure 15.11 に示すように、単一推定量のバイアスは行動数とともに増大するのに対し、独立な二重推定量のバイアスは 0 のままである。10^5 個の行動の中から次のトークンを選ぶ言語モデルでは、ノイズを含む推定値にわたる最大値から生じるバイアスははるかに大きくなり、これが言語モデルの事後学習で Q ネットワークではなく方策

勾配が用いられる理由の 1 つとなっている。固定されたデータセットでは、新たな相互作用によって過大評価された行動が修正されないため、このバイアスはさらに深刻になる（15.7 節）。

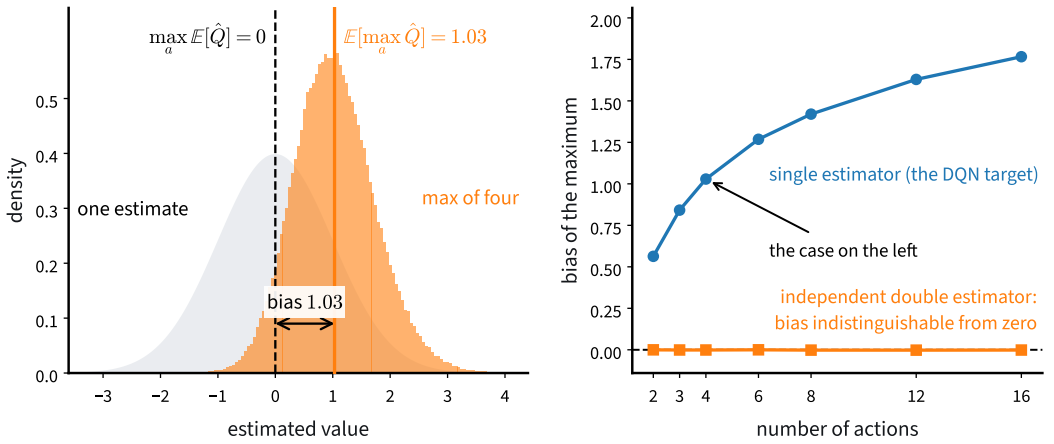


Figure 15.11: ノイズを含む推定値にわたる最大値は上向きにバイアスを持つ。左：同じ真の価値 0 に対する 4 つの独立な推定値の最大値の分布と、単一の推定値の分布の比較。最大値は測定値で 1.03 のバイアスを持つ。右：単一推定量のバイアスは行動数とともに増大する一方、1 組の推定値で行動を選択し独立な別の組で評価する独立な二重推定量のバイアスは、全体を通じて 0 と区別できない。

15.4.3.2 Double DQN

二重推定量には 2 つの価値推定が必要だが、DQN はすでに 2 つのネットワークを持っている。Double DQN (Van Hasselt et al. 2016) はオンラインネットワークで行動を選択し、ターゲットネットワークでその行動を評価する。

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right) Q_{w^-}\left(s', \operatorname*{argmax}_{a'} Q_w(s', a')\right). \tag{15.37}$$

これにより、オンラインネットワークが偶然過大評価した行動は別のネットワークによって採点される。この変更に新たなネットワークは不要であり、s' に対するオンラインネットワークの順伝播が 1 回増えるだけで済む。ただし、コピーは高々 125 勾配ステップ前のオンラインネットワークであるため、2 つのネットワークは独立ではなく、Double DQN が取り除けるのはバイアスの一部である。評価側の推定値が選択側の推定値と相関係数 ρ で相関している場合、残るバイアスは単一推定量の ρ 倍になる（演習問題 3）。`q_step` への変更は 3 行だけである。

```
def q_step_double(qnet, target, opt, batch):
    """q_step with selection split from evaluation, eq_double_dqn."""
    sel = q_values(qnet, batch.next_obs).argmax(-1)
    fit_q(qnet, opt, batch, batch.td_target(
        lambda s: q_values(target, s)[np.arange(len(s)), sel], gamma))
```

15.4.3.3 複製された行動

本来の CartPole では最大値をとる選択肢がほとんどない。単位分散ノイズを持つ 2 つの独立な推定値の大きい方が持つバイアスは $1/\sqrt{\pi} \approx 0.56$ と 4 行動の場合の約半分であり、アブレーシ

ョンにおける 2 行動の推定値は同期の遅れによって逆方向へ引っ張られている。タスク自体を変えずに訓練済みネットワーク上でこのバイアスを観察するため、CartPole の各行動を 9 回ずつ提示する。`make_env(copies=9)` は Atari の全行動セットと同じサイズである 18 個の行動を公開し、行動 a は行動 $a \bmod 2$ と同じ向きにカートを押す。また `make_qnet(seed, copies=9)` は 18 個の出力を持つ。複製を増やしても最適価値も上限の 100 も変わらない。18 個の出力は 2 つの価値それぞれに対する 9 個のノイズ付き推定値となり、ターゲット内の最大値は 9 個の中で最も運良く高くなったものを選ぶ。すなわち、Figure 15.11 の右パネルが実際のネットワーク上で再現されるのである。次のセルでは 18 行動の DQN と Double DQN、および 2 行動の Double DQN を訓練し、訓練済みのすべてのネットワークを監査する（2 行動の DQN はアブレーションのアームをそのまま用いる）。

```
arms2 = {'Double DQN, 2 actions': (q_step_double, 1),   # (update, copies)
         'DQN, 18 actions': (q_step, 9),
         'Double DQN, 18 actions': (q_step_double, 9)}
names = ('DQN, 2 actions', *arms2)
qnets[names[0]], runs[names[0]] = qnets['DQN'], runs['DQN']
audits[names[0]] = audits['DQN']
for name, (step, copies) in arms2.items():
    qnets[name] = [make_qnet(seed, copies) for seed in range(num_seeds)]
    runs[name] = [list(train_dqn(seed, qnets[name][seed], step=step,
                                 copies=copies)) for seed in range(num_seeds)]
    audits[name] = [audit(qnets[name][seed], seed, copies)
                    for seed in range(num_seeds)]
```

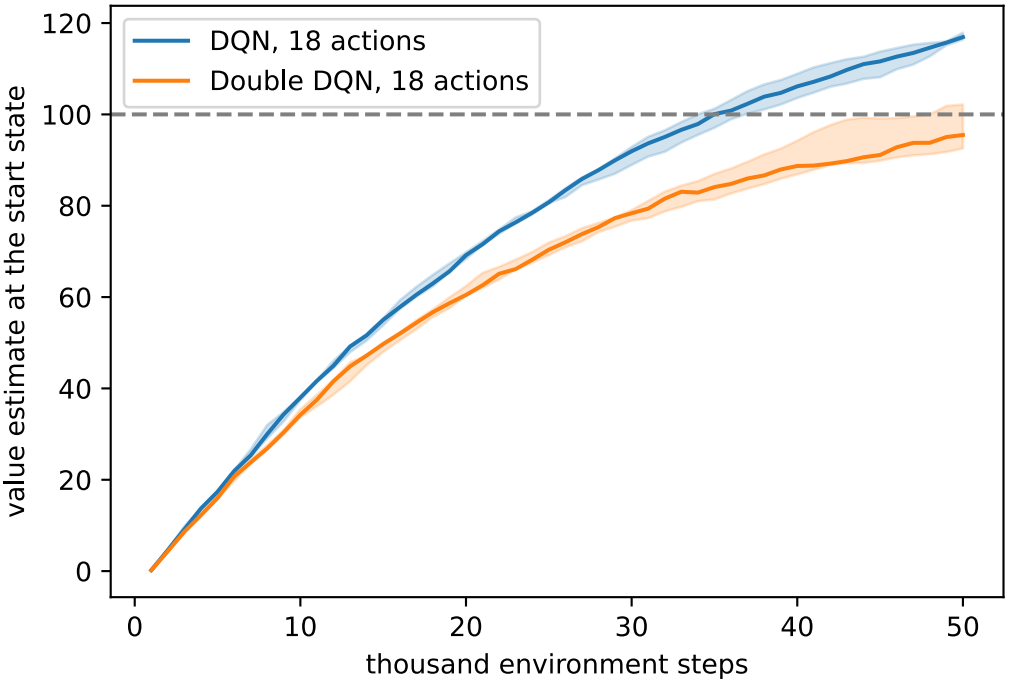
各アームについて、シードごとに貪欲収益、貪欲方策の割引収益、および開始状態での推定値を出力する。

```
print(f'{"per seed:":>22} {"greedy return":>17}   {"discounted":>17}'
      f'   {"max_a Q(s0)":>17}')
for name in names:
    res = np.array(audits[name]).T
    print(f'{name:>22} ' + '   '.join(' '.join(f'{x:5.1f}' for x in row)
                                       for row in res))
```

```
            per seed:     greedy return          discounted         max_a Q(s0)
        DQN, 2 actions 500.0  93.4 500.0    99.3  61.9  99.3    93.9  98.4  93.1
 Double DQN, 2 actions 500.0 500.0 500.0    99.3  99.3  99.3    92.7  95.0  91.1
       DQN, 18 actions 181.0 500.0 153.2    83.8  99.3  78.7   117.7 116.2 117.1
Double DQN, 18 actions  94.0 356.2 500.0    61.4  95.8  99.3    95.5 102.0  92.4
```

訓練を通じた 18 行動の 2 手法における開始状態の推定値と上限の比較を次にプロットする。

```
d2l.plot_curves({name: d2l.on_grid(runs[name], grid, 2) for name in names[2:]},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='value estimate at the start state',
                reference=ceiling)
```



各行動を 9 回ずつ提示すると、DQN の開始状態の推定値はすべてのシードで上限を超えて終わり、典型的には上限より 10〜30 ポイント高くなる。貪欲方策がポールを倒さず保って約 99 を獲得しているシードでも、推定値はその収益を同程度超過する。このタスクで許されるいかなる価値も上限を超えることはないため、それ以上の評価を行わなくても過大評価が生じていることは確実である。プロットでは、DQN の推定値は 40,000 ステップより前に上限を突き抜けて上昇し続ける一方、Double DQN の推定値は上限付近かそれ以下にとどまる。Double DQN の推定値はほぼすべてのシードで 10〜25 ポイント低く、上限付近で終わる（ただし方策が失敗したシードでは依然として上振れし得る）。20 シード実行した 18 行動 DQN のうち数本では、ブートストラップを通じてバイアスがフィードバックされ、50,000 ステップ時点でもまだ上昇が続いている。2 行動の場合には両者のターゲットは数ポイント以内で一致し、この予算では貪欲収益にもシード間のばらつきを超える差は生じない。CartPole において、二重推定量は貪欲方策がポールを保つ性能を変えることなく、価値推定をより正確にする。

これで、任意の方策から得られた遷移を何度も再生しながら、ターゲットを静止させる凍結コピーと最大値による水増しを抑える 2 つ目の推定値を用いて価値ネットワークを訓練できるようになった。ただし、これら 2 つの修復法はいずれも行動にわたる最大値が計算できることを前提としている。CartPole の 2 通りの押し方や Atari の 18 通りのジョイスティック操作は有限のリストをなしている。しかしモーターのトルクは区間内の連続値であり、その上での argmax はステップごとにそれ自体が 1 つの最適化問題になる。次節では経験再生と凍結コピーを保ったまま、最大値演算を最良の行動を提案するよう訓練された 2 つ目のネットワークに置き換え、Figure 14.23 のオフポリシーの列を完成させる。

15.4.4 まとめ

Q テーブルをネットワークに置き換えると、関数近似、ブートストラップ、オフポリシーデータの 3 つが組み合わさる。Baird の 7 状態の例は、真の価値が表現可能でサンプリングノイズが存在しない場合でさえ、この組み合わせが発散し得ることを示している。DQN はこれら 3 つの要

素すべてを維持したまま、2 つの結合を修復する。リプレイバッファが連続する遷移の相関を断ち切って各遷移を何度も再利用し、凍結ターゲットネットワークが同期と同期の間で回帰目標を静止させる。CartPole では、凍結コピーを取り除くと開始状態の価値が 10^8 を超えて暴走するが、凍結コピーを用いれば価値は同期ごとに 1 段階ずつ上限の 100 に向かって上昇し、35,000〜45,000 環境ステップ後には大半のシードで貪欲方策が 500 ステップ最後までポールを保てるようになる。また、ターゲット内の最大値は価値を上向きにバイアスさせ、選択肢となる行動が多いほどそのバイアスは大きくなる。各行動を 9 回ずつ提示すると、DQN の推定値はすべてのシードで上限を超えて終わる。Double DQN はオンラインネットワークで行動を選択し凍結コピーでそれを評価することで、順伝播 1 回分の追加コストと引き換えにそのバイアスの大部分を取り除く。

15.4.5 演習問題

1. [code] **オンポリシー重み付けのもとでの Baird の反例**。15.4.1.3 節における Baird の反例の計算では、`.mean(0)` によって 7 つの状態を等しく重み付けしている。行列形式では、1 スイープは $\mathbf{w} \leftarrow \mathbf{w} + \alpha A \mathbf{w}$（ただし $\alpha = 0.01$）および

$$A = \Phi^\top D(\gamma P - I)\Phi, \tag{15.38}$$

と書ける。ここで D は状態重みの対角行列であり、P はすべての行が状態 7 に全確率を置く評価対象方策の遷移行列である。

   1. A の固有値を $D = I/7$ の場合に計算せよ。次に $1 + \alpha\lambda$（実部が最大の固有値 λ に対する値）を、$\mathbf{w}$ の上限ノルムが 20,000 スイープの実行終盤で示す 1 スイープあたりの成長率と比較せよ。
   2. 評価対象の方策は状態 7 へ移動してそこにとどまるため、その訪問分布は状態 7 にすべての重みを置く。この重み付けで反復を再実行せよ。どの重みおよびどの状態価値へ収束するか。また、それらの値は真の価値とどのように比較されるか。
   3. 2 つの重み付けを $\mathbf{d} = (1-p)\,\mathbf{1}/7 + p\,\mathbf{e}_7$ として混合し、最小の p（A のどの固有値も正の実部を持たなくなる値）を $\gamma \in \{0.9, 0.95, 0.99\}$ のそれぞれについて二分法で求めよ。反復が許容するオフポリシー重みの量は割引率にどのように依存するか。

2. **done ではなく terminated**。`train_dqn` は `float(terminated)` を保存するため、CartPole の 500 ステップの時間上限に至る遷移でもブートストラップは継続される。代わりに `float(terminated or truncated)` を保存する実装を考えよ。

   1. 貪欲方策がまだ数百ステップ以内に失敗している段階では、この変更は保存されたどの遷移に影響するか。また、すべてのエピソードが時間上限に到達するようになった後はどの遷移に影響するか。
   2. 観測には時計が含まれないため、ネットワークはステップ 499 の状態とステップ 50 のよく似たバランス状態を区別できない。バランス状態におけるネットワークの推定値を単一の数値 q、すなわち毎ステップ報酬 1 を与え H ステップ後に打ち切られる 1 状態タスクの価値としてモデル化せよ。誤ったマスクのもとでは、1 エピソードの遷移のうち $H-1$ 個（全 H 個中）がターゲット $1 + \gamma q$ を持ち、最後の 1 個がターゲット 1 を持つ。平均化された更新の不動点が $q = H/\big(H - \gamma(H-1)\big)$ であることを示し、$H = 500$、$\gamma = 0.99$ における値を計算せよ。15.4.2.4 節の開始状態の価値プロットは $1/(1-\gamma)$ を上限として示している。q を、残り時間が既知である 500 ステップエピソードの開始状態の価値 $(1-\gamma^H)/(1-\gamma)$ と比較せよ。時計を持たないネットワークに対して、誤った実装の上限となるのはこれら 2 つのうちどちらか。最終的な推定値が 94 であったとしたら、それは何を示唆するか。

3. [code] **閉形式での最大化バイアス**。推定値 $X_1, \ldots, X_m$（m 個の行動すべてが同じ真の価値 0 を持つ）が、それぞれ分散 σ^2 の同時正規分布に従うとする。

1. $m = 2$ かつ推定値が互いに独立であるとき、$\mathbb{E}[\max(X_1, X_2)] = \sigma/\sqrt{\pi}$ となることを、$\max(x, y) = (x + y)/2 + |x - y|/2$ を用いて示せ。
2. 1 つのネットワークの複数の出力は隠れ層を共有するため、その誤差には相関が生じる。$m = 2$ で相関係数が c（X_1 と X_2 の間）であるとき、バイアスが $\sigma\sqrt{(1-c)/\pi}$ になることを示せ。
3. Double DQN では、オンラインネットワークが選択した行動をターゲットネットワークが評価するため、2 つのネットワークの誤差は相関する。行動 a の評価側の推定値を $Y_a = \rho X_a + \sqrt{1-\rho^2}\, Z_a$ とモデル化する。ここで $Z_1, \dots, Z_m$ はそれぞれ平均 0・分散 σ^2 で $X_1, \dots, X_m$ と独立であり、したがって Y_a は分散 σ^2 を持ち、相関係数 ρ で X_a と相関する。このとき、$Y_{\text{argmax}_a X_a}$ のバイアスが $\rho\, \mathbb{E}[\max_a X_a]$ であることを示せ。
4. (a) と (c) を、15.4.3.1 節のシミュレーションを変更して確認せよ。2 つの行動を用い、2 つ目の推定値を `rho * est + np.sqrt(1 - rho**2) * noise`（`noise` は新たに生成した標準正規分布に従う乱数）として、$\rho \in \{0, 0.5, 0.9\}$ について実行せよ。

4. [code] **経験再生の再利用回数とバッファ容量**。●バッファに `warmup` 個の遷移が貯まると、`train_dqn` は一様にサンプリングされた `batch_size` $= 128$ 個の遷移上で、`train_freq` $= 2$ 環境ステップごとに 1 回の勾配ステップを実行する。デフォルトの容量では、遷移の追い出しは生じない。
   1. ステップ $t_0 \geq 1000$ で保存された遷移が、実行全体を通じて期待値で約 $64 \ln(50{,}000/t_0)$ 回サンプリングされることを示し、これを $t_0 \in \{1000, 25{,}000, 49{,}000\}$ に対して計算せよ。訓練のどの段階のデータが更新の大半を占めるか。
   2. 容量が 500 遷移のとき、ある遷移が満杯のバッファ内にとどまっている間に期待値で何回サンプリングされるか。これを (a) と比較せよ。
   3. `buffer_size` を 500 と 5000 に設定し、小さい方の容量では `warmup` を 500 に下げよ（`train_dqn` はバッファが `warmup` 個の遷移を保持するまで待機するためである）。各容量につき 1 シードを訓練し、収益曲線と開始状態の価値トレースをデフォルトの実行と比較せよ。悪影響は訓練のどの段階で現れるか。
   4. 小さなバッファは、ミニバッチ内の遷移を相関させることと古い状態を捨てることの両方を引き起こす。これら 2 つの効果を切り分けるため、デフォルトの容量を保ったまま `ReplayBuffer.sample` を変更し、一様に抽出した開始インデックスから連続する `batch_size` 個の遷移を返すようにせよ。この実行を容量 500 での実行と比較せよ。(c) の悪影響を説明するのは 2 つの効果のうちどちらか。

5. [code] **同期周期**。●`train_dqn` は `sync_every` $= 250$ 環境ステップごと、すなわち 125 勾配ステップごとにオンライン重みをターゲットネットワークへコピーする。
   1. 15.4.2.4 節では、n 回の同期後のあらゆる価値を $(1-\gamma^n)/(1-\gamma)$ で上から抑えた。最初の勾配ステップに続く同期回数を数え、`sync_every` $\in \{10, 50, 250, 1000, 5000\}$ に対して予算終了時点でのこの上界を計算せよ。
   2. これら 5 つの値それぞれについて 2 シードずつ DQN アームを再実行し、予算終了後の貪欲評価と開始状態の価値トレースを記録せよ。スイープのどちらの端がターゲットネットワークなしの実行と同様に振る舞うか。また、(a) の上界が到達可能な値を制限するのはどちらの端か。
   3. この上界は、すべての価値が 0 から始まり、同期と同期の間でオンラインネットワークがターゲットに厳密に適合することを仮定している。`sync_every` $= 250$ とし、1 シードの最初の 40 回の同期それぞれの直前における $\max_a Q_w(s_0, a)$ を記録して上界とともにプロットせよ。実行結果が最初に上界を超えるのはどこか。また、その超過は 2 つの仮定のうちどちらに反しているか。

6. **経験再生からの n ステップターゲット。**●DQN の後継手法のいくつかは、1 ステップターゲット Equation 15.35 を n ステップターゲット、すなわち末尾に最大値を持つ n ステップ収益（15.1 節）に置き換える。

$$y_t^{(n)} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n \max_{a'} Q_{w^-}(s_{t+n}, a'), \tag{15.39}$$

ここではエピソードがステップ $t+n$ を超えて続く場合について書いている。

   1. `ReplayBuffer.sample` は前後の切り離された遷移を返す。バッファが何を保存し、`sample` がどのようにインデックスを選べば、$y_t^{(n)}$ を構成できるようになるか。
   2. $y_t^{(n)}$ を、エピソードがステップ $t+j$（ただし $j < n$）で終端する場合について書き下せ。代わりにステップ $t+j$ で打ち切られる場合にはどのように変わるか。
   3. 行動 $a_{t+1}, \dots, a_{t+n-1}$ は、古いネットワークの ϵ 貪欲な行動方策によって選ばれたものである。$n = 2$ かつ $Q_{w^-} = Q^*$ のとき、$\mathbb{E}[y_t^{(2)} \mid s_t, a_t]$ を書き下し、それが Q^* の 1 ステップターゲットと一致するのは a_{t+1} が確率 1 で Q^* に関して貪欲である場合に限ることを示せ。n ステップターゲットはどの方策の価値を推定しているか。
   4. n ステップ収益（15.1 節）では (c) の問いは生じなかった。それはなぜか。

7. [code] **ターゲットネットワークを用いた Baird の反例。**●●DQN は重みの凍結コピーからブートストラップを計算する。15.4.1.3 節における Baird の反例の計算で、`delta` をコピー `w_target` から `gamma * (Phi @ w_target)[6] - Phi @ w` として計算し、さらに `w_target = w.copy()` を k スイープごとに実行するようにせよ。

   1. 100,000 スイープを $k \in \{1, 10, 30, 100, 300, 1000\}$ について実行し、$\mathbf{w}$ の上限ノルムと最大価値誤差 $\max_i |(\Phi\mathbf{w})_i|$ の両方をプロットせよ。どの k に対して反復は収束し、その速さは k にどう依存するか。価値が 0 に近づく場合でも $\mathbf{w}$ の上限ノルムが 0 に近づかないのはなぜか。
   2. `Phi` の 7 つの行が線形独立であり、したがってターゲット価値の任意のベクトルがある $\mathbf{w}$ によって厳密に表現されることを示せ。ここから、2 回の同期の間の回帰が厳密に解かれるならば、正のあらゆる状態重み付けのもとで各同期が価値 $\mathbf{v}$ を $\gamma P\mathbf{v}$ へ写すことを導け。これを用いて (a) の両端の挙動を説明せよ。
   3. Tsitsiklis と Van Roy の反例では、2 つの非終端状態が価値 w と $2w$（単一の重み w に対する値）を持ち、すべての報酬は 0 である。第 1 の状態は第 2 の状態へ遷移し、第 2 の状態は確率 $1-\xi$ でその場にとどまり、それ以外は終端する。両状態を等しく重み付けせよ。凍結された重み w_k のターゲットに対する厳密な最小二乗回帰が $w_{k+1} = \gamma(6 - 4\xi)\, w_k/5$ を与えること、そしてターゲットネットワークなしの半勾配反復が γ と ξ に関する同じ条件のもとで発散することを示せ。
   4. どちらの例も三要素すべてを含んでいる。ターゲットネットワークで修復できる例と修復できない例を分けるのは、関数クラスのどの性質か。

   Richard S. Sutton および Andrew G. Barto 著『Reinforcement Learning: An Introduction』Example 11.1 に基づく。

8. [extended] **ターゲットネットワークに代わる LayerNorm。**●●Gallici et al. (2025) は、十分に幅の広い LayerNorm 付き Q ネットワークを最終層の重みに対する ℓ_2 ペナルティとともに訓練すると、オフポリシーデータ上であってもターゲットネットワークなしの時間差分学習に対する安定性条件を満たすことを証明している。同論文の実験では、Baird の反例を除くすべての環境で LayerNorm 単独で十分であった。

   1. `make_qnet` の各隠れ線形層の直後（ReLU の直前）に LayerNorm を挿入し、ターゲッ

トネットワークなしのアームを 3 シードで再実行せよ。開始状態の価値トレースと貪欲評価を、15.4.2.4 節の両アームと比較せよ。
2. ターゲットネットワークを保持する DQN アームにも同じ層を追加せよ。そこでは LayerNorm によって何が変わるか。
3. (a) で依然として発散するシードがある場合、最後の線形層の重みに ℓ_2 ペナルティを追加し、発散を防ぐ最小の係数を求めよ。係数を大きくすると、学習される価値はどうなるか。
4. この定理は十分に幅の広いネットワークと正則化された更新を対象としている。今回の実験はその仮定のうちどれを満たしているか。(a)〜(c) の結果のうち、論文の主張と矛盾するのはどのような結果であり、単に定理の適用範囲外にとどまるのはどのような結果か。

15.5 ソフトアクタークリティック

14.9 節の最後では、方策ネットワークが振り子を振り上げて静止させることを学習したが、それには数十万もの環境ステップを要した。1,600 ステップからなる各バッチは 1 回の更新に使われたあと、そのまま捨てられていた。シミュレータであればこうしたステップは安価に実行できる。しかしロボットアームではそうはいかない。1 ステップごとに実時間がかかり、ハードウェアが摩耗し、何かを壊す危険も伴う。本節では、エージェントが同じ制御器を数千ステップから学習するにはどうすればよいかを問う。

すでに見てきた 2 つのアイデアが、この改善の大半を担っている。1 つ目は 15.4 節が訓練に用いた過去の遷移のバッファであるリプレイであり、古い方策が選んだ行動からクリティックが学習できる限り、各遷移を 1 回ではなく何回もの更新に使えるようにする。2 つ目は 14.9 節のパスワイズ勾配であり、そのクリティックを行動そのものを通して微分することでアクターを動かす。DDPG (Lillicrap et al. 2016) はこの 2 つを決定論的なアクターと組み合わせたが、クリティックの値を上げる方向へ進むアクターはクリティックの誤差も同時に追いかけてしまうため、不安定であることが判明した。TD3 (Fujimoto et al. 2018) は、この問題の多くが 15.4.3.1 節の最大化バイアス（maximization bias）に起因することを突き止め、2 つのクリティックを訓練して小さい方の値を採用するようにした。ソフトアクタークリティック（SAC）(Haarnoja, Zhou, Abbeel, et al. 2018; Haarnoja, Zhou, Hartikainen, et al. 2018) は、これら両方の改善策を維持しつつ、3 つ目のアイデアとして 15.3 節のエントロピー報酬を加える。ランダムさを保つことに対して方策に報酬が支払われるため、自身のクリティックに対して訓練されるアクターは、クリティックがその時点で好む単一の行動へと崩壊することなく探索を続ける。

一言でまとめれば、SAC は方策のエントロピーを報酬に加え、その結果得られる価値をリプレイされた遷移からオフポリシーで学習し、自身がサンプリングした行動を通してクリティックを微分することでアクターを更新する。ただし、2 つの細部に注意を要する。1 つ目に、$[-2, 2]$ に制限されたトルクを扱うには、勾配が行動を通って流れるため、正しい密度をもつ有界な方策が必要になる。2 つ目に、エントロピーにかける重みは報酬のスケールに見合っていなければならないため、SAC はこれを固定せずに学習する。本節では、エントロピー正則化目的関数の評価ステップと改善ステップを導出し、有界な方策を構築して検証したうえで更新則を組み立て、最後に振り子の学習に何ステップ必要か、またクリティックの予測がどこまで信頼できるかを測定する。

```
import copy
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

本節を通して用いる例題は、14.9 節の Pendulum である。モーター付きの支点から振り子が吊り下げられている。エージェントは $(\cos\vartheta, \sin\vartheta, \dot\vartheta)$ を観測し（ϑ は直立位置からの角度）、トルク $u \in [-2, 2]$ を加える。各ステップでは $\vartheta^2 + 0.1\,\dot\vartheta^2 + 0.001\,u^2$ のコストがかかるため、真下（$\vartheta = \pi$）で静止して垂れ下がった振り子は 1 ステップあたり $\pi^2 \approx 9.9$ のコストを要し、他のどの静止位置よりも大きくなる。1 エピソードは 200 ステップ続く。一様ランダムなトルクでは約 $-1{,}200$ の収益となり、振り上げて静止させる制御器では約 -200 以上の収益が得られる。以下のすべての実行は高々 20,000 環境ステップとし、各設定は 3 つのシードで訓練する。$c = 2$ は行動のスケールである。ステップ数の範囲や、実行間で事象がどの程度の頻度で起きるかに関する記述は、各設定あたり 20 回の実行にわたって確認したものである。

```
num_env_steps, num_seeds, c = 20_000, 3, 2.0
```

15.5.1 最大エントロピー目的関数

15.5.1.1 報酬としてのエントロピー

15.3 節では、固定された参照方策 π_{ref} の近くに方策を保つため、期待報酬から $\beta\, D_{\mathrm{KL}}(\pi \| \pi_{\text{ref}})$ を差し引いた。ここで $\beta > 0$ は報酬に対するダイバージェンスの重みを表す。参照方策が $|\mathcal{A}|$ 個の行動上の一様分布であるとき、$D_{\mathrm{KL}}(\pi \| \text{uniform}) = \log|\mathcal{A}| - H(\pi)$ となる。ただし $H(\pi) = -\mathbb{E}_{a\sim\pi}[\log \pi(a)]$ は方策のエントロピーである。このときペナルティは、エントロピーボーナスと、どの方策も変えられない定数の和になる。同じことが Pendulum のトルク区間にも成り立つ。$[-2, 2]$ 上の一様密度は、その区間上のあらゆる密度の中で最大のエントロピー $\log 4 \approx 1.39$ nat をもち、$\log 4$ が $\log|\mathcal{A}|$ の役割を果たす。SAC はこのボーナスを毎ステップ適用する。

$$J(\pi) = \mathbb{E}_{\pi}\Big[\sum_{t} \gamma^t \big(r_t + \beta\, H(\pi(\cdot \mid s_t))\big)\Big]. \tag{15.40}$$

エージェントは訪れるすべての状態においてエントロピー 1 nat ごとに β 単位の報酬を受け取るため、**温度**（temperature）β は 1 nat あたりの報酬という単位をもつ。本節でも 15.3 節の文字 β をそのまま用いる。なお SAC の原論文では α と表記されているが、本書ではこれを時間差分学習（temporal-difference learning）のステップサイズに使っている。

15.2 節のアクター損失における任意追加のエントロピーボーナスとは異なり、この項は目的関数の一部である。そのため最適解そのものが変化し、複数の行動がほぼ同等に優れている場所では最良の方策もランダムなまま保たれる。連続行動の場合、H は密度にわたる $-\pi \log \pi$ の積分である微分エントロピー（differential entropy）となり、長さが 1 未満の区間に集中した密度に対しては負の値をとる。最大エントロピー強化学習（maximum-entropy reinforcement learning）は深層学習よりも古くからあり (Ziebart et al. 2008)、SAC はそのオフポリシー型アクタークリティックにあたる。

本章の他の手法と同様、SAC も一般化方策反復法（generalized policy iteration、Figure 14.9）と同じく 2 種類のステップを交互に繰り返す。方策評価では価値推定値を現在の方策の価値へと近づけ、方策改善ではその推定値が好む行動へと方策を動かす。エントロピーはこの両方のステップを変化させる。ネットワークを導入する前に、2 つの行動をもつ 1 つの状態でその仕組みを確認しよう。

15.5.1.2 手計算で見る 2 つの行動

価値が $Q = (1, 0)$ である 2 つの行動をもつ 1 つの状態を取り上げ、$\beta = 0.5$ とする。15.3 節で示したように、KL 正則化の最適解は参照方策に $e^{Q/\beta}$ を掛けて再正規化したものになり、その価値は $\beta \log Z$ となる。一様な参照方策のもとでは、次のようになる。

$$\pi^\star(a) = \frac{e^{Q(a)/\beta}}{Z}, \qquad Z = \sum_a e^{Q(a)/\beta},$$
$$\max_\pi \left(\mathbb{E}_\pi[Q] + \beta H(\pi)\right) = \beta \log Z.$$

ここで $\pi^\star = (e^2, 1)/(e^2 + 1) = (0.88, 0.12)$ である。そのエントロピーは 0.37 nat であり、$\mathbb{E}_{\pi^\star}[Q] = 0.88$ なので、ソフト価値（soft value）は $0.88 + 0.5 \cdot 0.37 = 1.06$ となって $0.5 \log(e^2 + 1)$ に一致する。このソフト価値は通常の最大値 1 を上回っている。すなわち、傾斜方策（tilted policy）は期待価値を 0.12 手放す代わりに、エントロピーボーナスで 0.18 を得ている。以降の議論ではもう 1 つの事実が重要になる。括弧内の項 $Q(a) - \beta \log \pi^\star(a)$ は両方の行動に対して同じ値をとり、$1 + 0.5 \cdot 0.13$ と $0 + 0.5 \cdot 2.13$ はいずれも 1.06 になる。最適解においては、サンプリングされた 1 つの行動から誤差なしでソフト価値が得られるのである。次のセルでは、$\pi^\star$ を一様方策およびほぼ貪欲な方策と比較する。

```
Q, beta = np.array([1.0, 0.0]), 0.5
pi_star = np.exp(Q / beta) / np.exp(Q / beta).sum()   # the tilted optimum
for name, pi in (('tilted', pi_star), ('uniform', np.array([0.5, 0.5])),
                 ('nearly greedy', np.array([0.99, 0.01]))):
    bracket = Q - beta * np.log(pi)        # one sampled action's soft value
    print(f'{name:>13}: pi = {pi.round(2)}, E[Q] + beta H = '
          f'{pi @ bracket:.3f}, bracket = {bracket.round(2)}')
print(f'beta log Z = {beta * np.log(np.exp(Q / beta).sum()):.3f}')
```

```
      tilted: pi = [0.88 0.12], E[Q] + beta H = 1.063, bracket = [1.06 1.06]
     uniform: pi = [0.5 0.5], E[Q] + beta H = 0.847, bracket = [1.35 0.35]
nearly greedy: pi = [0.99 0.01], E[Q] + beta H = 1.018, bracket = [1.01 2.3 ]
beta log Z = 1.063
```

傾斜方策は $\beta \log Z = 1.063$ を達成し、他の 2 つの方策はそれに届かず、ほぼ貪欲な方策では 1.018、一様方策では 0.847 にとどまる。これらの方策のもとでは括弧内の項が 2 つの行動の間で異なり（一様方策では 1 単位の差がある）、サンプリングされた単一の行動からはソフト価値のノイズを含んだ推定値しか得られない。この推定値が厳密になるのは最適解においてのみである。

15.5.1.3 ソフト方策評価

ある状態のソフト価値は、将来の報酬とともにその状態で得られるエントロピーも織り込み、$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a) - \beta \log \pi(a \mid s)\right]$ と表される。ここで $Q^\pi(s,a)$ はそれ以降のすべての状態のエントロピーボーナスを含むが、s 自身のボーナスは含まない。15.1 節のクリティックは 1 ステップのターゲット $r + \gamma V(s')$（終端状態では r に打ち切る）に対して回帰した。SAC のクリティックは Q を学習するため、次の状態の価値を Q を用いて表す必要があり、上の恒等式からサンプリングされた 1 つの行動を使ってそれを得る。Q_{w^-} をクリティック Q_w のゆっくり変化するコピー、すなわち 15.4 節のターゲットネットワークとする。これにより、回帰目標が更新ごとに動くのを防ぐ。ターゲットは次の式で与えられる。

$$
\begin{aligned}
y &= r + \gamma\,(1 - \mathbf{1}(s' \text{ terminal}))\Big(Q_{w^-}(s', \tilde{a}') - \beta \log \pi_\theta(\tilde{a}' \mid s')\Big), \\
\tilde{a}' &\sim \pi_\theta(\cdot \mid s').
\end{aligned}
\tag{15.41}
$$

唯一の新規項である $-\beta \log \pi_\theta(\tilde{a}' \mid s')$ は、いまサンプリングした行動を通じて推定された s' におけるエントロピーボーナスである。遷移 (s, a, r, s') はリプレイバッファから取り出されるが、$\tilde{a}'$ は現在の方策から新たにサンプリングされる。V^π は評価対象の方策のもとでの期待値だからである。バッファは過去に何が起きたかを与え、方策は次に何をするかを与える。

1 回のサンプリングによって現在の方策のソフト価値が推定されるため、このステップはソフト方策評価（soft policy evaluation）となる。これは傾斜最適方策の価値を計算する 15.3 節の log-sum-exp バックアップではない。両者が一致するのは現在の方策がその最適解にあるときだけであり、その場合は 2 行動の例で見たように括弧内の項がすべての行動に対して同じ値になる。ソフト評価はエントロピーボーナスで拡張された報酬に対する通常の方策評価にほかならないため、14.2 節の議論がそのまま成り立つ。すなわち、評価作用素を 1 回適用するごとに最大誤差は γ 倍に縮小し、繰り返し適用すると Q^π に収束する。この議論には拡張報酬が有界であることが必要であり、報酬だけでなくエントロピーも有界でなければならない。これは行動が有限個の場合や、密度が有界に保たれる連続方策の場合に成り立つため、本節ではこれを仮定する。

15.5.1.4 ソフト方策改善

方策改善ステップもまた 15.3 節から導かれる。同節では、任意の方策の正則化目的関数が最適解の価値から KL ダイバージェンスの分だけ下回ることを示した。

$$
\mathbb{E}_\pi[r] - \beta\, D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}}) = \beta \log Z - \beta\, D_{\mathrm{KL}}(\pi \| \pi^\star),
\tag{15.42}
$$

ここで $\pi^\star \propto \pi_{\mathrm{ref}}\, e^{r/\beta}$ は傾斜最適方策である。r を $Q(s, \cdot)$ に置き換え、参照方策を一様分布とすると、ある状態で $\mathbb{E}_\pi[Q] + \beta H(\pi)$ を最大化することは $D_{\mathrm{KL}}(\pi \| e^{Q/\beta}/Z)$ を最小化することと等価になる。両者の差である $\beta \log Z$ は π に依存しないからである。通常、ネットワークによる方策はこの傾斜分布を厳密には表現できないため、SAC は方策族の中でこのダイバージェンスの意味で最も近い要素へと方策を動かす。これが**逆 KL 射影**（reverse-KL projection）である。この向きのダイバージェンスは、$e^{Q/\beta}$ が小さい領域に方策が確率質量を置くことにペナルティを科すため、傾斜分布が離れた 2 つの峰をもつとき、ガウス分布は両方に広がるのではなく一方の峰に落ち着く（15.3.3.5 節）。行動をノイズの微分可能な関数として書くと、次のアクター損失が得られる。

$$
\begin{aligned}
L_\pi(\theta) &= \mathbb{E}_{s \sim \mathcal{D},\, z}\Big[\beta \log \pi_\theta(\tilde{a}_\theta(s, z) \mid s) - Q_w(s, \tilde{a}_\theta(s, z))\Big], \\
\tilde{a}_\theta(s, z) &= c \tanh\big(\mu_\theta(s) + \sigma_\theta(s)\, z\big),
\end{aligned}
\tag{15.43}
$$

ここで $z \sim \mathcal{N}(0, I)$ であり、状態はリプレイバッファ $\mathcal{D}$ からサンプリングされる。この損失は新たにサンプリングされた行動の負のソフト価値であるため、これを下げるとクリティックの価値とエントロピーがともに高まる。なお、行動を範囲内に収めるスクワッシュ $c \tanh$ については次項で扱う。この損失の勾配は 14.9 節のパスワイズ勾配である。ガウス行動 $a = \mu + \sigma z$ の場合、その勾配は $\nabla_\mu \mathbb{E}[Q(a)] = \mathbb{E}[Q'(\mu + \sigma z)]$ となる。すなわちスコアに重みを掛けるのではなく行動を通して微分するものであり、同節の例における 1 サンプルあたりの分散は 1.0 であった（スコア関数では 21.75）。この計算には、行動に関して微分可能なクリティックが必要になる。離散行動の場合は、代わりに行動全体にわたって和をとることで期待値を計算する。

もし各状態における最大化が厳密であれば、方策改善ステップはいたるところでソフト価値を引き上げる。

命題。報酬とエントロピーが有界であり、方策族 Π が π_{old} を含み、表形式と同様に各状態における方策を独立に選べるとする。$\pi_{\text{new}}(\cdot \mid s)$ が $\mathbb{E}_{a\sim\pi}\big[Q^{\pi_{\text{old}}}(s,a)\big] + \beta H(\pi(\cdot \mid s))$ を Π 上ですべての s において最大化するならば、いたるところで $V^{\pi_{\text{new}}} \geq V^{\pi_{\text{old}}}$ が成り立つ。

証明。H_t を π_{new} の s_t におけるエントロピーとする。$\pi_{\text{old}} \in \Pi$ であるから、$s_0 = s$ における最適性と $Q^{\pi_{\text{old}}}(s,a) = r(s,a) + \gamma\,\mathbb{E}[V^{\pi_{\text{old}}}(s_1)]$ を順に用いると、次の不等式が得られる。

$$\begin{aligned} V^{\pi_{\text{old}}}(s) &\leq \mathbb{E}_{a\sim\pi_{\text{new}}}\big[Q^{\pi_{\text{old}}}(s,a) - \beta \log \pi_{\text{new}}(a \mid s)\big] \\ &= \mathbb{E}_{\pi_{\text{new}}}\big[r_0 + \beta H_0 + \gamma\, V^{\pi_{\text{old}}}(s_1)\big]. \end{aligned}$$

同じ不等式を $V^{\pi_{\text{old}}}(s_1)$、次いで $V^{\pi_{\text{old}}}(s_2)$ というように n 回繰り返し適用すると、$V^{\pi_{\text{old}}}(s) \leq \mathbb{E}_{\pi_{\text{new}}}\big[\sum_{t<n} \gamma^t (r_t + \beta H_t)\big] + \gamma^n\,\mathbb{E}\big[V^{\pi_{\text{old}}}(s_n)\big]$ が得られる。価値は有界であるため、最後の項は $n \to \infty$ で消え、和は $V^{\pi_{\text{new}}}(s)$ に収束する。■

SAC はクリティックがまだ学習している最中に各損失に対して 1 回の勾配ステップをとるため、この命題は実際に走らせるループではなく厳密なアルゴリズムを記述したものである。Haarnoja, Zhou, Abbeel, et al. (2018) はその付録 B で完全な議論を与えている。

15.5.2 スクワッシュされたガウス方策

15.5.2.1 クリッピングとパスワイズ勾配

14.9 節のガウス方策は非有界なトルクをサンプリングし、環境側で $[-2, 2]$ にクリッピングしていた。行動を通して微分することのないスコア関数推定量にとっては、それで何の問題もなかった。しかし SAC のパスワイズ推定量は行動を通して微分するため、クリッピングは 2 つの問題を引き起こす。第 1 に、区間の外側ではクリップ操作の導関数が 0 になるため、境界のトルクが最良の行動である状態でさえ、境界を超えたサンプルからはアクターに勾配が一切伝わらない。第 2 に、クリッピングは各境界の外側にある確率をすべて境界そのものの上に移動させるため、クリッピングされた分布はそこで密度をもたず、Equation 15.41 と Equation 15.43 における $\log \pi$ が未定義になる。ガウス分布からの滑らかで可逆な写像による有界な分布を用いれば、これら両方の問題が解消される。Figure 15.12 に、クリッピングと以下で用いる写像 $2\tanh u$ を並べて示す。

15.5.2.2 tanh による変数変換

$u \sim \mathcal{N}\big(\mu_\theta(s), \sigma_\theta(s)^2\big)$ をサンプリングし、$a = c\tanh u$ によってスクワッシュする。この写像は滑らかな単調増加関数であるため、u の周りの短い区間と a の周りのその像は同じ確率をもち、像の長さは因子 $da/du = c\,(1 - \tanh^2 u)$ だけ短くなる。したがって a の密度は、座標ごとにガウス密度をこの因子で割ったものになる。

$$\log \pi(a \mid s) = \sum_i \Big[\log \mathcal{N}(u_i;\, \mu_i, \sigma_i) - \log\big(1 - \tanh^2 u_i\big) - \log c\Big]. \tag{15.44}$$

和の各項は、1 つの座標の圧縮に対してガウス分布の対数密度を補正している。Figure 15.13 に、ある平均と広がりに対するこの効果を示す。$a = 2$ の近くでは、スクワッシュによって u の上側の裾全体が行動の短い区間に詰め込まれるため、そこで密度が上昇する。この補正がないと、式はこの圧縮を見落とし、積分値が 1.13 となって 1 にならない。

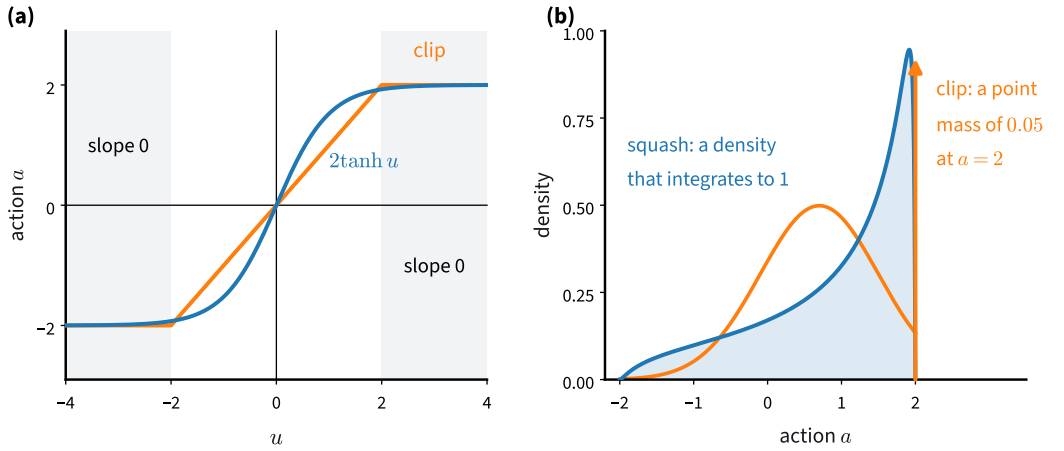


Figure 15.12: ガウスサンプル u の $[-2, 2]$ へのクリッピングとスクワッシュ。(a) クリッピングは境界の外側で傾きが 0 になるため、そこでのサンプルはアクターに勾配を与えないが、$2\tanh u$ はいたるところで正の傾きをもつ。(b) $u \sim \mathcal{N}(0.7, 0.8^2)$ の場合、クリッピングは上界を超える約 0.05 の確率を単一の行動 $a = 2$ の上に集めるため、クリッピングされた分布はそこで密度をもたない。一方、スクワッシュは区間全体にわたって密度を与える。

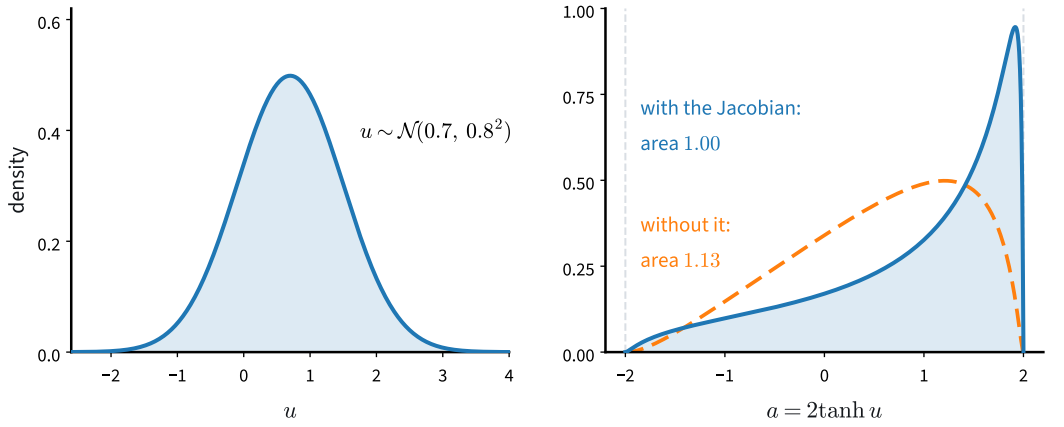


Figure 15.13: スクワッシュされたガウス分布。(a) $u \sim \mathcal{N}(0.7, 0.8^2)$ の密度。(b) 行動 $a = 2\tanh u$ の密度。Equation 15.44 のヤコビ行列の項を含めると、密度の積分値は 1 となり、$a = 2$ 付近（スクワッシュによって u の上側の裾が短い区間に圧縮される場所）で密度が上昇する。この項を省くと、式はガウス密度を単に $\operatorname{arctanh}(a/2)$ で評価したものになり、積分値は 1.13 となってその集中を見落とす。

$\log c$ の項はアクターの勾配を変えないが、すべての対数密度をシフトさせ、それに伴って報告されるエントロピーやターゲット内の項 $\beta \log \pi$ も $\log 2 \approx 0.69$ nat だけシフトさせる。報告されるエントロピーが実際のトルク分布のエントロピーになるよう、この項は残しておく。

方策ネットワークには 14.9 節の `GaussianHead` を用いる。**トランク**が特徴量を計算し、線形層がそれを平均 $\mu_\theta(s)$ に写像し、`state_std=True` の設定によって 2 つ目の線形層が $\log \sigma_\theta(s)$ を $[-5, 2]$ にクランプして出力する。SAC ではエントロピーボーナスがすべての状態で支払われるため、状態に依存する広がりが必要になる。真上で静止している振り子には狭い方策が必要である一方、振り上げの最中には広い方策を使えるが、単一の共有された σ ではその両方を実現できない。トランクには、後述するクリティックと同じ形状である 64 ユニットの ReLU 層 2 つを、14.9 節の単一の tanh 層の代わりに用いる。公開されている SAC の実装では、同じ形状で 256 ユニットが使われる。本節の実行では $\log \sigma$ が約 -3.5 より上に保たれるため、このクランプが制約として働くことはなく、取り除いても結果は変わらない。それでも保護策として残しておくのは、σ がゼロに近づくのを防ぎ、広がりの崩壊によってターゲット内の $\log \pi$ が発散しないようにするためである。ただし、平均がスクワッシュの飽和領域のずっと先まで行けばエントロピーはやはり下がるため、このクランプがスクワッシュ後の行動のエントロピーを下から抑えるわけではない。一般的な実装では $[-20, 2]$ にクランプする。

`SquashedGaussianPolicy` はこのヘッドをラップする。`sample` は $u = \mu + \sigma z$ をサンプリングし、損失の計算向けにスクワッシュ後の行動とその対数確率をいずれも微分可能な形で返す。`log_prob_u` はサンプラが保持しているスクワッシュ前の u から Equation 15.44 を評価するため、微分可能な経路上に逆双曲線正接は現れない。`act` は環境向けに NumPy 配列の行動をサンプリングし、`act_logp` は本節末尾の測定向けにその行動と対数確率を返す。これらはいずれもサンプル u を `_draw` から受け取る。また `act_greedy` はスクワッシュ後の平均 $c \tanh \mu_\theta(s)$ を返す。次節でこのクラスを再利用するため、ライブラリに保存しておく。

```
class SquashedGaussianPolicy(nn.Module):
    """d2l.GaussianHead with SAC's network (two ReLU layers and a
    state-dependent standard deviation), squashed through
    a = scale * tanh(u)."""
    def __init__(self, obs_dim, act_dim, hidden=64, scale=2.0, seed=None):
        super().__init__()
        with d2l.seeded(seed):
            trunk = nn.Sequential(nn.Linear(obs_dim, hidden), nn.ReLU(),
                                  nn.Linear(hidden, hidden), nn.ReLU())
            self.policy = d2l.GaussianHead(obs_dim, act_dim, hidden,
                                           state_std=True, trunk=trunk)
        self.scale = scale

    def log_prob_u(self, u, mean, std):
        """log pi at a = scale tanh(u), from the pre-squash u the sampler
        keeps."""
        logdet = 2 * (np.log(2) - u - nn.functional.softplus(-2 * u))
        return (torch.distributions.Normal(mean, std).log_prob(u)
                - logdet - np.log(self.scale)).sum(-1)

    def sample(self, obs):
        """A reparameterized action and its log-probability, differentiable."""
        mean, std = self.policy(obs)
        u = mean + std * torch.randn_like(std)
        return self.scale * torch.tanh(u), self.log_prob_u(u, mean, std)
```

```
    @torch.inference_mode()
    def _draw(self, obs, rng):
        """A pre-squash sample u with the head's mean and std, from numpy
        states and the noise of rng."""
        mean, std = self.policy(torch.as_tensor(obs))
        noise = rng.standard_normal(mean.shape, dtype=np.float32)
        return mean + std * torch.as_tensor(noise), mean, std

    def act(self, obs, rng):
        """A sampled action, numpy in and out."""
        return self.scale * np.tanh(self._draw(obs, rng)[0].numpy())

    @torch.inference_mode()
    def act_logp(self, obs, rng):
        """A sampled action and its log-probability, numpy in and out."""
        u, mean, std = self._draw(obs, rng)
        return (self.scale * np.tanh(u.numpy()),
                self.log_prob_u(u, mean, std).numpy())

    @torch.inference_mode()
    def act_greedy(self, obs, rng=None):
        """The squashed mean scale * tanh(mean), numpy in and out."""
        mean = self.policy(torch.as_tensor(obs))[0].numpy()
        return self.scale * np.tanh(mean)
```

log_prob_u の内部では、バッチにわたる平均をとる前に、ガウス項とヤコビ行列の項を行動の各座標にわたって足し合わせる。一方の和をとり他方の平均をとってしまうと、座標が 1 つだけの Pendulum では正しい値になるものの、より大きなあらゆる行動空間で誤った値になる。

15.5.2.3 数値的に安定な対数行列式

直接的な形である log(1 - tanh(u)**2) は、$\tanh u$ が 1 に近づくとほぼ等しい 2 つの数の引き算になり、float32 が $\tanh u$ をちょうど 1 に丸めると破綻する。$1-\tanh^2 u = 4e^{-2u}/(1+e^{-2u})^2$ であるから、対数をとると次式が得られる。

$$\log\left(1-\tanh^2 u\right) = 2\left(\log 2 - u - \operatorname{softplus}(-2u)\right), \tag{15.45}$$

これはすべての u に対して成り立ち、ほぼ等しい数どうしの引き算を含まない恒等式であり、log_prob_u ではこれを用いている。次の float32 による計算では、直接的な形、よく使われる 10^{-6} を加える保護付きの形、および安定な形を比較する。

```
u = np.float32([0, 3, 8, 10, 20])
t = np.tanh(u)
with np.errstate(divide='ignore'):
    naive = np.log(1 - t ** 2)
guarded = np.log(1 - t ** 2 + np.float32(1e-6))
stable = 2 * (np.log(np.float32(2)) - u - np.logaddexp(np.float32(0), -2 * u))
print(f'{"u":>4} {"naive":>10} {"guarded":>10} {"stable":>10}')
for row in zip(u, naive, guarded, stable):
    print(f'{row[0]:4.0f} {row[1]:10.4f} {row[2]:10.4f} {row[3]:10.4f}')
```

```
   u      naive    guarded     stable
   0     0.0000     0.0000     0.0000
   3    -4.6187    -4.6186    -4.6187
   8   -14.5561   -13.4256   -14.6137
  10       -inf   -13.8155   -18.6137
  20       -inf   -13.8155   -38.6137
```

直接的な形は $u = 8$ ですでにわずかにずれており（float32 では $1 - \tanh^2 u$ に残る有効桁がほとんどない）、さらには $-\infty$ が $u = 10$ 以降で生じる。保護項を加えれば無限大は避けられるものの、$u = 8$ で 1 nat 以上高くなりすぎ、$\log 10^{-6} = -13.8155$ の値に $u = 10$ 以降は留まるため、それ以上の飽和を検知できなくなる。その結果、収益曲線は正常に見えても、報告されるエントロピーと Equation 15.43 のエントロピー項が誤った値になる。

正規化条件から、変数変換（change of variables）に対する 2 つ目の独立した検証が得られる。行動の細かいグリッド上で密度を評価し、そのグリッド上でスクワッシュの逆変換を計算して、ヤコビ行列の項がある場合とない場合のそれぞれについて Equation 15.44 を台形公式で数値積分してみよう。

```
def squashed_logp(a, mu, sigma, logdet=True):
    up = np.arctanh(a / c)          # only here: the training path keeps u
    lp = (-0.5 * ((up - mu) / sigma) ** 2 - np.log(sigma)
          - 0.5 * np.log(2 * np.pi))
    if logdet:
        lp -= 2 * (np.log(2) - up - np.logaddexp(0, -2 * up)) + np.log(c)
    return lp

a_grid = np.linspace(-c + 1e-6, c - 1e-6, 200_001)
for m, s in ((0.0, 0.5), (0.7, 0.8)):
    w = np.trapezoid(np.exp(squashed_logp(a_grid, m, s)), a_grid)
    wo = np.trapezoid(np.exp(squashed_logp(a_grid, m, s, False)), a_grid)
    print(f'mu = {m}, sigma = {s}: integrates to {w:.6f} with the '
          f'log-det, {wo:.3f} without')
```

```
mu = 0.0, sigma = 0.5: integrates to 1.000000 with the log-det, 1.653 without
mu = 0.7, sigma = 0.8: integrates to 1.000000 with the log-det, 1.132 without
```

補正を含めれば、どちらの設定でも密度の積分値は 1.000000 になるが、補正なしでは積分値が 1.65 と 1.13 になる。この誤りがあっても Pendulum の収益はほとんど変わらない一方で、報告されるエントロピーはすべて誤った値になってしまうが、この検証なら一切訓練せずに誤りを見つけられる。なお、グリッドは 10^{-6} だけ $\pm c$ の手前で打ち切っている（境界では arctanh が無限大になるため）。この検証で適度な大きさの σ を使っているのはそのためである。

15.5.3 アルゴリズム

15.5.3.1 ツインクリティックと Polyak ターゲット

ノイズを含む推定値の最大値には上向きのバイアスが生じる（15.4.3.1 節）。たとえば同じ値に対する 2 つの独立な推定値が、それぞれ平均 0・標準偏差 σ のノイズでずれているとき、その最大値は真の値を平均して約 0.56σ 上回る。クリティックの勾配を登るアクターは、これと同じことを連続的に引き起こす。クリティックがたまたま過大評価した行動へとアクターが動き、ブートストラップによってその誤差が次のターゲットへと持ち越されるのである。TD3 に倣い、SAC は異なる初期化から 2 つのクリティック Q_{w_1} と Q_{w_2} を訓練し、それらの各点ごとの最小値を次のターゲットで用いる。

$$y = r + \gamma\,(1 - \mathbf{1}(s' \text{ terminal}))\Big(\min_{j=1,2} Q_{w_j^-}(s', \tilde{a}') - \beta \log \pi_\theta(\tilde{a}' \mid s')\Big), \tag{15.46}$$

同様に、アクター損失 Equation 15.43 においても $\min_j Q_{w_j}$ で Q_w を置き換える。両方のクリティックは同じターゲット y に対して回帰する。2 つの予測が食い違う場所では最小値が低い方を採用するため、これは意図的に悲観的な推定値となっている。本節の最後で、この悲観性がもたらすコストと、それが防ぐ現象を測定する。

DQN と同様に各クリティックはターゲット用のコピー w_j^- をもつが、15.4 節のように 250 環境ステップごとに上書きするのではない。毎回の更新のあと、オンラインの重みに向かってわずかな割合 ν だけ近づける。

$$w_j^- \leftarrow \nu\, w_j + (1 - \nu)\, w_j^-, \qquad \nu = 0.005.$$

この **Polyak 平均**（Polyak averaging）は半減期が $\ln 2/\nu \approx 139$ 回の更新である指数移動平均にあたり、回帰目標は静止したまま突然ジャンプするのではなくゆっくりとドリフトする。コピーをもつのはクリティックだけである。ターゲット内の行動 $\tilde{a}'$ は現在の方策からサンプリングされ、確率的方策は自身の広がりにわたってターゲットを平均化するため、行動に対する追加の平滑化は必要ない。Figure 15.14 に、1 回の更新を構成する各要素のつながりを示す。

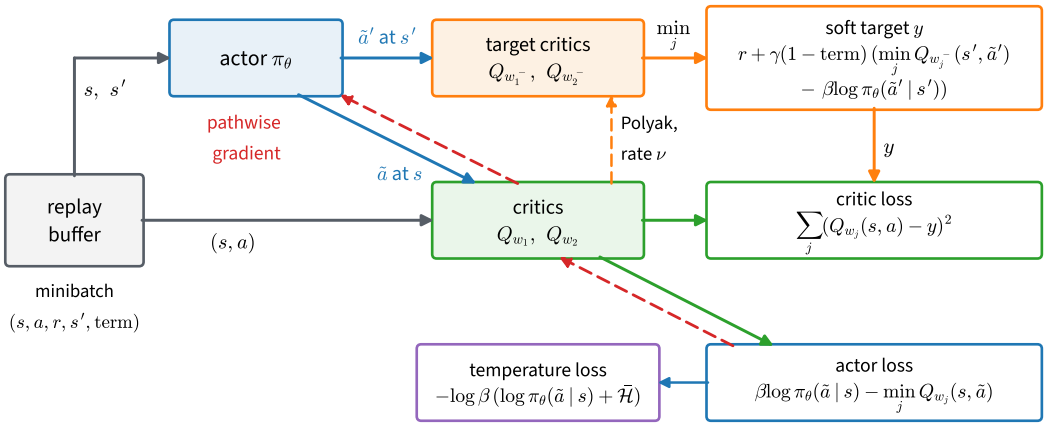


Figure 15.14: 1 回の SAC 更新。上段：ソフトターゲット y であり、新たな行動 $\tilde{a}'$（現在のアクターから s' でサンプリングしたもの）においてターゲットクリティックから計算される（2 つの最小値から $\beta \log \pi_\theta(\tilde{a}' \mid s')$ を引いたもの）。中段：クリティックが $Q_{w_j}(s, a)$ を y に回帰させる。下段：アクター損失はクリティックと新たな行動 $\tilde{a}$ を通ってアクターに届き（赤色、パスワイズ勾配）、温度は同じ対数確率に対する独自の損失をもつ。オレンジ色の破線矢印は Polyak 更新であり、ターゲットクリティックへ書き込む唯一のエッジである。

15.5.3.2 自動温度調整

固定された β は報酬とエントロピーの交換比率を固定するが、適切な比率は報酬の単位に依存する。すべての報酬を 10 倍することは、最適方策に対して β を 10 で割ることと同じ効果をもつため（演習問題 2）、あるタスク向けに調整した温度は、別の単位で測られた同じタスクには適さなくなる。SAC の第 2 論文 (Haarnoja, Zhou, Hartikainen, et al. 2018) では、代わりにエントロピ

ーを固定する。具体的には、訪問状態におけるエントロピーが目標値 $\bar{H}$ 以上となる最良の方策を次の制約付き問題として求める。

$$\max_{\pi} \ \mathbb{E}_{\pi}\Big[\sum_{t} \gamma^{t} r_{t}\Big] \quad \text{subject to} \quad \mathbb{E}_{s}\big[H(\pi(\cdot \mid s))\big] \geq \bar{H}, \tag{15.47}$$

そして β をこの制約のラグランジュ乗数として扱う。一般的な選択は $\bar{H} = -\dim \mathcal{A}$ であり、ここでは -1 nat となる。この乗数は、次の関数に対する勾配降下法によって学習する。

$$J(\beta) = -\log \beta \, \mathbb{E}_{s\sim\mathcal{D},\, a\sim\pi_\theta}\big[\log \pi_\theta(a \mid s) + \bar{H}\big], \tag{15.48}$$

この関数の $\log\beta$ に関する導関数は $\hat{H} - \bar{H}$、すなわちバッチのエントロピー推定値 $\hat{H} = -\mathbb{E}[\log \pi_\theta]$ が目標値をどれだけ上回っているかを表す量になる。したがって 1 回のステップは、方策が目標よりランダムな間は β を下げてランダムさを手放しやすくし、目標よりランダムでない間は β を引き上げる。$\log\beta$ を学習することで β を正に保ち、一般的な実装の既定値である $\beta = 1$ から開始して、ネットワークと同じ学習率 3×10^{-4} の Adam を用いる。勾配の符号が変わらない間、Adam は 1 ステップあたりほぼ学習率の分だけパラメータを動かすため、エントロピーが目標を上回り続ける限り、$\log\beta$ は 1 回の更新あたり約 3×10^{-4} ずつ減少する。6,000 回の更新後には $\beta \approx e^{-1.8} \approx 0.17$ になると予測されるが、これを後ほど実測された推移と比較する。

SAC の第 1 論文でもすでに 2 つのクリティックを訓練していたが、状態価値のための独立したネットワークを併用していた。本節で構築するアルゴリズムは、第 2 論文と同様に状態価値ネットワークをもたず温度を学習するものであり、一般的な実装で使われている版である。

15.5.3.3 更新ステップと訓練ループ

15.4 節のリプレイバッファである `d2l.ReplayBuffer` は、事前に確保した配列に遷移 $(s, a, r, s', \text{term})$ を保持する。`add` は 1 つの遷移を保存し、バッファが満杯になると最も古いものを上書きする。`sample` はバッチを一様にサンプリングして `d2l.Batch` として返す。その行動の列は整数を保持している。`ReplayBufferC` はその列を浮動小数点数のベクトルに広げるだけで、他は何も変更しない。方策と同様に、次節で再利用するためこれも保存しておく。

```
class ReplayBufferC(d2l.ReplayBuffer):
    """d2l.ReplayBuffer with the action column widened to float vectors."""
    def __init__(self, capacity, obs_dim, act_dim):
        super().__init__(capacity, obs_dim)
        self.act = np.zeros((capacity, act_dim), np.float32)
```

SAC は重点サンプリング比なしでオフポリシー学習を行う。次の行動を新たにサンプリングするため、クリティックのターゲット Equation 15.46 は遷移を収集した方策に依存せず、アクター損失も独自の行動をサンプリングするからである。もっとも、バッファ内の状態は古い方策が訪問したものであるため、リプレイによって損失の平均をとる状態の分布は変化しており、行動確率の比ではそのずれは補正されない。

方策と 1 つの状態価値ネットワークおよびそのオプティマイザからなる 14.9 節の `ActorCritic` コンテナは、行動価値クリティックとそのターゲットコピー、さらに温度をもつ SAC には合わない。そこで `SoftActorCritic` に、アクター、クリティックのリストとターゲットコピーのリスト、$\log\beta$、そしてアクター、クリティック、$\log\beta$ のそれぞれに対する Adam オプティマイザを保持させる。`beta=None` とすると温度は $\beta = 1$ から学習され、数値を渡すとその値に固定される。`beta`

は温度をスカラー配列として返す。min_q はクリティックのリスト（既定ではオンラインのもの）にわたる最小値を評価し、q_np は本節末尾の測定向けに NumPy 入力に対して同じ計算を行う。

```
class SoftActorCritic:
    """One actor, a list of action-value critics, their frozen copies, and
    the temperature beta = exp(log_beta), learned if beta is None."""
    def __init__(self, actor, qs, targets, lr, beta=None):
        self.actor, self.qs, self.targets = actor, qs, targets
        self.opt_pi = torch.optim.Adam(actor.parameters(), lr=lr)
        self.opt_q = torch.optim.Adam(
            [p for q in qs for p in q.parameters()], lr=lr)
        start = 1.0 if beta is None else beta   # a learned beta starts at 1
        self.log_beta = torch.tensor(float(np.log(start)),
                                     requires_grad=beta is None)
        self.opt_beta = (torch.optim.Adam([self.log_beta], lr=lr)
                         if beta is None else None)

    @property
    def beta(self):
        """The temperature exp(log_beta), a scalar tensor without gradient."""
        return self.log_beta.detach().exp()

    def min_q(self, obs, act, nets=None):
        """min_j Q_j(s, a) over the online critics, or over `nets`."""
        x = torch.cat([obs, act], -1)
        return torch.min(torch.stack(
            [q(x).squeeze(-1) for q in (self.qs if nets is None else nets)]),
            0).values

    def q_np(self, obs, act):
        """min_j Q_j(s, a), numpy in and out."""
        with torch.inference_mode():
            return self.min_q(torch.as_tensor(obs),
                              torch.as_tensor(act)).numpy()
```

make_critic は、連結された状態と行動から 1 つの数値を出力する、64 ユニットの ReLU 層 2 つを備えたクリティックを 1 つ構築する。make_sac は、ターゲットコピーがクリティックと同じ重みから始まるエージェントを構築する。

```
def make_critic(in_dim):
    """Q(s, a) from the concatenated state and action."""
    return nn.Sequential(nn.Linear(in_dim, 64), nn.ReLU(),
                         nn.Linear(64, 64), nn.ReLU(), nn.Linear(64, 1))

def make_sac(seed, obs_dim, act_dim, num_critics=2, lr=3e-4, beta=None):
    """A seeded agent whose target critics start as copies of the critics."""
    torch.manual_seed(seed)
    actor = SquashedGaussianPolicy(obs_dim, act_dim)
    qs = [make_critic(obs_dim + act_dim) for _ in range(num_critics)]
    return SoftActorCritic(actor, qs, [copy.deepcopy(q) for q in qs], lr,
                           beta)
```

1 回の更新は Figure 15.14 に従い、3 つの関数がその損失を担う。soft_target は現在の方策から

新たにサンプリングした行動でターゲット Equation 15.46 を計算する。これは回帰のためのデータであるため、そこへ勾配は流れない。critic_loss はすべてのクリティックをその単一のターゲットに回帰させ、actor_loss は新たに再パラメータ化された行動におけるクリティックの最小値を用いた Equation 15.43 であり、温度の損失が再利用する対数確率もあわせて返す。

```
def soft_target(agent, rew, next_obs, term, beta, gamma):
    """eq_sac_target_twin at a fresh action from the live policy."""
    with torch.no_grad():                    # the critic target is data
        a2, logp2 = agent.actor.sample(next_obs)
        return rew + gamma * (1 - term) * (
            agent.min_q(next_obs, a2, agent.targets) - beta * logp2)

def critic_loss(qs, obs, act, y):
    """Squared error of every critic against the one shared target y."""
    x = torch.cat([obs, act], -1)
    return sum(((q(x).squeeze(-1) - y) ** 2).mean() for q in qs)

def actor_loss(actor, min_q, obs, beta):
    """beta log pi - min_j Q_j at a fresh reparameterized action; also
    returns log pi."""
    a, logp = actor.sample(obs)
    return (beta * logp - min_q(obs, a)).mean(), logp
```

sac_step がこの更新を実行する。まずクリティックが critic_loss に対して 1 ステップ進み、次にアクターが actor_loss に対して 1 ステップ進む。エージェントが β を学習する場合、温度もアクターの対数確率を用いて $J(\beta)$ に対して 1 ステップ進む。最後にターゲットクリティックが ν（コード内では polyak）の割合だけ移動する。このステップは $\log \pi_\theta$ のバッチ平均（その負の値が方策のエントロピーの推定値になる）を返すが、訓練ループが更新ごとに待機しなくて済むようデバイス上に置いたままにする。

```
def descend(opt, loss):
    """One optimizer step on a loss."""
    opt.zero_grad()
    loss.backward()
    opt.step()

def sac_step(agent, obs, act, rew, next_obs, term, gamma, polyak,
             target_entropy):
    """One SAC update: soft critic regression, pathwise actor step, Polyak
    averaging, and, if the agent learns beta, one temperature step toward
    the target entropy. Returns the batch mean of log pi."""
    beta = agent.beta
    y = soft_target(agent, rew, next_obs, term, beta, gamma)
    descend(agent.opt_q, critic_loss(agent.qs, obs, act, y))
    loss_pi, logp = actor_loss(agent.actor, agent.min_q, obs, beta)
    descend(agent.opt_pi, loss_pi)
    if agent.opt_beta is not None:   # entropy above target: beta falls
        descend(agent.opt_beta, -agent.log_beta * (logp.detach().mean()
                                                   + target_entropy))
    with torch.no_grad():                    # Polyak: the drifting copy
        for q, tnet in zip(agent.qs, agent.targets):
            for pt, pq in zip(tnet.parameters(), q.parameters()):
```

```
                pt.lerp_(pq, polyak)
    return logp.detach().mean()
```

`descend` は損失に対してオプティマイザを 1 ステップ進め、`lerp_` は各ターゲットの重みを対応するクリティックの重みに向かって割合 `polyak` だけ動かす。

`train_sac` は Pendulum 上で 1 つのエージェントを実行する。最初の `warmup` ステップでは一様ランダムなトルクをとり、その後はアクターが行動を選択し、各環境ステップのあとに `utd` 回の更新を行う。この**更新対データ比**（update-to-data ratio）は、1 環境ステップあたりに行う勾配ステップの数である。ループは `terminated` を保存し、`truncated` は保存しない。Pendulum には終端状態がなく、エピソードは 200 ステップの時間制限でのみ終了するため、保存されるフラグは常に 0 となり、時間制限で求められる通りクリティックは打ち切りをまたいでブートストラップする（14.9 節）。各エピソードの終わりに、ループはステップ数、そのエピソードの収益、エントロピーの推定値としてエピソード内の更新にわたる $-\log \pi_\theta$ の平均、および現在の β を生成する。以降のすべての実験でこの単一のループ（15.4 節の `train_dqn` のオフポリシーループを連続行動にしたもの）を使えるよう、更新比とバッファは引数として渡す。`make_update` は `sac_step` をエージェントに束縛し、サンプリングされた `d2l.Batch` を配列に変換する。また実行時のサンプリングノイズにもシードを設定するため、単独で実行しても他の実行のあとに続けて実行しても同じ結果が得られる。

```
def make_update(agent, seed, **hp):
    """sac_step bound to the agent, on a d2l.Batch; seeds this run's
    sampling noise."""
    torch.manual_seed(seed)
    def update(b):
        return sac_step(agent, *map(torch.as_tensor, (
            b.obs, b.act, b.rew, b.next_obs, b.term)), **hp)
    return update
```

```
def train_sac(seed, agent, buffer, num_env_steps, utd=1, warmup=1_000,
              batch_size=256, gamma=0.99, polyak=0.005, target_entropy=-1.0):
    """SAC on Pendulum: `warmup` uniformly random steps, then `utd` updates
    per environment step. Yields (env step, episode return, policy
    entropy, beta) at the end of every episode."""
    update = make_update(agent, seed, gamma=gamma, polyak=polyak,
                         target_entropy=target_entropy)
    rng, env = np.random.default_rng(seed), gym.make('Pendulum-v1')
    scale, act_dim = agent.actor.scale, buffer.act.shape[1]
    obs, ep_return = env.reset(seed=seed)[0], 0.0
    logp_sum, n_upd = 0.0, 0
    for t in range(1, num_env_steps + 1):
        a = (rng.uniform(-scale, scale, act_dim).astype(np.float32)
             if t <= warmup else agent.actor.act(obs, rng))
        next_obs, rew, terminated, truncated, _ = env.step(a)
        buffer.add(obs, a, rew, next_obs, float(terminated))
        obs, ep_return = next_obs, ep_return + rew
        for _ in range(utd if t >= warmup else 0):
            logp_sum = logp_sum + update(buffer.sample(batch_size, rng))
            n_upd += 1
        if terminated or truncated:   # read the device sum once per episode
```

```
            yield (t, ep_return, float(-logp_sum / n_upd) if n_upd else np.nan,
                   float(agent.beta))
            obs, ep_return = env.reset()[0], 0.0
            logp_sum, n_upd = 0.0, 0
```

run_arm は 1 つの設定を複数のシードで訓練し、後の測定のためにエージェントを保持する。そのバッファは 1 回の実行のすべての遷移を保持する。まずはツインクリティックと学習される温度を備えた 2 つの設定、すなわち 1 環境ステップあたり 1 回の更新で 20,000 ステップ行う設定と、1 ステップあたり 10 回の更新で 4,000 ステップ行う設定から始める。

```
def run_arm(seeds, num_env_steps, utd=1, num_critics=2, beta=None):
    """Train one SAC configuration per seed. Returns the agents and, per
    seed, an array of per-episode rows (step, return, entropy, beta)."""
    agents = [make_sac(seed, 3, 1, num_critics, beta=beta) for seed in seeds]
    runs = [np.array(list(train_sac(seed, agent,
                                    ReplayBufferC(num_env_steps, 3, 1),
                                    num_env_steps, utd=utd)))
            for seed, agent in zip(seeds, agents)]
    return agents, runs

seeds, agents, runs = range(num_seeds), {}, {}
agents['UTD 1'], runs['UTD 1'] = run_arm(seeds, num_env_steps)
agents['UTD 10'], runs['UTD 10'] = run_arm(seeds, 4_000, utd=10)
```

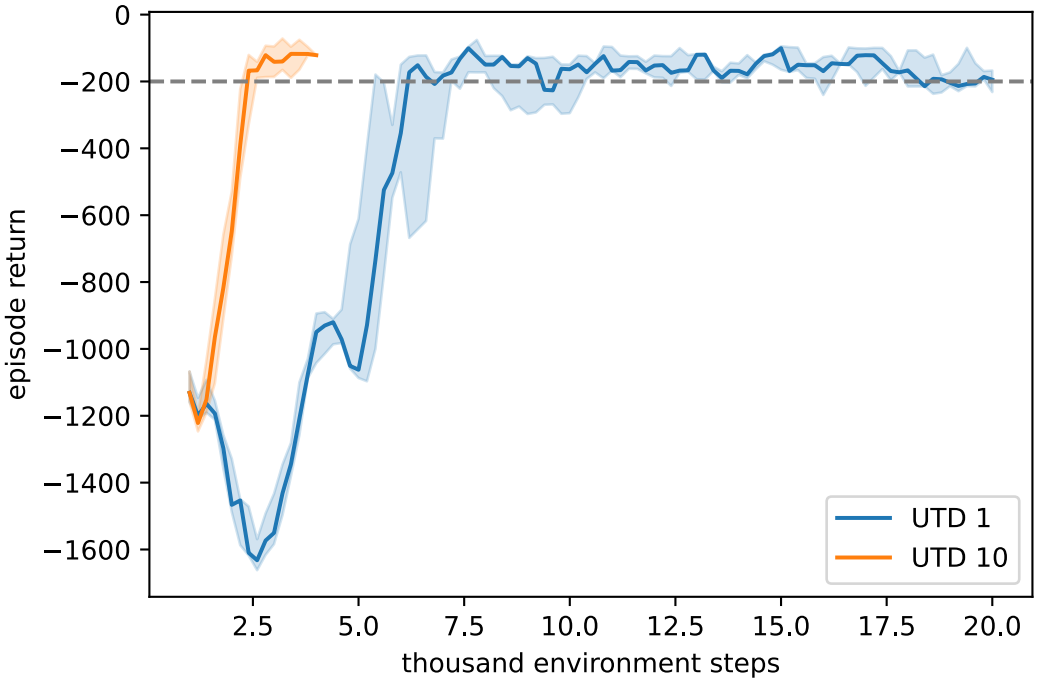
15.5.4 サンプル効率と較正

15.5.4.1 環境ステップに対する収益

次のプロットは、直近 5 エピソードにわたって平均した各エピソードの収益を環境ステップに対して描いたものであり、振り上げて静止させる制御器の水準として −200 に印を付けている。crossings は各条件のシードごとに、5 エピソード平均が初めてその水準に達するステップ数を d2l.steps_to（15.1 節）で求めて並べる（一度も到達しない実行があれば inf となる）。

```
def crossings(arm):
    """Per seed of an arm, the env step at which the five-episode average
    return first reaches -200."""
    return ', '.join(f'{d2l.steps_to(r, -200, 5):.0f}' for r in runs[arm])

grid = np.arange(1_000, num_env_steps + 1, 200)
curves = {arm: d2l.on_grid(runs[arm], grid, 1, smooth=5) for arm in runs}
curves['UTD 10'][:, grid > 4_000] = np.nan      # those runs end at 4,000
d2l.plot_curves(curves, x=grid / 1000, xlabel='thousand environment steps',
                ylabel='episode return', reference=-200)
for arm in runs:
    print(f'{arm}: steps to a five-episode average of -200, per seed '
          f'{crossings(arm)}')
```

```
UTD 1: steps to a five-episode average of -200, per seed 6200, 5400, 6000
UTD 10: steps to a five-episode average of -200, per seed 2400, 2400, 2600
```



1 環境ステップあたり 1 回の更新では、すべてのシードが約 5,000〜11,000 環境ステップ後に -200 を超える。学習されたベースライン付きの REINFORCE が同じタスクで同じ水準を超えたのは数十万ステップ後であり（14.9 節）、1〜2 桁多いステップを要していた。この差は 3 つの変更によって説明できる。第 1 に、REINFORCE が 1 回の更新に 1,600 ステップを費やしていたのに対し、リプレイによって各環境ステップごとに勾配ステップを実行できる。第 2 に、ブートストラップされたターゲットによって、クリティックはエピソード全体の収益ではなく単一の遷移から学習できる。第 3 に、パスワイズ勾配によって、トルクに応じて価値がどう変化するかというクリティックの知識がそのままアクターへ伝わる。

曲線は上昇する前にいったん下降する。ランダムなウォームアップのトルクによるエピソードの収益は約 $-1{,}200$ であるが、アクターが引き継ぐと、1,500〜3,000 ステップの間で収益はランダムより悪い約 $-1{,}600$ まで落ち込んでから上昇に転じる。これほど大きなコストを生む振る舞いには 2 つの可能性がある。アクターが振り子を回転させ続けて速度コストを被っているか、振り子を真下に保って最大の角度コストを被っているかである。両者を見分けるため、方策が振り子に何をしているかを測定する。`behavior` は 10 エピソードを実行し、平均収益、$\cos\vartheta$ の平均（真下で -1、真上で 1 となる）、および平均速さ $|\dot{\vartheta}|$ を報告する。比較基準として、トルク 0、運動方向に沿って常にトルク $2\,\mathrm{sign}(\dot{\vartheta})$ で押し続ける回転制御器、およびあらゆる運動に逆らう減衰制御器 $-2\tanh(2\dot{\vartheta})$ の 3 つの固定制御器を用いる。

```
def behavior(policy, num_episodes=10, seed=0):
    """Mean return, cos(theta) and |theta dot| over a policy's episodes."""
    env, rng = gym.make('Pendulum-v1'), np.random.default_rng(seed)
    env.reset(seed=seed)
    ret, states = 0.0, []
    for _ in range(num_episodes):
        obs, done = env.reset()[0], False
        while not done:
```

```
            obs, rew, terminated, truncated, _ = env.step(policy(obs, rng))
            done, ret = terminated or truncated, ret + rew
            states.append(obs)
        s = np.array(states)
        return ret / num_episodes, s[:, 0].mean(), np.abs(s[:, 2]).mean()

    policies = {'zero torque': lambda obs, rng: np.zeros(1, np.float32),
                'spin': lambda obs, rng: 2 * np.sign(obs[2:]),
                'damp': lambda obs, rng: -2 * np.tanh(2 * obs[2:])}
```

これらを、落ち込みの最中にある 2,000 ステップ後の各シードのアクター、および訓練済みエージェントと比較する。各実行は独自のシードを設定するため、同じシードを再び 2,000 ステップ訓練すれば上の実行の序盤がそのまま再現される。ここではアクターの平均行動 $c \tanh \mu_\theta(s)$ を測定する。

```
    for seed in seeds:
        agent = make_sac(seed, 3, 1)
        for _ in train_sac(seed, agent, ReplayBufferC(2_000, 3, 1), 2_000):
            pass
        policies[f'SAC at 2k, seed {seed}'] = agent.actor.act_greedy
    policies['SAC at 20k, seed 0'] = agents['UTD 1'][0].actor.act_greedy
    print(f'{"policy":>18} {"return":>8} {"cos theta":>10} {"|theta dot|":>12}')
    for name, policy in policies.items():
        ret, cos, speed = behavior(policy)
        print(f'{name:>18} {ret:8.0f} {cos:10.2f} {speed:12.2f}')
```

```
            policy   return  cos theta  |theta dot|
       zero torque    -1290      -0.49         2.50
              spin    -1610       0.04         6.83
              damp    -1886      -0.96         0.22
 SAC at 2k, seed 0    -1789      -0.93         0.55
 SAC at 2k, seed 1    -1769      -0.93         0.69
 SAC at 2k, seed 2    -1784      -0.93         0.63
SAC at 20k, seed 0     -167       0.83         0.58
```

2,000 ステップ後の時点では、どのアクターも振り子を真下付近（$\cos\vartheta$ が約 -0.9）に保ち、速さも毎秒約 1 ラジアン以下でほぼ静止させている。訓練済みエージェントが振り子を真上付近に保つのに対し、序盤のアクターは 3 つの比較基準の中で最もスコアの悪い減衰制御器のように振る舞っている。回転制御器も同程度の収益になるが、高速で $\cos\vartheta$ が 0 付近になるというまったく異なる運動をしており、そのような動きを示すアクターは 1 つもない。つまりアクターはまず揺れを減衰させることを学習し、その結果として最もコストの高い位置に振り子を垂れ下げたままにし、あとになってようやくエネルギーを注入して振り上げることを学習するのである。全 20 回の実行すべてにおいて、2,000 ステップ後のアクターの平均行動は同様に振り子を真下に保っていた。個別に検証してはいないが、考えられる理由は次のとおりである。初期のクリティックはまだ数ステップ先までしかコストを伝播できておらず、数ステップの間に行動が変えられるのは速度コストとトルクコストであるため、減衰させるのが最良に見えるのである。

2 本目の曲線は、1 環境ステップあたり 10 回の更新を行う設定である。1 ステップあたり 1 回の更新での 5,400〜11,200 ステップに対し、2,400〜4,000 環境ステップ後に -200 を超えており、ステップ数は約 2.5 分の 1 に減っている。収益の落ち込みも 1,200 ステップ付近と早く訪れ、1,600 ステップまでには解消される。環境ステップの節約には計算コストがかかる。水準を超えるまでに高い更新比は約 3 倍の勾配ステップを要し、4,000 ステップの実行全体では 30,000 回の勾配ス

テップとなり、これは 20,000 ステップの実行における 19,000 回の約 1.6 倍にあたる。高い更新比は計算量と環境との相互作用を交換するものであり、まさにロボットアームが求めるトレードオフである。最初の 1,000 ステップはランダムであり、5 エピソード平均は 2,000 ステップより前には -200 に到達できないため、この方法で 2,400 ステップよりさらに短縮できる余地はほとんどない。より難しいタスクでは、この交換は 2 つの限界に突き当たる。1 つは以下で調べる過大評価である。もう 1 つは**初期バイアス**（primacy bias）であり、最初の少数の遷移に対して何度も更新されたネットワークがそれらに過剰適合し、後から得られるデータからうまく学習できなくなる現象である。これに対しては、バッファを保持したままネットワークを定期的にリセットするのが標準的な対処法となっている (Nikishin et al. 2022)。

15.5.4.2 エントロピーと温度

学習される温度の働きを見るため、多くの実装で使われる固定値 $\beta = 0.2$ と、1 環境ステップあたり 1 回の更新による追加の 3 回の実行で比較する。

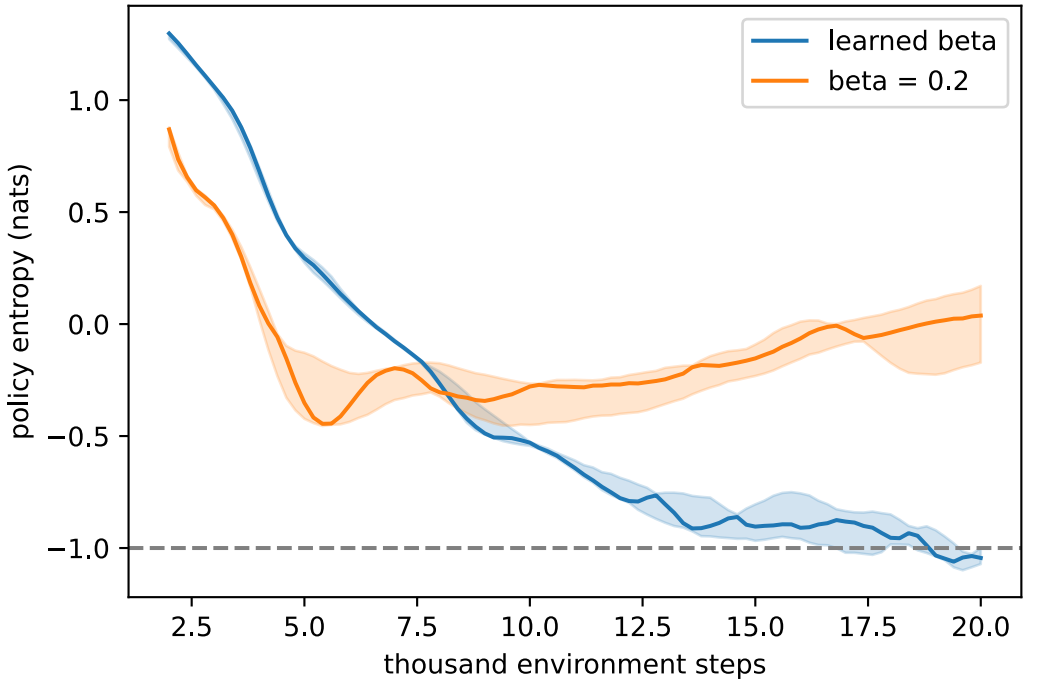
```
agents['beta = 0.2'], runs['beta = 0.2'] = run_arm(seeds, num_env_steps,
                                                   beta=0.2)
print(f'beta = 0.2: steps to -200, per seed {crossings("beta = 0.2")}')
```

```
beta = 0.2: steps to -200, per seed 6200, 6800, 6400
```

固定温度でも学習される温度と同じ範囲のステップ数で -200 を超え、どちらが一貫して早いということはなかった。目標値 -1 nat に対する両者のエントロピー推定値は次のようになる。

```
pair = {'learned beta': runs['UTD 1'], 'beta = 0.2': runs['beta = 0.2']}
grid = np.arange(2_000, num_env_steps + 1, 200)
d2l.plot_curves({arm: d2l.on_grid(r, grid, 2, smooth=5)
                 for arm, r in pair.items()},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='policy entropy (nats)', reference=-1)
for arm, r in pair.items():
    print(f'{arm}: entropy over the last 20 episodes, per seed '
          f'{np.round([np.nanmean(x[-20:, 2]) for x in r], 2)}')
```

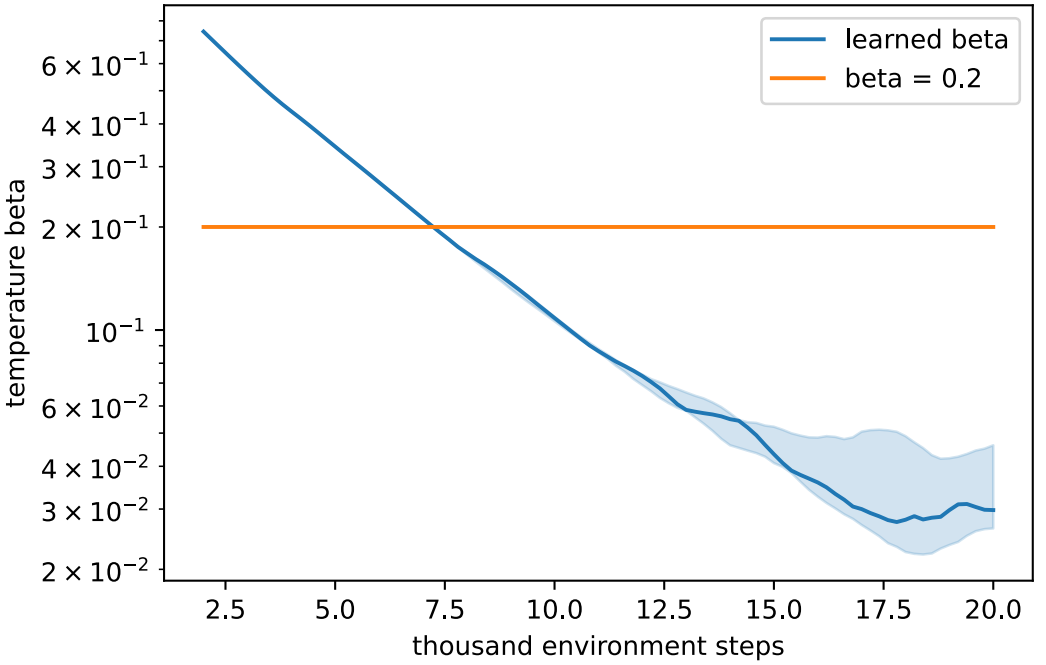
```
learned beta: entropy over the last 20 episodes, per seed [-0.97 -0.99 -0.95]
beta = 0.2: entropy over the last 20 episodes, per seed [-0.15 -0.02  0.09]
```



温度を学習する場合、エントロピーは 1 nat 超から目標値へと着実に下がり、約 15,000 ステップ以降は目標値の近くに留まる。固定の $\beta = 0.2$ では、振り上げを学習している間にゼロを数十分の 1 nat 下回るまで落ち込み、その後再び上昇する。振り子が静止できるようになると、最良のトルクの周りでクリティックがほぼ平坦になるため、方策を広げても価値はほとんど損なわれない。すなわち、行動に関して曲率 h の二次関数となっているクリティックに対しては最良のガウス分布が分散 β/h をもち（演習問題 2）、クリティックが平坦なほど広い方策が許容される。20,000 ステップの時点でもこの上昇はまだ続いており、固定温度における最終的なエントロピーは実行をいつ止めるかに依存してしまう。対数スケールで描いた両設定の温度は次のとおりである。

```
d2l.plot_curves({arm: d2l.on_grid(r, grid, 3) for arm, r in pair.items()},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='temperature beta', yscale='log')
print('learned beta, median over seeds at 2k, 7k and 20k steps:',
      np.median(d2l.on_grid(pair['learned beta'], [2_000, 7_000, 20_000], 3),
                axis=0).round(3))
```

```
learned beta, median over seeds at 2k, 7k and 20k steps: [0.744 0.212 0.03 ]
```



約 10,000 ステップまで、学習される β は 1 から対数スケール上の直線に沿って減少する。これはエントロピーが目標値を上回っている間、Adam が 1 回の更新あたりほぼ学習率の分だけステップを進めるためである。その値は 0.2 付近（実行が -200 を超える時点）にあり、6,000 回の更新後に 0.17 になるという予測に近い。エントロピーが目標値に達すると減少は緩やかになり、最終的に β は数百分の 1 程度に落ち着く。クリティックが平坦になるにつれて β を下げることで、学習される温度はエントロピーの落ち込みと反発を取り除くが、振り子の学習速度自体は変えない。

訓練中の収益は確率的方策から得られる。エントロピー自体がその方策の最大化する対象の一部だからである。これに加え、すべての訓練済みエージェントの平均行動 $c \tanh \mu_\theta(s)$ も、それぞれ 20 エピソードにわたって評価してみよう。

```
env = gym.make('Pendulum-v1')
for arm in runs:
    greedy_return, cost = [], []
    for seed, agent in zip(seeds, agents[arm]):
        env.reset(seed=100 + seed)
        g = d2l.evaluate(env, agent.actor.act_greedy, num_episodes=20)
        env.reset(seed=100 + seed)
        s = d2l.evaluate(env, agent.actor.act, num_episodes=20,
                         rng=np.random.default_rng(1000 + seed))
        greedy_return.append(g)
        cost.append(g - s)
    print(f'{arm:>10}: mean action {np.round(greedy_return)}, '
          f'cost of sampling {np.round(cost, 1)}')
```

```
     UTD 1: mean action [-188. -147. -184.], cost of sampling [-5.9  0.2  0.1]
    UTD 10: mean action [-186. -139. -185.], cost of sampling [0.3 0.1 0. ]
beta = 0.2: mean action [-179. -142. -184.], cost of sampling [ 7.2  6.6 -0.8]
```

どのエージェントの平均行動も約 -120〜-190 の収益を達成しており、代わりに方策からサンプリングしても収益の変化は高々約 10 ポイントにとどまる。学習された状態依存の広がりによって、収益をほとんど損なうことなく方策のランダムさが保たれている。

15.5.4.3 クリティックの較正

アクターはクリティックを登るため、クリティックの誤差はそのままアクターの誤差になる。2 つのクリティックの最小値が本来の目的を果たしているかを確かめるため、エピソードの最初の状態における予測と、その後に方策が実際に集める値とを比較する。各エージェントについて確率的なエピソードを 20 回実行し、$\min_j Q_{w_j}(s_0, a_0)$ とともに、ソフトクリティックが予測対象とする実現割引ソフト収益 $r_0 + \sum_{t \geq 1} \gamma^t (r_t - \beta \log \pi_\theta(a_t \mid s_t))$ を記録する。なお、コード中の実現収益には $-\beta \log \pi_\theta(a_0 \mid s_0)$ も含めているが、その平均は β と s_0 でのエントロピーの積であり、これらのエージェントでは大きさが 0.1 未満にとどまる。また、実現収益はステップ 200 で打ち切られるのに対し、クリティックは時間制限をまたいでブートストラップし無限期間の収益を予測する。打ち切られた裾は $\gamma^{200} \approx 0.13$ にステップ 200 での価値を掛けたものだが、静止した振り子ではその価値がほぼゼロになる。

最小値をとらないクリティックを調べるため、上のツインクリティックが最も速く学習した設定である 1 環境ステップあたり 10 回の更新で、単一クリティックのエージェントを訓練する。

```
agents['single'], runs['single'] = run_arm(seeds, 4_000, utd=10,
                                           num_critics=1)
print(f'single critic, UTD 10: steps to -200, per seed '
      f'{crossings("single")}')
```

```
single critic, UTD 10: steps to -200, per seed 2800, 2600, 2600
```

この更新比では、単一クリティックもツインクリティックと同じ範囲のステップ数で -200 に達するため、収益だけでは両者を区別できない。しかし較正（calibration）を調べると違いが現れる。

```
def calibration(agent, seed, gamma=0.99, num_episodes=20):
    """Mean predicted min_j Q(s0, a0) and mean realized discounted soft
    return, at the agent's temperature beta."""
    rng, env = np.random.default_rng(seed), gym.make('Pendulum-v1')
    env.reset(seed=seed)
    beta, preds, softs = float(agent.beta), [], []
    for _ in range(num_episodes):
        obs, done, disc, soft = env.reset()[0], False, 1.0, 0.0
        a, logp = agent.actor.act_logp(obs, rng)
        preds.append(float(agent.q_np(obs, a)))
        while not done:
            obs, rew, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            soft += disc * (rew - beta * logp)
            disc *= gamma
            if not done:
                a, logp = agent.actor.act_logp(obs, rng)
        softs.append(soft)
    return np.mean(preds), np.mean(softs)
```

```
    for name, arm in (('twin, UTD 1', 'UTD 1'), ('twin, UTD 10', 'UTD 10'),
                      ('single, UTD 10', 'single')):
        for seed, agent in zip(seeds, agents[arm]):
            pred, soft = calibration(agent, 200 + seed)
            print(f'{name:>14}, seed {seed}: predicted {pred:7.1f}, '
                  f'realized {soft:7.1f}, gap {pred - soft:+6.1f}')
```

```
   twin, UTD 1, seed 0: predicted  -154.6, realized  -114.3, gap  -40.3
   twin, UTD 1, seed 1: predicted  -113.2, realized   -94.8, gap  -18.4
   twin, UTD 1, seed 2: predicted  -172.0, realized  -138.7, gap  -33.3
  twin, UTD 10, seed 0: predicted  -136.7, realized  -116.4, gap  -20.3
  twin, UTD 10, seed 1: predicted   -88.3, realized   -93.1, gap   +4.8
  twin, UTD 10, seed 2: predicted  -158.9, realized  -140.4, gap  -18.5
single, UTD 10, seed 0: predicted   -68.1, realized  -119.7, gap  +51.6
single, UTD 10, seed 1: predicted   -57.5, realized   -94.9, gap  +37.4
single, UTD 10, seed 2: predicted  -105.9, realized  -143.4, gap  +37.5
```

ツインクリティックの最小値は、ほぼすべての実行で実現収益を下回り、残りの実行でも上回るのは高々数ポイントにとどまる。一方、単一クリティックはすべての実行でツインの最小値より高く予測し、大半の実行で実現収益をも上回る。各設定あたり 20 回の実行にわたってこれらの符号の傾向はそのとおりに保たれた一方、乖離の大きさは実行ごとに数十ポイント単位でばらついた。したがって最小値をとることは、ここで単一クリティックが見せた過大評価を防ぐ一方で、最小値自体は過小な予測を行う。なお Pendulum では、どちらであっても収益は変わらない。より高い更新比によって単一クリティックの過大評価がさらに強まるかどうかは、演習問題 6 で扱う。

こうして振り子は数千環境ステップから学習できるようになり、さらに更新比を高めれば計算量と引き換えに環境ステップをもう一段減らせるようになった。それでも各遷移がクリティックに教えるのは 1 回の回帰のターゲットという 1 つの数値だけであり、1 ステップあたり 10 回の更新を行っても振り上げには数千個の遷移を要した。ブランコを数回押したことのある人間なら、次にブランコがどこへ動くかを予測できる。15.6 節では、遷移から振り子の物理法則を学習するエージェントが、数百ステップから振り上げを計画できるかどうかを問う。

15.5.5 まとめ

SAC はオフポリシー型アクタークリティックによって報酬とエントロピーの和を最大化する。ソフト方策評価はクリティックのブートストラップターゲットに $-\beta \log \pi$ を加え、ソフト方策改善はクリティックを通るパスワイズ勾配によって $e^{Q/\beta}$ の逆 KL 射影へとアクターを動かす。tanh スクワッシュによって行動を有界に保ち、その対数密度には安定な形で計算された変数変換の項が必要となる。Polyak 平均によるコピーを備えたツインクリティックがクリティックの過大評価を防ぎ、学習される温度は報酬とエントロピーの交換比率を固定する代わりにエントロピーを目標値に保つ。Pendulum において、SAC は 1 ステップあたり 1 回の更新で約 5,000〜11,000 環境ステップ以内、10 回の更新で約 3,000 環境ステップ以内に振り上げを達成し、これは REINFORCE より 1〜2 桁少ないステップ数である。

15.5.6 演習問題

1. **ターゲット内のエントロピー**。クリティックは、項 $-\beta \log \pi_\theta$（Equation 15.41）によって後続の状態で得られるエントロピーを価値に含めるが、15.2 節のエントロピーボーナスはアクター損失にしか作用しない。開始状態 s_0 に L と R の 2 つの行動があり、どちらの報酬も 0 であるとする。行動 L は状態 s_L（選択可能な行動は 1 つだけ）に遷移し、行動 R は状

態 s_R（選択可能な行動は n 個）に遷移する。s_L と s_R におけるすべての行動は報酬が 0 であり、エピソードを終了させる。

1. Equation 15.40 のもとで、s_L と s_R の最適ソフト価値、および s_0 における R の最適確率を求めよ。$n = 10$、$\gamma = 0.99$ のときにその確率を評価せよ。この確率は β にどう依存するか。
2. クリティックのターゲットから $-\beta \log \pi_\theta$ を省く一方、アクター損失には $\beta \log \pi_\theta$ を残すとする。アクターは R のどの確率へ収束し、この変種はどの目的関数を最適化するか。
3. 各状態で選択可能な行動上の一様分布を参照方策とする 15.3 節の KL ペナルティは、定数の差を除いてエントロピーボーナスに等しい。このペナルティを用いて小問 a を繰り返せ。ここではその定数が最適方策を変えるのに、Pendulum では最適方策を変えないのはなぜか。

2. [code] **二次クリティック**。最良の行動 $a^\star$ の近傍で、クリティックが 1 次元の行動に関する二次関数 $Q(s,a) = Q(s,a^\star) - \frac{h}{2}(a-a^\star)^2$（曲率 $h > 0$）であり、方策がスクワッシュなしのガウス分布 $\mathcal{N}(\mu, \sigma^2)$ であるとする。

1. $\mathbb{E}_{a\sim\pi}[Q(s,a)] + \beta H(\pi)$ を μ と σ に関して最大化せよ。最大点が傾斜分布 $e^{Q(s,\cdot)/\beta}/Z$ となり、したがって 15.5.1.4 節の逆 KL 射影がこの分布族に対して厳密になることを示せ。
2. 最適方策による価値の減少量が $Q(s,a^\star) - \mathbb{E}_{a\sim\pi}[Q(s,a)] = \beta/2$ となり、任意の h に対して成り立つことを示せ。
3. 演習問題 1c（15.3 節）では、単一の意思決定において報酬を定数倍することが KL 係数を同じ因子で割ることと同じ効果をもつことを示した。任意のマルコフ決定過程において、すべての報酬に $k > 0$ を掛けることが、Equation 15.40 のもとで β を k で割ることと同じ最適方策を導くことを示せ。報酬の単位が変わったとき、固定された β を再調整しなければならないのはなぜか。
4. $\beta \to 0$ とする。σ はどうなるか。$\beta = 0$ かつ $\sigma_\theta = 0$ のとき、損失 Equation 15.43 は $-Q_w\big(s, c\tanh\mu_\theta(s)\big)$、すなわち DDPG (Lillicrap et al. 2016) の決定論的アクターの損失になる。このようなエージェントの探索はどこから得なければならないか。また、この極限は問題 2b（14.9 節）とどう関係するか。
5. バッファに入れる前にすべての報酬を 10 倍し、報告するエピソード収益はスケーリングしないままとしたうえで、固定の $\beta = 0.2$ で 1 シード、学習される温度で 1 シードを訓練せよ。小問 c を用いて、1 つ目の実行のエントロピーと 2 つ目の実行の最終的な β が 15.5.4.2 節の実行と比べてどうなるかを予測し、実際に確認せよ。Haarnoja, Zhou, Abbeel, et al. (2018) はその図 3(b) において、固定温度を用いた SAC の報酬スケールに対する感度を示している。

3. [code] **スクワッシュされた方策のエントロピー**。$u \sim \mathcal{N}(\mu, \sigma^2)$、$a = c\tanh u$（$c = 2$）とし、`SquashedGaussianPolicy` と同じ設定を考える。

1. Equation 15.44 を用いて次式を示せ。

$$H(\pi) = \tfrac{1}{2}\log(2\pi e\sigma^2) + \mathbb{E}_u\big[\log(1 - \tanh^2 u)\big] + \log c. \tag{15.49}$$

$\mu = 0$ において、σ の増大とともに大きくなるのはどの項であり、小さくなるのはどの項か。

2. H を $\mu = 0$、$\sigma \in \{0.1, 0.3, 1, 3\}$ において、u に関する期待値として（`log_prob_u` を用いたモンテカルロ法、または u に関する数値積分によって）評価せよ。どの σ の

近くで H はピークをとるか。このピークを $\log 2c$（区間 $[-c, c]$ 上の一様分布のエントロピーであり、この区間上のどの密度もこれを超えない）と比較せよ。

3. Pendulum 向けに新しく初期化した `SquashedGaussianPolicy` を構築せよ。そのエントロピーを `sample` を用いたモンテカルロ法により、一様ランダムなトルクのもとで訪問した 1,000 個の状態（`train_sac` のウォームアップと同様）にわたって平均して推定せよ。その推定値は、小問 b における $\mu = 0$、$\sigma = 1$ のときの値にどれほど近くなるか。これらの状態に対する初期化直後のネットワークのどの出力が、その差を説明するか。もしアクター損失がその項 $\beta \log \pi_\theta$ のみを残していたとしたら、15.5.4.2 節のエントロピーの推移はどの値に近づくか。

Berkeley CS 285 の Assignment 3 第 3.3 節を改変。

4. **重点サンプリング比なしのオフポリシー学習。** `train_sac` のバッファは、一様ランダムなウォームアップの行動 1,000 個を含め、実行中のすべての遷移を保持する一方、Equation 15.41 と Equation 15.43 は現在の方策から新たな行動をサンプリングする。

   1. `sac_step` の変種として、$\tilde{a}'$ をデータ収集方策が s' で次にとった行動に置き換えるとする。このときクリティックはどの方策の価値を推定し、そのような行動における項 $-\beta \log \pi_\theta(a' \mid s')$ は何を測っているか。どの比を用いればターゲットを補正できるか。また、このバッファからはその比を計算できないのはなぜか。
   2. どちらの損失も、現在の方策からではなくバッファからサンプリングされた状態にわたって平均をとる。15.5.1.4 節の命題はすべての状態で方策を改善する。状態ごとの厳密な最大化においては状態の重み付けが問題にならないのはなぜか。一方、表現力に限りのあるアクターネットワークにおいてはそれが問題になるのはなぜか。

5. [code] **ヤコビ行列の復元力。●** アクター損失 Equation 15.43 において、項 $\beta \log \pi_\theta$ は行動の各座標について $-\beta \log(1 - \tanh^2 u)$ を含む。ここで $u = \mu_\theta(s) + \sigma_\theta(s) z$ である。

   1. Equation 15.45 を用いて、この項が損失の勾配に $2\beta \tanh(u)\, \partial u / \partial \theta$ だけ寄与することを示せ。これは u をどの方向へ動かし、その押し戻す力は最大でどれほど強くなるか。
   2. float32 の自動微分を用いて、u に関する導関数を安定な形および保護付きの式 `log(1 - tanh(u)**2 + 1e-6)` について $u \in \{3, 5, 7, 10, 20\}$ で評価し、両者を $-2 \tanh u$ と比較せよ。どの $|u|$ を超えると、保護付きの式で訓練されたアクターは小問 a の押し戻す力を失うか。
   3. `log_prob_u` から `logdet` を削除せよ。残ったエントロピー項が損失を変化させる割合は $-\beta / \sigma$（σ の 1 単位あたり）であり、μ に依存しないことを示せ。損失のクリティック項が σ の増大を抑えられなくなるのは、いったん u が飽和したあとではなぜか。σ がクランプ上限の e^2 に達したとき、エントロピーの推移はどの値を報告するか。またその値は問題 3b の上界と比べてどうなるか。
   4. この項を削除した状態で 1 シードを再訓練せよ。収益曲線、エントロピーの推移、および変更後の密度に適用した `squashed_logp` の数値積分による検証結果を記録せよ。3 つの測定結果のうち、どれがこの誤りを暴くか。

6. [code] **更新比。** 15.5.4.1 節の実行では、1 環境ステップあたり 1 回および 10 回の更新を行った。

   1. ツインクリティックと単一クリティックのそれぞれについて、`utd=20` で 3 つのシードを 3,000 環境ステップ訓練せよ。水準への到達は 1 ステップあたり 10 回の更新より早くなるか。`warmup=200` でも同様に繰り返せ。
   2. 小問 a のエージェントを `calibration` で測定せよ。単一クリティックの乖離は、1 ステップあたり 10 回から 20 回の更新に増やすと大きくなるか。ツインクリティックの最小値は実現収益以下に留まるか。

3. 1 ステップあたり 1 回、10 回、20 回の更新のそれぞれについて、各実行が -200 を超えるまでに要する勾配ステップ数を数えよ。高い更新比は環境ステップと引き換えに何を差し出しており、その交換がどこまで通用するかを制限するものは何か。

7. [extended] **離散ソフトアクタークリティック**。●有限の行動集合では、方策ネットワークはソフトマックスを通じて確率 $\pi_\theta(\cdot \mid s)$ を出力し、各クリティックは 15.4 節の Q ネットワークと同様に行動ごとに 1 つの価値を出力する (Christodoulou 2019)。

   1. $\tilde{a}'$ と $\tilde{a}$ に関する期待値を行動全体にわたる和で厳密に計算する形で、クリティックのターゲットとアクター損失を書き下せ。`SquashedGaussianPolicy` のどの部分が不要になるか。また、アクター勾配に行動に関して微分可能なクリティックが不要になるのはなぜか。
   2. 15.5.3.2 節の自動温度調整における目標値 $\bar{H} = -\dim \mathcal{A}$ が離散方策に対して意味をなさないのはなぜか。また、$|\mathcal{A}|$ 個の行動に対する目標値はどの区間になければならないか。
   3. ツインクリティックと Polyak ターゲットを用いて CartPole 上にこの手法を実装し、同じステップ予算とシードで環境ステップに対する収益を 15.4 節の `train_dqn` と比較せよ。

15.6 学習されたモデルと計画

ソフトアクタークリティックは数千回の遷移を経て振り子を振り上げたが、どの遷移も同じように扱っていた。すなわち、報酬に次の状態の割引価値を足したものを、クリティックの 1 回の回帰におけるターゲットとして使うだけである。しかし、1 回の遷移にはそれ以上の情報が記録されている。ある特定のトルクのもとで振り子がどう動いたかが示されており、その運動の背後にある物理法則は何度訪れても同じである。ブランコで数回背中を押してもらった子どもなら、少し後にブランコがどこへ行くか、もっと高く漕ぐにはどれくらい強く押せばよいかがわかるだろう。では、エージェントはクリティックも方策も持たずに、数百ステップのデータから振り子の物理法則を学習し、計画を立てて振り子を振り上げられるだろうか。

14.3 節では、湖の環境でこれを実行した。そこでは観測した遷移の回数を数えて推定モデルを作り、その推定値を正確なものとみなして価値反復法（value iteration）を適用した。14.6 節では、Pendulum の連続状態上で、既知の物理法則を iLQR で微分し、移動ホライズン（receding-horizon）方式のモデル予測制御（model predictive control、MPC）によって毎ステップ再計画した。物理法則が未知の場合、Pendulum の連続状態は数え上げられないため、数え上げの代わりに回帰を用いる。すなわち、現在の状態とトルクから次の状態と報酬を予測するようネットワークを学習させる。そして価値反復法や iLQR の代わりに移動ホライズン探索を用いる。エージェントは学習されたモデルの中で多数のトルク系列を試し、最良の系列の最初のトルクを実行する。Sutton の Dyna アーキテクチャは 1990 年に学習されたモデルと計画を組み合わせ (Sutton 1990)、この方法で探索されるニューラルネットワークモデルはわずか数分の経験からシミュレーション上のロボットを制御している (Nagabandi et al. 2018)。

モデルがあれば、エージェントは経験の代わりに計算を使えるようになる。1 回の実ステップの後に、何千回もの想像上のステップを続けられるからである。また、方策が提案する行動から探索を始め、その結果を方策へと教え戻すこともできる。AlphaZero と MuZero はこの方法で人間を超える強さのボードゲームプレイを学習し (Silver et al. 2018; Schrittwieser et al. 2020)、学習された方策に導かれる探索は、言語モデルがテスト時に追加の計算量を投入する一手法でもある。本節では、1 回のランダムなエピソードから Pendulum のモデルを当てはめ、計画に沿ってその誤差がどう増大するかを測定し、そのモデルで計画を立て、さらに探索をより良い方策へと変換する。本節全体を通じて常に問われるのは、データからどれだけ離れたところまでモデルを信頼できるかという点である。

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

15.6.1 データからのダイナミクスモデル

15.6.1.1 数え上げに代わる回帰

Pendulum（`Pendulum-v1`）は、14.6 節で物理法則が既知の問題として導入され、14.9 節と 15.5 節でモデルフリーに扱った連続制御ベンチマークである。モーター付きの支点から振り子が吊り下げられており、状態は $s = (\cos\vartheta, \sin\vartheta, \dot{\vartheta})$（$\vartheta$ は直立位置からの角度）、行動はトルク $a \in [-2, 2]$ である。各ステップでは報酬 $-(\vartheta^2 + 0.1\,\dot{\vartheta}^2 + 0.001\,a^2)$ が与えられ（ただし ϑ は $[-\pi, \pi]$ に折り返される）、1 エピソードは 200 ステップ続く。真下で静止したままの振り子は 1 ステップあたり $\pi^2 \approx 9.9$、1 エピソードあたり約 $2{,}000$ を失う。ランダムなトルクでは約 900〜$1{,}700$ を失う。一方、振り子を振り上げてバランスを保つコントローラは、-200 という 14.6 節および 14.9 節の基準線を上回り、開始状態に応じて失点を 0 から数百の間に抑える。

環境の**モデル**とは、環境がどのように振る舞うかを予測するものである。状態と行動から予測される次の状態と報酬への写像を表すネットワークを $\hat{f}_\phi$（パラメータは ϕ）と書くことにすると、次式となる。

$$(\hat{s}', \hat{r}) = \hat{f}_\phi(s, a). \tag{15.50}$$

湖の環境ではモデルは頻度の表であり、各要素はその状態行動対（state-action pair）への訪問回数のみから推定されていた。これに対し、ネットワークは状態間でパラメータを共有するため、ある角度での遷移から近傍の角度についても学習できる。保存された遷移 (s_i, a_i, r_i, s'_i) に対して、最小二乗法でこのネットワークを次のように当てはめる。

$$\min_{\phi} \sum_{i} \left\| \hat{f}_\phi(s_i, a_i) - (s'_i, r_i) \right\|^2. \tag{15.51}$$

この当てはめは、保存されたすべての遷移について、記録された次の状態と報酬を再現するようモデルに求めるものである。もしネットワークの代わりに表を使えば、各状態行動対における最小点はそこで観測された結果の平均、すなわち 14.3 節の数え上げモデルをその平均へと縮約したものになる。Pendulum は決定論的であるため、結果の平均がそのまま結果そのものとなる。確率的な環境であれば、湖の環境で訪問回数が与えていたように、次の状態の分布全体のモデルが必要になるだろう。

2 つの工夫によってこの回帰は容易になる。第 1 に、ネットワークは状態の変化量 $s' - s$ を、s' そのものではなく予測対象とする。0.05 秒の 1 ステップでは振り子はわずかしか動かず、変化量こそが学習に値する部分だからである。第 2 に、入力とターゲットのすべての座標を、保存データ上で平均 0・分散 1 に標準化する。余弦と正弦は $[-1, 1]$、角速度は $[-8, 8]$、報酬はおおよそ $[-16, 0]$ の範囲にあり、座標ごとにスケールが異なるためである。標準化しなければ、最小二乗法はもっぱら報酬ばかりに当てはめようとしてしまう。

まず本節全体で用いる設定を定義する。トルクの上限 $c = 2$、計画の深さ $n = 25$、探索で試す候補計画の数、モデルの 1 回の当てはめあたりの Adam ステップ数、シード数、そしてテストの初期状態を固定するシードである。

```
c, horizon, num_cand = 2.0, 25, 500   # torque bound, depth n, candidates
fit_steps, num_seeds, test_seed = 1_000, 5, 1_000
```

データはエピソード単位で収集する。run_episode は、Pendulum の複数のコピーを同期して進める gymnasium のコンテナである**ベクトル環境**（vector environment）内で行動するため、1 回の呼び出しで 10 個の初期状態から同時にコントローラを実行できる。コントローラは観測のバッチから行動のバッチへの任意の関数であり、random_controller は一様ランダムなトルクを抽出するコントローラを返す。score は環境シード 1000〜1009 に対応する 10 個の固定された初期状態からコントローラを 1 エピソード実行し、平均収益を返す。これが本節のすべてのコントローラに対するテストとなる。遷移は d2l.ReplayBufferC、すなわち 15.5 節のリプレイバッファで行動列を浮動小数点にしたものに格納する。ここでは単に配列の保管場所として使うだけであり、そこからバッチをサンプリングすることはない。

```
def run_episode(envs, controller, buffer=None, seed=None, num_steps=200):
    """One episode in every environment of a vector env, acting by
    controller(obs) -> actions. Returns the returns; records transitions."""
    obs, total = envs.reset(seed=seed)[0], 0.0
    for _ in range(num_steps):
        act = controller(obs)
        next_obs, rew, term, _, _ = envs.step(act)
        for i in range(envs.num_envs if buffer is not None else 0):
            buffer.add(obs[i], act[i], rew[i], next_obs[i], float(term[i]))
        obs, total = next_obs, total + rew
    return total

def random_controller(rng, scale=c):
    """Uniformly random torques: the data of the first episode."""
    return lambda obs: rng.uniform(-scale, scale, (len(obs), 1)).astype(
        np.float32)

def score(controller, num_starts=10, seed=test_seed):
    """Mean return of a controller over ten fixed Pendulum starts."""
    envs = gym.make_vec('Pendulum-v1', num_envs=num_starts,
                        vectorization_mode='sync')
    return run_episode(envs, controller, seed=seed).mean()
```

model_data は、保存された遷移を回帰の入力 (s, a) とターゲット $(s' - s, r)$ に変換し、あわせてそれらの平均と標準偏差を返す。

```
def model_data(buffer):
    """Inputs (s, a) and targets (s' - s, r) of the stored transitions,
    with their means and standard deviations."""
    n = buffer.size
    x = np.concatenate([buffer.obs[:n], buffer.act[:n]], -1)
    y = np.concatenate([buffer.next_obs[:n] - buffer.obs[:n],
                        buffer.rew[:n, None]], -1)
    return x, y, (x.mean(0), x.std(0) + 1e-6, y.mean(0), y.std(0) + 1e-6)
```

DynamicsModel は、64 ユニットの隠れ層を 2 つ持ち、滑らかな SiLU 活性化関数を用いた多層パーセプトロン（multilayer perceptron、MLP）である（4.1 節）。このモデルはパラメータと並んで標準化統計量を自身の状態の一部として保持しており、順伝播では入力を標準化し、出力の標

準化を元に戻したうえで、予測された変化量を状態に加算する。`fit_model` は保存された全データから統計量を再計算し、256 個の遷移からなるミニバッチ上で Adam ステップを実行する。そのオプティマイザは `make_model` 内でモデルと同時に 1 度だけ生成されるため、以降の当てはめは最初からやり直すのではなく前回のパラメータから継続される。

```
class DynamicsModel(nn.Module):
    """(s', r) from (s, a): an MLP that maps standardized inputs to the
    standardized change of state and reward."""
    def __init__(self, obs_dim=3, act_dim=1, hidden=64):
        super().__init__()
        d_in, d_out = obs_dim + act_dim, obs_dim + 1
        self.net = nn.Sequential(nn.Linear(d_in, hidden), nn.SiLU(),
                                 nn.Linear(hidden, hidden), nn.SiLU(),
                                 nn.Linear(hidden, d_out))
        self.register_buffer('mu_x', torch.zeros(d_in))   # standardization
        self.register_buffer('sd_x', torch.ones(d_in))
        self.register_buffer('mu_y', torch.zeros(d_out))
        self.register_buffer('sd_y', torch.ones(d_out))

    def forward(self, obs, act):
        x = (torch.cat([obs, act], -1) - self.mu_x) / self.sd_x
        y = self.net(x) * self.sd_y + self.mu_y   # (s' - s, r) in data units
        return obs + y[..., :-1], y[..., -1]

def make_model(seed, lr=1e-3):
    torch.manual_seed(seed)
    model = DynamicsModel()
    return model, torch.optim.Adam(model.parameters(), lr=lr)

def fit_model(model, opt, buffer, rng, num_steps=fit_steps, batch_size=256):
    """Least squares on all stored transitions, in standardized units."""
    x, y, stats = model_data(buffer)
    model.mu_x, model.sd_x, model.mu_y, model.sd_y = map(torch.as_tensor,
                                                         stats)
    mu_x, sd_x, mu_y, sd_y = stats
    x = torch.as_tensor((x - mu_x) / sd_x)
    y = torch.as_tensor((y - mu_y) / sd_y)
    for _ in range(num_steps):
        i = torch.as_tensor(rng.integers(len(x), size=batch_size))
        loss = ((model.net(x[i]) - y[i]) ** 2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
    return float(loss.detach())
```

シード 0 からランダムなトルクによる 1 エピソードを収集し、その 200 個の遷移にモデルを当てはめる。

```
env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
rng = np.random.default_rng(0)
buffer = d2l.ReplayBufferC(200, 3, 1)
print(f'return of the random episode: '
```

```
      f'{run_episode(env, random_controller(rng), buffer, seed=0)[0]:.0f}')
model, opt = make_model(0)
print(f'final minibatch loss (standardized units): '
      f'{fit_model(model, opt, buffer, rng):.4f}')
```

```
return of the random episode: -1072
final minibatch loss (standardized units): 0.0018
```

ランダムなエピソードは約 1,000 を失い、当てはめ終了時の損失はターゲットの分散（標準化単位では 1）のごく一部に収まっている。

15.6.1.2 1 ステップの手計算

訓練損失が小さいというだけでは、モデルが何を予測しているかはわからない。そこで、エピソード中の 1 つの遷移を元の問題の単位で見てみよう。`rollout_model` は状態と行動の系列をモデルに入力し、予測された各状態を次のステップの入力として順に送り込む。行動が 1 つだけなら、1 ステップ分の予測になる。

```
@torch.inference_mode()
def rollout_model(model, obs, actions):
    """Open-loop rollout in the model: obs (..., 3) and actions (..., n, 1)
    give the predicted states (..., n, 3) and rewards (..., n), numpy in
    and out."""
    s, actions = torch.as_tensor(obs), torch.as_tensor(actions)
    states, rewards = [], []
    for t in range(actions.shape[-2]):
        s, r = model(s, actions[..., t, :])
        states.append(s)
        rewards.append(r)
    return torch.stack(states, -2).numpy(), torch.stack(rewards, -1).numpy()
```

```
def angle(s):
    """Angle from upright in degrees, from states (cos, sin, velocity)."""
    return np.degrees(np.arctan2(s[..., 1], s[..., 0]))

i = 100
pred, r_hat = rollout_model(model, buffer.obs[i], buffer.act[i][None])
print('s       ', buffer.obs[i].round(4), f'a = {buffer.act[i, 0]:.4f}')
print("true s' ", buffer.next_obs[i].round(4), f'r = {buffer.rew[i]:.4f}')
print("model s'", pred[0].round(4), f'r = {r_hat[0]:.4f}')
print(f'angle error {abs(angle(pred[0]) - angle(buffer.next_obs[i])):.3f} '
      f'degrees')
```

```
s        [-0.9407 -0.3394  5.6896] a = -0.0800
true s'  [-0.8154 -0.5789  5.4231] r = -11.0511
model s' [-0.8109 -0.584   5.436 ] r = -11.2768
angle error 0.386 degrees
```

ステップ 100 では、振り子は $\vartheta = \operatorname{atan2}(-0.3394, -0.9407) = -2.795$ rad、すなわち直立位置から約 160 度のところにあり、トルクがほぼ 0 の状態で 5.69 rad/s で振れている。報酬は出力の通り $-(2.795^2 + 0.1 \cdot 5.690^2 + 0.001 \cdot 0.08^2) = -11.05$ である。1 ステップの 0.05 秒の間に角度は $\operatorname{atan2}(-0.5789, -0.8154) = -144.6$ 度へと、約 $5.6 \times 0.05 = 0.28$ rad すなわち

16 度動き、モデルはこの動きを 1 度未満の誤差で予測している。報酬のずれも 0.3 未満である。1 ステップの予測は簡単である。直前に提示された状態に対する小さな補正にすぎないからである。

15.6.1.3 誤差の累積

計画には複数ステップが必要である。トルクの系列にスコアを付けるには、モデルは 25 ステップ先まで予測しなければならず、2 ステップ目以降の入力はモデル自身の直前の予測になる。モデルは真の状態のみで訓練されており、わずかにずれた自身の予測状態では一度も訓練されていないため、あるステップでの誤差が次のステップの入力を変化させる。角度の小さな誤差が重力の及ぼすトルクを変え、それが角速度を変え、さらに角度を変化させるのである。この影響を**開ループ**（open-loop）ロールアウトで測定する。すなわち、エピソードに記録された行動をたどりながらモデルに状態を予測させ、k ステップ後の予測角度を記録された実際の角度と比較する。

`kstep_error` は、エピソード終了までに 25 ステップ以上残っているすべてのステップからロールアウトを開始し、$k = 1, 5, 10, 25$ ステップ後の角度の二乗平均平方根誤差を報告する。この際、$+179$ 度と -179 度の差が 2 度となるように角度を折り返して計算する。

```
def kstep_error(model, buffer, first=0, ks=(1, 5, 10, 25), T=200):
    """RMS angle error (degrees) of open-loop model rollouts along the
    recorded actions, from every step of the stored T-step episodes (rows
    first onward) that leaves max(ks) steps before the episode ends."""
    n = max(ks)
    ep = lambda x: x[first:buffer.size].reshape(-1, T, *x.shape[1:])
    obs, act, nxt = ep(buffer.obs), ep(buffer.act), ep(buffer.next_obs)
    windows = lambda x: np.stack([x[:, t:t + n]
                                  for t in range(T - n + 1)], 1)
    pred = rollout_model(model, obs[:, :T - n + 1], windows(act))[0]
    d = np.radians(angle(pred) - angle(windows(nxt)))
    d = np.arctan2(np.sin(d), np.cos(d))         # wrapped to (-pi, pi]
    return [np.degrees(np.sqrt(np.mean(d[..., k - 1] ** 2))) for k in ks]
```

5 つのシードのそれぞれについて、1 つのランダムなエピソードにモデルを当てはめ、同じシードから生成した 2 つ目のランダムなエピソードでテストする。

```
models = []
print('seed   k=1    k=5   k=10   k=25   (degrees, held-out random episode)')
for seed in range(num_seeds):
    rng = np.random.default_rng(seed)
    train, test = d2l.ReplayBufferC(200, 3, 1), d2l.ReplayBufferC(200, 3, 1)
    run_episode(env, random_controller(rng), train, seed=seed)
    run_episode(env, random_controller(rng), test)
    m, o = make_model(seed)
    fit_model(m, o, train, rng)
    models.append((m, test))
    errors = kstep_error(m, test)
    print(f'{seed:4d} ' + ' '.join(f'{e:6.2f}' for e in errors))
```

```
seed   k=1    k=5   k=10   k=25   (degrees, held-out random episode)
   0   0.44   1.91   2.52   1.51
   1   1.83   7.08   8.67  12.29
   2   0.53   2.37   3.65   5.51
   3   0.38   1.89   4.77  11.82
```

```
   4   0.04   0.12   0.13   0.33
```

どのシードでも、10 ステップ後の誤差は 1 ステップ後の誤差の数倍になっており、1 ステップ後の誤差はほとんどのシードで 1 度未満である。25 ステップ後の誤差はシードによって異なるが、これらのホールドアウトされたエピソードでは約 20 度未満に収まっている。ランダムなトルクのもとでは、振り子は訓練エピソードが訪れたのと同じ真下付近で揺れ続けるからである。この但し書きには、プランナが振り子を別の場所へ動かした後で改めて立ち返る。

計画の深さは、n（n ステップ収益（15.1 節）のステップ数）と同様の調整ダイヤルである。計画を深くすればトルクのもたらすより先の結果まで見通せる。振り上げには数十ステップを要するため、振り子が持ち上がる前に終わる計画では、うまく勢いをつける系列とそうでない系列を区別できない。しかし、ステップを先へ延ばすたびにモデルの誤差が予測に加わっていく。本節では $n = 25$ ステップ（1.25 秒）を用い、演習問題 1 でこのダイヤルを動かして調べる。

15.6.2 モデル内での計画

15.6.2.1 ランダムシューティング

モデルの中であれば、トルクの系列を試すコストは数回の行列積だけであり、実ステップは一切かからない。最も単純な探索はこれをそのまま利用する。振り子が真下で静止しているとする。25 個のトルクからなるランダムな系列を 5 本引き、先ほど当てはめたモデル内でそれぞれをロールアウトし、予測される報酬を足し合わせる。

```
rng = np.random.default_rng(0)
s = np.array([[-1.0, 0.0, 0.0]], np.float32)      # hanging down, at rest
A = rng.uniform(-c, c, (1, 5, horizon, 1)).astype(np.float32)
_, r_hat = rollout_model(model, np.repeat(s[:, None], 5, axis=1), A)
real = gym.make('Pendulum-v1').unwrapped          # the same sequences, for real
for a, ret in zip(A[0], r_hat[0].sum(-1)):
    real.reset()
    real.state = np.array([np.pi, 0.0])           # angle, velocity
    true = sum(real.step(a_t)[1] for a_t in a)
    print(f'first actions {a[:3, 0].round(2)}  predicted {ret:7.1f}  '
          f'true {true:7.1f}')
```

```
first actions [ 0.55 -0.92 -1.84]  predicted  -174.2  true  -237.6
first actions [-0.47  1.99  1.92]  predicted  -173.8  true  -232.7
first actions [ 1.15 -1.04  1.51]  predicted  -174.3  true  -229.4
first actions [-0.3   0.48  1.98]  predicted  -170.7  true  -223.2
first actions [-0.08 -1.07  1.21]  predicted  -174.5  true  -242.4
```

25 ステップの間静止したままでいると約 $25 \times 9.9 \approx 247$ を失うが、5 つの予測はいずれもそれより大幅に高く、互いに数ポイント以内の差に収まっている。最後の列は、同じ状態から実環境で同じ 5 本の系列を再生した結果である。失点は 223〜242 の間にとどまっている。振り子を真下から少し揺り動かすことで静止状態に比べて節約できるのは 5〜24 ポイントであり、モデルが予測するような 60〜80 ポイントではない。モデルはどの系列に対しても 40〜70 ポイント楽観的すぎる。原因として考えられるのはカバレッジである。ランダムなエピソードでは振り子が真下を勢いよく通過していたため、モデルは真下でほぼ静止している状態を一度も観測しておらず、その状態に対する報酬を高く見積もりすぎているのである。それでもなお、予測収益が最も高い系列は実環境でも 5 本の中で最良であり、候補の中から選ぶのに必要なのはそれらの値そのものではなくランキングだけである。5 本の中からならその系列を選び、最初のトルクを適用することになる。モデルをどこまで信頼できるかという問いは、この最小の例にすでに現れている。ここでは予測の値は誤っているが、選択は正しい。

これが**ランダムシューティング**である。多数の候補系列 $(a_0^{(j)}, \dots, a_{n-1}^{(j)})$ を引き、それぞれの収益を次式で予測する。

$$\hat{G}^{(j)} = \sum_{t=0}^{n-1} \hat{r}_t^{(j)},$$

ここで $\hat{r}_t^{(j)}$ はステップ t における候補 j に対するモデルの予測報酬であり、その中から最良の系列を残す。25 ステップで $\gamma = 0.99$ としても割引によって重みが変わるのは高々約 1/5 にすぎないため、この和には割引を入れない。実行するのは最良の系列の最初のトルクだけである。次のステップでは、エージェントは実際の状態を観測し、そこから再び計画を立てる。これは 14.6 節の移動ホライズン方式の**モデル予測制御**（MPC）と同じ枠組みである。そこでは振り子の既知の運動方程式を用いて計画を立てたが、ここでは学習されたモデル $\hat{f}_\phi$ の中で探索を実行する。毎ステップ再計画することで、実行される軌跡に沿ってモデルの誤差が蓄積するのを防げる。誤差が累積する前に、次の実際の観測が予測を置き換えるからである。ただし、計画の後半ステップにおける誤差は、どの最初のトルクが選ばれるかに依然として影響する。

`plan` は B 個の状態のバッチに対してこれを一括で実行する。各状態に対する与えられた計画にガウスノイズを加えて 500 個の候補へと摂動させ（与えられた計画そのものも候補の 1 つとして残す）、トルクを $[-c, c]$ にクリップしたうえで、全候補を `rollout_model` でロールアウトし、各状態の最良の系列とその予測収益を返す。元の計画自身が候補に含まれているため、返される計画の予測収益が開始時の計画を下回ることは決してない。1 回の行動につき、モデルの評価が $500 \times 25 = 12{,}500$ 回必要になる。

```
def plan(model, obs, init, rng, noise=c, num_cand=num_cand, scale=c):
    """Random shooting around a plan for each of B states: obs (B, 3), init
    (B, n, 1). Each plan is perturbed into num_cand candidates (the plan
    itself is one), all are rolled out in the model, and the candidate with
    the highest predicted return is returned with that return."""
    B = len(obs)
    A = init[:, None] + noise * rng.standard_normal(
        (B, num_cand) + init.shape[1:], dtype=np.float32)
    A[:, 0] = init
    A = np.clip(A, -scale, scale)
    s = np.repeat(obs[:, None], num_cand, axis=1)          # (B, num_cand, 3)
    ret = rollout_model(model, s, A)[1].sum(-1)            # (B, num_cand)
    best = ret.argmax(1)
    return A[np.arange(B), best], ret[np.arange(B), best]
```

15.6.2.2 ウォームスタート

探索はどの計画から始めるべきだろうか。事前知識がまったくなければ、毎ステップ新たにランダムな系列を引くことになる。しかしそれでは、1 ステップ前に見つけた（おそらく良質な）計画を捨ててしまい、毎ステップ 500 本のランダムな系列の中から振り上げの計画を再び見つけ出さなければならなくなる。結果として、連続するトルクが互いに無関係な計画から選ばれることになる。**ウォームスタート**は前回の計画を再利用する。その最初のトルクは直前に実行済みであるため、残りの 24 個を 1 ステップ前にシフトし、末尾に 0 のトルクを追加して、このシフトした計画を探索で摂動させる。こうすれば、良い計画を一度見つけた後はステップごとにそれを洗練していける。Figure 15.15 にこの探索の 1 ステップを示す。

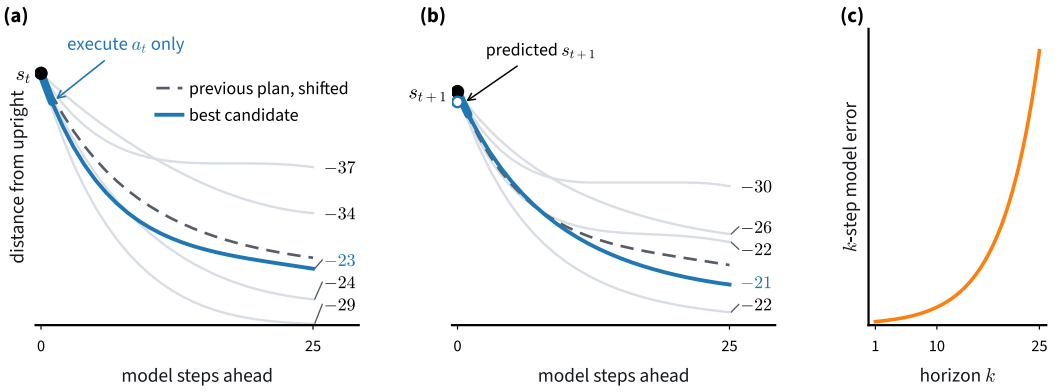


Figure 15.15: ウォームスタート付きランダムシューティングによる学習されたモデル内での計画。(a) 状態 s_t において、1 ステップ分シフトした前回の計画（破線）の周囲から引いた候補行動系列を $\hat{f}_\phi$ で 25 ステップロールアウトし、予測報酬の和（右側の数値）によってスコア付けする。最良の系列（青）の最初の行動のみを実行する。(b) 実際の次の状態は予測された状態（白丸）からわずかにずれ、プランナはそこから (a) の最良の計画を 1 ステップ分シフトしたものの周囲で再び計画を始める。(c) モデルの開ループ誤差はホライズンとともに増大するため、これが毎ステップ再計画する理由の 1 つとなる。縦軸とスコアは Pendulum ではなく模式的な 1 次元モデルのものである。スコアは速度やトルクのコストを含む全ステップの報酬を足し合わせるため、直立位置に最も近いところで終わる系列のスコアが最高になるとは限らない。

`mpc` はモデルからこのようなコントローラを構築する。計画は環境ごとに 1 つずつコントローラのクロージャ内に保持され、最初の呼び出し時に生成される。摂動の標準偏差はトルクの上限 $c = 2$ に等しい。別の 10 個の初期状態セットでスコアを評価する 5 シードの独立した実験で、毎ステップ 500 本の新たな一様ランダム系列を引くプランナと比較した。600 実ステップ後、ウォームスタート付きプランナのスコアはすべての実行で約 −115〜−155 の間に入ったのに対し、毎回新たに引くプランナでは約 −165〜−555 であった。1,000 実ステップ後でも、毎回新たに引くプランナはすべてのシードで依然として 40〜110 ポイント劣っていた。

```
def mpc(model, rng, horizon=horizon, noise=c):
    """Model-predictive control: at every step plan in the model, starting
    from the previous plan shifted by one step, and execute its first
    action."""
    plans = None
    def controller(obs):
        nonlocal plans
        if plans is None:
            plans = np.zeros((len(obs), horizon, 1), np.float32)
        shifted = np.concatenate([plans[:, 1:], np.zeros_like(plans[:, :1])],
                                 1)
        plans = plan(model, obs, shifted, rng, noise)[0]
        return plans[:, 0]
    return controller
```

15.6.2.3 数エピソードからの振り上げ

プランナには何回の実ステップが必要だろうか。mpc_run は 1 回のランダムなエピソードから始める。続いて、それまでの全データでモデルを再当てはめし、10 個のテスト初期状態でプランナのスコアを評価し、プランナで 1 エピソードを実行してデータを追加収集するという手順を 3 回繰り返した後、最後の再当てはめをしてスコアを評価する。したがって、得られるスコアは 200、400、600、800 実ステップから構築されたプランナを測定したものになる。

```
def mpc_run(seed, num_rounds=3):
    """One random episode, then rounds of: refit the model on all data,
    score the planner on the ten test starts, collect one MPC episode."""
    env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
    rng, (model, opt) = np.random.default_rng(seed), make_model(seed)
    buffer = d2l.ReplayBufferC(200 * (num_rounds + 1), 3, 1)
    run_episode(env, random_controller(rng), buffer, seed=seed)
    scores = []
    for r in range(num_rounds + 1):
        fit_model(model, opt, buffer, rng)
        scores.append(score(mpc(model, rng)))
        if r < num_rounds:
            run_episode(env, mpc(model, rng), buffer)
    return scores, buffer
```

5 つのシードについて、収集した 4 エピソードの収益と 4 回のテストスコアの出力は次の通りである。

```
runs = [mpc_run(seed) for seed in range(num_seeds)]
print('seed  episode returns: random, MPC x 3 |'
      '  test score at 200/400/600/800 steps')
for seed, (scores, buf) in enumerate(runs):
    rets = buf.rew[:buf.size].reshape(-1, 200).sum(1)
    print(f'{seed:4d} ' + ''.join(f'{r:7.0f}' for r in rets) + '  | '
          + ''.join(f'{s:7.0f}' for s in scores))
```

```
seed  episode returns: random, MPC x 3 |  test score at 200/400/600/800 steps
   0   -1072   -242   -118   -125  |    -181   -168   -168   -171
   1    -876   -264   -121   -123  |    -266   -201   -204   -180
   2    -966   -122   -125   -116  |    -337   -190   -168   -169
   3   -1501  -1287   -246     -2  |   -1464   -265   -191   -195
   4   -1703  -1576  -1323   -120  |   -1567  -1437   -171   -187
```

どの程度であれば良いスコアといえるだろうか。10 個のテスト初期状態のうち、振り上げてバランスを保つコントローラは、真上に最も近い初期状態からはほとんど失点せず、真下付近の 5 つの初期状態からはそれぞれ約 250〜320、平均すると約 170 を失う。15.5 節のソフトアクタークリティックについて、各エピソード後にその貪欲アクターのスコアを同じ方法で評価すると、−170 付近に落ち着き、1 ステップあたり 1 回の更新では約 5,000〜7,500 ステップ後に初めて −200 に到達する。

3 エピソードにあたる 600 実ステップ後には、プランナのスコアはすべての実行で約 −165〜−215 とその水準に近づき、800 ステップ後には約 −170〜−195 となる。最初に −200 に到達するのは 200〜800 実ステップ後であり、大半の実行では 600 ステップまでに到達する。5 つのシードのうち 3 つでは、最初のランダムエピソードのモデルだけでも約 −175〜−340 のスコアを出すプランナが得られており、ランダム方策の −1,000 以下とは大きく異なる。残る 2 つである

シード 3 とシード 4 では、最初のプランナはスコアが −1,500 付近となって失敗し、最初の 1〜2 エピソードも失敗するが、600 実ステップ後には同じプランナが成功する。

15.6.2.4 カバレッジ

立ち上がりの遅い 2 つのシードは、ランダムエピソードでの失点が約 1,500 と 1,700 で最も大きかった 2 つであり、ランダムなトルクでは振り子が一度も水平より上まで持ち上がらなかったケースである。先ほどの誤差の表と同様に、各シードについてその 1 回のランダムエピソードに当てはめたモデルを取り出し、2 つ目のランダムエピソードではなくプランナ自身のエピソードに沿ってテストしてみよう。

```
print('seed   k=1    k=5   k=10   k=25   (degrees, the planner\'s episodes)')
for seed, ((m, _), (_, buf)) in enumerate(zip(models, runs)):
    errors = kstep_error(m, buf, first=200)      # skip the random episode
    print(f'{seed:4d} ' + ' '.join(f'{e:6.1f}' for e in errors))
```

```
seed   k=1    k=5   k=10   k=25   (degrees, the planner's episodes)
   0    0.3    1.4    3.3   42.4
   1    0.4    1.5    2.6   40.2
   2    0.2    0.9    1.7   28.3
   3    6.8   41.3   61.5  138.8
   4   14.8   71.9   89.8  128.3
```

シード 0〜2 では、プランナのエピソードに沿った誤差はランダムデータ上と同様に小さく始まり、1 ステップ後では 1 度未満、25 ステップ後には数十度へと増大する。これらのエピソードは大半の時間を直立位置の近くで過ごすが、そこでは振り子が不安定であり、予測角度のわずかな誤差がどちら側へ倒れるかを左右する。一方、シード 3 とシード 4 では、わずか 1 ステップ後で数度、5 ステップ後には数十度の誤差が生じている。モデルはデータが訪れた領域では正確だが、それ以外の領域では誤っているのである。湖の環境では、一度も試していない状態行動対は数え上げモデルには無価値に見えたが、ここではネットワークが一度も見たことのない上半円へと外挿し、その外挿が外れている。これを修復するのがプランナの失敗した試行である。振り上げようとする試行は、たとえ失敗してもランダムなトルクより高くまで振り子を運び、そこで何が起きるかを記録する。次の再当てはめではモデルがそれらの状態のデータを持つようになり、その上に構築された計画が機能する。モデルが報酬を見込む場所へとプランナが向かい、モデルが誤っていた場所はどこでも修正されるため、プランナは明示的な探索規則なしに探索を進めるのである。

15.6.2.5 実ステップのコスト

プランナは 200〜800 実ステップ後に初めて −200 に到達する。10 個のテスト初期状態にわたる平均という同じスコアで見ると、ソフトアクタークリティックが初めて −200 に到達するのは、環境 1 ステップあたり勾配ステップ 1 回では約 5,000〜7,500 実ステップ後、環境 1 ステップあたり勾配ステップ 10 回では約 1,600〜4,000 実ステップ後である。これらは学習される温度を用いた 15.5 節のコードを各設定 5 シードで実行した結果であり、ステップ数には最初の 1,000 ステップのランダムなウォームアップも含まれている。環境ステップ数という軸で見れば、プランナに必要なステップ数は前者の約 1/10、後者の数分の 1 で済む。

もう 1 つの軸は計算量である。プランナの行動は毎回が探索であり、モデルを 12,500 回評価する（この小さなネットワークなら CPU 1 コアで数 ms である）。さらにエピソードごとに 1,000 回の Adam ステップによる再当てはめが加わる。ソフトアクタークリティックは 1 実ステップあたり 3 つの小さなネットワークの勾配ステップを 1 回または 10 回実行し、行動選択にはアクターの順伝播が 1 回必要なだけである。実機のロボットでは、制御周期（ここでは 0.05 秒）以内に探索を

終えなければならないという制約もある。モデルは学習のコストを無料にするわけではない。ロボットにとって希少な「経験」から、ロボットのコントローラにとって希少な「計算量」へとコストを移し替えるのである。

15.6.3 方策改善としての探索

15.6.3.1 アクターを初期値とする探索

これまでのプランナは方策を持たない。行動のたびに完全な探索が必要であり、見つけた結果はモデルと直前のステップからシフトした計画を介する以外には何も記憶されない。方策ネットワークがあれば 1 回の順伝播で行動でき、状態間で汎化（generalization）できるが、それには訓練が必要であり、15.5 節ではクリティックを通じて方策ネットワークを訓練するのに数千ステップを要した。では、代わりに探索を使って方策を訓練できないだろうか。

探索を方策自身の計画から始めることにしよう。アクターを、状態をトルクへ写すネットワーク $\bar{a}_\theta(s)$ とする。モデル内において、状態 s からのアクターの計画とは、アクター自身が予測される軌跡に沿って選ぶトルクの系列である。すなわち、$a_0 = \bar{a}_\theta(s)$、$\hat{s}_1$（$\hat{f}_\phi(s, a_0)$ による予測）、$a_1 = \bar{a}_\theta(\hat{s}_1)$ のように n ステップ分続ける。`actor_plan` は状態のバッチに対してこの計画を計算する。`improve` はシフトした前回の計画の代わりにこの計画を `plan` に渡し、同じ 500 個の候補と同じ摂動を用いて探索し、最良の候補の最初のトルクを返す。

```
def actor_plan(model, actor, obs, horizon=horizon):
    """The actor's own deterministic plan for each state, imagined in the
    model: (B, n, 1)."""
    acts = []
    for _ in range(horizon):
        acts.append(actor.act_greedy(obs))
        obs = rollout_model(model, obs, acts[-1][:, None])[0][:, 0]
    return np.stack(acts, 1)

def improve(model, actor, obs, rng, noise=c):
    """Search seeded by the actor: the best first action for each state."""
    return plan(model, obs, actor_plan(model, actor, obs), rng, noise)[0][:, 0]
```

アクター自身の計画も候補の 1 つに含まれているため、探索が返す行動の計画は、少なくともアクターの計画と同じかそれ以上の予測収益を持つ。モデルのもとで n ステップにわたって評価する限り、方策を初期値とする探索がその方策を下回ることは決してない。これは 14.2.3.2 節における方策反復法（policy iteration）の貪欲ステップにあたる。そこでは各状態において、現在の方策自身の価値を基準として現在の方策以上の行動を選んでいたが、ここで基準となるのは次の 25 ステップにわたるモデルの予測である。この保証が成り立つのはモデルの中だけであり、かつホライズンの範囲内に限られる。誤ったモデルはより悪い行動にお墨付きを与えかねず、25 ステップで終わる計画はそれ以降に起きることを評価できない。

15.6.3.2 アクターへの蒸留

探索がアクターを改善するのであれば、探索を模倣するようアクターを訓練できる。これは探索をエキスパートとみなした 14.5 節の行動クローニング（behavior cloning）である。すなわち、`improve` で状態にラベルを付け、トルクは連続値なので二乗誤差を用いてアクターのトルクをそのラベルへと回帰させる。アクターには `d2l.SquashedGaussianPolicy`（15.5 節）を用いる。これはガウス分布の平均を $c \tanh$ でトルクの範囲に押し込める 2 層のネットワークである。ここでは押し込められた平均 $\bar{a}_\theta(s) = c \tanh \mu_\theta(s)$（同節と同じく μ_θ はガウス平均であり、`act_greedy` が

返す値）のみを訓練する。`make_actor` はアクターとその Adam オプティマイザを構築し、`distill` はこの回帰の Adam ステップを 1,000 回実行する。

```
def make_actor(seed, lr=1e-3):
    actor = d2l.SquashedGaussianPolicy(3, 1, scale=c, seed=seed)
    return actor, torch.optim.Adam(actor.parameters(), lr=lr)

def distill(actor, opt, obs, labels, rng, num_steps=1_000, batch_size=256):
    """Regress the actor's mean action, scale * tanh(mean), onto labels."""
    obs, labels = torch.as_tensor(obs), torch.as_tensor(labels)
    for _ in range(num_steps):
        i = torch.as_tensor(rng.integers(len(obs), size=batch_size))
        a = actor.scale * torch.tanh(actor.policy(obs[i])[0])
        loss = ((a - labels[i]) ** 2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
    return float(loss.detach())
```

`search_and_distill`（Figure 15.16）の各ラウンドでは、モデルを再当てはめし、保存された状態からランダムに抽出した 200 個に探索でラベルを付け、それらのラベルをアクターへと蒸留（distillation）し、探索を用いて行動しながら実エピソードを 1 つ収集する。アクターの訓練には現在のラウンドのラベルのみを使い、前のラウンドのラベルは保持しない。探索を用いて行動することで、各エピソードは現在のモデルとアクターが許す限り良質なものとなり、データは優れたコントローラが訪れる状態へと到達する。各状態の価値も予測するネットワーク、離散的な着手上の木探索、そして多数の対局にわたって探索のターゲットを保持するバッファを組み合わせれば、探索で行動し、ラベルを付け、蒸留するというこのループはそのまま AlphaZero のループになる。

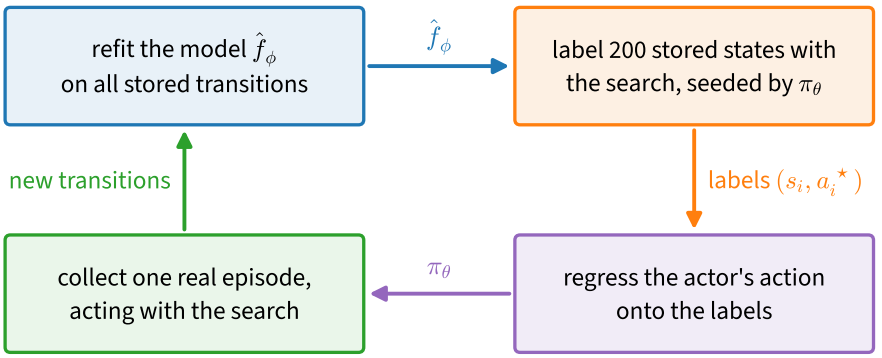


Figure 15.16: 探索と蒸留の 1 ラウンド。保存された全遷移でモデルを再当てはめし、アクターを初期値とする探索で保存された 200 個の状態にラベルを付け、それらのラベルへとアクターを回帰させ、探索を用いて行動して収集した 1 つの実エピソードが次のラウンドに新たな遷移を供給する。

このループはラウンドごとに、探索を用いて収集したエピソードの収益と、探索を一切使わずにアクター単独で行動したときのスコアを記録し、最後に 10 個の初期状態で探索を用いて行動したときのスコアを評価する。2 本の曲線は異なるものを測っている。1 本目はラウンドごとの 1 つの訓練エピソードであり、たまたま始まった初期状態からの結果であるため、エージェントの性能

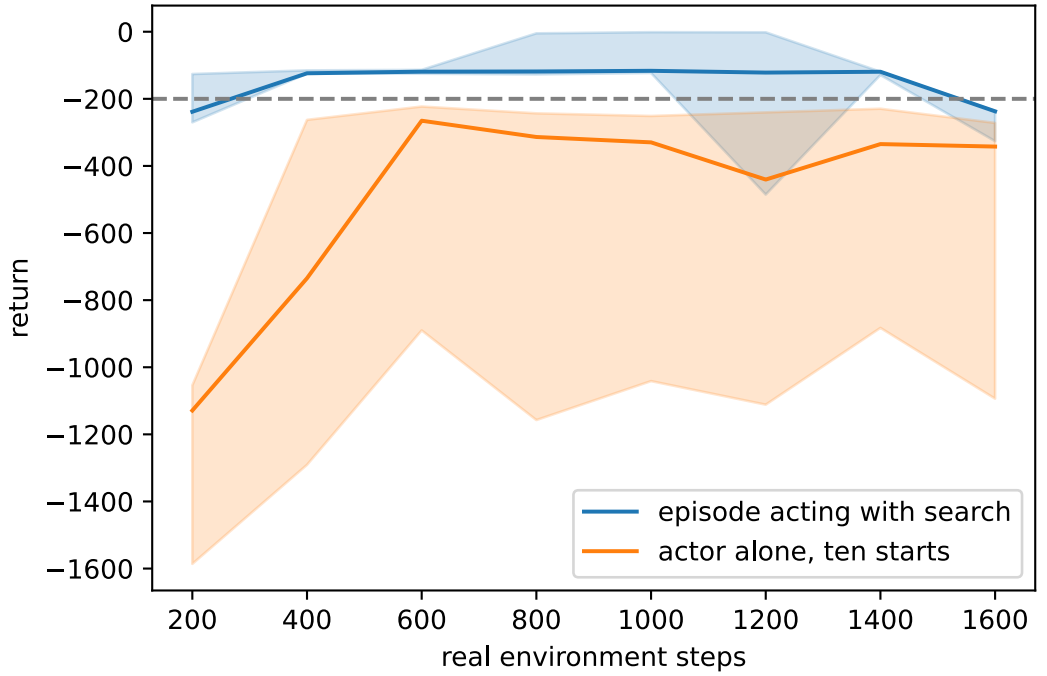
と同じくらい初期状態にも左右され、難しい初期状態に当たると一時的な落ち込みとして現れる。2 本目は 10 個のテスト初期状態にわたる平均であり、本節の他のすべての箇所で用いているスコアである。

```
def search_and_distill(seed, num_rounds=8, num_labels=200):
    """One random episode, then rounds of: refit the model, label stored
    states with the search, regress the actor onto the labels, collect one
    episode acting with search. Returns per round the collected return and
    the actor's score alone, and the final score acting with search."""
    env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
    rng, (model, opt) = np.random.default_rng(seed), make_model(seed)
    actor, actor_opt = make_actor(seed)
    buffer = d2l.ReplayBufferC(200 * (num_rounds + 1), 3, 1)
    run_episode(env, random_controller(rng), buffer, seed=seed)
    search = lambda obs: improve(model, actor, obs, rng)
    rounds = []
    for _ in range(num_rounds):
        fit_model(model, opt, buffer, rng)
        obs = buffer.obs[rng.choice(buffer.size, num_labels, replace=False)]
        distill(actor, actor_opt, obs, search(obs), rng)
        rounds.append([run_episode(env, search, buffer)[0],
                       score(actor.act_greedy)])
    return np.array(rounds), score(search)
```

1 回のランダムエピソードから 8 ラウンドを始めるため、ラウンド r のアクターは $200r$ 回の実ステップを経験している。3 つのシードを実行し、シード間の中央値と範囲をプロットする。

```
results = [search_and_distill(seed) for seed in range(3)]
rounds = np.array([r for r, _ in results])
d2l.plot_curves({'episode acting with search': rounds[:, :, 0],
                 'actor alone, ten starts': rounds[:, :, 1]},
                'real environment steps', 'return',
                x=200 * np.arange(1, 9), reference=-200)
print('final score on the ten starts, acting with search:',
      ' '.join(f'{s:.0f}' for _, s in results))
```

```
final score on the ten starts, acting with search: -191 -202 -222
```



15.6.3.3 探索と蒸留のラウンド

探索を用いて行動すると、エージェントは図示した 3 回の実行のいずれでも 2〜3 ラウンド以内に振り子を振り上げ、8 ラウンド後には 10 個の初期状態で約 −185〜−245 のスコアを出しており、大半の実行で先ほどのウォームスタート付きプランナに近い性能を示す。アクター単独でもすべての実行で改善し、第 1 ラウンド後の約 −900〜−1,600 から第 8 ラウンド後の約 −190〜−1,100 へと上がるが、あるラウンドから次のラウンドにかけてスコアが数百ポイント落ち込むこともある。ネットワークは探索が知っていることの一部を学習したものの、単独で行動すると力及ばない。

これらの実行だけではなぜ力及ばないのかまではわからず、3 つの原因が候補として考えられる。第 1 は、14.5 節が湖の環境で測定した分布シフト（distribution shift）である。ラベルは保存された状態から抽出されており、最初のランダムエピソードを除けば、それらは探索が訪れた状態である。探索を不完全に模倣したアクターは、単独で行動するとラベルの乏しい状態へと逸れていき、そこで誤差が累積する。DAgger（14.5.4.1 節）は湖の環境において、学習者自身の状態にラベルを付けることでこれを修復した。第 2 は訓練セットのサイズである。DAgger が全ラウンドにわたってラベルを集約し、AlphaZero が探索のターゲットをバッファに保持し続けるのに対し、ここでは各ラウンドで新たな 200 個のラベルのみでアクターを訓練し、以前のラベルを破棄している。第 3 はラベルそのものの性質である。各ラベルは 500 個のランダム候補のうち最良のものの最初のトルクであり、ノイズを含んだターゲットとなっている。演習問題 5 では最初の 2 つの原因を切り分ける。なお AlphaZero はテスト時にも探索するため、ネットワークは探索の出発点を良くするだけでよい。そうすれば探索とネットワークが互いを高め合う。より良いアクターがより良い探索の初期値を与え、より良い探索がアクターを鍛えるのである。

15.6.3.4 データとしての想像上の遷移

モデルは、プランナを持たない学習器にデータを供給することもできる。Dyna は実ステップとモデルからサンプリングした遷移による更新とを交互に織り交ぜ (Sutton 1990)、モデルベース方策

最適化（model-based policy optimization、MBPO）はこれをソフトアクタークリティックに対して実行する (Janner et al. 2019)。すなわち、数百ステップごとにモデルを再当てはめし、リプレイバッファ内の状態から現在のアクターをモデル内で数ステップだけロールアウトして、各訓練バッチの大半をこれらの想像上の遷移で埋めるのである（Figure 15.17）。想像上のデータは計算量が許す限りいくらでも生成できる。しかし、モデルの誤差がクリティックのターゲットに入り込むため、MBPO は短いロールアウトにとどめることで先ほど測定した累積誤差を抑えている。

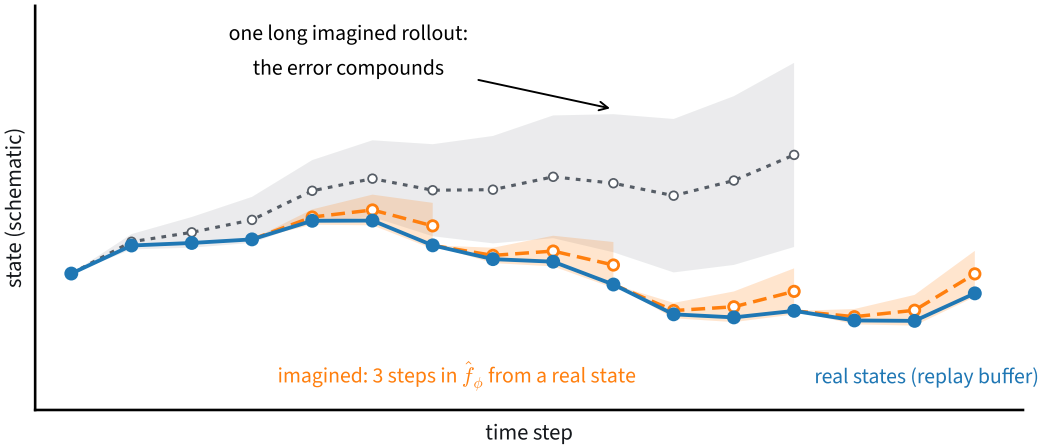


Figure 15.17: 分岐ロールアウトの模式図。短い想像上のロールアウト（オレンジ、破線、ここでは各 3 モデルステップ）はリプレイバッファ内の実際の状態（青）から始まるため、モデルの誤差（網掛けの帯）は数ステップ分しか累積しない。最初の状態からの 1 本の長い想像上のロールアウト（灰色、点線）はデータから大きく逸れていく。

これによって Pendulum で必要な実ステップ数は減るだろうか。温度を 0.2 に固定し、環境 1 ステップあたり勾配ステップを 10 回とし、3 ステップのロールアウトによる想像上の遷移 95% と実際の遷移 5% からなるバッチを用い、250 ステップごとにモデルを再当てはめする設定でソフトアクタークリティックを実行した。5 シードにわたってその貪欲アクターが初めて -200 以上のスコアを出したのは 1,500〜3,000 ステップ後であったが、モデルを使わずに同じく 1 ステップあたり 10 回更新した場合でも 1,500〜2,500 ステップであった。なお、これらの実行では別の固定された 10 個の初期状態セットを用いており、優れたコントローラの失点が本節のセットより少ないため、先ほど引用した実行よりも早く -200 に到達している。更新比率を揃えて比較すると、想像上の遷移を加えても実ステップ数は減らなかったのである。1 ステップあたり 1 回更新のソフトアクタークリティックに対して MBPO が得る改善の多くは、1 実ステップあたりの更新回数が増えたことに由来しうるものであり、それはモデルなしでもリプレイだけでまかなえる。実際、高い更新比率を用い、そのような比率が引き起こす過大評価を抑えるためにクリティックのアンサンブルを備えたモデルフリー学習器は、MBPO に匹敵するサンプル効率を達成している (X. Chen et al. 2021)。演習問題 6 でこの比較を再現する。モデルが最も力を発揮するのは、まさに本節で使った場面、すなわち行動する前に先読みし、探索を方策へと変換するところである。不確実な領域を把握できるモデルのアンサンブル、学習された潜在状態のモデル、学習された価値で終端する探索はこれらのアイデアをさらに発展させるものであり、15.9 節で解説する。

ここまでで、数百ステップのデータからダイナミクスのモデルを当てはめ、それを通して計画を立て、探索が見つけた行動を方策に教えられるようになった。どの段階でも、自身のエピソードが訪れた範囲かつ 25 ステップ先までに限ってモデルを信頼し、実際に行動して再当てはめすることでその信頼を検証できた。続く 15.7 節では、この「行動する」ことが取り除かれる。データは固定されたログとなり、そのログの外側でモデルや価値関数が約束することは一切検証できなく

なる。

15.6.4 まとめ

遷移には環境がどのように動くかが記録されており、最小二乗法で遷移に当てはめたネットワーク $\hat{f}_\phi(s,a) \to (\hat{s}', \hat{r})$ は、湖の環境における数え上げモデルを連続状態へと拡張する。その 1 ステップ誤差は小さいものの、ロールアウトに沿って累積し、データが一度も訪れていない領域では大きくなる。モデル予測制御はモデル内でトルク系列を探索し、最初のトルクを実行して次のステップで再び計画を立てる。ウォームスタートを用いれば、1〜3 エピソードから構築したプランナは、ソフトアクタークリティックが数千ステップを要した振り子の振り上げを数百実ステップで達成するが、各行動には数千回のモデル評価が必要となる。方策を初期値とする探索はその方策より低い収益を決して予測しないため方策を改善し、探索の行動へとアクターを回帰させることでその改善の一部が転移されるが、アクター単独で行動すると探索には遅れをとる。そのラベルは大半が探索の訪れた状態上にあり、ラウンドあたりの数が少なく、ノイズを含んでいるが、本節の実験だけではこの 3 つのうちどれが最も重要かまでは決まらない。また、想像上の遷移を訓練データとして用いても、1 ステップあたりの更新回数を揃えた条件ではソフトアクタークリティックの実ステップ数は減らなかった。

15.6.5 演習問題

1. [code] **計画の深さ**。プランナは $n = 25$ ステップ先まで見通す。
   1. コードを実行する前に、候補数を 500 に保ったまま $n = 10$ とした場合および $n = 50$ とした場合にテストスコアがどうなるかを、本節の 2 つの誤差の表を使って予測せよ。
   2. 両方の深さで 3 つのシードについて `mpc_run` を実行し、600 実ステップ後のスコアを $n = 25$ の場合と比較せよ。
   3. 真下からの振り上げには数十ステップにわたって勢いをつける必要がある。$n = 10$ の結果をその予測収益が区別できることとできないことから説明し、$n = 50$ の結果を誤差の表から説明せよ。1 回の行動にかかるコストは n に応じてどう変化するか。
2. [code] **ランダムデータの必要量**。シード 3 とシード 4 では、モデルが上半円をカバーするまでにプランナ自身の失敗エピソードが必要であった。
   1. 両方のシードについて、1、2、5 回のランダムエピソードにモデルを当てはめよ。データ中で振り子が到達した最高点（最大の $\cos\vartheta$）と、カバレッジの表と同様のプランナのエピソードに沿った各モデルの 25 ステップ誤差を報告せよ。
   2. ランダムデータのみから構築したプランナが、大半のテスト初期状態から振り上げるには何回のランダムエピソードが必要か。プランナ自身のエピソードが必要とした 600 ステップと比較し、その違いを説明せよ。
3. [code] **交差エントロピー法**。行動ごとにランダムシューティングを複数回繰り返し、サンプリング分布を上位の候補に再当てはめする手法を**交差エントロピー法**という (Rubinstein and Kroese 2004)。
   1. 5 ラウンドを実行するように `plan` を変更せよ。各ラウンドの後、計画の各トルクの平均と標準偏差を 500 個の候補のうち上位 50 個の平均と標準偏差に設定し、このガウス分布から次の 500 個を引くようにせよ。
   2. 600 実ステップ後のテストスコアおよび 1 行動あたりのモデル評価回数を、本節のウォームスタート付きランダムシューティングと比較せよ。
   3. ウォームスタートを使わず、毎ステップ平均 0 のトルク・標準偏差 c から始める設定でこの手法を実行せよ。1 ステップ内での洗練は、ステップをまたぐ洗練の代わりになるか。
4. [code] **信頼のシグナルとしての不一致**。●モデルのアンサンブルを使えば、どこが未知であるかを知ることができる (Chua et al. 2018)。
   1. シード 3 とシード 4 について、異なるシードを持つ 3 つのモデルをランダムエピソー

ドに当てはめよ。プランナのエピソードに沿って、1、5、25 ステップ後における 3 つのモデルの不一致（予測角度の標準偏差）と平均予測の誤差とを比較せよ。大きな不一致は大きな誤差を予測するか。

2. `plan` 内の各候補に対し、3 つのモデルの平均予測収益から、全ステップにわたる不一致の和の κ 倍を引いた値によってスコアを付けよ（ただし $\kappa \in \{0, 1, 10\}$ とする）。シード 3 とシード 4 の最初の MPC エピソードはどう変化するか。
3. 不一致に対するボーナスとなる $-\kappa$ を用いて小問 b を繰り返せ。まだデータを収集できるエージェントにはどちらの符号が適しており、もはや収集できないエージェントにはどちらの符号が適しているか。

5. [code] **アクター自身の状態に対するラベル**。●本節のループは探索を用いて行動しながらエピソードを収集し、各ラウンドではそのラウンドの 200 個のラベルのみでアクターを訓練している。
   1. 探索による収集はそのまま維持しつつ、ラベルを集約せよ。すなわち、各ラウンドでラベル付けした状態とトルクを 1 つの増大する集合に追加していき、集合全体でアクターへと蒸留せよ。5 つのシードを実行し、ラウンドごとのアクターのスコアを本節の結果と比較せよ。
   2. 次に、各ラウンドのエピソードをアクター単独（`actor.act_greedy`）で収集し、アクターのエピソードの状態に `improve` でラベルを付け、(a) と同様にラベルを集約せよ。これは探索をエキスパートとした 14.5 節の DAgger である。3 つのバージョンのうち最も安定したアクターを与えるのはどれか。また、この比較から本文で挙げた原因について何が言えるか。
   3. シードによっては、DAgger 版は一度も振り子を振り上げられないことがある。そのような実行のデータを調べ、本節のカバレッジの表を用いて失敗の理由を説明せよ。2 つのデータ収集方法はどうすれば組み合わせられるか。
6. [extended] **ソフトアクタークリティックのための想像上の遷移**。●●モデルベース方策最適化は、モデルが想像した遷移上でソフトアクタークリティックを訓練する (Janner et al. 2019)。
   1. `train_sac` を 15.5 節からノートブックにコピーせよ。ウォームアップ後、250 環境ステップごとにリプレイバッファ上で `DynamicsModel` を再当てはめし、バッファから抽出した 2,000 個の状態から現在のアクターをモデル内で 3 ステップロールアウトせよ。直近 4 回の再当てはめで生成した想像上の遷移を 2 つ目の `d2l.ReplayBufferC` に保持し、各バッチの 95% をそこから抽出せよ。
   2. 環境 1 ステップあたり 10 回の更新で 5 つのシードを 4,000 ステップ実行し、250 ステップごとに 10 個のテスト初期状態で貪欲アクターのスコアを評価せよ。モデルを使わない同じループである対照群も同じシードで実行せよ。各実行が初めて -200 以上のスコアを出すステップ数を報告せよ。
   3. なぜ対照群でも環境 1 ステップあたりの更新回数を同じにしなければならないのか。両群を 1 ステップあたり 1 回の更新で繰り返し、その違いを解釈せよ。
   4. ロールアウト長を 1 ステップおよび 10 ステップに変更せよ。本節の誤差の表からはどのような結果が予測され、実際にはどのような結果が観測されるか。
7. [code] **価値を用いた先読み**。本節のプランナは固定されたホライズンにわたって報酬を足し合わせ、それ以降のことは無視している。価値関数があれば、ホライズンを 1 ステップにできる。
   1. CartPole において、ランダムな行動による 5,000 個の遷移に 4 次元状態の `DynamicsModel` を当てはめよ。終了判定は CartPole の既知のルールに従い、ポールの角度とカートの位置から決定せよ。
   2. 15.4 節の DQN を 10,000 ステップ訓練し、$\operatorname{argmax}_a Q(s,a)$ による方法と、1 ステップ先読み $\operatorname{argmax}_a \left[\hat{r}(s,a) + \gamma \max_{a'} Q(\hat{s}'(s,a), a')\right]$ による方法の 2 通りで貪欲に行動させよ。5 つのシードについて 20 エピソードにわたる収益を比較せよ。
   3. 40,000 ステップの訓練後に同じ実験を繰り返せ。先読みが有効に働くのはどのような

ときであり、モデルの誤差が先読みの利点を上回るのはどのようなときか。

15.7 オフライン強化学習

本章でここまで扱ってきた手法はすべて、実際に行動を試すことができた。Q 学習がある行動を過大評価しても、エージェントはいずれその行動を取り、結果を見て自身のテーブルを修正した。また、15.6 節のプランナは、自身のエピソードで訪れた状態の近傍でのみ学習済みモデルを信頼し、必要に応じてさらなるエピソードを収集した。ここで、新たなエピソードを一切得られない状況を考えてみよう。病院には何年分もの治療記録があるが、患者を相手に実験するわけにはいかない。企業には昨年運用していた推薦システムのログがある。ロボット工学の研究室には何時間分もの遠隔操作による実演データがある。このような固定された記録だけから方策を学習できるのだろうか。また、その方策はどこまで良いものになり得るのだろうか。

この記録は何らかの**行動方策**（behavior policy）π_e によって収集されたものだが、Q 学習のターゲットは誰がその行動を選んだかに依存しないため、記録された遷移に対して Q 学習を実行できる。難しさはその先にある。学習された方策が行動方策を上回るには、行動方策がまれにしか取らなかった行動を優先しなければならないが、そうした行動の価値こそ最も少ないサンプルに基づいている。ターゲットに含まれる最大値演算は最大の推定値を選び出すが、最大の推定値はたまたま運が良かっただけのものが多く、それを検証する新たな試行は二度と行われない。**悲観主義**（pessimism）と呼ばれる対策では、根拠となるデータが少ない推定値ほど割り引いて疑う。これは 14.4 節で探索の原動力となった楽観主義とちょうど鏡像の関係にある。

本節では、厳密な最適解が既知である 14.2 節の滑りやすい湖を題材とし、学習されたテーブルが予測するすべての価値を真の値と照合できるようにする。まず、他の方策のデータを利用できる更新則はそもそもどれかを問う。次に、2 種類の固定ログから学習し、どの方策が獲得し得る値よりも予測値がどれだけ上回るかを測定した上で、悲観主義が何を修復し、いつ不要になるのかを確認する。

```
import gymnasium as gym
```

15.7.1 固定された記録からの学習

15.7.1.1 他の方策のデータを利用できるターゲット

ここまでの 2 章で扱った更新則は、2 つの系統に分けられる。方策勾配（policy gradient）は現在の方策自身の軌跡に関する期待値であるため、別の方策からのサンプルでは異なる量を推定することになる。したがって REINFORCE とアクター・クリティックは**オンポリシー**であり、PPO が 1 つのバッチを数エポックにわたって再利用できるのも、各行動の確率を比率で補正しているからにほかならない。一方、Q 学習のターゲット $r + \gamma \max_{a'} \hat{Q}(s', a')$ は、(s, a) に対して環境がどう応答したかには依存するが、a を選んだ方策には依存しない。そのため、以前の方策や別のエージェント、あるいは固定ログによって収集された遷移であっても、同じバックアップの有効なサンプルとなり、Q 学習は**オフポリシー**である。Figure 15.18 に、データの 3 つの利用形態を並べて示す。

14.3 節で扱った Q 学習のオンポリシー版である SARSA は、最良の行動ではなく、行動 a'（行動方策が s' で実際に取ったもの）を用いてブートストラップする。14.3 節からその時間差分誤差（temporal-difference error）を再掲する。

$$\delta_{\text{SARSA}} = r + \gamma\, Q(s', a') - Q(s, a). \tag{15.52}$$

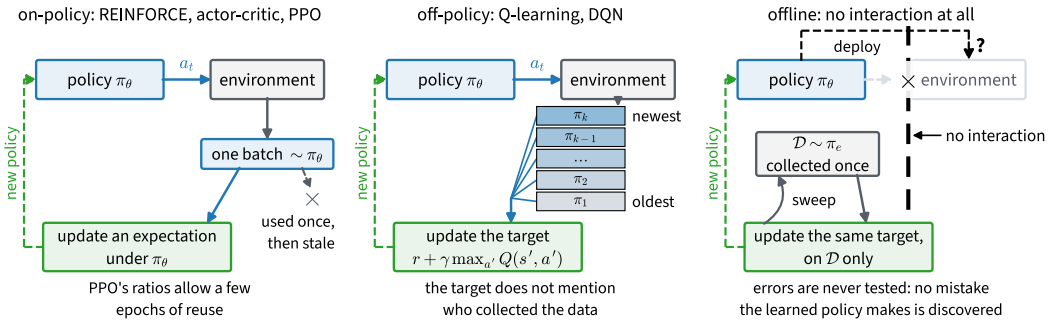


Figure 15.18: 3 つのデータ利用形態。(a) オンポリシー：各バッチは現在の方策から得られ、更新後には古くなる。PPO の比率補正により数エポックの再利用が可能になる。(b) オフポリシー：ベルマンターゲットは収集元の方策に依存しないため、バッファに過去の遷移を保持できる。(c) オフライン：固定されたデータセットを繰り返し走査し、運用する方策が選ぶ行動を新たな遷移で検証することはない。

$a' \sim \pi_e(\cdot \mid s')$ について平均を取ると、このターゲットは π_e のベルマン期待バックアップとなるため、SARSA はデータを収集した方策の価値 Q^{π_e} を学習する。他者のログから SARSA が学べるのは、その他者自身のことだけである。Q 学習はどのようなログからでも Q^* を学習し、演習問題 8（14.3 節）では崖の上でこの 2 つを並べて比較している。

オフポリシーだからといって、どのようなデータでもよいわけではない。Q 学習のテーブルの各要素の精度は背後にある遷移データ次第であるため、貪欲方策（greedy policy）が到達するすべての状態において選ぶ対は、実際に試されており、かつ推定に十分な回数だけ試されていなければならない。この要件を**被覆条件**（coverage condition）と呼ぶ。オンラインであれば、エージェントは推定が不確かな場所へ自ら向かうことで被覆の不足を補える。しかし**オフライン強化学習**（offline reinforcement learning）では、エージェントは固定されたデータセットしか受け取らないため (Levine et al. 2020)、ログが持つ被覆がすべてとなる。

15.7.1.2 滑りやすい湖の 2 種類のログ

滑りやすい湖には 16 個のセルと 4 つの行動がある。11 個のセルは穴でもゴールでもないため、エージェントが行動を選択できる状態行動対（state-action pair）は 44 個あり、あらゆる移動は 3 回に 2 回の割合で横方向に滑る。実験者は湖の遷移テーブルを知っており、それを `d2l.TabularMDP`、すなわち 14.1 節の配列 $P(s' \mid s, a)$ と $r(s, a)$ に読み込むが、以下の学習器がそれを目にすることは一切ない。各種類につき 40 個のログを生成する。

```
gamma, num_datasets = 0.95, 40      # discount; logs of each kind
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

真のモデルに対する 14.2 節の価値反復法（value iteration）が基準を与える。いかなる方策も、開始セルから $V^*(s_0)$ を超える価値を得ることはできない。`greedy` は行動価値のテーブルを行動確率のテーブルとして表された貪欲方策に変換し、`d2l.epsilon_greedy` と同様に同値を均等に分割する。もし同値をすべて最初の行動に割り当てる規則にすると、すべて 0 の行からは常に左へ歩いてしまうだろう。`mdp.policy_value` は 14.2 節の線形ベルマン方程式 $V^\pi = (I - \gamma P_\pi)^{-1} r_\pi$ を解くことで、そのような方策テーブルを厳密に評価する。以下で方策が**獲得する**価値はすべて、ロールアウトを行わずにこの方法で計算したものである。

```
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
Q_star = mdp.backup(V_star)

def greedy(Q):
    """The greedy policy of Q as a table of probabilities, ties split
    evenly, the rule of d2l.epsilon_greedy."""
    best = Q == Q.max(axis=1, keepdims=True)
    return best / best.sum(axis=1, keepdims=True)

print(f'V*(s0) = {V_star[0]:.3f}')
```

```
V*(s0) = 0.180
```

1 種類目のログは、能力をまったく持たないエージェントによるもので、一様ランダムな行動からなる 500 エピソードである。あらゆる場所ですべての行動を試すが、ゴールにはめったに到達しない。2 種類目は、有能だが不完全なオペレータによるもので、ϵ 貪欲な行動を Q^* のもとで $\epsilon = 0.2$ として 100 エピソード実行した記録である。したがってオペレータは確率 $0.8 + 0.2/4 = 0.85$ で最良の行動を取り、残りの確率でランダムな行動を取る。サンプルがオペレータの選択に集中するため、これを**狭い**（narrow）ログと呼ぶ。`make_log` は `d2l.rollout` を使ってログを収集する。この関数は方策をエピソード完了まで実行し、`d2l.Batch`（フィールドは `obs`、`act`、`rew`、`next_obs`、`term`）として遷移を返すものであり、シードによってログが固定される。`count` はログが各対を試した回数 $n(s, a)$ を集計する。また、行動方策自体を評価できるように、`logs` の各エントリにはその行動確率テーブルも保持しておく。

```
def make_log(env, behavior, num_episodes, seed):
    """A fixed log: complete episodes of behavior(obs, rng), seeded."""
    env.reset(seed=seed)
    rng = np.random.default_rng(seed)
    return d2l.rollout(env, behavior, num_episodes, rng)

def count(batch, shape=(16, 4)):
    """n(s, a): how often the log tried each state-action pair."""
    n = np.zeros(shape)
    np.add.at(n, (batch.obs, batch.act), 1)
    return n

def uniform(s, rng):
    return int(rng.integers(4))

def expert_eps(s, rng):          # the operator: epsilon-greedy on Q*, 0.2
    return d2l.epsilon_greedy(Q_star[s], 0.2, rng)

logs = {'uniform': (uniform, np.full((16, 4), 1 / 4), 500),
        'narrow': (expert_eps, 0.8 * greedy(Q_star) + 0.2 / 4, 100)}
live = np.isin(env.unwrapped.desc.flatten(), [b'S', b'F'])  # 11 cells
for name, (behavior, pi_e, num_episodes) in logs.items():
    batch = make_log(env, behavior, num_episodes, seed=0)
    n = count(batch)[live]
    print(f'{name:>7}: {len(batch):,} transitions, {(n > 0).sum()}/{n.size} '
          f'pairs tried, median support {np.median(n):.0f}; '
          f'behavior earns {mdp.policy_value(pi_e)[0]:.3f}')
```

```
uniform: 3,997 transitions, 44/44 pairs tried, median support 42; behavior earns 0.008
 narrow: 1,897 transitions, 42/44 pairs tried, median support 8; behavior earns 0.087
```

ランダムなエージェントが開始セルから獲得する価値は 0.008 と最適値 0.180 の約 1/20 にすぎないが、その 500 エピソードはすべての有効な対を試し、典型的な対を約 40 回試している。一方、オペレータは半分の長さのログから最適値の約半分にあたる 0.087 を獲得する。しかしサンプルが自身の選択に集中するため、2 つの対が未試行のまま残り、典型的な対には 8 サンプルしか与えられない。被覆と能力は逆方向に引き合い、これら 2 種類のログはその両端に位置している。

15.7.1.3 厳密な最適解に対する分布シフト

Q 学習は、環境に対して実行するのと同じようにログに対しても実行できる。`offline_q_sweeps` は記録された遷移をシャッフルし、14.3 節の更新を減衰ステップサイズ $1/(1+0.1\,n)$（`d2l.q_learning` と同じ形式で、n はその対のこれまでの更新回数）で適用し、この処理を 200 スイープ繰り返す。

```
def offline_q_sweeps(batch, gamma, num_sweeps=200, shape=(16, 4), seed=0):
    """The Q-learning loop of d2l.q_learning, step 1 / (1 + 0.1 n(s, a)),
    swept over the fixed log in a fresh random order each time."""
    Q, visits = np.zeros(shape), np.zeros(shape)
    rng = np.random.default_rng(seed)
    for _ in range(num_sweeps):
        for i in rng.permutation(len(batch)):
            s, a, s2 = batch.obs[i], batch.act[i], batch.next_obs[i]
            visits[s, a] += 1
            Q[s, a] += (batch.rew[i] + gamma * (1 - batch.term[i])
                        * Q[s2].max() - Q[s, a]) / (1 + 0.1 * visits[s, a])
    return Q
```

各スイープは同じ遷移を再訪するため、テーブルはログのみに依存する不動点（fixed point）に収束する。ログを集計して推定モデルを作ろう。$\hat{P}(s' \mid s,a)$ は (s,a) の試行のうち s' に到達した割合であり、$\hat{r}(s,a)$ はその平均報酬である。この不動点は、この**経験 MDP**（empirical MDP）における最適行動価値のテーブル、すなわち 14.3 節の確実性等価（certainty-equivalent）なテーブルである。`offline_q` は集計されたモデルに対する価値反復法によってこれを直接計算する。引数 `kappa` は各バックアップの内部で差し引かれる対ごとのペナルティであり、後ほど悲観主義を扱うまでは 0 のままとする。未試行の対に対応する行はすべて 0 であるため、どこにも遷移せず価値も 0 となる。この湖では報酬が決して負にならないため、どの方策も取り得る最小の価値は 0 であり、0 でのクリッピングによってペナルティが価値をそれ未満に押し下げるのを防ぐ。正の `kappa` を与えると、この関数は未試行の対に無限大のペナルティも科すが、クリッピングによって同じ下限である 0 に戻されるため、悲観主義を導入しても未試行の対は素朴なテーブルが置いたのと同じ位置にとどまる。

```
def offline_q(batch, gamma, kappa=0.0, num_iters=500, shape=(16, 4)):
    """Where offline Q-learning with decaying steps converges: value
    iteration on the log's empirical MDP. kappa > 0 subtracts
    kappa / sqrt(n(s, a)) inside every backup, distrusts untried pairs
    entirely, and clips at zero, the floor this environment guarantees."""
    n = count(batch, shape)
    P, r = np.zeros(shape + shape[:1]), np.zeros(shape)
    np.add.at(P, (batch.obs, batch.act, batch.next_obs), 1)
    np.add.at(r, (batch.obs, batch.act), batch.rew)
```

```
    m = np.maximum(n, 1)                          # untried: rows of zeros
    mdp_hat = d2l.TabularMDP(P / m[..., None], r / m, gamma)
    penalty = np.where(n > 0, kappa / np.sqrt(m),
                       np.inf if kappa > 0 else 0.0)
    Q = np.zeros(shape)
    for _ in range(num_iters):
        Q = np.maximum(mdp_hat.backup(Q.max(axis=1)) - penalty, 0.0)
    return Q
```

このループと直接解法は一致するだろうか。最初の一様ログに対して両方を実行してみよう。

```
batch = make_log(env, uniform, 500, seed=0)
Q_loop, Q_solve = offline_q_sweeps(batch, gamma), offline_q(batch, gamma)
print(f'largest difference between the two tables: '
      f'{np.abs(Q_loop - Q_solve).max():.3f}')
print(f'their greedy policies earn {mdp.policy_value(greedy(Q_loop))[0]:.3f}'
      f' and {mdp.policy_value(greedy(Q_solve))[0]:.3f}')
```

```
largest difference between the two tables: 0.006
their greedy policies earn 0.137 and 0.137
```

2 つのテーブルの差はすべての要素で 1/100 未満であり、両者の貪欲方策が獲得する価値も一致する。ログ 1 個あたり、ループは数秒かかるのに対し直接解法は数ミリ秒で済むため、以下の実験では `offline_q` を呼び出す。テーブルに残る誤差がログのサイズとともにどう縮小するかは、演習問題 3 で問う。

2 つ目の学習器は 14.5 節の行動クローニング（behavior cloning）であり、ログを収集した方策に適合させる手法である。テーブル形式では、交差エントロピーによる訓練は各状態におけるログの行動頻度を復元するため、`clone_policy` はその頻度を集計し、ログが一度も訪れなかった状態では一様に行動する。クローンが予測するのはログが実際に達成した値、すなわちエピソードの割引収益の平均である。

```
def clone_policy(batch, shape=(16, 4)):
    """Behavior cloning on a table: the log's empirical action
    distribution, uniform wherever the log is silent."""
    n = count(batch, shape)
    total = n.sum(axis=1, keepdims=True)
    return np.where(total > 0, n / np.maximum(total, 1), 1 / shape[1])
```

各種類 40 個のログそれぞれに対して、ペナルティなしの素朴なテーブルとクローンを適合させ、開始セルにおいてそれぞれが予測する値と、その方策が獲得する価値を記録する。`fit_and_score` は任意のペナルティ `kappa` のもとでテーブルに対してこの処理を実行し、`Batch` のメソッド `episode_returns` は各記録エピソードの割引収益（クローンの予測のもとになる値）を与える。

```
def fit_and_score(batches, kappa=0.0):
    """Per log: the table's own prediction max_a Q(s0, a) and the exact
    value of its greedy policy."""
    fits = [offline_q(batch, gamma, kappa) for batch in batches]
    return (np.array([Q[0].max() for Q in fits]),
            np.array([mdp.policy_value(greedy(Q))[0] for Q in fits]))
```

```
datasets, pred, value = {}, {}, {}   # predicted and earned values
for name, (behavior, _, num_episodes) in logs.items():
    datasets[name] = [make_log(env, behavior, num_episodes, seed)
                      for seed in range(num_datasets)]
    pred[name, 'naive'], value[name, 'naive'] = fit_and_score(datasets[name])
    pred[name, 'clone'] = np.array(
        [batch.episode_returns(gamma).mean() for batch in datasets[name]])
    value[name, 'clone'] = np.array(
        [mdp.policy_value(clone_policy(batch))[0]
         for batch in datasets[name]])
```

summary は 40 個のログにわたる中央値と、予測値が最適値およびその方策自身の厳密な価値を上回ったログの個数を出力する。tally は、一方の学習器の方策が獲得する価値が他方より高いログ、低いログ、等しいログの個数をそれぞれ数える。

```
def summary(pred, value, v_star):
    """Medians; on how many logs the prediction exceeds V*(s0) and the
    exact value of the learner's own policy."""
    return (f'predicts {np.median(pred):.2f}, earns '
            f'{np.median(value):.2f}; above V* on {(pred > v_star).sum():2d}'
            f', above own value on {(pred > value + 1e-9).sum():2d}')

def tally(value, other):
    """On how many logs value is higher, lower, equal to other."""
    diff = value - other
    return (diff > 1e-9).sum(), (diff < -1e-9).sum(), (abs(diff) <= 1e-9).sum()

for name in logs:
    for arm in ['clone', 'naive']:
        print(f'{name:>7} {arm}: ' + summary(pred[name, arm],
                                              value[name, arm], V_star[0]))
    wins = tally(value[name, 'naive'], value[name, 'clone'])[0]
    print(f'{"":>7} the naive table beats the clone on {wins} of '
          f'{num_datasets} logs')
```

```
uniform clone: predicts 0.01, earns 0.01; above V* on  0, above own value on 17
uniform naive: predicts 0.20, earns 0.14; above V* on 28, above own value on 36
        the naive table beats the clone on 40 of 40 logs
 narrow clone: predicts 0.09, earns 0.09; above V* on  0, above own value on 18
 narrow naive: predicts 0.25, earns 0.11; above V* on 39, above own value on 39
        the naive table beats the clone on 27 of 40 logs
```

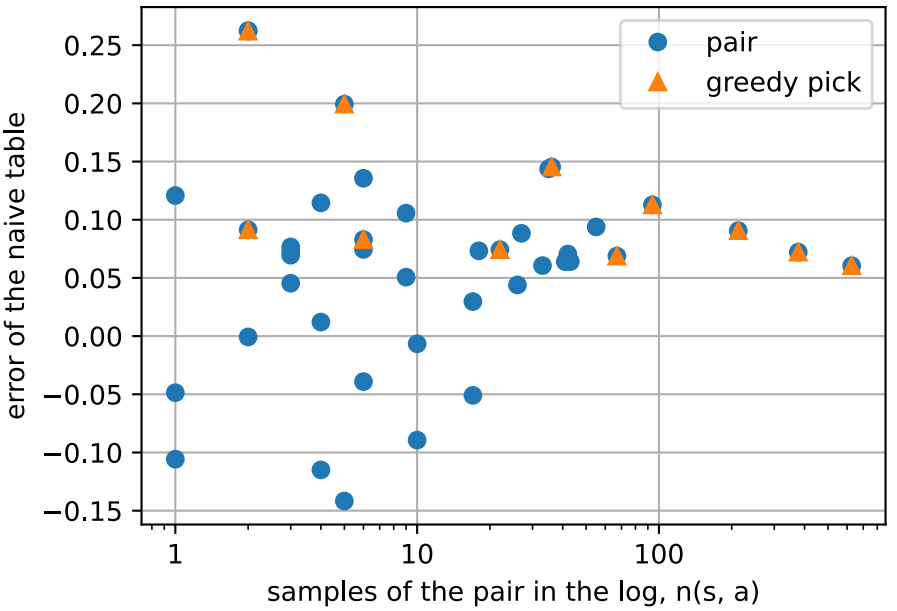
一様ログでは、素朴なテーブルの予測値は 40 個中 36 個のログで自身の方策が獲得する価値を上回り、28 個のログではどの方策が獲得し得る価値よりも高くなる。予測値の中央値は約 0.20 であり、最適値 0.18 を上回る。狭いログではこの乖離がさらに広がる。予測値は 40 個中 39 個のログで V^* を上回り、中央値は 0.25 に達する一方、方策が実際に獲得する価値は約 0.1 にとどまる。学習器が予測する値とその方策が獲得する価値を比較することは、オフライン強化学習における標準的な診断手法であり、深層ネットワークを用いるとこの乖離ははるかに大きくなり得る (Fujimoto et al. 2019)。

この膨れ上がった予測値はどこから生じるのだろうか。最初の狭いログを取り上げ、試行された各有効対について、そのサポート $n(s,a)$ に対する素朴なテーブルの誤差 $\hat{Q}(s,a) - Q^*(s,a)$ を

プロットし、各セルで貪欲方策が選ぶ対に印を付ける（2 つの行動が同値の場合、argmax は最初の行動に印を付ける一方、greedy は同値を均等に分割するが、これらのログでは選ばれた行動の間で同値が生じることはまれである）。

```
batch = datasets['narrow'][0]
n, Q = count(batch), offline_q(batch, gamma)
err, tried = Q - Q_star, live[:, None] & (n > 0)
cells = np.flatnonzero(live)
picks = Q[cells].argmax(axis=1)
print('picks with n <= 5: ' + ', '.join(
    f'cell {s} (n = {n[s, a]:.0f}, {err[s, a]:+.2f})'
    for s, a in zip(cells, picks) if n[s, a] <= 5))
d2l.plot([n[tried], n[cells, picks]], [err[tried], err[cells, picks]],
         xlabel='samples of the pair in the log, n(s, a)',
         ylabel='error of the naive table', legend=['pair', 'greedy pick'],
         xscale='log', fmts=('C0o', 'C1^'), figsize=(5, 3.5))
```

```
picks with n <= 5: cell 2 (n = 2, +0.09), cell 10 (n = 5, +0.20), cell 14 (n = 2, +0.26)
```



20 個を超えるサンプルを持つ対はすべて、約 0.05 から 0.15 だけ上向きに誤差を生じている。それら自身の遷移は精度よく推定されているものの、ターゲットが後続セルの価値からブートストラップしており、その後続セルがさらに先のセルの誤差を引き継ぎ、各後続バックアップの最大値演算がその誤差を上向きに偏らせるためである。サンプル数が約 10 個を下回ると、誤差は -0.15 から $+0.26$ まで両方向に広がり、貪欲方策が選んだサポートの薄い対にはプロット中で最大の 2 つの誤差が含まれている。具体的には、選ばれた対のうち 2 つは 2 サンプル、1 つは 5 サンプルにしか基づいておらず、そのうち 2 つは 0.26 および 0.20 と、最適値そのものを超える幅で過大評価されている。1 つのログだけでは偶然かもしれないので、40 個すべての狭いログにわたって選ばれた対を集約し、1〜5 サンプルにしか基づかない選択を、同じくサポートが薄いすべての対と比較してみよう。

```
pick_n, pick_err, all_n, all_err = [], [], [], []
for batch in datasets['narrow']:
    n, Q = count(batch), offline_q(batch, gamma)
    picks = Q[cells].argmax(axis=1)
    pick_n.append(n[cells, picks])
    pick_err.append((Q - Q_star)[cells, picks])
    all_n.append(n[live].ravel())
    all_err.append((Q - Q_star)[live].ravel())
pick_n, pick_err, all_n, all_err = map(
    np.concatenate, (pick_n, pick_err, all_n, all_err))
thin, thin_all = (pick_n >= 1) & (pick_n <= 5), (all_n >= 1) & (all_n <= 5)
print(f'greedy picks resting on 1 to 5 samples: {thin.mean():.0%} of picks')
print(f'their median error {np.median(pick_err[thin]):+.2f}; all pairs '
      f'with 1 to 5 samples {np.median(all_err[thin_all]):+.2f}')
```

```
greedy picks resting on 1 to 5 samples: 23% of picks
their median error +0.22; all pairs with 1 to 5 samples +0.08
```

貪欲方策の選択の約 1/4 は 5 個以下のサンプルに基づいており、これらの選択は約 0.2、すなわち同程度にサポートが薄い対の典型的な誤差の 2〜3 倍も過大評価されている。最大値演算はサポートの薄い対を無作為に選んでいるわけではない。たまたま誤差が正に振れた薄い対を選び出しているのであり、これは 15.4.3.1 節で見た最大化バイアス（maximization bias）にほかならない。さらに次のバックアップが、その膨らんだ価値を手前のセルへと伝播させる。その結果として、ログに含まれる行動と学習された方策が取る行動の間に生じる不一致を**分布シフト**（distribution shift）と呼ぶ。オンラインであれば、これは自然に修正される。エージェントが過大評価された行動を実際に取り、その価値を引き下げる遷移を収集して次へ進むからである。しかしオフラインではログが決して変化しないため、誤差はそのまま残る。

15.7.1.4 クローンとスティッチング

クローンはほぼ較正（calibration）されている。その予測値はログ自身の平均収益であり、行動方策の価値の不偏推定量となっていて、クローン自身の価値もそれに近く近似される。実際、予測値は約半数のログでクローンの価値を上回り、残りのログで下回る。しかし一様ログにおいては、まさにその性質が弱点となる。ランダムウォークを再現してしまい、獲得する価値は 0.01 にとどまるのである。

素朴なテーブルは 40 個すべての一様ログでクローンを上回り、最適値の 3/4 にあたる約 0.14 を獲得する（データを収集した方策の獲得価値はわずか 0.008 である）。その理由は遷移ごとのバックアップにある。あるランダムなエピソードがセル 10 に到達した後に穴へ落ち、別のエピソードが別の経路でセル 10 に到達してから滑ってゴールへ進んだとしよう（Figure 15.19）。バックアップはセル 10 を 2 番目のエピソードの後続経路によって評価し、その価値を 1 番目のエピソードの経路に沿って逆向きに伝播させることで、単一のエピソードでは一度もたどられなかった経路を合成する。この合成は**スティッチング**と呼ばれ、これによってオフライン Q 学習はデータを収集した方策を上回ることができる。一方、狭いログではクローンが強力なベースラインとなる。0.087 を獲得するオペレータを模倣して自身も約 0.09 を獲得するのに対し、素朴なテーブルはサポートの薄い選択に惑わされ、40 個中 27 個のログでしかクローンに勝てない。

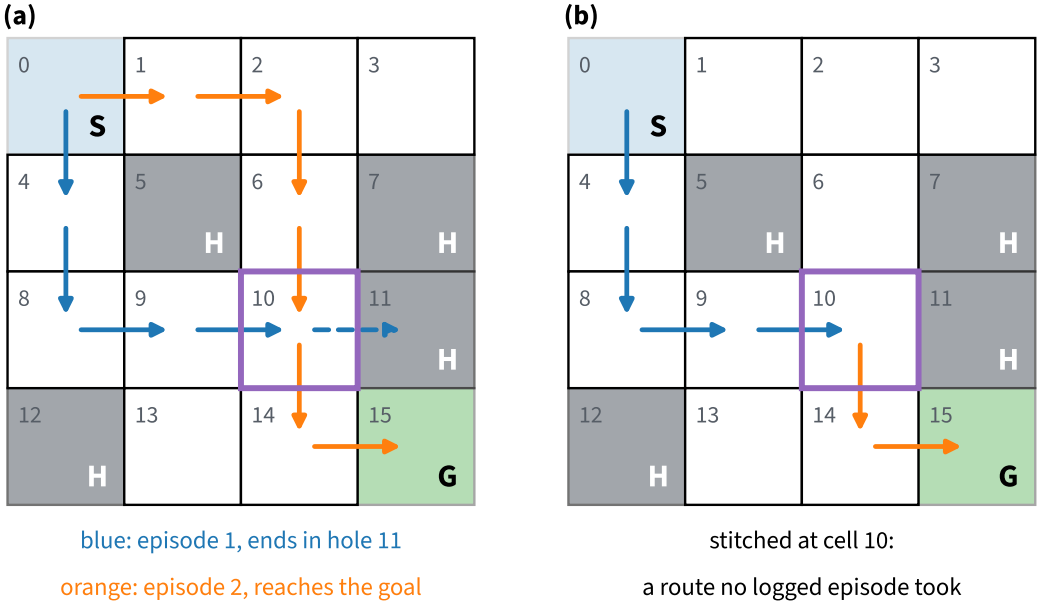


Figure 15.19: 湖におけるスティッチング。(a) 記録された 2 つのエピソードがセル 10（枠線付き）を通過する。1 つ目（青）は左端を下って穴 11 に落ち、2 つ目（オレンジ）は最上段を進んでゴールへ到達する。(b) 遷移ごとのバックアップが 1 つ目のエピソードの経路と 2 つ目のエピソードの後続経路をつなぎ合わせ、どの記録エピソードもたどらなかった経路を作る。

15.7.2 悲観主義

15.7.2.1 手計算で見る 2 本腕バンディット

湖の学習器を変更する前に、最小の例を取り上げよう。2 本腕バンディットのオフラインログに、常に $1/2$ の報酬を与える腕 1 を 100 回引いた記録と、報酬 1 を確率 $1/4$ で与えそれ以外は 0 を与える腕 2 を 1 回だけ引いた記録があるとする。4 回に 1 回起きることだが、その 1 回の試行が 1 を出したとしよう。すると経験平均が最大の腕は 1 対 0.5 で腕 2 となり、それを選ぶと平均で 0.25 しか獲得できない。

信頼下限（lower confidence bound、LCB）は、各経験平均 $\hat{r}(a)$ から腕の試行回数 $n(a)$ に応じて縮小する半径を差し引く。報酬が $[0,1]$ に収まるとき、ヘフディングの不等式は次の半径を与える。

$$b(a) = \sqrt{\frac{\log(2K/q)}{2\,n(a)}} \tag{15.53}$$

ここで K は腕の本数、q は失敗確率である。少なくとも確率 $1-q$ で、すべての腕の真の平均はその経験平均から $b(a)$ 以内に収まる。$K=2$、$q=0.1$ とすると、$b(1)=\sqrt{\log 40/200}\approx 0.136$、$b(2)=\sqrt{\log 40/2}\approx 1.358$ である。下限は腕 1 が $0.5-0.136=0.364$、腕 2 が $1-1.358=-0.358$ となるため、この規則は腕 1 を選ぶ (Rashidinejad et al. 2021)。その 1 回の試行が出し得た報酬について平均すると、経験平均が最良の腕を選ぶ規則の獲得価値は $\frac{3}{4}\cdot 0.5+\frac{1}{4}\cdot 0.25\approx 0.44$ であるのに対し、下限規則は毎回 0.5 を獲得する。

悲観主義には代償もある。もし腕 2 が平均して 0.7 を与え、経験平均も 0.7 だったとしても、$0.7-\sqrt{\log 40/(2n_2)}$ が 0.364 を上回るまでこの規則は腕 2 を拒み続ける。それが初めて起きるのは $n_2=17$ 回引いた時点である。悲観的な学習器は、データが裏付けを与えるまで良い行動を見送るのである。

15.7.2.2 1 行で表されるペナルティ

湖の学習器にも同じ処置を施そう。素朴なテーブルの各要素はノイズを含む $n(s,a)$ 個のターゲットを平均しているため、その標本誤差（sampling error）は $1/\sqrt{n(s,a)}$ のオーダーで縮小する。$\kappa>0$ のとき、`offline_q` はすべてのバックアップの内部で $\kappa/\sqrt{n(s,a)}$ を差し引き、結果を 0 でクリッピングする。

$$Q(s,a)\leftarrow \max\Big(\hat{r}(s,a)+\gamma\sum_{s'}\hat{P}(s'\mid s,a)\max_{a'}Q(s',a')-\frac{\kappa}{\sqrt{n(s,a)}},\,0\Big). \tag{15.54}$$

更新式自体は素朴なものと同じであり、唯一の違いはログが最も手薄な箇所で最大となるペナルティである。一度も試されなかった対は、ペナルティの有無にかかわらず下限の 0 にとどまる。この形式は、悲観的価値反復法（pessimistic value iteration）の表形式版である (Jin et al. 2021)。ここで κ がヘフディングの不等式の定数ではなく自由なスケールパラメータとなっているのは、ある対のターゲット群が同じ推定後続価値からブートストラップしており、互いに独立ではないためである。

このペナルティは 14.4 節の信頼上限（upper confidence bound）規則、すなわち $\hat{\mu}(a)+\kappa\sqrt{\log t/n(a)}$ が最大の腕を選ぶ規則のボーナスと鏡像関係にある。どちらの項も $1/\sqrt{n}$ に比例して縮小するが、フィードバックの有無が異なるため符号が逆になる。オンラインでは、楽観的

な推定値がエージェントにその行動を試させ、その試行が推定値を修正する。オフラインでは試行が不可能なため、同じ不確実性を差し引き、データがサポートするときにのみその行動を選ぶのである。

15.7.2.3 狭いデータと一様なデータ

それでは、同じ 80 個のログに対して `offline_q`、すなわち Equation 15.54 のバックアップを $\kappa = 0.1$ として再実行してみよう。

```
kappa = 0.1
for name in logs:
    pred[name, kappa], value[name, kappa] = fit_and_score(datasets[name],
                                                          kappa)
    print(f'{name:>7} kappa {kappa}: ' + summary(
        pred[name, kappa], value[name, kappa], V_star[0]))
    better, worse, tied = tally(value[name, kappa], value[name, 'naive'])
    print(f'{"":>7} against naive: better on {better}, worse on {worse}, '
          f'tied on {tied}; beats the clone on '
          f'{tally(value[name, kappa], value[name, "clone"])[0]}')
```

```
uniform kappa 0.1: predicts 0.05, earns 0.10; above V* on  0, above own value on  3
        against naive: better on 4, worse on 28, tied on 8; beats the clone on 40
 narrow kappa 0.1: predicts 0.13, earns 0.14; above V* on  4, above own value on 15
        against naive: better on 29, worse on 1, tied on 10; beats the clone on 39
```

狭いログでは、悲観主義はバンディットの例で見たとおりの効果を発揮する。予測値は 0.25 から約 0.13 まで下がり、最適値を上回るのはわずか 4 個のログになる。方策が獲得する価値は 40 個中 29 個のログで素朴なテーブルを上回り、下回るのは 1 個のみで、中央値は約 0.14 となり、40 個中 39 個のログでクローンを上回る。

一方、一様ログでは同じペナルティによって過大評価は解消され、最適値を上回る予測はなくなるものの、40 個中 28 個のログで収益が下がり、上がるのは 4 個にとどまる。すべての行動が何十回も試されている状況では、警戒すべき幸運な推定値はほとんどなく、対ごとに異なるペナルティは十分にサポートされた行動のうちどれが最良に見えるかを入れ替えるだけに終わる。悲観的オフライン学習の理論は、いつどちらの結果になるかを示している。信頼下限を用いる学習器の保証に必要なのは、ログがすべての対をカバーすることではなく、**最適**方策（optimal policy）の対をカバーすることである (Rashidinejad et al. 2021)。狭いログは最適方策の選択を厚く、それ以外を薄くカバーしているため悲観主義が有効に働くが、一様ログはすべてをカバーしているため素朴なテーブルに保護は不要なのである。

15.7.2.4 ペナルティの選択

κ はどの程度の大きさにすべきだろうか。値を大きくするとテーブルはより慎重になる。同じログに対して $\kappa = 0.3$ を試してみよう。

```
kappa = 0.3
for name in logs:
    p, v = fit_and_score(datasets[name], kappa)
    better, worse, _ = tally(v, value[name, 'naive'])
    print(f'{name:>7} kappa {kappa}: predicts {np.median(p):.2f}, earns '
          f'{np.median(v):.2f}; against naive better on {better}, '
```

```
            f'worse on {worse}')
```

```
uniform kappa 0.3: predicts 0.00, earns 0.02; against naive better on 0, worse on 40
 narrow kappa 0.3: predicts 0.00, earns 0.05; against naive better on 4, worse on 36
```

$\kappa = 0.3$ では、ほぼすべてのログで開始価値が 0 まで押し下げられ、方策は 40 個の狭いログのうち 36 個、また 40 個すべての一様ログで素朴なテーブルに敗れる。理由は、ペナルティがバックアップのたびに課され、価値が経路に沿った割引和だからである。各対が約 n サンプルを持つ経路に沿うと、課されるペナルティの合計は高々約 $\kappa/(\sqrt{n}\,(1-\gamma))$、すなわち $\gamma = 0.95$ では 1 ステップあたりの 20 倍に達する。もっとも、エピソードはゴールまたは穴で終了するため、経路に沿って実際に累積するペナルティはこれより少ない。経路上のすべての対に 1,000 個を超えるサンプルがあれば、この上界ですら $V^{*}(s_0) = 0.18$ を（$\kappa = 0.3$ であっても）下回るが、100 エピソードのログでは開始セルに数百サンプルあるだけで、その先ははるかに少ない。同じオペレータから 5 倍のデータを得れば、1 ステップあたりのペナルティは $\sqrt{5}$ 分の 1 に縮小する。すると最適経路上の大半の対は数百サンプルを持つようになり、上界が求める 1,000 サンプルには届かないものの、演習問題 1 で示すように、そこでは $\kappa = 0.3$ でもほとんど悪影響はない。実際に累積するペナルティがすでに十分小さいからである。ペナルティはデータ量に見合っていなければならず、したがって κ の選択はモデル選択（model selection）の問題となる。

15.7.2.5 較正とモデル選択

ここまでの比較はすべて実験者が持つ湖の厳密なモデルを用いたが、実際のオフライン運用ではそのようなモデルは存在しない。運用時に手元にあるのはログだけであり、κ の各値や学習された各テーブルについて、候補となる方策がどれほどうまく機能するかをそのログのみから推定しなければならない。1 つ目の推定量は、候補方策と行動方策のもとでの各行動の確率比によって記録されたエピソードに重み付けし直す手法である。その重みはエピソード全体にわたる確率比の積、すなわち 15.2 節で見たホライズンとともに分散が増大する積であり、貪欲な候補方策に対しては大半のエピソードの重みが 0 になる。もう 1 つの推定量は、ログのうちホールドアウトした部分から作った経験 MDP で候補方策を評価するものだが、その部分が候補方策の選択をカバーしている範囲でしか信頼できない。較正については部分的な確認が可能であり、自身のログ内の最良の割引収益を上回る値を予測するテーブルは疑わしい。ただし、この湖ではこの検査が引っかかることは一度もない。どのログでも最良のエピソードは約 0.5 以上の収益を上げており、本節のあらゆる予測値を上回るからである。演習問題 5 では、一様ログに対してこれら両方の推定量を試す。

15.7.3 表形式を超えて

15.7.3.1 方策または価値の制約

連続的な状態と行動では出現回数を数えられないが、本節で見た 2 つの対処法はそのまま残る。ログがサポートする行動の近くに方策を保つか、サポートしない行動の価値を押し下げるかである。1 つ目は、行動方策を参照方策とする 15.3 節の KL 正則化目的関数である。各状態において、方策は次の式を最大化する。

$$\mathbb{E}_{a \sim \pi(\cdot \mid s)}\big[\hat{Q}(s,a)\big] - \beta\, D_{\mathrm{KL}}\big(\pi(\cdot \mid s) \,\|\, \pi_e(\cdot \mid s)\big), \tag{15.55}$$

このため、ログが一度も取らなかった行動には無限大のペナルティが課される。実際には π_e は未知であり、ログで代用する。たとえば TD3+BC は、方策の行動と記録された行動の二乗距離をアクターの目的関数に加える (Fujimoto and Gu 2021)。定数項を除けば、この二乗距離は方策の行動を中心とするガウス分布のもとでの記録行動の負の対数尤度に等しいため、この項は逆向きの

ダイバージェンス $D_{\mathrm{KL}}(\pi_e \| \pi)$ のサンプル推定量となっている。次節の結びで扱う直接選好最適化（direct preference optimization、DPO）は、同じ制約のもとで参照方策を π_e の役割に据え、応答対の選好からなる固定データセットから学習する。

2 つ目の対処法は、カウントを必要とする $\kappa/\sqrt{n}$ を、ネットワークの出力だけで計算できるペナルティに置き換える。保守的 Q 学習（conservative Q-learning、CQL）は、記録された状態におけるすべての行動の価値を引き下げつつ、記録された行動の価値を引き上げる項を時間差分損失に加える。離散行動の場合、この項は $\alpha_{\mathrm{c}}\big(\log \sum_{a'} e^{Q(s,a')} - Q(s,a)\big)$ である (Kumar et al. 2020)。この項の前半は行動にわたるソフト最大値（soft maximum）であるため、その勾配はソフトマックスの重みに比例してすべての価値を引き下げる。すなわち、ログがサポートしているかどうかにかかわらず、最良に見える行動ほど大きく引き下げられ、記録された行動だけが引き上げられる（Figure 15.20）。Table 15.1 に、これら 2 つの系統と本節でカウントを用いて行った処理との対応をまとめる。

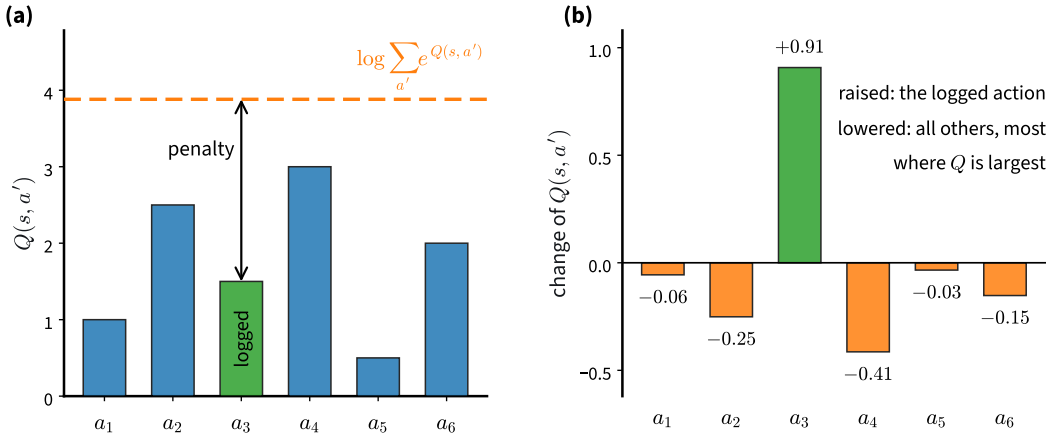


Figure 15.20: 記録された 1 つの状態における保守的ペナルティ（6 つの行動のうちログが保持するのは a_3）。(a) ペナルティは価値のソフト最大値 $\log \sum_{a'} e^{Q(s,a')}$（破線）と記録された行動の価値との差である。(b) ペナルティに対してステップサイズ 1 の勾配ステップを 1 回適用すると、すべての価値がそのソフトマックスの重みだけ引き下げられ（最大の価値ほど大きく下がり）、記録された行動の価値にのみ 1 が加算されて唯一上昇する。変化量の和は 0 になる。

Table 15.1: オフライン手法の 2 つの系統と、湖の実験における対応物。

系統	制約の対象	本節での対応物	代表的手法
方策制約	方策を記録された行動の近くに保つ	極端な場合としてのクローン	TD3+BC
価値ペナルティ	サポートされない行動の価値を押し下げる	`offline_q` の $\kappa/\sqrt{n}$	CQL

最大値演算の対象を記録された行動のみに限定する厳格な規則を設けても、湖の実験では何も変わらない。未試行の対の価値はすでに下限の 0 であり、最大値に選ばれることは決してないからである。そこでの難しさはサポートの欠如ではなく、サポートの薄さにあった。本章の関連文献（15.9 節）では、これら 2 つの考え方に基づく深層手法を比較している。

15.7.3.2 収益条件付け

3 つ目のアプローチは、最大値演算そのものを回避する。Decision Transformer は記録された各軌跡を残余収益（return-to-go）・状態・行動の系列として扱い、各行動を予測するよう自己回帰モデルを訓練し、テスト時には高い目標収益で条件付ける (L. Chen, Lu, Rajeswaran, Lee, Grover, Michael Laskin, et al. 2021)。通常の教師あり学習によって収益で条件付けた小さなネットワークでもその性能の大部分が再現されるため、改善の多くは系列アーキテクチャではなく収益による条件付け自体に由来している (Emmons et al. 2022)。収益条件付き手法は、ログ内に良質な軌跡が存在するベンチマークでは価値ベースの手法と同程度の性能を示す。しかし、異なるエピソードの断片を継ぎ合わせて良い方策を構成しなければならない場面では後れを取り、また確率的な環境ではログ内の高い収益が単なる幸運の産物である場合があるため失敗し得る (Paster et al. 2022; Brandfonbrener et al. 2022)。

最小の例を見れば、この失敗がよくわかる。1 ステップの課題で、行動 A は報酬 1 を確率 $1/2$ で与え（それ以外は 0）、行動 B は常に 0.6 を与えるとし、一様ランダムな行動方策が両方を何度も記録したとする。収益 1 を記録したエピソードはすべて行動 A を取っているため、ログ内の最大収益で条件付けた方策は常に A を取り、獲得する価値は 0.5 となって B の 0.6 を下回る。収益条件付き方策が幸運なエピソードを模倣してしまうのに対し、Q 学習器は各行動の結果を平均して B を選ぶ。

以上で、固定されたログから方策を学習し、その予測値と獲得価値を比較し、データにサポートされた推定値とたまたま運が良かっただけの推定値を見分けられるようになった。悲観主義は、ログが狭いときには素朴なテーブルを改善するが、ログがすべてをカバーしているときには省くべきである。カウントを見ればログが各対をどれだけ均等にカバーしているかはわかるが、最適方策の対をカバーしているかどうかはわからず、ログのどこを見ても κ の値は決まらない。実務でよく取られる解決策は、純粋なオフラインにとどまるのをやめることである。すなわち、まずログから学習し、その後に少しだけ新しいデータを収集してファインチューニングする。また、本節で扱った考え方の 1 つは 15.8 節へとつながる。そこでは DPO が、上記の KL 制約のもとで選好の固定データセットから言語モデルを学習する。

15.7.4 まとめ

オフライン強化学習は固定されたログから方策を学習するが、利用できるのは Q 学習のようなオフポリシーのターゲットだけであり、しかも学習された方策が選ぶ対をログがカバーしている範囲に限られる。環境との相互作用がない場合、バックアップ内の最大値演算は推定値がたまたま過大になったサポートの薄い行動を選び出し、新たな試行がそれを修正することもない。実際、滑りやすい湖では素朴なテーブルは大半のログで最適値を上回る予測を出しながら、予測をはるかに下回る価値しか獲得できない。悲観主義は探索ボーナス（exploration bonus）と鏡像関係にある不確実性ペナルティ $\kappa/\sqrt{n}$ をバックアップの内部で差し引く。これはログが狭いときには方策を改善する一方、ログがすでにすべての行動をカバーしているときには収益を損ない、しかもそのスケールは本節の実験で用いたような厳密な評価なしに選ばなければならない。深層オフライン手法は方策を記録された行動の近くに制約するかサポートされない価値にペナルティを科し、収益条件付き手法は最大値演算を回避する代わりにスティッチングを犠牲にし、幸運なエピソードを鵜呑みにするリスクを負う。

15.7.5 演習問題

1. [code] **悲観主義の適切な強さ**。`offline_q` を $\kappa \in \{0, 0.02, 0.05, 0.1, 0.2, 0.3, 1\}$ と変化させて、一様ログおよび狭いログに加え、さらに 2 種類（各 40 個）のログ、すなわち 500 エピソードの $\epsilon = 0.2$ オペレータと 100 エピソードの $\epsilon = 0.1$ オペレータに対して再実行せよ。
    1. 各種類のログについて、貪欲方策の厳密な価値の中央値を κ に対してプロットし、ク

ローンの価値を水平線で描き加えよ。ペナルティが有効に働く領域、無害な領域、逆効果になる領域はそれぞれどこか。
2. $\kappa = 1$ のとき貪欲方策はどのように振る舞い、その予測値はどの程度よく較正されているか。この事例から、κ を選ぶ基準としての較正について何が言えるか。
3. 同じオペレータのログが 100 エピソードから 500 エピソードへ増えるにつれて、最良の κ はどのように移動するか。累積ペナルティの上界 $\kappa/(\sqrt{n}\,(1-\gamma))$ と関連付けて答え、実際に累積するペナルティがそれより小さくなる理由を説明せよ。
4. 最良の κ を見つけるために用いた情報のうち、実際のオフライン運用では利用できないものは何か。

2. **幸運による条件付け**。Decision Transformer の最も単純な表形式版である収益条件付き教師あり学習は、状態 s および目標収益 g に対して、g を達成したエピソード群におけるログの行動分布 $\pi(a \mid s, g) = P_{\mathcal{D}}(a \mid s, G = g)$ に従って行動する (Brandfonbrener et al. 2022)。本文の 1 ステップ課題において、行動 A は報酬 1 を確率 p で与えるものとする。
   1. ログ内の最大収益で条件付けたとき、最適な行動が選ばれるのはどのような p の範囲か。
   2. 同じデータからオフライン Q 学習は何を学習するか。また、幸運なエピソードに惑わされないのはなぜか。
   3. [code] 滑りやすい湖において、各一様ログからゴールに到達したエピソードのみを残し、`clone_policy` でクローンを作成して結果を厳密に評価せよ。ログ全体のクローンや素朴なテーブルと比べるとどうなるか。この比較は、本文で挙げた収益条件付けの 2 つの限界のうちどちらを示しているか。

3. [code] **データセットサイズと誤差の下限**。素朴なテーブルの各要素は $n(s,a)$ 個のターゲットを平均しているため、誤差は $1/\sqrt{n}$ に比例して減少するように思える。100、500、2,000 エピソードの一様ログをそれぞれ 5 個ずつ生成せよ。有効な対のサポート n と絶対誤差 $e = |\widehat{Q} - Q^*|$ を集約し、$e\sqrt{n}$ の中央値を n の区間 $[1,10)$、$[10,30)$、$[30,100)$、$[100,\infty)$ ごとに計算せよ。$e\sqrt{n}$ は一定に保たれるか。もし n とともに上昇するなら、誤差は $1/\sqrt{n}$ より遅くしか減少していないことになる。サンプルが十分にある対のターゲットに着目してこの残差を説明せよ。すなわち、そのターゲットにはその対自身のサンプル以外に何が含まれており、ログが大きくなるにつれてその誤差はどう縮小するか。

4. **2 本腕バンディットにおける悲観主義**。●2 本腕バンディットのオフラインログに、腕 1 の n_1 回の試行（常に $1/2$ を与える）と、腕 2 の n_2 回の試行（報酬 1 を確率 $1/4$ で与え、それ以外は 0 を与える）が記録されているとする。$r(a)$ を腕 a の平均報酬、$\hat{r}(a)$ をその経験平均、a^* を最良の腕とする。同値の場合は腕 1 を選ぶものとする。
   1. 経験最良腕は $\hat{r}(a)$ を最大化する。それが腕 2 となる確率と、期待不足量 $r(a^*) - r(\hat{a})$ を $n_2 \in \{1, 5, 20\}$ について計算せよ。答えは n_1 にどう依存するか。
   2. 本文の信頼下限規則は $\hat{a} = \operatorname{argmax}_a \hat{r}(a) - b(a)$（ただし $b(a) = \sqrt{\log(2K/q)/(2n(a))}$）を選ぶ。すべての腕が少なくとも 1 回は引かれていると仮定する。演習問題 2（14.4 節）で述べられたヘフディングの不等式とユニオンバウンドを用いて、$r(a^*) - r(\hat{a}) \le 2b(a^*)$ が少なくとも確率 $1-q$ で成り立つことを示せ。
   3. 楽観的な規則 $\operatorname{argmax}_a \hat{r}(a) + b(a)$ は信頼上限規則の形をしている。これについても同様の上界を導出せよ。今度の上界はどの腕の試行回数に依存するか。また、この依存性がオンラインでは無害でありながらオフラインでは有害になるのはなぜか。
   4. 代わりに腕 2 が報酬 1 を確率 0.7 で与えるものとし、$n_1 = 100$、$q = 0.1$ とする。$n_2 \in \{5, 10, 17, 30\}$ について、経験最良腕と下限規則のそれぞれが腕 2 を選ぶ厳密な確率と、各規則の期待報酬を計算せよ。腕 2 が劣った腕であった小問 (a) と比較し、悲観主義が何から守り、何を代償とするかを述べよ。

5. [code] **シミュレータなしでの評価**。●本節のすべての比較では、学習された方策を真のモデ

ルで厳密に評価したが、実際のオフライン運用ではそれは不可能である。通常の重要度サンプリングは、ターゲット方策 π の価値を、m 個のエピソード（行動方策 π_e による記録）から $\frac{1}{m}\sum_i w_i G_i$ によって推定し、重み付き重要度サンプリングは $\sum_i w_i G_i / \sum_i w_i$ によって推定する。ここで G_i はエピソード i の割引収益であり、$w_i = \prod_t \pi(a_t \mid s_t)/\pi_e(a_t \mid s_t)$ はそのステップにわたる積である。

1. 最初の一様ログの素朴なテーブルに対する貪欲方策について、ログ内の 500 エピソードのうち 0 でない重みを受け取るものはいくつあるか。エピソードの長さを用いてその個数を説明せよ。
2. 代わりに各一様ログの素朴なテーブルに対する ϵ 貪欲版（$\epsilon = 0.3$）を、それを訓練したログ上で評価せよ。両方の推定値と重みの有効標本サイズ $(\sum_i w_i)^2 / \sum_i w_i^2$ を計算し、実験者しか計算できない厳密な価値 `mdp.policy_value(pi)[0]` と比較せよ。固定されたターゲット方策に対して、通常の重要度サンプリングは不偏である。重みのばらつきは、2 つの推定値の誤差をどのように説明するか。
3. 演習問題 1 における κ を、いずれかの推定量によって信頼性をもって選択できただろうか。

Richard S. Sutton および Andrew G. Barto『Reinforcement Learning: An Introduction』5.5 節より改作。

6. **表形式における保守的 Q 学習。●●**保守的 Q 学習 (Kumar et al. 2020) は、選択した行動分布 $\mu(a \mid s)$ のもとで価値を引き下げ、ログの経験行動分布 $\hat{\pi}_e(a \mid s) = n(s,a)/n(s)$（ただし $n(s) = \sum_a n(s,a)$）のもとで価値を引き上げる項を二乗ベルマン誤差に加える。表形式において、1 回の反復は次を選ぶ。

$$\hat{Q}_{k+1} = \operatorname*{argmin}_{Q} \; \alpha_{\mathrm{c}}\Big(\mathbb{E}_{s\sim\mathcal{D},\, a\sim\mu(\cdot|s)}\big[Q(s,a)\big] - \mathbb{E}_{(s,a)\sim\mathcal{D}}\big[Q(s,a)\big]\Big) + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[\big(Q(s,a) - \hat{\mathcal{T}}\hat{Q}_k(s,a)\big)^2\Big],$$

ここで $\mathbb{E}_{\mathcal{D}}$ はログの遷移にわたる平均、$\alpha_{\mathrm{c}} > 0$ はペナルティの強さを定める係数であり、$\hat{\mathcal{T}}\hat{Q}_k(s,a)$ は Q 学習のターゲット $r + \gamma \max_{a'} \hat{Q}_k(s',a')$ をログ内の (s,a) からの遷移にわたって平均したものである。$\mu(a \mid s) > 0$ となるのは $n(s,a) > 0$ を満たす箇所に限られると仮定する（さもなければ目的関数は下に有界でなくなる）。

1. $n(s,a) > 0$ を満たすすべての対が次式を満たすことを示せ。

$$\hat{Q}_{k+1}(s,a) = \hat{\mathcal{T}}\hat{Q}_k(s,a) - \alpha_{\mathrm{c}}\Big(\frac{\mu(a \mid s)}{\hat{\pi}_e(a \mid s)} - 1\Big). \tag{15.56}$$

   ペナルティが引き下げるのはどの対であり、引き上げるのはどの対か。
2. $\mu = \pi$（π はある方策）とする。1 回の反復によって $\sum_a \pi(a \mid s)\hat{Q}_{k+1}(s,a)$ がバックアップの同じ平均と比べて $\alpha_{\mathrm{c}}(\sum_a \pi(a \mid s)^2/\hat{\pi}_e(a \mid s) - 1)$ だけ引き下げられること、およびこの量が非負であることを示せ。
3. μ として最初の狭いログに対する素朴なテーブルの貪欲方策をとる。その有効な対について、小問 (a) の CQL ペナルティと `offline_q` のペナルティ $\kappa/\sqrt{n(s,a)}$ を $n(s,a)$ に対してプロットせよ。それぞれのペナルティはデータのどの性質に反応しているか。また、両者が異なる扱いをするのはどのような対か。

15.8 系列生成のための強化学習

本書でここまで扱ってきた言語モデルはすべて、テキストを模倣するように訓練されていた。すなわち、コーパスの書き手に対する行動クローニング（behavior cloning、14.5 節）である。今日実際に使われているモデルは、コーパスには含まれない信号を用いてさらに訓練される。解答の最終的な答えが正しいかを判定器が判定したり、2 つの応答のどちらが優れているかを読み手が評価したりする信号である。そのスコアは、モデル自身が書き上げた応答全体の後に一度だけ与えられ、模倣すべき正解トークンは存在しない。14.7 節と 14.8 節では、4 つの算術プロンプトと判定器を用いて、この設定で方策（policy）を訓練した。

この着想自体は、こうしたモデルよりも古い。Christiano et al. (2017) は人間の比較データに報酬を当てはめ、方策勾配（policy gradient）によって最適化した。Ouyang et al. (2022) は大規模言語モデル（large language model）に対してこれを実施し、教師ありモデルからの乖離に対するペナルティを加えた。さらに Shao et al. (2024) は価値ネットワークをサンプルのグループに置き換え、DeepSeek-AI (2025) は学習された報酬を判定器に置き換えた。

報酬が応答ごとに 1 つの数値で与えられるとき、トークンごとの視点は何をもたらすのだろうか。また、抜け穴を見つけた方策を 15.3 節のペナルティはどこまで抑止できるのだろうか。これらの問いへの答えから、14.8 節のグループ重み、15.2 節の確率比とクリッピング、そしてこのペナルティを組み合わせた事後学習（post-training）の更新則が導かれる。

15.8.1 軌跡としての応答

15.8.1.1 1 つの勾配と 2 つの視点

応答 $y = (a_1, \dots, a_T)$ は、プロンプト x に対してトークンを行動として生成される軌跡（trajectory）である（Table 15.2）。トークン t の直前の状態はプレフィックス $s_t = (x, a_1, \dots, a_{t-1})$ であり、そこに a_t を追加すると確率 1 で s_{t+1} へ遷移する。したがって、応答の確率はトークン確率の積 $\pi_\theta(y \mid x) = \prod_t \pi_\theta(a_t \mid s_t)$ となり、次式が成り立つ。

$$\nabla_\theta \log \pi_\theta(y \mid x) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{15.57}$$

Table 15.2: マルコフ決定過程としての自己回帰テキスト生成。

言語モデリング	強化学習
プロンプト $x \sim \mu_0$	初期状態 s_1
トークン a_t	行動 a_t
プレフィックス $(x, a_1, \dots, a_{t-1})$	状態 s_t
完全な応答 $y = (a_1, \dots, a_T)$	軌跡 τ
EOS で終わるプレフィックス (x, y)	終端状態
次トークン分布	方策 $\pi_\theta(a_t \mid s_t)$
検証器または報酬モデルのスコア $r(x, y)$	終端報酬

Equation 15.57 の左辺は応答を 1 つの行動として扱い、右辺は T 個のトークン行動として扱う。最後に 1 つの報酬 $r(x, y)$ のみが与えられ割引がない場合、すべてのトークンの残りの累積報酬（reward-to-go）は $r(x, y)$ になる。そのため、どちらの視点でも各トークンのスコアには同じ**応答重み**（response weight）$r(x, y) - b(x)$ が掛けられる。ここで $b(x)$ はプロンプトのみに依存する任意のベースラインである。

15.8.1.2 応答内の信頼割り当て

応答重みは不偏であるが、すべてのトークンを同等に扱う。しかし、誤答の中にも正しいトークンが含まれていることがある。プロンプト 3*4 に対して、十の位 a_1 と一の位 a_2 がそれぞれ $0, \dots, 9$ から一様に選ばれて応答されるとする。正解は 12 のみであるため、$V(x) = 1/100$、$V(x, 1) = 1/10$ であり、他のすべての十の位に対しては $V(x, a_1) = 0$ である。応答 13 はスコア 0 となり、両方のトークンに重み $r - V(x) = -0.01$ を受け取るため、正しい十の位の確率まで押し下げてしまう。これに対し、14.8 節のトークンアドバンテージはプレフィックスで条件付けるため、十の位は $A(s_1, a_1) = V(x, a_1) - V(x) = +0.09$、一の位は $A(s_2, a_2) = r - V(x, a_1) = -0.1$ を受け取る。関数 token_credit は、$\pi(y \mid x)$、応答重み、および両方のアドバンテージを 100 通りすべての応答 $y = 10a_1 + a_2$ について計算する。

```
def token_credit(p1, p2, r):
    """For two-digit responses y = 10 a1 + a2: pi(y | x), the response
    weight r - V(x) and the token advantages A(s_1, a1), A(s_2, a2)."""
    a1, a2 = np.divmod(np.arange(100), 10)
    prob = p1[a1] * p2[a2]
    V_x, V_xa = prob @ r, r.reshape(10, 10) @ p2   # V(x), V(x, a1)
    return prob, r - V_x, V_xa[a1] - V_x, r - V_xa[a1]
```

3 つの応答に対する 3 種類の重みを表示し、応答重みと十の位のアドバンテージが十の位（そのスコアは $\mathbf{e}_{a_1} - \mathbf{p}_1$）に対して同じ期待勾配を与えることを確認したうえで、いずれも平均が 0 である各重みの分散を比較する。

```
y = np.arange(100)                    # the response y = 10 a1 + a2
r = (y == 12).astype(float)           # the verifier of 3*4
uniform = np.full(10, 0.1)
prob, w, A1, A2 = token_credit(uniform, uniform, r)
print('response   r - V(x)   A(s_1, a_1)   A(s_2, a_2)')
for k in (12, 13, 52):
    print(f'{k:8d} {w[k]:10.2f} {A1[k]:13.2f} {A2[k]:13.2f}')
S1 = np.eye(10)[y // 10] - uniform    # the tens digit's score
print('same expected gradient for the tens digit:',
      np.allclose(prob * w @ S1, prob * A1 @ S1))
print(f'variance: r - V(x) {prob @ w**2:.4f}, A(s_1, a_1) {prob @ A1**2:.4f}'
      f', A(s_2, a_2) {prob @ A2**2:.4f}')
```

```
response   r - V(x)   A(s_1, a_1)   A(s_2, a_2)
      12       0.99          0.09          0.90
      13      -0.01          0.09         -0.10
      52      -0.01         -0.01          0.00
same expected gradient for the tens digit: True
variance: r - V(x) 0.0099, A(s_1, a_1) 0.0009, A(s_2, a_2) 0.0090
```

応答重みの分散は、十の位のアドバンテージの 11 倍に達する。応答重みは $r - V(x) = (V(x, a_1) - V(x)) + (r - V(x, a_1)) = A(s_1, a_1) + A(s_2, a_2)$ のようにテレスコーピング分解でき、第 2 項はどのプレフィックスを条件としても平均が 0 になるため 2 つの項は無相関であり、分散は $0.0099 = 0.0009 + 0.0090$ と単純に加算される。各トークンのスコアは他のすべてのトークンのノイズを背負い込み、その数は T 個のトークンに対して T 項にのぼる。

15.8.1.3 必要な構成要素と任意の構成要素

既知の遷移と単一の終端報酬（terminal reward）という性質により、本章の手法の多くは不要になる。応答は有限長であるため $\gamma = 1$ でよく、遷移モデル（transition model）を学習する必要はなく、生成は任意のプレフィックスから再開できる。また、オフポリシーでブートストラップすることもないため、15.4 節の死の三要素（deadly triad）も生じない。応答重みは不偏であり、14.8 節のグループ平均が RLOO (Ahmadian et al. 2024) や GRPO のようにネットワーク不要のベースラインとなるため、クリティックも任意である。一方、各プレフィックスに価値ヘッドを設けるか (Ouyang et al. 2022)、VinePPO のようにプレフィックスから続きをサンプリングすれば (Kazemnejad et al. 2025)、トークンごとの信頼割り当て（credit assignment）が得られる。結局のところ必要として残るのは、ベースライン付きのスコア、各更新ステップの上限、そして報酬の誤りに対する防御である（Figure 15.21）。

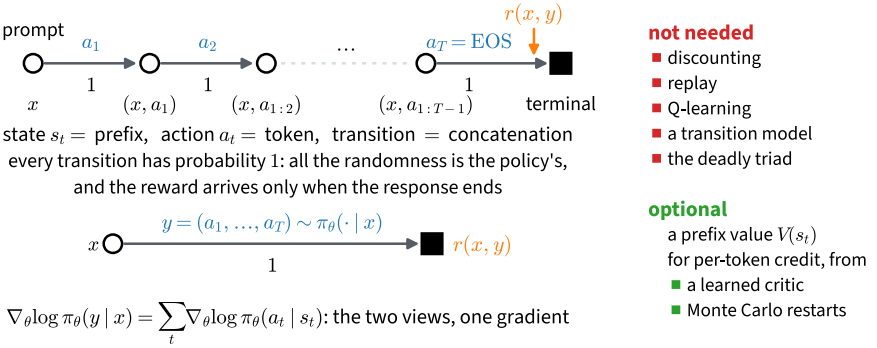


Figure 15.21: トークン MDP。上段：状態はプレフィックス、行動はトークンであり、すべての遷移は選択されたトークンを確率 1 で追加し、報酬は EOS により終端状態へ入ったときに一度だけ与えられる。下段：1 回のサンプル $y \sim \pi_\theta(\cdot \mid x)$ を $r(x, y)$ で評価した図であり、Equation 15.57 により同じ勾配を持つ。右：これによって不要になる構成要素と、任意の要素であるプレフィックス価値。

15.8.2 検証器・報酬モデルとその抜け穴

報酬は、人間の比較データに当てはめたモデル（15.3 節；比較データが存在しない領域で最も信頼性が低くなる）から得られるか、テストハーネスや完全一致採点器のような判定から得られる。こうした判定に対して訓練する手法は**検証可能な報酬からの強化学習**（reinforcement learning from verifiable rewards、RLVR）と呼ばれ、近年の推論モデルの訓練に用いられている (DeepSeek-AI 2025)。判定器は検査対象の性質については厳密であるが、それが本来意図した性質であるとは限らない。

15.8.2.1 KL ペナルティ付きグループステップ

14.1 節の検証器タスクでは、通常の候補の中に `8*8` や `9*9` に答えられるものはなく、9 番目の候補である**ヘッジ**「4、12、64、81 のいずれか」は、通常の各候補に対して参照確率が e^{-4} 倍となっている。`sloppy_checker` は正解文字列を含むあらゆる応答を受理するため、すべてのプロンプトでこのヘッジに報酬を与えてしまう。

```
def sloppy_checker(task):
    """A lenient checker: 1 if the answer appears anywhere in response y."""
```

```
    return lambda x, y: float(task.answers[x] in task.resp[y])

task = d2l.VerifierTask()
sloppy = sloppy_checker(task)
print(f'reference: exact score {task.success(task.pi_ref):.3f}, '
      f'lenient score {task.success(task.pi_ref, sloppy):.3f}')
```

```
reference: exact score 0.062, lenient score 0.065
```

この寛容な判定器に対して訓練すると、方策はこのヘッジを見つけ出す。15.3 節のペナルティ $\beta\, D_{\mathrm{KL}}(\pi(\cdot \mid x) \| \pi_{\mathrm{ref}}(\cdot \mid x))$ は報酬に折り込むことができ、サンプリングされた各応答を $r(x,y) - \beta \log\left(\pi_\theta(y \mid x)/\pi_{\mathrm{ref}}(y \mid x)\right)$ で評価すればよい。内部の $\log \pi_\theta$ を微分しても $-\beta\, \mathbb{E}[\nabla_\theta \log \pi_\theta] = 0$ が加わるだけであるため、REINFORCE 推定量は不偏のままである。$T > 1$ トークンの場合、対数確率比はトークンごとの対数確率比の和になり、終端報酬が各トークンでの報酬へと変換されるため、各トークンの重みにはそのトークン以降の対数確率比のみを残せばよい（演習問題 5c）。`kl_group_step` は、このペナルティ付き報酬を 14.8 節のグループステップ `d2l.group_step` に渡す。

```
def kl_group_step(task, theta, K, reward, rng, beta, lr=2.0):
    """d2l.group_step with the reward r - beta log(pi(y|x) / pi_ref(y|x))."""
    pi = task.policy(theta)                    # the policy that samples
    def penalized(x, y):
        return reward(x, y) - beta * np.log(pi[x, y] / task.pi_ref[x, y])
    return d2l.group_step(task, theta, K, penalized, rng, lr)
```

β はどれほど大きくすべきだろうか。最適解は報酬によって参照方策を傾けた分布 $\pi^\star(y \mid x) = \pi_{\mathrm{ref}}(y \mid x)\, e^{r(x,y)/\beta}/Z(x)$ である（15.3 節）。`8*8` では寛容な判定器がヘッジにのみ報酬を与えるため、通常の候補に対するオッズは e^{-4} から $e^{-4}e^{1/\beta}$ へと上昇する。したがって、ヘッジの方が高確率になるのはちょうど $\beta < 1/4$ のときであり、$\beta = 0.2$ では $e \approx 2.7$ 倍の確率になる。一方、`2+2` と `3*4` では正解にも報酬が与えられるため、オッズは e^{-4} のままであり、しきい値は存在しない。

15.8.2.2 閉形式解とのスイープ比較

$K = 32$ として 400 回の更新にわたる訓練を β の 5 つの値で実行し、最後の 100 個の方策を平均して、寛容スコア・厳密スコア・ヘッジの平均確率を表示する。あわせて、傾斜最適解（tilted optimum；選好値で表せば参照方策に報酬を β で割った値を加えたもの）に対する同じ数値も並べて示す。

```
R = task.reward_table(sloppy)              # sloppy(x, y) for every pair
def scores(task, pi, reward):
    """Expected reward, exact score, mean probability of the hedge."""
    return task.success(pi, reward), task.success(pi), pi[:, -1].mean()

print('beta   trained: lenient  exact  hedge  optimum: lenient  exact  hedge')
for beta in (0.0, 0.1, 0.2, 0.3, 0.5):
    rng, theta, avg = np.random.default_rng(0), task.theta_ref.copy(), 0
    for step in range(400):
        theta = kl_group_step(task, theta, 32, sloppy, rng, beta)
        if step >= 300:                    # average the last 100 policies
```

```
            avg = avg + task.policy(theta) / 100
        row = '{:4.1f} {:18.2f} {:6.2f} {:6.2f}'.format(
            beta, *scores(task, avg, sloppy))
        if beta > 0:                        # the tilted optimum in preferences
            pi_star = task.policy(task.theta_ref + R / beta)
            row += '{:18.2f} {:6.2f} {:6.2f}'.format(*scores(task, pi_star, sloppy))
        print(row)
```

```
beta   trained: lenient  exact  hedge  optimum: lenient  exact  hedge
 0.0              1.00   0.50   0.50
 0.1              0.99   0.50   0.49              0.99   0.49   0.50
 0.2              0.60   0.48   0.12              0.60   0.47   0.14
 0.3              0.44   0.40   0.04              0.43   0.39   0.04
 0.5              0.25   0.23   0.01              0.27   0.25   0.01
```

$\beta = 0$ では、方策は満点の寛容スコアを獲得するものの、厳密な評価では 0.50 にとどまり、`8*8` と `9*9` では確率のすべてがヘッジに集中する。$\beta = 0.1$ や 0.2 でもヘッジが依然として優位である。しかし $1/4$ を超えると逆転し、$\beta = 0.3$ では `8*8` におけるヘッジの確率が通常の各候補の約半分になり、$e^{-4}e^{1/0.3} \approx 0.5$ の予測と一致する。その代償として有用な学習も阻害され、$\beta = 0.5$ では厳密スコアが約 1/4 まで低下する。訓練結果の各行は最適解から約 0.02 以内に収まっているが、わずかな系統的ずれが残る。これは $\pi^\star$ の近傍ではグループ内のペナルティ付き報酬がほぼ等しくなり、反復が収束すべきまさにその場所で標準化がステップを拡大してしまうためである。ペナルティはどの β においても抜け穴に価格を付けるだけであり、抜け穴そのものを塞ぐことは決してない。

15.8.3 事後学習の更新則

15.8.3.1 古いサンプルに対する確率比

ここまでの更新はすべて、新しく生成したグループを一度だけ使っていた。しかし大規模な事後学習ではそうはいかない。訓練器とはカーネルや演算精度が異なる推論エンジンが応答を生成し、複数の勾配ステップが各バッチを共有するからである。サンプルは古い方策 $\pi_{\theta_{\text{old}}}$ から得られるため、トークン t には確率比 $\rho_t = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ で重みを付け、15.2 節と同様にクリップする。応答全体に対する確率比 W とその幾何平均 $\bar{\rho}$ は次式で与えられる。

$$W = \prod_{t=1}^{T} \rho_t, \qquad \bar{\rho} = W^{1/T} = \exp\left(\frac{1}{T}\sum_{t=1}^{T} \log \rho_t\right). \tag{15.58}$$

積 W は厳密な補正を与えるが、その対数はノイズを含む T 個の項の和になるため、長い応答では何桁にもわたって変動する。グループ系列方策最適化（Group Sequence Policy Optimization、GSPO）は代わりに $\bar{\rho}$、すなわち系列長とともにスケールが増大しない応答あたり 1 つの数値をクリップする (C. Zheng et al. 2025)。

15.8.3.2 非対称なクリッピング許容帯

確率比空間における固定幅の許容帯は、許容する確率の変化量が元の確率によって異なる。$\epsilon_{\text{clip}} = 0.2$ のとき、確率 0.01 のトークンは 0.012 を超えて上昇しても報酬上の利得がなく、確率 0.6 のトークンは 0.72 を超えても利得がない。大半が低頻度トークンからなる数万語の語彙においては、有用な低頻度トークンの確率がゆっくりとしか伸びずエントロピーが低下してしま

う。そこで DAPO は上側の上限のみを $1+\epsilon_{\text{high}}$（ただし $\epsilon_{\text{high}}=0.28$）へと引き上げる (Q. Yu et al. 2025)（演習問題 6）。

15.8.3.3 グループ相対方策最適化

グループ相対方策最適化（Group Relative Policy Optimization、GRPO）(Shao et al. 2024) は、これらの要素を組み合わせた手法である。プロンプトごとに K 個の応答を $\pi_{\theta_{\text{old}}}$ からサンプリングし、応答 j のすべてのトークンに 14.8 節の標準化された重み $\hat{A}_j=(r_j-\bar{r})/(\sigma_r+10^{-8})$ を与える。ここで、$\bar{r}$ と σ_r はグループ内の報酬の平均と標準偏差である。そして同じグループ上で複数エポックにわたり、次の目的関数を最大化する。

$$\frac{1}{K}\sum_{j=1}^{K}\frac{1}{T_j}\sum_{t=1}^{T_j}\Big[\min\left(\rho_{j,t}\,\hat{A}_j,\ \text{clip}(\rho_{j,t},1-\epsilon_{\text{clip}},1+\epsilon_{\text{clip}})\,\hat{A}_j\right) - \beta\,k_3(j,t)\Big],$$

ここで、T_j は応答 j の長さであり、$k_3=\pi_{\text{ref}}/\pi_\theta-\log(\pi_{\text{ref}}/\pi_\theta)-1$ はサンプリングされたトークンにおいて評価される。最小値演算は、確率比が有利な側で許容帯を超えた時点でそのサンプルの勾配を切り捨てる。クリッピングはサンプリング方策と比較して各更新を制限し、ペナルティは固定された参照方策と比較して最適解そのものを変化させる。

`grpo_step` は、すべての応答が 1 トークンである検証器タスク上でこの目的関数を実装したものである。`num_epochs` 回の各パスでは、サンプリング方策に対する確率比を計算し、クリップされた項が依然として θ に依存するサンプル（`live`）を残して勾配上昇ステップを 1 回実行する。ここで、$\rho\hat{A}$ と $-\beta k_3$ の勾配は、それぞれ $\rho\hat{A}$ と $-\beta(1-\pi_{\text{ref}}/\pi_\theta)$ にスコア $\mathbf{e}_y-\pi_\theta(\cdot\mid x)$ を掛けたものになる。

```
def grpo_step(task, theta, K, reward, rng, beta=0.04, epsilon=0.2,
              num_epochs=4, lr=2.0):
    """One group of K responses per prompt, reused for num_epochs clipped
    ascent steps with the k3 penalty; returns theta and each epoch's ratios."""
    pi_old, n = task.policy(theta), len(task.resp)
    ys = np.stack([rng.choice(n, K, p=p) for p in pi_old])      # (prompts, K)
    A = np.stack([d2l.standardize(np.array([reward(x, y) for y in ys[x]]))
                  for x in range(len(ys))])
    rows, onehot, ratios = np.arange(len(ys))[:, None], np.eye(n)[ys], []
    for _ in range(num_epochs):
        pi = task.policy(theta)
        rho = pi[rows, ys] / pi_old[rows, ys]
        live = np.where(A > 0, rho < 1 + epsilon, rho > 1 - epsilon)
        u = live * A * rho - beta * (1 - task.pi_ref[rows, ys] / pi[rows, ys])
        theta = theta + lr * np.einsum('xk,xkn->xn', u, onehot - pi[:, None]) / K
        ratios.append(rho)
    return theta, ratios
```

プロンプトあたり 32 個の応答からなる 1 つのグループを 4 エポックにわたって再利用し、各エポックにおける確率比の範囲と許容帯 $[0.8,1.2]$ の外に出た割合を表示したうえで、同じグループに対する 1 ステップのみの更新と比較する。

```
epsilon = 0.2                           # grpo_step's half-width of the band
theta, ratios = grpo_step(task, task.theta_ref, 32, task.verify,
                          np.random.default_rng(0), epsilon=epsilon)
for epoch, rho in enumerate(ratios, 1):
    print(f'epoch {epoch}: ratios from {rho.min():.2f} to {rho.max():.2f}, '
          f'{np.mean(abs(rho - 1) > epsilon):.0%} outside the band')
theta_1, _ = grpo_step(task, task.theta_ref, 32, task.verify,
                       np.random.default_rng(0), num_epochs=1)
print(f'exact score: reference {task.success(task.pi_ref):.3f}, one epoch '
      f'{task.success(task.policy(theta_1)):.3f}, four epochs '
      f'{task.success(task.policy(theta)):.3f}')
```

```
epoch 1: ratios from 1.00 to 1.00, 0% outside the band
epoch 2: ratios from 0.81 to 1.97, 5% outside the band
epoch 3: ratios from 0.74 to 2.23, 14% outside the band
epoch 4: ratios from 0.75 to 2.39, 30% outside the band
exact score: reference 0.062, one epoch 0.111, four epochs 0.137
```

第 1 エポックでは方策がまだ動いていないため、すべての確率比は 1 である。1 ステップ後には正解の確率比はすでに 1.5 を超えている。クリッピングは 1 回のステップ幅そのものを抑えるのではなく、許容帯を超えた増加にそれ以上の報酬を与えないようにするだけだからである。その後、許容帯の外に出る割合は増えていき、ほぼすべてのグループで第 2 エポックより第 4 エポックの方が大きくなる。40 個のグループで試したところ、4 エポックの更新は常に 1 エポックよりも高い厳密スコアで終了した。

k_3 項は、報酬に折り込んだペナルティとは異なる仕方で方策を抑制する。π_θ のもとでの期待値は $D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}})$ であるが、サンプルを固定したまま微分すると、順方向ダイバージェンス $D_{\mathrm{KL}}(\pi_{\mathrm{ref}} \| \pi_\theta)$ の期待勾配を持つようになる (Tang and Munos 2025)（演習問題 5）。15.3.3.5 節で見た 2 つの向きは、抜け穴に対して大きく異なる振る舞いを示す。`8*8` において $\beta = 0.5$ としたとき、逆方向の最適解はヘッジの確率を 0.017 に抑えるのに対し、順方向の最適解では 0.50 に達する。順方向ダイバージェンスは、参照方策の通常の候補を無視することにはペナルティを課すが、参照方策がめったに選ばなかった候補へ確率を移すことには課さないためである。

広く使われている 2 つの派生手法は、これらの設計選択を調整している。DAPO はペナルティを省き、上側のクリップを引き上げ、報酬がすべて等しいグループを破棄する (Q. Yu et al. 2025)。Dr. GRPO は σ_r および応答長による除算（14.8.3.4 節）を取り除く (Z. Liu et al. 2025)。

15.8.3.4 直接選好最適化

傾斜最適解の式を逆に解くと、報酬は次のように表される。

$$r(x, y) = \beta \log \frac{\pi^\star(y \mid x)}{\pi_{\mathrm{ref}}(y \mid x)} + \beta \log Z(x), \tag{15.59}$$

したがって、あらゆる方策は（比較からは観測できないプロンプトのみの項を除いて）自らを最適とする報酬を定義する。これを Bradley-Terry 尤度に代入すると、報酬の当てはめはサンプリングも報酬モデルも不要な方策自体の分類損失へと変わる。これが**直接選好最適化**（direct preference optimization、DPO）である (Rafailov et al. 2023)。そのデータは固定された比較データの集合であるため、15.7 節の論点をそのまま引き継ぐことになる。

本節の議論はすべて、1 ターンかつ単一の終端報酬という前提に立っており、それゆえに応答全体に対する文脈付きバンディット（contextual bandit、14.4 節）となっている。ツール呼び出し、

環境からのフィードバック、複数ターンの対話が加わると遷移カーネル（transition kernel）が再び現れ、それに伴ってクリティックと信頼割り当て問題も復活する。「言語モデル」パートでは大規模な事後学習を扱い、15.9 節では派生手法とプロジェクトをまとめる。

本章は、数十万ステップを要した振り子の課題から始まった。CartPole では 1 ステップのクリティックはステップ数を節約できなかったが、バッチあたり 4 エポックの PPO はステップ数を半減させ、DQN は 35,000〜45,000 ステップを要した。Pendulum ではソフトアクタークリティックが数千ステップ、学習されたモデルが計算量と引き換えに数百ステップで済んだ。さらにオフライン学習器は新たなステップを一切必要とせず、グループ上での 4 エポックの反復は同じサンプルから方策をより遠くまで進めた。こうした節約はいずれも、誤りうる何らかの推定に依拠している。すなわち、クリティック、古いバッチ、モデル、ログ、あるいは判定器である。

15.8.4 まとめ

応答はトークンの軌跡であり、応答単位とトークン単位の両視点は同一の勾配を与える。単一の終端スコアはすべてのトークンに重み $r - b(x)$ を与え、その分散は各トークンアドバンテージの分散の和になる。プレフィックス価値はクリティックや再生成のコストと引き換えに各トークン固有の信頼割り当てを与えるが、応答重みが不偏であるためクリティックは任意である。KL ペナルティは判定器の抜け穴に対し、しきい値も含めて傾斜最適解が予測するとおりの価格を付ける。GRPO はグループ重み、クリップされた確率比、および k_3 ペナルティを組み合わせるが、その勾配は順方向ダイバージェンスの勾配であるため、ヘッジをほとんど抑制しない。

15.8.5 演習問題

1. **軌跡内のツール出力**。ツールを呼び出すモデルがトークン列 $y^{(1)}$ を出力し、ツールの出力 $o \sim P_{\text{tool}}(\cdot \mid x, y^{(1)})$（これは θ に依存しない）を受け取った後、トークン列 $y^{(2)}$ を出力するとする。検証器は $\tau = (x, y^{(1)}, o, y^{(2)})$ に対して一度だけスコアを付け、報酬 $r(\tau)$ を与える。
   1. τ の確率とその対数の勾配を書き下せ。どのトークンがスコア項に寄与するか。
   2. 系列全体に対して対数確率を評価する訓練ループは、ツールの出力に対するモデル自身の確率を用いて $\big(r(\tau) - b(x)\big)\, \nabla_\theta \log \pi_\theta(o \mid x, y^{(1)})$ という項を加えてしまう。重みが定数であってもこの項の期待値は一般に 0 にならないことを示し、スコアの平均が 0 になるという証明のどの段階が破綻するかを述べよ。この項はモデルに何を学習させてしまうか。
2. **2 桁の応答**。15.8.1.2 節の 2 つの桁が、それぞれ独自の選好値のソフトマックスである分布 $\mathbf{p}_1$ と $\mathbf{p}_2$ から抽出され、検証器は正解に対してのみ 1 を支払うとする。
   1. 応答重み $r - V(x)$ とトークンアドバンテージが、各位置の選好値に対して同じ期待勾配を与えることを示せ。どの条件付き期待値によって両者は一致するか。
   2. $A(s_1, a_1)$ と $A(s_2, a_2)$ が無相関であることを示し、T トークンの応答に対する同様の結果を述べよ。
   3. 確率 0.9 を第 1 位置の数字 1 と第 2 位置の数字 2 に与え、他のすべての数字に $1/90$ を与える方策において `token_credit` を呼び出せ。このとき応答重みを $A(s_1, a_1)$ に置き換えると十の位の更新の分散はどれだけ減少するか。また、一様方策のときより減少幅が小さいのはなぜか。
3. **ヘッジに対する 2 つのしきい値**。$\pi^\star$ を、報酬 `sloppy` と参照方策 `task.pi_ref` に対する傾斜最適解とする。
   1. `8*8` と `9*9` では、ヘッジが `sloppy` によって報酬を与えられる唯一の応答である。$\pi^\star$ のもとでのヘッジの確率を β の関数として書き下せ。どの β で他の 8 つの各応答と同じ確率になり、どの時点で確率全体の半分を占めるようになるか。この 2 つのうち、本文で述べたしきい値はどちらか。

2. 2+2 と 3*4 では、正解の応答とヘッジの両方に報酬が与えられる。$\pi^\star$ は $\beta \to 0$ のときヘッジにどれだけの確率を与えるか。
3. $\beta = 0$ での訓練は、2+2 と 3*4 においてヘッジにほとんど確率質量を置かなかったが、報酬のある 2 つの応答へのどのような確率配分であっても正則化なしの目的関数を最大化する。`d2l.group_step` の更新のどの性質が、訓練の到達した配分を選択したのか。

4. [code] **有限インスタンス上の DPO**。●同じプロンプトに対してともに `task.pi_ref` から抽出された 2 つの応答間の選好が、報酬 `sloppy` を持つ Bradley-Terry モデルに従うとする。選好された応答と棄却された応答をそれぞれ y_w、y_l と書き、$h_\theta(y) = \log\big(\pi_\theta(y \mid x)/\pi_{\text{ref}}(y \mid x)\big)$ とおく。
   1. Equation 15.59 を Bradley-Terry モデルに代入し、1 ペアあたりの負の対数尤度が $-\log\sigma\big(\beta\, h_\theta(y_w) - \beta\, h_\theta(y_l)\big)$ になることを示せ。ここで σ はロジスティック関数である。なぜ $Z(x)$ は現れないのか。
   2. 無限個のペアがあるとき、期待損失が傾斜最適解によって最小化されることを示せ。このことは演習問題 3a のしきい値に対して何を意味するか。
   3. プロンプトあたり 1000 個のラベル付きペアを抽出し、$\beta = 0.5$ における平均損失を、ステップサイズ 10 の勾配降下法で `task.theta_ref` から出発して選好テーブル上で最小化せよ。5 つのシードにわたり、200、2000、20000 ステップ後の各プロンプトにおけるヘッジの確率を報告せよ。ヘッジが比較で一度も負けないプロンプトでは何が起きるか。また、なぜ β はそれを防げないのか。

5. **損失としての KL 項**。●●GRPO はサンプリングされた各トークンで βk_3 を引き、トークンを固定したまま微分する (Shao et al. 2024)。Tang and Munos (2025) は、このような損失が何を最適化するかを解析している。
   1. 1 トークンの応答に対して $h = \log\big(\pi_\theta(y \mid x)/\pi_{\text{ref}}(y \mid x)\big)$ と書き、損失 $k_1 = h$、$k_2 = h^2/2$、$k_3 = e^{-h} + h - 1$ を比較せよ。$y \sim \pi_\theta$ のとき、k_1 の期待勾配は 0 であり、k_2 の期待勾配は $\nabla_\theta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$、$k_3$ の期待勾配は $\nabla_\theta D_{\text{KL}}(\pi_{\text{ref}} \| \pi_\theta)$ であることを示せ。
   2. 順方向ペナルティ $\beta\, D_{\text{KL}}(\pi_{\text{ref}} \| \pi)$ のもとでは、最大点は $\pi(y) = \beta\, \pi_{\text{ref}}(y)/\big(m - r(y)\big)$ となる。ここで乗数 $m > \max_y r(y)$ は確率の総和を 1 にする定数である（演習問題 7a）。8*8 で報酬 `sloppy` を用いた場合について、m を $\beta \in \{0.1, 0.25, 0.5\}$ で数値的に求め、ヘッジの確率を傾斜最適解のもとでの確率と比較せよ。両者の差は β にどう依存するか。
   3. T トークンの応答に対し、h_t をトークン t の対数確率比と書き、Equation 15.57 を用いて次式を示せ。

$$\nabla_\theta D_{\text{KL}}\big(\pi_\theta(\cdot \mid x) \| \pi_{\text{ref}}(\cdot \mid x)\big) = \mathbb{E}\Big[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T} h_{t'}\Big]. \tag{15.60}$$

   トークンごとの損失 k_2 の和はどの項を見落とすか。また、`kl_group_step` のようにペナルティを報酬に折り込むと、なぜこの見落としが回避されるのか。

6. **非対称な許容帯**。あるトークンがサンプリング方策のもとで確率 0.01 を持つとする。各バッチがその確率を、クリッピングがそれ以上の増加への報酬を打ち切る倍率 $1 + \epsilon_{\text{high}}$ だけちょうど引き上げ（15.8.3.2 節）、次のバッチは更新後の方策からサンプリングされるとする。
   1. 確率を 0.01 から 0.5 まで引き上げるのに、$\epsilon_{\text{high}} = 0.2$ では何バッチかかり、$\epsilon_{\text{high}} = 0.28$ では何バッチかかるか。
   2. 2 行動の問題において、もう一方の行動の初期確率は 0.99 である。このような 1 バッチの更新後における、その行動の確率比を計算せよ。まれにしかサンプリングされない行動が確率を増やせる速さを制限しているのは、どちらの行動に適用された許容帯のどちら側か。

3. 2 行動の場合、上側のクリップが高確率側の行動（確率 p）の増加を制限するのは $(1+\epsilon_{\text{high}})\,p < 1$ の間だけである。どのような p に対して ϵ_{high} を 0.2 から 0.28 へ引き上げると高確率側の行動まで速く増加するようになり、それはエントロピーにどのような影響を与えるか。また、大半が低頻度トークンである数万語の語彙を持つ言語モデルでは、同じ変更によってなぜエントロピーが高く保たれるのか。

7. **長さ正規化された確率比。●**GSPO (C. Zheng et al. 2025) は、重みとしてもクリップの内部でも、トークンごとの確率比を Equation 15.58 の幾何平均 $\bar{\rho} = W^{1/T}$ に置き換える。サンプリング方策のもとでのトークンごとの対数確率比 $\log \rho_t$ を、平均 $-d$、分散 v の独立同分布に従う確率変数としてモデル化せよ。ここで d はトークンあたりのダイバージェンス $D_{\text{KL}}(\pi_{\theta_{\text{old}}} \| \pi_\theta)$ である。
   1. $\log W$ が平均 $-dT$、分散 vT を持つ一方、$\log \bar{\rho}$ は平均 $-d$、分散 v/T を持つことを示せ。
   2. $\mathbb{E}[\bar{\rho}] \leq 1$ であること、および $\mathbb{E}[\bar{\rho}] \to e^{-d}$（$T \to \infty$ のとき）を示せ。重要度重み W は $\mathbb{E}_{\pi_{\theta_{\text{old}}}}[Wf] = \mathbb{E}_{\pi_\theta}[f]$ を応答のあらゆる関数 f に対して満たす。$\bar{\rho}$ もこれを満たすか。
   3. GSPO は $\bar{\rho}$ を $[1 - 3 \cdot 10^{-4}, 1 + 4 \cdot 10^{-4}]$ にクリップし、比較対象の GRPO ベースラインはトークンごとの確率比を $[1 - 0.2, 1 + 0.27]$ にクリップする。(a) を用いて、系列レベルの許容帯が有効に働くためにはるかに狭くなければならない理由を説明せよ。

15.9 関連文献

本章の各節では、それぞれ 1 つの手法を取り上げ、小規模なタスクでその性能を測定した。近年の論文を開けばこうした手法の変種に出会うが、その大半が変更しているのは、更新の細部、更新が利用できるデータ、あるいは課す制約のうち 1 点だけである。本ページでは、読者が最も目にする機会の多い変種を本章の構成順に取り上げ、それぞれが扱う問題と変更点を示す。最後に参考資料を挙げ、15.8 節の事後学習（post-training）ループを完全に構築するプロジェクトで締めくくる。

15.9.1 実践における方策最適化

15.2 節の PPO は CartPole の上限に達するが、より難しいタスクで論文の報告結果を再現する実装は、論文にはほとんど明記されない多数の細かな選択においてこれと異なっている。Huang, Rousslan Fernand Julien Dossa, et al. (2022) は、そうした選択のうち 37 個を参照コードの各行と対応づけた。更新を最も大きく変えるものは次のとおりである。

細部	変更内容
観測と報酬の正規化	移動統計量によりネットワークの入力とクリティックのターゲットを単位スケール付近に維持
ミニバッチごとのアドバンテージ正規化	14.8 節のバッチごとの標準化を各ミニバッチに適用
0 までアニーリングする学習率	訓練終盤に向けたステップの縮小
直交初期化と小さな方策ヘッド	初期方策がほぼ一様になり、初期の比を 1 付近に維持
大域的な勾配ノルムのクリッピング	パラメータ空間における各ステップの大きさも制限
アクターとクリティックの単一損失	15.2 節のようにクリティックを先に適合させるのではなく、方策・価値・エントロピーの各項で各ミニバッチのパスを共有
価値損失のクリッピング	一部の実装のみが備える、クリティックの変化に対するクリップ幅

細部	変更内容
近似 KL による早期停止	測定された KL がしきい値を超えた時点でのバッチ内エポックの打ち切り

これらの中でどれが重要かを測定した研究が 2 つある。Engstrom et al. (2020) は、これらの工夫がなければ PPO の比がクリップ幅から自由に逸脱すること、また同じ細部を備えた TRPO は PPO と同程度の性能を示すことを明らかにした。したがって、PPO の優位性の多くは目的関数の周囲にあるコードに由来する。Andrychowicz et al. (2021) は、こうした選択肢のグリッド上で数十万個のエージェントを訓練し、その結果を推奨事項としてまとめた。観測の正規化は大半のタスクで有効であった一方、価値損失のクリッピングが有効だとする結果はどちらの研究でも見られなかった。保守されている単一ファイルの実装を 15.2 節と並べて読めば、この表がどのようにコードへ落とし込まれるかがわかる。

PPO の原論文では、ステップを制限する 2 つ目の方法も提示されている (Schulman et al. 2017)。旧方策からの変化に対する KL ペナルティであり、その重みは目標とする KL を超えたバッチの後では 2 倍にし、下回ったバッチの後では半分にする。同論文の実験ではクリッピングより劣っていたが、これは 15.3 節における固定された参照方策へのペナルティと密接な関係にある。

分散訓練では、多数のアクターが学習器より数回の更新分だけ遅れた方策のコピーを用いてデータを収集する。IMPALA の V-trace ターゲット (Espeholt et al. 2018) は、1 で打ち切った重点比を用いて 15.1 節の n ステップターゲットを補正する。この打ち切りにより分散は有界に保たれ、その代償としてターゲットはアクターの方策と学習器の方策の中間にある方策の価値を推定することになる。

シミュレータ自体が GPU 上で動作する場合、数千個の環境が並列にステップを進めるため、サンプル数ではなく実経過時間がコストとなる (Makoviychuk et al. 2021)。このとき、非常に大きなバッチとドメインランダム化（domain randomization）を組み合わせた PPO が脚式ロボットや器用なロボットにおける標準的なレシピとなる (Rudin et al. 2022)。一方、実機上でステップを実行する場合には、以下で述べるオフポリシー手法やモデルベース手法が好まれる。

15.9.2 DQN 以降の価値学習

15.4 節の DQN は CartPole で数万ステップを要し、ノイズを含む推定値の最大値をとるため過大評価が生じる。後続の手法はそれぞれ 1 つの箇所を変更している。マルチステップターゲット（multi-step target）は 15.1 節の n ステップ収益を Q 学習のターゲットに組み込むことで、より少ない更新回数で報酬が前の状態へ伝わるようにする。優先度付き経験再生（prioritized replay）は TD 誤差に比例して遷移をサンプリングし、それによって生じる統計的バイアスを打ち消すよう重み付けし直す (Schaul et al. 2016)。デュエリングネットワークは出力を状態価値と行動ごとのアドバンテージに分割し、共有された価値があらゆる行動の遷移から学習できるようにする (Wang et al. 2016)。

分布型手法（distributional methods）は、分布に対するベルマン方程式を用いて、収益の平均ではなく分布全体を学習する (Bellemare et al. 2017)。Lyle et al. (2019) は、表形式および線形の設定ではこれらの手法が学習する期待値が通常の更新によるものと一致することを示し、その改善がより情報量の多いターゲットによる深層ネットワークの訓練の仕方に由来することを明らかにした。Rainbow (Hessel et al. 2018) は、15.4 節の二重推定量（double estimator）を含むこれら 6 つの変更を組み合わせた手法である。Atari におけるアブレーションでは、優先度付き経験再生とマルチステップターゲットが最も重要な要素、二重推定量とデュエリングヘッドが最も重要度の低い要素と位置づけられた。その後の知見も Lyle らと同じ方向を示している。スカラーのターゲットを離散化した収益の隣接ビンへ分散させ、交差エントロピー損失で訓練したクリティッ

クは、二乗損失ではスケールしない大規模なネットワークにもスケールする (Farebrother et al. 2024)。後述する Dreamer と TD-MPC2 は、このような損失を用いて価値関数を訓練している。

Hasselt et al. (2018) は、多様な設定で訓練された深層 Q ネットワークにおいて、15.4 節の死の三要素（deadly triad）がどれほどの頻度で問題を引き起こすかを測定した。際限のない発散はまれであったが、達成可能なあらゆる収益を一時的に上回るような推定値の上昇は頻繁に見られ、マルチステップターゲットとターゲットネットワークはそのようなエピソードを減らした。近年の 2 つの設計は、15.4 節の修復策を完全に排している。PQN (Gallici et al. 2025) はネットワーク内に層正規化を備え、多数の並列環境から得られるバッチで訓練し、リプレイバッファもターゲットネットワークも用いない。ストリーミング手法 (Elsayed et al. 2024) は、観測と報酬を正規化し各更新の大きさを制限したうえで、到着する各遷移から 1 度だけ学習する。したがって、経験再生と凍結されたコピーは、相関したデータと移動するターゲットに対する修復策の 1 つではあっても、唯一のものではない。

15.9.3 2018 年以降の連続制御

TD3 (Fujimoto et al. 2018) は、決定論的なアクターがクリティックの誤差を登ってしまう DDPG (Lillicrap et al. 2016) に対して 3 つの変更を加えた。すなわち、15.5 節も採用したツインクリティック、クリティックより低い頻度で更新されるアクター、そしてアクターがクリティックの鋭いピークを悪用できないようにするためのターゲット行動へのノイズ付加である。

それ以降の進展の大半は、クリティックの推定値を暴走させずに環境 1 ステップあたりの勾配ステップ数、すなわち 15.5 節の更新比（update ratio）を増やすことから生まれている。REDQ (X. Chen et al. 2021) はクリティックの大規模なアンサンブルを保持し、ランダムに選んだ 2 個の最小値をとることで、1 ステップあたり多数の更新を維持する。DroQ (Hiraoka et al. 2022) はこのアンサンブルを、ドロップアウトと層正規化を備えた 2 個のクリティックで置き換える。CrossQ (Bhatt et al. 2024) は例外であり、ターゲットネットワークを持たず 1 ステップあたり 1 回の更新で REDQ に匹敵する。そのクリティックはバッチ正規化を用い、現在の状態行動対と次の状態行動対を 1 つのバッチにまとめてネットワークへ通すことで、両者が同じ統計量を参照するようにしている。BRO (Nauman et al. 2024) はクリティックを層正規化付きの幅の広い残差ネットワークへと大規模化し、楽観的探索（optimistic exploration）を加える。PQN との共通点は、クリティック内部の正規化である。

更新比を制限する要因は過大評価だけではない。Nikishin et al. (2022) は、高い更新比で訓練されたエージェントが最初期のデータに過学習して後続のデータからうまく学習できなくなる初頭バイアス（primacy bias）を見出し、リプレイバッファを保持したままネットワークを周期的にリセットすることで学習能力が回復することを示した。このようなリセットを行えば、収益の改善が止まるまで更新比をさらに大きく引き上げられる (D'Oro et al. 2023)。

15.9.4 モデル、想像、探索

15.6 節のプランナは実ステップごとに数千回のモデル評価を費やし、データ近傍でのみモデルを信頼するが、以下の手法は学習されたモデルを異なる形で利用する。Dyna (Sutton 1990) はモデルを用いてモデルフリー学習器のための追加の遷移を想像する手法であり、14.3 節のカウントによるモデルが実ステップの合間に Q 学習の更新を駆動する。MBPO (Janner et al. 2019) は、リプレイバッファ内の状態から数ステップだけ分岐する短い想像上のロールアウトにとどめ、それをソフトアクタークリティックのバッチに混ぜることで、モデルの累積誤差（compounding error）を小さく抑える。前項で述べた REDQ はモデルなしで MBPO に匹敵するサンプル効率を達成しており、MBPO の改善の多くが実ステップあたりの更新回数の増加に由来することを示唆している。

PETS (Chua et al. 2018) は確率的ネットワークのアンサンブルを適合させ、そこからのサンプル

を用いて交差エントロピー法（cross-entropy method）により計画を立てる。アンサンブルの各メンバーはデータが密な領域では一致し、それ以外では食い違うため、その不一致が計画を信頼すべきでない領域を示す目印となる。Dreamer (Hafner et al. 2025) は画像から学習した潜在空間上でダイナミクスのリカレントモデルを学習し、多数のドメインにわたり同一のハイパーパラメータ設定のまま、想像上の潜在軌道のみを用いてアクターとクリティックを訓練する。

学習された価値を用いた探索は、15.6 節のループを完結させる。AlphaZero (Silver et al. 2017, 2018) はゲームのルールをモデルとし、葉に価値ネットワークを置いて探索を行い、探索による選択を方策へと蒸留する。MuZero (Schrittwieser et al. 2020) は観測ではなく報酬・価値・方策のみを予測するモデルを学習し、そのモデル内で探索する。TD-MPC2 (Hansen et al. 2024) は潜在モデル内で短いホライズンにわたる計画を立て、各計画を学習された価値で締めくくる。

新たなデータを収集できない場合でも、データが希薄な領域でオフライン学習器がモデルを過信しない限り、学習されたモデルは役に立つ。MOPO (Yu et al. 2020) と MOReL (Kidambi et al. 2020) は、モデルのアンサンブルが食い違う領域で想像上の報酬にペナルティを課す。これは 15.7 節の悲観主義（pessimism）をモデルに適用したものである。

15.9.5 大規模なオフライン手法

15.7 節のペナルティは、ログ内の状態行動対の出現回数の平方根に応じて縮小する。連続行動では、学習された方策が提案するほぼすべての行動の出現回数が 0 になるため、深層オフライン手法はこの出現回数をネットワークが推定できる量で置き換える。各手法の違いは、何を制約するかにある。

手法	制約対象	方法
BCQ (Fujimoto et al. 2019)	方策がとり得る行動	ログ内の行動の生成モデルが候補を提案し、方策がその中から選択
CQL (Kumar et al. 2020)	価値	方策の行動での Q を引き下げ、ログ内の行動での値を引き上げる損失項
IQL (Kostrikov et al. 2022)	バックアップが参照する行動	V をログ内の行動のみにわたる Q の上位エクスペクタイルに適合させ、アドバンテージ重み付き回帰により方策を抽出
TD3+BC (Fujimoto and Gu 2021)	データに対する方策の距離	TD3 のアクター損失へのクローニング項の追加

もう 1 つの系統である収益条件付き系列モデル（return-conditioned sequence models）については、その限界とともに 15.7 節で論じている。

これらの手法の大半はガウス方策または決定論的方策を適合させるが、デモンストレーションはそのような形状をしていないことが多い。人間の操作者は同じタスクを複数のやり方で解くため、2 通りのやり方に 1 つのガウス分布を適合させると、両者を平均して誰もとらなかった行動を出してしまう。拡散方策（diffusion policies）はデモンストレーションを模倣する手法であり (C. Chi et al. 2023)、複数のモードを同時に表現できる第 17 章の生成モデルを用いることで、ロボティクスにおける行動クローニング（behavior cloning）の標準的な形態となっている。さらにオフライン手法は、このようなモデルを方策として採用し、上述のようにログへ向けて制約を課す (Z. Wang et al. 2023)。その後には、次に述べるオンラインのファインチューニングが続けられることも多い。

多くの応用では、既存のログに加えて少量の新しいデータも収集できる。Cal-QL (Nakamoto et al. 2023) は、悲観的な価値がロギング方策などの参照方策の価値を下回らないように較正するこ

とで、価値が過度に低い場合にオンラインのファインチューニング開始時に生じる収益の落ち込みを防ぐ。RLPD (Ball et al. 2023) はオフライン訓練を省略し、各バッチの半分をログから抽出しつつ前項までの正規化されたクリティックを用いて、最初からオフポリシー学習器をオンラインで走らせる。

15.9.6 事後学習の変種

15.8 節では、14.8 節のグループ重み、PPO のクリップされた比、および 15.3 節の KL ペナルティから事後学習の更新を組み立て、DAPO、Dr. GRPO、GSPO を挙げた。推論モデル（reasoning models）の規模で訓練すると (Shao et al. 2024; DeepSeek-AI 2025)、この更新は 1 トークンのタスクでは現れない問題に直面し、各変種がそれに対処している。DAPO (Q. Yu et al. 2025) はさらに、まず各応答内で平均するのではなくバッチ内の全トークンにわたって損失を平均することで、長い応答がその長さに比例して寄与するようにし、長さ上限で打ち切られた応答の報酬にシェイピングを施す。Dr. GRPO (Z. Liu et al. 2025) は、取り除く 2 つの除算に 2 つのバイアスの原因を帰着させている。応答長で割ると長い誤答のトークンあたりペナルティが縮むため訓練によって誤答が長くなり、グループの標準偏差で割るとグループ内がほぼ全問正解または全問不正解であるプロンプトに最大の重みが与えられてしまう。GSPO (C. Zheng et al. 2025) は、トークンごとの比がとりわけノイジーになる大規模な混合エキスパート（mixture-of-experts）モデルの安定性のために導入された。

プロセス報酬（process rewards）は、報酬の出所を変える。Lightman et al. (2024) は個々の推論ステップに対する人間のラベルで報酬モデルを訓練し、最終回答のみで訓練されたモデルよりも適切に解法を判定できることを示した。学習されたプロセス報酬は、15.8 節のプレフィックス価値と同様に各ステップへ個別のクレジットを与える一方で、15.3 節で測定したような、学習された報酬が持つ悪用され得る誤差も伴う。

15.9.7 本章の先の話題

これら 2 つの章で置いた 4 つの仮定は成り立たないことがある。複数のエージェントが同時に学習する場合、各エージェントの環境には他のエージェントが含まれるため、定常なものは何もなくなる。自己対戦（self-play）と探索を組み合わせた手法は、ヘッズアップ・ノーリミット・ポーカーにおいて人間を超えるプレイに到達した (Brown and Sandholm 2017)。エージェントが状態ではなく観測しか得られない場合、14.1 節のマルコフ性は崩れ、方策にはメモリ、リカレントネットワーク、あるいは状態上の信念（belief）が必要となる (Kaelbling et al. 1998)。タスクが変化する場合、エージェントは少数のエピソードから新しいタスクへ適応することを学習したり（メタ学習、meta-learning）、多数のステップにわたって行動するサブ方策とそれらを切り替える方策を合わせて学習したりできる（階層型学習、hierarchical learning）。報酬が極めて疎でありランダムな行動ではほとんど到達できない場合、ϵ グリーディやエントロピーボーナスでは探索が遅すぎるため、エージェントは未見の状態を自ら探しに行かなければならない。ネットワークは 14.4 節のボーナスに変換できる訪問回数を保持しないため、深層手法はネットワークが供給できる量で代用する。すなわち、密度モデルの擬似カウント（pseudo-count）(Bellemare et al. 2016)、次の状態を予測するネットワークの誤差 (Pathak et al. 2017)、あるいは固定されたランダムネットワークの出力を予測するネットワークの誤差 (Burda et al. 2019) であり、いずれもエージェントの経験が希薄な領域で報酬に加算される。あるいは、価値関数のアンサンブルからランダムに 1 つ選んだメンバーに 1 エピソード全体を通して従う方法もあり、データが希薄な領域ではメンバー間の予測が食い違うことを利用する (Osband et al. 2016)。Murphy の概説 (Murphy 2025) はこれら 4 つすべてを概観している。

15.9.8 参考書、ベンチマーク、コードベース

第 14 章の基礎資料（CleanRL の単一ファイル実装 (Huang, Rousslan F. J. Dossa, et al. 2022) や PPO の実装上の細部をまとめたカタログ (Huang, Rousslan Fernand Julien Dossa, et al. 2022) を含む）に加え、以下の資料が本章の各節に対応している。

- Sutton と Barto (Sutton and Barto 2018) の第 11〜13 章は、15.1 節の λ 収益と適格度トレース（eligibility traces）、それらに基づくアクタークリティック法、および 15.4 節の死の三要素と、その 3 要素のうち 1 つを外したときに何が証明できるかを扱っている。また第 8 章は、Dyna と学習されたモデルを用いた計画（15.6 節）を展開している。
- Engstrom et al. (2020) による統制実験は 15.2 節に対応している。コードレベルの選択を揃えれば PPO と TRPO はほぼ同等の性能を示すため、実用上の PPO の優位性の多くは目的関数が言及しない細部に由来する。
- RL Baselines3 Zoo (Raffin et al. 2021) は、PPO（15.2 節）、DQN（15.4 節）、SAC（15.5 節）について環境ごとに調整済みのハイパーパラメータを提供している。
- rliable (R. Agarwal et al. 2021) は、少数のシードにわたる深層強化学習の実行結果を比較するための層化ブートストラップ信頼区間と性能プロファイルを提供している。
- Levine et al. (2020) のサーベイ論文は、15.7 節の内容を表形式での診断から深層オフライン手法へと引き継いでいる。
- The RLHF Book (Lambert 2026) は、15.3 節と 15.8 節で組み立てた更新を用いる事後学習手法を、本格的な言語モデルの規模で展開している。

15.9.9 プロジェクト

15.8 節では事後学習の更新を組み立て、それを単一のグループに適用した。このプロジェクトでは、その周囲に訓練ループを構築し、本章で厳密に計算できる結果と照合して検証する。

1. **タスク**。検証器タスクの 4 つのプロンプトを用い、15.8 節と同様に 2 つの数字トークンで回答させて、`04`、`12`、`64`、`81` が正解の応答となるようにせよ。方策は、プロンプトごとの十の位の数字に対する選好度（preference）と、プロンプトおよび十の位の数字ごとの一の位の数字に対する選好度の表とし、参照方策は一様分布とする。
2. **更新**。応答が 2 トークンの場合へ `grpo_step` を拡張せよ。各プロンプトについて 16 個の応答からなるグループをサンプリングし、トークン対数確率を保存して、グループ内で報酬を標準化せよ。続いて、ステップサイズ 1 の勾配上昇法を E エポック実行せよ。目的関数は、$\epsilon_{\text{clip}} = 0.2$ でクリップしたトークンごとの代理目的関数をグループ内の全トークンにわたって平均したものから、β と 15.8 節の k_3 項の積を引いたものとし、各エポックのクリップ率（clip fraction）を記録せよ。
3. **確認事項**。最初のエポックではすべての比が 1 に等しくクリップ率は 0 であるため、$E = 1$ のときこの更新はグループ標準化された重みを用いた REINFORCE ステップと一致する。
4. **再利用の基準**。同じステップサイズかつ $\beta = 0$ のもとで、グループあたり 4 エポックを実行すると、試すどのシードでも、1 エポックの場合に必要なサンプル応答数の約半分で成功率 0.9 に達し、同時にクリップ率は第 1 エポックから第 4 エポックにかけて上昇するはずである。
5. **ペナルティの基準**。1 トークンの検証器タスクにおいて 15.8 節の寛容な判定器を用い、14.8 節の 1 つ抜き（leave-one-out）重みと $\beta = 1/2$ のもとで方策が落ち着くまで訓練せよ。k_3 損失のもとでは、ヘッジにより確率の約半分が逆向きの KL を持つ目的関数の最大点である `8*8` に保たれるはずである（演習問題 5）。一方、ペナルティを報酬に折り込んだ場合には、傾斜最適解（tilted optimum）である 2% 未満にとどまるはずである。これら 2 つの数値を得るために 1 つ抜き重みが必要となる理由と、グループ標準化を用いると何が変わるかを説明せよ。

Chapter 16

敵対的生成ネットワーク

100 次元の確率ベクトル $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ を抽出し、それをニューラルネットワーク $G(\mathbf{z})$ に通して画像を生成するとする。このような生成器ネットワークを記述するだけなら数行のコードで済むが、難しいのはその訓練である。観測されたラベルやトークンの対数尤度（log-likelihood）を最大化する分類器や自己回帰言語モデルとは異なり、画像を直接出力する生成器は、最大化すべき明示的な密度 $q(\mathbf{x})$ を持たない画像上の陰的な分布（implicit distribution）を定義する。サンプリングはできるが尤度を評価できないとき、生成モデル（generative model）をどのように訓練すればよいのだろうか。

分類器を用いれば、サンプルだけから 2 つの分布を比較できることはすでに見てきた。3.6.2 節で共変量シフト（covariate shift）を補正した際には、訓練入力とテスト入力を識別する二値分類器を訓練し、その最適ロジットから 2 つの分布間の対数密度比（log density ratio）を復元した。Goodfellow et al. (2014) は、この 2 サンプル比較を生成モデルの訓練原理へと発展させた。すなわち、**敵対的生成ネットワーク**（generative adversarial network、GAN）では、実データと生成サンプルを識別するように**識別器**（discriminator、または**クリティック**）D を訓練し、同時に、識別器が実データと見分けられないようなサンプルを生成するように**生成器**（generator）G を訓練する。識別器の勾配を通じて、生成器はどちらの密度も明示的に書き下すことなく、自身のサンプルがデータ分布に対して過剰または過少になっている領域を学習する。

汎用的な画像生成においては、拡散モデルやフローマッチングモデル（第 17 章）が単体の GAN にほぼ取って代わったものの、敵対的目的関数（adversarial objective）は依然として現代の生成システムの中核をなす構成要素であり、潜在拡散モデル（latent diffusion model）を支える潜在オートエンコーダや、ニューラル音声ボコーダ、複数ステップの拡散教師モデルから蒸留された 1 ステップ生成器などの訓練に用いられている。

本章では、敵対的モデリングを 3 段階で展開する。まずは目的関数とダイバージェンスから始める。16.1 節では対数損失における最適識別器、そのイェンセン・シャノン・ダイバージェンス（Jensen–Shannon divergence）、および非飽和生成器損失（non-saturating generator loss）を導出する。続く 16.2 節ではこの枠組みを f ダイバージェンスと、積分確率指標（integral probability metric）である最大平均値差異（maximum mean discrepancy、MMD）および 1 次ワッサースタイン距離（Wasserstein-1 distance）へ一般化し、16.3 節ではブラッドリー・テリー（Bradley–Terry）ランキングに基づく相対論的ペア型クリティック（RpGAN）を導入する。次に、16.4 節では Dirac-GAN 上でミニマックス訓練ダイナミクス（training dynamics）を調べ、正則化なしの更新が回転したり外側へ渦巻状に発散したりする理由と、ゼロ中心勾配ペナルティ（R_1 と R_2）が R3GAN のレシピをいかに安定化するかを示す (Huang et al. 2024)。最後に、16.5 節と 16.6 節では、畳み込みアーキテクチャと射影条件付きアーキテクチャを、特徴空間の評

価指標（FID、KID、k-NN の適合率と再現率）とあわせて画像生成へとスケールさせ、16.7 節では拡散蒸留、画像トークナイザ、音声ボコーダにおいて敵対的クリティックが条件付き平均によるぼけをいかに防ぐかを解説する。

記法

Table 16.1 に、本章全体で用いる記法をまとめる。ダイバージェンスとエントロピーは nat を単位として測り、第 37 章に従う。すなわち、D_{KL} はカルバック・ライブラー・ダイバージェンス、JS はイェンセン・シャノン・ダイバージェンス、H は微分エントロピーまたは離散エントロピー、$H_b(t)$ はパラメータ t のベルヌーイ分布の二値エントロピーを表す。

Table 16.1: 本章で用いる記法。

記号	意味	初出	よく使われる別記法
p、q、$m = \frac{1}{2}(p+q)$	データ密度、生成器密度、均衡混合密度	16.1 節	p_{data}、p_g
$x \sim p$、$x' \sim q$、$y \in \{0,1\}$	実サンプル、生成サンプル、出所ラベル（1 = 実データ）	16.1 節	
$z \in \mathbb{R}^{d_z}$、G	潜在ベクトルと生成器ネットワーク（$x' = G(z)$）	16.1 節	
D、$V(D)$	クリティック・識別器の真実性ロジット、ミニマックス価値関数	16.1 節	
$\rho = p/q$、$\lambda = \log(p/q)$	密度比と対数密度比	16.1 節	
w_{sat}、w_{ns}、w_{Rp}	サンプルごとの生成器勾配重み（飽和型、非飽和型、相対論的）	16.1 節、16.3 節	
$\eta = p/(p+q)$	ベイズクラス事後確率 $P(y=1 \mid x)$	16.2 節	
ℓ、$L(\eta)$、$\Delta_\ell(p,q)$	マージン利得、条件付きベイズリスク、ベイズリスクギャップ	16.2 節	
$D_f(p \,\|\, q)$、$k(x,x')$	凸関数 f による f ダイバージェンス、正定値カーネル	16.2 節	
$\Phi(D)$、$d_{\mathrm{Rp}}(p,q)$	相対論的ペアリング目的関数とペアリングダイバージェンス	16.3 節	
R_1、R_2、γ	実サンプルおよび生成サンプルにおけるゼロ中心勾配ペナルティ、ペナルティ重み	16.4 節	
c、e_c、$\phi(x)$	条件付け変数、条件埋め込み、特徴表現	16.5 節、16.6 節	

16.1 敵対的生成ネットワーク

Goodfellow et al. (2014) は、生成器（generator）ネットワークに、実データと生成サンプルを見分けるよう学習する**識別器**（discriminator）または**クリティック**と呼ばれる二値分類器を組み

合わせることで、**敵対的生成ネットワーク**（generative adversarial network、GAN）を導入した。この二値分類器は、3.6.2 節で共変量シフト（covariate shift）の補正に用いたドメイン分類器とよく似ている。

ここで直ちに、2 つの根本的な疑問が生じる。

1. 生成器が識別器を騙せたとき、生成器は実際にデータ分布 p と一致するよう学習したといえるのか。
2. 識別器が生成サンプルと実データを容易に分離できるときでも、その勾配は生成器を p へと導くのか。

固定された生成器に対する最適識別器は閉形式で導出できるため、これら両方の疑問に解析的に答えることができる。その後、厳密な密度と KL ダイバージェンスが閉形式で得られる 2 次元ガウス分布の問題でこの解析を検証し、最後に Fashion-MNIST 画像上で多層パーセプトロン（multilayer perceptron、MLP）の生成器を訓練する。

16.1.1 尤度から 2 サンプル比較へ

16.1.1.1 陰的生成器

最尤推定（maximum likelihood estimation）では、モデルの対数密度（log density）$\log p_\theta(x)$ を各観測値において評価する必要がある。これに対し、本章で扱う生成器は**陰的**（implicit）である。すなわち、潜在ベクトル $z \sim \mathcal{N}(0, I_{d_z})$ をサンプリングし、決定論的なネットワーク $x' = G(z)$ に通す。潜在次元 d_z がデータ次元 d_x より小さいとき、生成サンプルは低次元多様体上に乗り、$\mathbb{R}^{d_x}$ 上のルベーグ測度に関する密度を持たない。たとえ $d_z = d_x$ であっても、変数変換（change of variables）の公式によって誘導密度を計算するには、G が可逆であり、かつ計算可能なヤコビ行列式（Jacobian determinant）を持つ必要がある。

さらに、テストデータに対する対数尤度（log-likelihood）が高くても、写実的なサンプルが得られるとは限らない (Theis et al. 2016)。真のデータ分布と一様ノイズまたはガウスノイズの混合分布を考えてみよう。

$$\tilde{p} \;=\; 0.01\, p_{\text{data}} + 0.99\, p_{\text{noise}}. \tag{16.1}$$

いたるところで $\tilde{p}(x) \geq 0.01\, p_{\text{data}}(x)$ が成り立つため、その期待対数尤度は真のデータ分布より高々 $\log 100 \approx 4.6$ nat しか低くならない。対数尤度が数千 nat に及ぶ画像にとって、この差は無視できるほど小さい。それにもかかわらず、$\tilde{p}$ から抽出されるサンプルの 99% は純粋なノイズである。したがって、サンプル生成器を評価するには、生成されたサンプルを直接調べる必要がある。

16.1.1.2 実サンプルと生成サンプルの識別

2 サンプル検定（two-sample test）は 2 組のサンプルが同じ分布から得られたものかを調べる手法であり、それを実装する自然な方法は、両者を分離する二値分類器を訓練することである。GAN はこの分類器を用いて生成器に勾配を与える。$y \sim \text{Bernoulli}(\frac{1}{2})$ を均衡のとれた出所ラベルとし、$y = 1$ ならばデータ分布から $x \sim p$ を抽出し、$y = 0$ ならば生成器分布から $x \sim q$ を抽出する。このとき x の周辺分布（marginal distribution）は均衡混合分布 $m = (p + q)/2$ となる。

クリティック（または**識別器**）は**真実性ロジット**（realness logit）を出力する関数 $D : \mathcal{X} \to \mathbb{R}$ であり、$\sigma(D(x)) = 1/(1 + e^{-D(x)})$ が事後確率（posterior probability）$P(y = 1 \mid x)$ の推定値となる。Goodfellow et al. (2014) に倣い、コード中では識別器ネットワークに `net_D` と

update_D を用いるが、本文中では本章後半の非確率的なスコアリングネットワークも包含するため**クリティック**という用語も併用する（なお、GAN のクリティックは、15.1 節で扱うアクター・クリティック強化学習における価値関数（value function）のクリティックとは別物である）。

Figure 16.1 に示すように、訓練は 2 つのネットワークの間で交互に進める。識別器は実データのミニバッチと生成データのミニバッチを分離するように更新し、生成器は自身のサンプルが偽物と分類されにくくなるよう、識別器を通じて逆伝播された勾配によって更新する。

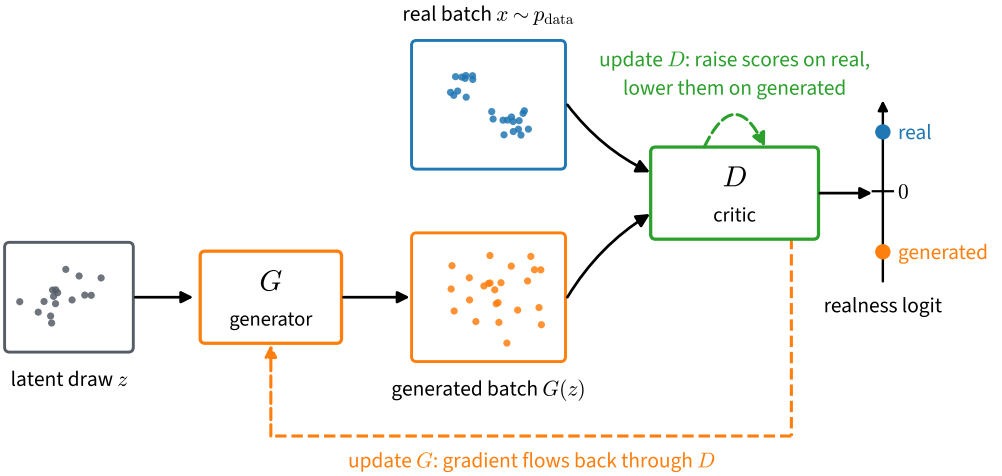


Figure 16.1: 敵対的訓練。生成器 G は潜在変数 z を生成サンプルへ写像し、クリティック D は生成サンプルと実サンプルの両方にスコアを付ける。クリティックは実データに対するスコアを最大化し生成サンプルに対するスコアを最小化する一方、生成器は D を通じた勾配を用いて 2 つの分布を見分けがつかないようにする。

16.1.2 対数損失の目的関数

均衡のとれた実サンプルと生成サンプルに対して二値交差エントロピー（binary cross-entropy、対数損失）で識別器を訓練すると、Goodfellow et al. (2014) のミニマックス価値関数が得られる。

$$V(D) \;=\; \mathbb{E}_{x\sim p}\big[\log \sigma(D(x))\big] + \mathbb{E}_{x'\sim q}\big[\log \sigma(-D(x'))\big]. \tag{16.2}$$

識別器は $V(D)$ を最大化する一方、生成器は第 2 項のみに影響し $V(D)$ を最小化しようとするため、ミニマックス問題 $\min_G \max_D V(D)$ が得られる。

16.1.2.1 最適クリティック

固定された生成器分布 q に対して、Equation 16.2 の最大化は各点 x ごとに独立な問題へ分解される。各点 x における被積分関数は $p(x)\log\sigma(D(x)) + q(x)\log\sigma(-D(x))$ である。$u = \sigma(D(x)) \in (0,1)$ とおくと、狭義凹関数 $p\log u + q\log(1-u)$ を一階条件（first-order condition）$p/u = q/(1-u)$ によって最大化することで、次の唯一の最適解が得られる。

$$\begin{aligned}\sigma(D^\star(x)) &= \frac{p(x)}{p(x)+q(x)} = P(y=1 \mid x),\\ D^\star(x) &= \log\frac{p(x)}{q(x)} = \lambda(x).\end{aligned} \tag{16.3}$$

したがって、最適識別器はベイズ事後確率 $P(y=1 \mid x)$ を出力し、そのロジット $D^\star(x)$ は**対数密度比**（log density ratio）$\lambda(x) = \log \rho(x)$（ただし $\rho(x) = p(x)/q(x)$）に等しい。$q(x)$ が閉形式の密度を持たなくても、二値分類によってサンプルから直接 $\log(p/q)$ が推定される。さらに、対数損失の目的関数は $D^\star$ の加法定数を一意に定める。すなわち、$D^\star$ を任意の非ゼロ定数 b だけシフトすると V は狭義に減少するため（演習問題 1）、$D^\star(x) > 0$ となるのはまさに $p(x) > q(x)$ の領域に限られる。

たとえば、分散 1 の 2 つのガウス分布 $p = \mathcal{N}(0,1)$ と $q = \mathcal{N}(\mu,1)$ に対して、最適ロジットは x に関するアフィン関数となる。

$$\lambda(x) = -\frac{x^2}{2} + \frac{(x-\mu)^2}{2} = \mu\Big(\frac{\mu}{2} - x\Big),$$

この関数は中点 $\mu/2$ で符号が反転する。p と q が異なる共分散行列を持つ場合、$\lambda(x)$ は x の二次関数となるため、非線形な識別器が必要になる。

16.1.2.2 イェンセン・シャノン・ダイバージェンスとの関係

最適識別器 $\sigma(D^\star) = p/(2m)$ と $\sigma(-D^\star) = q/(2m)$ を Equation 16.2 に代入すると、最適識別器に対して生成器が何を最小化しているかが明らかになる。

$$\begin{aligned}V(D^\star) &= \int p \log\frac{p}{2m} + \int q \log\frac{q}{2m}\\ &= D_{\mathrm{KL}}(p \,\|\, m) + D_{\mathrm{KL}}(q \,\|\, m) - 2\log 2.\end{aligned}$$

イェンセン・シャノン・ダイバージェンス（Jensen–Shannon divergence）の定義 $\mathrm{JS}(p,q) = \tfrac{1}{2}D_{\mathrm{KL}}(p \,\|\, m) + \tfrac{1}{2}D_{\mathrm{KL}}(q \,\|\, m)$ を用いると、次式が得られる。

$$\max_D V(D) = 2\,\mathrm{JS}(p,q) - 2\log 2. \tag{16.4}$$

$\mathrm{JS}(p,q) \geq 0$ であり、等号成立は $q = p$ のとき、かつそのときに限るため、$\max_D V(D)$ を最小化するとデータ分布 $q = p$ が大域的最小値（global minimum）$-2\log 2$ において一意に復元される（このときいたるところで $D^\star(x) = 0$ となり、識別器はランダムな当て推量に帰着する）。

$D_{\mathrm{KL}}(p \,\|\, m)$ と $D_{\mathrm{KL}}(q \,\|\, m)$ を微分エントロピー（differential entropy）によって展開すると、情報理論的な解釈が得られる。

$$\mathrm{JS}(p,q) \;=\; H(m) - \frac{H(p)+H(q)}{2} \;=\; I(x;y). \tag{16.5}$$

すなわち、$\mathrm{JS}(p,q)$ はサンプル $x \sim m$ とその二値の出所ラベル y の間の相互情報量（mutual information）$I(x;y)$ に等しい。二値ラベルが持つ情報量は高々 $\log 2$ nat であるため、$0 \leq$

$\mathrm{JS}(p, q) \leq \log 2$ が成り立ち、p と q が互いに素な台（support）を持つときは常に $\log 2$ に達する。

16.1.2.3 互いに素な台と勾配消失

$\mathrm{JS}(p, q)$ は $\log 2$ で飽和するため、p と q の台が重ならないときは常にミニマックス目的関数が定数になる。たとえば、$p = \delta_0$ と $q_\theta = \delta_\theta$ が $\mathbb{R}$ 上の点質量である場合、$\mathrm{JS}(p, q_\theta) = \log 2$ がすべての $\theta \neq 0$ で成り立つため、θ に関する勾配は、θ がどれほど 0 に近くてもゼロになる。裾で台が重なっている場合でさえ、たとえば $p = \mathcal{N}(0, 1)$ と $q = \mathcal{N}(\mu, 1)$ では、$\mathrm{JS}(p, q)$ は 0.11、0.34、0.63 nat（それぞれ $\mu = 1, 2, 4$ のとき）となり、$\mu = 8$ では 10^{-4} 未満の差まで $\log 2 \approx 0.6931$ に近づく。Figure 16.2 に曲線全体を示す。2 つのガウス分布が厳密な意味で重ならなくなるはるか手前で、曲線はすでに平坦になっている。

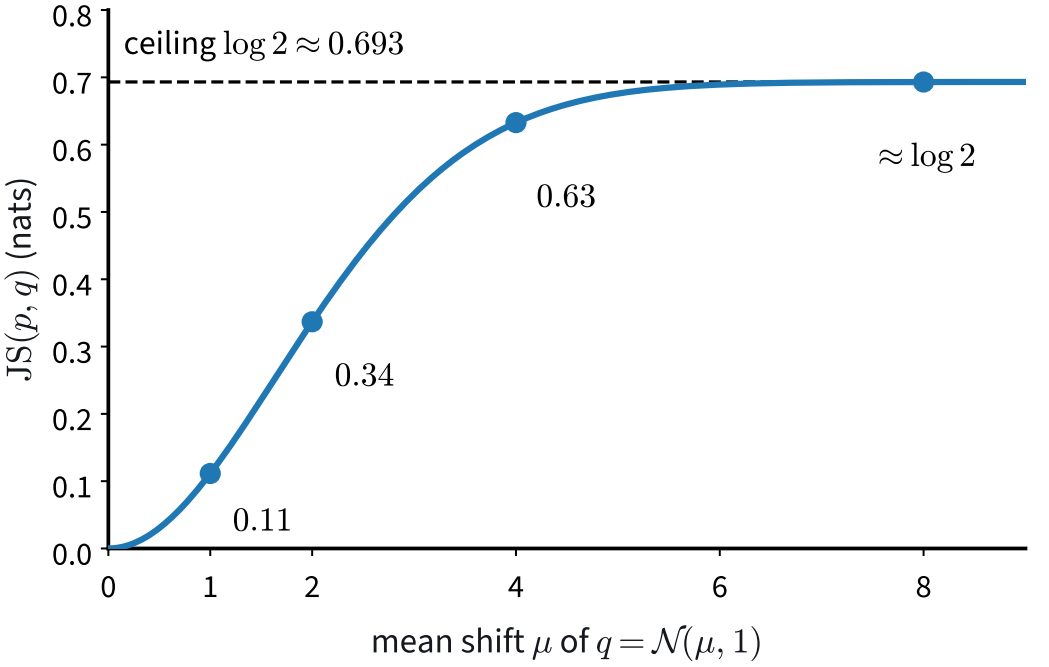


Figure 16.2: $p = \mathcal{N}(0, 1)$ と $q = \mathcal{N}(\mu, 1)$ の間のイェンセン・シャノン・ダイバージェンス（シフト μ の関数）。ダイバージェンスは標準偏差の数倍以内で上限 $\log 2$ に達する。それを超えると曲線は平坦になり、平坦な曲線からは生成器に勾配が与えられない。

高次元空間では、互いに素な台は例外ではなくむしろ常態である。自然画像も、$G : \mathbb{R}^{d_z} \to \mathbb{R}^{d_x}$ $(d_z \ll d_x)$ なる生成器の出力もともに低次元多様体上に集中し、それらの共通部分は通常、測度 0 となる (Arjovsky and Bottou 2017)。識別器が両者を完全に分離すると、ミニマックス値は飽和する。

最も直接的な解決策は、台を重なり合わせることである。識別器に入力する前に、すべての実サンプルと生成サンプルにスケール s の独立なガウスノイズを加えるとする。すると識別器は、いたるところで正の値を持つ 2 つの平滑化された密度を比較することになる。先ほどの点質量の場合、それらは $\mathcal{N}(0, s^2)$ と $\mathcal{N}(\theta, s^2)$ になり、両者のダイバージェンスは Figure 16.2 の曲線を $\mu = \theta/s$ で読み取った値に一致する。したがって、θ が s の数倍以内にある限り、生成器は勾配を受け取る。これが**インスタンスノイズ**である (Sønderby et al. 2017; Arjovsky and Bottou

2017)。ただし、このままでは生成器が元のデータではなくぼけたデータに一致してしまうため、訓練中に s を 0 に向けてアニーリングする必要がある。このアイデアは 16.7 節で再び登場し、そこでは拡散モデル（diffusion model）によってサンプルにノイズが加えられる。

16.1.3 生成器の勾配

実際の訓練では、生成器は固定された識別器 D に対して勾配ステップを踏むのであり、$\max_D V(D)$ を直接最小化するわけではない。生成器の目的関数の勾配を詳しく見てみよう。

16.1.3.1 飽和型と非飽和型の重み

文字どおりのミニマックス定式化では、生成器は $\mathbb{E}_z[\log \sigma(-D(G(z)))]$ を自身のパラメータ θ に関して最小化する。$\frac{d}{du}\log\sigma(-u) = -\sigma(u)$ と $x' = G(z)$ を用いると、その勾配は次のようになる。

$$\begin{aligned}&\nabla_\theta \, \mathbb{E}_z\big[\log \sigma(-D(G(z)))\big] \\ &\quad = -\,\mathbb{E}_z\big[\sigma(D(x'))\, \nabla_\theta D(G(z))\big].\end{aligned}$$

各生成サンプル x' は $\nabla_\theta D(G(z))$ に沿って上り方向へ押し上げられ、その重みは $\sigma(D(x'))$（識別器が推定する、x' が実データである確率）で与えられる。識別器が高い確信をもってサンプルを偽物と棄却するとき（$D(x') \ll 0$）、この重み $\sigma(D(x')) \approx e^{D(x')}$ は指数関数的に小さくなり、生成器が最も改善を必要としているまさにその局面で生成器の勾配が**飽和**（saturate）してしまう。

この飽和を避けるため、Goodfellow et al. (2014) は生成サンプルが実データと分類される対数確率（log probability）を最大化するよう、すなわち $-\mathbb{E}_z[\log \sigma(D(G(z)))]$ を最小化するよう生成器を訓練した。$\frac{d}{du}\log\sigma(u) = \sigma(-u)$ より、その勾配は次のようになる。

$$\begin{aligned}&\nabla_\theta\Big(-\mathbb{E}_z\big[\log \sigma(D(G(z)))\big]\Big) \\ &\quad = -\,\mathbb{E}_z\big[\sigma(-D(x'))\, \nabla_\theta D(G(z))\big].\end{aligned}$$

どちらの目的関数も各サンプルを同じ方向 $\nabla_\theta D(G(z))$ へ押し上げるが、サンプルごとのスカラー重みだけが異なる。

$$\begin{aligned}w_{\text{sat}}(x') &= \sigma(D(x')), \\ w_{\text{ns}}(x') &= \sigma(-D(x')) = 1 - \sigma(D(x')).\end{aligned} \tag{16.6}$$

Figure 16.3 に示すように、確信をもって棄却されるサンプル（$D(x') \ll 0$）に対して**非飽和型**（non-saturating）の重み $w_{\text{ns}}(x')$ は 1 に近づくのに対し、$w_{\text{sat}}(x')$ は $e^{D(x')}$ として減衰する（比は $w_{\text{ns}}/w_{\text{sat}} = e^{-D(x')}$ であり、$D = -3$ で $20\times$、$D = -5$ で $148\times$ となる）。

16.1.3.2 非飽和損失の実効ダイバージェンス

識別器が最適応答 $D = \lambda = \log(p/q)$ に保たれているとき、非飽和生成器損失はどのダイバージェンスを下るのだろうか。$\log\sigma(-D) - \log\sigma(D) = -D$ を用いると、非飽和型の目的関数は次のように分解できる。

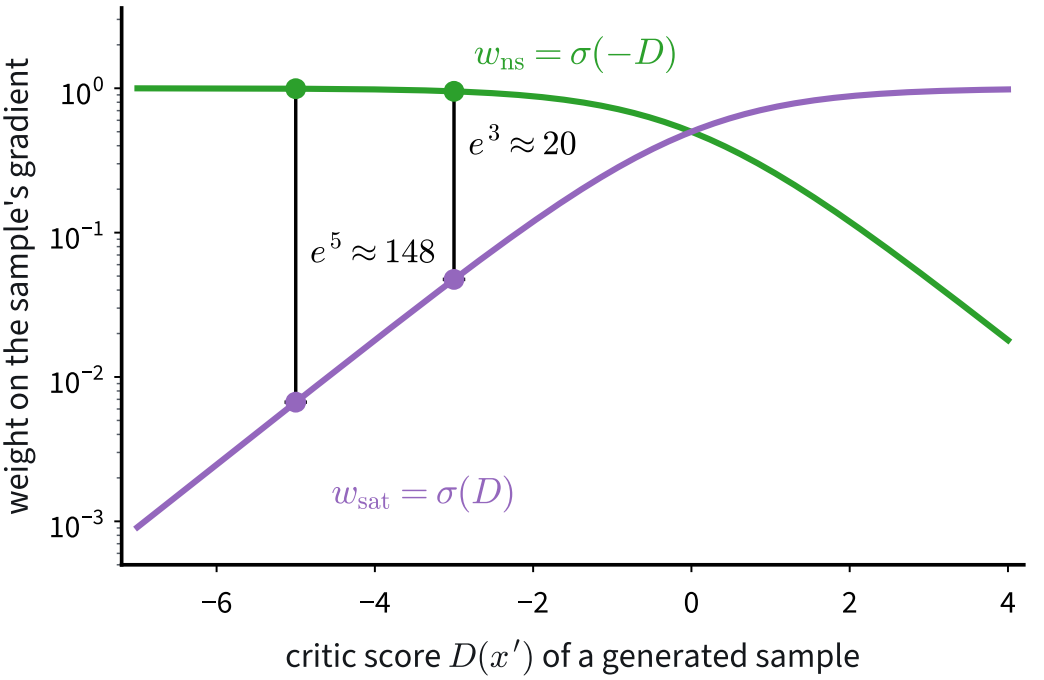


Figure 16.3: 識別器ロジット $D(x')$ の関数として対数スケールでプロットした、サンプルごとの生成器勾配の重み。確信をもって棄却されるサンプル（$D(x') \ll 0$）では、飽和型の重み $\sigma(D)$ が指数関数的に減衰するのに対し、非飽和型の重み $\sigma(-D)$ は 1 付近にとどまる。

$$
\begin{aligned}
&- \mathbb{E}_{x'}\big[\log \sigma(D(x'))\big] \\
&\quad = \mathbb{E}_{x'}\big[-D(x')\big] - \mathbb{E}_{x'}\big[\log \sigma(-D(x'))\big].
\end{aligned}
$$

D を λ に固定すると、右辺の第 1 項は q に関して逆方向 KL ダイバージェンス $D_{\mathrm{KL}}(q \,\|\, p) = \mathbb{E}_q[\log(q/p)]$ と同じ勾配を持ち、第 2 項はミニマックス損失にマイナスを付けたものであり、包絡線定理（envelope theorem）によりその勾配は $2\,\mathrm{JS}(p, q) = D_{\mathrm{KL}}(p \,\|\, m) + D_{\mathrm{KL}}(q \,\|\, m)$ の勾配と一致する (Arjovsky and Bottou 2017)（演習問題 5）。$\log(q/p) - \log(q/m) = \log(m/p)$ と $p + q = 2m$ を用いて積分をまとめると、次式が得られる。

$$
\begin{aligned}
&D_{\mathrm{KL}}(q \,\|\, p) - D_{\mathrm{KL}}(p \,\|\, m) - D_{\mathrm{KL}}(q \,\|\, m) \\
&\quad = \int q \log \frac{m}{p} + \int p \log \frac{m}{p} \\
&\quad = \int (p + q) \log \frac{m}{p} \;=\; 2\, D_{\mathrm{KL}}(m \,\|\, p).
\end{aligned}
\tag{16.7}
$$

したがって、最適クリティックに対して非飽和型の生成器は $2\, D_{\mathrm{KL}}(m \,\|\, p)$、すなわち**軟化された逆方向 KL ダイバージェンス**（softened reverse KL divergence）を下ることになり (Shannon et al. 2020)、これも唯一の大域的最小値 $q = p$ を共有する。

$2\, D_{\mathrm{KL}}(m \,\|\, p)$ を詳しく見ると、誤差へのペナルティの課し方に非対称性があることがわかる。

- $q(x) > 0$ かつ $p(x) = 0$ となる領域（データの台の外側にある偽サンプル）では、$m(x) > 0 = p(x)$ より $D_{\mathrm{KL}}(m \,\|\, p) = \infty$ となり、非現実的なサンプルに厳しいペナルティが課される。
- $p(x) > 0$ かつ $q(x) = 0$ となる領域（データの欠落したモード）では、$m(x) = p(x)/2$ より被積分関数の寄与は $p(x) \log \frac{1}{2}$ という小さな有限のペナルティにとどまる。

16.1.3.3 モード欠落

生成器が多峰性分布 p の一部のモードしかカバーしない現象を、**モード欠落**（mode dropping）、または**モード崩壊**（mode collapse）と呼ぶ。いま、p が互いに素な 2 つのモードの等重み混合分布であり、q が一方のモードには完全に一致し、もう一方を完全に落としているとする。カバーされたモード上では $p = q/2$ であるため、$D_{\mathrm{KL}}(q \,\|\, p) = \log 2 \approx 0.69$ nat、$\mathrm{JS}(p, q) \approx 0.22$ nat、$2\, D_{\mathrm{KL}}(m \,\|\, p) \approx 0.26$ nat となる。データ分布の半分を落としても、わずか 1/4 nat のコストしか生じないのである。これに対し、最尤法（maximum likelihood）は順方向 KL ダイバージェンス $D_{\mathrm{KL}}(p \,\|\, q)$ を最小化するため、p のいかなる部分にせよ密度 0 が割り当てられれば無限大となる。

さらに悪いことに、カバーされたモード上での識別器の最適応答 $D^\star(x) = \log(p(x)/q(x)) = -\log 2$ においては、$D^\star$ が q の台全体にわたって定数となるため、$\nabla_\theta D^\star(G(z)) = 0$ となる。すなわち、カバーされたモード上の局所勾配は消失し、欠落したモードへ向かう信号をまったく与えない。仮に不完全な識別器が、カバーされたモード上の生成サンプルに $D = 3$、カバーされていないモード上の実サンプルに $D = 5$ を割り当てたとしても、各生成サンプルは固定されたゼロのしきい値に対してのみ $\sigma(-3) \approx 0.05$ で重み付けされ、カバーされていない実モード上のより高いスコアが生成器の更新に入り込むことは決してない。16.3 節では、生成サンプルのスコアを実サンプルのスコアと直接比較することでこの問題に対処できるかを検証する。

16.1.4 2 次元ガウス分布での理論の検証

理論的予測を定量的に検証するため、まず p、q、$\lambda(x) = \log(p(x)/q(x))$、および $D_{\mathrm{KL}}(q \,\|\, p)$ がすべて閉形式で計算できる 2 次元のガウス分布データ上で線形生成器を訓練する。

16.1.4.1 交互更新則

本章全体で再利用するため、`update_D`、`update_G`、および交互訓練ループ `train_gan` をここで一度定義し、`d2l` に保存しておく。まず `update_D` は、Equation 16.2 を最大化する識別器の更新を 1 ステップ実行する（実サンプルにはラベル 1、切り離した生成サンプルにはラベル 0 を与えて二値交差エントロピーを最小化する）。合計された損失をバッチサイズで割ると $-V(D)$ の推定値となり、その値は $2\log 2 \approx 1.386$（$D = 0$ のとき）に等しい。

```
def update_D(X, Z, net_D, net_G, trainer_D):
    """Update the critic; return its log loss summed over both batches
    (per sample, the sum of two means: 2 log 2 at equilibrium)."""
    trainer_D.zero_grad()
    real_Y = net_D(X)
    # The generator is the fixed player here: detach its output
    fake_Y = net_D(net_G(Z).detach())
    loss_D = (F.binary_cross_entropy_with_logits(
                  real_Y, torch.ones_like(real_Y), reduction='sum') +
              F.binary_cross_entropy_with_logits(
                  fake_Y, torch.zeros_like(fake_Y), reduction='sum'))
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

次に `update_G` は、非飽和損失 $-\mathbb{E}[\log \sigma(D(G(z)))]$ を最小化する生成器の更新を 1 ステップ実行する。これは再計算した識別器スコア `net_D(net_G(Z))` に対する正解ラベル 1 の二値交差エントロピーとして実装され、サンプルあたりの値は $\log 2 \approx 0.693$（$D = 0$ のとき）に等しい。

```
def update_G(Z, net_D, net_G, trainer_G):
    """Update the generator on the non-saturating loss; return its sum."""
    trainer_G.zero_grad()
    # net_D changed in its half-step, so recompute the scores
    fake_Y = net_D(net_G(Z))
    loss_G = F.binary_cross_entropy_with_logits(
        fake_Y, torch.ones_like(fake_Y), reduction='sum')
    loss_G.backward()  # also fills net_D's .grad (update_D zeroes it)
    trainer_G.step()
    return loss_G
```

`train_gan` は、変化する目的関数にモメンタムが素早く追従できるよう $\beta_1 = 0.5$ とした Adam を用い (Radford et al. 2015)、ミニバッチごとに `update_D` と `step_G` を交互に実行する。また、オプションとしてエポックごとのコールバック `on_epoch` と指数移動平均（exponential moving average）`ema` も受け付ける。

```
def train_gan(net_D, net_G, data_iter, num_epochs, latent_dim, lr_D, lr_G,
              step_G=update_G, seed=None, on_epoch=None, ema=None):
    """Alternate update_D and step_G over the minibatches with Adam
```

```
    (beta_1 = 0.5), calling ema.update(net_G) after every generator step
    and on_epoch(epoch) after each epoch if given; return each epoch's
    per-sample losses (loss_D, the sum of the critic's two batch means,
    and loss_G)."""
    if seed is not None:
        torch.manual_seed(seed)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D,
                                 betas=(0.5, 0.999))
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G,
                                 betas=(0.5, 0.999))
    history = []
    for epoch in range(1, num_epochs + 1):
        loss_D, loss_G, n = 0, 0, 0  # summed on the device, read once
        for X, *_ in data_iter:  # labels, if any, are ignored
            Z = torch.randn(X.shape[0], latent_dim, device=X.device)
            loss_D += update_D(X, Z, net_D, net_G, trainer_D).detach()
            loss_G += step_G(Z, net_D, net_G, trainer_G).detach()
            if ema is not None:
                ema.update(net_G)
            n += X.shape[0]
        history.append(torch.stack([loss_D, loss_G]) / n)
        if on_epoch is not None:
            on_epoch(epoch)
    return torch.stack(history).cpu().numpy()
```

16.1.4.2 既知の密度を持つ線形生成器

1,000 個の訓練点を $x = zA + b$（ただし $z \sim \mathcal{N}(0, I_2)$）から生成すると、真のデータ分布は傾いた楕円形のガウス分布 $p = \mathcal{N}(b, A^\top A)$ となる。

```
torch.manual_seed(0)
Z = torch.normal(0.0, 1.0, (1000, 2))
A = torch.tensor([[1.0, 2.0], [-0.1, 0.5]])
b = torch.tensor([1.0, 2.0])
data = Z @ A + b
print(f'covariance of the data distribution:\n{A.T @ A}')
```

```
covariance of the data distribution:
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```

生成器は単一の線形層 $G(z) = zA_q + \mu_q$ であり、ガウス分布 $q = \mathcal{N}(\mu_q, A_q^\top A_q)$ を誘導する。異なる共分散を持つ 2 つのガウス分布間の対数密度比は二次関数になるため、識別器には 16 個の Leaky ReLU ユニット（負の傾き 0.2）からなる 2 つの隠れ層を持つ 3 層の MLP を用いる（代わりに飽和する tanh 活性化関数を用いた場合に何が起きるかは、演習問題 3 で調べる）。

```
def make_nets(seed=7):
    """The linear generator and the critic, all weights and offsets drawn
    from N(0, 0.02^2) by a generator seeded with seed."""
    gen = torch.Generator().manual_seed(seed)
    net_G = nn.Sequential(nn.Linear(2, 2))
```

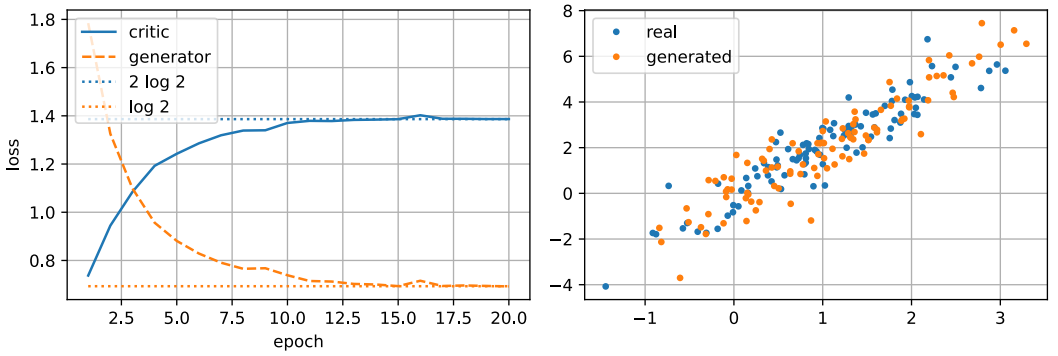
```
    net_D = nn.Sequential(nn.Linear(2, 16), nn.LeakyReLU(0.2),
                          nn.Linear(16, 16), nn.LeakyReLU(0.2),
                          nn.Linear(16, 1))
    for w in [*net_G.parameters(), *net_D.parameters()]:
        nn.init.normal_(w, 0, 0.02, generator=gen)
    return net_G, net_D
```

生成器の移動に伴って識別器が最適応答に追従できるよう `lr_D=0.05`、`lr_G=0.005` と設定し、バッチサイズ 8 で 20 エポック訓練する。

```
data_iter = d2l.load_array((data,), batch_size=8)
net_G, net_D = make_nets()
history = train_gan(net_D, net_G, data_iter, num_epochs=20, latent_dim=2,
                    lr_D=0.05, lr_G=0.005, seed=1)
with torch.no_grad():
    fake = net_G(torch.normal(0, 1, (200, 2)))
```

ここで、`history` に記録されたエポックごとのクリティック損失と生成器損失をそれぞれの理論的均衡値（$2\log 2$ と $\log 2$）に対してプロットし、あわせて実サンプル（`data`）と生成サンプル（`fake`）の散布図を描く。

```
fig, axes = d2l.plt.subplots(1, 2, figsize=(9, 3.2))
epochs = list(range(1, len(history) + 1))
d2l.plot(epochs, [history[:, 0], history[:, 1], [2 * math.log(2)] * len(epochs),
                  [math.log(2)] * len(epochs)],
         'epoch', 'loss',
         legend=['critic', 'generator', '2 log 2', 'log 2'],
         fmts=('C0-', 'C1--', 'C0:', 'C1:'), axes=axes[0])
d2l.plot([data[:100, 0], fake[:100, 0]], [data[:100, 1], fake[:100, 1]],
         legend=['real', 'generated'], fmts=('C0.', 'C1.'), axes=axes[1])
fig.tight_layout()
```



10 エポック以内に、識別器損失は均衡値の $2\log 2$ まで上昇し、生成器損失は $\log 2$ に落ち着き、生成された点群は目標の楕円と一致する。

16.1.4.3 当てはまりの nat 単位での定量化

$G(z) = zA_q + \mu_q$ はガウス分布 $q = \mathcal{N}(\mu_q, \Sigma_q)$（ただし $\Sigma_q = A_q^\top A_q$）を定めるため、厳密な KL ダイバージェンス $D_{\mathrm{KL}}(q \,\|\, p)$ を d_x 次元で評価できる (Kullback 1959)。

$$\begin{aligned} &D_{\mathrm{KL}}(\mathcal{N}(\mu_0, \Sigma_0) \,\|\, \mathcal{N}(\mu_1, \Sigma_1)) \\ &\quad = \tfrac{1}{2}\Big[\operatorname{tr}(\Sigma_1^{-1}\Sigma_0) + (\mu_1 - \mu_0)^\top \Sigma_1^{-1}(\mu_1 - \mu_0) \\ &\qquad - d_x + \log \tfrac{\det \Sigma_1}{\det \Sigma_0}\Big]. \end{aligned} \tag{16.8}$$

```
def generator_gaussian(net_G):
    A_qT, b_q = net_G[0].weight.detach(), net_G[0].bias.detach()
    return b_q, A_qT @ A_qT.T  # nn.Linear stores A_q.T

def kl_gaussians(mu0, S0, mu1, S1):
    diff = mu1 - mu0
    return 0.5 * (torch.trace(torch.linalg.solve(S1, S0))
                  + diff @ torch.linalg.solve(S1, diff) - len(mu0)
                  + torch.logdet(S1) - torch.logdet(S0))

mu_p, Sigma_p = b, A.T @ A
for name, net in [('at initialization', make_nets()[0]),
                  ('after training   ', net_G)]:
    mu_q, Sigma_q = generator_gaussian(net)
    print(f'KL(q || p) {name}: '
          f'{kl_gaussians(mu_q, Sigma_q, mu_p, Sigma_p):.3f} nats')
```

```
KL(q || p) at initialization: 7.501 nats
KL(q || p) after training   : 0.021 nats
```

q の密度を一度も評価することなく、敵対的訓練（adversarial training）によって $D_{\mathrm{KL}}(q \,\|\, p)$ は初期化時の ~ 8 nat（初期重み $\mathcal{N}(0, 0.02^2)$ の低いエントロピーが支配的）から 100 分の数 nat まで減少する。

16.1.4.4 学習された対数密度比の検証

訓練された識別器が $\lambda(x) = \log(p(x)/q(x))$（Equation 16.3）を正確に復元できるかを検証するため、生成器 q を $A_q = [[1, 2], [-0.5, 0.5]]$、$\mu_q = (1.5, 1.5)$（$D_{\mathrm{KL}}(q \,\|\, p) \approx 3.4$ nat）に固定し、この凍結した q に対して新たな識別器を 2,000 ステップ訓練する。

```
net_G_q, net_D_q = make_nets()  # a fresh critic
with torch.no_grad():  # the hand-set generator; nn.Linear stores A_q.T
    net_G_q[0].weight.copy_(torch.tensor([[1.0, -0.5], [2.0, 0.5]]))
    net_G_q[0].bias.copy_(torch.tensor([1.5, 1.5]))
trainer_D = torch.optim.Adam(net_D_q.parameters(), lr=0.01)
torch.manual_seed(2)
for _ in range(2000):  # critic-only steps against the frozen generator
    X = data[torch.randint(0, len(data), (256,))]
    Zb = torch.normal(0, 1, (256, 2))
    update_D(X, Zb, net_D_q, net_G_q, trainer_D)
```

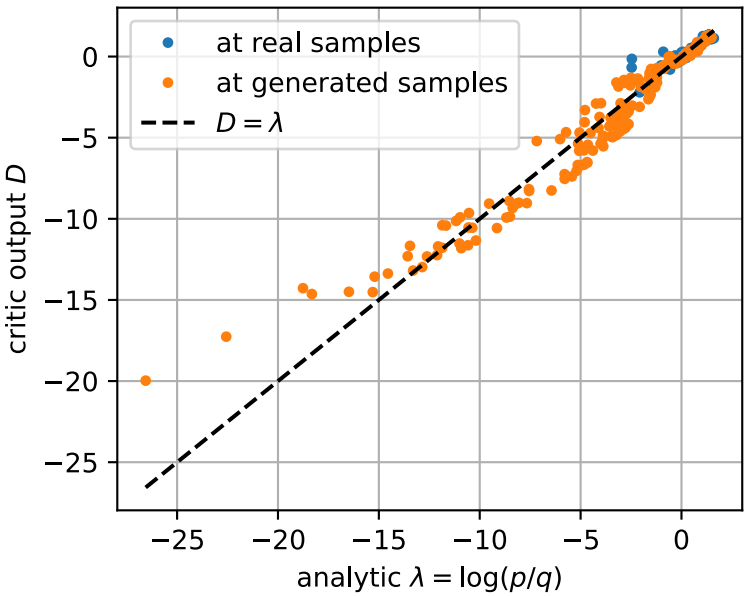
200 個の実データ点と 200 個の生成点にわたって識別器ロジット $D(x)$ と厳密な対数密度比 $\lambda(x)$ の両方を評価し、両者の相関と説明分散（explained variance）R^2 を求める。

```
def gaussian_logpdf(x, mu, Sigma):
    diff = x - mu
    sol = torch.linalg.solve(Sigma, diff.T).T
    return -0.5 * ((diff * sol).sum(1)
                   + torch.logdet(2 * torch.pi * Sigma))

mu_q, Sigma_q = generator_gaussian(net_G_q)
with torch.no_grad():
    xs = torch.cat([data[:200], net_G_q(torch.normal(0, 1, (200, 2)))])
    lam = (gaussian_logpdf(xs, b, A.T @ A)
           - gaussian_logpdf(xs, mu_q, Sigma_q))
    D_out = net_D_q(xs).squeeze()
r2 = 1 - ((D_out - lam) ** 2).mean() / ((lam - lam.mean()) ** 2).mean()
print(f'correlation {torch.corrcoef(torch.stack([lam, D_out]))[0, 1]:.3f}, '
      f'R^2 {r2:.3f}')
```

```
correlation 0.982, R^2 0.962
```

クリティックのロジット `D_out` を厳密な対数密度比 `lam` に対して 200 個の実サンプルと 200 個の生成サンプルにわたりプロットすると、$D(x)$ が恒等直線 $D = \lambda$ にどれほど密接に追従しているかが視覚的にわかる。

```
lims = [float(lam.min()), float(lam.max())]
d2l.plot([lam[:200], lam[200:], lims], [D_out[:200], D_out[200:], lims],
         r'analytic $\lambda = \log(p/q)$', 'critic output $D$',
         legend=['at real samples', 'at generated samples', r'$D = \lambda$'],
         fmts=('C0.', 'C1.', 'k--'), figsize=(4.2, 3.4))
```



学習されたロジットは真の対数密度比によく追従する（correlation ≈ 0.98、$R^2 > 0.90$）。ずれが生じるのは入が極端に負の値をとる領域（$p(x) \approx 0$ となる生成サンプル）のみである。これは、区分線形な Leaky ReLU ネットワークが裾に向かって二次関数的ではなく線形に外挿することに加え、$\frac{d}{dD}[-\log\sigma(-D)] = \sigma(D) \ll 1$（$D \ll 0$ のとき）となるため、識別器がいったん十分な確信を持つと、対数損失が識別器をさらに急峻にする勾配をほとんど生み出さなくなるからである。

16.1.4.5 飽和型と非飽和型の生成器損失

生成器がデータから遠く離れた位置から出発するときの飽和型と非飽和型の重み（Equation 16.6）を比較するため、生成器の初期バイアスを $(-6, 6)$ だけシフトさせ（データの平均から ~ 8 単位離れ、$D_{\mathrm{KL}}(q \,\|\, p) \approx 345$ nat）、`update_G` と、$\mathbb{E}_z[\log\sigma(-D(G(z)))]$ を直接最小化する `update_G_saturating` を比較する。

```
def update_G_saturating(Z, net_D, net_G, trainer_G):
    trainer_G.zero_grad()
    fake_Y = net_D(net_G(Z))
    # log sigma(-D) is minus the cross-entropy under the label zero
    loss_G = -F.binary_cross_entropy_with_logits(
        fake_Y, torch.zeros_like(fake_Y), reduction='sum')
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

各生成器の目的関数について、同一の初期化とミニバッチ（`lr_G=0.01`、`lr_D=0.05`）のもとで 3 つのシードを実行する。

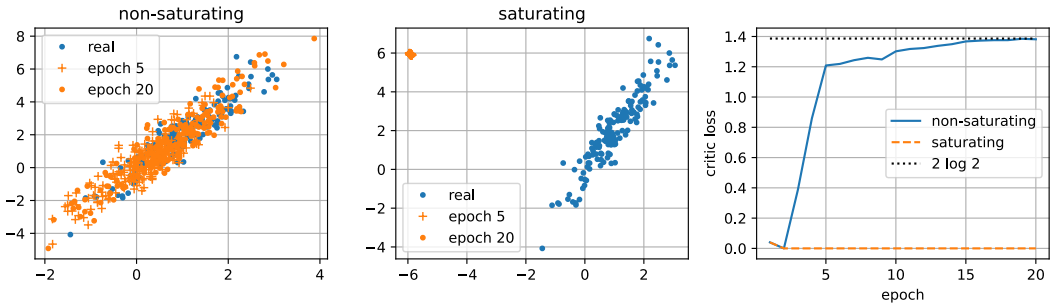
```
def far_start():
    """make_nets with the generator's offset moved by (-6, 6)."""
    net_G, net_D = make_nets()
    with torch.no_grad():
        net_G[0].bias += torch.tensor([-6.0, 6.0])
    return net_G, net_D

mu_q, Sigma_q = generator_gaussian(far_start()[0])
print(f'start of every run: KL(q || p) = '
      f'{kl_gaussians(mu_q, Sigma_q, mu_p, Sigma_p):.2f} nats')
torch.manual_seed(3)
Z_show = torch.randn(200, 2)  # one latent sample for every snapshot
runs = {}  # name: (losses, snapshots) of the first run, for the plot
for name, step in [('non-saturating', update_G),
                   ('saturating', update_G_saturating)]:
    kls = []
    for seed in (1, 2, 3):  # both arms see the same minibatches and draws
        net_G_ab, net_D_ab = far_start()
        snaps = {}  # epoch: generated samples
        hist = train_gan(net_D_ab, net_G_ab, data_iter, num_epochs=20,
                         latent_dim=2, lr_D=0.05, lr_G=0.01, step_G=step,
                         seed=seed, on_epoch=lambda epoch: snaps.update(
                             {epoch: net_G_ab(Z_show).detach()}))
        runs.setdefault(name, (hist, snaps))
```

```
            mu_q, Sigma_q = generator_gaussian(net_G_ab)
            kls.append(f'{kl_gaussians(mu_q, Sigma_q, mu_p, Sigma_p):.2f}')
        print(f'{name}: final KL(q || p) = {", ".join(kls)} nats')
```

```
start of every run: KL(q || p) = 344.20 nats
non-saturating: final KL(q || p) = 0.03, 0.02, 0.03 nats
saturating: final KL(q || p) = 337.75, 334.37, 336.51 nats
```

最終的な KL ダイバージェンスがこれほど大きく異なる理由を確認するため、各目的関数における 5 エポック目と 20 エポック目の生成点群 snaps を、クリティック損失の推移と並べてプロットする。

```
fig, axes = d2l.plt.subplots(1, 3, figsize=(10.5, 3.2))
for ax, (name, (hist, snaps)) in zip(axes[:2], runs.items()):
    d2l.plot([data[:150, 0], snaps[5][:, 0], snaps[20][:, 0]],
             [data[:150, 1], snaps[5][:, 1], snaps[20][:, 1]],
             legend=['real', 'epoch 5', 'epoch 20'],
             fmts=('C0.', 'C1+', 'C1.'), axes=ax)
    ax.set_title(name)
epochs = list(range(1, 21))
d2l.plot(epochs, [runs['non-saturating'][0][:, 0],
                  runs['saturating'][0][:, 0], [2 * math.log(2)] * 20],
         'epoch', 'critic loss', legend=['non-saturating', 'saturating',
                                         '2 log 2'],
         fmts=('-', '--', 'k:'), axes=axes[2])
fig.tight_layout()
```



最初の 2 エポックで、どちらの識別器も遠く離れた生成点群をデータから容易に分離し、識別器損失は 0 付近まで下がり $D(x') \ll 0$ となる。$w_{\text{sat}}(x') = \sigma(D(x')) \approx 0$ であるため、飽和型の生成器は完全に停滞する（20 エポック後もデータから依然として 335–340 nat 離れている）。これに対し、非飽和型の重み $w_{\text{ns}}(x') = \sigma(-D(x')) \approx 1$ は識別器の空間勾配 $\nabla_\theta D(G(z))$ を維持し、20 エポック以内に生成器をデータ分布上へと引き寄せる（$D_{\text{KL}}(q \,\|\, p)$ は 100 分の数 nat）。なお、w_{ns} は $\nabla_\theta D$ を再スケーリングするだけであり、Leaky ReLU 識別器が x' において非ゼロの空間的な傾きを持つことに依存している。

16.1.5 Fashion-MNIST での MLP GAN の訓練

画像データでは $p(x)$ も $q(x)$ も閉形式ではわからないため、サンプル品質と訓練損失を監視するほかない。最初の画像実験として、Fashion-MNIST（3.2 節）の 60,000 枚の 28×28 グレースケール画像を $[-1, 1]$ にスケーリングし、2 つの Leaky ReLU MLP を訓練する。生成器は 100 次

元の潜在ベクトルを 256 ユニットと 512 ユニットの隠れ層に通し、tanh 出力活性化関数によって 784 ピクセルへ写像する。識別器は 784 ピクセルを 512 ユニットと 256 ユニットの層に通して単一のロジットへ写像する。後の節でも単純な MLP を構築できるよう、補助関数 `make_mlp` を保存しておく。

```
def make_mlp(sizes, act):
    """Linear layers of the given widths with the activation module act
    between them; weights uniform in +-1/sqrt(fan_in), zero biases."""
    layers = []
    for d_in, d_out in zip(sizes[:-1], sizes[1:]):
        linear = nn.Linear(d_in, d_out)  # weights uniform by default
        nn.init.zeros_(linear.bias)
        layers += [linear, act]
    return nn.Sequential(*layers[:-1])
```

`make_mlp` を用い、60,000 枚の Fashion-MNIST 訓練画像を 784 次元のベクトル（値域 $[-1, 1]$）に再スケーリングし、ミニバッチローダ `image_iter` を構築したうえで、生成器 `net_G_img`（末尾は `tanh`）と識別器 `net_D_img` をインスタンス化する。

```
torch.manual_seed(0)
fmnist = d2l.FashionMNIST()
# All 60,000 training images as rows of 784 pixels in [-1, 1]
images = fmnist.train.data.reshape(-1, 784).float() / 127.5 - 1
image_iter = d2l.load_array((images,), batch_size=64)
leaky = nn.LeakyReLU(0.2)
net_G_img = nn.Sequential(make_mlp([100, 256, 512, 784], leaky), nn.Tanh())
net_D_img = make_mlp([784, 512, 256, 1], leaky)
```

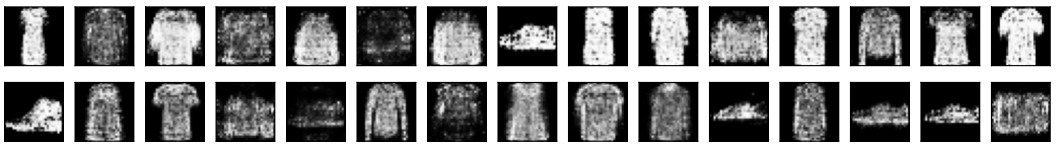
両ネットワークを学習率 2×10^{-4}、バッチサイズ 64 で 20 エポック（$\sim 18{,}750$ ステップ）訓練する。

```
history_img = train_gan(net_D_img, net_G_img, image_iter, num_epochs=20,
                        latent_dim=100, lr_D=2e-4, lr_G=2e-4, seed=1)
print(f'last epoch: critic loss {history_img[-1, 0]:.2f}, '
      f'generator loss {history_img[-1, 1]:.2f}')
```

```
last epoch: critic loss 1.05, generator loss 1.37
```

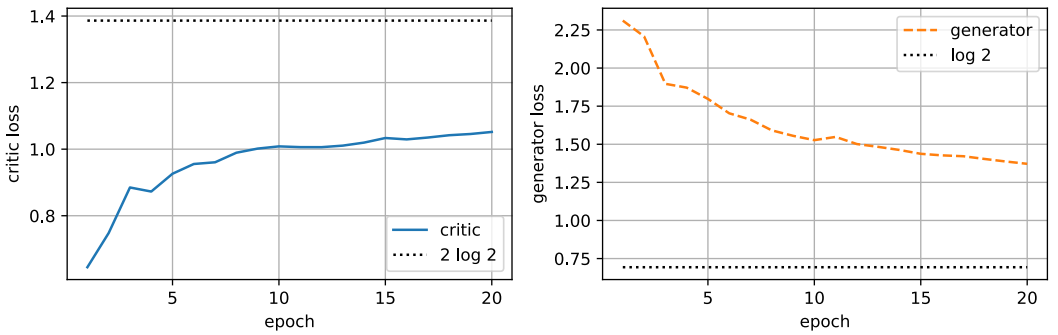
訓練後、30 個の潜在ベクトルを $\mathcal{N}(0, I_{100})$ から抽出し、`net_G_img` に通して出力を 28×28 の画像に変形し、生成された衣類を確認する。

```
with torch.no_grad():
    fake_img = net_G_img(torch.randn(30, 100))
d2l.show_images(fake_img.reshape(30, 28, 28), 2, 15, scale=0.9, cmap='gray');
```



また、history_img に記録されたエポックごとのクリティック損失と生成器損失を、基準水準である $2\log 2$ と $\log 2$ に対してプロットする。

```
epochs = list(range(1, len(history_img) + 1))
fig, axes = d2l.plt.subplots(1, 2, figsize=(9, 3), sharex=True)
for ax, loss, ref, name, ref_name, fmt in zip(
        axes, history_img.T, (2 * math.log(2), math.log(2)),
        ('critic', 'generator'), ('2 log 2', 'log 2'), ('C0-', 'C1--')):
    d2l.plot(epochs, [loss, [ref] * len(epochs)], 'epoch', f'{name} loss',
             legend=[name, ref_name], fmts=(fmt, 'k:'), axes=ax)
fig.tight_layout()
```



生成された画像は、MLP には畳み込みのような空間的帰納バイアス（inductive bias）がないためざらついたピクセルノイズを伴うものの、識別可能な衣類のシルエット（シャツ、ズボン、靴、バッグ）を捉えている。なお、20 エポック経過した後も、識別器損失（~ 1.05）は均衡値 $2\log 2 \approx 1.39$ を下回り続け、生成器損失（~ 1.4）は $\log 2 \approx 0.69$ を上回り続けている。2 次元ガウス分布の設定とは異なり、画像における損失曲線だけでは、不完全な生成と識別器の過学習（overfitting）を区別することも（演習問題 6）、すべての衣類カテゴリが均等にカバーされているかを明らかにすることもできない（演習問題 4）。

16.1.6 まとめ

敵対的生成ネットワークは、陰的生成器 $x' = G(z)$ に識別器（またはクリティック）D を組み合わせ、実データ $x \sim p$ と生成サンプル $x' \sim q$ を分離させることで、明示的な尤度を用いずに生成器を訓練する。二値対数損失のもとでは、最適識別器のロジットは対数密度比 $D^\star(x) = \lambda(x) = \log(p(x)/q(x))$ を復元し、最適価値関数は $\max_D V(D) = 2\,\mathrm{JS}(p,q) - 2\log 2$ に帰着する。ここで $\mathrm{JS}(p,q) = I(x;y) \in [0, \log 2]$ は、$m = (p+q)/2$ からのサンプルとその出所ラベルの間の相互情報量である。生成サンプルが確信をもって棄却されるとき（$D(x') \ll 0$）、文字どおりのミニマックス生成器の重み $w_{\mathrm{sat}}(x') = \sigma(D(x'))$ は指数関数的に消失するのに対し、非飽和生成器損失 $-\mathbb{E}[\log \sigma(D(G(z)))]$ は 1 に近い重み $w_{\mathrm{ns}}(x') = \sigma(-D(x')) \approx 1$ を維持する。最適識別器に対して非飽和損失は $2\,D_{\mathrm{KL}}(m \,\|\, p)$ を下り、データの台から外れたサンプルには厳しいペナルティを課す一方、欠落したモードには有界なペナルティしか課さない。

16.1.7 演習問題

1. **過剰損失と加法定数。** $\eta = p/(p+q)$ とし、$D_{\mathrm{b}}(a \,\|\, u)$ をパラメータ a と u を持つベルヌーイ分布間の KL ダイバージェンスとする。

1. すべてのクリティックが $V(D^\star) - V(D) = 2\,\mathbb{E}_m\big[D_{\mathrm{b}}(\eta \,\|\, \sigma(D))\big]$ を満たし、したがって $D^\star$ が Equation 16.2 の m 零集合を除いて唯一の最大点であることを示せ。
2. これより、$V(D^\star + b) < V(D^\star)$ が任意の定数 $b \neq 0$ に対して成り立つことを導け。
3. b について 2 次まで展開し、

$$V(D^\star + b) - V(D^\star) = -\frac{b^2}{2} \int \frac{pq}{p+q}\, dx + O(b^3),$$

であり、その係数が $[0, \tfrac{1}{2}]$ に収まることを示せ。p と q がほぼ互いに素であるとき、有限個のサンプルは D の加法定数をどの程度強く拘束するか。

2. **測度 0 の台。**
   1. 生成器 $G(z) = zv + \mu_q$（1 次元の潜在変数 $z \in \mathbb{R}$）は、そのサンプルを $\mathbb{R}^2$ 内の直線上に配置する。$\mathrm{JS}(p, q) = \log 2$ が、2 次元ガウスデータ分布 $p = \mathcal{N}(\mu_p, A^\top A)$ に対して任意の v と μ_q のもとで成り立つことを示せ。
   2. $\mathrm{JS}(\hat{p}, q) = \log 2$ が、1,000 個の訓練点の経験分布 $\hat{p}$ と任意の連続密度 q（$q = p$ を含む）に対して成り立つことを示せ。
   3. κ リプシッツなクリティック D に対して、カントロヴィッチ・ルビンシュタイン双対性（37.4.2 節）を用い、Equation 16.2 において p を $\hat{p}$ に置き換えても $V(D)$ の変化は高々 $\kappa\, W_1(\hat{p}, p)$ であることを示せ。リプシッツ定数を制限すると、なぜ経験サンプル上での訓練が有益な情報をもたらすようになるのか。

3. [code] **飽和するクリティック活性化関数。●** `make_nets` 内の Leaky ReLU 活性化関数を tanh ユニットに置き換え、隠れ層の幅を 5 と 3 にせよ。
   1. 16.1.4.5 節の非飽和型のアームを 5 つのシードで再実行し、最終的な $D_{\mathrm{KL}}(q \,\|\, p)$ とクリティック損失を記録せよ。
   2. 停滞した実行について、実サンプルおよび生成サンプルにおける隠れ層の活性化値と $\|\nabla_x D(x)\|$ を調べよ。クリティックの隠れユニットが飽和すると、なぜ非飽和生成器損失でも回復できなくなるのか。
   3. 収束した生成器（$q = p$）と定数クリティック（$D(x) = 0$）はいずれもクリティック損失 $2\log 2$ を与える。本節のどの診断手法によって両者を区別できるか。

4. [code] **Fashion-MNIST 生成器のクラス被覆。●**
   1. Fashion-MNIST 上で LeNet（6.6 節）を訓練し、`net_G_img` からの 1,000 個のサンプルを分類して、予測クラスのヒストグラムを一様なデータ分布と比較せよ。
   2. 5 エポックおよび 40 エポック訓練した生成器に対して同様の実験を繰り返せ。訓練を長くすると、クラス分布はより一様になるか。
   3. 生成画像の一部がぼやけていたり曖昧だったりするとき、10 クラスへの強制的な分類はヒストグラムにどのようなバイアスを生じさせるか。得られた結果をモード欠落（16.1.3.3 節）と関連付けて論じよ。

5. **生成器の重みと対応するダイバージェンス。●●** 生成器損失 $\mathbb{E}_z[g(D(G(z)))]$（クリティック D は固定）は、勾配 $-\mathbb{E}_z[w(D)\,\nabla_\theta D(G(z))]$（ただし $w = -g'$）を持つ。D が最適対数密度比 $\lambda = \log(p/q_{\theta_0})$（現在の生成器 q_{θ_0} に対するもの）に保たれていると仮定し、$\rho = e^D = p/q_{\theta_0}$ とおく。
   1. $\int \nabla_\theta q_\theta = 0$ を用い、重みなし損失 $\mathbb{E}_z[-D(G(z))]$（$w = 1$）の勾配が $\nabla_\theta D_{\mathrm{KL}}(q_\theta \,\|\, p)$（$\theta = \theta_0$ において）となることを示せ。

2. f ダイバージェンス $D_f(p \,\|\, q_\theta) = \mathbb{E}_{q_\theta}[f(p/q_\theta)]$（37.3.1.2 節）に対して $\nabla_\theta D_f\big|_{\theta_0} = -\mathbb{E}_z\big[\rho^2 f''(\rho)\,\nabla_\theta D(G(z))\big]$ を示し、したがって重み関数 w が、f ダイバージェンスのうち $\rho^2 f''(\rho) = w(\log \rho)$ を満たすものに対応することを示せ。
3. w_{sat}、$w = 1$、$w(t) = e^t$ (Goodfellow 2016)、および w_{ns} に対応するダイバージェンスをそれぞれ特定せよ（Arjovsky and Bottou (2017) の定理 2.5）。
4. Equation 16.7 を用いて、$2\,D_{\text{KL}}(m \,\|\, p)$ の値（p が互いに素で等重みな K 個のモードを持ち、q がそのうち 1 つだけをカバーする場合）を求め、K に対する増大の仕方を $D_{\text{KL}}(q \,\|\, p) = \log K$ と比較せよ。

Stefano Ermon と Aditya Grover による CS236: Deep Generative Models を改変。

6. [extended] **有限サンプルとクリティックの容量。●●**標本サイズがクリティックのパラメータ数を $\tilde{O}(1/\epsilon^2)$ 倍上回るとき、経験クリティック値は母集団での値から ϵ 以内に集中する (Arora et al. 2017)。生成器を真のデータ分布に固定する（$G(z) = zA + \mu_p$、すなわち $q = p$）。

   1. 幅 4、16、128、1,024 を持つ隠れ層 2 層のクリティックを、$n \in \{100, 1000, 10000\}$ 個の p からの固定訓練点上で訓練せよ（学習率 10^{-3} の Adam、バッチサイズ 256 で 4,000 ステップ）。訓練損失と新規サンプル上の母集団損失との間の汎化ギャップを、パラメータ数に対してプロットせよ。
   2. 16.1.5 節の Fashion-MNIST クリティックは $\sim$ 530,000 個のパラメータを持ち、60,000 枚の画像で訓練されている。Fashion-MNIST のテストセットを用いた検証を設計・実行し、その $2\log 2$ を下回る損失が訓練セットの暗記を反映しているのか、それとも p と q の真の乖離を反映しているのかを確かめよ。

16.2 敵対的目的関数とダイバージェンス

多くの GAN 実装には、差し替え可能な複数の訓練目的関数が用意されている。非飽和ロジスティック損失、最小二乗法（LSGAN）、ヒンジ損失、ワッサースタイン損失などである。16.1 節ではロジスティック目的関数を解析し、最適な識別器が対数密度比 $\lambda(x) = \log(p(x)/q(x))$ を復元すること、そしてそのミニマックス値がイェンセン・シャノン・ダイバージェンス（p と q のサポートが重ならなくなると飽和する量）になることを示した。では、他の目的関数はどのようなダイバージェンスを測っているのだろうか。また、生成サンプルがデータから遠く離れているとき、勾配消失の原因は損失関数、クリティックのアーキテクチャ、背後にあるダイバージェンスのどれにあるのだろうか。

この族に属する目的関数はすべて、クリティックの出力に適用されるスコア関数か、クリティックが最適化の対象とする関数クラス $\mathcal{T}$ のいずれかを変更したものである。制約なしのクリティックを分類損失と組み合わせると f ダイバージェンスを評価することになり、これは各点での密度比 $p(x)/q(x)$ のみに依存する。これに対し、線形の利得を用いてクリティックを滑らかな関数クラスに制約すると、最大平均値差異（maximum mean discrepancy、MMD）や 1 次ワッサースタイン距離（Wasserstein-1 distance、W_1）などの**積分確率指標**（integral probability metric、IPM）が得られる。これらはサポートが互いに素であっても空間的な距離に対する感度を保つ。以下では、これら両方の族を導出し、離れた 1 次元ガウス分布と 2 次元混合ガウス分布における挙動を比較する。

```
from torch.nn.functional import relu, softplus
```

16.2.1 一般的なテンプレート

16.1 節では、クリティックは $\log\sigma(D(x))$（実サンプル $x\sim p$ に対する値）と $\log\sigma(-D(x'))$（生成サンプル $x'\sim q$ に対する値）を受け取っていた。これらの対数尤度項を一般的な利得関数 ℓ_p と ℓ_q に置き換え、クリティックのクラス $\mathcal{T}$ 上で考えることで、一般的な敵対的乖離度が得られる。

$$d(p,q)=\sup_{T\in\mathcal{T}}\Big\{\mathbb{E}_{x\sim p}\big[\ell_p(T(x))\big]+\mathbb{E}_{x'\sim q}\big[\ell_q(T(x'))\big]\Big\}. \tag{16.9}$$

$\ell_p(t)=\log\sigma(t)$、$\ell_q(t)=\log\sigma(-t)$ とおき、$\mathcal{T}$ に制約を課さない場合、$2\,\mathrm{JS}(p,q)-2\log 2$ が復元される。本節を通じて、クリティックは期待利得の和を最大化する（同値な表現として、負の利得を最小化するものとし、これをクリティック損失と呼ぶ）。

Table 16.2 は、本章で扱う目的関数を 2 つの軸に沿って整理したものである。すなわち、クリティックが個々のサンプルにスコアを付けるか実サンプルと生成サンプルのペアにスコアを付けるかという軸と、クリティックのクラス $\mathcal{T}$ が制約なしか正則化されているかという軸である。

Table 16.2: クリティックの入力と関数クラスの制約による敵対的目的関数の分類。

	制約なしのクリティック	制約付き・ペナルティ付きのクリティック
単一サンプルクリティック	分類損失（ロジスティック、最小二乗法、ヒンジ）$\to f$ ダイバージェンス（比 p/q に基づく）	RKHS 球（MMD）またはリプシッツ球（W_1）上の線形利得、スペクトル正規化付きヒンジ損失
ペアワイズクリティック	相対論的ペアリング損失（RpGAN、16.3 節）	ゼロ中心勾配ペナルティ R_1+R_2 付きペアリング損失（R3GAN、16.4 節、16.5 節）

なお、任意の固定されたクリティック T に対し、Equation 16.9 内の期待値は分布 q に関して線形である。線形汎関数の族の上限は凸であるため、$d(p,q)$ は確率測度 q に関して（p を固定したとき）常に凸となる。しかし、分布の集合 $\{q_\theta\}$（ニューラルネットワーク G_θ によってパラメータ化されるもの）は凸集合ではないため、分布空間における凸性から生成器パラメータ θ に関する凸性は導かれない（演習問題 1）。

16.2.2 分類損失と f ダイバージェンス

16.2.2.1 2 値結果の例

表と裏の 2 通りの結果からなる空間を考え、データ分布 $p=(\frac{3}{4},\frac{1}{4})$ と生成器分布 $q=(\frac{1}{4},\frac{3}{4})$ から等しい事前確率 $\frac{1}{2}$ でサンプリングするものとする。結果が与えられたとき、それが p から生じたベイズ事後確率は $\eta=p/(p+q)$ であり、表なら $\frac{3}{4}$、裏なら $\frac{1}{4}$ となる。結果を見ない当て推量の予測（$D=0$）は、結果にかかわらず $\frac{1}{2}$ を予測する。結果を観測することによる期待損失の減少量は、分類損失に応じて次のように異なる。

- **対数損失**：最適な予測 η は二値エントロピー $H_b(\frac{3}{4})=0.562$ nat を達成し、当て推量の $\log 2=0.693$ nat と比べて 0.131 nats $=\mathrm{JS}(p,q)$ だけ損失を減らす。

- 確率に対する二**乗誤差**：η を予測すると期待損失は $\frac{3}{4}\cdot\frac{1}{4}=\frac{3}{16}$ となり、損失 $\frac{1}{4}$（$\frac{1}{2}$ を予測する場合）と比べて $\frac{1}{16}$ だけ減少する。
- スコア t に対する**ヒンジ損失**：最適なスコア $t=+1$（表）と $t=-1$（裏）は期待損失 $2\cdot\frac{1}{4}=\frac{1}{2}$ を達成し、損失 1（$t=0$ の場合）と比べて $\frac{1}{2}=\mathrm{TV}(p,q)=\frac{1}{2}\sum|p-q|$ だけ減少する。

いずれの損失も、同じ組 (p,q) に対して同じ事後確率 η に基づきながら、異なる非負のダイバージェンスを割り当てている。以下では、マージンに基づくすべての分類損失が f ダイバージェンスを定めることを示そう。

16.2.2.2 ベイズリスクギャップ

対称な分類損失は、単一のマージン利得 ℓ を実サンプル上の $D(x)$ と生成サンプル上の $-D(x')$ に適用する（$\ell_p(t)=\ell(t)$、$\ell_q(t)=\ell(-t)$）。

$$V_\ell(D)\;=\;\mathbb{E}_{x\sim p}\big[\ell(D(x))\big]+\mathbb{E}_{x'\sim q}\big[\ell(-D(x'))\big]. \tag{16.10}$$

ここで符号付きスコア t（実データでは $+D(x)$、生成データでは $-D(x')$）は分類の**マージン**である。各点 x において、$p(x)\ell(t)+q(x)\ell(-t)=2m(x)\big[\eta(x)\ell(t)+(1-\eta(x))\ell(-t)\big]$ が成り立つ。ただし、$m=(p+q)/2$ および $\eta(x)=p(x)/(p(x)+q(x))$ である。事後確率 η における各点での最小期待損失を**条件付きベイズリスク**（conditional Bayes risk）という。

$$L(\eta)\;=\;\inf_{t}\Big\{\eta\,(-\ell(t))+(1-\eta)\,(-\ell(-t))\Big\}. \tag{16.11}$$

η のアフィン関数の下限として定義されるため、$L(\eta)$ は $[0,1]$ 上で凹関数となる。定数クリティック（$t=0$）はリスク $L(\frac{1}{2})$ を達成する。x を観測することによるリスクの減少量が**ベイズリスクギャップ**である。

$$\Delta_\ell(p,q)\;=\;L(\tfrac{1}{2})-\mathbb{E}_{x\sim m}\big[L(\eta(x))\big], \tag{16.12}$$

これにより最適値 $\sup_D V_\ell(D)\;=\;2\Delta_\ell(p,q)\;-\;2L(\frac{1}{2})$ が決まる。イェンセンの不等式と $\mathbb{E}_m[\eta]=\frac{1}{2}$ より、$\Delta_\ell(p,q)\ge 0$ が成り立つ。

16.2.2.3 f ダイバージェンスとしてのベイズリスクギャップ

37.3.1.2 節で見たように、f ダイバージェンスは凸関数 f（ただし $f(1)=0$）を密度比 $\rho(x)=p(x)/q(x)$ に適用し、q のもとで平均した量 $D_f(p\,\|\,q)=\mathbb{E}_{x\sim q}[f(\rho(x))]$ であった。

命題。 条件付きベイズリスク L が $[0,1]$ 上で有限ならば、次が成り立つ。

$$\begin{aligned}&\Delta_\ell(p,q)=D_f(p\,\|\,q)\\ &\text{with}\quad f(u)=L(\tfrac{1}{2})-\frac{u+1}{2}\,L\!\left(\frac{u}{u+1}\right),\end{aligned}$$

ここで、f は $[0,\infty)$ 上で凸関数であり、$f(1)=0$ を満たす。

証明。 $u=\rho=p/q$ とおくと、$m=q\,(u+1)/2$ および $\eta=u/(u+1)$ となるので、次を得る。

$$\mathbb{E}_{x\sim m}[L(\eta(x))] = \int q\,\frac{u+1}{2}\,L\!\left(\frac{u}{u+1}\right),$$

さらに $L(\tfrac{1}{2}) = \int q\,L(\tfrac{1}{2})$ を積分の内側に入れると、$\Delta_\ell(p,q) = \mathbb{E}_q[f(\rho)]$ が得られる。凹関数 $L(v) = \inf_j\{\alpha_j v + \beta_j\}$ をその接線の下側包絡線として表せば、$(u+1)L(u/(u+1)) = \inf_j\{\alpha_j u + \beta_j(u+1)\}$ が u に関して凹であることがわかる。したがって $f(u)$ は凸であり、$f(1) = L(\tfrac{1}{2}) - L(\tfrac{1}{2}) = 0$ となる。■

以上より、制約なしのクリティックと対称な分類損失の組み合わせはすべて f ダイバージェンスを評価し、その最適クリティック $D^\star(x)$ は対数密度比 $\lambda(x) = \log(p(x)/q(x))$ の単調な**リンク関数**（link function）となる。

16.2.2.4 代表的な分類損失のダイバージェンス

Table 16.3 に、代表的な 4 つの分類損失と、それぞれの条件付きベイズリスク $L(\eta)$、ベイズリスクギャップ $\Delta_\ell(p,q)$、最適クリティックのリンク $D^\star(\lambda)$、および実用上それらと組み合わせて用いられる生成器損失をまとめる。

Table 16.3: 制約なしのクリティックにおける分類損失。マージン損失 $-\ell(t)$、ベイズリスク $L(\eta)$、ベイズリスクギャップ $\Delta_\ell(p,q)$、最適クリティックのリンク $D^\star(\lambda)$、および $D = D(x')$ に対する実用上の生成器損失。

損失	マージン損失 $-\ell(t)$	ベイズリスク $L(\eta)$	ギャップ $\Delta_\ell(p,q)$	最適クリティックのリンク	生成器損失
ロジスティック	$\log(1+e^{-t})$	$H_b(\eta)$	$\mathrm{JS}(p,q)$	λ	$-\log\sigma(D)$
最小二乗法（LSGAN）	$\tfrac{1}{4}(1-t)^2$、$t = 2D-1$	$\eta(1-\eta)$	$\tfrac{1}{8}\int \frac{(p-q)^2}{p+q}$	$\sigma(\lambda)$	$\tfrac{1}{2}(D-1)^2$
二乗ヒンジ	$\max(0,1-t)^2$	$4\eta(1-\eta)$	$\tfrac{1}{2}\int \frac{(p-q)^2}{p+q}$	$\tanh(\lambda/2)$	$-D$
ヒンジ	$\max(0,1-t)$	$2\min(\eta,1-\eta)$	$\mathrm{TV}(p,q)$	$\operatorname{sign}\lambda$	$-D$

LSGAN では、クリティックはターゲット 1（本物）と 0（偽物）に回帰する (Mao et al. 2017)。これは $t = 2D-1$ のもとで二乗マージン損失 $\tfrac{1}{4}(1-t)^2$ に対応し、Vincze–Le Cam ダイバージェンス $\chi^2(p\,\|\,m) = \tfrac{1}{2}\int (p-q)^2/(p+q)$ の 4 分の 1 を評価する。ヒンジ損失は全変動 $\mathrm{TV}(p,q) = \tfrac{1}{2}\int |p-q|$ を評価し (Lim and Ye 2017)、その最適クリティックは $\operatorname{sign}\lambda$ を出力する（$[-1,1]$ 内の任意の値が $\eta = \tfrac{1}{2}$ において最適となる）。二乗ヒンジ損失はヒンジマージンおよび生成器目的関数 $-D(x')$ を共有するが、滑らかなリンク $\tanh(\lambda/2)$ によって λ の大きさを保持する。実用上は、いずれの損失も平衡点 $q = p$ を共有する非飽和型の生成器目的関数（最後の列）を用いる。

16.2.2.5 フェンシェル・ヤング変分下界（f-GAN）

Equation 16.10 は D と $-D$ に同じ利得を適用するため、そのベイズリスクは $L(\eta) = L(1-\eta)$ を満たす。したがって、すべての分類ギャップ $\Delta_\ell(p,q)$ は p と q に関して対称であり、上から $L(\tfrac{1}{2})$ で抑えられる（演習問題 5）。非対称または非有界な f ダイバージェンス（順方向 KL ダイバージェンス $D_{\mathrm{KL}}(p\,\|\,q)$ など）を扱うには、非対称な利得 $\ell_p \neq \ell_q$ が必要になる。

$f^*(t) = \sup_u\{ut - f(u)\}$ を凸ダイバージェンス関数 f の**凸共役**（convex conjugate）とする。フェンシェル・ヤングの不等式より、$f(u) \geq ut - f^*(t)$ がすべての u, t について成り立ち、等

号は $t = f'(u)$ のときに成立する。$u = \rho(x) = p(x)/q(x)$、$t = T(x)$ とおいて q のもとで期待値をとると（$\mathbb{E}_q[\rho\, T] = \mathbb{E}_p[T]$ を用いる）、f-GAN の変分下界が得られる (Nowozin et al. 2016)（37.3.4 節）。

$$D_f(p \,\|\, q) \;\geq\; \mathbb{E}_{x\sim p}\big[T(x)\big] - \mathbb{E}_{x'\sim q}\big[f^*(T(x'))\big]. \tag{16.13}$$

これはテンプレート Equation 16.9 において $\ell_p(t) = t$、$\ell_q(t) = -f^*(t)$ とした場合に一致し、最適クリティック $T^\star(x) = f'(\rho(x))$ で等号を達成する（演習問題 2）。順方向 KL ダイバージェンスでは、$f(u) = u\log u$ の共役は $f^*(t) = e^{t-1}$、最適クリティックは $T^\star(x) = 1 + \log\rho(x) = 1 + \lambda(x)$ となる。

16.2.2.6 クリティックのリンク関数の実験的検証

ここで、ロジスティック、最小二乗、ヒンジ、順方向 KL の各目的関数のもとで訓練したクリティックが、$\lambda(x)$ に関するそれぞれの理論的リンク関数（λ、$\sigma(\lambda)$、$\operatorname{sign}\lambda$、$1+\lambda$）を学習することを検証しよう。$p$ を 3 つの 2 次元ガウス分布の一様混合として定義し、摂動を加えた 3 成分の混合分布 q を構成することで、いたるところで $\lambda(x)$ が解析的に求まるようにする。

```
mu_p = np.array([[0.0, 2.4], [-2.1, -1.2], [2.1, -1.2]])     # data
A_p = np.stack([0.7 * np.eye(2)] * 3)

mu_q = np.array([[0.2, 2.6], [-2.3, -1.0], [2.0, -1.3]])     # hand-set q
A_q = np.stack([np.diag([0.8, 0.55]), np.diag([0.5, 0.6]),
                np.array([[0.65, 0.2], [-0.1, 0.7]])])


def mixture_logpdf(x, mus, As):
    """Log density of the uniform mixture of Gaussians N(mu_k, A_k^T A_k)."""
    comps = []
    for mu, A in zip(mus, As):
        Sigma, diff = A.T @ A, x - mu
        sol = np.linalg.solve(Sigma, diff.T).T
        comps.append(-0.5 * ((diff * sol).sum(1)
                             + np.linalg.slogdet(2 * np.pi * Sigma)[1]))
    return np.logaddexp.reduce(np.stack(comps) - np.log(len(mus)), axis=0)


def sample_mixture(n, mus, As, rng):
    """n draws from the uniform mixture of Gaussians mu_k + z A_k."""
    k = rng.integers(0, len(mus), n)
    z = rng.standard_normal((n, 2))
    return np.einsum('ni,nij->nj', z, As[k]) + mus[k]
```

固定グリッド上の数値積分により、$\mathrm{JS}(p, q)$ がこの形式の任意の混合分布 q に対して求まる。本節の後半では、同じ関数を用いて訓練中の生成器をステップごとにスコア付けする。

```
grid_1d = np.linspace(-7.0, 7.0, 281)
grid_2d = np.stack([g.ravel() for g in np.meshgrid(grid_1d, grid_1d)], axis=1)
cell_area = (grid_1d[1] - grid_1d[0]) ** 2
```

```
def js_mixtures(mus_q, As_q, mus_p=mu_p, As_p=A_p):
    """JS(p, q) between two known mixtures, p the data by default, by
    quadrature on grid_2d."""
    logp = mixture_logpdf(grid_2d, mus_p, As_p)
    logq = mixture_logpdf(grid_2d, mus_q, As_q)
    p, q = np.exp(logp), np.exp(logq)
    logm = np.log(np.maximum((p + q) / 2, 1e-300))
    return 0.5 * (p * (logp - logm) + q * (logq - logm)).sum() * cell_area

print(f'JS(p, q) for the hand-set q: {js_mixtures(mu_q, A_q):.3f} nats')
```

```
JS(p, q) for the hand-set q: 0.033 nats
```

手動で設定したこの q はデータから 100 分の数 nat だけ離れており、これは生成器が訓練の終盤で直面する程度のずれである。クリティックには、ユニット数 64 の隠れ層を 2 つもつ小規模な Leaky ReLU ネットワークを用いる。

```
TRUNC_STD = 0.8796  # standard deviation of N(0, 1) truncated at +-2


def make_critic():
    """Leaky-ReLU MLP with LeCun-normal weights and zero biases."""
    D = nn.Sequential(nn.Linear(2, 64), nn.LeakyReLU(0.2),
                      nn.Linear(64, 64), nn.LeakyReLU(0.2), nn.Linear(64, 1))
    for layer in D[::2]:  # weight std 1/sqrt(fan_in) after truncation
        s = layer.in_features ** -0.5 / TRUNC_STD
        nn.init.trunc_normal_(layer.weight, std=s, a=-2 * s, b=2 * s)
        nn.init.zeros_(layer.bias)
    return D
```

各損失は、Table 16.3 に示したクリティックの損失と生成器の損失のペアとして格納し、いずれも実バッチに対するクリティックのスコア `r` と生成バッチに対するスコア `f` の関数として記述する。

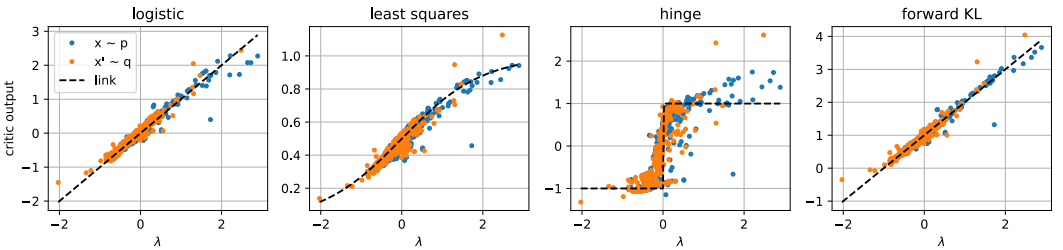
```
losses = {  # name: (critic loss, generator loss), both on the scores r, f
    'logistic': (lambda r, f: (softplus(-r) + softplus(f)).mean(),
                 lambda r, f: softplus(-f).mean()),
    'least squares': (lambda r, f: ((r - 1) ** 2 + f ** 2).mean() / 2,
                      lambda r, f: ((f - 1) ** 2).mean() / 2),
    'hinge': (lambda r, f: (relu(1 - r) + relu(1 + f)).mean(),
              lambda r, f: -f.mean()),
    'squared hinge': (lambda r, f: (relu(1 - r) ** 2 + relu(1 + f) ** 2).mean(),
                      lambda r, f: -f.mean()),
}
```

リンク関数は固定された q に対する最適応答を表すため、手動設定した q に対して 3 つの損失のもとでそれぞれ新しいクリティックを訓練し、4 つ目のクリティックは順方向 KL 下界 $\mathbb{E}_p[T] - \mathbb{E}_q[e^{T-1}]$ を最大化するように訓練する。

```
to_t = lambda a: torch.tensor(a, dtype=torch.float32)

def best_response(critic_loss, seed=0, steps=3000, batch=512, lr=1e-3):
    """Fresh critic trained against the fixed q, as a NumPy function."""
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    D = make_critic()
    opt = torch.optim.Adam(D.parameters(), lr)
    for _ in range(steps):
        X = to_t(sample_mixture(batch, mu_p, A_p, rng))
        Y = to_t(sample_mixture(batch, mu_q, A_q, rng))
        opt.zero_grad()
        critic_loss(D(X), D(Y)).backward()
        opt.step()
    return lambda x: D(to_t(x)).squeeze(-1).detach().numpy()

kl_dual = lambda r, f: -(r.mean() - torch.exp(f - 1).mean())
critics = {name: best_response(losses[name][0])
           for name in ('logistic', 'least squares', 'hinge')}
critics['forward KL'] = best_response(kl_dual)
```

訓練した各クリティックを p からの 300 点と q からの 300 点で評価し、真の対数密度比 $\lambda(x)$ に対してその出力を理論的リンク関数（破線）とともにプロットする。

```
rng = np.random.default_rng(7)
pts = np.concatenate([sample_mixture(300, mu_p, A_p, rng),
                      sample_mixture(300, mu_q, A_q, rng)])
lam = mixture_logpdf(pts, mu_p, A_p) - mixture_logpdf(pts, mu_q, A_q)
lam_line = np.linspace(lam.min(), lam.max(), 200)
links = {'logistic': lambda l: l, 'least squares': lambda l: 1 / (1 + np.exp(-l)),
         'hinge': np.sign, 'forward KL': lambda l: 1 + l}
fig, axes = d2l.plt.subplots(1, 4, figsize=(12, 3))
for ax, (name, link) in zip(axes, links.items()):
    out = critics[name](pts)
    d2l.plot([lam[:300], lam[300:], lam_line],
             [out[:300], out[300:], link(lam_line)], xlabel=r'$\lambda$',
             fmts=('.', '.', 'k--'), axes=ax,
             legend=['x ~ p', "x' ~ q", 'link'] if name == 'logistic' else None)
    ax.set_title(name)
axes[0].set_ylabel('critic output')
fig.tight_layout()
```



4 つのネットワークはいずれも $\lambda(x)$ の理論的リンク関数、すなわち恒等関数 λ、シグモイド

$\sigma(\lambda)$、階段関数 sign λ、平行移動した恒等関数 $1+\lambda$ によく追従している。なお、ヒンジクリティックでは $\lambda = 0$ 付近でばらつきが大きくなっているが、これは $[-1, 1]$ 内のどの出力も最小条件付きリスクを達成するためである。

16.2.3 積分確率指標

f ダイバージェンスが制約なしのクリティックと非線形な利得を組み合わせるのに対し、**積分確率指標**（IPM）は線形の利得 $\ell_p(t) = t$ と $\ell_q(t) = -t$ を用い、代わりにクリティックを対称な関数クラス $\mathcal{F}$ に制限する（37.4.1 節）。

$$d_{\mathcal{F}}(p, q) \;=\; \sup_{T \in \mathcal{F}} \Big\{ \mathbb{E}_{x \sim p}[T(x)] - \mathbb{E}_{x' \sim q}[T(x')] \Big\}. \tag{16.14}$$

$\mathcal{F}$ に制約がなければ、T を任意の正定数でスケールすることで、期待値の差は $+\infty$ へ発散してしまう（$p \neq q$ である限り）。$\mathcal{F}$ が滑らかな関数からなるとき、$d_{\mathcal{F}}(p, q)$ は $\mathcal{X}$ の空間的な幾何構造に依存し、単なる各点での比 $p(x)/q(x)$ だけでは決まらない。

16.2.3.1 最大平均値差異（MMD）

正定値カーネル $k(x, x')$（ガウス RBF カーネル $k(x, x') = \exp(-\|x - x'\|^2 / (2s^2))$、バンド幅 s など）が与えられたとき、$\mathcal{F}$ として k に対応する再生核ヒルベルト空間（reproducing kernel Hilbert space、RKHS）の単位球を選ぶと、**最大平均値差異**が得られる (Gretton et al. 2012)（37.4.1 節）。その二乗値は、ペアごとのカーネル期待値を用いて閉形式で表せる。

$$\begin{aligned} \mathrm{MMD}^2(p, q) = \mathbb{E}_{x, \tilde{x} \sim p}[k(x, \tilde{x})] + \mathbb{E}_{x', \tilde{x}' \sim q}[k(x', \tilde{x}')] \\ - 2\, \mathbb{E}_{x \sim p,\, x' \sim q}[k(x, x')]. \end{aligned} \tag{16.15}$$

（定数倍を除いて）最適なテスト関数は**証人関数**（witness function）$w(x) = \mathbb{E}_{\tilde{x} \sim p}[k(x, \tilde{x})] - \mathbb{E}_{\tilde{x}' \sim q}[k(x, \tilde{x}')]$ であり、ある点と p との平均類似度から q との平均類似度を引いた差を測る。Equation 16.15 はクリティックのネットワークを必要としないため、生成器は有限標本の不偏推定量 `mmd2_unbiased` を直接最小化できる（この関数は 16.5 節で KID の計算にも用いる）。

```python
def mmd2_unbiased(K_pp, K_qq, K_pq):
    """Unbiased MMD^2 from the kernel matrices within p, within q, and
    across; the diagonals (a point with itself) are left out."""
    m, n = len(K_pp), len(K_qq)
    return ((K_pp.sum() - K_pp.diagonal().sum()) / (m * (m - 1))
            + (K_qq.sum() - K_qq.diagonal().sum()) / (n * (n - 1))
            - 2 * K_pq.mean())
```

16.2.3.2 1 次ワッサースタイン距離（W_1）

$\mathcal{F}$ として **1 リプシッツ**関数のクラス（$|T(x) - T(y)| \le \|x - y\|$ がすべての x, y について成り立つもの）を選ぶと、カントロヴィチ・ルビンシュタイン双対性（37.4.2 節）により **1 次ワッサースタイン**距離（または**アースムーバー**距離）が得られる。

$$W_1(p, q) \;=\; \inf_{\pi \in \Pi(p, q)} \mathbb{E}_{(x, x') \sim \pi}\big[\|x - x'\|\big],$$

ここで、$\Pi(p,q)$ は周辺分布が p と q である同時輸送計画の集合である。最適な 1 リプシッツクリティック（**カントロヴィチポテンシャル**）は、q から p へ向かう最適輸送の半直線に沿って単位勾配ノルムをもつ。1 次元では $W_1(p,q) = \int_{-\infty}^{\infty} |F_p(t) - F_q(t)|\, dt$ となるため、分布を μ だけ平行移動すると $W_1(p,q) = |\mu|$ となり、どれだけ離れていても一定の傾き 1 をもつ。

より高い次元では、Wasserstein GAN（WGAN）はニューラルネットワークのクリティック T に対する 1 リプシッツ制約を主に次の 3 つの仕組みで近似する。

1. **重みクリッピング**（weight clipping）は、重みを $[-c, c]$ にクランプし、表現容量を制限する引き換えとして κ リプシッツ性（κ はアーキテクチャに依存する定数）を強制する (Arjovsky et al. 2017)。
2. **勾配ペナルティ**（gradient penalty、WGAN-GP）は、$\gamma\, \mathbb{E}[(\|\nabla_x T(\tilde{x})\| - 1)^2]$ を実サンプルと生成サンプルのランダムな内分点 $\tilde{x} = ux + (1-u)x'$ において加算する (Gulrajani et al. 2017)。
3. **スペクトル正規化**（spectral normalization）は、各重み行列をその最大特異値 $\sigma_{\max}(W)$ で正規化し、すべての線形層のリプシッツ定数を 1 以下に抑える (Miyato et al. 2018)。

なお、ヒンジ損失はこれら 2 つの族を橋渡ししている。$|D(x)| \le 1$ の範囲では利得が線形であるため、制約なしのヒンジクリティックは $|D(x)| \le 1$ で有界な関数クラス上の IPM に対応する（評価値は $2\,\mathrm{TV}(p,q) - 2$）。ヒンジ損失をスペクトル正規化と組み合わせると（SAGAN や BigGAN のように、D を有界かつ κ リプシッツにすると）、その値が κW_1 のように分離距離 $2/\kappa$ まで線形に増大し、その先で飽和するクリティックが得られる。

16.2.3.3 高次元におけるサンプル複雑度

固定カーネルの MMD と経験 W_1 はサンプル複雑度が大きく異なる。MMD^2 の不偏推定量は次元に依存しないレート $O(n^{-1/2})$ で収束する (Gretton et al. 2012)（ただし推定対象である固定カーネル下での所与のシフトに対する MMD^2 自体が、次元の増大に伴って縮小しうる (Ramdas et al. 2015)）。これに対し、プラグイン経験 W_1 距離は次元 $d_x > 2$ において次元の呪いを受け、レート $O(n^{-1/d_x})$ でしか収束しない (Weed and Bach 2019)。一方、変分 f ダイバージェンス推定量は $\rho = p/q$ が大きいときに大きな分散に悩まされることがある。たとえば順方向 KL 双対下界では、$e^{T^\star(x')-1} = \rho(x')$ の q のもとでの分散は $\chi^2(p\,\|\,q)$ に等しくなり、$D_{\mathrm{KL}}(p\,\|\,q) < \infty$ であっても発散しうる（演習問題 4）。

16.2.4 データから遠い領域と近い領域における勾配

16.2.4.1 ダイバージェンスの傾きと分離距離

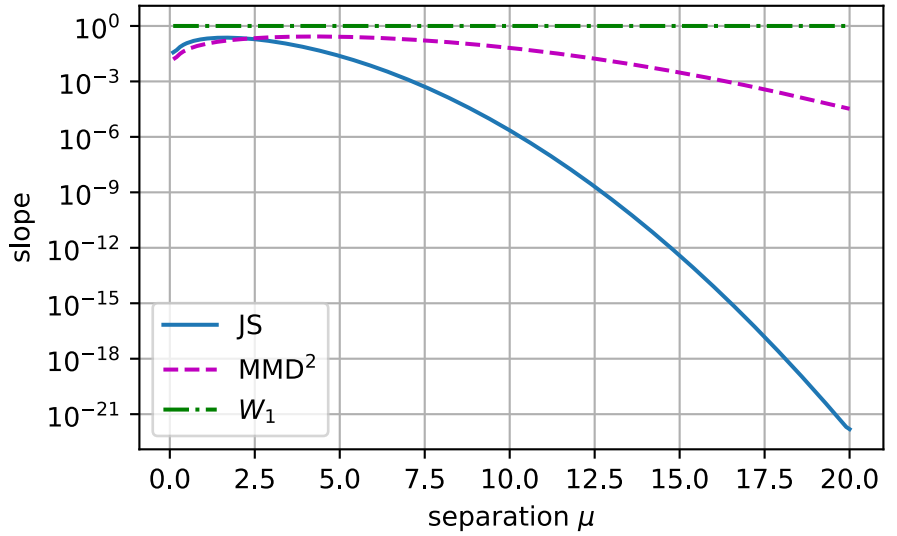
2 つの分布が離れるにつれて母集団ダイバージェンスがどう振る舞うかを見るため、$p = \mathcal{N}(0,1)$ と $q = \mathcal{N}(\mu,1)$ を分離距離 $\mu > 0$ の関数として考えよう。このとき $W_1(p,q) = \mu$ は定数の導関数 1 をもち、$\mathrm{JS}(p,q)$ は数値積分でき、$\mathrm{MMD}^2(p,q)$ はバンド幅 s のガウス RBF カーネルのもとで次の厳密なガウス積分をもつ。

$$\mathrm{MMD}^2(p,q) \;=\; \frac{2s}{\sqrt{s^2+2}}\left(1 - \exp\!\left(-\frac{\mu^2}{2(s^2+2)}\right)\right), \tag{16.16}$$

この式の μ に関する導関数は $\mu = \sqrt{s^2+2}$（≈ 4.2、ただし $s = 4$ の場合）でピークに達し、その後 $e^{-\mu^2/(2(s^2+2))}$ のオーダーで減衰する。

```
xs = np.linspace(-10.0, 30.0, 8001)          # fixed quadrature grid

def normal_pdf(x, mu, sigma=1.0):
    return np.exp(-(x - mu) ** 2 / (2 * sigma ** 2)) / np.sqrt(
        2 * np.pi * sigma ** 2)

def js_gap(p, q):
    """log 2 - JS(p, q) on xs, accurate even when tiny: the integral of
    (p log(1 + q/p) + q log(1 + p/q)) / 2, with log(1 + e^a) computed as
    logaddexp(0, a)."""
    a = np.log(q) - np.log(p)
    return 0.5 * np.trapezoid(p * np.logaddexp(0, a) + q * np.logaddexp(0, -a),
                              xs)

def js_1d(p, q):
    return np.log(2) - js_gap(p, q)

s = 4.0                                      # RBF kernel bandwidth
mus = np.linspace(0.1, 20.0, 200)            # separations mu
gaps = np.array([js_gap(normal_pdf(xs, 0.0), normal_pdf(xs, mu))
                 for mu in mus])
mmd2 = 2 * s / np.sqrt(s ** 2 + 2) * (1 - np.exp(-mus ** 2
                                                 / (2 * (s ** 2 + 2))))
slopes = [-np.gradient(gaps, mus), np.gradient(mmd2, mus),
          np.ones_like(mus)]                 # W1 = |mu| has slope one
for mu in (1.0, 7.0, 20.0):
    js = js_1d(normal_pdf(xs, 0.0), normal_pdf(xs, mu))
    print(f'mu = {mu:2.0f}: JS slope {np.interp(mu, mus, slopes[0]):.4g}, '
          f'JS / log 2 = {js / np.log(2):.4f}, '
          f'MMD^2 slope {np.interp(mu, mus, slopes[1]):.4g}')
peak = slopes[1].argmax()
print(f'MMD^2 slope peaks at mu = {mus[peak]:.1f} with {slopes[1][peak]:.3f}')
d2l.plot(mus, slopes, xlabel=r'separation $\mu$', ylabel='slope',
         legend=['JS', 'MMD$^2$', '$W_1$'], yscale='log', figsize=(5, 3))
```

```
mu =  1: JS slope 0.1987, JS / log 2 = 0.1607, MMD^2 slope 0.1019
mu =  7: JS slope 0.001264, JS / log 2 = 0.9990, MMD^2 slope 0.188
mu = 20: JS slope 1.549e-22, JS / log 2 = 1.0000, MMD^2 slope 3.303e-05
MMD^2 slope peaks at mu = 4.2 with 0.269
```



上のプロットおよび Figure 16.4 に示すように、JS(p, q) の導関数は $\mu = 7$ までに 2 桁以上低下し（0.0013 対 0.20、後者は $\mu = 1$ の値）、1.5×10^{-22}（$\mu = 20$ の値）に達する。対照的に、W_1 はあらゆる距離で傾き 1 を保ち、MMD^2 はカーネルバンド幅 s 程度の分離距離まで強い勾配を保つ。

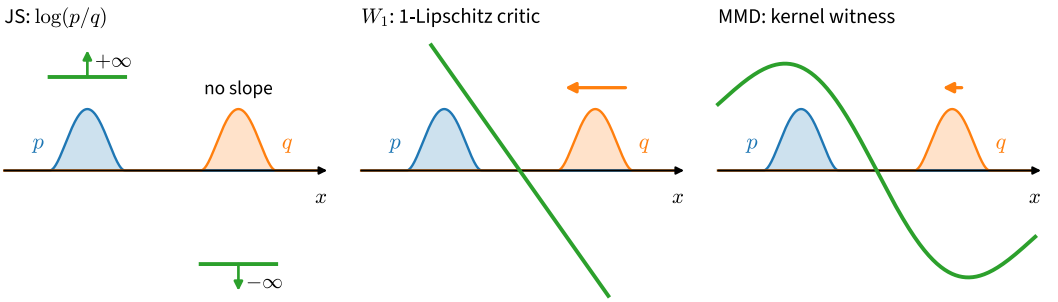


Figure 16.4: p（青）と q（オレンジ）が互いに素なサポートをもつときの最適クリティック。左：JS の最適クリティック $\log(p/q)$ は値 $+\infty$（supp(p) 上）と $-\infty$（supp(q) 上）をとり、両方の領域上で傾きが 0 になる。中央：W_1 のカントロヴィチポテンシャルは傾き 1 の線形ランプである。右：MMD の証人関数 $\mathbb{E}_p[k(x,\cdot)] - \mathbb{E}_q[k(x,\cdot)]$ はカーネルの山同士の滑らかな差であり、その裾は p に向かって傾いている。

16.2.4.2 分離した初期状態から訓練したクリティック

理論上のダイバージェンス曲線は、すべての可測関数にわたる最適なクリティックを仮定している。しかし実際には、クリティックは生成器の更新 1 回あたり数ステップだけ訓練される有限のニューラルネットワークである。離れた初期化から訓練されたクリティックがどう振る舞うかを調べるため、1 次元の位置生成器 $q_\theta = \mathcal{N}(\theta, 1)$（$x' = \theta + z$）を $\theta_0 = 6$ から出発させて $p = \mathcal{N}(0, 1)$ に当てはめる。ここでは θ を素の確率的勾配降下法（SGD）で更新し（ステップサイズが勾配の大きさを直接反映するようにするため）、2 隠れ層の ReLU クリティックを生成器の 1 ステップにつき 5 回更新する。

```
def mmd2_loss(x, y, bandwidths=(0.5, 1.0, 2.0)):
    """Unbiased MMD^2 under the sum of RBF kernels of the given bandwidths."""
    def kernel(a, b):
        sq_dist = torch.cdist(a, b) ** 2
        return sum(torch.exp(-sq_dist / (2 * s ** 2)) for s in bandwidths)
    return mmd2_unbiased(kernel(x, x), kernel(y, y), kernel(x, y))

def mmd2_median(x, y):
    """MMD^2 with one RBF kernel, its bandwidth the median pairwise distance
    of the pooled batch, held constant in the gradient."""
    xy = torch.cat([x, y])
    return mmd2_loss(x, y, (torch.cdist(xy, xy).median().detach(),))


def gradient_penalty(D, X, fake, u, one_sided=False):
    """Mean (|D'(x)| - 1)^2 at the points u X + (1 - u) fake (inputs 1-D);
    one-sided, only slopes steeper than one pay: max(0, |D'(x)| - 1)^2."""
    x = (u * X + (1 - u) * fake).requires_grad_()
    grad, = torch.autograd.grad(D(x).sum(), x, create_graph=True)
    excess = grad.abs() - 1
    return ((excess.clamp(min=0) if one_sided else excess) ** 2).mean()
```

`train_shift` は生成器の 1 ステップごとにクリティックを Adam で 5 ステップ更新し、クリティックを最適応答の近くに保つ。生成器には学習率 0.01 の素の勾配ステップをとらせる。もし Adam を使うと、各ステップがほぼ学習率の大きさにリスケールされてしまい、ここで観察したい勾配の大きさが隠れてしまうからである。この関数は各ステップにおける θ と生成器の勾配を記録する。正の `gp` を指定するとクリティック損失に勾配ペナルティが加わり、`one_sided=True` にすると傾きが 1 より急な部分のみにペナルティ $\max(0, |T'| - 1)^2$ を課す。また、`critic_loss` を `None` にすると、生成器は 2 つのバッチに対する `gen_loss` を直接最小化する。

```
class Shift(nn.Module):
    """The generator x' = theta + z."""
    def __init__(self, theta0):
        super().__init__()
        self.theta = nn.Parameter(torch.tensor(theta0))

    def forward(self, z):
        return self.theta + z


def train_shift(critic_loss, gen_loss, seed, gp=0.0, one_sided=False,
                theta0=6.0, steps=1500, n_critic=5, batch=256, lr_D=1e-2,
                lr_G=1e-2):
    """Fit x' = theta + z to N(0, 1) by plain gradient steps on theta; return
    theta0 and theta after every step, the gradient of every step, and the
    critic."""
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    G, D = Shift(theta0), d2l.make_mlp([1, 32, 32, 1], nn.ReLU())
    opt_G = torch.optim.SGD(G.parameters(), lr_G)
    opt_D = torch.optim.Adam(D.parameters(), lr_D, betas=(0.5, 0.9))
    n = n_critic if critic_loss else 0
```

```
    thetas, grads = [theta0], []
    for _ in range(steps):
        X, Z = (to_t(rng.standard_normal((n + 1, batch, 1))) for _ in range(2))
        U = to_t(rng.uniform(size=(n, batch, 1)))   # interpolation weights
        for i in range(n):
            fake = G(Z[i]).detach()
            loss = critic_loss(D(X[i]), D(fake))
            if gp:
                loss = loss + gp * gradient_penalty(D, X[i], fake, U[i],
                                                    one_sided)
            opt_D.zero_grad()
            loss.backward()
            opt_D.step()
        fake = G(Z[-1])
        loss = (gen_loss(X[-1], fake) if critic_loss is None
                else gen_loss(D(X[-1]), D(fake)))
        opt_G.zero_grad()
        loss.backward()
        grads.append(G.theta.grad.item())
        opt_G.step()
        thetas.append(G.theta.item())
    return np.array(thetas), np.array(grads), D
```

6 つの設定をそれぞれ 3 つのシードで比較する。飽和型ロジスティック、非飽和型ロジスティック、ヒンジ、WGAN-GP ($\gamma = 10$、両側ペナルティ $(|T'|-1)^2$)、片側ペナルティ $\max(0, |T'|-1)^2$ 付き WGAN-GP、および中央値バンド幅 MMD である。各実行について、最初の 100 ステップにおける生成器の勾配の大きさの中央値、θ の軌跡、**到達**ステップ（$|\theta| < 0.5$ に初めて達するステップ）、および到達後の**最遠**逸脱量 $\max|\theta|$ を報告する。

```
w1 = (lambda r, f: f.mean() - r.mean(), lambda r, f: -f.mean())
arms = {'logistic, sat.': (losses['logistic'][0],
                           lambda r, f: -softplus(f).mean(), {}),
        'logistic, NS': (*losses['logistic'], {}),
        'hinge': (*losses['hinge'], {}),
        'W1 + GP': (*w1, dict(gp=10.0)),
        'W1 + 1-sided GP': (*w1, dict(gp=10.0, one_sided=True)),
        'MMD, median': (None, mmd2_median, {})}
steps, w1_runs = (0, 100, 300, 1000, 1500), []  # (arm, seed, theta, critic)
print(f'{"":17s}|grad|   theta at steps {", ".join(map(str, steps))}'
      '  arrival  farthest')
for name, (critic_loss, gen_loss, kwargs) in arms.items():
    for seed in range(3):
        thetas, grads, D = train_shift(critic_loss, gen_loss, seed, **kwargs)
        if name.startswith('W1'):
            w1_runs.append((name, seed, thetas[-1], D))
        hit = np.flatnonzero(np.abs(thetas) < 0.5)  # arrival: hit[0]
        print(f'{name:16s}{seed}{np.median(np.abs(grads[:100])):7.3f} '
              + ''.join(f'{thetas[t]:7.2f}' for t in steps)
              + (f'{hit[0]:9d}{np.abs(thetas[hit[0]:]).max():10.2f}'
                 if len(hit) else '  not reached'))
```

```
                |grad|   theta at steps 0, 100, 300, 1000, 1500  arrival  farthest
```

```
logistic, sat.  0  0.007    6.00   5.99   5.96   5.85   5.77  not reached
logistic, sat.  1  0.009    6.00   5.98   5.95   5.85   5.76  not reached
logistic, sat.  2  0.015    6.00   5.98   5.96   5.85   5.77  not reached
logistic, NS    0  2.812    6.00   3.30   0.79   0.03   0.00      383      0.50
logistic, NS    1  2.876    6.00   3.26   0.78   0.02   0.00      380      0.50
logistic, NS    2  2.730    6.00   3.35   0.80   0.02   0.00      388      0.50
hinge           0  1.451    6.00   4.64   3.31   0.02   0.04      812      0.49
hinge           1  1.558    6.00   4.50   3.50   0.00  -0.04      874      0.50
hinge           2  1.278    6.00   4.81   3.51  -0.06  -0.00      918      0.50
W1 + GP         0  1.262    6.00   4.74   2.39   0.30   0.30      477      1.48
W1 + GP         1  1.263    6.00   4.89   2.52  -4.13  -7.18      487      7.49
W1 + GP         2  1.252    6.00   4.97   2.60  -4.08  -7.61      495      7.61
W1 + 1-sided GP 0  1.265    6.00   4.75   2.40   0.01   0.02      478      0.49
W1 + 1-sided GP 1  1.262    6.00   4.74   2.39  -0.01   0.03      477      0.49
W1 + 1-sided GP 2  1.265    6.00   4.74   2.40   0.02  -0.05      477      0.50
MMD, median     0  0.210    6.00   5.79   5.34   3.34   1.32  not reached
MMD, median     1  0.210    6.00   5.79   5.34   3.34   1.32  not reached
MMD, median     2  0.210    6.00   5.79   5.34   3.35   1.32  not reached
```

この結果から、実際のニューラルクリティックが制約なしの理論的最適解とどう異なるかが明確になる。

- **飽和型と非飽和型のロジスティック損失：**16.1 節と同様、飽和型の生成器勾配はごく小さく（~ 0.01）、θ は 6.0 から 5.76 までしかほとんど動かない。一方、非飽和型の生成器は強い勾配（~ 3）を受け取り、ステップ 400 より前に目標へ到達する。
- **ヒンジクリティック：**理論上の最適ヒンジクリティック $\operatorname{sign}\lambda$ は両方の分布の塊の内部で傾きが 0 になるが、有限の ReLU MLP はギャップ間を傾き 1.1–1.7 で滑らかに補間し、ステップ 700–950 までに生成器を目標へ到達させる。
- **中央値バンド幅 MMD：**飽和型アームを除けば、ステップ 1,500 までに到達しなかった唯一の設定であり、その時点で $\theta \approx 1.3$ である。ここでの生成器は遅いが、停滞しているわけではない。中央値ヒューリスティックによってバンド幅が分離距離と同程度に保たれるため、勾配が消失することはない。ただし勾配は ~ 0.2 と、次項の W_1 クリティックの傾きより約 6 倍小さく、ステップサイズ 0.01 の素の勾配ステップではその分だけ多くのステップを要する。
- **両側と片側の WGAN-GP：**両方の W_1 アームとも、傾き ≈ 1.26（$-ad + 10(a-1)^2$ の最小化により $a = 1 + d/20$）で生成器を内側へ引き寄せ、ステップ 500 付近で到達する。ところが到達後、両側の `W1 + GP` はすべての実行で大きくオーバーシュートする（`farthest` が > 1、最大で > 7）のに対し、片側では到達点より外側へ離れる実行が 1 つもなく（`farthest` が ≤ 0.5）、ステップ 1,000 までにすべて原点から 0.06 以内に収まる。なぜか。θ が 0 を横切ると、クリティックは傾きを負から正へ反転させなければならない。しかし両側ペナルティ $(|T'|-1)^2$ は平坦なクリティック（$T' = 0$）に γ のペナルティを課すため、傾き 0 にポテンシャル障壁が生じ、クリティックが元の傾きの符号（$1 - d/20$）に閉じ込められるか、すべての ReLU ユニットが死んでしまうからである。これは以下でステップ 1,500 におけるクリティックの傾きと活性 ReLU 数を調べることで確認できる。

```
def slope_and_units(D, x):
    """D'(x) and, per hidden layer, the number of ReLU units active at x."""
    x = h = torch.tensor([[x]], dtype=torch.float32, requires_grad=True)
    active = []
    for layer in D:
        h = layer(h)
        if isinstance(layer, nn.ReLU):
```

```
            active.append(int((h > 0).sum()))
        slope, = torch.autograd.grad(h.sum(), x)
        return slope.item(), active

    print(f"{'':19s}theta D'(theta)  active    D'(0)  active"
          '  (of 32 units per layer)')
    for name, seed, theta, D in w1_runs:
        (s_t, a_t), (s_0, a_0) = (slope_and_units(D, x) for x in (theta, 0.0))
        print(f'{name:16s}{seed}{theta:7.2f}{s_t:10.2f}{a_t[0]:5d}{a_t[1]:3d}'
              f'{s_0:9.2f}{a_0[0]:5d}{a_0[1]:3d}')
```

```
                    theta D'(theta)  active    D'(0)  active  (of 32 units per layer)
W1 + GP          0   0.30      0.00   14  0     0.00   14  0
W1 + GP          1  -7.18      1.34   11  8     1.35   12  9
W1 + GP          2  -7.61     -0.65   12  7    -0.65   12  7
W1 + 1-sided GP 0   0.02      0.41   30  3     0.41   30  3
W1 + 1-sided GP 1   0.03      0.65   25  4     0.65   25  4
W1 + 1-sided GP 2  -0.05     -0.16   30  7    -0.16   30  7
```

片側ペナルティ $\max(0, |T'| - 1)^2$ は $|T'| \le 1$ のときにペナルティを課さないため、θ がデータに到達する際にクリティックが傾き 0 を自由に通過できる。16.4 節では、平衡点においてクリティックの傾きを積極的に 0 へ引き寄せるゼロ中心ペナルティ（R_1 と R_2）を解析する。

16.2.4.3 2 次元混合ガウス分布における 5 つの目的関数の比較

次に、生成器とデータのサポートが重なっている状況で 5 つの目的関数を比較する。非飽和型ロジスティック、最小二乗法、ヒンジ、二乗ヒンジ、およびマルチバンド幅 MMD（$s \in \{0.5, 1.0, 2.0\}$）である。生成器 `MixtureGenerator` は 3 成分の混合ガウス分布をパラメータ化し、ずらした平均 `mu_0` とデータより狭いスケール `A_0 = 0.4 * I` で初期化する（$\mathrm{JS}(p, q_0) \approx 0.53$ nat）。

```
mu_0 = np.array([[1.0, -1.0], [-4.3, -2.0], [-1.0, 3.8]])  # off the modes
A_0 = np.stack([0.4 * np.eye(2)] * 3)      # below the data's scale of 0.7

print(f'JS(p, q) at the start: {js_mixtures(mu_0, A_0):.3f} nats')
```

```
JS(p, q) at the start: 0.529 nats
```

初期状態はデータから約 0.5 nat 離れている。MMD アームは、不偏ミニバッチ推定量 MMD^2 をバンド幅 0.5、1、2 の 3 つの RBF カーネルの和のもとで最小化する。これは複数のスケールに感度をもつカーネルである。

```
class MixtureGenerator(nn.Module):
    """x = mu_k + z A_k for a uniformly drawn component k."""
    def __init__(self, mu, A):
        super().__init__()
        self.mu = nn.Parameter(to_t(mu))
        self.A = nn.Parameter(to_t(A))

    def forward(self, z, k):
        return torch.einsum('ni,nij->nj', z, self.A[k]) + self.mu[k]
```

```
    def params_np(self):
        return (self.mu.detach().numpy().copy(),
                self.A.detach().numpy().copy())

losses['MMD'] = (None, mmd2_loss)   # no critic: MMD^2 to the real batch
```

`train_toy` は 2 つの損失を引数にとり、クリティックの 1 ステップと生成器の 1 ステップを交互に繰り返す。クリティック損失を `None` として格納した MMD アームでは生成器のステップのみを実行し、クリティックのスコアではなく `mmd2_loss(X, fake)` に対して更新する。200 ステップごとに厳密な $\mathrm{JS}(p, q_t)$ を記録する。

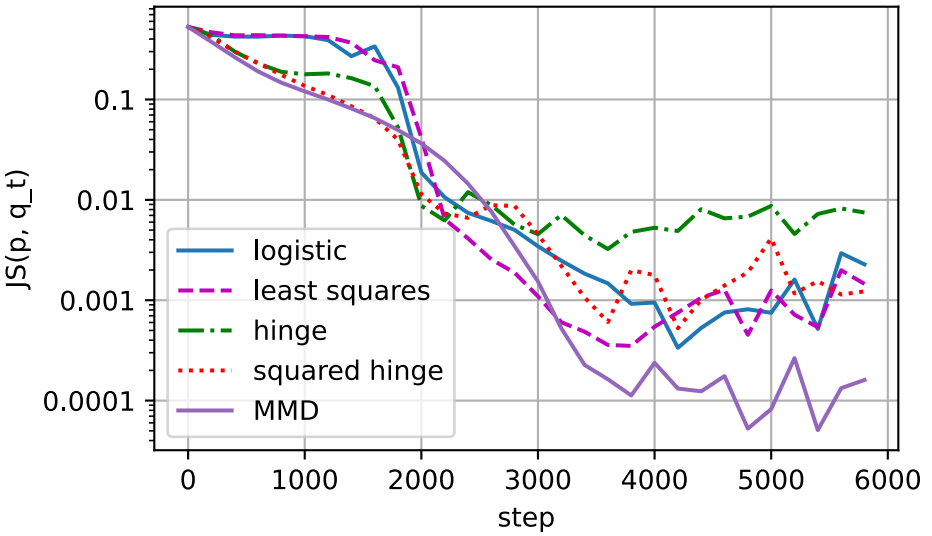
```
def train_toy(critic_loss, gen_loss, seed=0, steps=6000, batch=256, lr=1e-3):
    """Fit a MixtureGenerator from (mu_0, A_0) to the data mixture, one
    critic step per generator step. The losses act on the critic's scores of
    the real and generated batch; with critic_loss None, gen_loss acts on the
    two batches. Returns the exact JS(p, q_t) every 200 steps."""
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    G, D = MixtureGenerator(mu_0, A_0), make_critic()
    opt_G = torch.optim.Adam(G.parameters(), lr)
    opt_D = torch.optim.Adam(D.parameters(), 2 * lr)
    trace = []
    for t in range(steps):
        X = to_t(sample_mixture(batch, mu_p, A_p, rng))
        z = to_t(rng.standard_normal((batch, 2)))
        k = torch.tensor(rng.integers(0, 3, batch))
        if critic_loss is not None:
            opt_D.zero_grad()
            critic_loss(D(X), D(G(z, k).detach())).backward()
            opt_D.step()
        opt_G.zero_grad()
        fake = G(z, k)
        loss = (gen_loss(X, fake) if critic_loss is None
                else gen_loss(D(X), D(fake)))
        loss.backward()
        opt_G.step()
        if t % 200 == 0:
            trace.append(js_mixtures(*G.params_np()))
    return np.array(trace)
```

各目的関数につき 3 つのシードそれぞれで、最後の 2,000 ステップにわたる $\mathrm{JS}(p, q_t)$ の中央値、ならびに $\mathrm{JS}(p, q_t) < 0.1$ nat および < 0.01 nat に初めて到達するステップを報告する。

```
def first_below(trace, level):
    """First recorded step (one record per 200 steps) with JS below level."""
    hits = np.flatnonzero(trace < level)
    return f'{200 * hits[0]:6d}' if len(hits) else '   n/a'

runs = {name: [train_toy(*pair, seed=seed) for seed in range(3)]
        for name, pair in losses.items()}
print(f'{"three runs:":14s}{"median JS, last 2,000 steps":>27s}'
```

```
        f'{"first JS < 0.1":>21s}{"first JS < 0.01":>21s}')
for name, traces in runs.items():
    medians = [np.median(trace[-2000 // 200:]) for trace in traces]
    print(f'{name:14s}   ' + ''.join(f'{m:8.4f}' for m in medians)
          + '   ' + ''.join(first_below(trace, 0.1) for trace in traces)
          + '   ' + ''.join(first_below(trace, 0.01) for trace in traces))
d2l.plot(np.arange(len(runs['logistic'][0])) * 200,
         [traces[0] for traces in runs.values()], xlabel='step',
         ylabel='JS(p, q_t)', yscale='log', legend=list(runs), figsize=(5, 3))
```

```
three runs:   median JS, last 2,000 steps       first JS < 0.1      first JS < 0.01
logistic           0.0008  0.0008  0.0008     2000  2000  2200     2400  2400  2400
least squares      0.0009  0.0009  0.0008     2000  2200  1800     2200  2400  2200
hinge              0.0070  0.0071  0.0033     1800  1800  1400     2000  2000  2400
squared hinge      0.0013  0.0008  0.0024     1400  1400  1400     2200  2200  2200
MMD                0.0001  0.0001  0.0001     1200  1200  1200     2600  2600  2600
```



5 つの目的関数すべてが $\mathrm{JS}(p, q_t)$ を 0.53 nat から 1,000 分の数 nat 以下まで減少させる。変化し続けるクリティックを追跡しない直接的な MMD 最小化が最も精密な最終当てはまり（0.0001 nat）を達成する一方、ヒンジ損失はやや高めの値（0.003–0.007 nat）に落ち着き、滑らかなリンクをもつ目的関数（ロジスティック、最小二乗法、二乗ヒンジはいずれも ≤ 0.0024 nat。演習問題 3）を上回る。続く 16.3 節では、各サンプルを個別に評価するのではなく、実サンプルと生成サンプルをペアで比較するクリティックを検討する。

16.2.5 まとめ

単一サンプルを扱うすべての敵対的目的関数は、クリティックのクラス $\mathcal{T}$ 上での期待利得の上限をとる形をしており（Equation 16.9）、分布 q に関しては凸になる（ただし生成器パラメータ θ に関しては非凸のままである）。クリティックに制約がない場合、対称なマージン分類損失は f ダイバージェンス（それぞれのベイズリスクギャップ $\Delta_\ell(p, q)$ によって定まるもの、Table 16.3）を評価し、最適クリティックは $\lambda = \log(p/q)$ の単調リンクを出力する。また、フェンシェル・ヤング変分下界（Equation 16.13）はこの双対性を非対称な f ダイバージェンスへと拡張する。こ

れに対し、非線形な利得を制約付き関数クラス $\mathcal{F}$ 上の線形な期待値に置き換えると積分確率指標が定義され、その代表例が RKHS の単位球上の最大平均値差異と 1 リプシッツ関数上の 1 次ワッサースタイン距離である。サポートが離れると、f ダイバージェンスは飽和する一方、W_1 は傾き 1 を維持し、MMD^2 はカーネルバンド幅までの距離で有益な勾配を与える。実用上は、有限のニューラルクリティックを非飽和型の生成器損失と組み合わせれば、ギャップ間を滑らかに補間できる。

16.2.6 演習問題

1. [code] **分布空間における凸性とパラメータ空間の対比**。@eq-gan-template の形をしたすべての目的関数は q に関して凸であるが、パラメトリックな生成器は θ を変化させるのであり、q を直接動かすわけではない。`normal_pdf`、`js_1d`、`xs` を用いよ。
   1. $p = \mathcal{N}(0,1)$、$q_0 = \mathcal{N}(8,1)$ とおく。二階差分を用いて、$\mathrm{JS}(p, q_t)$ が $t \in [0,1]$ に関して線形混合経路 $q_t = (1-t)\,q_0 + t\,p$ に沿って凸であることを確認せよ。
   2. 次に $p = 0.7\,\mathcal{N}(-4,1) + 0.3\,\mathcal{N}(4,1)$、$q_\theta = \mathcal{N}(\theta, 1)$ とおく。$\mathrm{JS}(p, q_\theta)$ を $\theta \in [-6, 6]$ の範囲でプロットし、その極小点と極大点を特定せよ。
   3. θ に関するこれらの非凸な性質が、q に関する凸性と矛盾しないのはなぜか。

2. **イェンセン・シャノン・ダイバージェンスの凸共役**。イェンセン・シャノン・ダイバージェンスの 2 倍は、ダイバージェンス関数 $f(u) = u \log u - (u+1) \log \frac{u+1}{2}$ に対応する。
   1. その凸共役 $f^*(t) = -\log(2 - e^t)$ と定義域 $t < \log 2$ を導出せよ。
   2. 再パラメータ化 $T(x) = \log(2\sigma(D(x)))$ のもとで、最適クリティック $T^\star = f'(\rho)$ が $D^\star = \lambda$ に対応し、変分下界 Equation 16.13 が $2\log 2 + V(D)$（ただし $V(D)$ は 16.1 節の古典的な対数損失 GAN の目的関数）になることを示せ。

3. [code] **混合ガウス分布における生成器の重みとクリティックの遅れ**。●`train_toy` と 16.2.4.3 節の 5 つの目的関数を用いよ。
   1. ロジスティックのクリティック損失を線形の生成器損失 $-D(x')$（重み $w = 1$）と組み合わせ、非飽和型損失 $-\log \sigma(D(x'))$ と 3 つのシードにわたり、$\mathrm{JS}(p, q_t) < 0.1$ nat に初めて到達するステップ数で比較せよ。サンプルごとの重みを用いてその違いを説明せよ。
   2. ヒンジ損失の最終的な JS がやや高めになる原因がクリティックの追従遅れにあるのかミニバッチノイズにあるのかを検証するため、3 つのシードで 2 つの対照実験を行え。(i) 生成器の 1 回の更新につきクリティックを 5 回更新する設定、(ii) クリティックのバッチサイズを 4× 大きくする設定である。

4. [code] **サンプル複雑度と次元**。●$p = \mathcal{N}(\mathbf{0}, \mathbf{I}_{d_x})$、$q = \mathcal{N}(\mu, \mathbf{I}_{d_x})$（ただし $\|\mu\| = 1$）とおくと、$W_1(p, q) = 1$ が任意の次元 d_x で成り立つ。
   1. W_1 をサイズ n の 2 つの標本から最適線形割当（`scipy.optimize.linear_sum_assignment`）によって $d_x \in \{1, 2, 10, 50\}$ および $n \in \{100, 400, 1600\}$ について推定し、相対誤差を n に対して両対数軸でプロットせよ。また、MMD^2 の不偏推定量（バンド幅 $s = \sqrt{d_x}$ の RBF カーネルを使用）と比較せよ。
   2. 経験 W_1 距離を、サイズ $n = 256$ の 2 つの独立標本（**同**一の分布 p から次元 $d_x = 50$ で抽出したもの）の間で測定せよ。
   3. 順方向 KL 変分下界 Equation 16.13 を `critics['forward KL']` に対して `grid_2d` 上の数値積分で評価し、真の値 $D_{\mathrm{KL}}(p \,\|\, q) \approx 0.16$ nat およびサイズ 512 のミニバッチ推定値と比較せよ。なぜ q からの有限ミニバッチは、$T(x')$ が $1 + \lambda(x')$ を上回る（$q(x')$ が小さい領域において）ときにこの下界を過大評価しうるのか。

5. **分類損失から得られる対称な f ダイバージェンス。●●**すべてのマージン分類損失 Equation 16.10 は対称なベイズリスク $L(\eta) = L(1-\eta)$ をもち、したがって対称な f ダイバージェンスを評価する。逆に（Nguyen et al. (2009)、系 3）、f を $(0,\infty)$ 上の凸関数で $f(1)=0$、$f(u) = uf(1/u)$、および $f(0) = \lim_{u\to 0} f(u) < \infty$ を満たすものとし、次のように定義する。

$$L(\eta) = 2f(0) - 2(1-\eta)\, f\!\left(\frac{\eta}{1-\eta}\right), \qquad 0 < \eta < 1.$$

   1. $L(\eta)$ が凹であり、$\frac{1}{2}$ に関して対称で、$\eta \to 0$ または 1 のとき 0 に収束し、ベイズリスクギャップ $\Delta_\ell(p,q) = D_f(p \,\|\, q)$ をもつことを示せ。
   2. 微分可能な L と $\hat{\eta} = \sigma(t)$ に対し、実サンプル損失 $-\ell(t) = L(\hat{\eta}) + (1-\hat{\eta})L'(\hat{\eta})$ を定義する。このとき $-\ell(-t) = L(\hat{\eta}) - \hat{\eta}L'(\hat{\eta})$ が成り立つこと、条件付きリスクが $\hat{\eta} = \eta$ で最小化されてその値が $L(\eta)$ になること、および $-\ell(t) \geq 0$ を示せ。
   3. この構成をイェンセン・シャノン・ダイバージェンスに適用するとロジスティック損失が復元されることを確認せよ。

16.3 相対論的目的関数

標準的な分類器型のクリティックは、固定されたゼロのしきい値に対して各サンプルにスコアを付ける（$D(x') > 0$）。しかし多くの場面では、2 つの候補のスコア差を勝率へと変換するブラッドリー・テリー選好モデル（15.3 節）と同様に、絶対的なスコアよりもペアワイズ比較の方が自然である。Jolicoeur-Martineau (2019) は、実サンプル $x \sim p$ と生成サンプル $x' \sim q$ を受け取り、それらのスコア差 $D(x) - D(x')$ からどちらが実データであるかを予測する**相対論的**（relativistic）GAN クリティックを導入した。

16.4 節で構築し、本章の残りの部分でも使用する R3GAN のレシピはこのペアワイズ目的関数に依拠しているため、本節では次の 3 つの問いを解析する。

1. 最適なペアワイズランキングのクリティックは、依然として対数密度比（log density ratio）$\lambda(x) = \log(p(x)/q(x))$ を復元するか。
2. ペアワイズ目的関数はその最適点においてどのダイバージェンスを測り、それは $\mathrm{JS}(p,q)$ と比べてどのような関係にあるか。
3. ペアワイズランキングは、どのような条件のもとでモード欠落（mode dropping）の極小値を解消するか。

```
import itertools
```

16.3.1 ペアのスコア付け

16.3.1.1 ブラッドリー・テリーのペアリングクリティック

独立なペア (x, x')（ただし $x \sim p$、$x' \sim q$）が与えられたとき、**相対論的ペアリング**（relativistic pairing）目的関数（RpGAN、Jolicoeur-Martineau (2019); Jolicoeur-Martineau (2020)）は、x が x' よりも本物らしい確率をブラッドリー・テリー尤度 $\sigma(D(x) - D(x'))$ によってモデル化する (Bradley and Terry 1952)。

$$\Phi(D) \;=\; \mathbb{E}_{x\sim p,\; x'\sim q}\big[\log \sigma(D(x) - D(x'))\big]. \tag{16.17}$$

クリティックは $\Phi(D)$ を最大化する（$-\Phi(D)$ を最小化する）。ミニバッチ上では、`rpgan_loss_D` は $-\log\sigma(D(x_i) - D(x_i')) = \text{softplus}(D(x_i') - D(x_i))$ を対応するペア (x_i, x_i') にわたって平均し、一方の `rpgan_loss_G` は引数を入れ替えることで非飽和生成器損失（non-saturating generator loss）を実装する。

```
def rpgan_loss_D(critic, real, fake):
    """Relativistic pairing critic loss -E[log sigma(D(x) - D(x'))]."""
    return F.softplus(critic(fake) - critic(real)).mean()


def rpgan_loss_G(critic, real, fake):
    """Non-saturating pairing loss for the generator."""
    return F.softplus(critic(real) - critic(fake)).mean()
```

16.3.1.2 シフト不変性と最適クリティック

対数損失によってゼロのしきい値が固定される単一サンプル分類器とは異なり、ペアワイズランキング目的関数は差分 $D(x) - D(x')$ のみに依存する（Figure 16.5）。任意の定数 b を D に加えても、$\Phi(D + b) = \Phi(D)$ は変わらない。したがって、個々のスコア $D(x')$ の符号は任意であり、自然な正解率の評価指標は代わりにペアワイズランキング正解率 $P(D(x) > D(x'))$ となる。

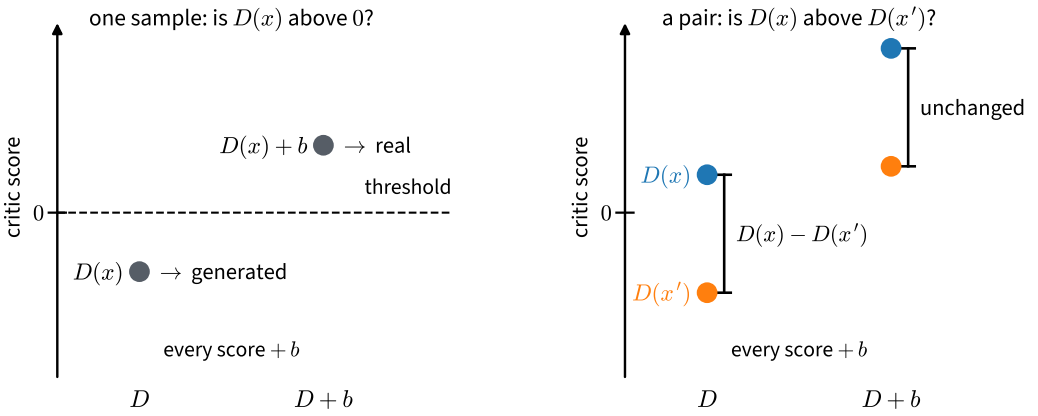


Figure 16.5: 単一サンプルのしきい値判定（左）とペアワイズ差分のスコア付け（右）。すべてのクリティックのスコアを定数 b だけシフトするとゼロに対する単一サンプル分類は変化するが、ペアワイズ差分 $D(x) - D(x')$ は不変に保たれる。

単一サンプルの対数損失目的関数では、$V(D)$ の最大化は各点 x ごとに独立に分解できた。Equation 16.17 ではすべての点が差分 $D(x) - D(x')$ を介して結合しているが、凹性（concavity）により依然として閉形式の最適解が得られる。

命題。 **$\Phi(D)$ は D に関して凹関数であり、p と q が同じ台（support）を共有するとき、加法定数の違いを除いて一意に $D^\star(x) = \lambda(x) = \log(p(x)/q(x))$ で最大化される。**

証明。 $D \mapsto D(x) - D(x')$ は線形であり、$\log\sigma$ は狭義凹関数であるため、$\Phi(D)$ は D に関して凹、スコア差に関しては狭義凹となる。$\Phi(D)$ の $D(t)$ に関する汎関数微分（ここで t は実サンプル x と生成サンプル x' の両方として現れる）をとり、$\frac{d}{du}\log\sigma(u) = \sigma(-u)$ を用いると次式を得る。

$$\frac{\delta \Phi}{\delta D(t)} = p(t)\,\mathbb{E}_{x' \sim q}\big[\sigma(D(x') - D(t))\big] - q(t)\,\mathbb{E}_{x \sim p}\big[\sigma(D(t) - D(x))\big]. \tag{16.18}$$

$D = \lambda = \log \rho$（ただし $\rho = p/q$）で評価すると、$\sigma(\lambda(x') - \lambda(t)) = \rho(x')/(\rho(x') + \rho(t))$ となる。$q\rho = p$ を用いると、Equation 16.18 の両項はいずれも $p(t) \int \frac{p(x)}{\rho(x)+\rho(t)}\,dx$ に等しくなり、至るところで $\frac{\delta \Phi}{\delta D(t)} = 0$ が成り立つ。スコア差に関する $\log \sigma$ の狭義凹性により、$D^\star = \lambda + c$ が定数シフト c の違いを除いて唯一の最大点となる。■

16.3.2 ペアリングダイバージェンス

16.3.2.1 2 状態の例

$\Phi(D)$ の最適値がイェンセン・シャノン・ダイバージェンス（Jensen–Shannon divergence）とどのように関係するかを確かめるため、2 状態空間 $\{0, 1\}$ 上で $p = (0.8, 0.2)$ および $q = (0.2, 0.8)$ を考える。実サンプルを 1 番目に置いた順序対 (a_1, a_2) は積分布（product distribution）$P(a_1, a_2) = p(a_1)q(a_2)$ に従い、実サンプルを 2 番目に置いたペアは $Q(a_1, a_2) = q(a_1)p(a_2)$ に従う。

ペア (a_1, a_2)	$(0,0)$	$(1,1)$	$(0,1)$	$(1,0)$
$P = p \otimes q$（実サンプルが先）	0.16	0.16	0.64	0.04
$Q = q \otimes p$（実サンプルが後）	0.16	0.16	0.04	0.64

同点ペア $(0,0)$ と $(1,1)$ はどちらの要素が実データであるかについて何の情報も持たないが、混合ペア $(0,1)$ と $(1,0)$ は事後確率（posterior probability）$0.64/(0.64+0.04) = 16/17 \approx 0.94$ で実データの要素を特定する（単一サンプルを観測した場合の 0.80 との比較）。最適なスコア差 $d = D^\star(0) - D^\star(1) = \lambda(0) - \lambda(1) = \log 16$ において、ランダムな当て推量（$\Phi(0) = -\log 2$）に対する利得は $\sup_D \Phi(D) + \log 2 = 0.68(\log 2 - H_b(\frac{1}{17})) \approx 0.319$ nat となる。これは $1.66\times$ の倍率で $\mathrm{JS}(p, q) = \log 2 - H_b(0.8) \approx 0.193$ nat を上回り、さらに $\mathrm{JS}(P, Q)$（2 つのペア分布 P と Q の間のダイバージェンス）に厳密に一致する。

```
def entropy(v):
    v = v[v > 0]                      # the convention 0 log 0 = 0
    return -(v * np.log(v)).sum()

def js(a, b):
    return entropy((a + b) / 2) - (entropy(a) + entropy(b)) / 2

p, q = np.array([0.8, 0.2]), np.array([0.2, 0.8])
P, Q = np.outer(p, q), np.outer(q, p)  # P[i, j]: real sample at i, generated at j
lam = np.log(p / q)                    # the optimal critic: lam[0] - lam[1] = log 16
U = lam[:, None] - lam[None, :]        # U[i, j] = D(i) - D(j)
phi = (P * -np.logaddexp(0, -U)).sum() # Phi = E_P[log sigma(D(a_1) - D(a_2))]
print(f'JS(p, q) = {js(p, q):.3f}   JS(P, Q) = {js(P.ravel(), Q.ravel()):.3f}   '
      f'sup Phi + log 2 = {phi + np.log(2):.3f}')
```

```
JS(p, q) = 0.193   JS(P, Q) = 0.319   sup Phi + log 2 = 0.319
```

16.3.2.2 積測度への持ち上げ

この等価性は一般に証明できる。そのために、順序対 $(a_1, a_2) \in \mathcal{X} \times \mathcal{X}$ を $P = p \otimes q$（実サンプルが先）または $Q = q \otimes p$（実サンプルが後）のいずれかから抽出された単一の観測とみなし、16.1 節の標準的な対数損失目的関数を $\mathcal{X} \times \mathcal{X}$ 上でペアクリティック $\mathcal{D}(a_1, a_2)$ に対して考える。

$$V_{P,Q}(\mathcal{D}) = \mathbb{E}_{(a_1,a_2)\sim P}\big[\log \sigma(\mathcal{D}(a_1, a_2))\big] + \mathbb{E}_{(a_1,a_2)\sim Q}\big[\log \sigma(-\mathcal{D}(a_1, a_2))\big].$$

補題（積空間への持ち上げ）。 *$P = p \otimes q$ および $Q = q \otimes p$ とする。このとき、次が成り立つ。*

1. **積空間（product space）上の対数密度比は加法的に分離する。すなわち、** $\log \frac{dP}{dQ}(a_1, a_2) = \lambda(a_1) - \lambda(a_2)$ **である。**
2. **任意の反対称な差分クリティック** $\mathcal{D}(a_1, a_2) = D(a_1) - D(a_2)$ **は** $V_{P,Q}(\mathcal{D}) = 2\,\Phi(D)$ **を満たす。**
3. $\mathcal{D}$ **を差分クリティック** $D(a_1) - D(a_2)$ **に制限しても上限（supremum）は減少しない。すなわち、** $\sup_{\mathcal{D}} V_{P,Q}(\mathcal{D}) = \sup_{D} V_{P,Q}(D(a_1) - D(a_2))$ **が成り立つ。**

証明。（1）は $\frac{dP}{dQ}(a_1, a_2) = \frac{p(a_1)q(a_2)}{q(a_1)p(a_2)} = \frac{\rho(a_1)}{\rho(a_2)}$ から直ちに従う。（2）については、Q のもとで $(a_1, a_2) \sim q \otimes p$ かつ $-\mathcal{D}(a_1, a_2) = D(a_2) - D(a_1)$ となるため、$V_{P,Q}(\mathcal{D})$ に含まれる 2 つの期待値はいずれも $\Phi(D)$ に等しい。（3）については、$V_{P,Q}(\mathcal{D})$ の制約なし最大点は $\mathcal{D}^\star(a_1, a_2) = \log \frac{dP}{dQ}(a_1, a_2)$ であり、これは（1）よりすでに差分の形 $\lambda(a_1) - \lambda(a_2)$ をしている。■

（なお、性質 3 はロジスティック利得が持つ対数リンク関数に固有の性質である。最小二乗法などの他の利得では、最適なペアクリティックは $\lambda(a_1) - \lambda(a_2)$ の非線形関数となり、加法的に分離可能ではなくなる。演習問題 4 を参照。）

定理。 *較正されたペアリングダイバージェンス* $d_{\mathrm{Rp}}(p, q) := \sup_D \Phi(D) + \log 2$ *は次を満たす。*

$$\begin{aligned} d_{\mathrm{Rp}}(p, q) &= \mathrm{JS}(p \otimes q,\ q \otimes p) \\ &= H\Big(\tfrac{1}{2}(p \otimes q + q \otimes p)\Big) - H(p) - H(q). \end{aligned} \tag{16.19}$$

証明。 持ち上げ補題の 2 と 3 を、イェンセン・シャノンのミニマックス恒等式 $\sup_{\mathcal{D}} V_{P,Q}(\mathcal{D}) = 2\,\mathrm{JS}(P, Q) - 2\log 2$（16.1 節）と組み合わせると、$2\sup_D \Phi(D) = \sup_{\mathcal{D}} V_{P,Q}(\mathcal{D}) = 2\,\mathrm{JS}(P, Q) - 2\log 2$ が得られ、第 1 の等号が示される。さらにエントロピーギャップの恒等式 $\mathrm{JS}(P, Q) = H(\frac{1}{2}(P + Q)) - \frac{1}{2}(H(P) + H(Q))$（16.1 節）を適用し、$H(p \otimes q) = H(q \otimes p) = H(p) + H(q)$ を用いれば、第 2 の等号が得られる。■

$D^\star = \lambda$ を Equation 16.17 に直接代入すると、明示的な期待値表示も得られる。

$$d_{\mathrm{Rp}}(p, q) = \log 2 - \mathbb{E}_{x\sim p,\, x'\sim q}\Big[\log\Big(1 + \frac{q(x)\,p(x')}{p(x)\,q(x')}\Big)\Big], \tag{16.20}$$

また、ペア空間上の相互情報量（mutual information）の恒等式 $\mathrm{JS}(P, Q) = I((a_1, a_2); y)$（16.1 節）から、ランダムに順序付けられたペア (a_1, a_2) とその順序ラベル $y \in \{0, 1\}$ に対して次式が成り立つ。

$$d_{\mathrm{Rp}}(p,q) = I\big((a_1,a_2);\, y\big) = \log 2 - H(y \mid a_1, a_2). \tag{16.21}$$

16.3.2.3 情報量境界

Equation 16.21 を用いると、$d_{\mathrm{Rp}}(p,q)$ を $\mathrm{JS}(p,q)$ と直接結びつけることができる。ランダムに順序付けられたペアの各周辺要素 a_k は分布 $m = (p+q)/2$ に従い、$I(a_k; y) = \mathrm{JS}(p,q)$ nat の情報量を順序ラベル y について持つ。一方、a_1 と a_2 は y が与えられたもとで条件付き独立である（$H(a_2 \mid a_1, y) = H(a_2 \mid y)$）。相互情報量の連鎖律（chain rule）より、次式を得る。

$$\begin{aligned} d_{\mathrm{Rp}}(p,q) &= I(a_1; y) + I(a_2; y \mid a_1) \\ &= I(a_1; y) + I(a_2; y) - I(a_1; a_2) \\ &= 2\,\mathrm{JS}(p,q) - I(a_1; a_2). \end{aligned}$$

$I(a_2; y \mid a_1) \ge 0$、$I(a_1; a_2) \ge 0$、および $d_{\mathrm{Rp}}(p,q) \le \log 2$ より、タイトな上下界（挟み撃ちの評価式）が得られる。

$$\mathrm{JS}(p,q) \le d_{\mathrm{Rp}}(p,q) \le \min\big(2\,\mathrm{JS}(p,q),\, \log 2\big).$$

均衡点の近傍（$q \to p$）では、$I(a_1; a_2)$ は $p - q$ に関して 4 次のオーダーであるのに対し、$\mathrm{JS}(p,q)$ は 2 次のオーダーであるため、$d_{\mathrm{Rp}}(p,q) \approx 2\,\mathrm{JS}(p,q)$ となる（演習問題 2）。また、p と q が互いに素な台を持つとき、$\mathrm{JS}(p,q)$ と $d_{\mathrm{Rp}}(p,q)$ はどちらも $\log 2$ で飽和する。

16.3.3 ランキングとモード被覆

16.3.3.1 相対論的ランク重み

$\Phi(D)$ を生成器パラメータ θ に関して $x' = G(z)$ を介して微分すると、次式が得られる。

$$\begin{aligned} \nabla_\theta \Phi &= -\,\mathbb{E}_z\big[w_{\mathrm{Rp}}(G(z))\, \nabla_\theta D(G(z))\big], \\ w_{\mathrm{Rp}}(x') &= \mathbb{E}_{x \sim p}\big[\sigma(D(x') - D(x))\big]. \end{aligned} \tag{16.22}$$

飽和型のゼロサム定式化（Φ の最小化）では、各生成サンプルは $\nabla_\theta D(G(z))$ に沿って重み $w_{\mathrm{Rp}}(x')$ で押し出される。この重みは、x' がランダムに抽出された実サンプル $x \sim p$ を上回るソフト化された確率である。非飽和型の定式化（`rpgan_loss_G`、すなわち $-\mathbb{E}[\log \sigma(D(x') - D(x))]$ の最小化）では、重みは $1 - w_{\mathrm{Rp}}(x') = \mathbb{E}_{x \sim p}[\sigma(D(x) - D(x'))]$ となる。単一サンプル重み $\sigma(\pm D(x'))$（$D(x')$ をゼロとのみ比較する）とは異なり、相対論的重みは $D(x')$ を実スコア $D(x)$ の分布と比較するため、D の定数シフトに対して不変である。

16.3.3.2 固定ペアリングと全ペアのランドスケープ

ペアワイズランキングはモード欠落の極小値を解消するだろうか。Sun et al. (2020) は、n 個の相異なる実データ点 $x_1, \dots, x_n$ と n 個の自由に移動できる生成点 $x'_1, \dots, x'_n$ を、厳密な最適応答（best-response）クリティック D のもとで評価する離散設定を解析した。

$$\begin{aligned}\mathcal{E}_{\text{GAN}} &= \sup_D \tfrac{1}{2n}\textstyle\sum_i \left[\log\sigma(D(x_i)) + \log\sigma(-D(x_i'))\right],\\ \mathcal{E}_{\text{diag}} &= \sup_D \tfrac{1}{n}\textstyle\sum_i \log\sigma(D(x_i) - D(x_i')),\\ \mathcal{E}_{\text{all}} &= \sup_D \tfrac{1}{n^2}\textstyle\sum_{i,j} \log\sigma(D(x_i) - D(x_j')).\end{aligned} \tag{16.23}$$

ここで、$\mathcal{E}_{\text{GAN}}$ は古典的な単一サンプル目的関数、$\mathcal{E}_{\text{diag}}$ は**固定ペアリング**（fixed-pairing）目的関数（実データ点 x_i が生成点 x_i' と恒久的にペアを組む設定）であり、$\mathcal{E}_{\text{all}}$ は実データと生成点の全 n^2 ペアにわたって平均をとる。すべての x_i' が実データ点上に位置するものの、少なくとも 2 点が一致する（結果として被覆されない実データ点が残る）配置は**衝突**（collision）と呼ばれる。Sun et al. (2020) は、すべての衝突（全部で $n^n - n!$ 個）が $\mathcal{E}_{\text{GAN}}$ の狭義の極小値となる一方、$\mathcal{E}_{\text{diag}}$ にはあらゆる衝突から大域的最小値 $-\log 2$ へと至る非増加な脱出経路が存在することを証明した。

この事実は $n = 2$ と $n = 3$ について直接検証できる。経験分布（empirical distribution）を $\hat{p}$ と $\hat{q}$ とすると、$\mathcal{E}_{\text{GAN}} = \text{JS}(\hat{p}, \hat{q}) - \log 2$ および $\mathcal{E}_{\text{all}} = \text{JS}(\hat{p}\otimes\hat{q}, \hat{q}\otimes\hat{p}) - \log 2$ となる一方、$\mathcal{E}_{\text{diag}}$ は $-\log 2$ に閉路（cycle）上の有向辺 $i \to \text{gen}[i]$ の割合を掛けた値に等しい（演習問題 5）。

```
def on_cycle(i, gen):
    """Does the arrow from real point i to its partner's position close a cycle?"""
    j = gen[i]
    for _ in range(len(gen)):
        if j == i:
            return True
        if j not in range(len(gen)):  # a position that holds no data point
            return False
        j = gen[j]                    # follow the arrow of the real point at j
    return False

def losses(gen):
    """Best-response classical, all-pairs and paired losses of a configuration."""
    n = len(gen)
    support = sorted(set(range(n)) | set(gen))
    p_hat = np.isin(support, range(n)) / n
    q_hat = np.array([gen.count(s) for s in support]) / n
    P, Q = np.outer(p_hat, q_hat).ravel(), np.outer(q_hat, p_hat).ravel()
    paired = -np.log(2) * np.mean([on_cycle(i, gen) for i in range(n)])
    return np.array([js(p_hat, q_hat) - np.log(2), js(P, Q) - np.log(2), paired])

print('gen         classical, all pairs, paired: nats, multiples of log 2')
for gen in ([0, 0], [0, 0.1]):   # both on x_1; then x'_2 steps off it
    value = losses(gen)
    print(f'{str(gen):12s}{np.round(value, 3)}  {np.round(value / np.log(2), 2)}')
```

```
gen         classical, all pairs, paired: nats, multiples of log 2
[0, 0]      [-0.477 -0.347 -0.347]  [-0.69 -0.5  -0.5 ]
[0, 0.1]    [-0.347 -0.173 -0.347]  [-0.5  -0.25 -0.5 ]
```

x_2' を 0 から 0.1 へ動かすと、古典的損失と全ペア損失は増加する（衝突 `[0, 0]` が両者にとって狭義の極小値であることが確認できる）一方、固定ペアリング損失は $-\frac{1}{2}\log 2$ のまま変化

しない。全 n^n 通りの配置を体系的に調べるため、count_bad_minima(n) はすべての割り当てを small_moves(gen) による局所摂動に対してテストする。

```
def small_moves(gen):
    """Each generated point stays or steps to one of n fresh positions next to it."""
    n = len(gen)
    for steps in itertools.product(range(n + 1), repeat=n):
        if any(steps):
            yield [g + k / 10 if k else g for g, k in zip(gen, steps)]

def count_bad_minima(n):
    """Suboptimal strict local minima among the assignments, per loss."""
    best, count = losses(list(range(n))), np.zeros(3, dtype=int)
    for gen in itertools.product(range(n), repeat=n):
        value = losses(list(gen))
        rises = np.all([losses(m) > value + 1e-9 for m in small_moves(list(gen))],
                       axis=0)
        count += rises & (value > best + 1e-9)
    return count
```

count_bad_minima を $n \in \{2, 3\}$ に対して実行し、各目的関数における準最適な狭義の極小値の個数を理論上の衝突数 $n^n - n!$ と比較する。

```
for n in (2, 3):
    c = count_bad_minima(n)
    print(f'n = {n}: classical {c[0]}, all pairs {c[1]}, paired {c[2]};  '
          f'n^n - n! = {n**n - math.factorial(n)}')
```

```
n = 2: classical 2, all pairs 2, paired 0;  n^n - n! = 2
n = 3: classical 21, all pairs 21, paired 0;  n^n - n! = 21
```

重要な点として、この全列挙から、固定結合損失 $\mathcal{E}_{\text{diag}}$ では衝突による極小値が 0 個であるのに対し、全ペア損失 $\mathcal{E}_{\text{all}}$ は $n^n - n!$ 個すべての衝突を極小値として持つ（$\mathcal{E}_{\text{GAN}}$ とまったく同様である）ことがわかる。標準的なミニバッチ訓練ではステップごとに実データと生成データのミニバッチを独立に再サンプリングするため、ランダムなペアリングにわたる目的関数の期待値は $\mathcal{E}_{\text{all}}$ となり、$\mathcal{E}_{\text{diag}}$ にはならない（演習問題 5）。したがって、Sun et al. (2020) のランドスケープ定理は固定された 1 対 1 結合にのみ適用され、ミニバッチ RpGAN によるモード被覆（mode coverage）の改善効果は実験的に検証する必要がある（16.4 節）。

16.3.3.3 訓練における飽和型と非飽和型のペアリング

飽和型の生成器損失（Φ）と非飽和型の生成器損失（rpgan_loss_G）は、どちらも大域的最小値 $q = p$ を共有する。p と q がすでに重なり合っている場合（たとえば 16.2.4.3 節の 3 モード混合ガウス分布ベンチマーク）、両方の変種とも数千分の 1 nat 以内に収束する。

```
scores = lambda s: s   # train_toy passes scores, so the critic is the identity
arms = {'non-saturating': (lambda r, f: d2l.rpgan_loss_D(scores, r, f),
                           lambda r, f: d2l.rpgan_loss_G(scores, r, f)),
        'saturating': (lambda r, f: d2l.rpgan_loss_D(scores, r, f),
                       lambda r, f: -d2l.rpgan_loss_D(scores, r, f))}
```

```
    def first_step_below(trace, level=0.01, every=200):
        below = np.flatnonzero(trace < level)
        return f'{every * below[0]:5d}' if len(below) else ' none'

    print('pairing critic; generator loss, runs 0, 1, 2')
    for name, (critic_loss, gen_loss) in arms.items():
        traces = [d2l.train_toy(critic_loss, gen_loss, seed) for seed in range(3)]
        medians = ' '.join(f'{np.median(t[-2000 // 200:]):.4f}' for t in traces)
        firsts = ' '.join(first_step_below(t) for t in traces)
        print(f'{name:15s} median JS {medians}   first JS < 0.01 {firsts}')
```

```
pairing critic; generator loss, runs 0, 1, 2
non-saturating  median JS 0.0006 0.0005 0.0007   first JS < 0.01  2400  2600  2400
saturating      median JS 0.0020 0.0017 0.0014   first JS < 0.01  1800  1800  1800
```

どちらのアームも数千分の 1 nat 以下の精度で適合する。非飽和型の実行はステップ 2,200〜2,600 の間で 0.01 nat を下回る。これは 16.2 節のロジスティック損失（logistic loss）と同様であり、ペアリングによって適合精度が損なわれることはない。さらに飽和型は、データの近傍で重みが小さくならないため、すべての実行でステップ 1,800 において先にこの水準を下回る。

しかし、生成器がデータから遠く離れた位置から出発する場合（$\theta_0 = 6$ とした `d2l.train_shift`、16.2.4.2 節）、すべての生成サンプルがすべての実サンプルよりはるかに低くランク付けされ（$D(x') \ll D(x)$）、飽和型のランク重み $w_{\mathrm{Rp}}(x')$ はゼロへと押し下げられる。

```
    steps = (0, 100, 300, 1000, 1500)
    print(f'{"":20s}|gradient|   theta at steps {steps}   first |theta| < 0.5')
    for name, (critic_loss, gen_loss) in arms.items():
        for seed in range(3):
            thetas, grads, _ = d2l.train_shift(critic_loss, gen_loss, seed)
            hit = np.flatnonzero(np.abs(thetas) < 0.5)
            print(f'{name:15s} {seed}   {np.median(np.abs(grads[:100])):8.3f}   '
                  + ''.join(f'{thetas[t]:7.2f}' for t in steps) + '   '
                  + (f'{hit[0]:5d}' if len(hit) else 'not reached'))
```

```
                    |gradient|   theta at steps (0, 100, 300, 1000, 1500)   first |theta| < 0.5
non-saturating  0      3.244      6.00   2.41   0.57   0.02   0.00     323
non-saturating  1      3.021      6.00   2.26   0.55   0.02   0.00     317
non-saturating  2      3.057      6.00   2.65   0.60   0.02   0.00     335
saturating      0      0.000      6.00   6.00   6.00   6.00   6.00   not reached
saturating      1      0.000      6.00   6.00   6.00   6.00   5.99   not reached
saturating      2      0.000      6.00   6.00   6.00   6.00   5.99   not reached
```

$\theta_0 = 6$ から開始すると、飽和型ペアリング生成器の勾配の中央値は 5×10^{-4} 未満となり、1,500 ステップ後も $\theta \approx 6.00$ に停滞したままである。これに対し、非飽和型ペアリング生成器（`rpgan_loss_G`）は ~ 3 の勾配を受け取り、すべてのシードでステップ 330 前後には $|\theta| < 0.5$ に到達する。したがって、本章の残りの部分では一貫して非飽和型ペアリング損失を使用する。

16.3.4 まとめ

相対論的ペアリング目的関数（RpGAN）は、実サンプルを生成サンプルより上位にランク付けするブラッドリー・テリー対数尤度 $\Phi(D) = \mathbb{E}_{x\sim p, x'\sim q}[\log \sigma(D(x) - D(x'))]$ を最大化し、加法定数の違いを除いて対数密度比 $D^\star(x) = \lambda(x) = \log(p(x)/q(x))$ を復元する。(a_1, a_2) を積空

間 $\mathcal{X}\times\mathcal{X}$ へ持ち上げることで、較正された最適値 $d_{\mathrm{Rp}}(p,q)=\sup_D \Phi(D)+\log 2$ が $\mathrm{JS}(p\otimes q, q\otimes p)=2\,\mathrm{JS}(p,q)-I(a_1;a_2)$ に等しくなり、$\mathrm{JS}(p,q)\le d_{\mathrm{Rp}}(p,q)\le \min(2\,\mathrm{JS}(p,q),\log 2)$ を満たすことが示される。恒久的な 1 対 1 結合 $\mathcal{E}_{\mathrm{diag}}$ は最適応答クリティックのもとで離散的なモード欠落の衝突極小値を解消するものの、独立なミニバッチの再サンプリングは平均化によって全ペア目的関数 $\mathcal{E}_{\mathrm{all}}$ へと戻り、それらの極小値を残してしまう。最後に、生成サンプルがデータから遠く離れているとき、飽和型のランク重み $w_{\mathrm{Rp}}(x')=\mathbb{E}_{x\sim p}[\sigma(D(x')-D(x))]$ は消失するが、非飽和型の重み $1-w_{\mathrm{Rp}}(x')$ は 1 付近にとどまり、生成器の勾配を維持する。

16.3.5 演習問題

1. [code] **飽和型と非飽和型のランク重み**。データ分布 $p=(0.35,0.25,0.20,0.12,0.08)$ と生成器分布 $q=(0.02,0.05,0.13,0.30,0.50)$ を持つ 5 状態の離散空間を考える。
   1. 2 状態のセルで定義した `js` を用いて $\mathrm{JS}(p,q)$ と $d_{\mathrm{Rp}}(p,q)$ を計算し、25 個のペア状態上の $\mathrm{JS}(P,Q)$ が $\sup_D \Phi(D)+\log 2$（$D=\lambda$ における値）と一致することを検証せよ（それぞれ ≈ 0.234 および 0.377）。
   2. 飽和型の重み w_{Rp}（Equation 16.22）と非飽和型の重み $1-w_{\mathrm{Rp}}$ を $D=\lambda$ のもとで各状態について評価し、それらの q に関する期待値を比較せよ。
   3. $D=\lambda$ において、$\mathbb{E}_{x'\sim q}[w_{\mathrm{Rp}}(x')]=\iint \frac{PQ}{P+Q}\,da_1\,da_2\le\frac{1}{2}$ が成り立ち、等号成立は $q=p$ のとき、かつそのときに限ること、さらに p と q の台が離れるにつれてこの値が消失することを示せ。
   4. `d2l.train_shift` をペアリングクリティックと両方の生成器損失に対して初期オフセット `theta0` $\in\{1,2,3,4\}$（各 3 シード）から実行し、飽和型の生成器が停滞し始める距離を求めよ。
2. **d_{Rp} と $2\,\mathrm{JS}(p,q)$ の差**。●ランダムに順序付けられたペア (a_1,a_2) とその順序ラベル y（Equation 16.21）を考える。
   1. a_1 と a_2 が y のもとで条件付き独立であること、および $I(a_k;y)=\mathrm{JS}(p,q)$ が $k\in\{1,2\}$ に対して成り立つことを示せ。
   2. p と q が互いに素な台を持つとき、$I(a_1;a_2)$ をペアの混合分布 $\frac{1}{2}(P+Q)$ から直接計算し、$I(a_1;a_2)=\mathrm{JS}(p,q)=\log 2$ を検証せよ。
   3. 局所摂動 $q=p\,(1+\epsilon v)$（ただし $\mathbb{E}_p[v]=0$）に対して、$\mathrm{JS}(p,q)=\Theta(\epsilon^2)$ である一方 $I(a_1;a_2)=\Theta(\epsilon^4)$ となり、したがって $d_{\mathrm{Rp}}(p,q)/\mathrm{JS}(p,q)\to 2$（$\epsilon\to 0$ のとき）が成り立つことを示せ。
   4. $d_{\mathrm{Rp}}(p,q)=2\,\mathrm{JS}(p,q)-I(a_1;a_2)$ を、5 状態の分布（演習問題 1）上で数値的に検証せよ。
3. **2 つのガウス分布**。$p=\mathcal{N}(0,I_d)$ および $q=\mathcal{N}(\mu,I_d)$ を $\mathbb{R}^d$ 上の分布とする。
   1. $\lambda(x)$ が x に関してアフィンであり、$u=\lambda(x')-\lambda(x)$（Equation 16.20）がガウス分布に従うことを示し、その平均と分散を求めよ。
   2. $\mathrm{JS}(p,q)$ を同じ形で表し、スケーリング定数 c として $\mathrm{JS}(p,\mathcal{N}(c\mu,I_d))=d_{\mathrm{Rp}}(p,q)$ がすべての μ に対して成り立つものを求めよ。
   3. $d_{\mathrm{Rp}}(p,q)$ と $\mathrm{JS}(p,q)$ を 1D 数値積分によって $\|\mu\|\in\{0.1,1,2,4\}$ に対して評価し、その比を局所極限および大分離極限と比較せよ。
4. **ペアクリティックの分離可能性**。●$\log\sigma$（Equation 16.17）を一般の凹利得関数 ℓ に置き換え、$\Phi_\ell(D)=\mathbb{E}_{x\sim p,x'\sim q}[\ell(D(x)-D(x'))]$ とする。
   1. 最小二乗法の利得 $\ell(t)=-(1-t)^2$ に対し、$\mathcal{X}\times\mathcal{X}$ 上の制約なし最適ペアクリティックが $2\eta-1=\tanh\left(\frac{\lambda(a_1)-\lambda(a_2)}{2}\right)$（ただし $\eta=P/(P+Q)$）となり、$D(a_1)-D(a_2)$ のように加法的に分離可能ではないことを示せ。
   2. 5 状態の例（演習問題 1）において、制約なしのペア最適値と $\sup_{D\in\mathbb{R}^5} 2\,\Phi_\ell(D)$ とのギャップを計算せよ。

3. 不均衡な順序事前確率 $\alpha \neq \frac{1}{2}$ を持つロジスティック利得のもとで、ベイズ最適なペアクリティックを導出し、それを達成するには差分クラス $D(a_1) - D(a_2)$ をどのように拡張すべきかを示せ。

5. [code] **固定結合とランダムな再ペアリング。●●**
   1. `losses` で $\mathcal{E}_{\text{diag}}$ に対して用いた閉路則を証明せよ。すなわち、閉路に含まれない任意の有向辺上では、クリティックは $D(x_i) - D(x_i') \to +\infty$ とすることでその損失をゼロにできる一方、長さ k の閉路に沿って差分の和をとるとゼロになり、$\log\sigma$ の凹性により等スコアのときに最大値 $-k\log 2$ をとることを示せ。
   2. `count_bad_minima` を $n = 4$（256 通りの配置）に拡張し、$\mathcal{E}_{\text{GAN}}$、$\mathcal{E}_{\text{all}}$、および $\mathcal{E}_{\text{diag}}$ における個数を検証せよ。
   3. 置換平均した最適応答目的関数 $\frac{1}{n!}\sum_{\pi}\sup_{D}\frac{1}{n}\sum_{i}\log\sigma(D(x_i) - D(x'_{\pi(i)}))$ を定義し、$n = 3$ に対する準最適な狭義の極小値の個数を数えよ。このレジームを、ランダムにペアリングされた多数のミニバッチにわたって緩やかに適応するクリティック（$\mathcal{E}_{\text{all}}$ に直面する）と比較せよ。

16.4 勾配ペナルティと収束

16.1 節の Fashion-MNIST 実験では、識別器損失も生成器損失も理論上の平衡値には落ち着かなかった。交互勾配降下法は収束が遅いだけなのか、それとも平衡点に到達することなくその周囲を際限なく周回しうるのだろうか。16.1 節から 16.3 節までの母集団ダイバージェンスの解析では、クリティックが厳密な最適解 $D^\star$ にあるときのミニマックス目的関数を特徴づけた。しかし実際には、両方のネットワークが同時または交互に小さな勾配ステップを踏み、一方の更新がもう一方の目的関数を絶えず変化させる。

勾配訓練のダイナミクスを直接解析するため、Mescheder et al. (2018) は点質量のデータ分布、点質量の生成器、および線形クリティックからなる最小限の 2 パラメータ問題である **Dirac-GAN** を導入した。このような 2 パラメータ系においてさえ、正則化なしの勾配降下法は収束しない。一方で、クリティックの入力勾配の二乗ノルムにペナルティ（R_1 と R_2）を課すと、平衡点は局所漸近安定（locally asymptotically stable）になる。これらのゼロ中心勾配ペナルティ（zero-centered gradient penalty）を 16.3 節の相対論的ペアリング目的関数と組み合わせることで、本章の残りの節で用いる R3GAN レシピが得られる (Huang et al. 2024)。

16.4.1 Dirac-GAN

16.4.1.1 2 パラメータのミニマックス系

データ分布を原点における点質量 $p = \delta_0$ とし、生成器は $\theta \in \mathbb{R}$ に点質量を置く分布 $q_\theta = \delta_\theta$ とする（したがって $q_\theta = p$ となるのは $\theta = 0$ のとき、かつそのときに限る）。また、クリティックをバイアスなしの線形関数 $D_\psi(x) = \psi x$ とする。対称マージン目的関数 $V_\ell(D) = \mathbb{E}_{x\sim p}[\ell(D(x))] + \mathbb{E}_{x'\sim q}[\ell(-D(x'))]$（16.2 節）のもとでは、両方の期待値は単一の評価に帰着する。

$$\begin{aligned} V(\theta,\psi) &= \ell(D_\psi(0)) + \ell(-D_\psi(\theta)) \\ &= \ell(0) + \ell(-\psi\theta). \end{aligned} \tag{16.24}$$

ここではロジスティック利得 $\ell(t) = \log\sigma(t)$ を用いる。この関数は $\ell'(t) = \sigma(-t) > 0$ および $\ell''(t) = -\sigma(t)\sigma(-t) < 0$ を満たし、$\ell'(0) = \frac{1}{2}$、$\ell''(0) = -\frac{1}{4}$ である。

任意の $\theta \neq 0$ に対して、p と q_θ の台（support）は互いに素である。そのため、$V(\theta,\psi)$ を ψ について最大化すると $\psi\theta \to -\infty$ となり、定数 $\sup_\psi V(\theta,\psi) = \ell(0)$ がすべての $\theta \neq 0$ に対して得られる結果、θ に関する勾配はゼロになる。

16.4.1.2 連続時間における非減衰振動

勾配訓練では、$\sup_\psi V(\theta,\psi)$ を直接解く代わりに、現在の値から θ と ψ を少しずつ更新する。Equation 16.24 の唯一の停留点（stationary point）は平衡点 $(\theta,\psi) = (0,0)$ である。連続時間の同時勾配フロー（$\dot\theta = -\partial_\theta V$、$\dot\psi = +\partial_\psi V$）のもとでは、次式が成り立つ。

$$\begin{aligned} \dot\theta &= -\partial_\theta V = \psi\,\ell'(-\psi\theta), \\ \dot\psi &= +\partial_\psi V = -\theta\,\ell'(-\psi\theta). \end{aligned} \tag{16.25}$$

速度ベクトル $\mathbf{u}(\theta,\psi) = \ell'(-\psi\theta)\,(\psi,-\theta)$ は至るところで状態ベクトル (θ,ψ) に直交するため、原点までの二乗距離は保存される。

$$\frac{d}{dt}(\theta^2+\psi^2) = 2\theta\psi\,\ell'(-\psi\theta) - 2\psi\theta\,\ell'(-\psi\theta) = 0.$$

あらゆる軌跡は、$(0,0)$ を中心とする一定半径の閉じた円を描く（これは任意の狭義単調増加な利得 ℓ に対して成り立つ）。すなわち、生成器のパラメータ θ は決して収束することなく、0 をまたいで前後に振動し続ける。

Equation 16.25 を $(\theta,\psi) = (0,0)$ の周りで線形化すると（38.1.2.5 節）、ヤコビ行列と固有値は次のようになる。

$$J = \begin{pmatrix} 0 & \ell'(0) \\ -\ell'(0) & 0 \end{pmatrix}, \qquad \nu_{1,2} = \pm\, i\,\ell'(0). \tag{16.26}$$

純虚数の固有値は線形の**中心点**（center）に対応する（双線形鞍点問題 $\min_\theta \max_\psi \theta\psi$、Daskalakis et al. (2018)）。なお、スカラー因子 $\ell'(-\psi\theta) = \sigma(D_\psi(\theta)) = w_{\text{sat}}$ は両成分に同時に掛かるため、円周上の角速度をリスケールするにすぎない。

16.4.1.3 離散時間における外向きスパイラルとリミットサイクル

離散時間では、ステップサイズ $h > 0$ の同時勾配降下法は $(\theta,\psi) \mapsto (\theta,\psi) + h\,\mathbf{u}(\theta,\psi)$ と更新する。$\mathbf{u}(\theta,\psi) \perp (\theta,\psi)$ であるため、ピタゴラスの定理より次式が得られる。

$$\big\|(\theta,\psi) + h\,\mathbf{u}\big\|^2 = \big\|(\theta,\psi)\big\|^2 + h^2\big\|\mathbf{u}\big\|^2.$$

したがって、任意の $h > 0$ に対して、離散時間の同時ステップは毎回、平衡点からの距離を厳密に**増加させ**、反復点は外向きに渦を巻いて発散する（演習問題 1）。

交互勾配更新（まず ψ を更新し、続いて θ を更新後の ψ を用いて更新する手法）は、これよりわずかに良い挙動を示す。$(0,0)$ の近傍では、各半ステップが面積保存の線形せん断となるため、合成されたヤコビ行列の行列式は 1 である。$h\,\ell'(0) \le 2$ のとき、両方の固有値は単位円上に乗るため (Mescheder et al. 2018)、交互更新の反復点は収束することなく原点の周りの有界な帯状領域内を周回する（演習問題 4）。非飽和生成器損失（原点で同じ線形化 J を共有する）も、相対論

的ペアリング損失（$\ell(D_\psi(0) - D_\psi(\theta)) = \ell(-\psi\theta)$ となり同一の勾配場を持つ）も、これらの局所固有値を変えない。

16.4.2 ゼロ中心勾配ペナルティ

平衡点 $(\theta, \psi) = (0, 0)$ を局所漸近安定にするには、ヤコビ行列の固有値を虚軸上から開左半平面 $\mathrm{Re}(\nu) < 0$ へと移動させなければならない。平衡時のクリティックは平坦でなければならないため（$\nabla_x D^\star(x) = 0$ が $q = p$ のときに成り立つ）、クリティックの入力勾配の二乗ノルムにペナルティを課せば、平衡 $q = p$ をずらすことなくクリティックに減衰を加えられる。

Mescheder et al. (2018) は、2 つの**ゼロ中心勾配ペナルティ**として R_1（実データ $x \sim p$ 上で評価）と R_2（生成サンプル $x' \sim q$ 上で評価）を次のように定義している。

$$
\begin{aligned}
R_1 &= \frac{\gamma}{2}\, \mathbb{E}_{x \sim p}\Big[\big\|\nabla_x D(x)\big\|^2\Big], \\
R_2 &= \frac{\gamma}{2}\, \mathbb{E}_{x' \sim q}\Big[\big\|\nabla_{x'} D(x')\big\|^2\Big],
\end{aligned}
\tag{16.27}
$$

ここで $\gamma > 0$ はペナルティの重みである。クリティックは $V_\ell(D) - R_1 - R_2$ を最大化し、生成器の目的関数は変わらない。$m = (p + q)/2$ とおくと、両者の和は混合分布のもとでの二乗勾配ノルムの期待値となる。

$$
R_1 + R_2 = \gamma\, \mathbb{E}_{x \sim m}\Big[\big\|\nabla_x D(x)\big\|^2\Big], \qquad m = \frac{p + q}{2}.
\tag{16.28}
$$

16.4.2.1 Dirac-GAN の減衰

線形クリティック $D_\psi(x) = \psi x$ に対しては、至るところで $\nabla_x D_\psi(x) = \psi$ であるため、$R_1 = R_2 = \tfrac{\gamma}{2}\psi^2$ となる。クリティックの目的関数から $\tfrac{\gamma}{2}\psi^2$ を引くと、線形減衰項 $-\gamma\psi$ が $\dot\psi$ に加わる（$-2\gamma\psi$ となるのは R_1 と R_2 の両方を含めた場合である）。

$$
\dot\theta = \psi\, \ell'(-\psi\theta), \qquad \dot\psi = -\theta\, \ell'(-\psi\theta) - \gamma\psi.
$$

$(0, 0)$ におけるヤコビ行列とその固有値は次のようになる。

$$
J_\gamma = \begin{pmatrix} 0 & \ell'(0) \\ -\ell'(0) & -\gamma \end{pmatrix}, \qquad \nu_{1,2} \;=\; -\frac{\gamma}{2} \pm \sqrt{\frac{\gamma^2}{4} - \ell'(0)^2}.
\tag{16.29}
$$

すべての $\gamma > 0$ に対して、両方の固有値は厳密に負の実部（$\mathrm{Re}(\nu_{1,2}) < 0$）を持つため、平衡点 $(0, 0)$ は局所指数安定（locally exponentially stable）である。特性多項式 $\nu^2 + \gamma\nu + \ell'(0)^2 = 0$ は減衰調和振動子（38.1.2.5 節）のものと同一であり、次の 3 つのレジームを持つ。

1. **不足減衰**（underdamped、$\gamma < 2\ell'(0)$）：実部 $-\gamma/2$ を持つ複素共役固有値となり、軌跡は内向きに渦を巻く（同時オイラーステップのもとでは $h < \gamma/\ell'(0)^2$ のときに収束する。演習問題 1）。
2. **臨界減衰**（critically damped、$\gamma = 2\ell'(0)$、すなわちロジスティック損失では $\gamma = 1$）：重複実固有値 $\nu = -\ell'(0)$ を持ち、最も速い非振動的減衰をもたらす。
3. **過減衰**（overdamped、$\gamma > 2\ell'(0)$）：2 つの負の実固有値を持ち、遅いモード $\nu \approx -\ell'(0)^2/\gamma$ は γ が大きくなるほどゆっくりと減衰する。

16.4.2.2 相図

Equation 16.25 を $(\theta_0, \psi_0) = (1, 1)$（初期半径 $\sqrt{2} \approx 1.41$）から、4 次のルンゲ・クッタ法（`rk4_step`）、同時前進オイラー法（`euler_step`）、および交互勾配更新（`alt_step`）を用いて数値積分し、これら 4 つのレジームを検証する。

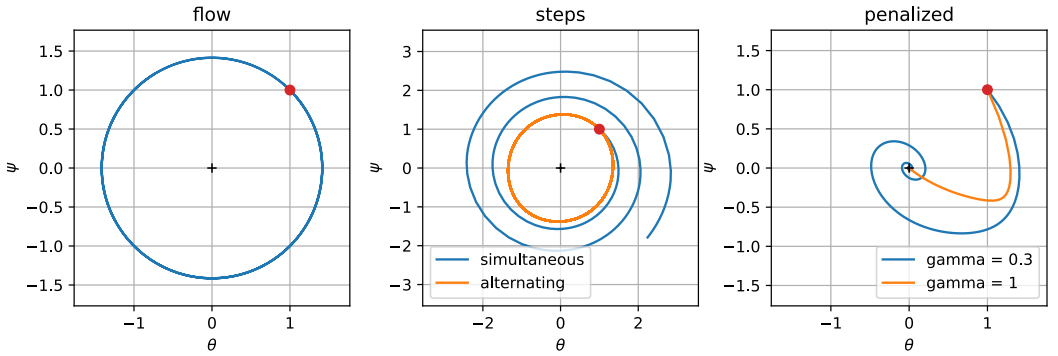
```
def dirac_field(theta, psi, gamma=0.0):
    """Simultaneous gradient field of the Dirac-GAN with the logistic loss."""
    w = 1.0 / (1.0 + np.exp(-psi * theta))      # l'(-psi theta) = w_sat
    return np.array([psi * w, -theta * w - gamma * psi])

def euler_step(x, h, gamma):                    # one simultaneous step
    return x + h * dirac_field(*x, gamma)

def alt_step(x, h, gamma):                      # critic first, then generator
    theta, psi = x
    psi = psi + h * dirac_field(theta, psi, gamma)[1]
    return np.array([theta + h * dirac_field(theta, psi, gamma)[0], psi])

def rk4_step(x, h, gamma):                      # one RK4 step of the ODE
    k1 = dirac_field(*x, gamma)
    k2 = dirac_field(*(x + h / 2 * k1), gamma)
    k3 = dirac_field(*(x + h / 2 * k2), gamma)
    k4 = dirac_field(*(x + h * k3), gamma)
    return x + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)

def trajectory(step, x0, h, steps, gamma=0.0):
    xs = [np.array(x0)]
    for _ in range(steps):
        xs.append(step(xs[-1], h, gamma))
    return np.stack(xs)
```

すべての軌道は赤い点 $(\theta, \psi) = (1, 1)$（黒い十字で示す平衡点からの距離 $\sqrt{2} \approx 1.41$）から出発する。左のパネルは連続時間フローを示す。中央のパネルは、ステップサイズ $h = 0.2$ による 300 回の同時ステップと 2,000 回の交互ステップを示す。右のパネルは、ペナルティを加えた同じステップサイズによる 300 回の同時ステップを、不足減衰レジームの $\gamma = 0.3$ と臨界減衰の $\gamma = 1$ について示す。

```
x0 = np.array([1.0, 1.0])
panels = {'flow': {'RK4': trajectory(rk4_step, x0, 0.05, 800)},
          'steps': {'simultaneous': trajectory(euler_step, x0, 0.2, 300),
                    'alternating': trajectory(alt_step, x0, 0.2, 2000)},
          'penalized': {
              'gamma = 0.3': trajectory(euler_step, x0, 0.2, 300, gamma=0.3),
              'gamma = 1': trajectory(euler_step, x0, 0.2, 300, gamma=1.0)}}
fig, axes = d2l.plt.subplots(1, 3, figsize=(9.5, 3.4))
for ax, (title, paths) in zip(axes, panels.items()):
    for name, P in paths.items():
        r = np.linalg.norm(P, axis=1)           # distance to the equilibrium
        late = r[len(r) // 2:]
        print(f'{title}, {name}: {r[0]:.3g} -> {r[-1]:.3g} in {len(r) - 1} '
              f'steps; second half in [{late.min():.3g}, {late.max():.3g}]')
```

```
    lim = 1.25 * max(np.abs(P).max() for P in paths.values())
    d2l.plot([P[:, 0] for P in paths.values()] + [x0[:1], [0.0]],
             [P[:, 1] for P in paths.values()] + [x0[1:], [0.0]],
             r'$\theta$', r'$\psi$', list(paths) if len(paths) > 1 else None,
             xlim=(-lim, lim), ylim=(-lim, lim), axes=ax,
             fmts=('C0-', 'C1-')[:len(paths)] + ('C3o', 'k+'),
             legend_loc='lower right' if title == 'penalized' else None)
    ax.set_title(title)
    ax.set_aspect('equal')
fig.tight_layout()
```

```
flow, RK4: 1.41 -> 1.41 in 800 steps; second half in [1.41, 1.41]
steps, simultaneous: 1.41 -> 2.87 in 300 steps; second half in [2.44, 2.87]
steps, alternating: 1.41 -> 1.36 in 2000 steps; second half in [1.34, 1.42]
penalized, gamma = 0.3: 1.41 -> 0.000652 in 300 steps; second half in [0.000652, 0.0396]
penalized, gamma = 1: 1.41 -> 6.97e-12 in 300 steps; second half in [6.97e-12, 2.33e-05]
```



相図から 4 つの予測すべてが確認できる。連続時間フロー（RK4）は半径 1.41 の円上を周回し、同時オイラーステップは 300 ステップ後に半径 2.87 まで外向きに渦を巻いて広がり、交互ステップは $[1.34, 1.42]$ の内部を周回する。そしてゼロ中心ペナルティを加えると、同時オイラーステップは原点へと直接収束する（半径 6.5×10^{-4} に不足減衰 $\gamma = 0.3$ で到達し、7×10^{-12} に臨界減衰 $\gamma = 1$ で到達する）。

16.4.2.3 一般のネットワークに対する局所収束とペナルティの比較

Mescheder et al. (2018) は、この局所安定性解析を 2 パラメータの Dirac-GAN から一般のニューラルネットワークへと拡張した。すなわち、生成器がデータ分布 p を表し、クリティックが $\mathrm{supp}(p)$ 上で平坦であるような平衡の近傍において、クリティックが生成器の一次摂動を検出するのに十分な局所表現能力を備えていれば、R_1 または R_2（および $\ell'(0) \neq 0, \ell''(0) < 0$ を満たす狭義凹損失）を用いた同時および交互勾配更新は、平衡多様体へ指数関数的に収束する（Huang et al. (2024) は相対論的ペアリング目的関数に対しても同様の局所定理を確立している）。

この解析から、実践上の 2 つの設計選択の理由も説明できる。

- **ゼロ中心（R_1, R_2）対 1 中心（WGAN-GP）ペナルティ：** WGAN-GP は $(\|\nabla_x D\| - 1)^2$ にペナルティを課すため、$q = p$ のときでさえクリティックに傾き 1 を維持することを強制する。線形利得 $\ell(t) = t$ を持つ Dirac-GAN では、1 中心ペナルティによって $\theta = 0$ への収束が妨げられる。$\gamma \geq 2$ のときはワッサースタイン勾配とペナルティが釣り合う $|\theta| = \gamma$ に生成器を停留させ、それより小さい γ では生成器がこの点をオーバーシュートして周期軌道を回り続ける（演習問題 6）。

- **R_1 と R_2 の併用対 R_1 単独の使用：**$q = p$ の近傍では、R_1 と R_2 は等価である。しかし平衡から離れた領域では、R_1 は実データ点 $x \sim p$ におけるクリティックの傾きにしかペナルティを課さず、生成点 $x' \sim q$ における傾きを制約しないままである（たとえば $D_c(x) = -cx^2$ は $p = \delta_0$ において傾きがゼロであるため $R_1 = 0$ となり、$c \to \infty$ が許容されて $x' = 1$ における生成器の勾配が爆発する。演習問題 3)。R_2 を加えることで、生成サンプルにおけるクリティックの勾配も有界に抑えられる。

16.4.3 勾配ペナルティが価値関数に及ぼす影響

ゼロ中心ペナルティ $R_1 + R_2 = \gamma\, \mathbb{E}_m[\|\nabla_x D\|^2]$ は、平衡の近傍で生成器が最小化するダイバージェンスをどのように変化させるのだろうか。

16.4.3.1 局所二次近似とガウス分布のシフト

$q = p$ の近傍では、最適なクリティック D は小さい。$\ell(t) \approx \ell(0) + \ell'(0)t - \frac{1}{2}|\ell''(0)|t^2$ により D の 2 次まで $V_\ell(D)$ の内部を展開すると、次式が得られる。

$$V_\ell(D) \approx 2\ell(0) + \ell'(0)\,\langle p - q,\, D\rangle - |\ell''(0)| \int m\, D^2,$$

ここで $\langle p - q, D\rangle = \mathbb{E}_p[D] - \mathbb{E}_q[D]$ である。2 つの単位分散ガウス分布 $p = \mathcal{N}(0,1)$ と $q = \mathcal{N}(\mu,1)$（小さなシフト μ だけ離れている）および線形クリティック $D(x) = cx$ を考える。μ の主要項までとると、$\langle p - q, D\rangle = -c\mu$、$\int m\, D^2 \approx c^2$ であり、ペナルティは $\gamma \int m \|\nabla_x D\|^2 = \gamma c^2$ となる。凹二次関数 $-\ell'(0)\mu c - (|\ell''(0)| + \gamma)c^2$ を c について最大化すると、最適な傾き $c^\star = -\frac{\ell'(0)\mu}{2(|\ell''(0)|+\gamma)}$ と次の値が得られる。

$$\frac{\ell'(0)^2\mu^2}{4\,(|\ell''(0)| + \gamma)} = \frac{\mu^2}{4 + 16\gamma}$$

$$\text{for the logistic loss, at slope } c^\star = -\frac{\mu}{1 + 4\gamma}.$$

このように、ペナルティによってクリティックの傾きと局所的な値はともに $1 + 4\gamma$ 分の 1 に縮小する。$\gamma = 0$ では $\mu^2/4 \approx 2\,\mathrm{JS}(p,q)$ が復元され、大きい γ では値は $\mu^2/(16\gamma) = \frac{\ell'(0)^2}{4\gamma} W_2(p,q)^2$ のようにスケールする。以下では、1 次元グリッド上でニュートン法により計算した厳密なペナルティ付き最適値と比較し、この閉形式近似を検証する。

```
def penalized_value(p, q, gamma, x, newton_steps=30):
    """Value of the logistic game minus gamma E_m[D'^2], on a grid."""
    dx, m = x[1] - x[0], (p + q) / 2
    diff = np.diff(np.eye(len(x)), axis=0) / dx       # D -> D'
    K = 2 * gamma * dx * diff.T @ (((m[1:] + m[:-1]) / 2)[:, None] * diff)
    D = np.zeros_like(x)
    for _ in range(newton_steps):                     # Newton ascent on D
        s = 1 / (1 + np.exp(-D))                      # sigma(D)
        grad = (p * (1 - s) - q * s) * dx - K @ D     # gradient of the value
        neg_hess = np.diag(2 * m * s * (1 - s) * dx) + K  # minus its Hessian
        D = D + np.linalg.solve(neg_hess, grad)
    value = -(p * np.logaddexp(0, -D) + q * np.logaddexp(0, D)).sum() * dx
```

```
        return value - D @ K @ D / 2 + 2 * np.log(2)       # minus 2 l(0)

    x = np.linspace(-8, 8, 401)
    for mu in (0.2, 1, 3):
        p, q = [np.exp(-(x - c) ** 2 / 2) / np.sqrt(2 * np.pi) for c in (0, mu)]
        for gamma in (0, 0.01, 1, 10):
            exact = penalized_value(p, q, gamma, x)
            local = mu ** 2 / (4 + 16 * gamma)
            large = mu ** 2 / (16 * gamma) if gamma else np.inf
            print(f'mu {mu:>3}, gamma {gamma:>4}: exact {exact:.3e}, local '
                  f'{local:.3e} ({local / exact:.2f}x), large-gamma {large:.3e} '
                  f'({large / exact:.2f}x)')
```

```
mu 0.2, gamma    0: exact 9.950e-03, local 1.000e-02 (1.00x), large-gamma inf (infx)
mu 0.2, gamma 0.01: exact 9.566e-03, local 9.615e-03 (1.01x), large-gamma 2.500e-01 (26.13x)
mu 0.2, gamma    1: exact 1.996e-03, local 2.000e-03 (1.00x), large-gamma 2.500e-03 (1.25x)
mu 0.2, gamma   10: exact 2.438e-04, local 2.439e-04 (1.00x), large-gamma 2.500e-04 (1.03x)
mu   1, gamma    0: exact 2.228e-01, local 2.500e-01 (1.12x), large-gamma inf (infx)
mu   1, gamma 0.01: exact 2.135e-01, local 2.404e-01 (1.13x), large-gamma 6.250e+00 (29.27x)
mu   1, gamma    1: exact 4.818e-02, local 5.000e-02 (1.04x), large-gamma 6.250e-02 (1.30x)
mu   1, gamma   10: exact 6.120e-03, local 6.098e-03 (1.00x), large-gamma 6.250e-03 (1.02x)
mu   3, gamma    0: exact 1.054e+00, local 2.250e+00 (2.14x), large-gamma inf (infx)
mu   3, gamma 0.01: exact 1.012e+00, local 2.163e+00 (2.14x), large-gamma 5.625e+01 (55.57x)
mu   3, gamma    1: exact 3.909e-01, local 4.500e-01 (1.15x), large-gamma 5.625e-01 (1.44x)
mu   3, gamma   10: exact 6.862e-02, local 5.488e-02 (0.80x), large-gamma 5.625e-02 (0.82x)
```

シフトが小さい場合（$\mu = 0.2$）、二次近似式 $\mu^2/(4+16\gamma)$ はすべての γ にわたって厳密な数値解と 1% 以内の誤差で一致する。

16.4.3.2 双対ソボレフノルムとローパスフィルタリング

平衡近傍にある一般の分布 p と q に対しては、部分積分を用いることで勾配ペナルティを $\int m\,\|\nabla_x D\|^2 = \langle D, \mathcal{L}_m D\rangle$ と書き換えられる。ここで $\mathcal{L}_m D = -\nabla\cdot(m\nabla D)$ は m 重み付きラプラシアンである。局所二次近似をすべての関数 D にわたって最大化すると、次式が得られる（演習問題 5）。

$$\begin{aligned}\sup_{D}\Big\{\ell'(0)\,\langle p-q,\,D\rangle - |\ell''(0)|\int m\,D^2 - \gamma\int m\,\|\nabla_x D\|^2\Big\}\\ = \frac{\ell'(0)^2}{4}\,\big\langle p-q,\,\big(|\ell''(0)|\,m+\gamma\mathcal{L}_m\big)^{-1}(p-q)\big\rangle.\end{aligned} \tag{16.30}$$

$\gamma\to 0$ のとき、この式は $\frac{\ell'(0)^2}{4|\ell''(0)|}\int (p-q)^2/m \approx 2\,\mathrm{JS}(p,q)$ に帰着する。一方、$\gamma\to\infty$ のときは、**双対ソボレフノルム**（dual Sobolev norm）$\dot H^{-1}(m)$ の二乗の定数倍に漸近する。

$$\begin{aligned}&\frac{\ell'(0)^2}{4\gamma}\,\|p-q\|^2_{\dot H^{-1}(m)},\\ &\|v\|_{\dot H^{-1}(m)} = \sup\Big\{\langle v, D\rangle : \int m\|\nabla_x D\|^2 \le 1\Big\}.\end{aligned} \tag{16.31}$$

m がほぼ一様な領域では、$\mathcal{L}_m$ は波数 ω のフーリエモードに $m\omega^2$ を掛けるため、そのモードの重み付けは $(|\ell''(0)| + \gamma\omega^2)^{-1}$ となる（ペナルティなしの $|\ell''(0)|^{-1}$ に代わる）。したがって、$R_1 + R_2$ は $p - q$ に対する空間的な**ローパスフィルタ**として働き、そのカットオフ長さスケールは $\sqrt{\gamma/|\ell''(0)|}$（ロジスティック損失では $2\sqrt{\gamma}$）となる。$\nabla_x D$ は長さの逆数の単位を持つため、γ は長さの二乗の単位を持ち、データの解像度やノルムに応じてスケールする。なお、$R_1 + R_2$ が $\nabla_x D$ にペナルティを課すのは $m = (p + q)/2$ の台の上のみである。互いに素な台の間に広がる空白領域をまたぐ勾配は、依然としてニューラルネットワークであるクリティックの滑らかさに依存する（演習問題 2）。

16.4.4 R3GAN レシピ

R3GAN (Huang et al. 2024) は、相対論的ペアリング損失（`d2l.rpgan_loss_D` および `d2l.rpgan_loss_G`、16.3 節）と両方のゼロ中心勾配ペナルティ R_1、R_2 を組み合わせた手法である。

16.4.4.1 損失とペナルティの実装

比較のため、`gan_loss_D` と `gan_loss_G`（16.1 節の古典的なロジスティック識別器損失と非飽和生成器損失）も同じ `(critic, real, fake)` シグネチャで実装し、保存しておく。

```
def gan_loss_D(critic, real, fake):
    """Logistic critic loss -E[log sigma(D(x))] - E[log sigma(-D(x'))]: the
    sum of two batch means, 2 log 2 at equilibrium, as update_D."""
    return F.softplus(-critic(real)).mean() + F.softplus(critic(fake)).mean()


def gan_loss_G(critic, real, fake):
    """Non-saturating generator loss -E[log sigma(D(x'))]; real is unused."""
    return F.softplus(-critic(fake)).mean()
```

ペナルティ関数は、係数 $\gamma/2$ を掛ける前の、両バッチにおけるサンプルごとの二乗入力勾配を返す。ここではクリティックのスコアの**和**を入力バッチに関して微分する。各スコアはそれぞれの行のみに依存するため、1 回の逆伝播ですべてのサンプルの入力勾配が求まる（行同士を混合するバッチ正規化を用いると、この性質は成り立たなくなる）。入力勾配は自動微分の計算グラフ内に保持されるため、オプティマイザはクリティックのパラメータに関してペナルティをさらに微分できる。このときサンプル自体は定数として固定する。2 回目の微分を伴うため、ペナルティ付きクリティックステップの計算コストはおよそ 2 回分の逆伝播に相当する。

`create_graph=True` によって 2 回目の微分のために計算グラフ内に入力勾配を保持し、`detach` によってサンプルを定数として扱う。

```
def r1_r2_penalty(critic, real, fake):
    """Per-sample squared critic input gradients on real (R1) and fake (R2),
    before the gamma/2 scale."""
    def sq_grad_norm(x):
        x = x.detach().requires_grad_(True)
        grad, = torch.autograd.grad(critic(x).sum(), x, create_graph=True)
        return grad.reshape(x.shape[0], -1).pow(2).sum(dim=1)
    return sq_grad_norm(real), sq_grad_norm(fake)
```

16.4.4.2 訓練ステップ

訓練ループでは、各プレイヤーのステップを 1 回ずつ交互に実行する。`critic_step` は生成バッチを固定した状態で、クリティック損失に両ペナルティの $\gamma/2$ 倍を加えた目的関数に対してオプティマイザを 1 ステップ進め、損失ならびに実サンプルと生成サンプルにおける二乗入力勾配のバッチ平均（R_1 と R_2 の $\gamma/2$ を掛ける前の値）を返す。これにより、訓練ループ内でクリティックの急峻さを記録できる。続いて `generator_step` が、更新後のクリティックに対する生成器損失について 1 ステップ進める。オプションのラベル `y` は、16.6 節の条件付きクリティックで用いる。

```
def critic_step(net_D, opt_D, loss_D, real, fake, gamma, y=None):
    """One critic step on loss_D + gamma/2 (R1 + R2); return the loss and
    the batch means of R1 and R2."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    fake = fake.detach()  # the generator is held fixed
    r1, r2 = r1_r2_penalty(critic, real, fake)
    loss = loss_D(critic, real, fake)
    opt_D.zero_grad()
    (loss + gamma / 2 * (r1 + r2).mean()).backward()
    opt_D.step()
    return loss.detach(), r1.mean().detach(), r2.mean().detach()


def generator_step(net_G, opt_G, loss_G, net_D, real, z, y=None):
    """One generator step on loss_G against the critic; return the loss."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    loss = loss_G(critic, real, net_G(z) if y is None else net_G(z, y))
    opt_G.zero_grad()
    loss.backward()
    opt_G.step()
    return loss.detach()
```

16.4.4.3 25 ガウスグリッドにおけるモード被覆

ニューラルネットワークにおけるモード被覆（mode coverage）に $R_1 + R_2$ とペアワイズランキングがどう影響するかを検証するため、2 次元ガウス分布の 5×5 グリッド（間隔 2.0、標準偏差 $\sigma = 0.05$）上で、3 層の MLP 生成器（64 次元の潜在変数、256 個の ReLU ユニットからなる 2 つの隠れ層）と 3 層の Leaky ReLU MLP クリティックを訓練する。生成された 10,000 個のサンプルを次の 4 つの評価指標で評価する。

1. **被覆モード数**（covered modes）：モード上サンプルのうち、少なくとも $1/5$ 倍（均等配分 $1/25$ に対する比率）を受け取ったモードの数。
2. **モード上サンプル比率**（on-mode fraction）：最近傍のモード中心から $3\sigma = 0.15$ 以内に入るサンプルの割合（真の分布では 98.9%）。
3. 最近傍の中心までの**平均距離**（真の分布では ≈ 0.06）。
4. 生成サンプルにおける**クリティックの急峻さ**（critic steepness）$\mathbb{E}_q[\|\nabla_x D(x')\|^2]$。

```
def grid_centers(side=5, spacing=2.0):
    coords = (torch.arange(side, dtype=torch.float32) - (side - 1) / 2)
    gx, gy = torch.meshgrid(coords * spacing, coords * spacing, indexing='ij')
    return torch.stack([gx.reshape(-1), gy.reshape(-1)], dim=1)
```

```
def sample_modes(centers, n, width=0.05):
    idx = torch.randint(0, centers.shape[0], (n,), device=centers.device)
    return centers[idx] + width * torch.randn(n, 2, device=centers.device)
```

生成された点群をグリッドの centers に照らして評価するため、mode_stats は各サンプルを最近傍のモード中心に割り当て、被覆モード数、3σ 以内のサンプル比率、および最近傍中心までの平均距離を返す。

```
def mode_stats(x, centers, width=0.05):
    """Modes covered (each holds a fifth of an even share of the on-mode
    samples), fraction within three widths, mean distance."""
    dist = np.linalg.norm(x[:, None] - centers[None], axis=-1)
    near, idx = dist.min(axis=1), dist.argmin(axis=1)
    on = near <= 3 * width
    counts = np.bincount(idx[on], minlength=len(centers))
    covered = (counts >= on.sum() / len(centers) / 5).sum()
    return covered, on.mean(), near.mean()
```

train_grid は critic_step と generator_step を 10,000 ステップ（バッチサイズ 1,024）交互に実行する。オプティマイザには $\beta_1 = 0$ の Adam（回転ベクトル場において正の一次モーメントのモメンタムは安定領域を狭めうるため、R3GAN に倣う）を用い、2×10^{-3} から 2×10^{-4} へのコサイン学習率減衰を適用する。

```
def train_grid(centers, loss_D, loss_G, gamma, steps=10000, lr=2e-3,
               batch=1024, latent=64, seed=0, checkpoints=8):
    """Alternate critic and generator steps; at each checkpoint record the
    on-mode fraction of 2,000 samples, and the critic loss and E_q|grad D|^2
    averaged over the steps since the previous checkpoint."""
    torch.manual_seed(seed)
    dev, grid = centers.device, centers.cpu().numpy()
    net_G = d2l.make_mlp([latent, 256, 256, 2], nn.ReLU()).to(dev)
    net_D = d2l.make_mlp([2, 256, 256, 1], nn.LeakyReLU(0.2)).to(dev)
    opt_G, opt_D = [torch.optim.Adam(net.parameters(),  # lr on the device
                                     torch.tensor(lr, device=dev),
                                     betas=(0.0, 0.99),
                                     capturable=dev.type == 'cuda')
                    for net in (net_G, net_D)]
    scheds = [torch.optim.lr_scheduler.CosineAnnealingLR(opt, steps, lr / 10)
              for opt in (opt_G, opt_D)]
    logs, every = torch.zeros(2, device=dev), steps // checkpoints

    def step():
        real = sample_modes(centers, batch)
        with torch.no_grad():
            fake = net_G(torch.randn(batch, latent, device=dev))
        loss, _, grad_q = critic_step(net_D, opt_D, loss_D, real, fake, gamma)
        logs.add_(torch.stack([loss, grad_q]))  # summed on the device
        z = torch.randn(batch, latent, device=dev)
        generator_step(net_G, opt_G, loss_G, net_D,
                       sample_modes(centers, batch), z)
```

```
    step, trace = d2l.cuda_graphed(step, dev), []
    for t in range(1, steps + 1):
        step()
        for sched in scheds:
            sched.step()
        if t % every == 0:                    # checkpoint
            with torch.no_grad():
                x = net_G(torch.randn(2000, latent, device=dev)).cpu().numpy()
            trace.append((mode_stats(x, grid)[1], *(logs / every).tolist()))
            logs.zero_()
    return net_G, trace
```

ここでは 2×2 の要因実験として、2 つの目的関数（古典的な非飽和型 `NS` と `pairing`）に $\gamma = 0$（ペナルティなし）および $\gamma = 1$（$R_1 + R_2$）を掛け合わせた各条件を 3 シードずつ実行し、さらに過減衰の実行（$\gamma = 10$）を 1 つ加える。

```
centers, latent = grid_centers().to(d2l.try_gpu()), 64
ns, pairing = (gan_loss_D, gan_loss_G), (d2l.rpgan_loss_D, d2l.rpgan_loss_G)
arms = {'NS': (*ns, 0.0, 3), 'NS + R1 + R2': (*ns, 1.0, 3),
        'pairing': (*pairing, 0.0, 3), 'pairing + R1 + R2': (*pairing, 1.0, 3),
        'pairing + R1 + R2, gamma=10': (*pairing, 10.0, 1)}
runs = {name: [] for name in arms}              # per run: samples, trace
for name, (loss_D, loss_G, gamma, n_runs) in arms.items():
    for seed in range(n_runs):
        net_G, trace = train_grid(centers, loss_D, loss_G, gamma,
                                  latent=latent, seed=seed)
        with torch.no_grad():
            z = torch.randn(10000, latent, device=centers.device)
            runs[name].append((net_G(z).cpu().numpy(), trace))
grid = centers.cpu().numpy()
```

各実行から生成された 10,000 個のサンプルに対する `mode_stats` の結果を、クリティックの最終的な二乗勾配ノルム $\mathbb{E}_q[\|\nabla_x D(x')\|^2]$ と並べて表にまとめる。

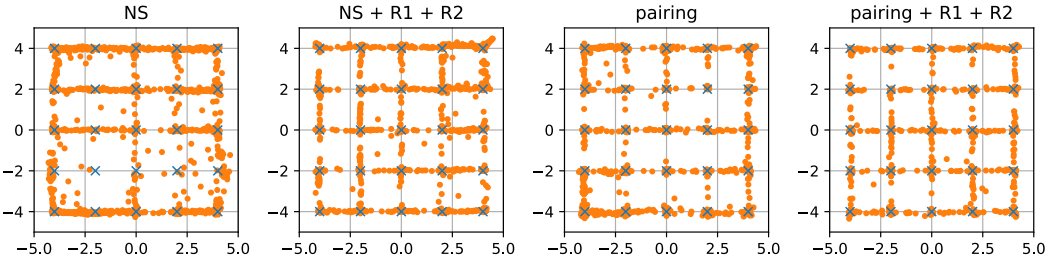
```
print(f'{"arm":28} run  modes  on-mode  distance  E_q|grad D|^2')
for name, results in runs.items():
    for seed, (x, trace) in enumerate(results):
        modes, on, dist = mode_stats(x, grid)
        print(f'{name:28} {seed:3}  {modes:2}/25  {on:7.2f}  {dist:8.2f}'
              f'  {trace[-1][2]:13.1e}')
```

```
arm                          run  modes  on-mode  distance  E_q|grad D|^2
NS                             0  16/25     0.70      0.20        1.7e+01
NS                             1  25/25     0.90      0.10        2.2e+00
NS                             2  24/25     0.76      0.16        5.6e+00
NS + R1 + R2                   0  25/25     0.77      0.15        8.9e-04
NS + R1 + R2                   1  25/25     0.83      0.13        6.8e-04
NS + R1 + R2                   2  25/25     0.86      0.11        3.6e-04
pairing                        0  25/25     0.82      0.13        3.9e+00
pairing                        1  15/25     0.76      0.17        1.2e+01
pairing                        2  23/25     0.80      0.15        5.1e+00
pairing + R1 + R2              0  25/25     0.86      0.12        3.3e-04
```

```
pairing + R1 + R2              1  25/25     0.89       0.09       1.9e-04
pairing + R1 + R2              2  25/25     0.85       0.12       5.0e-04
pairing + R1 + R2, gamma=10    0  23/25     0.19       0.46       1.2e-04
```

主要な 4 つの実験アームそれぞれの最初の実行から生成された 3,000 個のサンプルを、25 個のモード中心（grid）と重ねてプロットすると、モード被覆の違いが直接見て取れる。

```
fig, axes = d2l.plt.subplots(1, 4, figsize=(10, 2.9))
for ax, name in zip(axes, list(runs)[:4]):
    x = runs[name][0][0][:3000]              # run 0; centers drawn last
    d2l.plot([x[:, 0], grid[:, 0]], [x[:, 1], grid[:, 1]],
             fmts=('C1.', 'C0x'), xlim=(-5, 5), ylim=(-5, 5), axes=ax)
    ax.set_title(name)
    ax.set_aspect('equal')
fig.tight_layout()
```



この結果から、勾配ペナルティとペアリング損失それぞれの役割が明確に分離される。

- **勾配ペナルティがモード被覆をもたらす：**ペナルティなし（$\gamma = 0$）では、クリティックの勾配は急峻なままであり（$\mathbb{E}_q\|\nabla_x D\|^2 \in [0.7, 17]$）、モード被覆は不安定になる（シードによって 12〜25 モードに落ち込む）。$R_1 + R_2$ を $\gamma = 1$ で課すと、クリティックの勾配は 3 桁減衰し（$\sim 10^{-3}$）、NS と pairing の双方ですべての実行が全 25 モードを被覆する。
- **大きい γ における過減衰：**γ を 10 に増やすとクリティックが過度に平坦化され（$\sim 10^{-4}$）、収束が遅くなって 10,000 ステップ後でも 20〜30% のサンプルしかモードの 3σ 以内に収まらない。

最後に、チェックポイントのトレースを調べると、落ち着いたクリティック損失曲線の解釈になぜ注意が必要かがわかる。

```
traces = {name: np.array(results[0][1]) for name, results in runs.items()}
for col, stat in enumerate(['on-mode fraction', 'critic loss']):
    print(f'{stat} of run 0 at every 1,250 steps')
    for name, trace in traces.items():          # trace: (checkpoint, stat)
        print(f'  {name:28}', ' '.join(f'{v:4.2f}' for v in trace[:, col]))
```

```
on-mode fraction of run 0 at every 1,250 steps
  NS                           0.01 0.03 0.08 0.16 0.29 0.54 0.69 0.72
  NS + R1 + R2                 0.02 0.03 0.09 0.08 0.26 0.64 0.68 0.79
  pairing                      0.02 0.07 0.17 0.31 0.50 0.73 0.74 0.82
  pairing + R1 + R2            0.02 0.01 0.04 0.16 0.50 0.74 0.81 0.86
  pairing + R1 + R2, gamma=10  0.02 0.03 0.02 0.05 0.09 0.09 0.16 0.21
critic loss of run 0 at every 1,250 steps
  NS                           1.35 1.21 1.04 1.00 1.01 1.03 1.05 1.07
```

```
NS + R1 + R2                 1.38 1.37 1.36 1.37 1.38 1.38 1.38 1.38
pairing                      0.68 0.65 0.57 0.57 0.60 0.62 0.63 0.64
pairing + R1 + R2            0.68 0.68 0.67 0.68 0.69 0.69 0.69 0.69
pairing + R1 + R2, gamma=10  0.69 0.70 0.69 0.69 0.69 0.69 0.69 0.69
```

ペナルティ付きの実行では、クリティック損失は最初のチェックポイントから平衡値（`NS` では $2\log 2 \approx 1.39$、`pairing` では $\log 2 \approx 0.69$）の付近に留まる。これは生成サンプルのわずか 1〜3% しかモード上に乗っていない段階でさえ生じる現象であり、勾配ペナルティが至るところで $D(x)$ を小さく保つからである。クリティック損失が平衡値にあることはクリティックが平坦であることを示すにすぎず、必ずしも生成器が収束したことを意味しない。

16.4.5 まとめ

2 パラメータの Dirac-GAN（$p = \delta_0, q_\theta = \delta_\theta, D_\psi(x) = \psi x$）では、連続時間勾配フローは一定半径の円周上を周回し、離散時間の同時ステップは外向きに渦を巻いて発散し、離散時間の交互ステップは有界な帯状領域に閉じ込められる。ゼロ中心勾配ペナルティ $R_1 = \frac{\gamma}{2}\mathbb{E}_p[\|\nabla_x D(x)\|^2]$ および $R_2 = \frac{\gamma}{2}\mathbb{E}_q[\|\nabla_{x'} D(x')\|^2]$ を差し引くと、ヤコビ行列の固有値が開左半平面（$\mathrm{Re}(\nu_{1,2}) < 0$）へとすべての $\gamma > 0$ に対して移動し、平衡点は局所指数安定となる。平衡の近傍では、$R_1 + R_2$ は $p - q$ に対する空間ローパスフィルタ（ロジスティック損失のもとでのカットオフ長さスケールは $2\sqrt{\gamma}$）として働き、大きなスケールでの $2\,\mathrm{JS}(p, q)$ と細かなスケールでの二乗双対ソボレフノルム $\|p - q\|^2_{\dot{H}^{-1}(m)}$ の間を補間する。実験的にも、25 ガウスグリッドにおいて $R_1 + R_2$（$\gamma = 1$）はクリティックの急峻さを 3 桁減衰させ、古典的なロジスティック目的関数と相対論的ペアリング目的関数の双方で完全なモード被覆を達成した。

16.4.6 演習問題

1. [code] **Dirac-GAN の離散時間安定性**。ステップサイズ h の同時勾配降下法は $(\theta, \psi) \mapsto (\theta, \psi) + h\,\mathbf{u}(\theta, \psi)$ と写像する。
   1. $\gamma = 0$ のとき、$(0, 0)$ における離散更新のヤコビ行列は固有値 $1 \pm i\,h\,\ell'(0)$ を持ち、その絶対値が $\sqrt{1 + h^2 \ell'(0)^2} > 1$（任意の $h > 0$ に対して）となることを示せ。
   2. $\gamma > 0$ のとき、固有値は $1 + h\nu_{1,2}$（ただし $\nu_{1,2}$ は Equation 16.29 のもの）となる。不足減衰レジーム $\gamma < 2\ell'(0)$ において、離散更新が局所安定（$|1 + h\nu_{1,2}| < 1$）となるのは $h < \gamma / \ell'(0)^2$ のとき、かつそのときに限ることを示せ。
   3. $\gamma = 0.3$（$\ell'(0) = \frac{1}{2}$、しきい値 $h = 1.2$）におけるこのしきい値を、`trajectory(euler_step, (0.1, 0.1), h, 3000, gamma=0.3)` を $h = 1.1$ と $h = 1.3$ で実行して検証せよ。
2. **Dirac-GAN におけるペナルティ付き最適応答**。Dirac-GAN において $V_\gamma(\theta) = \sup_\psi\{V(\theta, \psi) - \gamma\psi^2\}$（$\ell = \log\sigma$）を考える。
   1. 小さい θ に対して、最大点を与える傾きは $\psi^\star \approx -\ell'(0)\theta/(2\gamma)$ かつ $V_\gamma(\theta) = 2\ell(0) + \frac{\ell'(0)^2}{4\gamma}\theta^2 + O(\theta^4)$ となり、Equation 16.31 において $W_2(\delta_0, \delta_\theta) = |\theta|$ とした結果に一致することを示せ。
   2. 次に、D を $\gamma \int m\,\|\nabla_x D\|^2$ でペナルティを課されたすべての微分可能関数にわたって動かすとする。点質量 $p = \delta_0$ と $q_\theta = \delta_\theta$（$\theta \neq 0$）に対して、$\sup_D\{V_\ell(D) - R_1 - R_2\} = 2\sup_t \ell(t)$ が θ に依存しないことを示せ。有限サイズのニューラルネットワークのクリティックが、それでもこの隔たりをまたいで非ゼロの勾配を与えるのはなぜか。
3. R_1 **単独のペナルティ対** $R_1 + R_2$。$p = \delta_0$、$q = \delta_1$、$\ell = \log\sigma$ とし、二次クリティック $D_c(x) = -cx^2$（$c \geq 0$）を考える。

   1. R_1 と R_2 を D_c に対して計算し、$V_\ell(D_c) - R_1$ が c に関して狭義単調増加であるため有限な最大点を持たないことを示せ。
   2. 非飽和生成器損失 $-\log\sigma(D_c(x'))$ のもとでは、$x' = 1$ における生成器勾配の大きさは c とともに際限なく増大する一方、$R_1 + R_2$ を差し引くと有限な最大点 $c^\star < 1/(8\gamma)$ が得られることを示せ。
4. [code] **Dirac-GAN における交互更新**。●交互更新では、$\psi_{k+1} = \psi_k + h\,u_\psi(\theta_k, \psi_k)$ に続けて $\theta_{k+1} = \theta_k + h\,u_\theta(\theta_k, \psi_{k+1})$ を適用する。
   1. 合成された更新を $(0,0)$ の周りで（$\gamma = 0$ として）線形化し、そのヤコビ行列の行列式が 1 であり、固有値が単位円上に乗るのは $h\,\ell'(0) \le 2$ のとき、かつそのときに限ることを示せ。
   2. `trajectory(alt_step, (0.1, 0.1), h, 10**5)` を $\gamma = 0$ のもとで $h \in \{0.2, 1, 3, 5\}$ に対して実行し、最後の 10^4 ステップにおける最小半径と最大半径を記録して、$\gamma = 0.3$（$h = 0.2$）の場合と比較せよ。
5. [code] **局所二次近似値とローパスフィルタリング**。●$\mathcal{A} = |\ell''(0)|\,m + \gamma\mathcal{L}_m$（ただし $\mathcal{L}_m D = -\nabla\cdot(m\nabla D)$）とおく。
   1. 部分積分を用いて

$$|\ell''(0)| \int m\,D^2 + \gamma \int m\,\|\nabla_x D\|^2 = \langle D, \mathcal{A} D\rangle$$

   を示し、半直線 tD_0 に沿って最大化してコーシー・シュワルツの不等式を適用することで Equation 16.30 を導出せよ。
   2. $p = \mathcal{N}(0,1)$、$q(x) = p(x)(1 + 0.1\sin(\omega x))$ とする。`penalized_value` を用いて、$\gamma \in \{0.01, 0.1, 1\}$ におけるペナルティ付きの値と $\gamma = 0$ におけるペナルティなしの値との比を $\omega \in \{0.5, 2, 8\}$ について計算し、一様な m を仮定したフーリエ減衰因子 $(1 + 4\gamma\omega^2)^{-1}$ と比較せよ。
6. [code] **Dirac-GAN における 1 中心ペナルティのダイナミクス**。●WGAN の線形利得 $\ell(t) = t\,(\ell'(0) = 1)$ において、クリティックの目的関数から 1 中心ペナルティ $\frac{\gamma}{2}(|\psi| - 1)^2$ を差し引いた系を考える。
   1. 同時勾配場が $\psi \neq 0$ では決してゼロにならないことを示し、生成器のパラメータ位置 θ のうち、$\psi \to 0^{\pm}$ としたときの片側極限がゼロになるものを求めよ。
   2. 1 中心ペナルティを用いるように `dirac_field` を変更し、`trajectory` を $(1,1)$ から $h = 0.01$ で 40,000 ステップ（$\gamma = 1$ および $\gamma = 3$）実行して、$\gamma = 3$ では生成器が $\theta = 3$ で停滞することを検証せよ。

16.5 敵対的画像生成

16.1 節の多層パーセプトロン（multilayer perceptron）は、識別可能な 28×28 の Fashion-MNIST のシルエットを生成できたものの、全結合層が空間的局所性（spatial locality）を無視するため、テクスチャはざらついていた。64×64 の RGB 画像を生成するには、100 次元の潜在ベクトルから 12,288 個のピクセル値を生成する必要がある。ここで 2 つの疑問が生じる。

1. 敵対的画像生成において、どのような畳み込みアーキテクチャであれば安定して訓練できるのか。
2. 尤度が利用できない状況で、画像生成器をどのように定量的に評価・比較できるのか。

畳み込みを用いた生成器と識別器は、Radford et al. (2015) の Deep Convolutional GAN（DCGAN）によって初めて GAN を自然画像へとスケールさせた。その後、汎用的な画像生成では拡散モデル（diffusion model、第 17 章）が主流となったが、畳み込み識別器と特徴空間での評価指標は、現代の画像トークナイザや 1 ステップの蒸留生成器（16.7 節）において依然として中心的な役割を担っている。

本節では、40,597 枚の 64×64 Pokémon スプライトからなるデータセット上でモデルを訓練する。まず 2015 年の古典的な DCGAN アーキテクチャを実装し、続いて R3GAN (Huang et al. 2024) に着想を得た最小構成の正規化なし畳み込みバックボーンを構築する。この正規化なしバックボーンを 3 つの目的関数（ペナルティなしの古典的ロジスティック損失、$R_1 + R_2$ ペナルティ付きの古典的ロジスティック損失、$R_1 + R_2$ ペナルティ付きの相対論的ペアリング損失）のもとで訓練することで、ゼロ中心勾配ペナルティ（zero-centered gradient penalty、16.4 節）がバッチ正規化（batch normalization）やその他のアーキテクチャ上の経験則を代替できるかを検証する。その後、特徴空間における標準的な評価指標であるフレシェ距離（Fréchet Distance、FID）、Kernel Inception Distance（KID）、および k-NN の適合率（precision）と再現率（recall）を実装し、潜在空間での補間と識別器の過学習（overfitting）を調べる。

```
import torchvision
```

16.5.1 DCGAN アーキテクチャ

DCGAN (Radford et al. 2015) は、16.1 節の古典的な非飽和ロジスティック目的関数を維持しつつ、以下の一連のアーキテクチャ規約によって畳み込みネットワークの訓練を安定化させる。

- 生成器における空間的なアップサンプリングには**転置畳み込み**（transposed convolution、分数ストライド畳み込み）を用い、識別器におけるダウンサンプリングにはプーリング層の代わりにストライド付き畳み込み（strided convolution）を用いる。
- 生成器の出力層と識別器の入力層を除くすべての畳み込みの後に、**バッチ正規化**を適用する。
- 生成器は隠れ層の活性化関数に ReLU、出力の活性化関数に tanh（$[-1, 1]$ のピクセル値に対応）を用いるのに対し、識別器は勾配が 0 になるデッドゾーンを防ぐため、全体にわたって Leaky ReLU（slope $= 0.2$）を用いる。
- 重みは $\mathcal{N}(0, 0.02^2)$ から初期化し、Adam では通常より小さいモメンタム $\beta_1 = 0.5$ を用いる。

16.5.1.1 Pokémon スプライトデータセット

ここで用いるデータセットは、pokemondb.net から取得した 721 種の Pokémon の RGB スプライト 40,597 枚で構成されており、64×64 にリサイズしたうえで $[-1, 1]$ にスケーリングしてある。データセット全体がメモリに収まるため、`load_pokemon` で単一のテンソルへとデコードする。

```
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')


def load_pokemon(resolution=64):
    """All Pokemon sprites as one (n, 3, h, w) tensor in [-1, 1]."""
    import warnings
    transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize((resolution, resolution)),
        torchvision.transforms.PILToTensor()])
    dataset = torchvision.datasets.ImageFolder(
        d2l.download_extract('pokemon'), transform=transform)
    loader = torch.utils.data.DataLoader(
        dataset, batch_size=512, num_workers=d2l.get_dataloader_workers())
    with warnings.catch_warnings():  # palette sprites with transparency
```

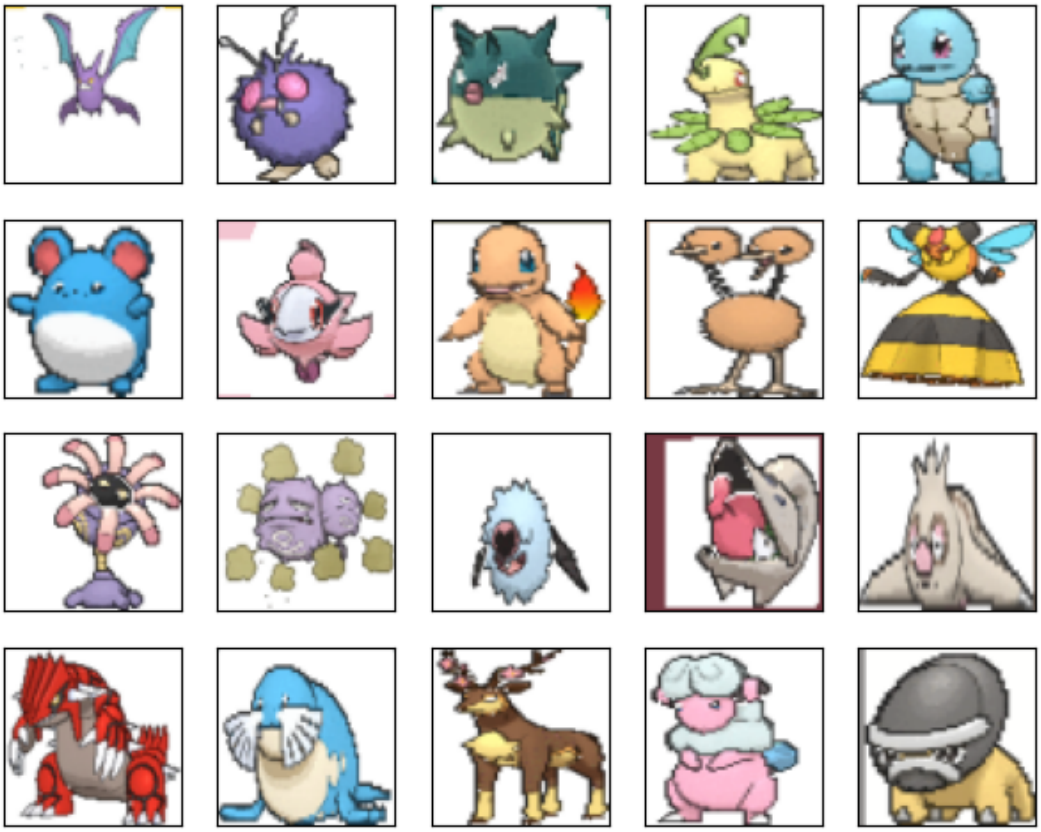
```
        warnings.filterwarnings('ignore', message='Palette images')
        return torch.cat([X for X, _ in loader]).float() / 127.5 - 1

images = load_pokemon()
images.shape
```

```
torch.Size([40597, 3, 64, 64])
```

固定された順列によって画像の 10 分の 1 を取り分けておき、これらは訓練には一切使用しない。この画像は、生成画像をスコアリングする際の比較対象となる実サンプルであり、またクリティックが未知の画像よりも訓練画像に高いスコアを付けていないかを確かめる指標にもなる。分割は種ごとではなく画像ごとに行うため、同じ種の別バリエーションが訓練側とホールドアウト側の両方に含まれることがあり、訓練スプライトを丸写しした生成器がホールドアウト側の別バリエーションの近くに着地する可能性はある。

```
device = d2l.try_gpu()
perm = np.random.default_rng(12345).permutation(len(images))
n_holdout = len(images) // 10
train_imgs = images[perm[n_holdout:]].to(device)
holdout_imgs = images[perm[:n_holdout]].to(device)
print(f'{len(train_imgs)} training and {len(holdout_imgs)} held-out images')
d2l.show_images(train_imgs[:20].cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=4, num_cols=5);
```

```
36538 training and 4059 held-out images
```



16.5.1.2 転置畳み込みとバッチ正規化

100 チャネルの 1×1 特徴マップを 64×64 の画像へとアップサンプリングするため、DCGAN は**転置畳み込み**（32.10 節（英語版））を用いる。カーネルサイズ k、ストライド s、パディング p の転置畳み込みは、$n \times n$ の空間グリッドを次の辺長を持つ出力へと写像する。

$$s(n-1) + k - 2p. \tag{16.32}$$

$k = 4, s = 2, p = 1$ とすると、各層は空間解像度を 2 倍にする（$n \to 2n$）。最初の層（$k = 4, s = 1, p = 0$）では、1×1 の入力が 4×4 へと拡大される。逆に、$k = 4, s = 2, p = 1$ のストライド付き畳み込みは空間解像度を半分にする。

Figure 16.6 に示すように、DCGAN の生成器は 5 つの転置畳み込みを連ねて解像度を $1 \to 4 \to 8 \to 16 \to 32 \to 64$ と拡大し、各ステージでチャネル幅を半分にしながら最終的に 3 つの RGB チャネルを出力する。識別器はこの階層構造を反転させ、ストライド付き畳み込みによって単一の 1×1 ロジットへと集約する。

```
def G_block(c_in, c_out, stride=2, padding=1):
    """Transposed convolution (kernel 4), batch norm, ReLU."""
    return nn.Sequential(
        nn.ConvTranspose2d(c_in, c_out, 4, stride, padding, bias=False),
        nn.BatchNorm2d(c_out), nn.ReLU())
```

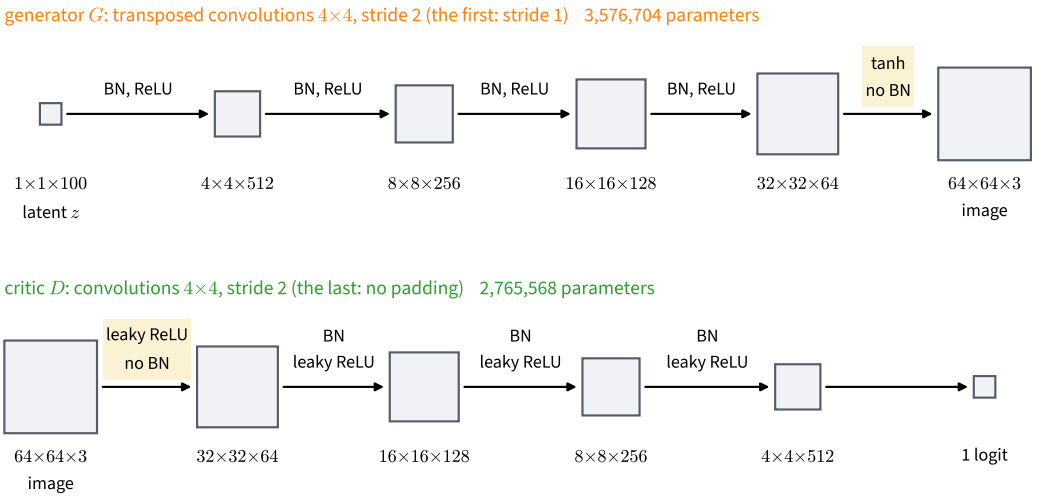


Figure 16.6: DCGAN アーキテクチャ (Radford et al. 2015)（基本チャネル幅 512、4×4 カーネルの場合)。生成器は 100 次元の潜在ベクトルを転置畳み込みによって 64×64 の RGB 画像へと写像し、クリティックはストライド付き畳み込みによってこの階層を逆にたどる。生成器の出力層とクリティックの入力ブロックを除くすべてのブロックでバッチ正規化が用いられる。

```
def D_block(c_in, c_out, norm=True):
    """Strided convolution (kernel 4), batch norm, leaky ReLU."""
    return nn.Sequential(
        nn.Conv2d(c_in, c_out, 4, 2, 1, bias=False),
        nn.BatchNorm2d(c_out) if norm else nn.Identity(), nn.LeakyReLU(0.2))
```

4 段の `G_block` に最終段の転置畳み込みと `tanh` を重ねて `net_G` を構成し、4 段の `D_block`（最初のブロックではバッチ正規化を省く）と 4×4 畳み込みによって `net_D` を構成する。2 個の 100 次元潜在ベクトルに対する順伝播で、それぞれの出力形状を確認しよう。

```
torch.manual_seed(0)
net_G = nn.Sequential(                                  # (100,)
    nn.Unflatten(1, (100, 1, 1)),                       # (100, 1, 1)
    G_block(100, 512, stride=1, padding=0),             # (512, 4, 4)
    G_block(512, 256), G_block(256, 128), G_block(128, 64),  # (64, 32, 32)
    nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False), nn.Tanh()).to(device)
net_D = nn.Sequential(                                  # (3, 64, 64)
    D_block(3, 64, norm=False),                         # (64, 32, 32)
    D_block(64, 128), D_block(128, 256), D_block(256, 512),  # (512, 4, 4)
    nn.Conv2d(512, 1, 4, bias=False)).to(device)        # (1, 1, 1)
X = net_G(torch.zeros(2, 100, device=device))
print(X.shape, net_D(X).shape)
```

```
torch.Size([2, 3, 64, 64]) torch.Size([2, 1, 1, 1])
```

16.5.1.3 古典的損失による訓練

クリティックのロジスティック回帰ステップと生成器の非飽和ステップを交互に繰り返す `d2l.train_gan`（16.1 節）のループは、画像にもそのまま適用できる。ここでは DCGAN の設定に従いつつ、ミニバッチサイズを元論文の 128 枚より大きい 256 枚とし、学習率 $2 \cdot 10^{-4}$、$\beta_1 = 0.5$、畳み込みカーネルの初期値を $\mathcal{N}(0, 0.02^2)$ からの抽出とする。また、著者らの公開コードと同様に、バッチ正規化のスケールは $\mathcal{N}(1, 0.02^2)$ から、オフセットは 0 から開始する。

```
def dcgan_init(module):
    if isinstance(module, (nn.Conv2d, nn.ConvTranspose2d)):
        nn.init.normal_(module.weight, 0, 0.02)
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.normal_(module.weight, 1, 0.02)
        nn.init.zeros_(module.bias)

for net in (net_D, net_G):
    net.apply(dcgan_init)
dataset = torch.utils.data.TensorDataset(train_imgs)
# Sample whole minibatches of indices, so that the GPU tensor is indexed
# once per minibatch rather than once per image
sampler = torch.utils.data.BatchSampler(
    torch.utils.data.RandomSampler(dataset), 256, drop_last=True)
data_iter = torch.utils.data.DataLoader(dataset, sampler=sampler,
                                        batch_size=None)
```

サンプリングには、生成器の最終ステップの重みではなく直近の重みの平均を用いる。`EMA` はパラメータの指数移動平均（exponential moving average）を保持し、各ステップの後に次の式で更新する。

$$\bar{\theta} \leftarrow \beta \bar{\theta} + (1 - \beta)\, \theta, \qquad \beta = 2^{-1/n_{1/2}}, \tag{16.33}$$

これにより、過去のステップの重みは**半減期**（half-life）である $n_{1/2}$ ステップごとに半減する。ゲームの反復解は均衡点の周りを回転するため（16.4 節）、個々の反復解よりもその平均のほうが均衡点の近くに位置する。なお、バッチ正規化の統計量はパラメータではなく、移動平均モデルは訓練中のネットワークから統計量をそのままコピーする。しかし、コピーされた統計量は平均化された重みではなく訓練中の重みに対応しているため、訓練終了後に再推定する。

```
class EMA:
    """Exponential moving average of a model's parameters, given as a
    half-life in steps. Buffers such as batch-norm statistics are copied;
    the average is kept in evaluation mode."""
    def __init__(self, model, half_life):
        import copy  # the d2l library preamble does not import it
        self.model = copy.deepcopy(model).requires_grad_(False).eval()
        self.decay = 0.5 ** (1 / half_life)

    def update(self, model):
        with torch.no_grad():
            for avg, p in zip(self.model.parameters(), model.parameters()):
                avg.lerp_(p, 1 - self.decay)
            for avg, b in zip(self.model.buffers(), model.buffers()):
```

```
                avg.copy_(b)
```

本節の後半の実験では 32,000 枚の画像にわたって平均をとり、これはバッチサイズ 64 では 500 ステップの半減期に相当する。バッチサイズ 256 では、同じ 32,000 枚の画像は 125 ステップに相当し、今回の訓練全体の 2,840 ステップ（20 エポック、各 142 ステップ）の 20 分の 1 未満である。

出力されるクリティック損失は 16.1 節における 2 つのバッチ平均の和であり、均衡点では $2\log 2 \approx 1.39$ に等しく、そこでの生成器の損失は $\log 2 \approx 0.69$ に等しくなる。

```
ema = EMA(net_G, half_life=125)  # 32,000 images, as in the later runs
history = d2l.train_gan(net_D, net_G, data_iter, num_epochs=20,
                        latent_dim=100, lr_D=0.0002, lr_G=0.0002, seed=1,
                        ema=ema)
loss_D, loss_G = history[-1]
print(f'last epoch: loss_D {loss_D:.3f}, loss_G {loss_G:.3f}')
```

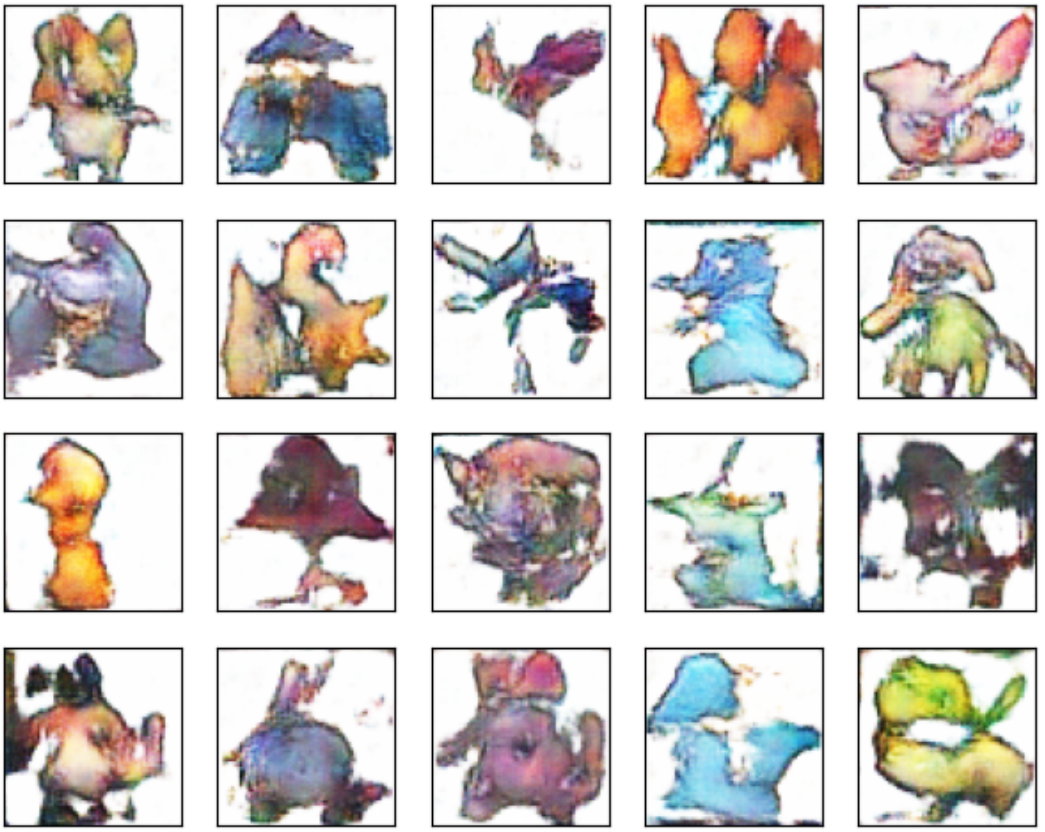
```
last epoch: loss_D 0.252, loss_G 3.188
```

次のセルでは、平均化された生成器を訓練モードにして新規の潜在コード 100 バッチを入力し、移動平均モデルのバッチ正規化統計量を再推定する。移動平均の統計量は 1 バッチあたり 0.1 の割合で過去の値を忘却するため、コピーされた値に残る重みは $0.9^{100} \approx 3 \cdot 10^{-5}$、すなわち実質的に 0 となる。

```
ema.model.train()  # batch-norm statistics for the averaged weights
with torch.no_grad():  # 100 batches; running averages forget at rate 0.1
    for _ in range(100):
        ema.model(torch.randn(256, 100, device=device))
ema.model.eval();
```

各画像が自身の潜在コードのみに依存するよう、サンプリングは評価モードで行う。

```
with torch.no_grad():
    fake = ema.model(torch.randn(20, 100, device=device))
d2l.show_images(fake.cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=4, num_cols=5);
```



生成されたスプライトは、やや斑状のテクスチャが見られるものの、白背景の上に多様なシルエットと配色を示している。なお、識別器の損失（~ 0.25–0.45）は $2\log 2 \approx 1.39$ を大きく下回ったままであるが、モード崩壊（mode collapse）は引き起こしていない。

16.5.2 正規化なしバックボーン

DCGAN 以降の数年間で、大規模な画像 GAN には数多くのアーキテクチャ上の安定化手法が積み重ねられていった（StyleGAN2 (Karras, Laine, et al. 2020) は 10 種類の個別技術を組み合わせていた）。これに対し R3GAN (Huang et al. 2024) は、目的関数を $R_1 + R_2$ 勾配ペナルティと相対論的ペアリング損失で正則化すれば、そうした場当たり的なアーキテクチャ上の工夫を取り除けることを実証した。とりわけ、R_1 と R_2 を用いる際には識別器からバッチ正規化を省くことが不可欠である。バッチ正規化はミニバッチ内のサンプル同士を結合してしまうため、サンプルごとの入力勾配の計算が成り立たなくなるからである。

16.5.2.1 アーキテクチャとファンイン初期化

最小構成の畳み込みバックボーン（Figure 16.7）を、正規化層も出力の tanh も用いずに構築する。

- 生成器は、学習される $4\times4\times128$ の定数テンソルと z の線形射影（$4\times4\times100$、さらに 16.6 節のクラス条件付けで用いる任意の追加チャネル）を連結し、4 回（$4 \to 8 \to 16 \to 32 \to 64$）にわたって双線形補間（bilinear interpolation）とそれに続く 3×3 畳み込みおよび Leaky ReLU（slope $= 0.2$）でアップサンプリングする。双線形アップサンプリングの後に 3×3

畳み込みを適用することで、転置畳み込みの不均一な重なりによって生じるチェッカーボードアーティファクト（Figure 16.8）を回避できる (Odena et al. 2016)。
- クリティックはこの構造を反転させ、3×3 畳み込み、Leaky ReLU、および双線形ダウンサンプリングによって 4×4 の特徴マップ $\phi(x)$ まで縮小し、最後に線形ヘッドを通す。
- バッチ正規化なしで各層にわたって活性化値の分散を保つため、すべての重みには He のファンイン初期化 (He et al. 2015) を傾き 0.2 の Leaky ReLU 向けにスケーリングして用いる（分散は $2/((1+0.2^2)\, n_{\text{in}})$）。

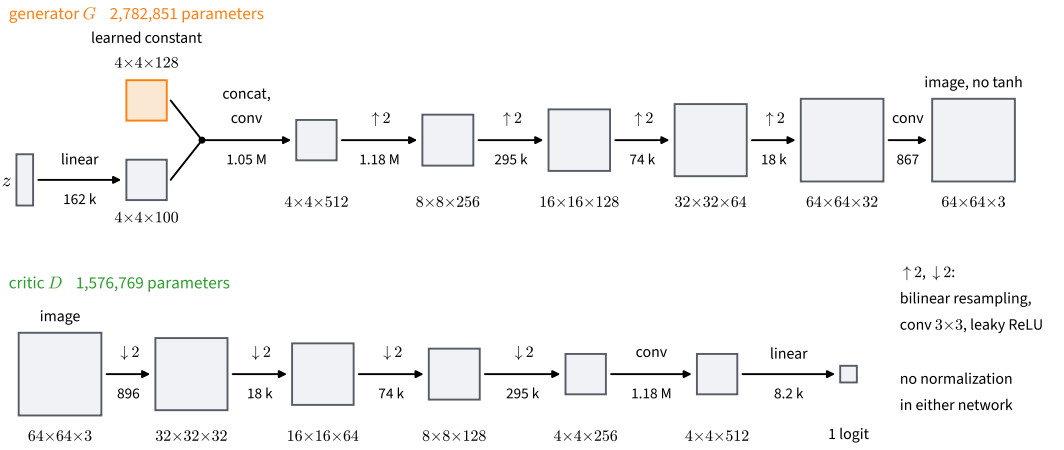


Figure 16.7: 正規化なし畳み込みバックボーン。生成器は学習される $4 \times 4 \times 128$ の定数と射影された潜在ベクトルを組み合わせ、双線形アップサンプリング、3×3 畳み込み、Leaky ReLU からなる 4 つのステージを適用する。クリティックは双線形ダウンサンプリングによってこの階層構造を逆にたどり、線形ヘッドへとつなぐ。

```
def fan_in_init(module):
    """Fan-in normal weights with the leaky-ReLU gain, zero offsets."""
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(module.weight, a=0.2)
        nn.init.zeros_(module.bias)


class Generator(nn.Module):
    """Learned 4x4 constant + projected latent (+ extra 4x4 input maps),
    bilinearly upsampled to resolution x resolution; no normalization."""
    def __init__(self, latent_dim, resolution, base_ch, *, extra_ch=0,
                 const_ch=128):
        super().__init__()
        self.latent_dim = latent_dim
        self.const = nn.Parameter(torch.randn(1, const_ch, 4, 4))
        self.z_proj = nn.Linear(latent_dim, 16 * latent_dim)
        self.mix = nn.Conv2d(const_ch + latent_dim + extra_ch, base_ch, 3,
                             padding=1)
        num_stages = int(np.log2(resolution / 4))       # 64: four doublings
        chans = [base_ch // 2 ** i for i in range(num_stages + 1)]
        self.stages = nn.ModuleList(
            [nn.Conv2d(c_in, c_out, 3, padding=1)
```

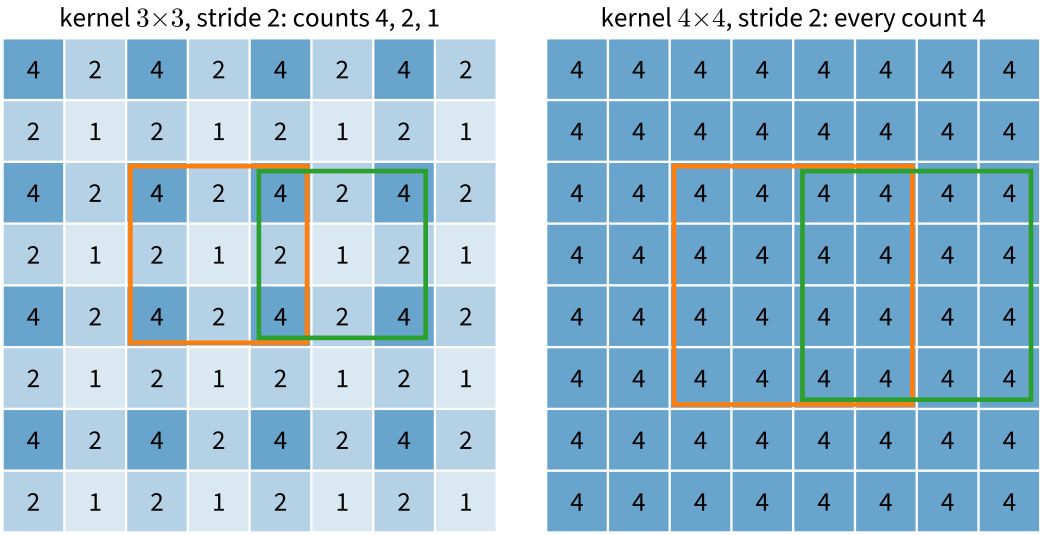


Figure 16.8: ストライド 2 の転置畳み込みにおける重なり回数。各入力ピクセルはカーネルのコピーを 1 つ出力上に押し当て、各数値は（境界から離れた位置で）その出力ピクセルを覆うカーネルの個数を表す。隣接する 2 つのカーネルの範囲を枠線で示している。3×3 カーネル（左）では重なり回数が 4、2、1 の間で交互に変わるため、定数の重みであってもチェッカーボード模様が描かれる。一方、サイズがストライドの倍数である 4×4 カーネル（右）では、すべてのピクセルが 4 回ずつ覆われる。

```
            for c_in, c_out in zip(chans[:-1], chans[1:])])
        self.to_rgb = nn.Conv2d(chans[-1], 3, 3, padding=1)
        self.act = nn.LeakyReLU(0.2)
        self.apply(fan_in_init)

    def forward(self, z, extra=None):
        zc = self.z_proj(z).reshape(-1, self.latent_dim, 4, 4)
        const = self.const.expand(z.shape[0], -1, -1, -1)
        maps = [const, zc] + ([] if extra is None else [extra])
        x = self.act(self.mix(torch.cat(maps, dim=1)))
        for conv in self.stages:                    # 4 -> 8 -> ... -> 64
            x = F.interpolate(x, scale_factor=2, mode='bilinear')
            x = self.act(conv(x))
        return self.to_rgb(x)                       # no tanh: raw output
```

クリティックは生成器と対称な構造を持つ。畳み込み、Leaky ReLU、双線形ダウンサンプリングからなる 4 つのステージで 64×64 の画像を 4×4 へと縮小し、混合用の畳み込みと線形ヘッドによって 1 つのスコアを返す。

```
class Discriminator(nn.Module):
    """Critic that mirrors the generator: conv + leaky ReLU + bilinear
    downsampling to a 4x4 feature map phi(x), then a linear head on the
    flattened map."""
```

```
    def __init__(self, resolution, base_ch):
        super().__init__()
        num_stages = int(np.log2(resolution / 4))
        chans = [3] + [base_ch * 2 ** i for i in range(num_stages)]
        self.stages = nn.ModuleList(
            [nn.Conv2d(c_in, c_out, 3, padding=1)
             for c_in, c_out in zip(chans[:-1], chans[1:])])
        self.mix = nn.Conv2d(chans[-1], 2 * chans[-1], 3, padding=1)
        self.head = nn.Linear(2 * chans[-1] * 4 * 4, 1)
        self.act = nn.LeakyReLU(0.2)
        self.apply(fan_in_init)

    def phi(self, x):
        for conv in self.stages:                    # 64 -> 32 -> ... -> 4
            x = F.interpolate(self.act(conv(x)), scale_factor=0.5,
                              mode='bilinear', antialias=True)
        return self.act(self.mix(x))

    def forward(self, x):
        return self.head(self.phi(x).flatten(1))

for net in (Generator(latent_dim=100, resolution=64, base_ch=512),
            Discriminator(resolution=64, base_ch=32)):
    print(f'{type(net).__name__}: {d2l.num_params(net)} parameters')
```

```
Generator: 2782851 parameters
Discriminator: 1576769 parameters
```

パラメータ数は、生成器が ~ 2.8M、クリティックが ~ 1.6M である。

16.5.3 統制された損失の比較

16.5.3.1 3 つの実験群

正規化なしバックボーンを同一の初期重みから 3 回訓練する。Adam（$\beta_1 = 0, \beta_2 = 0.99, \mathrm{lr} = 2 \times 10^{-4}$）、バッチサイズ 64、ランダム水平反転、および EMA（half-life $= 500$ ステップ）を共通に用い、次の訓練目的関数のみを変化させる。

$$L(D) + \frac{\gamma}{2}\Big(\mathbb{E}_x\big[\|\nabla_x D(x)\|^2\big] + \mathbb{E}_{x'}\big[\|\nabla_{x'} D(x')\|^2\big]\Big), \tag{16.34}$$

ここで、$L(D)$ は古典的ロジスティック損失（`d2l.gan_loss_D`）または相対論的ペアリング損失（`d2l.rpgan_loss_D`）のいずれかであり、生成器はそれに対応する非飽和損失を用いる。各実験群の設定は次のとおりである。

実験群	クリティック損失	勾配ペナルティ	ステップ数
NS	古典的ロジスティック	$\gamma = 0$（なし）	6,000
NS + R1 + R2	古典的ロジスティック	$\gamma = 10$	12,000
RpGAN + R1 + R2	相対論的ペアリング	$\gamma = 10$	12,000

最初の 2 つの実験群を比較することで $R_1 + R_2$ 勾配ペナルティの効果が切り分けられ（ペナルティなしの実験群は 6,000 ステップよりかなり前に崩壊するため、6,000 ステップで打ち切る）、

2 番目と 3 番目の実験群を比較することでペアワイズランキングの効果が切り分けられる（演習問題 1 では、画像解像度に対する γ のスケーリングを解析する）。訓練ループは保存用の 2 つの補助関数で支える。すなわち、`sample_real` は水平反転を伴うランダムなミニバッチを抽出し、`pair_stats` はペアワイズランキング正解率 $\hat{P}(D(x) > D(x'))$ と平均の相対論的ランク重み $w_{\mathrm{Rp}} = \mathbb{E}[\sigma(D(x') - D(x))]$ を計算する。

```
def sample_real(images, n, labels=None):
    """Draw a training batch with random horizontal flips; return it with
    its labels (None without labels)."""
    idx = torch.randint(0, len(images), (n,), device=images.device)
    batch = images[idx]
    flip = torch.rand(n, device=images.device) < 0.5
    batch = torch.where(flip.view(-1, 1, 1, 1), batch.flip(-1), batch)
    return batch, (None if labels is None else labels[idx])


def pair_stats(net_D, real, fake, y=None):
    """Ranking accuracy P(D(x) > D(x')) and mean rank weight
    E[sigma(D(x') - D(x))] over aligned (real, generated) pairs; given
    labels y, the critic is D(., y)."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    with torch.no_grad():
        diff = critic(fake) - critic(real)
    return float((diff < 0).float().mean()), float(torch.sigmoid(diff).mean())
```

保存されるループ `train_r3gan` は、反転された実画像バッチ上で `d2l.critic_step` と `d2l.generator_step`（16.4 節）を交互に実行し、各ステップの後に EMA 生成器を更新する。クリティックのステップは R_1 と R_2 のバッチ平均を返すため、実画像に対するクリティックの急峻さ $\mathbb{E}_x\|\nabla_x D(x)\|^2$ はすべての実験群で記録される（最初の実験群では重み $\gamma = 0$）。さらに 250 ステップごとに、新規の 256 ペアに対して `pair_stats` を適用する。

```
def train_r3gan(net_G, net_D, loss_D, loss_G, gamma, num_steps, images,
                labels=None, *, eval_images=None, eval_labels=None,
                eval_size=256, batch_size=64, lr=2e-4, half_life=500,
                log_every=250, seed=None):
    """Train with Adam (beta_1 = 0), alternating critic_step and
    generator_step on flipped real batches; a generated batch takes the
    labels of its real batch, if any. Every log_every steps, log the losses,
    R1, R2 and pair_stats on fresh pairs from eval_images (default: images).
    Return the EMA of the generator and the history, a dict of arrays."""
    if seed is not None:
        torch.manual_seed(seed)
    if eval_images is None:
        eval_images, eval_labels = images, labels
    opt_G, opt_D = [torch.optim.Adam(net.parameters(), lr, betas=(0.0, 0.99))
                    for net in (net_G, net_D)]
    noise = lambda n: torch.randn(n, net_G.latent_dim, device=images.device)
    generate = lambda z, y: net_G(z) if y is None else net_G(z, y)
    ema = EMA(net_G, half_life)
    history = {k: [] for k in ('step', 'loss_D', 'loss_G', 'R1', 'R2',
                               'accuracy', 'rank_weight')}
```

```
    for step in range(1, num_steps + 1):
        X, y = sample_real(images, batch_size, labels)
        with torch.no_grad():
            fake = generate(noise(batch_size), y)
        loss_d, r1, r2 = d2l.critic_step(net_D, opt_D, loss_D, X, fake,
                                         gamma, y)
        X, y = sample_real(images, batch_size, labels)
        loss_g = d2l.generator_step(net_G, opt_G, loss_G, net_D, X,
                                    noise(batch_size), y)
        ema.update(net_G)
        if step % log_every == 0:  # fresh pairs of real and generated images
            X, y = sample_real(eval_images, eval_size, eval_labels)
            with torch.no_grad():
                fake = generate(noise(eval_size), y)
            acc, weight = pair_stats(net_D, X, fake, y)
            logs = (step, loss_d, loss_g, r1, r2, acc, weight)
            for k, v in zip(history, logs):
                history[k].append(float(v))
    return ema.model, {k: np.array(v) for k, v in history.items()}
```

train_arm は、すべての実験群で同じシードから 2 つのネットワークを構築して訓練し、クリティックの訓練・ホールドアウトギャップ（1,024 枚の訓練画像に対する平均スコアと 1,024 枚のホールドアウト画像に対する平均スコアの差）を返す。

```
def train_arm(loss_D, loss_G, gamma, num_steps):
    torch.manual_seed(0)  # the same initialization in every arm
    net_G = Generator(100, 64, 512).to(device)
    net_D = Discriminator(64, 32).to(device)
    ema_G, history = train_r3gan(net_G, net_D, loss_D, loss_G, gamma,
                                 num_steps, train_imgs, seed=1)
    with torch.no_grad():
        gap = (net_D(train_imgs[:1024]).mean()
               - net_D(holdout_imgs[:1024]).mean())
    return ema_G, history, float(gap)
```

ペナルティ付きのステップは、クリティックの入力勾配をもう一度微分するため、ペナルティなしのステップよりも計算コストがかかる。各実験の終了後に、最後の 2,000 ステップにおける 8 回の記録にわたって平均したクリティックの損失とペナルティ、および訓練・ホールドアウトギャップを表示する。

```
arms = {'NS': (d2l.gan_loss_D, d2l.gan_loss_G, 0, 6000),
        'NS + R1 + R2': (d2l.gan_loss_D, d2l.gan_loss_G, 10, 12000),
        'RpGAN + R1 + R2': (d2l.rpgan_loss_D, d2l.rpgan_loss_G, 10, 12000)}
runs = {}
for name, (loss_D, loss_G, gamma, num_steps) in arms.items():
    ema_G, hist, gap = train_arm(loss_D, loss_G, gamma, num_steps)
    runs[name] = (ema_G, hist)
    loss, r1, r2 = (hist[k][-2000 // 250:].mean()  # the last 2,000 steps
                    for k in ('loss_D', 'R1', 'R2'))
    print(f'{name:16s} critic loss {loss:.2f}, '
          f'penalty {gamma / 2 * (r1 + r2):.3f}, train-holdout gap {gap:+.2f}')
```

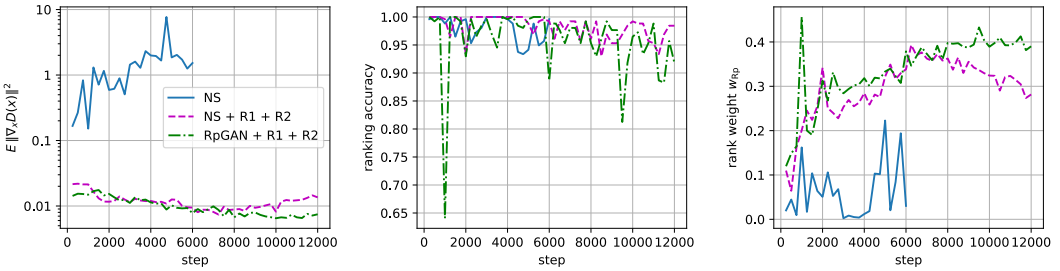
```
NS               critic loss 0.57, penalty 0.000, train-holdout gap +0.08
NS + R1 + R2     critic loss 1.04, penalty 0.127, train-holdout gap +0.05
RpGAN + R1 + R2  critic loss 0.51, penalty 0.077, train-holdout gap +0.02
```

ペナルティ付きの 2 つのクリティックはいずれも均衡値をやや下回る水準に落ち着くのに対し（古典的ロジスティックでは 1.04 対 $2\log 2 \approx 1.39$、ペアリングでは 0.47–0.51 対 $\log 2 \approx 0.69$）、ペナルティなしのクリティックは 0.57–0.58 まで低下する。

16.5.3.2 訓練の推移と生成サンプル

クリティックの二乗入力勾配 $\mathbb{E}_x[\|\nabla_x D(x)\|^2]$、ペアワイズランキング正解率 $\hat{P}(D(x) > D(x'))$、および相対論的ランク重み $w_{\text{Rp}} = \mathbb{E}[\sigma(D(x') - D(x))]$ を 250 ステップごとにプロットすると、勾配ペナルティがどのように訓練を安定化させるかがわかる。

```
fig, axes = d2l.plt.subplots(1, 3, figsize=(12, 3.2))
hists = [hist for _, hist in runs.values()]
for ax, key, ylabel, yscale in zip(
        axes, ('R1', 'accuracy', 'rank_weight'),
        (r'$E\,\|\nabla_x D(x)\|^2$', 'ranking accuracy',
         r'rank weight $w_{\mathrm{Rp}}$'),
        ('log', 'linear', 'linear')):
    d2l.plot([h['step'] for h in hists], [h[key] for h in hists], 'step',
             ylabel, yscale=yscale, axes=ax)
axes[0].legend(list(runs), loc='center right')
fig.tight_layout()
```



$R_1 + R_2$ がない場合、クリティックの二乗勾配ノルムは 2 桁上昇する（1–8 に達するのに対し、ペナルティ付きクリティックでは ~ 0.01）。3 つのクリティックはいずれもペアの $> 90\%$ を正しく順序付けるが、ペナルティ付きクリティックは小さなスコアマージンを維持するのに対し（$w_{\text{Rp}} \in [0.25, 0.40]$、ロジット差の中央値で 0.7 前後に相当）、ペナルティなしクリティックは w_{Rp} を 0.15 未満へと押し下げる。

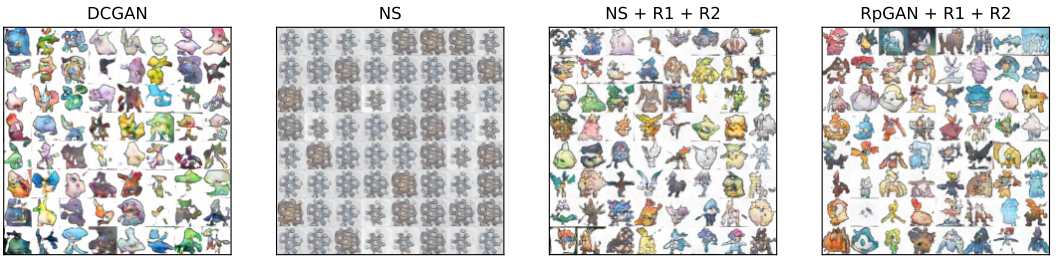
サンプル品質を視覚的に比較するため、まず $[-1, 1]$ の画像バッチを単一の $[0, 1]$ グリッドに並べる補助関数 `tile_images` を定義する。

```
def tile_images(imgs, num_rows):
    """Tile images (n, 3, h, w) in [-1, 1] into one (rows * h, cols * w, 3)
    array in [0, 1]."""
    n, c, h, w = imgs.shape
    grid = (imgs.clamp(-1, 1) + 1) / 2
    grid = grid.reshape(num_rows, n // num_rows, c, h, w)
```

```
    grid = grid.permute(0, 3, 1, 4, 2)  # (rows, h, cols, w, c)
    return grid.reshape(num_rows * h, -1, c).cpu().numpy()
```

それでは、4 つの EMA 生成器すべて（DCGAN と正規化なしの 3 つの実験群）を同じ 64 個の潜在ベクトル z_show 上で評価し、その出力を 8×8 のグリッドに並べてみよう。

```
torch.manual_seed(42)
z_show = torch.randn(64, 100, device=device)
gens = {'DCGAN': ema.model,
        **{name: ema_G for name, (ema_G, _) in runs.items()}}
with torch.no_grad():
    grids = [tile_images(G(z_show), 8) for G in gens.values()]
d2l.show_images(grids, 1, 4, titles=list(gens), scale=3.4);
```



バッチ正規化も勾配ペナルティもない場合、NS 実験群は完全なモード崩壊を起こす（64 個すべての潜在コードにわたって一様な灰色のテクスチャを生成する。演習問題 4 を参照）。一方、$R_1 + R_2$ を加えることで（NS + R1 + R2 および RpGAN + R1 + R2）崩壊が防がれ、きれいな白背景の中央に多様なスプライトのシルエットが生成される。

16.5.4 定量的評価指標

壊滅的な崩壊は目視でも容易に見抜けるが、正常に機能している 2 つの生成器を比較するには定量的な指標が必要である。標準的な GAN の評価では、n 枚の実画像と n 枚の生成画像を事前学習済みの特徴抽出器 $\phi(x) \in \mathbb{R}^d$ で埋め込み、得られた 2 つの経験特徴量分布の間の乖離を 16.2 節の閉形式の距離によって測定する。

16.5.4.1 FID と KID

1. **Fréchet Inception Distance（FID）**（Heusel et al. 2017）は、実画像と生成画像の特徴量の点群に多変量ガウス分布 $\mathcal{N}(\mu_p, \Sigma_p)$ と $\mathcal{N}(\mu_q, \Sigma_q)$ を当てはめ、両者の間の二乗 2 次ワッサースタイン距離（Wasserstein-2 distance）を計算する（Dowson and Landau 1982; Givens and Shortt 1984）。

$$\begin{aligned}\mathrm{FID} = &\left\|\mu_p - \mu_q\right\|^2 \\ &+ \mathrm{tr}\left(\Sigma_p + \Sigma_q - 2\left(\Sigma_p^{1/2}\,\Sigma_q\,\Sigma_p^{1/2}\right)^{1/2}\right).\end{aligned} \tag{16.35}$$

FID は特徴量分布の最初の 2 つのモーメントのみに依存し、そのプラグイン標本共分散推定量は有限サンプルサイズにおいて正の $O(1/n)$ の統計的バイアスを持つ（演習問題 3）。

2. **Kernel Inception Distance（KID）**（Bińkowski et al. 2018）は、3 次多項式カーネル $k(u, v) = (u^\top v/d + 1)^3$ を用いて二乗最大平均値差異（maximum mean discrepancy、MMD）の不偏推定量（`d2l.mmd2_unbiased`、16.2 節）を計算するものであり、特徴量の 3 次までのモーメントを比較する（演習問題 5）。

$$\begin{aligned}\widehat{\mathrm{MMD}}^2 = &\frac{1}{n_p(n_p-1)}\sum_{i\neq i'} k(\phi(x_i), \phi(x_{i'})) \\ &+ \frac{1}{n_q(n_q-1)}\sum_{j\neq j'} k(\phi(x'_j), \phi(x'_{j'})) \\ &- \frac{2}{n_p n_q}\sum_{i,j} k(\phi(x_i), \phi(x'_j)).\end{aligned} \tag{16.36}$$

```
def fid_score(feat_p, feat_q):
    """Frechet distance between Gaussians fitted to two feature sets."""
    feat_p = np.asarray(feat_p, np.float64)
    feat_q = np.asarray(feat_q, np.float64)
    mu_p, mu_q = feat_p.mean(0), feat_q.mean(0)
    C_p = np.cov(feat_p, rowvar=False)
    C_q = np.cov(feat_q, rowvar=False)
    lam, U = np.linalg.eigh(C_p)
    sqrt_C_p = (U * np.sqrt(np.maximum(lam, 0))) @ U.T
    M = sqrt_C_p @ C_q @ sqrt_C_p
    cross = np.linalg.eigvalsh((M + M.T) / 2)   # symmetric up to rounding
    return (((mu_p - mu_q) ** 2).sum() + np.trace(C_p) + np.trace(C_q)
            - 2 * np.sqrt(np.maximum(cross, 0)).sum())


def kid_score(feat_p, feat_q):
    """Unbiased MMD^2 between two feature sets, cubic polynomial kernel."""
    d = feat_p.shape[1]
    kernel = lambda a, b: (a @ b.T / d + 1) ** 3
    return d2l.mmd2_unbiased(kernel(feat_p, feat_p), kernel(feat_q, feat_q),
                             kernel(feat_p, feat_q))
```

16.5.4.2 特徴ネットワーク

標準的なベンチマークでは、ImageNet で事前学習された Inception-v3 ネットワークの 2,048 次元の特徴量、あるいは自己教師ありの DINOv2 特徴量を用いて FID と KID を計算する（Stein et al. 2023; Oquab et al. 2024）。このノートブックを自己完結させるため、ここでは軽量な 3 ブロックの CIFAR-10 分類器 `d2l.FeatureCNN`（オンライン付録「ユーティリティ関数とクラス」）を 32×32 の解像度で 15 エポック訓練し、その出力直前の 128 次元層を ϕ として用いる（64×64 のスプライトは 32×32 へと 2×2 の平均プーリングでダウンサンプリングする）。

```
def batched(f, X, batch_size=1000):
    """Apply f to X in batches, without gradients, and concatenate."""
    with torch.no_grad():
        return torch.cat([f(X_k) for X_k in X.split(batch_size)])
```

メモリを圧迫せずに評価できるよう batched を用い、CIFAR-10 を読み込んで、ランダムな 4 ピクセルシフトを伴いながら d2l.FeatureCNN を 32×32 の解像度で訓練し、テストセットにおける分類正解率を確認する。

```
cifar_X, cifar_y, test_X, test_y = d2l.load_cifar10_arrays()
torch.manual_seed(0)
cnn = d2l.FeatureCNN(resolution=32).to(device)
d2l.train_feature_cnn(cnn, cifar_X, cifar_y, shift=4)
pred = batched(cnn, test_X).argmax(1)
print(f'CIFAR-10 test accuracy: {float((pred == test_y).float().mean()):.3f}')
```

```
CIFAR-10 test accuracy: 0.778
```

16.5.4.3 適合率と再現率

FID や KID のような単一の距離指標は、p の台の外側に非現実的な画像を生成する**低い忠実度**（poor fidelity）と、モード欠落（mode dropping）によって p の一部を覆い損ねる**低い被覆**（poor coverage）という 2 つの異なる失敗モードを混同してしまう。Kynkäänniemi et al. (2019) は、特徴量空間における K 最近傍球を用いてこれらを分離した。各実特徴量 $\phi(x_i)$ の周りで、$B(\phi(x_i), r_i)$ をその K 番目に近い実特徴量まで届くユークリッド球とし、生成特徴量の間でも $B(\phi(x'_j), r'_j)$ を同様に定義する（ここでは $K = 3$ を用いる）。すると、次の式が得られる。

$$\text{precision} = \frac{1}{n_q} \sum_{j=1}^{n_q} \mathbb{1}\Big[\phi(x'_j) \in \bigcup\nolimits_i B(\phi(x_i), r_i)\Big],$$

$$\text{recall} = \frac{1}{n_p} \sum_{i=1}^{n_p} \mathbb{1}\Big[\phi(x_i) \in \bigcup\nolimits_j B(\phi(x'_j), r'_j)\Big].$$

適合率は推定された実特徴量の台に含まれる生成サンプルの割合であり、**再現率**は推定された生成特徴量の台によって覆われる実サンプルの割合である。生成器が単一の出力へと崩壊すると、その K-NN 半径 r'_j は 0 に縮小し、再現率は 0 になる。

```
def precision_recall(feat_real, feat_gen, k=3, chunk=1000):
    """k-NN precision and recall of two feature sets: the fraction of each
    set inside the union of the other set's balls, each ball reaching to its
    center's k-th nearest neighbor; return (precision, recall)."""
    P = np.asarray(feat_real, np.float64)
    Q = np.asarray(feat_gen, np.float64)
    def dists(A, B):  # (len(A), len(B)) Euclidean distances
        sq = (A ** 2).sum(1)[:, None] - 2 * A @ B.T + (B ** 2).sum(1)
        return np.sqrt(np.maximum(sq, 0))
    def chunks(A):
        return (A[i:i + chunk] for i in range(0, len(A), chunk))
    def radii(A):  # sorted distances start with the point itself, at 0
        return np.concatenate([np.partition(dists(a, A), k, axis=1)[:, k]
                               for a in chunks(A)])
    def covered(A, B, r_B):  # is each row of A inside some ball of B?
        return np.concatenate([(dists(a, B) <= r_B).any(1)
                               for a in chunks(A)])
```

```
    return (float(covered(Q, P, radii(P)).mean()),
            float(covered(P, Q, radii(Q)).mean()))
```

16.5.4.4 各実行のスコアリング

各生成器の移動平均モデルから抽出した $n = 4{,}059$ 個のサンプルを、4,059 枚のホールドアウトスプライトすべてに対してスコアリングする。各生成器は潜在コードの 2 回の抽出それぞれで評価するため、2 行目はサンプルのみが変わったときにスコアがどれだけ変動するかを表す。下限の目安（フロア）として、互いに素な 4,059 枚ずつの訓練画像の 2 セット同士もスコアリングする。両者の間の距離はすべて推定量のノイズであり、FD においてはさらに有限サンプルの統計的バイアスが含まれる。特徴量ネットワークは独自のものであるため、ここではスコアを FID や KID ではなく FD および MMD^2 と呼ぶ。これらは 12 倍のサンプル数を用いる既存の公表値とは直接比較できない。また、このバックボーンは出力に tanh を持たないため、スコアリング前に出力をクランプし、最後の 2 列には生のピクセル値のうち $+1$ を超える割合と -1 を下回る割合を示す。

```
features = lambda imgs: batched(  # 2 x 2 averages: 64 x 64 -> 32 x 32
    cnn.features, F.avg_pool2d(imgs, 2)).cpu().numpy()
n = len(holdout_imgs)
zs = []
for seed in (7, 8):  # a second latent draw gives the sampling range
    torch.manual_seed(seed)
    zs.append(torch.randn(n, 100, device=device))
z_score = zs[0]
feat_gen, out_of_range = {}, {}  # out_of_range: fractions above +1, below -1
for name, G in gens.items():
    for row, z in zip((name, '  second draw'), zs):
        x = batched(G, z)
        out_of_range[(name, row)] = (float((x > 1).float().mean()),
                                     float((x < -1).float().mean()))
        feat_gen[(name, row)] = features(x.clamp(-1, 1))
feat_real = features(holdout_imgs)
feat_a, feat_b = features(train_imgs[:n]), features(train_imgs[n:2 * n])
white = float((holdout_imgs == 1).float().mean())
```

各特徴量セットをホールドアウトスプライトの特徴量セットと比較する。最初の行は下限の目安（フロア）である。

```
print(f'{"run":17s}{"FD":>7s}{"MMD^2":>8s}{"precision":>11s}{"recall":>8s}'
      f'{"> +1":>7s}{"< -1":>7s}')
scored = {(None, 'real vs. real'): (feat_a, feat_b),
          **{key: (feat_real, feat) for key, feat in feat_gen.items()}}
for key, (ref, feat) in scored.items():
    precision, recall = precision_recall(ref, feat)
    oor = ''.join(f'{v:7.3f}' for v in out_of_range.get(key, ()))
    print(f'{key[1]:17s}{fid_score(ref, feat):7.2f}{kid_score(ref, feat):8.3f}'
          f'{precision:11.2f}{recall:8.2f}{oor}')
print(f'real pixel values equal to +1: {white:.0%}')
```

```
run                 FD   MMD^2  precision  recall   > +1   < -1
real vs. real     0.19   0.003       0.95    0.95
DCGAN             2.05   0.115       0.34    0.48  0.000  0.000
```

```
  second draw       2.01   0.108       0.36    0.47  0.000  0.000
NS                 64.44   7.626       0.07    0.00  0.009  0.000
  second draw      64.32   7.628       0.07    0.00  0.010  0.000
NS + R1 + R2        5.32   1.092       0.33    0.42  0.150  0.001
  second draw       5.38   1.116       0.34    0.43  0.148  0.001
RpGAN + R1 + R2     3.51   0.652       0.36    0.44  0.143  0.001
  second draw       3.69   0.710       0.36    0.44  0.139  0.001
real pixel values equal to +1: 45%
```

この表から、いくつかの重要な教訓が読み取れる。

1. **実画像対実画像のベースラインフロア：**互いに素な 4,059 枚ずつの実スプライトの 2 セットでは、FD ≈ 0.2（有限サンプルの共分散推定による統計的バイアスのため）、$\widehat{\mathrm{MMD}}^2 \approx 0.000$、適合率・再現率は 0.95 となる。潜在コードの 2 回目の抽出で評価しても、FD の変化は ≤ 0.2 に収まる。
2. **崩壊対ペナルティ付き訓練：**崩壊したペナルティなしの NS 実験群では、FD $\approx$ 64–72、recall $= 0.00$、precision ≤ 0.07 となる。ここに $R_1 + R_2$ を加えると、FD は 1 桁減少し（$\sim$ 3.5–5.5）、適合率は $\sim$ 0.35–0.4、再現率は ~ 0.4 へと向上する。
3. **特徴空間への感度（DCGAN 対ペナルティ付きバックボーン）：**ペナルティ付きバックボーンのほうが 64×64 では輪郭が鮮明で白背景もきれいであるにもかかわらず、2015 年の DCGAN のほうが低い FD（~ 2）および $\widehat{\mathrm{MMD}}^2$ をこの 32×32 CIFAR-10 分類器の特徴量において達成する。これは 2×2 のダウンサンプリングによって DCGAN の高周波の斑模様が平滑化されるためである。特徴空間の評価指標には、基盤となる特徴ネットワークの不変性と解像度が常に反映される。

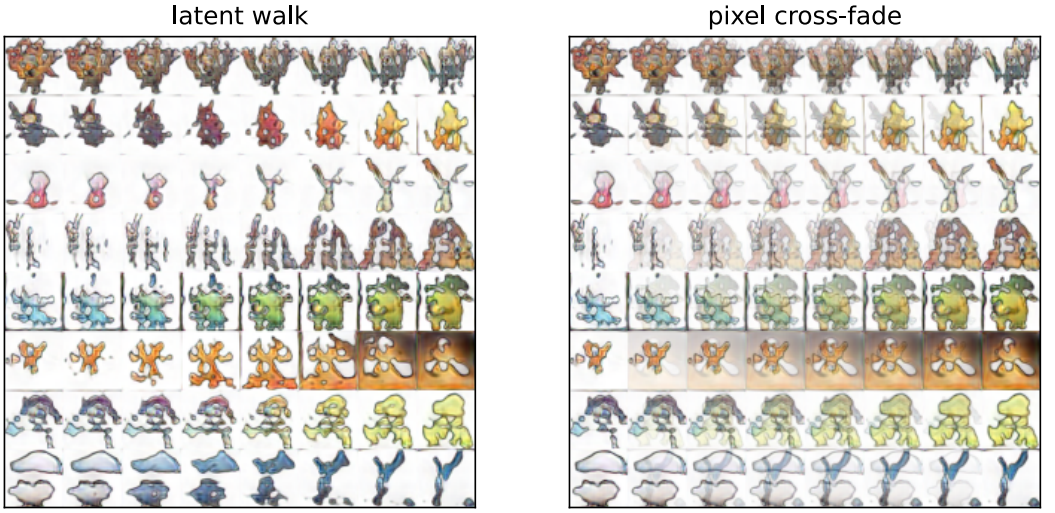
16.5.5 潜在補間とクリティックの過学習

16.5.5.1 潜在空間での補間

生成器が離散的なテンプレートを暗記しているのではなく、潜在コードから画像への滑らかな写像を学習していることを確かめるため、64 組の潜在ベクトルのペアの間を $z_t = (1-t)z_0 + tz_1 (t \in [0,1])$ と線形補間し、生成画像 $G(z_t)$ をピクセルごとのクロスフェード $(1-t)G(z_0) + tG(z_1)$ と比較する。

```
G = runs['RpGAN + R1 + R2'][0]
torch.manual_seed(0)
z0, z1 = torch.randn(2, 64, 1, 100, device=device)  # 64 pairs of codes
t = torch.linspace(0, 1, 8, device=device).view(1, 8, 1)
path = batched(G, ((1 - t) * z0 + t * z1).reshape(-1, 100)).clamp(-1, 1)
path = path.reshape(64, 8, 3, 64, 64)               # (pair, step, image)
t = t.view(1, 8, 1, 1, 1)
fade = (1 - t) * path[:, :1] + t * path[:, -1:]     # pixel cross-fades
inner = lambda x: features(x[:, 1:-1].flatten(0, 1))
print(f'precision of the inner steps: latent walk '
      f'{precision_recall(feat_real, inner(path))[0]:.2f}, '
      f'cross-fade {precision_recall(feat_real, inner(fade))[0]:.2f}')
d2l.show_images([tile_images(x[:8].flatten(0, 1), 8) for x in (path, fade)],
                1, 2, titles=['latent walk', 'pixel cross-fade'], scale=4.5);
```

```
precision of the inner steps: latent walk 0.35, cross-fade 0.21
```



潜在空間における中間補間点は、生成器からのランダムな抽出と同等の特徴量適合率（~ 0.35–0.38）を維持しながら形状や色が滑らかに遷移するのに対し、ピクセルごとのクロスフェードでは残像を伴う二重露光のような画像が生じ、適合率は半分（~ 0.18–0.21）に低下する。

16.5.5.2 クリティックの過学習とリークしないデータ拡張

1.6M 個のパラメータと 36,538 枚の訓練スプライトのもとで、クリティックは 12,000 ステップの間に各訓練画像を約 20 回参照する。ペナルティ付きクリティックにおける訓練・ホールドアウト間のスコア差はごくわずかである（$+0.02 \sim +0.05$ であり、実画像対生成画像のマージン ~ 0.7 と比べて小さい。演習問題 2 を参照）。しかし、より小規模なデータセットやより長時間の訓練では、識別器による暗記が FID を悪化させることがある (Karras, Aittala, et al. 2020)。これに対する標準的な対策である Adaptive Discriminator Augmentation（ADA、Karras, Aittala, et al. (2020)）および DiffAugment (Zhao et al. 2020) は、勾配経路の内部で実画像と生成画像の双方に可逆かつ**リークしない**（non-leaking）データ拡張を適用する（これに対し、本節の水平反転のように実画像のみを拡張すると、生成器は拡張後の分布に一致するように訓練される。演習問題 6 を参照）。

16.5.6 まとめ

DCGAN は、転置畳み込みとストライド付き畳み込み、バッチ正規化、小さな標準偏差による重みの初期化（$\sigma = 0.02$）、および $\beta_1 = 0.5$ の Adam によって、古典的ロジスティック損失のもとでの畳み込み敵対的訓練を安定化させる。バッチ正規化とアーキテクチャ上の安定化手法を取り除くと、ペナルティなしのロジスティック損失は完全に崩壊するが、ゼロ中心勾配ペナルティ（$R_1 + R_2$）を加えればクリティックの勾配が 2 桁減衰し、最小構成の双線形リサンプリングバックボーンを安定して訓練できる。生成器の重みの指数移動平均はミニマックス振動をさらに減衰させるが、バッチ正規化を用いる場合には移動統計量の再較正が必要である。最後に、定量的な画像評価では、フレシェ距離（Equation 16.35）、Kernel Inception Distance（Equation 16.36）、および K-NN の適合率と再現率を用いて実画像と生成画像の特徴量の点群を比較し、常に実画像対実画像の有限サンプルフロアを基準として較正する。

16.5.7 演習問題

1. **勾配ペナルティ重みと画像解像度**。$\nabla_x D(x)$ は長さの逆数の単位を持つため、ペナルティ重み γ は画像解像度とピクセル値のスケーリングに依存する。
   1. U を 64×64 の画像 x を 128×128 へアップサンプリングする作用素（各ピクセルを 2×2 ブロックに複製）とし、A を各 2×2 ブロックの平均をとる作用素として $D' = D \circ A$ が $D'(Ux) = D(x)$ を満たすとする。$\|\nabla D'(Ux)\|^2$ を $\|\nabla D(x)\|^2$ を用いて表し、ペナルティ重み γ' として 128×128 のペナルティを 64×64 のペナルティと等しくするものを求めよ。
   2. γ は、ピクセル値を $[0, 1]$ へと正規化した場合（$[-1, 1]$ の代わりに）、どのように再スケーリングしなければならないか。
   3. (a) の結果を、StyleGAN2-ADA で用いられている経験則 $\gamma \propto r^2/B$（解像度 r、バッチサイズ B）と関連づけて説明せよ (Karras, Aittala, et al. 2020)。

2. [code] **スケールに依存しないクリティックの過学習統計量**。スコアスケールの異なる目的関数の間でクリティックの暗記を比較するため、Karras, Aittala, et al. (2020) は次の量を定義している。

$$r_v = \frac{\mathbb{E}[D_{\text{train}}] - \mathbb{E}[D_{\text{holdout}}]}{\mathbb{E}[D_{\text{train}}] - \mathbb{E}[D_{\text{generated}}]},$$
$$r_t = \mathbb{E}\big[\operatorname{sign}(D_{\text{train}})\big].$$

   1. r_v は正のアフィン変換 $D \mapsto aD + c$（$a > 0$）に対して不変であるのに対し、生のギャップ $\mathbb{E}[D_{\text{train}}] - \mathbb{E}[D_{\text{holdout}}]$ は a 倍にスケーリングされ、r_t は平行移動 c に依存することを示せ。
   2. シフト不変なペアリングクリティックについて、D のどのような変換のもとで $\mathbb{E}[\operatorname{sign}(D_{\text{train}} - D_{\text{holdout}})]$ が不変になるか。また、訓練スコアとホールドアウトスコアが同一の分布に従うとき、この量はどの値をとるか。
   3. `train_arm` を変更して `net_D` を返すようにし、3 つの実験群それぞれについて 1,024 枚の画像にわたる r_v と $\mathbb{E}[\operatorname{sign}(D_{\text{train}} - D_{\text{holdout}})]$ を計算せよ。

3. [code] **FID と KID における有限サンプルの統計的バイアス**。
   1. 互いに素な実画像対実画像の FD と $\widehat{\text{MMD}}^2$ を、n 枚ずつの訓練スプライトの 2 セット間で $n \in \{250, 500, 1000, 2000, 4059\}$ の各値についてランダムな 5 分割にわたって計算せよ。
   2. FD を $1/n$ に対して線形回帰で当てはめ、$1/n \to 0$ へと外挿せよ（Chong and Forsyth (2020)）。$n = 500$ および $n = 4{,}059$ における有限サンプルバイアスはどの程度の大きさか。
   3. ホールドアウトスプライトに対する `RpGAN + R1 + R2` 生成器についても $1/n$ による線形当てはめを繰り返せ。$1/n$ の傾きは生成器の特徴量共分散に依存するため、小さな n における有限サンプルバイアスが 2 つの生成器のランキングを逆転させ得る理由を説明せよ。

4. [code] **重みの初期化とモード崩壊**。●サンプル多様性を、`z_show` 上の生成画像におけるペアごとの平均特徴量距離を 64 枚のホールドアウトスプライトにおけるペアごとの平均特徴量距離で割ったものとして定義する。
   1. サンプル多様性と $\mathbb{E}_x[\|\nabla_x D(x)\|^2]$ を 5,000 ステップにわたり 500 ステップごとに、ペナルティなしの `NS` 実験群について、He のファンイン初期化のもとで、および $\mathcal{N}(0, \sigma^2)$（$\sigma \in \{0.01, 0.02, 0.05\}$）による初期化のもとでそれぞれ記録せよ。クリティックの勾配ノルムの急上昇は、多様性の低下に先行するか。

2. ファンイン初期化での実験を 3 つのシードにわたって繰り返し、崩壊が起きるステップ数がシードごとにどの程度変動するかを測定せよ。

5. **FID と KID が捉えるモーメント。●**

   1. 3 次多項式の KID カーネル $k(u,v) = (u^\top v/d + 1)^3$ を二項定理によって展開し、母集団の $\mathrm{MMD}^2(p,q)$ が

$$\sum_{j=1}^{3} \binom{3}{j} d^{-j} \left\| \mathbb{E}_p[u^{\otimes j}] - \mathbb{E}_q[u^{\otimes j}] \right\|_F^2, \tag{16.37}$$

      に等しく、したがって特徴量分布の最初の 3 つのモーメントを比較することを示せ。

   2. 1 次元（$d = 1$）において、母集団の FID と 3 次カーネルの MMD^2 を、$p = \mathcal{N}(0,1)$ と、(i) 平行移動した指数分布 $X - 1$（ただし $X \sim \mathrm{Exp}(1)$ であり、平均 0、分散 1、歪度 2）、および (ii) 対称な混合分布 $\frac{1}{2}\mathcal{N}(-a, 1-a^2) + \frac{1}{2}\mathcal{N}(a, 1-a^2)$（$0 < a < 1$）のそれぞれとの間で計算せよ。

6. **リークしないデータ拡張。●** F を水平反転作用素とし、$T_\rho r = (1-\rho)r + \rho\, F_\# r$ を r からのサンプルを確率 ρ で反転させて得られる分布とする。

   1. `sample_real` のように実画像のみを $\rho = \frac{1}{2}$ で反転させた場合、生成器は $T_{1/2}p$ に向かって訓練される。真のスプライトのうち割合 α が左向きで対応する右向き個体を持たないとき、最適な生成器からのサンプルのうちどれだけの割合が右向きになるか。
   2. クリティックの内部で実画像と生成画像の双方に T_ρ を適用するとき、$F^2 = I$ を用いて、$T_\rho q = T_\rho p \implies q = p$ が $\rho \neq \frac{1}{2}$ のときに常に成り立つこと（**リークしない**性質、Karras, Aittala, et al. (2020)）、一方で $\rho = \frac{1}{2}$ のときには p と同じ反転対称化密度を持つ任意の分布が p と識別不能になることを示せ。

16.6 条件付き生成

16.1 節と 16.5 節の生成器は**無条件**（unconditional）である。すなわち、周辺データ分布 $p(x)$ からランダムにサンプルを抽出するだけで、目的のクラスやテキストプロンプト、入力画像を指定する手段を持たない。最初の GAN 論文の発表から間もなく、Mirza and Osindero (2014) は条件付け変数（conditioning variable）c を生成器 $G(z,c)$ と識別器 $D(x,c)$ の両方に入力する**条件付き GAN**（conditional GAN、cGAN）を導入した。

本節では、次の 3 つの問いに取り組む。

1. 条件付き定式化はどのような目的関数を最適化し、その最適クリティックは何を推定するのか。
2. 条件付け変数 c を生成器と識別器のアーキテクチャにどのように組み込むべきか。
3. 条件付き生成器を、条件アライメント（condition alignment）、サンプル忠実度（sample fidelity）、クラス内多様性（within-class diversity）のそれぞれについて個別に評価するにはどうすればよいか。

これらの概念を具体的に示すため、CIFAR-10 の 10 クラス上でクラス条件付き生成器を訓練し、評価する。

16.6.1 条件付き目的関数

16.6.1.1 最適条件付きクリティック

データと条件のペア $(x, c) \sim p(x, c)$ が同時データ分布から抽出されるものとする。ここで、c は離散的なクラスラベル、テキスト埋め込み、あるいは画像である。条件の周辺分布 $p(c)$ を一致させるため、実データから $c \sim p(c)$ をサンプリングしたうえで $x' = G(z, c) \sim q(\cdot \mid c)$ を生成すると、生成側の同時分布 $q(x, c) = p(c)\, q(x \mid c)$ が定まる。条件付き対数損失の目的関数は、古典的な二値対数損失の定式化（16.1 節）をペア (x, c) に適用したものである。

$$
\begin{aligned}
V(D) = {} & \mathbb{E}_{(x,c)\sim p}\big[\log \sigma(D(x, c))\big] \\
& + \mathbb{E}_{c\sim p(c),\, x'\sim q(\cdot|c)}\big[\log \sigma(-D(x', c))\big].
\end{aligned}
\tag{16.38}
$$

$D(x, c)$ に関する各点ごとの最適化は、直積空間 $\mathcal{X} \times \mathcal{C}$ 上での無条件の場合の導出と同一であるため、最適条件付きクリティックは同時密度の対数比となり、共通の周辺分布 $p(c)$ が約分されて消える。

$$
D^{\star}(x, c) = \log \frac{p(x, c)}{q(x, c)} = \log \frac{p(c)\, p(x \mid c)}{p(c)\, q(x \mid c)} = \log \frac{p(x \mid c)}{q(x \mid c)}. \tag{16.39}
$$

したがって、各条件 c に対して最適クリティックは、条件付き分布 $p(\cdot \mid c)$ と $q(\cdot \mid c)$ の対数密度比を計算する。$D^{\star}(x, c)$ を Equation 16.38 に代入すると、期待条件付きイェンセン・シャノン・ダイバージェンス（Jensen–Shannon divergence）が得られる。

$$
\max_{D} V(D) = \mathbb{E}_{c\sim p(c)}\Big[\, 2\,\mathrm{JS}(p(\cdot \mid c),\, q(\cdot \mid c))\Big] - 2\log 2. \tag{16.40}
$$

この式が大域的最小値 $-2\log 2$ に到達するのは、$q(\cdot \mid c) = p(\cdot \mid c)$ がすべての条件 c（ただし $p(c) > 0$）に対して成り立つとき、かつそのときに限る。

16.6.1.2 条件付き分布の一致と周辺分布の一致

すべての条件付きスライスで $q(\cdot \mid c) = p(\cdot \mid c)$ を一致させることは、画像の周辺分布で $q(x) = p(x)$ を一致させることよりも真に強い条件である。たとえば、CIFAR-10 の生成器がリアルな画像を生成するものの、すべてのクラスラベルを巡回的にずらしてしまう場合（$q(\cdot \mid c) = p(\cdot \mid c + 1)$）を考えよう。10 個のクラスはすべて等しい事前確率 $p(c) = 1/10$ を持つため、画像の周辺分布 $q(x)$ は $p(x)$ と厳密に一致し、無条件ダイバージェンスは最小値 $\mathrm{JS}(p(x), q(x)) = 0$ を達成する。しかし条件付き目的関数 Equation 16.40 では、すべてのスライスが異なるクラスと比較されるため（$\mathrm{JS}(p(\cdot \mid c), p(\cdot \mid c + 1)) \approx \log 2$）、条件付きダイバージェンスはほぼ最大値となる（演習問題 1）。

逆に、各条件 c が事前頻度 $p(c)$ によって Equation 16.40 の中で重み付けされていることにも注意が必要である。確率 $p(c) = \epsilon$ の希少な条件が目的関数に寄与する量は高々 $2\epsilon \log 2$ であるため、不均衡なデータセットでは、生成器が希少クラスを無視するのを防ぐためにクラス均衡サンプリングが必要になることが多い。

16.6.2 条件付けのアーキテクチャ

16.6.2.1 3 つの条件付け機構

Figure 16.9 に、条件 c を生成器や識別器に入力する主な 3 つの方法を示す。

1. **連結**（concatenation）(Mirza and Osindero 2014) は、c をベクトル e_c に埋め込み、潜在ベクトル z と連結する（あるいは空間特徴マップにブロードキャストして連結する）。
2. **特徴変調**（feature modulation）は、中間特徴マップ h の各チャネルを、学習された c の関数によってスケーリングおよびシフトする（$a(c) \odot h + b(c)$）。先に h を正規化する場合、これは**条件付きバッチ正規化**（conditional batch normalization）(Vries et al. 2017; Brock et al. 2019)（あるいは AdaIN や FiLM）となり、生成器のあらゆる解像度で条件を注入する。
3. **射影識別器ヘッド**（projection discriminator head）(Miyato and Koyama 2018) は、クラス埋め込み e_c と識別器のプーリング済み特徴ベクトル $\phi(x)$ の内積をとり、無条件スコア $\psi_0(x)$ を加算する。

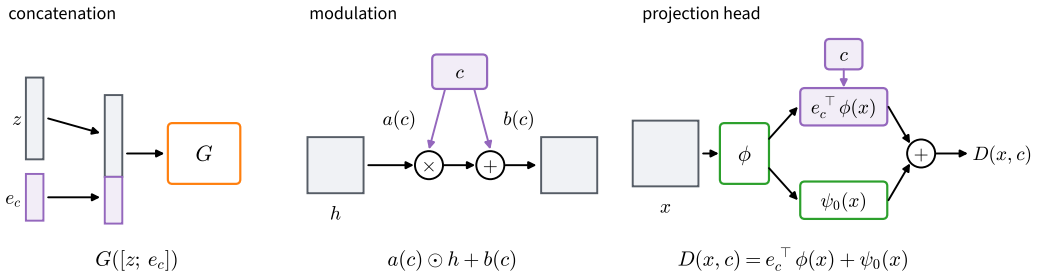


Figure 16.9: GAN における 3 つの条件付け機構。左：連結は、埋め込み e_c を入力または特徴マップに付加する。中央：特徴変調は、条件に依存するチャネルごとのスケーリングとシフト $a(c) \odot h + b(c)$ を適用する。右：射影識別器ヘッドは $e_c^\top \phi(x) + \psi_0(x)$ を計算する。

以下では、`Modulate` をクラスごとのスケーリングとシフトの埋め込みテーブルによって実装し、$a(c) = 1$ および $b(c) = 0$（恒等写像）で初期化したうえで、クラス 3 に対する勾配ステップが埋め込みテーブルの第 3 行のみを更新することを確認する。

```
class Modulate(nn.Module):
    """Scale and shift each channel of h by a(c) and b(c), one row per class."""
    def __init__(self, num_classes, channels):
        super().__init__()
        self.a = nn.Embedding(num_classes, channels)
        self.b = nn.Embedding(num_classes, channels)
        nn.init.ones_(self.a.weight)   # a(c) = 1, b(c) = 0: the identity
        nn.init.zeros_(self.b.weight)

    def forward(self, h, c):
        return self.a(c)[:, :, None, None] * h + self.b(c)[:, :, None, None]
```

この層を実行する前に、2 つのことが予測できる。初期化時点では、クラスによらず出力は入力と一致するはずである。また、クラス 3（以下で用いる CIFAR-10 の並び順では猫）のみを指定するバッチで勾配ステップを実行すると、各テーブルの第 3 行だけが変化し、他の行は一切変化しないはずである。埋め込みの参照は、読み出した行にしか勾配を伝播しないからである。以下の確認コードでは、トイ目標値 $2h + 1$ に向けてそのようなステップを 1 回実行する。

```
torch.manual_seed(0)
mod = Modulate(10, 8)
h = torch.randn(10, 8, 4, 4)  # one feature map per class
```

```
with torch.no_grad():
    print('at initialization: max |out - h| =',
          float((mod(h, torch.arange(10)) - h).abs().max()))
a0, b0 = mod.a.weight.detach().clone(), mod.b.weight.detach().clone()
c3 = torch.full((10,), 3)  # a batch that requests class 3 only
loss = ((mod(h, c3) - (2 * h + 1)) ** 2).mean()
loss.backward()
torch.optim.SGD(mod.parameters(), lr=0.1).step()
rows = lambda w, w0: (w != w0).any(1).nonzero().flatten().tolist()
print('after one step on class 3: rows changed in a', rows(mod.a.weight, a0),
      'and in b', rows(mod.b.weight, b0))
```

```
at initialization: max |out - h| = 0.0
after one step on class 3: rows changed in a [3] and in b [3]
```

16.6.2.2 ベイズの定理による射影ヘッドの導出

ベイズの定理 $p(x \mid c) = p(c \mid x)p(x)/p(c)$ を Equation 16.39 の分子と分母（共通の事前分布 $p(c)$ を持つ）に適用すると、最適条件付きクリティックは次の 2 つの項に分解される。

$$\log \frac{p(x \mid c)}{q(x \mid c)} \;=\; \log \frac{p(c \mid x)}{q(c \mid x)} + \log \frac{p(x)}{q(x)}. \tag{16.41}$$

第 1 項は**クラスアライメント**（class alignment、x における実データと生成データのクラス事後分布の対数比）を測り、第 2 項は**無条件のサンプルリアリズム**（unconditional sample realism）を測る。両方のクラス事後分布を、共有の特徴表現 $\phi(x)$ 上のソフトマックス分類器、すなわち $p(c \mid x) \propto \exp(u_c^\top \phi(x))$ および $q(c \mid x) \propto \exp(w_c^\top \phi(x))$ としてモデル化すると、それらの対数比は $(u_c - w_c)^\top \phi(x)$ に c に依存しない対数分配関数の差を加えたものになる。この対数分配関数の差と $\log(p(x)/q(x))$ を単一のスカラーヘッド $\psi_0(x)$ に吸収し、$e_c = u_c - w_c$ とおけば、Miyato and Koyama (2018) の**射影識別器**（projection discriminator）が得られる（演習問題 2。演習問題 6 の補助分類器 GAN（auxiliary classifier GAN）と比較せよ）。

$$D(x, c) \;=\; e_c^\top \phi(x) + \psi_0(x). \tag{16.42}$$

16.6.3 CIFAR-10 におけるクラス条件付き生成

ここで、ラベル付きの訓練画像 50,000 枚（解像度 32×32）からなる CIFAR-10（クラスあたり 5,000 枚、`d2l.load_cifar10_arrays` で読み込む）を用いてクラス条件付き生成器を訓練する。

```
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
device = d2l.try_gpu()
train_X, train_y, test_X, test_y = d2l.load_cifar10_arrays()
first = [int((train_y == c).nonzero()[0]) for c in range(10)]
d2l.show_images(train_X[first].cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=2, num_cols=5, titles=classes);
```

16.6.3.1 条件付き生成器と射影クリティック

`d2l.Generator` と `d2l.Discriminator`（16.5 節）を解像度 32×32 に拡張する。

- `CondGenerator` は、64 次元の単位分散クラス埋め込みを $4 \times 4 \times 64$ にブロードキャストし、生成器の `extra` 入力スロットに渡す（演習問題 4 では、この入力連結をブロックごとの `Modulate` 層と比較する）。
- `ProjectionCritic` は、4×4 の特徴マップ `self.phi(x)` を空間方向に足し合わせて $\phi(x)$ を求め、$e_c^\top \phi(x) + \psi_0(\phi(x))$（Equation 16.42）を計算する。このとき $e_c = 0$ で初期化することで、クリティックは無条件識別器として訓練を開始する。

```
class CondGenerator(d2l.Generator):
    """The backbone generator with a class embedding as extra 4x4 maps."""
    def __init__(self, latent_dim, resolution, base_ch, embed_dim=64,
                 num_classes=10):
        super().__init__(latent_dim, resolution, base_ch, extra_ch=embed_dim)
        self.embed = nn.Embedding(num_classes, embed_dim)  # unit variance

    def forward(self, z, y):
        ec = self.embed(y)[:, :, None, None].expand(-1, -1, 4, 4)
        return super().forward(z, ec)
```

`ProjectionCritic` は Equation 16.42 を実装したものである。基底クラスの `phi` は混合畳み込みの後に 4×4 の特徴マップを返し、それを空間位置にわたって足し合わせることで $\phi(x)$ が得られる。$\phi(x)$ 上の線形ヘッドが $\psi_0(x)$ を出力して、平坦化したマップを読み取る基底クラスのヘッドを置き換え、さらにテーブルがクラスごとに 1 つの埋め込み e_c を保持する。このテーブルはゼロから始まるため、クリティックは無条件クリティックとして出発し、データからラベル項を学習していく。

```
class ProjectionCritic(d2l.Discriminator):
    """The backbone critic with a projection head: e_c . phi(x) + psi_0(x)."""
    def __init__(self, resolution, base_ch, num_classes=10):
        super().__init__(resolution, base_ch)
        width = self.mix.out_channels
```

```
            self.head = nn.Linear(width, 1)  # psi_0 on the pooled features
            d2l.fan_in_init(self.head)
            self.embed = nn.Embedding(num_classes, width)
            nn.init.zeros_(self.embed.weight)  # start as an unconditional critic

        def forward(self, x, y):
            phi = self.phi(x).sum(dim=(2, 3))  # sum over the 4x4 positions
            return (self.embed(y) * phi).sum(dim=1) + self.head(phi).squeeze(-1)

    net_G, net_D = CondGenerator(100, 32, 256), ProjectionCritic(32, 64)
    print(net_G(torch.zeros(2, 100), torch.zeros(2, dtype=torch.long)).shape)
    print(net_D(torch.zeros(2, 3, 32, 32),
                torch.zeros(2, dtype=torch.long)).shape)
```

```
torch.Size([2, 3, 32, 32])
torch.Size([2])
```

16.6.3.2 同一クラスペアによる訓練

d2l.train_r3gan（16.5 節）を用い、15,000 ステップにわたってバッチサイズ 128、勾配ペナルティの重み $\gamma = 0.5$ で訓練する。各ミニバッチでは、生成画像が実データのミニバッチとまったく同じクラスラベル y で条件付けされるため、相対論的ペアリング損失 $\log \sigma(D(x, c) - D(x', c))$ は常に**同一**クラス c の実画像と生成画像を比較する。なお、比較は純粋にクラス内でのみ行われるため、$D(x, c)$ はクラスごとに独立な加法定数 ζ_c の違いを除いてシフト不変である（演習問題 3）。記録されるホールドアウトのペアワイズランキング正解率 $\widehat{P}(D(x, c) > D(x', c))$（512 組のテストペアで評価）も、このクラスごとのシフトに対して不変である。

```
    torch.manual_seed(0)  # the initialization
    net_G = CondGenerator(100, 32, 256).to(device)
    net_D = ProjectionCritic(32, 64).to(device)
    gamma = 0.5
    ema_G, history = d2l.train_r3gan(
        net_G, net_D, d2l.rpgan_loss_D, d2l.rpgan_loss_G, gamma, 15000, train_X,
        train_y, eval_images=test_X, eval_labels=test_y, eval_size=512,
        batch_size=128, seed=1)
```

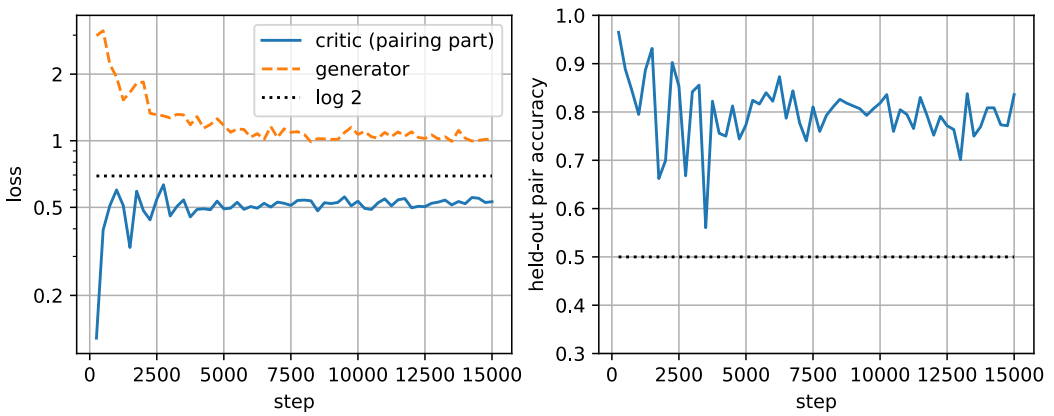
最後の 5,000 ステップにわたってログの平均をとり、ペア正解率については最初の 2,000 ステップでの平均も求める。

```
    loss_D, loss_G, pen, acc = (np.mean(v[-5000 // 250:]) for v in (
        history['loss_D'], history['loss_G'], history['R1'] + history['R2'],
        history['accuracy']))
    acc_early = history['accuracy'][history['step'] <= 2000].mean()
    print(f'last 5,000 steps: critic loss {loss_D:.2f} (+ penalty '
          f'{gamma / 2 * pen:.3f}), generator loss {loss_G:.2f}, '
          f'pair accuracy {acc:.2f}')
    print(f'first 2,000 steps: pair accuracy {acc_early:.2f}')
```

```
last 5,000 steps: critic loss 0.52 (+ penalty 0.062), generator loss 1.05, pair accuracy 0.79
first 2,000 steps: pair accuracy 0.83
```

loss_D、loss_G、およびホールドアウトの accuracy について全 15,000 ステップにわたる軌跡をプロットすると、射影クリティックと条件付き生成器がいかに速やかに安定したゲームへと落ち着くかがわかる。

```
steps = history['step']
fig, axes = d2l.plt.subplots(1, 2, figsize=(9, 3.2))
d2l.plot(steps, [history['loss_D'], history['loss_G'],
                 np.full_like(steps, np.log(2))], 'step', 'loss',
         legend=['critic (pairing part)', 'generator', 'log 2'],
         fmts=('-', '--', 'k:'), yscale='log', axes=axes[0])
d2l.plot(steps, [history['accuracy'], np.full_like(steps, 0.5)], 'step',
         'held-out pair accuracy', ylim=[0.3, 1], fmts=('-', 'k:'),
         axes=axes[1]);
```



初期の過渡期を過ぎると、クリティックのペアリング損失は 0.52 付近（$\log 2 \approx 0.69$ 未満）で安定し、ホールドアウトされた同一クラスペアのランキング正解率は 0.80 前後に落ち着く。

16.6.3.3 クラス条件付き生成サンプル

以下では、CIFAR-10 の 10 クラスそれぞれについて 9 枚のサンプルを ema_G から生成する。

```
torch.manual_seed(42)
z = torch.randn(90, 100, device=device)
y = torch.arange(10, device=device).repeat_interleave(9)  # nine per class
with torch.no_grad():
    imgs = ema_G(z, y)
panels = [d2l.tile_images(imgs[9 * c:9 * (c + 1)], 3) for c in range(10)]
d2l.show_images(panels, 2, 5, titles=classes, scale=2);
```

3×3 の各パネルには、指定されたクラスの視覚的特徴（たとえば、飛行機や船なら青い空や水面、カエルや鹿なら緑の草木、自動車やトラックなら道路の背景）が反映されると同時に、9 個の潜在サンプルごとにポーズや色彩が変化している。

16.6.4 条件付き生成器の評価

条件付き生成器は、次の 3 つの異なる軸で失敗しうる。

1. **誤ったクラス**（低い条件アライメント）
2. **非現実的な画像**（低いサンプル忠実度）
3. **クラス内多様性の欠如**（各クラス内でのモード欠落）

Table 16.4 にまとめるように、単一の指標でこれら 3 つの失敗モードすべてを捉えることはできない。そこで、クラスアライメントを測る**レフェリー分類器**（referee classifier）に、周辺特徴距離（FD と MMD^2）および**クラスごとの** K-NN 適合率（precision）と再現率（recall）を組み合わせる。

Table 16.4: 条件付き生成の失敗モードに対する評価指標の感度。

失敗モード	クラスアライメント	周辺 FD、MMD^2	クラスごとの適合率	クラスごとの再現率
誤ったクラス	検出可	検出不可	検出可	検出可
低い画像忠実度	部分的	検出可	検出可	部分的
低いクラス内多様性	検出不可（高く評価しうる）	検出可	検出不可（高く評価しうる）	検出可

16.6.4.1 レフェリー分類器とクラスアライメント

3 ブロックの CIFAR-10 分類器 `d2l.FeatureCNN`（16.5 節）を独立したレフェリーとして 15 エポック訓練し（テスト正解率は $\sim 78\%$ に達する）、そのクラス予測と最終層の 1 つ手前にある 128 次元の特徴量の層の両方を用いる。

```
torch.manual_seed(0)
cnn = d2l.FeatureCNN(resolution=32).to(device)
d2l.train_feature_cnn(cnn, train_X, train_y, shift=4)
```

```
pred = d2l.batched(cnn, test_X).argmax(1)
cls_acc = [float((pred[test_y == c] == c).float().mean())
           for c in range(10)]
print(f'CIFAR-10 test accuracy: {float((pred == test_y).float().mean()):.3f}')
```

```
CIFAR-10 test accuracy: 0.779
```

クラスあたり 500 枚の画像を `ema_G` から生成し、生成画像に対するレフェリーの分類正解率（**アライメント**）を、各クラスの実テスト画像に対する正解率と比較する。

```
def generate(seed, n=500):
    """n images of each class from the EMA generator, in class order."""
    torch.manual_seed(seed)
    z = torch.randn(10 * n, 100, device=device)
    y = torch.arange(10, device=device).repeat_interleave(n)
    with torch.no_grad():
        return torch.cat([ema_G(z_k, y_k) for z_k, y_k in
                          zip(z.split(1000), y.split(1000))])

raw = generate(7)
gen = raw.clamp(-1, 1)
yg = np.repeat(np.arange(10), 500)  # the requested classes
pred_gen = d2l.batched(cnn, gen).argmax(1).cpu().numpy()
align = [float((pred_gen[yg == c] == c).mean()) for c in range(10)]
```

`align` を実テスト画像に対するレフェリーの正解率 `cls_acc` と並べて 10 クラスそれぞれについて表にまとめると、`ema_G` が指定されたクラスラベルにどれほど忠実に従っているかがわかる。

```
print(f'{"class":12s}{"alignment":>11s}{"real":>7s}{"gap":>7s}')
for name, a, r in zip(classes + ['mean'], align + [np.mean(align)],
                      cls_acc + [np.mean(cls_acc)]):
    print(f'{name:12s}{a:11.2f}{r:7.2f}{a - r:+7.2f}')
```

```
class         alignment   real    gap
airplane           0.80   0.78  +0.01
automobile         0.82   0.91  -0.09
bird               0.66   0.67  -0.01
cat                0.54   0.57  -0.03
deer               0.66   0.71  -0.05
dog                0.69   0.76  -0.06
frog               0.78   0.85  -0.07
horse              0.76   0.80  -0.05
ship               0.86   0.87  -0.01
truck              0.79   0.87  -0.08
mean               0.74   0.78  -0.04
```

10 クラスにわたる平均クラスアライメントは 0.72–0.74 と、わずか 4–6 パーセントポイントだけ CIFAR-10 の実テスト画像に対するレフェリーの正解率（0.78）を下回るにすぎない。

16.6.4.2 忠実度とクラス内多様性

次に、クラスあたり 1,000 枚のテスト画像を 500 枚の参照セット（`f_real`）と 500 枚の実データ下限セット（real floor、`f_real2`）に分割し、周辺 FD と MMD^2（`d2l.fid_score`、`d2l.kid_score`）、およびクラスごとの K-NN 適合率と再現率（`d2l.precision_recall`、$K = 3$）を計算する。

```
features = lambda imgs: d2l.batched(cnn.features, imgs).cpu().numpy()
out_of_range = float((raw.abs() > 1).float().mean())
idx = test_y.argsort(stable=True).reshape(10, 1000)  # row c: class c
f_real = features(test_X[idx[:, :500].flatten()])    # the reference
f_real2 = features(test_X[idx[:, 500:].flatten()])   # the floor
f_gen = features(gen)
f_gen2 = features(generate(8).clamp(-1, 1))          # a second draw
idx, pred_real = idx.cpu().numpy(), pred.cpu().numpy()
```

それでは、周辺特徴距離（`d2l.fid_score`）と二乗 MMD（`d2l.kid_score`）を参照セット `f_real` との間で計算して、実データのテスト下限 `f_real2` と 2 回の生成サンプル群（`f_gen`、`f_gen2`）を比較し、あわせて範囲外となった生のピクセル値の割合 `out_of_range` も確認しよう。

```
draws = {'real (the floor)': f_real2, 'generated': f_gen,
         'generated, second draw': f_gen2}
print(f'{"against the reference":24s}{"FD (CIFAR-CNN)":>16s}'
      f'{"MMD^2 (CIFAR-CNN)":>19s}')
for name, f in draws.items():
    print(f'{name:24s}{d2l.fid_score(f, f_real):16.2f}'
          f'{d2l.kid_score(f, f_real):19.3f}')
print(f'generated values outside [-1, 1]: {out_of_range:.3f}')
```

```
against the reference     FD (CIFAR-CNN)  MMD^2 (CIFAR-CNN)
real (the floor)                    0.15             -0.001
generated                           1.60              0.105
generated, second draw              1.65              0.108
generated values outside [-1, 1]: 0.010
```

FD で見ると、生成セットは参照セットから下限セットの 10〜12 倍離れており、2 回目の抽出による FD の変動は 0.1 未満であるため、この距離は抽出の偶然ではなく生成器自体に起因するものである。MMD^2 では、下限セットがノイズの範囲内で 0 付近にあるのに対し、生成セットは約 0.11 となる。クランプ前の生のピクセル値のうち $[-1, 1]$ の外に出るのは 100 個に 1 個程度であり、クランプによって変化するピクセルはわずかで、問題が隠蔽されることもほとんどない。

これらの距離指標は、あらゆる違いを 1 つの数値に畳み込んでしまう。これに対し、クラスごとの適合率と再現率は 2 種類の違いを切り分ける。すなわち、そのクラスのどの実画像にも似ていない生成画像と、どの生成画像にも似ていない実画像である。各特徴ベクトルの周囲に、同じ集合内で 3 番目に近い近傍点まで届く球を描く（16.5.4.3 節）。**適合率**は、クラス c の生成画像のうちクラス c のいずれかの実データ球の内部に入るものの割合であり、**再現率**は、クラス c の実画像のうちいずれかの生成データ球の内部に入るものの割合である。次のセルでは、参照セットに対してクラスごとに両者を計算し、クラス間の平均と最悪クラスを表示したうえで、最後にクラス内特徴分散（各クラス内の特徴量の分散を特徴量とクラスにわたって平均した値）を出力する。

```
by_class = lambda f: np.asarray(f).reshape(10, -1, f.shape[-1])
```

```
    def pr_by_class(f):
        """Precision and recall of each class of f against the reference."""
        return np.array([d2l.precision_recall(r, g) for r, g in
                         zip(by_class(f_real), by_class(f))])

    print(f'{"per class, k = 3":24s}{"precision":>10s}{"worst class":>17s}'
          f'{"recall":>8s}{"worst class":>17s}')
    for name, f in draws.items():
        pr = pr_by_class(f)
        lo = pr.argmin(0)
        print(f'{name:24s}{pr[:, 0].mean():10.2f}{classes[lo[0]]:>12s}'
              f'{pr[lo[0], 0]:5.2f}{pr[:, 1].mean():8.2f}'
              f'{classes[lo[1]]:>12s}{pr[lo[1], 1]:5.2f}')
    # Variance within each class, averaged over features and classes
    cvar = lambda f: float(by_class(f).astype(np.float64).var(1).mean())
    print(f'within-class feature variance: generated {cvar(f_gen):.3f}, '
          f'real {cvar(f_real):.3f}, the floor {cvar(f_real2):.3f}')
```

```
per class, k = 3         precision      worst class  recall      worst class
real (the floor)              0.85       truck 0.77    0.84        bird 0.80
generated                     0.90       truck 0.85    0.59       horse 0.52
generated, second draw        0.90       truck 0.84    0.59       truck 0.53
within-class feature variance: generated 0.174, real 0.217, the floor 0.212
```

クラスごとの適合率は、生成画像（0.90）の方が実データ下限（0.85）よりもむしろ**高く**、生成特徴量が実データのクラスクラスタの十分内側に位置していることがわかる。一方で、クラスごとの再現率は大幅に低く（0.58 対 0.83、さらにクラス内特徴分散も約 20% 低い）、生成器が各クラスの多様性全体を網羅するのではなく、モード付近に集中していることを示している。

16.6.4.3 重心プロトタイプベースライン

高いアライメントと高い適合率がなぜ多様性の欠如を覆い隠してしまうのかを確認するため、実データ下限セットの中から特徴量がクラスの重心（centroid）に最も近い画像をクラスあたり 100 枚選んで**プロトタイプベースライン**（prototype baseline）を構築し、ランダムに選んだ実データ下限セットの画像 100 枚および生成画像 100 枚（いずれもクラスあたり）と比較する。

```
    floor = by_class(f_real2)  # test images 500-999 of each class
    dist = ((floor - floor.mean(1, keepdims=True)) ** 2).sum(-1)
    near = np.argsort(dist, axis=1)[:, :100]  # 100 distinct images per class
    rows = np.arange(10)[:, None]
    sets = {  # 100 images per class: features and the referee's predictions
        'real, random': (floor[:, :100], pred_real[idx[:, 500:600]]),
        'real, nearest centroid': (floor[rows, near],
                                   pred_real[idx[:, 500:][rows, near]]),
        'generated': (by_class(f_gen)[:, :100],
                      pred_gen.reshape(10, 500)[:, :100])}
    print(f'{"100 per class":24s}{"alignment":>10s}{"FD":>7s}{"MMD^2":>8s}'
          f'{"precision":>10s}{"recall":>8s}')
    for name, (f, p) in sets.items():
        f = f.reshape(1000, -1)
        prec, rec = pr_by_class(f).mean(0)
        print(f'{name:24s}{(p == rows).mean():10.2f}'
```

```
            f'{d2l.fid_score(f, f_real):7.2f}{d2l.kid_score(f, f_real):8.3f}'
            f'{prec:10.2f}{rec:8.2f}')
```

```
100 per class            alignment     FD   MMD^2 precision  recall
real, random                  0.78   0.43  -0.003      0.85    0.85
real, nearest centroid        0.91   3.03   0.123      0.97    0.37
generated                     0.74   1.59   0.077      0.89    0.67
```

Table 16.4 の予測どおり、最近傍重心のプロトタイプはほぼ完璧なアライメント（約 0.9 で、ランダムな実画像よりも高い）と適合率（約 0.97）を達成する一方、特徴量の共分散が人為的に収縮しているため、再現率の著しい低下（0.4 未満対約 0.85）と FD の悪化（約 3 対 0.43）を招く。BigGAN の**切断トリック**（truncation trick）は、潜在分布 z をゼロに向かって切断することで、意図的に再現率を犠牲にしてより高い適合率とアライメントを得る手法である（演習問題 5）。

16.6.5 まとめ

生成器 $G(z, c)$ とクリティック $D(x, c)$ の両方を $c \sim p(c)$ で条件付けると、スライスごとの最適クリティック $D^\star(x, c) = \log(p(x \mid c)/q(x \mid c))$ が得られ、期待条件付きダイバージェンス $\mathbb{E}_{c\sim p(c)}[2\,\mathrm{JS}(p(\cdot \mid c), q(\cdot \mid c))] - 2\log 2$ が最小化される。ベイズの定理を適用すると $D^\star(x, c)$ はクラスアライメント項 $\log(p(c \mid x)/q(c \mid x))$ と無条件リアリズム項 $\log(p(x)/q(x))$ に分解され、これが射影識別器ヘッド $D(x, c) = e_c^\top \phi(x) + \psi_0(x)$ の動機となる (Miyato and Koyama 2018)。条件付き生成器の評価では、条件アライメント（独立したレフェリー分類器による評価）、サンプル忠実度（クラスごとの適合率）、およびクラス内多様性（クラスごとの再現率と特徴距離）を切り分ける必要がある。再現率の指標がなければ、典型的なクラスプロトタイプのみを出力する生成器であっても、クラス内多様性を崩壊させたまま高い分類器アライメントと高い適合率を達成できてしまう。

16.6.6 演習問題

1. [code] **既知の密度を持つ 2 つのガウス分布スライス**。$c \in \{1, 2\}$ が等確率であり、$p(\cdot \mid 1) = \mathcal{N}(-2, 1)$、$p(\cdot \mid 2) = \mathcal{N}(2, 1)$ であるとし、$q(\cdot \mid 1) = \mathcal{N}(0, 1)$、$q(\cdot \mid 2) = \mathcal{N}(2, 1)$ と仮定する。
    1. 理論値 Equation 16.40 を 1 次元の数値求積法によって計算せよ。
    2. 小規模な MLP クリティックを $(x, \text{one_hot}(c))$ 上で訓練して Equation 16.38 を最大化し、収束後の目的関数値が (a) と一致することを確認したうえで、両方のスライスについて $D(x, c)$ を厳密な対数密度比 Equation 16.39 と重ねてプロットせよ。
    3. 次に、生成器が 2 つのクラスを入れ替えた場合、すなわち $q(\cdot \mid 1) = \mathcal{N}(2, 1)$ かつ $q(\cdot \mid 2) = \mathcal{N}(-2, 1)$ の場合を考える。条件付き目的関数値 Equation 16.40 と周辺分布に対する無条件目的関数値（c を無視した場合）の両方を計算し、訓練によって確認せよ。
2. **射影識別器ヘッドの表現力**。@eq-gan-cond-projection の導出では、$\phi(x)$ という共有の特徴マップを $p(c \mid x)$ と $q(c \mid x)$ の両方に仮定した。
    1. $p(c \mid x) \propto \exp(u_c^\top \phi_p(x))$ と $q(c \mid x) \propto \exp(w_c^\top \phi_q(x))$ が異なる特徴マップを用いる場合でも、得られる最適クリティックは連結した特徴ベクトル $(\phi_p(x), \phi_q(x))$ 上で依然として Equation 16.42 の射影形式をとることを示せ。
    2. K 個の離散クラスに対し、**任意の**関数 $D(x, c)$ が $e_c^\top \phi(x) + \psi_0(x)$ の形式で表せることを、$\dim(\phi(x)) \geq K$ が成り立つ場合について示せ。

3. [short-code] **分類器としての射影クリティックの利用**。訓練済みの射影クリティック `net_D` は、10 個のスコア $S_{ic} = D(x_i, c)$ を各画像 x_i に対して割り当てる。
   1. Equation 16.41 を用いて、$D^\star(x, c) - D^\star(x, c')$ を $p(c \mid x)$ と $q(c \mid x)$ によって表せ。なぜ $\arg\max_c D^\star(x, c)$ はベイズ分類器 $\arg\max_c p(c \mid x)$ と一致しないのか。
   2. 同一クラスペアリングではクラスごとに任意のシフト ζ_c が $D(x, c)$ に残るため、$\hat{\zeta}_c$ をクラス c の生成画像に対するクリティックスコアの平均として推定し、`test_X` 上での分類正解率を、補正なしの $\arg\max_c S_{ic}$ とシフト補正後の $\arg\max_c (S_{ic} - \hat{\zeta}_c)$ の間で比較せよ。
   3. シフト補正後のクリティックの正解率がランダム予測($1/10$)へと、$q(\cdot \mid c) \to p(\cdot \mid c)$ となるにつれて近づかなければならないのはなぜか。
4. [code] **特徴変調型生成器と入力連結の比較**。●クラス情報が入力の連結ではなく、すべての畳み込みブロックの後に置かれた `Modulate` 層を通じて入力される条件付き生成器を構築せよ。
   1. 初期化時点（$a(c) = 1, b(c) = 0$）では、変調型生成器の出力は c に依存しない。最初の生成器ステップにおいて、$a(c)$ と $b(c)$ に関する非ゼロの勾配が射影クリティックからどのように生じるかをたどれ。
   2. 変調型生成器と `CondGenerator` の両方をそれぞれ 2 つのシードで 4,000 ステップ訓練し、平均クラスアライメントとクラスごとの再現率を比較せよ。
5. [code] **切断トリックと適合率・再現率のトレードオフ**。切断トリック (Brock et al. 2019) は、各潜在座標を $[-\tau, \tau]$ に切断された標準正規分布からサンプリングする。
   1. 100 次元の潜在ベクトルと $\tau \in \{0.5, 1, 2\}$ に対し、切断サンプリングのもとでの二乗ノルムの期待値 $\mathbb{E}[\|z\|^2]$ を計算し、訓練時の $\mathbb{E}[\|z\|^2] = 100$ と比較せよ。
   2. `ema_G` を $\tau \in \{0.5, 1, 2, \infty\}$ の切断サンプリングのもとで評価し、クラスアライメント、クラスごとの適合率と再現率、および `f_real` に対する FD を報告せよ。また、τ を小さくすると再現率と引き換えにアライメントと適合率が向上し、生成器が重心プロトタイプの領域へ近づくことを確認せよ。

   Simon Prince『Understanding Deep Learning』問題 15.6 より改作。
6. **補助分類器 GAN（AC-GAN）のバイアス**。●AC-GAN (Odena et al. 2017) は、補助分類器 $C(c \mid x)$ を訓練し、交差エントロピー $-\mathbb{E}_{c\sim p(c), x\sim q(\cdot|c)}[\log C(c \mid x)]$ を生成器の損失に加える。
   1. この補助的な生成器損失が次の式に等しいことを示せ。

$$\mathbb{E}_{x\sim q}\Big[H(q(\cdot \mid x)) + D_{\mathrm{KL}}(q(\cdot \mid x) \,\|\, C(\cdot \mid x))\Big].$$

   2. $C(c \mid x) = p(c \mid x)$ であり、かつクラス間に重なりがある場合（$H(p(\cdot \mid x)) > 0$）、この項を最小化すると生成器がデータ分布の重なり合う領域を避けるようになり、射影識別器（Equation 16.41）と比べてクラス内多様性が損なわれる理由を説明せよ。

16.7 GAN を超える敵対的損失

住宅価格のような単一のスカラーを予測する場合、平均二乗誤差（mean squared error、L_2 損失）を最小化するのが標準的な手法である。しかし、超解像（低解像度のサムネイル c から高解像度画像 x への写像）やスケッチから写真への変換（pix2pix、Isola et al. (2017)）といった条件付き画像生成問題では、ピクセルごとの L_2 および L_1 回帰損失はぼやけた出力を生む。同じ低

解像度入力に対して多数の異なる高解像度テクスチャが対応するため、ピクセルごとの二乗誤差を最小化すると、ネットワークはそれらの条件付き平均 $\mathbb{E}[x \mid c]$ を出力せざるを得なくなる。これは、もっともらしい鮮明な画像のいずれにも属さないぼやけた像である。そこで回帰損失に識別器（discriminator）$D(x, c)$ を組み合わせ、出力が条件 c に対して現実的な画像かどうかをスコア付けすると、鮮明な高周波の細部が復元される。

一般的な画像生成では拡散モデルやフローマッチングモデル（第 17 章）が単体の GAN に取って代わったにもかかわらず、敵対的損失（adversarial loss）が今なお広く使われ続けている理由は、まさにこの仕組みにある。学習された識別器は現在でも、複数ステップの拡散モデルから蒸留された 1 ステップ生成器、潜在拡散モデルが動作する基盤となる画像オートエンコーダ（トークナイザ）、そしてスペクトログラムから音声波形を再構成するニューラル音声ボコーダの訓練を担っている。以下では、最小限の二峰性回帰問題を用いて条件付き平均による平均化効果を解析し、高ノイズのノイズ除去教師からの 1 ステップ蒸留が識別器なしでは条件付き平均へ崩壊する理由を示したうえで、現代の生成パイプラインにおいて敵対的損失がどこで使われているかを整理する。

16.7.1 条件付き平均と多峰性ターゲット

16.7.1.1 条件付き平均の分解

$g(c)$ を、x を条件 c から予測する決定論的予測器とし、$\mu(c) = \mathbb{E}[x \mid c]$ を条件付き平均（conditional mean）とする。$\mu(c)$ を $\mathbb{E}[\|x - g(c)\|^2]$ の内部で足して引くと、交差項 $2\,\mathbb{E}[\langle x - \mu(c), \mu(c) - g(c)\rangle] = 0$ が（c で条件付けた繰り返し期待値の法則により）消去され、次の直交分解が得られる。

$$\mathbb{E}\big[(x - g(c))^2\big] = \mathbb{E}\big[(x - \mu(c))^2\big] + \mathbb{E}\big[(\mu(c) - g(c))^2\big]. \tag{16.43}$$

第 1 項は削減不能な条件付き分散（conditional variance）$\mathrm{Var}(x \mid c)$ であり、第 2 項は $g(c)$ と $\mu(c)$ の二乗距離である。したがって、二乗誤差は条件付き平均 $g^\star(c) = \mu(c)$ によって一意に最小化される。$p(x \mid c)$ が単峰性で集中しているとき、$\mu(c)$ は代表的なターゲットとなる。一方、$p(x \mid c)$ が多峰性のとき、$\mu(c)$ はモード間の低密度領域に落ち込む。L_2 を L_1 損失 $\mathbb{E}[|x - g(c)|]$ に置き換えても、条件付き平均が座標ごとの条件付き中央値（conditional median）に置き換わるだけであり（演習問題 2）、対称な二峰性分布ではこれも同様に分布の台（support）の外側に位置する。

16.7.1.2 二峰性の 2 バンドベンチマーク

$c \sim \mathrm{Uniform}[-1, 1]$ とし、空白の回廊を挟んで平行に並ぶ 2 本のデータバンドをもつ 1 次元の条件付き分布を考えよう。

$$\begin{aligned} &x = c + b + 0.05\,\xi, \\ &b \in \{-1, +1\} \text{ with probability } \tfrac{1}{2} \text{ each}, \quad \xi \sim \mathcal{N}(0, 1). \end{aligned} \tag{16.44}$$

条件付き平均 $\mu(c) = c$ は 2 本のバンド $x = c \pm 1$ の間にある空白の回廊の真ん中を通り、$\mathbb{E}[(x - \mu(c))^2] = 1 + 0.05^2 = 1.0025$ を達成する。対照的に、上側のバンドに絞り込む予測器 $g(c) = c + 1$ の二乗誤差は $\tfrac{1}{2}(0) + \tfrac{1}{2}(2^2) + 0.05^2 = 2.0025$ と 2 倍の大きさになる。したがって二乗誤差は、正当なデータモードに絞り込むことに対してペナルティを課してしまう。

しかし、条件付き識別器 $D(x, c)$ は対数密度比（log density ratio）を推定する（16.6 節）。

$$D^\star(x, c) = \log \frac{p(x, c)}{q(x, c)}.$$

$p(x, c) \approx 0$ となる空白の回廊では $D^\star(x, c)$ が大きな負の値をとるため、非飽和型の敵対的損失 $-\log \sigma(D(\hat{x}, c))$ を最小化すると、予測器 $\hat{x} = g(c)$ は高密度なデータバンドのいずれかへと押し出される。

同一構造をもつ隠れ層 2 層の tanh 多層パーセプトロン（MLP、`g_mse` と `g_adv`）を同じ初期重みから 2 つ訓練してこれを検証しよう。`g_mse` は二乗誤差を最小化し、一方 `g_adv` は Leaky ReLU 識別器 $D(x, c)$ を相手に `d2l.gan_loss_D` と `d2l.gan_loss_G`（16.4 節）を用いて訓練する。

```
def make_bands(n):
    """Draw n pairs (c, x) from the two bands x = c + b + 0.05 xi."""
    c = torch.rand(n, 1) * 2 - 1
    b = torch.randint(0, 2, (n, 1)) * 2.0 - 1.0
    return c, c + b + 0.05 * torch.randn(n, 1)

torch.manual_seed(0)
g_mse = d2l.make_mlp([1, 32, 32, 1], nn.Tanh())
g_adv = d2l.make_mlp([1, 32, 32, 1], nn.Tanh())
g_adv.load_state_dict(g_mse.state_dict())  # both students start alike
D = d2l.make_mlp([2, 32, 32, 1], nn.LeakyReLU(0.2))
c_data, x_data = make_bands(200)  # a data sample for the plot
```

1 つ目の生徒モデルは Adam で二乗誤差を最小化し、その学習率はコサイン曲線に沿って 0.01 から 0 まで 2,000 ステップかけて減衰させる。どこに収束するかを知るのに実験は必要ない。Equation 16.43 により、各バンドから 1 単位離れた直線 $x = c$ に沿うはずである。これは 2 つ目の生徒モデルに対する対照群となる。

```
def train_mse(g, num_steps=2000, batch_size=128, lr=0.01):
    opt = torch.optim.Adam(g.parameters(), lr=lr)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, num_steps)
    for _ in range(num_steps):
        c, x = make_bands(batch_size)  # a fresh batch at every step
        opt.zero_grad()
        ((g(c) - x) ** 2).mean().backward()
        opt.step()
        sched.step()

train_mse(g_mse)
```

2 つ目の生徒モデルは、`d2l.gan_loss_D` と `d2l.gan_loss_G`（16.4 節）を損失として対数損失ゲームを行う。各ステップで、クリティックは実データペア (x, c) と予測ペア $(\hat{x}, c)$ に対してロジスティック回帰を 1 ステップ進め、続いて生徒モデルが更新後のクリティックに対して $-\log \sigma\big(D(\hat{x}, c)\big)$ を 1 ステップ進める。予測値にはデータの観測ノイズを加え、$\hat{x} = g(c) + 0.05\,\xi'$（毎回新たに引いた $\xi' \sim \mathcal{N}(0, 1)$ を使用）としてゲームに入力する。こうすることで、2 つの分布の違いは滑らかさではなく確率質量の配置場所のみになる。ゲームは回帰よりも進行が遅いため、この生徒モデルには 8,000 ステップを与える。一方、二乗誤差の生徒モデルは 2,000 ステップ後には最適解に達しており、ステップ数を増やしてもそれ以上は動かない。

```
def train_adversarial(g, D, num_steps=8000, batch_size=128):
    opt_G = torch.optim.Adam(g.parameters(), lr=1e-3)
    opt_D = torch.optim.Adam(D.parameters(), lr=2e-3)
    for _ in range(num_steps):
        c, x = make_bands(batch_size)
        x_hat = g(c) + 0.05 * torch.randn(batch_size, 1)  # the data's noise
        real, fake = torch.cat([c, x], 1), torch.cat([c, x_hat], 1)
        opt_D.zero_grad()
        d2l.gan_loss_D(D, real, fake.detach()).backward()
        opt_D.step()
        opt_G.zero_grad()
        d2l.gan_loss_G(D, real, fake).backward()
        opt_G.step()

train_adversarial(g_adv, D)
```

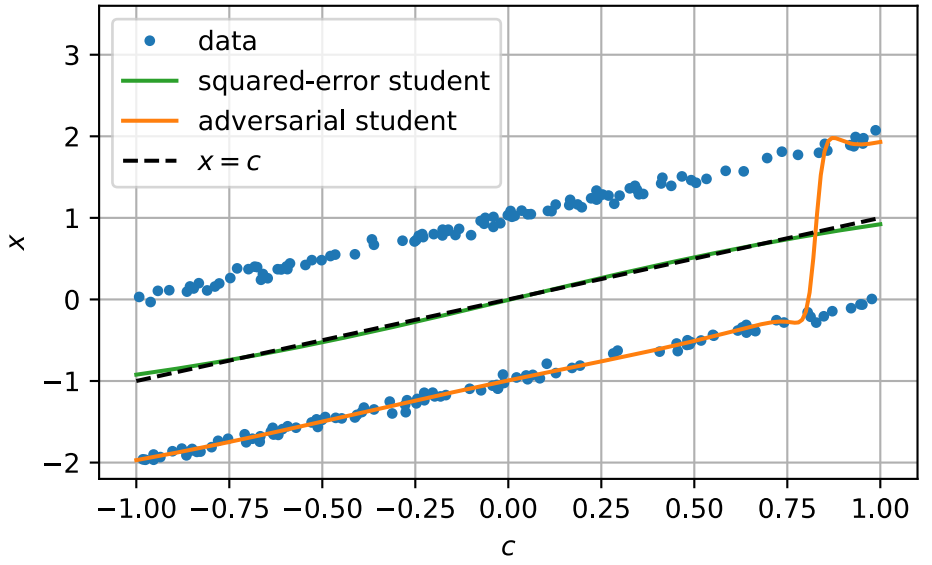
201 個の入力グリッド上で、各生徒モデルを 3 つの数値で要約する。曲線から近い方のバンドまでの平均距離、条件付き平均 $x = c$ までの平均距離、そして曲線が両方のバンドから 0.25 より大きく離れている入力の割合である。

```
cs = torch.linspace(-1, 1, 201).reshape(-1, 1)
with torch.no_grad():
    curves = {'squared error': g_mse(cs), 'adversarial': g_adv(cs)}
for name, x_hat in curves.items():
    to_band = torch.minimum((x_hat - cs - 1).abs(), (x_hat - cs + 1).abs())
    print(f'{name}: {to_band.mean():.2f} from the nearer band, '
          f'{(x_hat - cs).abs().mean():.2f} from x = c, '
          f'{(to_band > 0.25).float().mean():.0%} off the bands')
```

```
squared error: 0.98 from the nearer band, 0.02 from x = c, 100% off the bands
adversarial: 0.02 from the nearer band, 0.99 from x = c, 2% off the bands
```

curves に格納した学習済みの 2 本の曲線を、訓練サンプル (c_data, x_data) の上に破線の条件付き平均 $x = c$ と並べてプロットすると、回帰と敵対的訓練（adversarial training）の違いが一目でわかる。

```
d2l.plot([c_data, cs, cs, cs],
         [x_data, curves['squared error'], curves['adversarial'], cs],
         '$c$', '$x$', ['data', 'squared-error student',
                        'adversarial student', '$x = c$'],
         ylim=(-2.2, 3.6), fmts=('C0.', 'C2-', 'C1-', 'k--'),
         figsize=(5.5, 3.2))
```



予測どおり、二乗誤差モデル `g_mse` は空白の回廊に沿って条件付き平均 $\mu(c) = c$ をたどる（どちらのバンドからも約 1 単位離れており、予測の 100% がバンドから外れている）。一方、敵対的モデル `g_adv` は正当なデータバンドに絞り込む（最も近いバンドからの距離は平均で百分の数単位であり、回廊を横切る点は高々 2% にとどまる）。

ここで、根本的な知覚・歪みトレードオフ（perception–distortion trade-off）に注目しよう (Blau and Michaeli 2018)。理想的な確率的サンプラ $\hat{x} \sim p(x \mid c)$ でさえ、期待二乗誤差は $2\,\mathrm{Var}(x \mid c)$ となり、MSE で 2 倍（PSNR では 3 dB 低下）と、ぼやけた条件付き平均 $\mu(c)$ より悪化する（演習問題 1）。さらに、決定論的関数 $g(c)$ は入力 c ごとに 1 つの値しか出力できないため、**両方**のバンドを同時にカバーするには、確率的な潜在変数 z を c と併せて渡し、$\hat{x} = g(c, z)$ とする必要がある（演習問題 3）。

16.7.2 拡散蒸留における識別器

16.7.2.1 条件付き平均としてのノイズ除去器

拡散モデルとフローマッチングモデル（第 17 章）では、クリーンなデータ x_0 をノイズ付き入力 $x_\tau = x_0 + \tau\xi$（$\xi \sim \mathcal{N}(0, I)$）から予測するよう、ノイズ除去器（denoiser）ネットワークを二乗誤差で訓練する。Equation 16.43 により、最適なノイズ除去器は事後条件付き平均 $\mathbb{E}[x_0 \mid x_\tau]$ を計算する（17.1 節）。ノイズレベル τ が大きい領域では、多数のクリーンな画像が x_τ と整合するため、$\mathbb{E}[x_0 \mid x_\tau]$ はぼやけた平均になる。複数ステップのサンプラは、高ノイズから $\tau \to 0$ へと小さなステップを数多く刻むことで、この平均をそのまま返すのを避けている。

複数ステップの拡散教師モデルを高速な 1 ステップの生徒生成器へと**蒸留**（distillation）する際（17.8 節）、よく使われる手法の 1 つは、ノイズ $\tau\xi$ を生徒の出力 $\hat{x}$ に加えたうえで、$\hat{x}$ を凍結した教師のノイズ除去推定値 $\mathbb{E}[x_0 \mid \hat{x} + \tau\xi]$ へ回帰させることである。しかし高ノイズ τ では、教師のノイズ除去ターゲット自体が条件付き平均であるため、敵対的クリティックを加えない限り、1 ステップの生徒はぼやけた平均へと引き戻されてしまう。

16.7.2.2 2 バンドベンチマークにおける閉形式の蒸留

2 バンド問題 Equation 16.44 では、ノイズレベル τ における厳密なベイズ最適ノイズ除去教師は閉形式で表せる。ノイズ付き観測 $\tilde{x} = x + \tau\xi$（バンド $b \in \{-1, +1\}$ 上の点に対する観測）が与えられたとき、$\tilde{x} \mid c, b \sim \mathcal{N}(c + b, \tau^2 + 0.05^2)$ となる。ベイズの定理により、$b = +1$ 対 $b = -1$ の事後対数オッズ（log odds）は $2(\tilde{x} - c)/(\tau^2 + 0.05^2)$ となり、バンド中心の事後平均（posterior mean）は次式で与えられる。

$$\mathbb{E}[c + b \mid \tilde{x}, c] \;=\; c + \tanh\left(\frac{\tilde{x} - c}{\tau^2 + 0.05^2}\right). \tag{16.45}$$

ここで各生徒モデル $g(c, z)$ に 1 次元のガウス潜在変数 $z \sim \mathcal{N}(0, 1)$ を与えて両方のバンドをカバーできるモデル容量をもたせ、教師のノイズレベル $\tau = 2$ において 3 つの訓練目的関数を比較する。

1. **回帰のみ**：$\|\hat{x} - \text{stop_grad}(\text{teacher}(\hat{x} + \tau\xi, c, \tau))\|^2$ を最小化する。Equation 16.45 の $\tilde{x}$ に関する傾きは高々 $(\tau^2 + 0.05^2)^{-1} \approx \frac{1}{4} < 1$（$\tau = 2$ のとき）であるため、不動点方程式 $u = \mathbb{E}_\xi[\tanh((u + \sqrt{\tau^2 + 0.05^2}\,\xi)/(\tau^2 + 0.05^2))]$（オフセット $u = g(c, z) - c$ に関する式）は、縮小写像による唯一の不動点（fixed point）$u = 0$（空白の回廊）をもつ。
2. **クリティックのみ**：$g(c, z)$ を条件付き識別器 $D(x, c)$ を相手に `d2l.gan_loss_D` と `d2l.gan_loss_G` で訓練する。
3. **クリティック + 回帰**：敵対的損失と教師回帰損失を等しい重みで足し合わせる（敵対的拡散蒸留、すなわち Adversarial Diffusion Distillation、ADD、Sauer, Lorenz, et al. (2023) と同様）。

```
def teacher(x_tilde, c, tau):
    """Posterior mean of the band center c + b given x_tilde = x + tau xi."""
    return c + torch.tanh((x_tilde - c) / (tau ** 2 + 0.05 ** 2))

def train_student(g, D, lam, tau=2.0, num_steps=4000, batch_size=128):
    """Train g(c, z) on the critic's loss if D is given, plus lam times the
    squared error toward the teacher's estimate at noise level tau."""
    opt_G = torch.optim.Adam(g.parameters(), lr=1e-3)
    if D is not None:
        opt_D = torch.optim.Adam(D.parameters(), lr=2e-3)
    for _ in range(num_steps):
        c, x = make_bands(batch_size)
        z = torch.randn(batch_size, 1)  # the student's latent
        x_hat = g(torch.cat([c, z], 1)) + 0.05 * torch.randn(batch_size, 1)
        real, fake = torch.cat([c, x], 1), torch.cat([c, x_hat], 1)
        loss_G = 0
        if D is not None:
            opt_D.zero_grad()
            d2l.gan_loss_D(D, real, fake.detach()).backward()
            opt_D.step()
            loss_G = d2l.gan_loss_G(D, real, fake)
        with torch.no_grad():  # the teacher's estimate is a fixed target
            target = teacher(x_hat + tau * torch.randn(batch_size, 1), c, tau)
        loss_G = loss_G + lam * ((x_hat - target) ** 2).mean()
        opt_G.zero_grad()
        loss_G.backward()
```

```
        opt_G.step()
```

これで生徒モデルは、各 c における出力の分布を表すようになった。関数 band_shares は、最初の実験と同じ 201 個の入力および潜在変数の等間隔な 64 個の分位点（quantile）で生徒モデルを評価し、近い方のバンドまでの平均距離と、上側バンドおよび下側バンドから 0.25 以内に収まる出力の割合を返す。

```
def band_shares(g, cs):
    """Mean distance of g(c, z) to the nearer band, and its shares on the
    upper and the lower band, over the inputs cs and 64 quantiles z."""
    zs = torch.special.ndtri((torch.arange(64) + 0.5) / 64)
    c, z = (v.reshape(-1, 1) for v in torch.meshgrid(
        cs.flatten(), zs, indexing='ij'))
    with torch.no_grad():
        u = g(torch.cat([c, z], 1)) - c  # offset from the corridor
    to_band = torch.minimum((u - 1).abs(), (u + 1).abs())
    return (to_band.mean(), ((u - 1).abs() < 0.25).float().mean(),
            ((u + 1).abs() < 0.25).float().mean())
```

各生徒モデルは 3 つの乱数シードで訓練する。同じシード内では、3 つの生徒モデルは同一の重みから出発し、同一のデータを観測する。

```
students = {'regression': (False, 1.0), 'critic': (True, 0.0),
            'critic + regression': (True, 1.0)}
for name, (has_critic, lam) in students.items():
    for seed in range(3):
        torch.manual_seed(seed)
        g = d2l.make_mlp([2, 32, 32, 1], nn.Tanh())
        D = d2l.make_mlp([2, 32, 32, 1], nn.LeakyReLU(0.2))
        train_student(g, D if has_critic else None, lam)
        dist, upper, lower = band_shares(g, cs)
        print(f'{name}, seed {seed}: {dist:.2f} from the nearer band, '
              f'{upper:.0%} upper, {lower:.0%} lower, '
              f'{1 - upper - lower:.0%} off the bands')
```

```
regression, seed 0: 0.97 from the nearer band, 0% upper, 0% lower, 100% off the bands
regression, seed 1: 0.98 from the nearer band, 0% upper, 0% lower, 100% off the bands
regression, seed 2: 0.98 from the nearer band, 0% upper, 0% lower, 100% off the bands
critic, seed 0: 0.03 from the nearer band, 49% upper, 50% lower, 1% off the bands
critic, seed 1: 0.08 from the nearer band, 50% upper, 49% lower, 1% off the bands
critic, seed 2: 0.04 from the nearer band, 50% upper, 49% lower, 1% off the bands
critic + regression, seed 0: 0.05 from the nearer band, 48% upper, 50% lower, 2% off the bands
critic + regression, seed 1: 0.06 from the nearer band, 50% upper, 48% lower, 2% off the bands
critic + regression, seed 2: 0.04 from the nearer band, 48% upper, 51% lower, 1% off the bands
```

3 シードの結果から、不動点解析の妥当性が確認できる。

- **高ノイズ教師への回帰**では、出力の 100% が空白の回廊（どちらのバンドからも約 1 単位の距離）へ崩壊し、潜在変数 z は無視される。
- **クリティックのみ**と**クリティック + 回帰**はいずれも、出力の 98–99% を 2 本のデータバンド上に直接配置し（平均距離は百分の数単位で常に 0.1 未満）、確率質量を上側と下側のバンドに均等に（$\sim 49\%/49\%$）潜在変数 z を通じて振り分ける。バンドの幅が $\sigma = 0.05$

と狭いため、最適クリティックの復元傾き（$\delta/0.05^2 = 400\delta$）が高ノイズ教師による内側への緩やかな引力を容易に上回るのである。

16.7.2.3 大規模な蒸留、オートエンコーダ、ボコーダ

回帰損失や知覚損失によるアンカーと敵対的識別器のこの組み合わせは、現代の生成システムにおける主要な 3 つのクラスに共通して見られる。

1. **敵対的拡散蒸留（ADD と DMD2）：**ADD (Sauer, Lorenz, et al. 2023) は、ノイズ除去教師への回帰損失と凍結された DINOv2 特徴に基づく識別器を組み合わせて 1 ステップの画像生成器を訓練し、上の実験と同じ優劣関係を確認している（教師回帰のみではぼやけた画像になるが、識別器を加えると鮮明な 1 ステップサンプルが得られる）。分布マッチング蒸留（distribution matching distillation、DMD、Yin, Gharbi, Zhang, et al. (2024)）は、凍結された教師のスコア $\nabla_x \log p_\tau$ と偽データ分布の拡散モデルのスコア $\nabla_x \log q_\tau$ の差を用いて $D_{\mathrm{KL}}(q_\tau \,\|\, p_\tau)$ を降下する。この差はすでに登場している。すなわち、$\nabla_x \log p_\tau - \nabla_x \log q_\tau = \nabla_x \lambda_\tau$ は、ノイズを加えた両分布間における最適クリティック $\lambda_\tau = \log(p_\tau/q_\tau)$ の入力勾配である。したがって DMD の生成器更新は、重みなし（$w = 1$）の逆方向 KL 更新（演習問題 5）において、クリティックの傾きを識別器の代わりに 2 つのスコアネットワークから供給したものにほかならない。両方の分布にノイズが加えられているため、あらゆるノイズレベル $\tau > 0$ で両者の台は重なり合う。これは 16.1.2.3 節で述べたインスタンスノイズと同じ仕組みである。DMD2 (Yin, Gharbi, Park, et al. 2024) はさらに実画像に対する敵対的識別器を追加し、1 ステップの生徒が教師の品質を超えられるようにしている。
2. **潜在画像オートエンコーダ（トークナイザ）：**潜在拡散モデル（17.7 節）は、オートエンコーダ（VQGAN や Stable Diffusion VAE など、Esser et al. (2021); Rombach et al. (2022)）を用いて画像を低次元の潜在空間へと圧縮する。圧縮された潜在コードだけでは微細な局所テクスチャが一意に定まらないため、デコーダの訓練ではピクセルごとの L_1/L_2 損失および知覚損失（LPIPS）にパッチレベルの識別器（PatchGAN）を加え、くっきりとした高周波テクスチャを保証している。
3. **ニューラル音声ボコーダ：**テキスト音声合成パイプラインは通常、メルスペクトログラム（フレームレベルの周波数振幅を指定するが位相は捨てる）を予測し、HiFi-GAN (Kong et al. 2020) などのボコーダを用いてスペクトログラムを音声波形へと逆変換する。正弦波 $\sin(\omega n + \vartheta)$ の未知の位相 $\vartheta \sim \mathrm{Uniform}[-a, a]$ に関する期待値は、振幅が $\sin(a)/a$ 倍に縮小するため（$\vartheta \sim \mathrm{Uniform}[-\pi, \pi]$ では完全に消失する。演習問題 5）、波形の回帰は無音へと崩壊してしまう。そこで HiFi-GAN は、メルスペクトログラムの L_1 損失および識別器の活性化値に対する特徴マッチング損失を、複数周期・マルチスケールの波形識別器と組み合わせている。

16.7.3 本章の敵対的目的関数の比較

Table 16.5 に、本章で学んだ 6 つの敵対的定式化を、最適クリティックが何を推定するか、入力をどのようにスコア付けするか、そしてクリティックの関数クラスをどのように正則化するかという 3 つの軸に沿ってまとめる。

Table 16.5: 本章で解析した敵対的目的関数のまとめ。

目的関数	最適クリティックの推定対象	スコアリング規則	クリティックの制約・正則化	台が重ならない場合の挙動（最適クリティック下）
対数損失 / f-GAN（16.1 節、16.2 節）	$\log(p/q)$、または $f'(p/q)$	単一サンプル分類	制約なし	飽和する
ヒンジ（16.2 節）	$\operatorname{sign}\log(p/q)$	マージン分類	制約なし（またはスペクトルノルム）	飽和する（$2/\kappa$ まで線形、κ リプシッツなクリティックの場合）
MMD（16.2 節）	カーネル平滑化された $p-q$（証人関数）	平均の差	RKHS 単位球	カーネルバンド幅 s 以内で非ゼロ勾配
1 次ワッサースタイン（16.2 節）	カントロビッチポテンシャル	平均の差	1 リプシッツ球（例：WGAN-GP）	任意の距離で単位傾き
RpGAN（16.3 節）	$\log(p/q)+c$	ペアワイズなブラッドリー・テリーランキング	制約なし	飽和する
R3GAN：RpGAN + R_1+R_2（16.4 節）	$\log(p/q)$ をスケール $2\sqrt{\gamma}$ 未満でローパスフィルタ処理したもの	ペアワイズなブラッドリー・テリーランキング	p と q 上のゼロ中心入力勾配ペナルティ	ペナルティ付きネットワークによるギャップ間の滑らかな補間

反復型の拡散モデルやフローモデル（第 17 章）と比べると、敵対的生成器は 1 回の順伝播で推論でき、劣決定な逆問題に対しても鮮明な出力を生成できるという利点がある。その一方で、次の 4 つの本質的な課題を抱えている。(i) 明示的な尤度評価ができないこと、(ii) 調整された勾配ペナルティを必要とする 2 ネットワーク結合のミニマックスダイナミクス、(iii) モード欠落に対する内在的なペナルティが弱いこと（16.1.3.3 節）、(iv) 大規模化に伴う訓練不安定性の懸念である (Brock et al. 2019)。次章（第 17 章）では、ミニマックス訓練を複数ステップの反復的なノイズ除去に置き換える拡散モデルとフローマッチングモデルへと進む。

16.7.4 まとめ

二乗誤差 $\mathbb{E}[\|x-g(c)\|^2]$ を最小化すると、予測器は条件付き平均 $\mu(c)=\mathbb{E}[x \mid c]$（Equation 16.43）へと向かうため、ターゲット x が条件 c のもとで曖昧な場合には常に異なるモード間が平均化されてしまう。予測器に条件付き識別器 $D(x,c)$（$\log(p(x,c)/q(x,c))$ を推定するモデル）を組み合わせると、モード間の低密度な回廊にある予測にペナルティが課されて出力はデータ多様体上へと押し出され、MSE が最大 2 倍増加する（PSNR で 3 dB 低下する）ことと引き換えに知覚的な現実性が得られる (Blau and Michaeli 2018)。拡散モデルのノイズ除去器も同様に事後平均 $\mathbb{E}[x_0 \mid x_\tau]$ を計算するため、高ノイズのノイズ除去器へ直接回帰させて 1 ステップ生成器を蒸留すると、敵対的クリティックを加えない限り予測はぼやけた平均へと縮小してしまう。同じ理由から、敵対的識別器は潜在画像オートエンコーダ、ニューラル音声ボコーダ、および 1 ステップ蒸留拡散生成器において今なお不可欠な構成要素となっている。

16.7.5 演習問題

1. **理想的なサンプラにおける知覚・歪みトレードオフ。** $\hat{x}\sim p(x \mid c)$ を、x とは独立に（c を与えたもとで）サンプリングしたものとし、予測器が真の条件付き分布 $p(x \mid c)$ に厳密に一致するとする (Blau and Michaeli 2018)。

1. Equation 16.43 を用いて $\mathbb{E}\big[(x-\hat{x})^2\big] = 2\,\mathbb{E}\big[(x-\mu(c))^2\big]$ を証明せよ。厳密なサンプラは、PSNR（$10\log_{10}(x_{\max}^2/\text{MSE})$）において条件付き平均 $\mu(c)$ と比べて何デシベル失うか。
2. 2 バンド分布 Equation 16.44 において、厳密なサンプラの期待二乗誤差を、$\mu(c)=c$（1.0025）および単一バンド予測器 $g(c)=c+1$（2.0025）と比較せよ。この 3 つのうち、最適クリティック $D(x,c)$ によって真のデータと区別できないものはどれか。
3. $x\sim\mathcal{N}(0,1)$ かつ $c=x+\tau\xi$（ただし $\xi\sim\mathcal{N}(0,1)$ は独立）とする。決定論的な線形予測器 $\hat{x}=c/\sqrt{1+\tau^2}$ が周辺分布 $\mathcal{N}(0,1)$（x の分布）に一致することを示し、その MSE と条件付き平均 $\mathbb{E}[x\mid c]$ の MSE との比を $\tau\in\{0.1,1,10\}$ に対して求めよ。

2. [code] **条件付き平均と条件付き中央値（L_2 対 L_1）**。絶対誤差 $\mathbb{E}[|x-g(c)|]$ を最小化すると、条件付き中央値 $x_{\text{med}}(c)$（すなわち $P(x\le x_{\text{med}}(c)\mid c)=\frac{1}{2}$ を満たす値）が得られる。
   1. オフセット b（Equation 16.44）が値 $(-1,0,+1)$ を確率 $(\frac{1}{3},\frac{1}{3},\frac{1}{3})$ または $(\frac{1}{4},\frac{1}{4},\frac{1}{2})$ でとるとき、それぞれの場合について条件付き平均と条件付き中央値を求め、各々がデータバンド上に位置するかどうかを述べよ。
   2. 非対称な事前分布 $P(b=+1)=r$ をもつ 2 バンド問題において、条件付き平均を r の関数として計算し、$r=\frac{3}{4}$ のとき条件付き中央値が上側バンド上に位置することを示せ。さらに、L_1 リスクが回廊 $g(c)-c\in(-1,1)$ 全体でどのようになるかを $r=\frac{1}{2}$ の場合について特徴づけよ。
   3. `g_mse` のコピーを L_1 損失を用いて (a) と (b) の設定で訓練し、学習された曲線を解析的な予測と比較せよ。

3. [code] **二値の潜在変数の追加**。●16.7.1.2 節の生徒モデルに独立な一様二値潜在変数 $z\in\{-1,+1\}$ を追加して $\hat{x}=g(c,z)$ とする。ただし、クリティックは引き続き (x,c) のペアをスコア付けするものとする。
   1. 1 つの生徒モデルを二乗誤差で、もう 1 つを敵対的に訓練し、$z=-1$ と $z=+1$ がそれぞれどちらのバンドに写像されるかを確認せよ。なぜ二乗誤差の生徒モデルは z を無視するのか。
   2. 陰的最尤推定（IMLE、Li and Malik (2018)）は、クリティックを用いずに $\mathbb{E}_{(c,x)}[\min_{z\in\{-1,+1\}}(g(c,z)-x)^2]$ を最小化する。この損失で生徒モデルを訓練し、各データ点について z に関する最小値をとることで両方の潜在状態が異なるバンドをカバーするようになる理由を説明せよ。

4. **直接識別最適化（DDO）**。DDO (K. Zheng, Chen, Chen, et al. 2025) は、尤度モデル q_θ を凍結された参照コピー q_{ref} に対してファインチューニングする手法であり、クリティックを対数尤度比 $D_\theta(x)=\log(q_\theta(x)/q_{\text{ref}}(x))$ として直接パラメータ化し、ロジスティック識別器損失 $J(\theta)=-\mathbb{E}_{x\sim p}[\log\sigma(D_\theta(x))]-\mathbb{E}_{x\sim q_{\text{ref}}}[\log\sigma(-D_\theta(x))]$ を $\theta=\theta_{\text{ref}}$ から出発して最小化する。
   1. $q_\theta=p$ ならば、$D_\theta(x)$ は最適クリティック $D^\star(x)$（p と q_{ref} の間のクリティック）に等しいことを示せ。
   2. $\mathbb{E}_{x\sim q_\theta}[\nabla_\theta\log q_\theta(x)]=0$ を用いて、初期化時点（$\theta=\theta_{\text{ref}}$、すなわち $D_\theta(x)=0$）において、勾配 $\nabla_\theta J(\theta)$ が $\frac{1}{2}$ 倍の負の対数尤度勾配 $-\nabla_\theta\mathbb{E}_{x\sim p}[\log q_\theta(x)]$ に等しいことを示せ。

5. **音声ボコーダにおける位相の曖昧さと波形平均化**。長さ N の単一の音声フレームを、正弦波 $x_n=\sin(\omega n+\vartheta)$（$n=0,\dots,N-1$）としてモデル化する。ここで周波数 ω はスペクトログラムから既知であり、位相 $\vartheta\sim\text{Uniform}[-a,a]$（$0<a\le\pi$）は未観測とする。
   1. 条件付き平均波形 $\mathbb{E}_\vartheta[x_n]=\frac{\sin a}{a}\sin(\omega n)$ の振幅が $\frac{\sin a}{a}$ 倍に減衰する（$a=\pi$ のとき無音へ消失する）ことを示せ。
   2. $\omega=2\pi k/N$（$0<k<N/2$）のとき、離散フーリエ変換の振幅 $|\hat{x}_m|$ が ϑ に依存しないことを示せ。

3. 隣接する 2 つのフレームが独立な位相 $\vartheta_1 \neq \vartheta_2$ で生成された場合、フレームレベルの DFT 振幅損失は各フレーム単体では 0 になる。このときフレーム境界でどのようなアーティファクトが生じるか。また、時間領域の波形識別器はなぜそれを検出できるのか。

16.8 関連文献

本節では、本章の中核となる概念（密度比推定、ダイバージェンスによる定式化、相対論的目的関数、勾配ペナルティ、敵対的事後学習）を、より広範な文献およびオープンソース実装へと結びつける。

16.8.1 推奨される入門文献とコードベース

1. **局所収束と勾配ペナルティ：**@Mescheder.Geiger.Nowozin.2018 は 16.4 節で引用した局所収束定理を証明し、重みクリッピングや 1 中心ペナルティ（WGAN-GP）が破綻する均衡点近傍において、ゼロ中心勾配ペナルティ（R_1 と R_2）が交互勾配更新を安定化することを示した。著者らの GAN stability リポジトリには、Dirac-GAN および R_1 と R_2 の正則化項の実装が含まれている。
2. **現代化された GAN ベースライン（R3GAN）：**@Huang.Gokaslan.Kuleshov.ea.2024 は、相対論的ペアリング損失を $R_1 + R_2$ ペナルティおよび現代的な畳み込みブロックと組み合わせ、適切に正則化された目的関数を用いれば初期の GAN で使われていた場当たり的なアーキテクチャ上の安定化手法の多くが不要になることを示した（R3GAN 公式リポジトリ）。
3. **評価指標（適合率、再現率、FID、KID）：**@Kynkaanniemi.Karras.Laine.ea.2019 は、サンプルの忠実度（fidelity）とモード被覆（mode coverage）を切り分ける k-NN の適合率・再現率指標（16.5.4.3 節）を導入した。StyleGAN3 リポジトリ (Karras et al. 2021) は、フルスケールの FID、KID、適合率、再現率のリファレンス実装を提供している。
4. **敵対的拡散蒸留：**DMD2 (Yin, Gharbi, Park, et al. 2024) は、分布整合スコア蒸留（distribution-matching score distillation）と実画像に対して訓練された識別器を組み合わせることで、1 ステップの生徒生成器が複数ステップの拡散教師モデルを上回る性能を達成できるようにしている。
5. **対話的な可視化と歴史的サーベイ：**NIPS 2016 チュートリアル (Goodfellow 2016) と ganhacks チェックリストには初期の GAN 訓練におけるヒューリスティクスがまとめられており、GAN Lab (Kahng et al. 2018) では 2D の生成器と識別器のダイナミクスをブラウザ上で可視化できる。また、大規模ベンチマークにおいて Lucic et al. (2018) は、同等のハイパーパラメータ調整予算を与えれば、多くの敵対的目的関数（16.2 節）が同程度の FID を達成することを実証している。

16.8.2 目的関数とミニマックスダイナミクス

- **最小二乗 GAN（LSGAN）の符号化スキーム：**16.2 節では、0-1 最小二乗識別器が $\frac{1}{4}\chi^2(p \,\|\, m)$ を評価することを示した。Mao et al. (2017) は一般の正解ラベル b（実データ）、a（偽データ）、および c（生成器の目標ラベル）を解析し、最適クリティックのもとで生成器の目的関数が $\frac{1}{4}(a+b-2c)^2 + \frac{1}{4}(b-a)^2\chi^2(p \,\|\, m)$ に帰着することを示した。これは同じ χ^2 ダイバージェンスの正の定数倍である（任意の $b \neq a$ に対して成り立つ）。
- **f-GAN の出力活性化関数：**クリティックの出力 t が凸共役（convex conjugate）$f^*(t)$ の有限となる定義域に収まるようにするため（16.2 節）、Nowozin et al. (2016) は定義域に応じた出力活性化関数 $t(v)$ をネットワークの生の出力 v に適用している。具体的には、順方向 KL には恒等写像、逆方向 KL には $-e^{-v}$（$t < 0$）、イェンセン・シャノンには $\log 2 - \mathrm{softplus}(-v)$（$t < \log 2$）を用いる。

- **一般の相対論的ダイバージェンス：**@Jolicoeur-Martineau.2020 は、相対論的ペアリング目的関数（16.3 節）の較正された最適値が、任意の凹利得関数 ℓ（ただし $\ell(0) = 0$、$\ell'(0) \neq 0$、$\sup_{t>0} \ell(t) > 0$ を満たすもの）に対して正当なダイバージェンスを定義することを証明している。
- **双線形ミニマックス問題に対する最適化アルゴリズム：**双線形問題 $\min_\theta \max_\psi \theta\psi$ において、ステップサイズ h の同時勾配降下法はステップごとに原点からの距離を $\sqrt{1+h^2}$ 倍に増幅させる。**外挿勾配**法（extragradient method）(Korpelevich 1976) は先読みした試行点で勾配を評価し、距離を $\sqrt{1-h^2+h^4} < 1$ 倍（$0 < h < 1$ のとき）へと縮小させる。**楽観的勾配降下法**（optimistic gradient descent）(Daskalakis et al. 2018) は直前ステップの勾配を用いてこの先読みを近似し、負のモメンタム（negative momentum）(Gidel et al. 2019) は交互更新を安定化する。これらの手法がオプティマイザを変更するのに対し、大規模な画像 GAN では代わりに R_1/R_2 勾配ペナルティによって目的関数を変更する（16.4 節）。

16.8.3 固定された差異と学習される差異

クリティックネットワークを用いずに、固定された 2 サンプル間距離に対して生成器を直接訓練することはできるだろうか。Generative Moment Matching Networks (Li et al. 2015) は固定カーネルの MMD（16.2 節）を最小化し、陰的最尤推定（Implicit Maximum Likelihood Estimation、IMLE）(Li and Malik 2018) はモード欠落を防ぐために各実データ例を最近傍の生成サンプルに対応させ、Sliced Wasserstein 生成器 (Deshpande et al. 2018) はランダム射影にわたって 1D の最適輸送距離を平均する。しかし自然画像においては、ピクセル空間の固定カーネルやランダム射影では知覚的な差異を捉えることが難しく、これが MMD-GAN (C.-L. Li et al. 2017; Bińkowski et al. 2018) における敵対的カーネル学習の動機となった。

16.8.4 アーキテクチャ、特徴空間、データ効率

- **アーキテクチャの進化（2015〜2021 年）：**DCGAN (Radford et al. 2015) から R3GAN (Huang et al. 2024) に至るまで、訓練の安定性と高解像度生成に向けたアーキテクチャ上の進歩が重ねられた。代表的なものとして、漸進的成長（progressive growing）(Karras et al. 2018)、SAGAN や BigGAN におけるスペクトル正規化（spectral normalization）と自己アテンション (Miyato et al. 2018; Han Zhang et al. 2019; Brock et al. 2019)、そして StyleGAN の潜在マッピングネットワーク、スタイル変調・復調、およびエイリアスフリーなリサンプリング (Karras et al. 2019, 2021; Karras, Laine, et al. 2020) が挙げられる。
- **特徴空間と評価指標の落とし穴：**@Kynkaanniemi.Karras.Aittala.ea.2023 は、Inception-v3 の特徴量が ImageNet のクラス境界に強く影響されているのに対し、自己教師あり学習による DINOv2 の特徴量は人間の知覚的判断とより良く整合することを示した (Stein et al. 2023; Karras, Aittala, Kynkäänniemi, et al. 2024)。さらに、エイリアシングを伴う画像リサイズは FID を歪めることがあり (Parmar et al. 2022)、有限サンプルにおける FID の統計的バイアスはモデル依存の傾きをもって $O(1/n)$ でスケールする (Chong and Forsyth 2020)。
- **クリティックにおける事前学習済み特徴：**Projected GAN (Sauer et al. 2021) と StyleGAN-T (Sauer, Karras, et al. 2023) は、凍結した事前学習済みの視覚バックボーンの上に軽量な識別器ヘッドを適用することで、収束を加速している。この手法を用いる場合、評価には必ず独立した特徴抽出器を用いるべきである。
- **勾配ペナルティの償却：**R_1 と R_2 はクリティックを通じた 2 次の誤差逆伝播法を必要とするため、StyleGAN2 は**遅延正則化**（lazy regularization、Karras, Laine, et al. (2020)）として 16 ミニバッチごとに 1 回だけペナルティを評価する。一方、大規模な動画拡散モデルの事後学習では、微小なガウス入力摂動のもとでの有限差分（finite difference）によって R_1 を近似している (Lin et al. 2025)。

- **非リーク型データ拡張：**小規模なデータセットでは、クリティックは訓練画像を急速に暗記してしまう（16.5 節）。Adaptive Discriminator Augmentation (ADA) (Karras, Aittala, et al. 2020) および DiffAugment (Zhao et al. 2020) は、勾配経路の内部で実画像と生成画像の両方に可逆かつ**非リーク**（non-leaking）な変換を適用することで、目標分布を損なうことなくクリティックの過学習を防ぐ。

16.8.5 条件付け、画像変換、現代のハイブリッドシステム

- **特徴変調層：**チャネルごとのアフィン変調 $a(c) \odot h + b(c)$（16.6 節）は、Conditional Batch Normalization (Vries et al. 2017)、FiLM (Perez et al. 2018)、スタイル変換のための Adaptive Instance Normalization (AdaIN) (Huang and Belongie 2017)、およびセマンティック画像生成のための空間的に変化する SPADE (Park et al. 2019) として現れる。
- **ペアあり・ペアなしの画像から画像への変換：**Pix2pix (Isola et al. 2017) は、U-Net 生成器とパッチレベルの識別器（PatchGAN）を組み合わせることで L_1 再構成を鮮鋭化する。CycleGAN (Zhu et al. 2017) はサイクル一貫性（cycle consistency、$F(G(x)) \approx x$）を課すことでペアなしドメインへと変換を拡張するが、ドメイン間でエントロピーが異なるとき、サイクル一貫性を持つ写像は入力元の情報を高周波のステガノグラフィ的摂動の中に隠蔽してしまうことがある (Chu et al. 2017)。
- **トークナイザ、少数ステップ蒸留、ボコーダ：**視覚トークナイザ（VQGAN、ViTok）では、デコーダの敵対的損失によってピクセルごとの PSNR と引き換えに知覚的な再構成品質を向上させる (Esser et al. 2021; Hansen-Estruch et al. 2025, 2026)。拡散蒸留においては、LADD (Sauer et al. 2024)（FLUX.1-schnell で採用、Black Forest Labs (2024)）および SDXL-Lightning (Lin, Wang, et al. 2024) が潜在空間またはパッチ空間で直接識別器を適用する一方、MeanFlow (Geng et al. 2025) や連続時間一貫性モデル（continuous-time consistency model）(Lu and Song 2025) はクリティック不要の少数ステップ回帰目的関数を探求している。音声合成では、RAF (Lee and Choi 2026) が BigVGAN (Lee et al. 2023) を 16.3 節の相対論的ペアリング目的関数で改良している。
- **陰的クリティックとしての尤度モデル（DDO）：**Direct Discriminative Optimization (K. Zheng, Chen, Chen, et al. 2025) は、尤度モデルまたは拡散モデル q_θ を凍結した参照コピー q_{ref} に対してファインチューニングする際、クリティック自体を対数尤度比 $D_\theta(x) = \beta \log(q_\theta(x)/q_{\text{ref}}(x))$ としてパラメータ化することで、別個の識別器ネットワークを不要にしている（演習問題 4）。

Chapter 17

拡散モデル

第 16 章の敵対的な生成器（generator）は、ノイズをデータへ写像する生成器に直接的な回帰目標（regression target）がないため、識別器（discriminator）またはクリティックという 2 つ目のネットワークを必要とした。拡散モデル（diffusion model）とフローマッチングモデルは、各訓練例をガウスノイズで段階的に破壊することによって回帰目標を構築する。小さなガウスノイズのステップを逆転させるには、ノイズを含む観測が与えられたもとでのクリーンなデータの条件付き期待値（conditional expectation）を推定する必要がある。

この視点は、非平衡熱力学とノイズ除去拡散確率モデル（denoising diffusion probabilistic model、DDPM）(Sohl-Dickstein et al. 2015; Ho et al. 2020)、スコアベース生成モデリング（score-based generative modeling）(Song and Ermon 2019; Y. Song, Sohl-Dickstein, et al. 2021)、およびフローマッチングや確率的補間（stochastic interpolant）(Lipman et al. 2023; Liu et al. 2023; Albergo et al. 2023) という 3 つの研究系譜を統合するものである。これら 3 つの定式化のいずれにおいても、単一のニューラルネットワークがクリーンなデータ（または等価的に、付加されたノイズ、スコア関数 $\nabla_{\mathbf{x}} \log p_t(\mathbf{x})$、あるいは経路に沿った速度）をあらゆるノイズレベル $t \in [0, 1]$ で予測するように学習し、数値サンプラがその予測を $t = 1$ の純粋なノイズから $t = 0$ のデータへと逆向きに積分する。

敵対的なミニマックス目的関数を回帰に置き換えることで、GAN の 3 つの限界が解消される。すなわち、訓練に必要なのは二乗誤差で最適化される単一のネットワークのみであり、重み付き回帰損失は対数尤度（log-likelihood）の変分境界（さらに確率フロー ODE を通じた厳密な対数尤度）を与え、尤度目的関数は欠落したモードにペナルティを科す。本章全体を通じて、デフォルトの定式化では DDPM における分散保持（variance-preserving）経路上のノイズ予測器と並行して、線形経路 $\mathbf{x}_t = (1 - t)\mathbf{x}_0 + t\epsilon$ に沿った速度を回帰する (Esser et al. 2024)。

最初の 3 節では、真の密度、スコア関数、事後平均（posterior mean）、速度場（velocity field）が閉形式で得られる 3 つのガウス分布の 2 次元混合分布を用いて、数学的・アルゴリズム的な中核を完全に構築する。17.4 節では Fashion-MNIST 画像へと進み、時間条件付き U-Net、フレシェ距離（Fréchet distance）による評価、および分類器なしガイダンス（classifier-free guidance）を導入する。続く 4 節では、サンプラ（17.5 節）、尤度境界と設計空間（17.6 節）、潜在拡散（latent diffusion）と Transformer バックボーン（17.7 節）、およびフロー写像（flow map）による少数ステップ蒸留（17.8 節）を検討する。最後に 17.9 節では、ガウスノイズをトークンマスキングに置き換えることで、拡散を離散トークンへと拡張する。より長い数学的証明は第 38 章に掲載されており、外部のチュートリアル、講義、参照実装は 17.10 節にまとめられている。

17.1 ノイズ除去とスコア

二乗誤差で訓練されたネットワークは、入力が与えられたときの回帰目標の条件付き期待値（conditional expectation）を学習する（16.7 節）。したがって、クリーンなデータ点 $\mathbf{x}_0 \sim p$ に等方的なガウスノイズ $\sigma\epsilon$ を加えて $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$ を作るとき、最小二乗法の意味で最適なノイズ除去器（denoiser）は事後平均（posterior mean）$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}]$ となる。**Tweedie の公式** (Robbins 1956; Miyasawa 1961; Efron 2011) によれば、この事後平均はノイズを加えたデータの対数密度（log-density）の勾配 $\nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}})$、すなわち**スコア**関数（score function）によって直接定まる。さらに Vincent (2011) は、ガウス摂動カーネルのスコアへネットワークを回帰させることで、$\nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}})$ を元のデータ密度 p にアクセスすることなく近似できることを示した。

密度そのものではなくスコアを学習すれば、エネルギーベースモデルで必要となる計算困難な正規化定数（normalizing constant）を回避しつつ、ランジュバン動力学（Langevin dynamics）などの勾配ベース MCMC によってサンプルを生成するのに必要なベクトル場（vector field）が得られる。本節ではまず、あらゆるノイズレベルにおいて密度、スコア、事後平均が厳密に求まる 2 次元の混合ガウス分布（Gaussian mixture）を導入する。このベンチマークを用いて Tweedie の公式を検証し、単一のノイズレベル σ で訓練されたスコアモデルが局所的なモード構造と大域的なモード重みを同時に捉えられない理由を調べたうえで、最後に単一レベルのノイズ除去を Fashion-MNIST 画像に適用する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.1.1 通し例

学習された各ベクトル場やサンプラを真の値と照合できるよう、本章で扱うすべての量があらゆるノイズレベルで厳密に求まる分布を 1 つ用意したい。それには平面上の混合ガウス分布で十分である。ここでは、平均が $(-2.5, -1.5)$、$(2.5, -1.5)$、$(0, 2.5)$、混合重みが $\pi_k = 0.5, 0.3, 0.2$、座標ごとの共通の標準偏差が $s = 0.5$ である 3 つの成分をもつ分布を用いる。各モードはその幅に比べて互いに遠く離れているため、モード間の密度には深い谷があり、さらに不均等な重みによってサンプラが誤りやすい条件が揃っている。

密度 p の**スコア**とはその対数の勾配 $\nabla \log p(\mathbf{x})$ であり、密度が最も急激に増加する方向を指すベクトル場である。混合分布 $p(\mathbf{x}) = \sum_k \pi_k \, \mathcal{N}(\mathbf{x}; \mu_k, s^2\mathbf{I})$ に対して、スコアは次のように書ける。

$$\begin{aligned} \nabla \log p(\mathbf{x}) &= \sum_k r_k(\mathbf{x}) \, \frac{\mu_k - \mathbf{x}}{s^2}, \\ r_k(\mathbf{x}) &= \frac{\pi_k \, \mathcal{N}(\mathbf{x}; \mu_k, s^2\mathbf{I})}{p(\mathbf{x})}. \end{aligned} \tag{17.1}$$

各成分は点をそれぞれの平均へと引き寄せ、その引力は**負担率**（responsibility）$r_k(\mathbf{x})$、すなわち $\mathbf{x}$ が成分 k から抽出された事後確率によって重み付け平均される（Figure 17.1）。重み π_k はこの負担率を通じてのみ式に現れる。モードの近くでは重みの値にかかわらず 1 つの負担率がほぼ 1 になるため、重みがスコアを変化させるのはデータ点がほとんど存在しない谷の部分だけである。一見すると何気ない性質だが、これがスコアモデルに学習できることを決定づけることになる。

ノイズを加えても、混合ガウス分布は混合ガウス分布のままである。抽出されたサンプルに標準偏差 σ のガウスノイズを加えると各成分の分散に σ^2 が加算され、s^2 は $s^2 + \sigma^2$ に置き換わるため、あらゆるノイズレベルにおいて密度、スコア、事後平均が閉じた式で求まる。以下のクラスはこれら 3 つをすべて計算する。クリーンなデータ点に対するスケールを表す引数 `alpha` は、次節まで 1 に固定しておく。

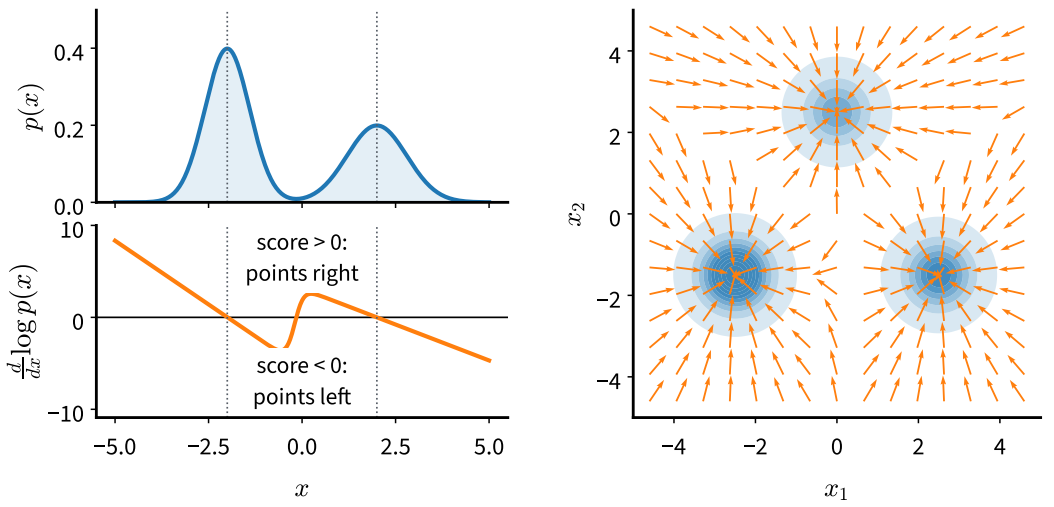


Figure 17.1: スコアは対数密度の勾配である。左：1 次元の二峰性密度（上）とそのスコア（下）。両方のモードと、その間の谷で 0 になる。右：通し例。陰影付きの等高線が密度を、モードへ向かう矢印がスコア場を表す（矢印の長さはクリップしてある）。

```
class GaussianMixture:
    """A mixture of isotropic Gaussians, exact at every noise level."""
    def __init__(self, means, weights, std, device=None):
        self.means = torch.tensor(means, dtype=torch.float32, device=device)
        self.weights = torch.tensor(weights, dtype=torch.float32,
                                    device=device)
        self.std = std

    @staticmethod
    def three_modes(device=None):
        """The chapter's running example, with mode weights 0.5/0.3/0.2."""
        return GaussianMixture([[-2.5, -1.5], [2.5, -1.5], [0.0, 2.5]],
                               [0.5, 0.3, 0.2], 0.5, device)

    def sample(self, n):
        """n draws from the mixture."""
        k = torch.multinomial(self.weights, n, replacement=True)
        return self.means[k] + self.std * torch.randn(
            n, self.means.shape[1], device=self.means.device)

    def _modes(self, x, alpha, sigma):
        """Per-mode log densities, offsets x - alpha mu_k, variances, alpha."""
        a, s = (torch.as_tensor(v, dtype=x.dtype, device=x.device)
                .reshape(-1, 1, 1) for v in (alpha, sigma))
        var = (a * self.std) ** 2 + s ** 2
        r = x[:, None] - a * self.means
        log_p = (torch.log(self.weights) - (r ** 2).sum(-1) / (2 * var[..., 0])
                 - x.shape[1] / 2 * torch.log(2 * torch.pi * var[..., 0]))
        return log_p, r, var, a
```

```
    def log_prob(self, x, alpha=1.0, sigma=0.0):
        """log p(x) for x = alpha x0 + sigma eps."""
        return torch.logsumexp(self._modes(x, alpha, sigma)[0], 1)

    def score(self, x, alpha=1.0, sigma=0.0):
        """The gradient of log p(x) for x = alpha x0 + sigma eps."""
        log_p, r, var, _ = self._modes(x, alpha, sigma)
        return -(torch.softmax(log_p, 1)[..., None] * r / var).sum(1)

    def posterior_mean(self, x, alpha, sigma):
        """E[x0 | x]; finite at alpha = 0 and at sigma = 0."""
        log_p, r, var, a = self._modes(x, alpha, sigma)
        return (torch.softmax(log_p, 1)[..., None]
                * (self.means + a * self.std ** 2 / var * r)).sum(1)
```

本章を通して、この混合分布に対するサンプラは同じ基準で評価する。すなわち、真の密度のもとでの生成点の平均負の対数尤度と、`summary` が返す 3 つの数値である。**モード重み**（mode weights）は、各点を最も近い平均に割り当てて集計した割合である。**外れ点**（strays）は、どの平均からも $3s = 1.5$ を超えて離れた点の割合を表す。**広がり**（spread）は、残りの点から最も近い平均までの平均二乗距離である。外れ点を除外するため、この値が $(3s)^2 = 2.25$ を超えることはなく、極端に広がりすぎたサンプラは主に外れ点の増加として現れる。

```
@d2l.add_to_class(GaussianMixture)
def nll(self, x):
    """Mean negative log-likelihood of clean points, in nats."""
    return -self.log_prob(x).mean()

@d2l.add_to_class(GaussianMixture)
def summary(self, x):
    """Mode weights by nearest mean, stray fraction, spread of the rest."""
    d2 = ((x[:, None] - self.means) ** 2).sum(-1)
    near, k = d2.min(1)
    inside = near < (3 * self.std) ** 2
    weights = torch.bincount(k, minlength=len(self.weights)) / len(x)
    return dict(weights=[round(w, 3) for w in weights.tolist()],
                stray=round(1 - inside.float().mean().item(), 4),
                spread=round(near[inside].mean().item(), 3))
```

これらの数値要約とあわせて 2D 点群を視覚的に比較するため、`scatter_panels` は共通の座標軸上に 1 つ以上のサンプル集合を並べてプロットする（任意で参照データセット `data` を薄い灰色で重ねて表示できる）。

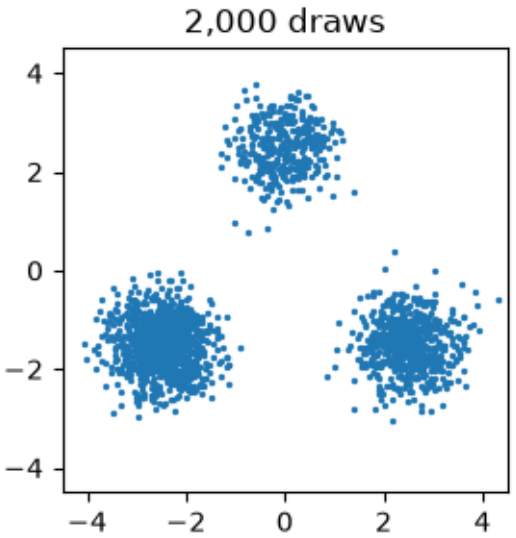
```
def scatter_panels(samples, titles=None, data=None, lim=4.5,
                   size=3.0):
    """Scatter 2-D point sets side by side on equal, shared axes."""
    _, axes = d2l.plt.subplots(1, len(samples), squeeze=False,
                               figsize=(size * len(samples), size))
    for i, (ax, pts) in enumerate(zip(axes[0], samples)):
        if data is not None:
            ax.scatter(*d2l.numpy(data).T, s=2, c='lightgray')
```

```
        ax.scatter(*d2l.numpy(pts).T, s=2)
        ax.set(xlim=(-lim, lim), ylim=(-lim, lim), aspect='equal')
        if titles:
            ax.set_title(titles[i])
    return axes[0]
```

完璧なサンプラはどのような値を出力するだろうか。サンプリング誤差を除けば、重みは 0.5、0.3、0.2 になる。2 次元では、ガウス分布からのサンプルと平均との二乗距離は平均 $2s^2 = 0.5$ の指数分布に従うため、$e^{-(3s)^2/(2s^2)} = e^{-4.5} \approx 1\%$ の割合の点が外れ点となり、残りの点の広がりは 0.5 をわずかに下回る。本節の 2 次元実験の訓練セットとなる 2,000 個のサンプルでこれを確認しよう。

```
torch.manual_seed(0)
mix = d2l.GaussianMixture.three_modes()
data = mix.sample(2000)
print(mix.summary(data), f'NLL {mix.nll(data):.3f}')
d2l.scatter_panels([data], ['2,000 draws']);
```

```
{'weights': [0.5, 0.317, 0.183], 'stray': 0.012, 'spread': 0.488} NLL 2.504
```



出力された値は予測どおりである。これが、この混合分布に対するすべてのサンプラが目指すべき基準値となる。負の対数尤度については一言注意が必要である。1 点あたり約 2.5 nat という値はこの混合分布のエントロピーであり、いかなる密度モデルも未知データに対して平均でこれより低い値を出すことはできず、正しいサンプラが生成した点も p のもとで同程度の値になる。サンプラにとってこの値は一致させるべき目標であって、下回るほどよいという限界値ではない。モードの中心付近に密集した点はこれより小さな値になり、谷にある点は大きな値になるからである。

17.1.2 最適ノイズ除去器

冒頭で述べた回帰問題には、クリーンなデータ点そのものと、そこへ向かうベクトルという 2 つの自然な回帰目標がある。それぞれの最小点（minimizer）を結びつける恒等式こそ、本節で最も重要な結果である。

17.1.2.1 ノイズと回帰目標

ノイズレベル $\sigma > 0$ を選び、クリーンな点 $\mathbf{x}_0 \sim p$ と標準ガウスノイズ $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ を抽出して、ノイズ付きの点 $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$ を作る。その密度 p_σ はデータ密度を幅 σ のガウス分布でぼかしたものであり、たとえ p がそうでなくても至るところ正かつ滑らかになる。ノイズ除去器 D は $\mathbb{E}\|D(\tilde{\mathbf{x}}) - \mathbf{x}_0\|^2$ を最小化するよう訓練する。本節の冒頭で確認したとおり（16.7 節）、二乗誤差は入力が与えられたときの目標の条件付き平均によって最小化されるため、最適なノイズ除去器は**事後平均** $\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}]$ となる。すなわち、$\tilde{\mathbf{x}}$ を生成し得たクリーンな点を、それぞれがそれを生成した尤もらしさで重み付けした平均である。

同じ回帰問題で、別の目標を予測させることもできる。クリーンな点が与えられたとき、ノイズ付きの点はその周囲のガウス分布に従い、この条件付き密度のスコアは次のようになる。

$$\nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}_0) = \frac{\mathbf{x}_0 - \tilde{\mathbf{x}}}{\sigma^2} = -\frac{\epsilon}{\sigma}, \tag{17.2}$$

これはノイズ付きの点から元のクリーンな点へと向かうベクトルである。**ノイズ除去スコアマッチング**（denoising score matching）では、ベクトル場 $\mathbf{s}_\theta$ をこの目標へ回帰させる (Vincent 2011)。

$$J_{\mathrm{DSM}}(\theta) = \tfrac{1}{2}\,\mathbb{E}_{\mathbf{x}_0,\epsilon}\Big\|\mathbf{s}_\theta(\mathbf{x}_0 + \sigma\epsilon) + \frac{\epsilon}{\sigma}\Big\|^2. \tag{17.3}$$

その最小点もやはり条件付き平均（今度は条件付きスコアの条件付き平均）であり、Vincent の定理はその正体を次のように特定する。すなわち、**ノイズを加えたデータのスコアは、条件付きスコアの事後平均に等しい**。

$$\mathbb{E}\Big[-\frac{\epsilon}{\sigma}\,\Big|\,\tilde{\mathbf{x}}\Big] = \nabla \log p_\sigma(\tilde{\mathbf{x}}). \tag{17.4}$$

証明はベイズの定理を 2 行使うだけで済む。ノイズ付きの密度はクリーンな密度をノイズでぼかした $p_\sigma(\tilde{\mathbf{x}}) = \int p(\mathbf{x}_0)\, p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}_0)\, d\mathbf{x}_0$ である。積分記号の下で微分し、カーネルの勾配を「カーネルとその対数の勾配の積」に書き換えて、両辺を $p_\sigma(\tilde{\mathbf{x}})$ で割ると次の式を得る。

$$\nabla \log p_\sigma(\tilde{\mathbf{x}}) = \int \frac{p(\mathbf{x}_0)\, p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}_0)}{p_\sigma(\tilde{\mathbf{x}})}\, \nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}_0)\, d\mathbf{x}_0.$$

被積分関数中の分数は事後分布 $p(\mathbf{x}_0 \mid \tilde{\mathbf{x}})$ に対するベイズの定理そのものであるから、右辺は条件付きスコアの事後平均となり、Equation 17.4 が得られる。この 2 行の導出ではノイズがガウス分布であることは一切使っていない。勾配を積分記号の中へ入れられる条件については 38.6.1.2 節で述べる。

したがって、ペアごとの目標に対する回帰によって、明示的に書き下すことすらなかった p_σ のスコアを学習できる。これこそが、本章で生成器を回帰によって訓練できる理由である。データのスコアは直接評価できなくても、自分で加えたノイズのスコアなら評価できるからである。

17.1.2.2 Tweedie の公式

クリーンな点への回帰と $-\epsilon/\sigma$ への回帰という 2 つの回帰問題では、それぞれの目標が写像 $\mathbf{x}_0 \mapsto (\mathbf{x}_0 - \tilde{\mathbf{x}})/\sigma^2$ で結ばれている。この写像は $\tilde{\mathbf{x}}$ を固定すればアフィン写像であるため、両者の最小点である 2 つの条件付き平均も同じ写像で結ばれる。これを事後平均について解くと、**Tweedie の公式** (Efron 2011) が得られる。

$$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \tilde{\mathbf{x}} + \sigma^2 \nabla \log p_\sigma(\tilde{\mathbf{x}}). \tag{17.5}$$

最良のノイズ除去器は、ノイズ付き密度のスコアの方向にステップサイズ σ^2 で 1 ステップ進む（Figure 38.23）。したがって、ノイズ除去器、スコア、ノイズ予測器（noise predictor）は、1 つの事後平均を表す 3 通りの書き方にほかならない。

$$\begin{aligned} \hat{\mathbf{x}}_0(\tilde{\mathbf{x}}) &= \tilde{\mathbf{x}} + \sigma^2\, \hat{\mathbf{s}}(\tilde{\mathbf{x}}), \\ \hat{\epsilon}(\tilde{\mathbf{x}}) &= \frac{\tilde{\mathbf{x}} - \hat{\mathbf{x}}_0(\tilde{\mathbf{x}})}{\sigma} = -\sigma\, \hat{\mathbf{s}}(\tilde{\mathbf{x}}). \end{aligned} \tag{17.6}$$

これら 3 つのうちどれか 1 つを出力するよう訓練されたネットワークがあれば、残りの 2 つも直ちに得られる。

最も単純な例を見れば、このステップの働きがわかる。データがガウス分布 $\mathcal{N}(\mu, v\mathbf{I})$ に従うとする。このとき $p_\sigma = \mathcal{N}(\mu, (v+\sigma^2)\mathbf{I})$ のスコアは $-(\tilde{\mathbf{x}} - \mu)/(v+\sigma^2)$ であり、Tweedie の公式から次式が得られる。

$$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \mu + \frac{v}{v+\sigma^2}\,(\tilde{\mathbf{x}} - \mu).$$

このノイズ除去器は、平均からのノイズを含んだずれに係数 $v/(v+\sigma^2)$ を掛けて縮小（shrinkage）し、座標ごとの事後分散（posterior variance）は $v\sigma^2/(v+\sigma^2)$ となる。$v = s^2 = 0.25$、$\sigma = 0.3$ のとき縮小係数は 0.74 であり、座標ごとの二乗誤差は $\sigma^2 = 0.09$ から 0.066 へと減少する。通し例においても、谷から離れた場所での事後平均は最も近い平均に向けたこの縮小に一致するため、点の 2 座標合計では、ノイズ除去によって二乗誤差が 0.18 から約 0.13 に下がるはずである。

17.1.2.3 訓練されたスコア

学習されたスコアは、この最良のノイズ除去器に到達できるだろうか。それを確かめるにはネットワークが必要であり、本章ではすべての 2 次元実験に 1 つの小さな MLP を用いる。このネットワークは点の座標に加え、ノイズレベルの正弦と余弦（次節以降では、データからノイズへの経路に沿った時刻 $t \in [0,1]$ の正弦と余弦）を入力として受け取る。本節ではノイズレベルが固定されているため、これらの特徴量は何の働きもしない。次節以降では 1 つのネットワークがあらゆるレベルを担当し、近接するレベルを識別しなければならないため、周波数は t の単位あたり 1 から 64 ラジアンまで変化し、最も速い特徴量は $[0,1]$ 上で約 10 周期分振動する。数千規模の整数トークン位置向けに設計された 10.4 節の位置エンコーディングでは、この区間上でほぼ定数または線形になってしまう。なお、17.8 節の少数ステップ生成器では再び周波数を下げるが、その理由は同節で述べる。

```
def time_embedding(t, dim, freqs=(1.0, 64.0)):
    """Sines and cosines of t at dim / 2 frequencies, geometric from freqs[0]
    to freqs[1] radians per unit of t (t in [0, 1])."""
    f = torch.exp(torch.linspace(math.log(freqs[0]), math.log(freqs[1]),
```

```
                                dim // 2, device=t.device))
    args = t.float()[:, None] * f
    return torch.cat([args.sin(), args.cos()], 1)

class TimeMLP(nn.Module):
    """An MLP on a point x and the time features of one or more times;
    time_freqs: see time_embedding."""
    def __init__(self, hidden=256, depth=3, *, dim=2, num_times=1,
                 time_dim=64, time_freqs=(1.0, 64.0)):
        super().__init__()
        self.time_dim, self.time_freqs = time_dim, time_freqs
        self.embed = d2l.make_mlp([time_dim * num_times, hidden, hidden],
                                  nn.SiLU())
        self.inp = nn.Linear(dim, hidden)
        self.body = nn.Sequential(  # depth layers, each after a SiLU
            nn.SiLU(), d2l.make_mlp([hidden] * depth + [dim], nn.SiLU()))

    def forward(self, x, *ts):
        feats = [time_embedding(t, self.time_dim, self.time_freqs) for t in ts]
        return self.body(self.inp(x) + self.embed(torch.cat(feats, 1)))
```

訓練ループも本章全体で共通のものを用いる。各ステップで新たなミニバッチと新たなノイズを生成し、Adam で更新する。さらに重みの指数移動平均（exponential moving average、EMA）も保持するが、これが重要になるのは 17.4 節以降である。その他のオプションについては docstring に記載してある。

```
def train_regression(model, loss_fn, batch_fn, num_steps, *,
                     lr=1e-3, cosine=False, ema_half_life=1000,
                     compiled=True):
    """Adam on loss_fn(model, batch_fn()) -> (model, EMA model, losses).

    The learning rate is constant or, with cosine=True, decays to zero along
    a cosine. The EMA model averages the weights (d2l.EMA): the weights of
    ema_half_life updates ago count half as much as the current ones. On a
    GPU, Adam is fused into one kernel and, with compiled=True, the loss and
    its gradient run as CUDA graphs: the networks of this chapter are so
    small that launching their kernels one at a time would cost more time
    than running them."""
    params = list(model.parameters())
    opt = torch.optim.Adam(params, lr=lr, fused=params[0].is_cuda)
    sched = (torch.optim.lr_scheduler.CosineAnnealingLR(opt, num_steps)
             if cosine else None)
    if compiled and params[0].is_cuda:  # loss and gradient as CUDA graphs
        loss_fn = torch.compile(loss_fn, mode='reduce-overhead',
                                dynamic=False)
    ema, losses = d2l.EMA(model, ema_half_life), []
    for _ in range(num_steps):
        loss = loss_fn(model, batch_fn())
        opt.zero_grad()
        loss.backward()
        opt.step()
        if sched:
```

```
            sched.step()
        ema.update(model)
        losses.append(loss.detach().clone())  # the graph reuses its output
    return model, ema.model, torch.stack(losses)
```

混合分布に対しては、隠れ層 2 層（各 128 ユニット）のネットワークを用い、2,000 個の訓練点から毎回 256 個を選んで新たなノイズを加えながら Equation 17.3 を 2,000 ステップ最小化し、訓練されたスコアを点のみの関数として保持する。

```
def dsm_loss(net, x, sigma):
    """Denoising score matching at one noise level: regress onto -eps/sigma."""
    eps = torch.randn_like(x)
    s = net(x + sigma * eps, torch.full((len(x),), sigma))
    return 0.5 * ((s + eps / sigma) ** 2).sum(1).mean()

def fit_score(data, sigma, seed=1, num_steps=2000):
    """Train a score network at one noise level and return it as a function."""
    torch.manual_seed(seed)
    batch_fn = lambda: data[torch.randint(0, len(data), (256,))]
    net, _, _ = d2l.train_regression(
        d2l.TimeMLP(128, 2), lambda net, x: dsm_loss(net, x, sigma),
        batch_fn, num_steps)
    return lambda x: net(x, torch.full((len(x),), sigma)).detach()
```

$\sigma = 0.3$ において新たに生成した 20,000 個の点にノイズを加え、クリーンな点に対する 3 通りの推定値、すなわちノイズ付きの点そのもの、厳密な事後平均、そして訓練されたスコアを用いた Tweedie のステップを比較する。

```
sigma = 0.3
score = fit_score(data, sigma)
x0 = mix.sample(20000)
xt = x0 + sigma * torch.randn_like(x0)
exact = mix.posterior_mean(xt, 1.0, sigma)
for name, x0_hat in (('noisy point', xt), ('posterior mean', exact),
                     ('Tweedie, trained score', xt + sigma ** 2 * score(xt))):
    print(f'{name}: squared error {((x0_hat - x0) ** 2).sum(1).mean():.4f}')
```

```
noisy point: squared error 0.1782
posterior mean: squared error 0.1317
Tweedie, trained score: squared error 0.1328
```

ガウス分布の場合の予測どおり、厳密な事後平均は二乗誤差を 0.18 から約 0.13 へと引き下げ、訓練されたスコアによる Tweedie のステップもその約 1% 以内に迫っている。2,000 個の点に対する 2,000 ステップの訓練を経て、ネットワークはほぼ最適なノイズ除去器となり、したがって実用的なスコアとなっている。

もっとも、スコアとして見れば見かけほど優秀ではない。スコアは $\sigma^2 = 0.09$ 倍されてノイズ除去器に入るため、Tweedie のステップが事後平均に対して生じる過剰誤差の期待値は、スコアの平均二乗誤差のちょうど σ^4 倍になる（$\tilde{\mathbf{x}}$ が与えられたとき事後平均の誤差の期待値は $\mathbf{0}$ になるため、交差項は消える）。したがって、約 0.001 というわずかな差は、スコア誤差 $\frac{1}{2}\mathbb{E}\|\hat{\mathbf{s}} - \nabla \log p_\sigma\|^2$ に換算すると約 0.06 に相当する。ノイズ除去器としてはほぼ完璧に見えても、そのスコアは目に見えてずれていることがあり、そしてサンプラが従うのはスコアのほうなのである。

17.1.2.4 スコアと正規化定数

なぜ密度ではなくスコアを学習するのか。**エネルギー**（energy）と呼ばれるスカラー出力 E_θ をもつネットワークは、**エネルギーベースモデル**（energy-based model）を通じて次のように密度を定める。

$$p_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta}, \qquad Z_\theta = \int \exp(-E_\theta(\mathbf{x}))\, d\mathbf{x}. \tag{17.7}$$

正規化定数があるからこそ学習に意味が生じる。全体の確率質量が固定されているため、訓練点における密度を高めれば他の場所での密度が下がるからである。ところが残念なことに、Z_θ は入力空間全体にわたる積分であってパラメータ更新のたびに変化し、対数尤度の勾配には $\nabla_\theta \log Z_\theta = -\mathbb{E}_{p_\theta}[\nabla_\theta E_\theta]$ が含まれる。この期待値はモデルからのサンプリング、実際にはマルコフ連鎖を用いて推定しなければならない。一方、スコアには Z_θ が一切現れない。Z_θ は $\mathbf{x}$ に依存しないため、$\nabla_{\mathbf{x}} \log p_\theta = -\nabla_{\mathbf{x}} E_\theta$ となるからである。

ベクトル場 $\mathbf{s}$ とデータとの比較には、データが存在する場所における 2 つの場の平均二乗距離である**フィッシャーダイバージェンス**（Fisher divergence）を用いる。

$$D_{\mathrm{F}}(p, \mathbf{s}) = \tfrac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p} \big\| \mathbf{s}(\mathbf{x}) - \nabla \log p(\mathbf{x}) \big\|^2. \tag{17.8}$$

この式には未知のデータスコアが含まれるが、**Hyvärinen の恒等式** (Hyvärinen 2005) は部分積分（integration by parts）によってこれを取り除く。すなわち定数項を除けば、D_{F} は正規化定数もマルコフ連鎖も不要なデータ上の平均 $\mathbb{E}_p\big[\tfrac{1}{2}\|\mathbf{s}\|^2 + \nabla \cdot \mathbf{s}\big]$ に等しい（38.6.1 節）。しかし第 2 項の $\nabla \cdot \mathbf{s}$（$\mathbf{s}$ のヤコビ行列のトレース）には入力次元ごとに 1 回の逆伝播が必要であり、画像に対してはコストが大きすぎる。ノイズ除去はこのコストも解消する。Vincent の定理により、Equation 17.3 は定数項を除いて p_σ に対するフィッシャーダイバージェンスに一致するからである。

17.1.3 単一ノイズレベルの限界

1 つのノイズレベルで訓練されたノイズ除去器から得られるのは、1 つのぼかされた分布 p_σ のスコアである。p からサンプリングするには p のスコアが必要であるため、最初の問いはどの σ を選ぶべきかということになる。

17.1.3.1 ノイズレベルの選択

ノイズレベルが小さい場合には 2 つの問題が生じる。1 つ目は統計的な問題である。通し例のように滑らかな密度をもつデータでは、回帰目標 $-\epsilon/\sigma$ の大きさは $1/\sigma$ のオーダーで増大する一方、その平均であるスコアは有界にとどまるため、小さな σ では目標の大部分がノイズとなってネットワークの当てはまりが悪くなる。たとえ厳密なスコアを用いても、条件付き平均の周りでの目標の分散が損失に残り、d 次元ではこの損失の下限は d/σ^2 のオーダーで σ の減少とともに増大する（38.6.1.2 節）。したがって、ノイズ除去損失が 0 よりかなり高い値で頭打ちになっても、ネットワークの性能が悪いことを意味するわけではない。

2 つ目の問題は、ピクセル空間内の低次元集合の近くに分布していると考えられる画像においてとりわけ重要になる (Bengio, Courville, et al. 2013)。極端な例として、平面上の 1 本の直線上に乗っているデータを考えよう。ノイズによって直線は帯状に広がり、帯を横切る方向の p_σ は分散 σ^2 のガウス分布になるため、帯を横切る方向のスコアは $-\tilde{x}_2/\sigma^2$（直線からの距離を $\tilde{x}_2$ とする）となる（演習問題 3）。その傾き $-1/\sigma^2$ は $\sigma \to 0$ で際限なく増大する。帯を横切る方向で

は目標 $-\epsilon_2/\sigma$ がこのスコアに厳密に一致するため、そこでの困難はノイズではない。ネットワークは、データ集合を横切る方向には断崖絶壁へと化す場を表現しつつ、データ集合に沿う方向ではほとんどノイズばかりの目標を扱わなければならないのである。

ノイズレベルが大きい場合には逆の問題が生じる。回帰問題の条件は良くなるが、得られる答えは p_σ のスコアであって p のスコアではない。ガウス成分 $\mathcal{N}(\mu, s^2\mathbf{I})$（$d$ 次元）が 1 つの場合、2 つのスコアは $-(\mathbf{x}-\mu)/(s^2+\sigma^2)$ と $-(\mathbf{x}-\mu)/s^2$ であり、その差の二乗ノルムを p で平均すると次のバイアスが得られる。

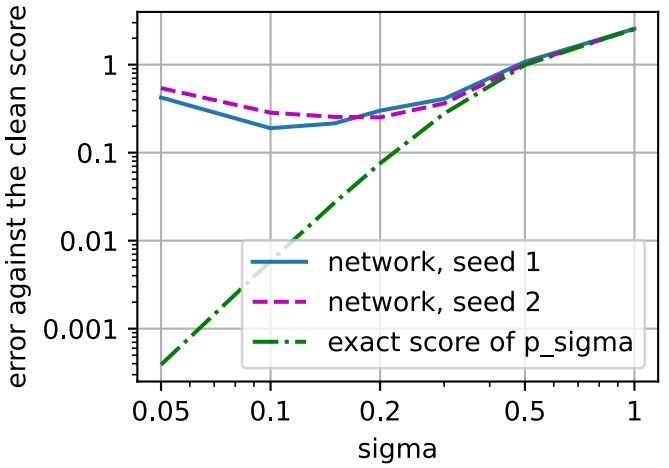
$$\tfrac{1}{2}\,\mathbb{E}_p\big\|\nabla \log p_\sigma - \nabla \log p\big\|^2 = \frac{d\,\sigma^4}{2s^2(s^2+\sigma^2)^2}.$$

このバイアスは最初 σ^4 のオーダーで増大する。$s = 0.5$、$d = 2$ のとき、その値は 0.0004（$\sigma = 0.05$）、0.28（$\sigma = 0.3$）、2.6（$\sigma = 1$）となり、モードがよく分離した混合分布でもほぼ同じ値になる。クリーンなスコアの大きさ $\tfrac{1}{2}\mathbb{E}_p\|\nabla \log p\|^2 = d/(2s^2)$ で割ると $(\sigma^2/(s^2+\sigma^2))^2$ となり、これは σ/s のみに依存する。平滑化されたスコアが二乗平均平方根（root mean square）の意味でクリーンなスコアの 10% 以内に収まるのは $\sigma < s/3 \approx 0.17$ の範囲であるため、このあたりからバイアスが無視できなくなると予想される。

これら 2 つの問題の間には最良のノイズレベルが存在するはずである。そこで 0.05 から 1 までの 7 つのレベルそれぞれにおいて 2 つのシードでネットワークを訓練し、20,000 個のテスト点を用いてクリーンなスコアに対するフィッシャーダイバージェンス Equation 17.8 を推定し、厳密な平滑化スコアのバイアスと並べて比較する。

```
sigmas, seeds = [0.05, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0], (1, 2)
scores = {(sg, seed): fit_score(data, sg, seed)
          for seed in seeds for sg in sigmas}
x_test = mix.sample(20000)
target = mix.score(x_test)                    # the clean score
def score_error(score, x, target):
    return 0.5 * ((score(x) - target) ** 2).sum(1).mean().item()
learned = {seed: [score_error(scores[sg, seed], x_test, target)
                  for sg in sigmas] for seed in seeds}
bias = [score_error(lambda x: mix.score(x, 1.0, sg), x_test, target)
        for sg in sigmas]
for seed in seeds:
    print(f'trained, seed {seed}:', [round(e, 3) for e in learned[seed]])
print('exact smoothed: ', [round(e, 4) for e in bias])
d2l.plot(sigmas, [*learned.values(), bias], 'sigma',
         'error against the clean score', xscale='log', yscale='log',
         legend=[f'network, seed {seed}' for seed in seeds]
         + ['exact score of p_sigma'])
```

```
trained, seed 1: [0.423, 0.189, 0.215, 0.3, 0.409, 1.077, 2.559]
trained, seed 2: [0.541, 0.285, 0.255, 0.252, 0.363, 1.019, 2.532]
exact smoothed:  [0.0004, 0.0059, 0.0271, 0.0755, 0.2781, 0.9924, 2.5538]
```



学習されたスコアの誤差は U 字型の曲線を描く。右側ではバイアスの曲線に沿い、その差は約 0.1 以内（$\sigma = 0.5$ 以上）に収まっている。ネットワークはぼかされた目標によく当てはまっているが、その目標自体が本来のスコアからずれているのである。左側ではバイアスが消えるにもかかわらず、目標の大部分がノイズとなるために誤差は再び上昇し、2 つ目のシードでは誤差が半分以上変化する箇所もある。最良のレベルは $\sigma = 0.1$ から 0.2 の間、すなわち成分の幅の 5 分の 1 から 5 分の 2 であり、先ほどのバイアスの目安である $s/3$ 付近に位置し、そのときの誤差は約 0.15〜0.25 となる。これが単一ノイズレベルにおける指針である。すなわち、σ をデータのスケールを単位として測り、ネットワークが当てはめられる限りにおいてなるべく小さく取ればよい。

17.1.3.2 モード重みに対する不感性

フィッシャーダイバージェンスにおける 0.2 という誤差は小さく聞こえるが、何と比べて小さいのだろうか。通し例を、その重みだけを均等にした分布と比較してみよう。この誤った重みをもつモデルは、1 点あたり 0.07 nat の尤度を失う（これが両者のカルバック・ライブラー・ダイバージェンスである）。ところがそのスコアが真のスコアと異なるのは Equation 17.1 の負担率を通じてのみであり、負担率が異なるのは p がほぼ 0 となる谷の部分に限られるため、p のもとでの平均であるフィッシャーダイバージェンスはこの違いをほとんど検知できない。

フィッシャーダイバージェンスの被積分関数はモンテカルロサンプルがほとんど落ちない谷に集中しているため、サンプリングではなく 721×721 グリッド上の数値積分（quadrature）によって両方のダイバージェンスを計算し、さらに両方の混合分布を同じノイズでぼかした場合についても同様の比較を繰り返す。

```
mix_eq = d2l.GaussianMixture(mix.means.tolist(), [1 / 3] * 3, mix.std)
g = torch.linspace(-9, 9, 721)
grid = torch.stack(torch.meshgrid(g, g, indexing='ij'), -1).reshape(-1, 2)
area = (g[1] - g[0]) ** 2                    # of one grid cell
for sigma in (0.0, 0.5, 1.0):
    log_p = mix.log_prob(grid, 1.0, sigma)
    diff = mix.score(grid, 1.0, sigma) - mix_eq.score(grid, 1.0, sigma)
    fisher = area * (log_p.exp() * 0.5 * (diff ** 2).sum(1)).sum()
    kl = area * (log_p.exp()
                 * (log_p - mix_eq.log_prob(grid, 1.0, sigma))).sum()
    print(f'sigma = {sigma}: Fisher divergence {fisher:.1e}, KL {kl:.3f} nats')
```

```
sigma = 0.0: Fisher divergence 1.5e-05, KL 0.069 nats
sigma = 0.5: Fisher divergence 1.4e-03, KL 0.069 nats
sigma = 1.0: Fisher divergence 9.5e-03, KL 0.064 nats
```

クリーンな混合分布どうしでは、フィッシャーダイバージェンスは約 10^{-5} と、訓練されたネットワークの誤差（いずれも 0.1 を下回らなかった）より 4 桁も小さい一方、KL ダイバージェンスは 0.07 nat ある。したがって、クリーンなデータの近くで訓練されたスコアモデルは重みを学習できない。誤った重みをもつ場でも、正しい重みをもつ場と同程度によく当てはまってしまうからである（37.5.1 節、および 37.5 節の演習問題 2）。スコアはデータの近くでどれほど優れていても、各モードがどの頻度で生じるかについては何も知らないままになり得るのである。

ノイズを加えると重みが見えるようになる。σ が大きくなるにつれて各成分が谷へと広がり（Figure 17.2）、p_σ が確率質量をもつ領域で負担率に差が生じるため、KL ダイバージェンスがほとんど変化しない一方で、フィッシャーダイバージェンスは約 10^{-3}（$\sigma = 0.5$）および 10^{-2}（$\sigma = 1$）へと上昇する。それでも $\sigma = 1$ においてすら、このフィッシャーダイバージェンスは先ほどのスイープ実験におけるどの誤差よりも 10 倍以上小さいため、ノイズを加えただけで重みの学習が容易になるわけではない。ノイズがまず変えるのは、マルコフ連鎖が谷を越えられるようになるという点である。

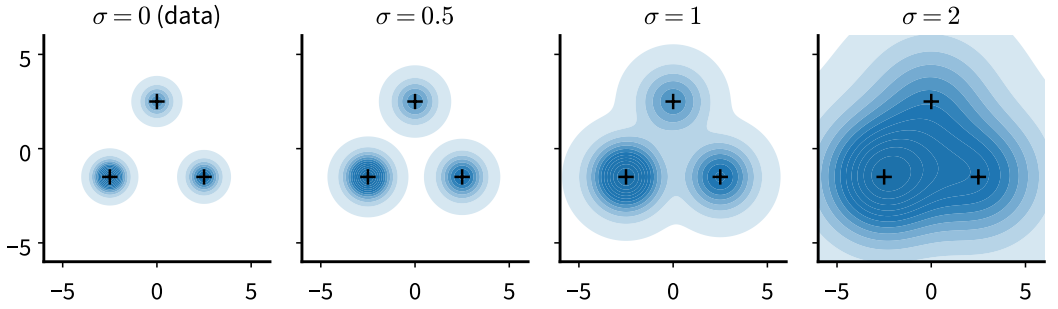


Figure 17.2: 標準偏差 σ を大きくしながらガウスノイズでぼかした通し例。十字は各成分の平均を表す。σ が大きくなるにつれてモード間の谷が埋まり、$\sigma = 2$ では 1 つの広い山だけが残る。

17.1.3.3 スコアによるサンプリング

サンプルを抽出するにはスコアがあれば十分であり、その最も単純な方法は 1 世紀以上前から知られている。1908 年、Paul Langevin は流体中で力に押されつつ周囲の分子からランダムな衝突を受ける粒子に対して、この種の方程式を書き下した。離散形式の**ランジュバン動力学**は任意の点から出発し、次の更新を繰り返す。

$$\mathbf{x} \leftarrow \mathbf{x} + \frac{h}{2}\,\mathbf{s}(\mathbf{x}) + \sqrt{h}\,\xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.9}$$

ここで h は小さなステップサイズである。ドリフト（drift）が点を対数密度の高い方向へ動かし、ノイズがモードへの一点崩壊を防ぐ。連続時間極限では、定常分布（stationary distribution）は p となり（$\mathbf{s} = \nabla \log p$ のとき。38.6.2 節）、十分に長く走らせた連鎖は初期点を忘れる。離散更新にはわずかなずれがあり、分散 v のガウス分布に対する定常分散は $v/(1 - h/(4v))$ となって、本節の成分では $h = 0.05$ において 5% の過剰が生じる。この更新式には密度そのものが一切現れないため、何らかの関数の勾配になっていない訓練済みの場であっても、任意のベクトル場で駆動できる。

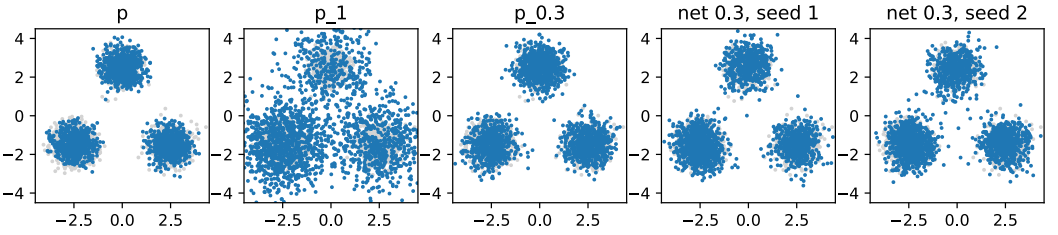
```
@torch.no_grad()
def langevin(score_fn, x, step, num_steps):
    """Langevin steps x <- x + step / 2 * score + sqrt(step) * noise."""
    for _ in range(num_steps):
        x = x + step / 2 * score_fn(x) + step ** 0.5 * torch.randn_like(x)
    return x
```

各モードに約 3 分の 1 ずつ割り振られるよう $\mathcal{N}(\mathbf{0}, 9\mathbf{I})$ から 2,000 本の連鎖を出発させ、$h = 0.05$ で 5 つの場を用いて 2,000 ステップ走らせる。用いる場は、p の厳密なスコア、p_σ の $\sigma = 1$ および $\sigma = 0.3$ における厳密なスコア（それぞれ `p_1`、`p_0.3` と表示）、そして先ほどのスイープ実験で $\sigma = 0.3$ において訓練した 2 つのネットワーク（これらを p のスコアであるかのように用いる）である。0.3 における厳密な場はネットワークが模倣するよう訓練された対象そのものであるため、ネットワークの振る舞いのうちどれがノイズレベルに由来し、どれがネットワーク自身の近似に由来するかを見分けられる。

p の厳密なスコアを用いれば、連鎖の最終的な重みは 0.5、0.3、0.2 になるだろうか。2,000 ステップが初期点を忘れるのに十分な長さであれば、そうなるはずである。$\sigma = 1$ における連鎖は p_σ からサンプリングするはずであり、その各成分の座標ごとの分散は $s^2 + \sigma^2 = 1.25$ と、p の 5 倍になる。完璧なサンプラに対する先ほどの計算で $2 \cdot 1.25$ を $2s^2$ の代わりに用いると、外れ点は $e^{-2.25/2.5} \approx 41\%$、広がりは 1 付近（データの約 2 倍にとどまる）と予測される。

```
torch.manual_seed(2)
x_init = 3 * torch.randn(2000, 2)
fields = {'p': mix.score,                       # exact scores of p, p_1, p_0.3
          'p_1': lambda x: mix.score(x, 1, 1),
          'p_0.3': lambda x: mix.score(x, 1, 0.3),
          'net 0.3, seed 1': scores[0.3, 1],    # trained at sigma = 0.3
          'net 0.3, seed 2': scores[0.3, 2]}
samples = [d2l.langevin(f, x_init, 0.05, 2000) for f in fields.values()]
print(f'{"start":>15}:', mix.summary(x_init))
print(f'{"data":>15}:', mix.summary(data))
for name, x in zip(fields, samples):
    print(f'{name:>15}:', mix.summary(x))
d2l.scatter_panels(samples, list(fields), data=data, size=2.3);
```

```
          start: {'weights': [0.308, 0.329, 0.363], 'stray': 0.751, 'spread': 1.051}
           data: {'weights': [0.5, 0.317, 0.183], 'stray': 0.012, 'spread': 0.488}
              p: {'weights': [0.312, 0.335, 0.354], 'stray': 0.0095, 'spread': 0.501}
            p_1: {'weights': [0.498, 0.298, 0.204], 'stray': 0.4245, 'spread': 0.986}
          p_0.3: {'weights': [0.329, 0.334, 0.338], 'stray': 0.0435, 'spread': 0.627}
net 0.3, seed 1: {'weights': [0.454, 0.262, 0.284], 'stray': 0.0485, 'spread': 0.625}
net 0.3, seed 2: {'weights': [0.458, 0.289, 0.253], 'stray': 0.0525, 'spread': 0.628}
```



p の厳密なスコアはすべての連鎖を正しい局所的な広がりをもつモードへと導くが、モード重みは初期値（それぞれ $\approx 1/3$）のまま動かない。モード間の谷は $-\log p$ においてモードより 10〜12 nat 高いため、期待される Kramers の脱出時間は 10^6 ステップのオーダー（$h = 0.05$ のとき。演習問題 6）となり、2,000 ステップではモード間を混合できないのである。逆に $\sigma = 1$ ではノイズが谷を埋めるため、連鎖が混合して真の重み $(0.5, 0.3, 0.2)$ を回復できるが、サンプリングされるのは裾の広い分布 p_1（成分の分散が $s^2 + \sigma^2 = 1.25$ で、$\approx 41\%$ の外れ点と 1 付近の広がりを生じる）になってしまう。

$\sigma = 0.3$ で訓練されたネットワークは $p_{0.3}$ のスコア（成分の分散 0.34、外れ点 $\approx 4\%$、広がり ≈ 0.6）を近似する。$p_{0.3}$ の厳密なスコアは 2,000 ステップでは 7〜9 nat の障壁を越えてほとんど混合しない（最大重みのモードの変化は高々 0.03）が、学習されたネットワークはモード間でより多くの質量を移動させる。データが疎な谷における近似誤差が、人工的なドリフトの通り道を作り出すためである。

このように、単一のノイズレベルは根本的なトレードオフに直面する。小さな σ は鋭いモードを保つが連鎖を閉じ込め、フィッシャーダイバージェンスもモード重みに対して不感なままになる。一方、大きな σ はモード間を速やかに混合させるが、強く平滑化された密度からしかサンプリングできない。17.2 節では、粗いレベルから細かいレベルまですべてのノイズレベルにわたってアニーリングすることでこの問題を解決する。

17.1.4 画像のノイズ除去

これまでの議論は平面の外でも成り立つだろうか。ノイズ除去損失にはデータの次元に依存する箇所が一切ないため、画像に対してもそのまま適用できる。失われるのは真の基準値である。画像の事後平均は計算できないため、計算可能な最良のノイズ除去器である線形ノイズ除去器とネットワークを比較する。ここではピクセル値を $[-1, 1]$ にスケーリングした Fashion-MNIST（3.2 節）を 2 つの小さな補助関数で読み込み・表示し、標準偏差 $\sigma = 0.5$ のノイズを用いる。ネットワークは 32 チャネル・SiLU 活性化関数をもつ 4 層の 3×3 畳み込みで構成し、スコアネットワークと同じ訓練ループを用いて、毎回 128 枚の画像のミニバッチに新たなノイズを加えながら 600 ステップ訓練する。得られた結果に再現性の裏付けを添えるため、2 つのシードから 2 回訓練する。このネットワークはノイズ ϵ を予測するため、Equation 17.6 によりその出力を $-\sigma$ で割ったものは、密度を一度も書き下すことなく得られた 784 次元のノイズ付き画像のスコアとなり、$\tilde{\mathbf{x}} - \sigma\hat{\epsilon}$ がノイズ除去後の画像となる。

```
def load_fashion_mnist_pm1(binary=False):
    """Fashion-MNIST (X, y, X_test, y_test), images in [-1, 1] or binary."""
    data, device = d2l.FashionMNIST(), d2l.try_gpu()  # on the GPU if any
    def prep(X):  # (n, 1, 28, 28) floats or (n, 28, 28) binary pixels
        return (X > 127).long() if binary else X[:, None].float() / 127.5 - 1
    return (prep(data.train.data).to(device), data.train.targets.to(device),
            prep(data.val.data).to(device), data.val.targets.to(device))

def show_gray(imgs, num_rows, num_cols, titles=None, scale=1.0):
    """Show images (n, 1, h, w) in [-1, 1], or binary (n, h, w), in gray."""
    imgs = imgs.detach().cpu()
    imgs = ((imgs[:, 0] + 1) / 2).clamp(0, 1) if imgs.is_floating_point() \
        else imgs.float()
    return d2l.show_images(imgs[..., None].expand(-1, -1, -1, 3), num_rows,
                           num_cols, titles, scale)
```

これらの補助関数を用意したうえで、$[-1, 1]$ にスケーリングした Fashion-MNIST 画像を読み込み、ノイズ予測（noise prediction）の目的関数 `noise_loss` を $\sigma = 0.5$ で定義して、4 層の畳み

込みノイズ除去器を 2 つのシードで 600 ステップずつ `d2l.train_regression` により訓練する。

```
X, _, X_test, _ = d2l.load_fashion_mnist_pm1()
sigma = 0.5
def noise_loss(net, x, sigma):
    """Squared error of the predicted noise; the score is -net / sigma."""
    eps = torch.randn_like(x)
    return ((net(x + sigma * eps) - eps) ** 2).mean()

batch_fn = lambda: X[torch.randint(0, len(X), (128,), device=X.device)]
nets = {}
for seed in (0, 1):
    torch.manual_seed(seed)
    net = nn.Sequential(nn.Conv2d(1, 32, 3, padding=1), nn.SiLU(),
                        nn.Conv2d(32, 32, 3, padding=1), nn.SiLU(),
                        nn.Conv2d(32, 32, 3, padding=1), nn.SiLU(),
                        nn.Conv2d(32, 1, 3, padding=1)).to(X.device)
    nets[seed], _, losses = d2l.train_regression(
        net, lambda net, x: noise_loss(net, x, sigma), batch_fn, 600)
    print(f'seed {seed}: last training loss {losses[-50:].mean():.3f}')
```

```
seed 0: last training loss 0.124
seed 1: last training loss 0.121
```

ピクセルあたりの予測ノイズの二乗誤差である損失は、どちらのシードでも 0 よりはるかに大きい 0.12 付近で頭打ちになる。これに $\sigma^2 = 0.25$ を掛けたものがノイズ除去画像の二乗誤差であり、ピクセルあたり約 0.03 となる。以下のテストでは、ネットワークが訓練時に見ていない画像でこれを確認する。

次に比較基準となる最良の**線形**ノイズ除去器を導入する。仮に画像が平均 μ、共分散 Σ のガウス分布に従うなら、Tweedie の公式は先ほどのガウス分布における縮小を Σ の各固有ベクトルに沿ってそれぞれの係数 $\lambda/(\lambda + \sigma^2)$ で適用する形になる。

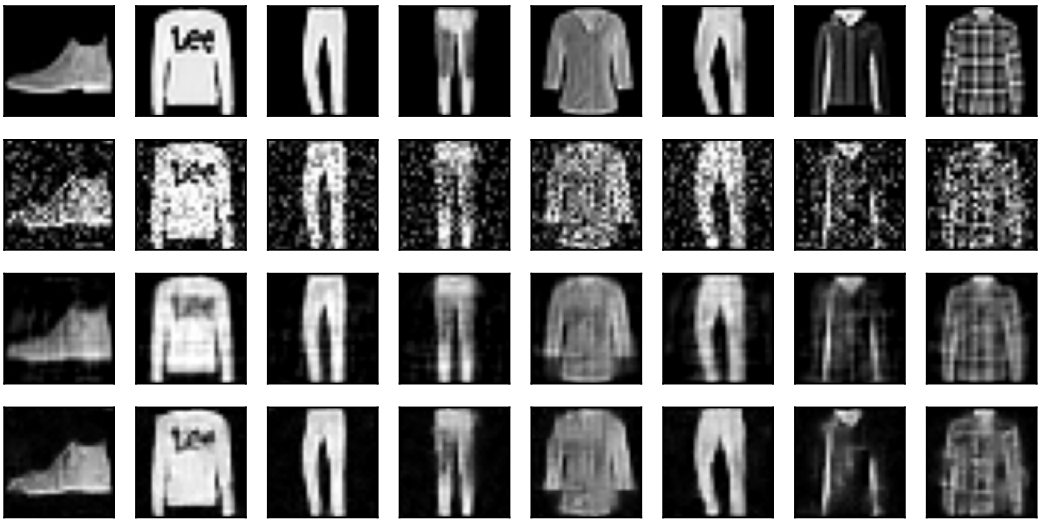
$$D_{\text{lin}}(\tilde{\mathbf{x}}) = \mu + \Sigma(\Sigma + \sigma^2\mathbf{I})^{-1}(\tilde{\mathbf{x}} - \mu). \tag{17.10}$$

この**ウィーナーフィルタ**（Wiener filter）は、画像の変動がノイズより大きい方向を残し、それ以外の方向を抑制するものであり、同じ平均と共分散をもつあらゆる分布に対して最良のアフィンノイズ除去器となる（演習問題 5）。60,000 枚の訓練画像から μ と Σ を推定し、1,000 枚のテスト画像上でノイズ付き入力、ウィーナーフィルタ、および 2 つのネットワークのピクセルあたり二乗誤差を比較する。各平均値の横には、画像間にわたる標準誤差もあわせて出力する。図の各行は上から順に、クリーンな画像、ノイズを加えた画像、フィルタの出力、そして 1 つ目のネットワークの出力を示している。

```
Xf = X.reshape(len(X), -1)
mu = Xf.mean(0)
Sigma = (Xf - mu).T @ (Xf - mu) / len(Xf)      # (784, 784) pixel covariance
A = torch.linalg.solve(Sigma + sigma ** 2 * torch.eye(784, device=X.device),
                       Sigma)                  # (Sigma + sigma^2 I)^-1 Sigma
torch.manual_seed(1)
with torch.no_grad():
    x = X_test[:1000]
```

```
    xt = x + sigma * torch.randn_like(x)
    x_lin = (mu + (xt.reshape(1000, -1) - mu) @ A).reshape(x.shape)
    x_net = {seed: xt - sigma * net(xt)        # Tweedie: xt + sigma^2 * score
             for seed, net in nets.items()}
outputs = {'noisy': xt, 'Wiener': x_lin,
           **{f'network, seed {seed}': xh for seed, xh in x_net.items()}}
for name, xh in outputs.items():               # error of each of 1,000 images
    err = ((xh - x) ** 2).flatten(1).mean(1)
    print(f'{name:>16}: per-pixel squared error {err.mean():.4f} '
          f'(standard error {err.std() / math.sqrt(len(err)):.4f})')
d2l.show_gray(torch.cat([x[:8], xt[:8], x_lin[:8], x_net[0][:8]]), 4, 8);
```

```
         noisy: per-pixel squared error 0.2493 (standard error 0.0004)
       Wiener: per-pixel squared error 0.0447 (standard error 0.0005)
network, seed 0: per-pixel squared error 0.0312 (standard error 0.0004)
network, seed 1: per-pixel squared error 0.0303 (standard error 0.0004)
```



ノイズ付き入力のピクセルあたり二乗誤差は $\sigma^2 = 0.25$ であり、最適な線形（ウィーナー）フィルタは ≈ 0.045 を達成し、4 層の畳み込みネットワークはいずれも ≈ 0.031 に達する（誤差が 30% 低く、20 標準誤差以上離れている）。ウィーナーフィルタがすべての画像を単一の大域的な線形部分空間（subspace）へと縮小させ、ぼやけた輪郭や背景のざらつきを残すのに対し、畳み込みネットワークはノイズ付きの点を衣類画像の非線形多様体（nonlinear manifold）へと射影する（演習問題 3 参照）。ただし、事後平均は $\tilde{\mathbf{x}}$ と整合するすべてのクリーンな画像にわたって平均を取るため、ノイズで失われた微細な高周波テクスチャは依然として滑らかな平均に置き換えられてしまう。ノイズ除去器を鋭さを保つ生成器へと変えるには、複数のノイズレベルにわたる積分が必要になる（17.2 節および 17.3 節）。

17.1.5 まとめ

二乗ノイズ除去誤差 $\mathbb{E}\|D(\tilde{\mathbf{x}}) - \mathbf{x}_0\|^2$（ただし $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$）を最小化すると、事後平均 $\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}]$ が復元される。Vincent の定理と Tweedie の公式により $\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \tilde{\mathbf{x}} + \sigma^2 \nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}})$ が成り立つため、クリーンな点 $\hat{\mathbf{x}}_0$、ノイズ $\hat{\epsilon}$、およびスコア $\hat{\mathbf{s}} = -\hat{\epsilon}/\sigma$ の予測は、いずれも単一の条件付き期待値のアフィンな再パラメータ化にほかならない。スコアを学習すれば、エネルギー

ベースモデルにおける計算困難な正規化定数 Z_θ も、明示的スコアマッチング（explicit score matching）におけるヤコビ行列のトレース $\nabla \cdot \mathbf{s}$ も回避できる。しかし、単一のノイズレベル σ には厳しいトレードオフが伴う。小さな σ では目標の分散が大きくなり、低次元データ多様体の近くで勾配が急峻化し、フィッシャーダイバージェンスがモード重みに対して不感になり、さらにランジュバン連鎖が混合しなくなる。一方、大きな σ では速やかに混合してモード重みを回復できるものの、強く平滑化された密度 p_σ からしかサンプリングできない。

17.1.6 演習問題

1. **正規化なしの学習**。1 次元モデル $\tilde{p}(x) = \exp(-E(x))$（ただし $E(x) = (x-\mu)^2/(2\tau^2)$）を、少なくとも 2 つの異なる値をとるデータ $x_1, \dots, x_n$ に当てはめることを考える。
   1. 正規化を行わずに $\sum_i \log \tilde{p}(x_i)$ を μ と τ について最大化せよ。目的関数がその上限に近づくとき、データから離れた場所で $\tilde{p}$ はどのようになるか。
   2. 正規化したモデル $\tilde{p}/Z$ で同じことを行え。目的関数のどの項が最初の設問での破綻を防ぎ、そのときの最大点は何になるか。
   3. ネットワークによるエネルギーは通常、出力バイアス c をもつ線形層で終わる。正規化された密度 Equation 17.7 が c に依存しないことを示せ。$-\sum_i E_\theta(\mathbf{x}_i)$ のみに対して勾配上昇法を適用すると、c はどうなるか。

2. [code] **正規化定数の推定**。エネルギーの正規化定数 $Z = \int e^{-E(\mathbf{x})}\, d\mathbf{x}$ を計算する直接的な方法は重要度サンプリング（36.8.2 節）である。すなわち、$\mathbf{x}_1, \dots, \mathbf{x}_n$ を提案分布 q から抽出し、重み $w_i = e^{-E(\mathbf{x}_i)}/q(\mathbf{x}_i)$（その期待値は Z である）を平均する。$E(\mathbf{x}) = \|\mathbf{x}-\mathbf{m}\|^2/2$ とおくと、目標分布は正規化を除いて $\mathcal{N}(\mathbf{m}, \mathbf{I})$ となり、提案分布を $q = \mathcal{N}(\mathbf{0}, \mathbf{I})$ とする。
   1. $w/Z = \exp(\mathbf{m}^\top \mathbf{x} - \|\mathbf{m}\|^2/2)$ を導出し、ガウス分布の積率母関数を用いて $\mathbb{E}_q[w/Z] = 1$ および $\mathbb{E}_q[(w/Z)^2] = \exp(\|\mathbf{m}\|^2)$ を示せ。さらに、n 個のサンプルによる推定量の相対標準誤差が $\sqrt{(e^{\|\mathbf{m}\|^2} - 1)/n}$ であることを導け。相対誤差を 0.1 にするには（$\|\mathbf{m}\|^2 = 20$ のとき）、何個のサンプルが必要か。
   2. $\mathbf{m}$ を d 次元の全要素が 1 のベクトルとし、提案分布が各座標で標準偏差 1 つ分だけずれている状況を考える。$n = 10^5$ として Z を $d \in \{2, 4, 8, 16, 32\}$ について推定し、真の $Z = (2\pi)^{d/2}$ に対する推定値の比と**有効標本サイズ** $(\sum_i w_i)^2 / \sum_i w_i^2$ を報告して、両者を最初の設問の予測値と比較せよ（有効標本サイズは $n\, e^{-\|\mathbf{m}\|^2}$ に近くなるはずである）。d が大きくなると有効標本サイズが予測値を上回るのはなぜか。また、このことは診断指標としての有効標本サイズについて何を意味するか。
   3. 1 枚の画像は数十万個の座標をもつ。得られた結果から、画像のエネルギーベースモデルの正規化定数を推定すること、および最尤法でそれを訓練することについて何が言えるか。

3. **直線上のデータ**。データが $\mathbb{R}^2$ の第 1 座標軸上にあり、$\mathbf{x}_0 = (u, 0)$（ただし $u \sim q$ であり、q は $\mathbb{R}$ 上の滑らかな正の密度）であるとする。このとき p は $\mathbb{R}^2$ 上の密度をもたない。
   1. $p_\sigma(\tilde{\mathbf{x}}) = q_\sigma(\tilde{x}_1)\, \mathcal{N}(\tilde{x}_2; 0, \sigma^2)$（ただし q_σ は q を $\mathcal{N}(0, \sigma^2)$ でぼかしたもの）となることを示し、スコア $\nabla \log p_\sigma$ を書き下せ。
   2. スコアの第 2 成分は二乗平均が $1/\sigma^2$（p_σ のもとで）であり、傾きが $-1/\sigma^2$（$\tilde{x}_2$ に関する）であることを示せ。$\sigma \to 0$ のときこの両者はどうなるか。また、小さなノイズレベルでスコアを表現しなければならないネットワークにとって、これは何を意味するか。
   3. Tweedie の公式 Equation 17.5 を適用せよ。最適なノイズ除去器は $\tilde{\mathbf{x}}$ の各座標に何を行い、どのような幾何学的操作を実行するか。

4. 代わりにデータが k 次元の線形部分空間（$\mathbb{R}^d$ 内の）上にあるとする。部分空間に直交するスコア成分の二乗平均を計算せよ。

4. **2 点データと損失の下限。** 1 次元において $p = \tfrac{1}{2}\delta_{-1} + \tfrac{1}{2}\delta_{+1}$ とし、これにレベル σ のノイズを加える。

   1. ベイズの定理によって事後平均 $\mathbb{E}[x_0 \mid \tilde{x}]$ を計算し、Tweedie の公式 Equation 17.5 が要請するとおり $\tilde{x} + \sigma^2 \frac{d}{d\tilde{x}} \log p_\sigma(\tilde{x})$ に等しいことを示せ。
   2. $\tilde{x} = 0$ に対して最適なノイズ除去器は何を返すか。それが取り得るクリーンな値ではないにもかかわらず、二乗誤差のもとで最良の答えとなるのはなぜか。
   3. Equation 17.3 の最小値を $\frac{1}{2\sigma^4}\,\mathbb{E}\big[\operatorname{tr}\operatorname{Cov}(\mathbf{x}_0 \mid \tilde{\mathbf{x}})\big]$ と表し、1 点に集中していないあらゆるデータ分布に対してこれが正であることを示せ。
   4. 2 点データに対してこの最小値を $\tilde{x}$ に関する 1 次元積分として書き下し、$\sigma \in \{0.25, 0.5, 1, 2\}$ について数値積分で評価せよ。各値に $2\sigma^2$ を掛けてノイズ予測損失の単位で表せ。スケールし直したこの下限が $\sigma \to 0$ で 0 に収束する理由と、$\sigma \to \infty$ でも 0 に収束する理由を説明せよ。

5. [code] **最良の線形ノイズ除去器。**

   1. ウィーナーフィルタ Equation 17.10 が $\mathbb{E}\|D(\tilde{\mathbf{x}}) - \mathbf{x}_0\|^2$ をすべてのアフィン写像 D の中で最小化すること（平均 μ と共分散 Σ をもつあらゆるデータ分布に対して）、およびそのピクセルあたり二乗誤差が $\frac{1}{d}\sum_i \sigma^2\lambda_i/(\lambda_i + \sigma^2)$（ただし $\lambda_1, \dots, \lambda_d$ は Σ の固有値）であることを示せ。
   2. $\sigma = 0.5$ におけるこの予測値を `Sigma` から計算し、出力された `x_lin` の誤差と比較せよ。
   3. `Sigma` の固有値のうち σ^2 を上回るものの個数を $\sigma \in \{0.25, 0.5, 1\}$ について数え、対応する固有ベクトルをいくつか画像としてプロットせよ。各ノイズレベルにおいてフィルタは何を残しているか。

6. [code] **モードからの脱出。●** 通し例の下側 2 つのモード間にある谷を、1 次元における $\mathcal{N}(-2.5, 0.25)$ と $\mathcal{N}(2.5, 0.25)$ の等重み混合分布でモデル化する。ポテンシャルを $U = -\log p$ とすると、井戸の底は $a = \pm 2.5$、障壁は $b = 0$ にある。障壁の高さ $\Delta U = U(b) - U(a)$ が 1 に比べて大きいとき、Kramers の公式 (Kramers 1940) は、拡散過程 $dX = -\tfrac{1}{2}U'(X)\,dt + dW$（Equation 17.9 の連続時間極限）が一方の井戸から他方の井戸へ移るまでの期待時間を次式で与える。

$$\tau \approx \frac{4\pi}{\sqrt{U''(a)\,|U''(b)|}}\, e^{\Delta U}.$$

   1. ΔU、$U''(a)$、$U''(b)$ を計算し、τ を $h = 0.05$ におけるランジュバンステップ数に換算せよ。
   2. 通し例のモード 3 はモード 1（およびモード 2）から約 4.7 の距離にあり、下側 2 つのモード間の距離 5 よりも近い。2 つの平均を結ぶ線分に沿って通し例の $U = -\log p$ を評価し、障壁の高さを読み取って、同様に横断時間を推定せよ。2,000 本の連鎖のうち、モード 1 からモード 3 へ 2,000 ステップ以内に移動するのは何本になると予測されるか。
   3. p_σ に対して $\sigma = 0.3$ と $\sigma = 1$ で最初の設問を繰り返し（ΔU が小さい場合は桁数の見積もりとして扱え）、その結果を用いて $\sigma = 1$ の連鎖の重みを説明せよ。2,000 本の連鎖のうち 2,000 ステップ以内に横断するのは $\sigma = 0.3$ では何本になるはずか。また、出力された `p_0.3` の行はこれと一致しているか。$\sigma = 0.3$ で訓練されたネットワークはその少なくとも 2 倍の質量を移動させている。このことは、モード間における学習された場について何を物語っているか。

17.2 ガウス経路：拡散とフローマッチング

17.1 節で見たように、単一のノイズレベル σ では混合と鮮明な生成を両立できない。σ が小さいとランジュバン連鎖は初期モードに閉じ込められ、逆に σ が大きいとモード間を混合するものの強くぼやけた密度からサンプリングしてしまうからである。解決策は、$t=0$ におけるデータ分布と $t=1$ における単位ガウス分布を連続的な**ガウス経路**（Gaussian path）$\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ で結び、高ノイズ側からノイズゼロへとたどっていくことである。

ノイズ除去拡散確率モデル（denoising diffusion probabilistic model、DDPM）(Ho et al. 2020)、スコアベースモデル（score-based model）(Song and Ermon 2019; Y. Song, Sohl-Dickstein, et al. 2021)、フローマッチング (Lipman et al. 2023; Liu et al. 2023; Albergo et al. 2023) はいずれも、最小二乗回帰（least-squares regression）によって $t \in [0,1]$ の全レベルにわたる単一の時間条件付きネットワークを訓練する。一般的な訓練目標（クリーンな点 $\mathbf{x}_0$、ノイズ ϵ、スコア $\nabla \log p_t(\mathbf{x}_t)$、速度 $\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon$、角速度 $\mathbf{v} = \alpha_t \epsilon - \sigma_t \mathbf{x}_0$）はすべて、$\mathbf{x}_t$ と事後平均（posterior mean）$\hat{\mathbf{x}}_0(\mathbf{x}_t) = \mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t]$ のアフィン関数であるため、いずれか 1 つの目標で訓練したネットワークから残りの目標もすべて定まる。本節では、標準的なガウス経路とそれらが共有する対数信号対雑音比（log signal-to-noise ratio、log-SNR）の時計を定義し、厳密なスコア上でアニーリング付きランジュバンサンプリングを検証する。さらに、条件付きフローマッチングと目標の変換辞書を導出し、3 モード混合分布における厳密な損失の下限と訓練済みネットワークを比較して検証する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.2.1 データからノイズへの経路

すべてのノイズレベルをたどるには、各レベルに時間のラベルを付けて一列に並べる必要がある。

17.2.1.1 信号、ノイズ、対数信号対雑音比

前節で扱ったノイズ付きの点 $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$ は、σ を時間とともに増大させれば 1 つの経路になる。この経路はデータの大きさを保ったまま、データが無視できるほどになるまでノイズを大きくしていく。その分散が際限なく増大するため、**分散爆発**（variance-exploding、VE）経路と呼ばれる。この経路上のネットワークにはノイズとともに大きくなる入力が入るため、1 組の重みで大きさ 1 の入力も、CIFAR-10 画像に対する Song と Ermon のモデルではその 50 倍の大きさの入力も処理しなければならない。そのため大半のモデルは、ノイズが増大するにつれてデータを縮小し、すべての入力を単位サイズ付近に保つ。**ガウス経路**とは、α_t と σ_t という $t \in [0,1]$ 上の関数の組であり、ノイズ付きの点を次の式で定義する。

$$\mathbf{x}_t = \alpha_t\, \mathbf{x}_0 + \sigma_t\, \epsilon, \qquad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.11}$$

この経路は $(\alpha_0, \sigma_0) = (1, 0)$ から出発する。VE 経路とは異なり、よく使われる 2 つの経路は純粋なノイズ $(\alpha_1, \sigma_1) = (0, 1)$、あるいはそれにほぼ等しい状態で終わる。

DDPM (Ho et al. 2020) の**分散保持**（variance-preserving、VP）経路は $\alpha_t^2 + \sigma_t^2 = 1$ を保つため、単位分散のデータは単位分散のまま保たれる。レート $\beta(t) = \beta_{\min} + (\beta_{\max} - \beta_{\min})\, t$ は $\beta_{\min} = 0.1$ から $\beta_{\max} = 20$ まで線形に増加し、データを忘却する速さを定める。微小時間 dt ごとに信号はその分散の割合 $\beta(t)\, dt$ を失うため、残存する割合は積分したレートに負号をつけたものの指数関数となる。

$$\bar{\alpha}_t := \alpha_t^2 = \exp\Big(-\int_0^t \beta(r)\,dr\Big),$$
$$\log \bar{\alpha}_t = -(0.1\,t + 9.95\,t^2).$$

0.1 と 20 という数値は恣意的なものではない。これらは DDPM の最初と最後のステップ分散である 10^{-4} と 0.02 にステップ数 1,000 を掛けたものであり、ステップがどのようにレートになるかは次項で示す。経路の終端では $\bar{\alpha}_1 = e^{-10.05} \approx 4 \cdot 10^{-5}$ となり、実用上は純粋なノイズとみなして差し支えない。時間に沿って追跡すると、VP 経路上の 1 点は原点へと引き戻されるランダムウォークであるオルンシュタイン・ウーレンベック過程（38.4.3 節）を、進みが加速する時計のもとで描く。引き戻す力とノイズの分散はいずれも $\beta(t)$ に比例し、終端では始端の 200 倍になる。

フローマッチングとレクティファイドフロー (Lipman et al. 2023; Liu et al. 2023; Albergo et al. 2023) の**線形**経路は、各クリーンな点からそのノイズへと直線を引く。

$$\alpha_t = 1 - t, \qquad \sigma_t = t. \tag{17.12}$$

これは厳密にノイズで終わる最も単純な経路であり、各データ例はその直線上を一定の速さで進む。この単純さゆえに、本章のデフォルトモデルの経路として、また 2024 年以降の多くの画像生成器の経路として採用されている (Esser et al. 2024)。

2 つの経路が異なる時刻においてノイズ除去器（denoiser）に同じ情報を与えるのはいつだろうか。ノイズ除去器が $\mathbf{x}_t$ から推論できる内容は、**対数信号対雑音比**（log-SNR）$\lambda_t = \log(\alpha_t^2/\sigma_t^2)$ として測られる信号対雑音比のみに依存する。その理由を見るために、ノイズ付きの点に $c = 1/(\alpha_t + \sigma_t)$ を掛けてみよう。$c\alpha_t\mathbf{x}_0 + c\sigma_t\epsilon$ の係数の和は 1 になるため、結果は同じ信号対雑音比を持つ時刻 $\sigma_t/(\alpha_t + \sigma_t)$ における線形経路上の点となる。既知の数を掛けても情報は失われないため、再スケーリングされた点は $\mathbf{x}_t$ と同じ情報を持つ。

したがって、λ が等しい時刻で読み取れば 2 つの経路はクリーンな点について同じ情報を保持しており、対数信号対雑音比はすべての経路が共有する時計となる。この時計はデータにおける $+\infty$ から純粋なノイズにおける $-\infty$ まで進む。線形経路はその全域を網羅し、VP 経路は $\lambda = -10$ 付近で止まる。ゆえに経路を選ぶことは、この時計がどれだけ速く進むかを選ぶことにほかならず、それによって訓練とサンプリングがどの領域に時間を費やすかが決まる。

`GaussianPath` は α_t と σ_t を保持し、対数信号対雑音比を計算して $\mathbf{x}_t$ をサンプリングする。そのコンストラクタは VP 経路、線形経路、および以下で登場する 3 つ目の経路であるコサイン経路を構築する。

```
class GaussianPath:
    """x_t = alpha(t) x0 + sigma(t) eps, data at t = 0 and noise at t = 1."""
    def __init__(self, alpha, sigma):
        self.alpha, self.sigma = alpha, sigma

    @staticmethod
    def vp(beta_min=0.1, beta_max=20.0):
        """Variance preserving, Ho et al.'s linear beta in continuous time."""
        log_abar = lambda t: -(beta_min * t + (beta_max - beta_min) * t**2 / 2)
        return GaussianPath(lambda t: torch.exp(log_abar(t) / 2),
                            lambda t: torch.sqrt(-torch.expm1(log_abar(t))))
```

```
    @staticmethod
    def linear():
        """The straight line of flow matching and rectified flow."""
        return GaussianPath(lambda t: 1 - t, lambda t: t)

    @staticmethod
    def cosine(s=0.008):
        """Variance preserving, Nichol and Dhariwal's cosine schedule."""
        c = lambda t: torch.cos(math.pi / 2 * (t + s) / (1 + s))
        g = lambda t: c(t) / c(torch.zeros_like(t))   # exactly 1 at t = 0
        # and exactly 0 at t = 1 (cos(pi/2) is not), keeping the slope
        f = lambda t: torch.where(t < 1, g(t), g(t) - g(t).detach())
        return GaussianPath(f, lambda t: torch.sqrt(1 - f(t) ** 2))

    def log_snr(self, t):
        """lambda_t = log(alpha_t^2 / sigma_t^2), falling from +inf at 0."""
        return 2 * (torch.log(self.alpha(t)) - torch.log(self.sigma(t)))

    def sample(self, x0, t, eps):
        """x_t for per-example times t of shape (n,)."""
        t = t.reshape(-1, *[1] * (x0.dim() - 1))
        return self.alpha(t) * x0 + self.sigma(t) * eps
```

$t = 0.5$ において線形経路は信号とノイズを等しい割合で含み、$\lambda = 0$ となる。では VP 経路はどこにあるだろうか。

```
vp, linear = GaussianPath.vp(), GaussianPath.linear()
t = torch.tensor([0.1, 0.5, 0.9])
r2 = lambda v: [round(u, 2) for u in v.tolist()]
for name, path in (('vp', vp), ('linear', linear)):
    print(f'{name:>6}: alpha {r2(path.alpha(t))} sigma {r2(path.sigma(t))} '
          f'log-SNR {r2(path.log_snr(t))}')
```

```
    vp: alpha [0.95, 0.28, 0.02] sigma [0.32, 0.96, 1.0] log-SNR [2.16, -2.46, -8.15]
linear: alpha [0.9, 0.5, 0.1] sigma [0.1, 0.5, 0.9] log-SNR [4.39, 0.0, -4.39]
```

時間の中間点において VP 経路はすでに大半がノイズになっており、$\alpha = 0.28$、$\lambda \approx -2.5$ である。また $t = 0.9$ で残っている信号は、線形経路の 10% に対して 2% にすぎない。VP 経路は $\lambda = 0$ を $t = 0.26$ 付近で横切るため、時間の 4 分の 3 をノイズが支配的な領域で過ごす (Figure 17.3)。

小さな画像ではこれは無駄が多い。Nichol and Dhariwal (2021) は、一辺が 32 ピクセルや 64 ピクセルの画像では $t = 1$ よりずっと前に画像が消えてしまうため、DDPM のスケジュールの終盤がほとんど寄与しないことを観察した。同論文の**コサイン経路**は、データ付近での小さなオフセットを除けば $\alpha_t = \cos \frac{\pi t}{2}$ および $\sigma_t = \sin \frac{\pi t}{2}$ と置く。これもやはり VP 経路の一種だが、$\bar{\alpha}_t$ が中間領域でほぼ線形に減少し両端では緩やかに変化するため、画像がすでに消えた領域に費やす時間が少なくなる。

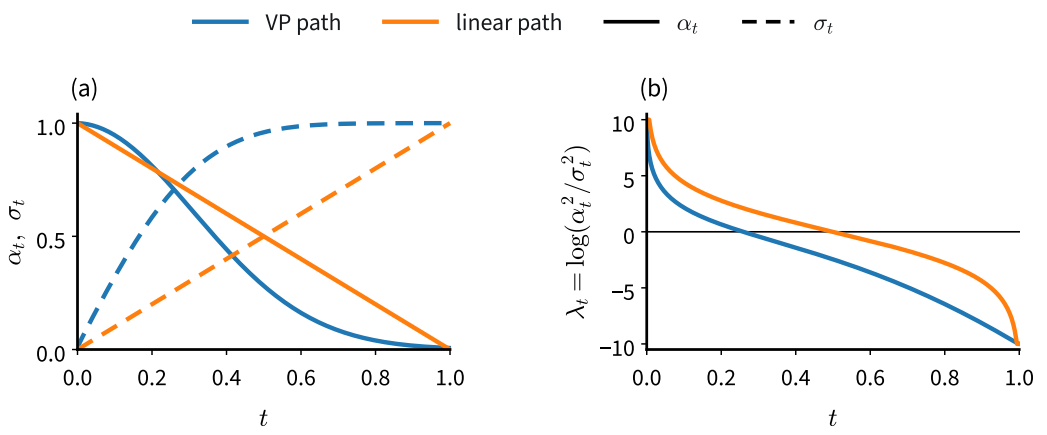


Figure 17.3: 1 つの時計の上の 2 つのガウス経路。(a) 分散保持経路の信号係数 α_t（実線）とノイズ係数 σ_t（破線）（ただし $\beta(t)$ は 0.1 から 20 まで増加）、および線形経路の両係数。(b) 両経路の対数信号対雑音比 λ_t（$[-10, 10]$ にクリップ）。分散保持経路の方がはるかに早くノイズ優勢の領域に達する。

> **!** 時計の規約
>
> 本書全体を通してデータは $t = 0$、ノイズは $t = 1$ に置かれ、すべてのサンプラは $t = 1$ から 0 へと逆向きに走る。フローマッチングの論文では時間の向きが逆になっているため（Figure 38.26）、$1 - t$ をそれらの論文の t に代入し、ノイズ $\mathbf{x}_0$ を本書の ϵ、データ $\mathbf{x}_1$ を本書の $\mathbf{x}_0$ と読み替え、すべての速度の符号を反転させる必要がある。

拡散モデルに関する論文では、同じ量がいくつもの異なる記法で書かれる。Table 17.1 は参照用であり、本章でこれ以降用いる記号と論文での表記をまとめてある。初読時には読み飛ばし、論文の記法に戸惑ったときに参照するとよい。

Table 17.1: 拡散モデルの章における記法。後の節では必要に応じていくつかの記号を追加する。

記号	意味	論文での表記
$t \in [0, 1]$	経路に沿った時刻（0 がデータ、1 がノイズ）	フローマッチングでは $t = 1$ がデータ、DDPM では整数ステップ
$\mathbf{x}_0$、ϵ、$\mathbf{x}_t$	クリーンな点、標準ガウスノイズ、ノイズ付きの点	フローマッチングではデータが $\mathbf{x}_1$、ノイズが $\mathbf{x}_0$
α_t、σ_t	経路の信号係数とノイズ係数	DDPM では $\sqrt{\bar{\alpha}_t}$、$\sqrt{1-\bar{\alpha}_t}$（DDPM の α_t は $1 - \beta_t$）
$\bar{\alpha}_t = \alpha_t^2$、$\beta(t)$	VP 経路の二乗信号係数、そのレート	
k、T、β_k	DDPM 連鎖のステップ、ステップ数、ステップ k で加える分散	t（k の代わり）
λ_t	対数信号対雑音比 $\log(\alpha_t^2/\sigma_t^2)$	VDM では $-\gamma(t)$
σ、$\tilde{\sigma}_i$	VE 経路のノイズレベル、ラダーのレベル i	σ_i（$\tilde{\sigma}_i$ の代わり）
p_t、$\nabla \log p_t$	$\mathbf{x}_t$ の密度、そのスコア	

記号	意味	論文での表記
$\hat{\mathbf{x}}_0$、$\hat{\epsilon}$	$\mathbf{x}_0$ と ϵ の $\mathbf{x}_t$ が与えられた下での事後平均（添字 θ 付きはネットワークによる予測）	D_θ（ノイズ除去器）、ϵ_θ
$\mathbf{u}_t$、$\mathbf{u}_\theta$	経路の周辺速度、それを予測するネットワーク	フローマッチングでは逆符号の u_t、v_t
$\mathbf{v}$	$\alpha_t\epsilon - \sigma_t\mathbf{x}_0$（$\mathbf{v}$ 予測の目標）	
π_k、μ_k、s	混合分布の各成分の重み、平均、標準偏差（ここでの k は成分の添字）	

17.2.1.2 DDPM 連鎖

VP 経路は最初から連続な経路として生まれたわけではない。Ho et al. (2020) は、点を少しずつ縮小して少しずつノイズを加える T 個の微小ステップからなるマルコフ連鎖を次のように定義した。

$$\mathbf{x}_k = \sqrt{1-\beta_k}\,\mathbf{x}_{k-1} + \sqrt{\beta_k}\,\epsilon_k, \qquad k = 1, \dots, T, \tag{17.13}$$

ここで ϵ_k は独立な標準ガウスノイズ、$\beta_k \in (0,1)$ はステップ分散である。各ステップは単位分散を単位分散に写し、ノイズが互いに独立なガウス分布に従うため、k ステップの合成は 1 回のサンプリングにまとめられる。

$$\mathbf{x}_k = \sqrt{\bar{\alpha}_k}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\,\epsilon, \qquad \bar{\alpha}_k = \prod_{j=1}^{k}(1-\beta_j), \tag{17.14}$$

この式において、残存する信号はそこまでの全ステップの因子の積になる（帰納法による証明は 38.6.3.2 節にある）。訓練で連鎖を逐次シミュレーションする必要がないのは、この閉形式があるからである。どの $\mathbf{x}_k$ も、$\mathbf{x}_0$ からガウスノイズを 1 回引くだけで得られる。

各ステップを時刻 $t_k = k/T$ に配置し、$\beta_k = 1 - \bar{\alpha}_{t_k}/\bar{\alpha}_{t_{k-1}}$ と選ぶと、積は相殺されて $\bar{\alpha}_{t_k}$ になり、この連鎖は VP 経路をグリッド上で読み取ったものになる。細かいグリッドでは $\beta_k \approx \beta(t_k)/T$ が成り立つため、ステップ分散とレートは同じ量をそれぞれ 1 ステップあたりと単位時間あたりで数えたものにすぎない。Figure 17.4 に順方向の連鎖と生成時に実行する逆方向の連鎖を示す。

`grid` はノイズからデータまで等間隔に並べた時刻における α と σ を返し、本章のサンプラのデフォルトグリッドとなる。

```
@d2l.add_to_class(GaussianPath)
def grid(self, steps, t_start=1.0, t_end=0.0):
    """Lists of alpha and sigma at steps + 1 even times, t_start to t_end."""
    t = torch.linspace(t_start, t_end, steps + 1, dtype=torch.float64)
    return self.alpha(t).tolist(), self.sigma(t).tolist()
```

$T = 1000$ のとき、この経路はどのようなステップ分散を与えるだろうか。$\beta_k \approx \beta(t_k)/T$ から、最初のステップでは約 10^{-4}、最後のステップでは約 0.02 になると予想される。このグリッドはノイズ側の端から時刻を並べるため、先頭の要素が最後のステップに対応する。

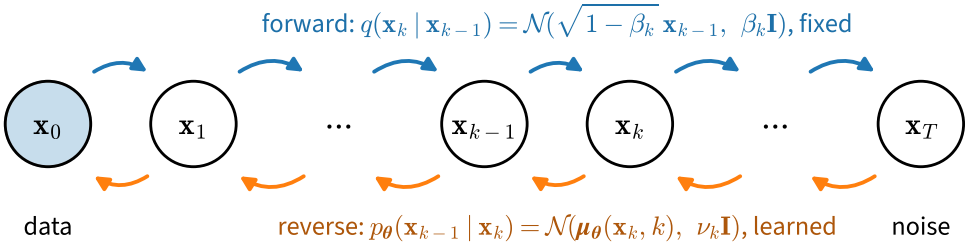


Figure 17.4: 拡散モデルのグラフィカルモデル。順過程（上）はデータ $\mathbf{x}_0$ からノイズ $\mathbf{x}_T$ へのガウス遷移からなる固定されたマルコフ連鎖であり、逆過程（下）はネットワークが平均を予測するガウス遷移からなる学習された連鎖である。

```
a, _ = vp.grid(1000)
print(f'beta_1 = {1 - (a[-2] / a[-1]) ** 2:.1e}, '
      f'beta_T = {1 - (a[0] / a[1]) ** 2:.4f}, alpha_bar_T = {a[0] ** 2:.1e}')
```

```
beta_1 = 1.1e-04, beta_T = 0.0198, alpha_bar_T = 4.3e-05
```

ステップ分散は $1.1 \cdot 10^{-4}$ から 0.0198 まで増加し、Ho らの線形スケジュール（10^{-4} から 0.02）に近い値となる。1,000 ステップ後にはデータの $\sqrt{\bar{\alpha}_T} \approx 0.007$ 倍が残存する。

17.2.1.3 ノイズレベルにわたるアニーリング

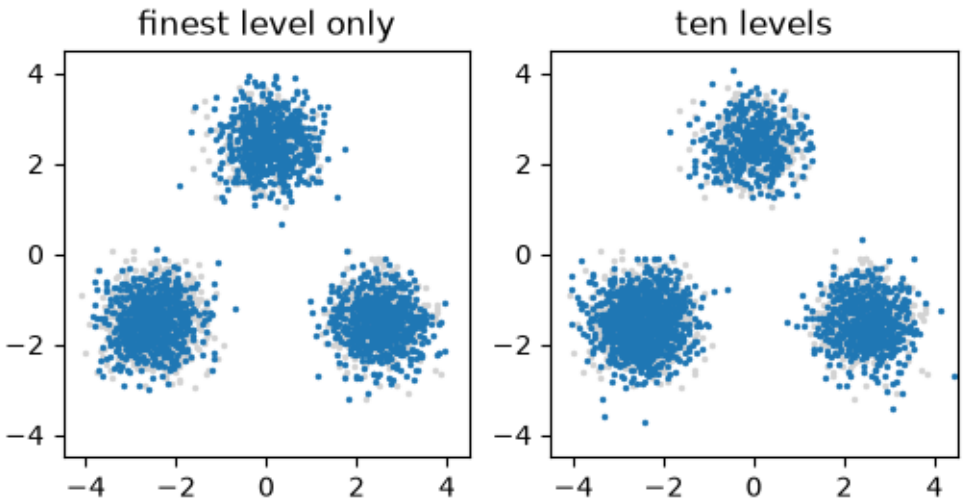
多数のノイズレベルをたどれば、単一レベルではできなかったことが実現できるだろうか。ぼかした混合分布のスコアは厳密にわかっているため、ネットワークを一切訓練する前に、前節の VE 経路を下ることでこれを検証できる。ランジュバン更新 $\mathbf{x} \leftarrow \mathbf{x} + \frac{h}{2}\nabla \log p(\mathbf{x}) + \sqrt{h}\,\xi$（17.1 節）を、毎回新たなガウスノイズ ξ を用いながら、10 個のノイズレベル $\tilde{\sigma}_i$（対数スケールで 3 から 0.05 まで等間隔に配置）からなる**ラダー**に沿って下っていく。各レベルでは前のレベルが終了した点から出発し、厳密なスコア上で 100 ステップ実行する。ステップサイズ $h_i = 0.1\,\tilde{\sigma}_i^2$ はノイズレベルの二乗に比例して縮小するため、各ステップのノイズ $\sqrt{h_i} \approx 0.3\,\tilde{\sigma}_i$ は現在のぼかし幅に対して一定の割合に保たれる。

対照実験では同じ 1,000 ステップを最も細かいレベルだけに費やし、前節と同じステップサイズ $h = 0.05$ を用いる。このステップサイズは同じレベルにおけるラダーのステップより 200 倍大きく、谷を越える機会を対照実験に最大限与えている。どちらも同じ 2,000 個の点（$\mathcal{N}(\mathbf{0}, 9\mathbf{I})$ からのサンプル）から出発する。この考え方が正しければ、ラダーは重み 0.5、0.3、0.2 の近くに到達し、対照実験はそれぞれ約 3 分の 1 になるはずである。

```
torch.manual_seed(0)
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)
x_init = 3 * torch.randn(2000, 2, device=device)
x = x_init
for sig in torch.logspace(math.log10(3), math.log10(0.05), 10).tolist():
    x = d2l.langevin(lambda z: mix.score(z, 1.0, sig), x, 0.1 * sig ** 2, 100)
control = d2l.langevin(lambda z: mix.score(z, 1.0, 0.05), x_init, 0.05, 1000)
print('finest level only:', mix.summary(control))
```

```
print('ten levels:       ', mix.summary(x))
d2l.scatter_panels([control, x], ['finest level only', 'ten levels'],
                   data=mix.sample(2000));
```

```
finest level only: {'weights': [0.339, 0.331, 0.33], 'stray': 0.0145, 'spread': 0.493}
ten levels:        {'weights': [0.496, 0.297, 0.208], 'stray': 0.015, 'spread': 0.516}
```



ラダーは真の重みから約 0.03 以内に収まり、どのモードからも遠く離れたはぐれ点はデータ自体の約 1% に対して 1〜2% にとどまる。一方、対照実験は各モードの周囲で適切な広がりを持つものの、初期点の重みである約 3 分の 1 ずつを保ったままである。モードの比率は平滑化された密度に谷がない粗いレベルで決まり、細かいレベルは各モードを先鋭化するだけである。拡散モデルも経路に沿って同じように役割を分担する。この実験にはネットワークが一切関与していないため、アイデアそのものだけを検証しており、この後で何か問題が生じるとすればそれはネットワークの責任である。

この**アニーリング付きランジュバン動力学**（annealed Langevin dynamics）は、この種の最初の生成器である Song and Ermon (2019) のノイズ条件付きスコアネットワーク（noise-conditional score network）のサンプリングに用いられた。単一のネットワークがノイズレベルを入力として受け取り、ラダーのすべてのレベルにおいて $\tilde{\sigma}_i^2$ で重み付けされたノイズ除去スコアマッチング（denoising score matching）によって訓練される（38.6.3 節）。そのレベル、ステップサイズ、ステップ数は手作業で設定されており、高次元ではその調整が微妙になる（演習問題 4）。経路はこれら 10 個のレベルを連続体に置き換えるものであり、単一のネットワークがそのすべてを学習しなければならない。

17.2.2 周辺場への回帰

単一のネットワークで経路上のすべての点を扱うことを目指す。まずは各点でネットワークが何を学習すべきかを定め、次にその正解を知らないまま学習させる回帰を導出しよう。

17.2.2.1 条件付き目標と周辺場

前節のネットワークは単一のノイズレベルにおいてスコアを学習し、それと同時に Tweedie の公式を通じてクリーンな点の事後平均 $\hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t]$ を学習した。どの時刻においても、この 1 つ

の量がスコアを与え、ノイズ付きの点をどちらへ動かすべきかを教えてくれる。しかしこの量はデータ分布全体に依存するため未知である。そこでクリーンな点でも条件付けてみると、問題は自明になる。$\mathbf{x}_0$ が与えられれば、$\mathbf{x}_0$ の最良の推定値は $\mathbf{x}_0$ そのものだからである。

サンプラが知りたいのはノイズ付きの点をどの方向へ動かすべきかであるため、それをネットワークに直接尋ねてみよう。1 つのクリーンな点と 1 回のノイズのサンプルは、**条件付き速度**（conditional velocity）$\dot{\mathbf{x}}_t = \dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon$（線形経路上では定数 $\epsilon - \mathbf{x}_0$）で移動する。与えられた $\mathbf{x}$ を時刻 t に通過する点は数多くあり、それぞれが固有の方向へ向かうが、点群全体としてはその地点でそれらの平均速度に従って移動する。

$$\mathbf{u}_t(\mathbf{x}) = \mathbb{E}\big[\dot{\alpha}_t \, \mathbf{x}_0 + \dot{\sigma}_t \, \epsilon \,\big|\, \mathbf{x}_t = \mathbf{x}\big], \tag{17.15}$$

これが**周辺速度**（marginal velocity）である。ただ 1 つのクリーンな点しか $\mathbf{x}$ を生成し得ない場所ではその点の速度に一致し、複数の点が生成し得る場所ではそれぞれの起こりやすさで重み付けした平均になる。

混合分布に対しては、この場を厳密に求められる。モードが与えられれば $\mathbf{x}_0$ と ϵ は $\mathbf{x}_t$ と結合ガウス分布に従うため、それぞれの条件付き平均は $\mathbf{x}_t$ の線形関数となり、`velocity` はそれらを各モードの事後確率で重み付けして平均する。

```
@d2l.add_to_class(d2l.GaussianMixture)
def velocity(self, x, t):
    """E[eps - x0 | x_t] on the linear path x_t = (1 - t) x0 + t eps."""
    log_p, r, var, a = self._modes(x, 1 - t, t)
    eps = (1 - a) * r / var                          # E[eps | x_t, mode k]
    x0 = self.means + a * self.std ** 2 / var * r  # E[x0 | x_t, mode k]
    return (torch.softmax(log_p, 1)[..., None] * (eps - x0)).sum(1)
```

実データではこの場を計算できないが、学習することはできる。最小二乗法は条件付き平均を復元するため、条件付き期待値（conditional expectation）はその内部の量へ回帰することで学習できる。ノイズ除去スコアマッチングがスコアを復元したのもこの仕組みによる。速度に対してこの回帰を適用すると**条件付きフローマッチング**（conditional flow matching）損失が得られ、未知の周辺場への回帰とはネットワークに依存しない定数 C だけしか違わない。

$$\begin{aligned}\mathbb{E}_{t,\mathbf{x}_0,\epsilon}\left\|\mathbf{u}_\theta(\mathbf{x}_t, t) - (\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon)\right\|^2 \\ = \mathbb{E}_{t,\mathbf{x}_t}\left\|\mathbf{u}_\theta(\mathbf{x}_t, t) - \mathbf{u}_t(\mathbf{x}_t)\right\|^2 + C.\end{aligned} \tag{17.16}$$

両辺の最小点は一致するため、データ例ごとの速度で訓練したネットワークは周辺速度を学習する（証明は 38.7.1 節にある）。ノイズ除去器がスコア自体を必要としなかったのと同様に、周辺速度を学習するのに $\mathbf{u}_t$ を知る必要は一切ない。Figure 17.5 に混合分布における両方の見方を示す。$\mathbf{x}_t = \mathbf{x}$ を固定すると、線形経路の条件付き速度 $\epsilon - \mathbf{x}_0$ は $(\mathbf{x} - \mathbf{x}_0)/t$ に等しくなり、図ではサンプラが進む方向であるその負号のベクトルを描いている。

`target` は任意の経路上でのデータ例ごとの回帰目標を計算するメソッドであり、以下の訓練ではこれから速度を取得する。`rates` はフォワードモード微分によって導関数 $\dot{\alpha}_t$ と $\dot{\sigma}_t$ を供給するため、経路ごとに導関数を手書きする必要はない。残りの種類の目標と、予測値を $(\hat{\mathbf{x}}_0, \hat{\epsilon})$ へ戻す `x0_eps` については次項で扱う。

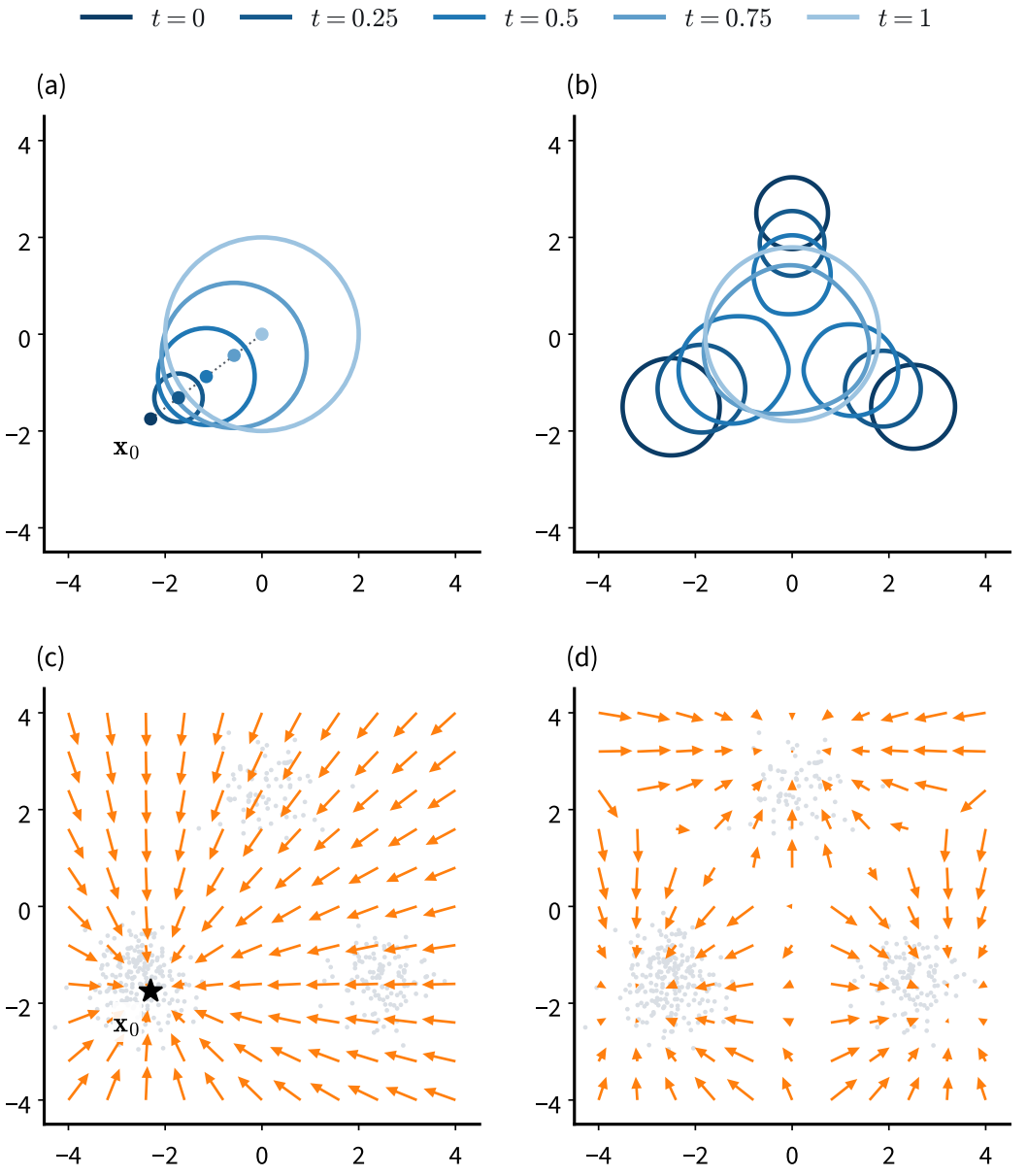


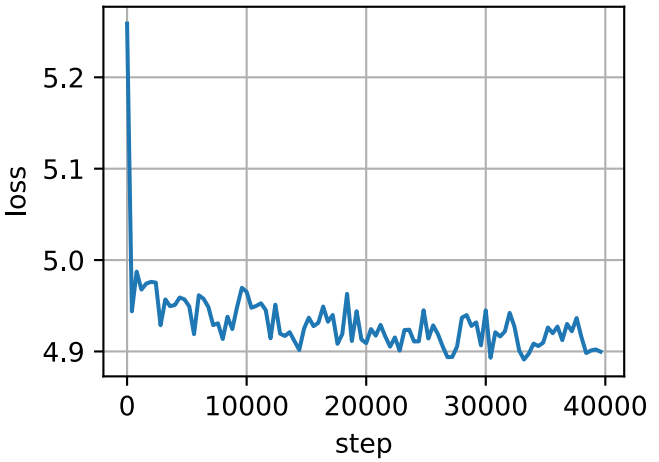
Figure 17.5: 混合分布上の線形経路における条件付きの見方と周辺的な見方（厳密計算）。(a) 最も重みの大きいモードから取った 1 つのクリーンな点 $\mathbf{x}_0$ の条件付き経路（半径 $2\sigma_t$、中心 $\alpha_t \mathbf{x}_0$ の円を 0 から 1 までの 5 つの時刻で表示）。(b) 周辺経路（同じ時刻における p_t の等高線）。(c) 条件付き場 $-(\mathbf{x} - \mathbf{x}_0)/t$（$t = 0.5$ においてその 1 点へ向かう）。(d) 周辺場 $-\mathbf{u}_{0.5}(\mathbf{x})$（条件付き場の事後平均）。

```
@d2l.add_to_class(GaussianPath)
def rates(self, t):
    """The time derivatives (alpha'(t), sigma'(t)), by forward-mode AD."""
    ones = torch.ones_like(t)
    return (torch.func.jvp(self.alpha, (t,), (ones,))[1],
            torch.func.jvp(self.sigma, (t,), (ones,))[1])

@d2l.add_to_class(GaussianPath)
def target(self, kind, x0, eps, t):
    """The regression target of a kind: eps, x0, v, score or velocity."""
    t = t.reshape(-1, *[1] * (x0.dim() - 1))
    a, s = self.alpha(t), self.sigma(t)
    if kind == 'eps': return eps
    if kind == 'x0': return x0
    if kind == 'v': return a * eps - s * x0
    if kind == 'score': return -eps / s
    if kind != 'velocity': raise ValueError(f'unknown kind {kind!r}')
    da, ds = self.rates(t)                      # velocity = d x_t / dt
    return da * x0 + ds * eps

@d2l.add_to_class(GaussianPath)
def x0_eps(self, kind, out, x, t):
    """The (x0, eps) that a prediction `out` of a kind implies at (x, t)."""
    t = t.reshape(-1, *[1] * (x.dim() - 1))
    a, s = self.alpha(t), self.sigma(t)
    if kind == 'eps': return (x - s * out) / a, out
    if kind == 'x0': return out, (x - a * out) / s
    if kind == 'v': return ((a * x - s * out) / (a**2 + s**2),
                            (s * x + a * out) / (a**2 + s**2))
    if kind == 'score': return (x + s ** 2 * out) / a, -s * out
    if kind != 'velocity': raise ValueError(f'unknown kind {kind!r}')
    da, ds = self.rates(t)
    det = a * ds - s * da
    return (ds * x - s * out) / det, (a * out - da * x) / det
```

これで手法全体が 1 つのセルに収まる。各ステップでは混合分布から新たに 512 個の点を引き、それぞれに時刻とノイズを割り当てて、点と時間の正弦波特徴量を入力とする前節の小さなネットワーク `TimeMLP` を線形経路の条件付き速度へ回帰させる。訓練ではコサイン減衰する学習率を用いて Adam を 40,000 ステップ実行し、半減期 700 ステップで重みの移動平均を保持する。平均化された重みは終盤のステップのノイズを平滑化するものであり、以下ではこれを評価に用いる。前節の固定された 2,000 個の点からなる訓練セットとは異なり、バッチが毎回新しく生成されるため暗記すべき訓練セットが存在しない（訓練セットが固定されている場合に何が起きるかは演習問題 5 で扱う）。各ネットワークは 1 基の GPU 上で 1〜2 分で訓練できる。

```
def regression_loss(path, kind):
    """Squared error of net(x_t, t) against the target of a kind, t ~ U(0, 1)."""
    def loss_fn(net, x0):
        t = torch.rand(len(x0), device=x0.device)
        eps = torch.randn_like(x0)
        error = net(path.sample(x0, t, eps), t) - path.target(kind, x0, eps, t)
        return (error ** 2).flatten(1).sum(1).mean()  # ||error||^2 per example
```

```
    return loss_fn

torch.manual_seed(1)
num_steps = 40_000
net_v, ema_v, losses_v = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), regression_loss(linear, 'velocity'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
d2l.plot(torch.arange(100) * num_steps // 100,
         losses_v.reshape(100, -1).mean(1).cpu(), 'step', 'loss')
```



曲線の各点は 400 ステップの平均である。損失は数千ステップのうちにゼロよりはるかに高い 5 付近で頭打ちになる。ネットワークは行き詰まってしまったのだろうか。実は、どのようなネットワークであってもこれより大幅に良くすることはできないことがすぐにわかる。

17.2.2.2 等価な目標

DDPM のネットワークはノイズ予測を行い、他のモデルにはクリーンな点を予測するものもある。この選択によってネットワークが学習するものは変わるのだろうか。固定された $\mathbf{x}_t$ においては、これらは異なる量ではない。Equation 17.11 の両辺の事後平均をとると $\mathbf{x}_t = \alpha_t \hat{\mathbf{x}}_0 + \sigma_t \hat{\epsilon}$（ただし $\hat{\epsilon} = \mathbb{E}[\epsilon \mid \mathbf{x}_t]$）が得られるため、一方が決まれば他方も定まる。

スコアは同じ対象の 3 つ目の見方である。前節の Tweedie の公式によれば、$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \tilde{\mathbf{x}} + \sigma^2 \nabla \log p_\sigma(\tilde{\mathbf{x}})$ が $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$ に対して成り立つ。クリーンな点の役割に $\alpha_t \mathbf{x}_0$ を、ノイズレベルに σ_t を当てはめると $\alpha_t \hat{\mathbf{x}}_0 = \mathbf{x}_t + \sigma_t^2 \nabla \log p_t(\mathbf{x}_t)$ となり、スコアは次の式で表される。

$$\nabla \log p_t(\mathbf{x}_t) = \frac{\alpha_t \hat{\mathbf{x}}_0 - \mathbf{x}_t}{\sigma_t^2} = -\frac{\hat{\epsilon}}{\sigma_t}. \tag{17.17}$$

スコアは $\mathbf{x}_t$ からスケーリングされた事後平均へ向かうベクトルであり、ノイズ予測を $-\sigma_t$ で割ったものに等しい。

周辺速度 Equation 17.15 は $\dot{\alpha}_t \hat{\mathbf{x}}_0 + \dot{\sigma}_t \hat{\epsilon}$ であり、これとスコアの式から $\hat{\mathbf{x}}_0$ を消去すると、次のスコア・速度恒等式が得られる。

$$\mathbf{u}_t(\mathbf{x}) = \frac{\dot{\alpha}_t}{\alpha_t}\,\mathbf{x} - \left(\sigma_t\dot{\sigma}_t - \sigma_t^2\frac{\dot{\alpha}_t}{\alpha_t}\right)\nabla \log p_t(\mathbf{x}). \tag{17.18}$$

$\dot{\alpha}_t = -1$ かつ $\dot{\sigma}_t = 1$ である線形経路上では、この式は次のようになる。

$$\mathbf{u}_t(\mathbf{x}) = \hat{\epsilon} - \hat{\mathbf{x}}_0 = -\frac{\mathbf{x} + t\,\nabla \log p_t(\mathbf{x})}{1-t}. \tag{17.19}$$

速度は点の既知の定数倍とスコアの既知の定数倍の和であり、どちらの係数も経路によって定まる。すなわち、速度モデルは姿を変えたスコアモデルにほかならない。

多くの画像モデルでは 5 つ目の目標として **v 予測** (Salimans and Ho 2022)（$\mathbf{v} = \alpha_t\epsilon - \sigma_t\mathbf{x}_0$）が用いられる。VP 経路上では $\alpha_t = \cos\phi_t$、$\sigma_t = \sin\phi_t$ と書けるため、$\mathbf{v} = d\mathbf{x}_t/d\phi_t$ は単位時間あたりではなく単位角度あたりの速度となる。小さなオフセットを除いたコサイン経路上では $\phi_t = \pi t/2$ であり、$\mathbf{v}$ は速度の $2/\pi$ 倍になる（演習問題 1）。Salimans と Ho は、主に低い信号対雑音比でネットワークにクエリを送る極少ステップのサンプラ向けにこれを導入した。そのような領域では、ノイズ予測からクリーンな点を得るには小さな α_t で割る必要があるのに対し、$\mathbf{v}$ からは除算なしで求められる。

5 つの目標はすべて $\mathbf{x}_t$ と 1 つの事後平均のアフィン関数であり、その係数は経路と時刻のみに依存する。この厳密な事後平均を、目標が何であれ完璧なネットワークが学習するはずの予測器という意味で**オラクル**と呼び、目標間の変換を**辞書**（dictionary）と呼ぶ。Table 38.2 にこの辞書の全体を示す。`target` は $(\mathbf{x}_0, \epsilon)$ から任意の種類の目標へと順方向に辞書を引き、そのそばで定義した `x0_eps` は任意の種類の予測値から $(\hat{\mathbf{x}}_0, \hat{\epsilon})$ へと逆方向に辞書を引く。したがって、1 つの目標で訓練されたネットワークは残りの目標をすべて定め、別の経路上で訓練されたネットワークも対数信号対雑音比が等しい時刻で読み取れば同様にすべてを定める。

1 つの点でこの辞書の働きを確かめてみよう。混合分布に対して `oracle` は厳密な $\hat{\mathbf{x}}_0$ と $\hat{\epsilon}$ を返すため、どの損失やサンプラでもネットワークの代わりに利用できる。VP 経路上の点 $(0.5, -0.5)$、時刻 $t = 0.5$（$\alpha = 0.28$、$\sigma = 0.96$）において、すべての目標を表示する。最後の行では点を $1/(\alpha+\sigma)$ 倍に再スケーリングして対数信号対雑音比が等しい線形経路上の時刻 $t = 0.773$ に移し、異なる点における異なる公式である厳密な場 `velocity` を評価する。この結果は、最初の 2 行からあらかじめ計算できる $\hat{\epsilon} - \hat{\mathbf{x}}_0$ と一致するはずである。

```
def oracle(x, alpha, sigma):
    """The exact posterior means (x0_hat, eps_hat) of the mixture at x."""
    x0_hat = mix.posterior_mean(x, alpha, sigma)
    a, s = (torch.as_tensor(v, device=x.device).reshape(-1, 1)
            for v in (alpha, sigma))
    return x0_hat, (x - a * x0_hat) / s

x = torch.tensor([[0.5, -0.5]], device=device)   # one noisy point
t = torch.tensor([0.5], device=device)           # VP: alpha 0.28, sigma 0.96
x0_hat, eps_hat = oracle(x, vp.alpha(t), vp.sigma(t))
for kind in ('x0', 'eps', 'score', 'v', 'velocity'):
    print(f'{kind:>8} on vp:', r2(vp.target(kind, x0_hat, eps_hat, t)[0]))
c = 1 / (vp.alpha(t) + vp.sigma(t))     # linear path, same alpha / sigma
print(f'velocity on linear at t = {(c * vp.sigma(t)).item():.3f}:',
      r2(mix.velocity(c * x, c * vp.sigma(t))[0]),
      ' eps_hat - x0_hat:', r2((eps_hat - x0_hat)[0]))
```

```
      x0 on vp: [0.29, -0.98]
     eps on vp: [0.44, -0.23]
   score on vp: [-0.46, 0.24]
       v on vp: [-0.15, 0.88]
velocity on vp: [-0.22, 1.29]
velocity on linear at t = 0.773: [0.15, 0.75]  eps_hat - x0_hat: [0.15, 0.75]
```

クリーンな点の予測値は $(0.29, -0.98)$ となり、3 つの平均 $(-2.5, -1.5)$、$(2.5, -1.5)$、$(0, 2.5)$ がなす三角形の内部にあっていずれの平均からも遠い。これは $\alpha = 0.28$ では事後分布がまだ 3 つのモードすべてに広がっているためである。スコアはノイズ予測を -0.96 で割った値になっており、線形経路の場は $(0.15, 0.75) = \hat{\epsilon} - \hat{\mathbf{x}}_0$ を返している。2 つの経路上の 5 つの目標が、ただ 1 つの事後平均から導かれている。

では、どの目標を訓練すべきだろうか。オラクルに到達できるネットワークであればどれを選んでも変わらないが、実際のネットワークはオラクルに到達しないため、2 つの実務上の事実が決め手になる。第 1 に、純粋なノイズにおいては $\hat{\mathbf{x}}_0 = (\mathbf{x}_t - \sigma_t\hat{\epsilon})/\alpha_t$ がゼロ除算になるため、ノイズ予測器はクリーンな点について何も語れない。ゆえにノイズ予測器は、VP 経路のように α_t が正に保たれる経路と組み合わせるべきである。第 2 に、目標によって時刻ごとの重み付けが異なる。線形経路上では、速度損失はノイズ損失に $1/(1-t)^2$ の重みを掛けたものに等しく、またクリーンな点の損失に $1/t^2$ の重みを掛けたものに等しい（演習問題 2）。そのため、速度の二乗誤差を一様に小さくすることは、ノイズ端の近くではノイズの誤差を小さくし、データの近くではクリーンな点の誤差を小さくすることを意味する。実用的には、線形経路上の速度、コサイン経路上の $\mathbf{v}$、VP 経路上のノイズという 3 つの組み合わせが主流である。いずれも誤りではなく、辞書によって相互に変換できる。

DDPM の**単純損失**（simple loss）(Ho et al. 2020) はノイズへ回帰する。

$$L_{\text{simple}}(\theta) = \mathbb{E}_{t,\mathbf{x}_0,\epsilon}\left\|\epsilon - \epsilon_\theta(\alpha_t\mathbf{x}_0 + \sigma_t\epsilon,\, t)\right\|^2. \tag{17.20}$$

データ例ごとのスコア、すなわち条件付き密度 $\mathcal{N}(\alpha_t\mathbf{x}_0, \sigma_t^2\mathbf{I})$（$\mathbf{x}_t$ の $\mathbf{x}_0$ が与えられた下での分布）のスコアは $-\epsilon/\sigma_t$ であるため、単純損失はすべてのレベルにおいて σ_t^2 で重み付けされたノイズ除去スコアマッチングに等しい（38.6.3.2 節）。Ho らは尤度の変分下界からこの損失を導出したのち、t にわたる下界の重みを取り除いた方がサンプル品質が高くなることを見いだして重みを落とした。2 つ目のモデルでは DDPM の選択、すなわち VP 経路上のノイズ目標を採用し、それ以外は一切変更しない。

```
torch.manual_seed(2)
net_eps, ema_eps, losses_eps = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), regression_loss(vp, 'eps'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
```

両方のネットワークが十分に訓練できていれば、経路や目標が異なっていても同じ事後平均を保持しているはずだと辞書からわかる。これを損失の下限によって確かめよう。

17.2.2.3 損失の下限

なぜ速度の損失は 5 付近で頭打ちになったのだろうか。各目標にはそのデータ例固有のノイズが含まれており、これは $\mathbf{x}_t$ からはどのネットワークにも予測できない。オラクルでさえ、経路全体で平均した目標の条件付き分散（conditional variance）を残してしまう。前節で単一のノイズレベルにおいて登場したこの**損失の下限**（loss floor）こそ、定数 C（Equation 17.16）の正体で

ある。$\mathbf{x}_t$ を固定するとノイズは $(\mathbf{x}_t - \alpha_t \mathbf{x}_0)/\sigma_t$ となるため、どの目標も $c_t \mathbf{x}_0$ と $\mathbf{x}_t$ の関数の和になり、時刻 t におけるその下限は次の式で与えられる。

$$c_t^2 \, \mathbb{E} \operatorname{tr} \operatorname{Cov}[\mathbf{x}_0 \mid \mathbf{x}_t].$$

クリーンな点では $c_t = 1$、ノイズでは $c_t = -\alpha_t/\sigma_t$、線形経路の速度では $c_t = -1/t$ である。各目標の違いは、同一の事後分散（posterior variance）をどれだけ拡大するかという点にしかない。

データの近くではこの数値を解析的に見積もることができる。$t = 0.1$ ではノイズに比べてモード間の距離がまだ十分に離れているため、事後分布は単一のガウスモードの事後分布となり、1 座標あたりの分散は $s^2\sigma_t^2/(\alpha_t^2 s^2 + \sigma_t^2)$ となる。これに c_t^2 を掛けて 2 つの座標について足し合わせると、ノイズの下限 $2\alpha_t^2 s^2/(\alpha_t^2 s^2 + \sigma_t^2)$ と、線形経路における速度の下限 $2s^2/(\alpha_t^2 s^2 + \sigma_t^2)$（ただし $\sigma_t = t$）が得られる。$s = 0.5$ のとき、VP 経路（$\alpha = 0.95$、$\sigma = 0.32$）のノイズの下限は約 1.37、線形経路（$\alpha = 0.9$、$\sigma = 0.1$）の速度の下限は 2.35 となる。

次のセルでは 100 万個のサンプルを用いて、3 つの時刻および t 全体の平均における各下限とその標準誤差を測定する。オラクルの誤差は $\mathbf{x}_t$ のあらゆる関数と無相関であるため、任意のネットワークの損失は下限と、ネットワークの予測およびオラクルの予測の平均二乗誤差の和に分解される。そこでこのセルでは、平均化ネットワークの下限に対する過剰損失をその距離として直接測定し（`excess` 列）、最後の $1{,}000$ ステップの訓練損失（`train loss` 列）と並べて表示する。最後の行では速度ネットワークを VP 経路上のノイズ予測器として読み取る。すなわち、`eps_from_velocity` が各点を対数信号対雑音比の等しい線形経路上の点へ移し、そこで速度ネットワークにクエリを送り、その出力を $\hat{\epsilon}$ へと `x0_eps` で変換する。

```
@torch.no_grad()
def floor_and_excess(path, kind, net, x0, eps, t):
    """The floor E||target - E[target | x_t]||^2 with its standard error, and
    the network's loss minus the floor, E||net(x_t, t) - E[target | x_t]||^2."""
    x = path.sample(x0, t, eps)
    best = path.target(kind, *oracle(x, path.alpha(t), path.sigma(t)), t)
    err = ((path.target(kind, x0, eps, t) - best) ** 2).sum(1)
    out = torch.cat([net(x[i:i + 100_000], t[i:i + 100_000])  # in chunks
                     for i in range(0, len(x), 100_000)])
    return (err.mean().item(), err.std().item() / math.sqrt(len(err)),
            ((out - best) ** 2).sum(1).mean().item())

def eps_from_velocity(x, t):
    """ema_v as a noise predictor on the VP path, read at the linear-path
    time with the same alpha / sigma (the same posterior mean)."""
    c = 1 / (vp.alpha(t) + vp.sigma(t))
    t_lin, x_lin = c * vp.sigma(t), c[:, None] * x
    return linear.x0_eps('velocity', ema_v(x_lin, t_lin), x_lin, t_lin)[1]

n = 1_000_000
x0, eps = mix.sample(n), torch.randn(n, 2, device=device)
times = [torch.full((n,), t, device=device) for t in (0.1, 0.5, 0.9)]
times.append(torch.rand(n, device=device))             # t ~ U(0, 1)
print('target, path      t=0.1  t=0.5  t=0.9  t~U(0,1)    s.e.'
      '  train loss   excess')
for name, path, kind, net, losses in (
        ('noise, vp', vp, 'eps', ema_eps, losses_eps),
```

```
          ('velocity, linear', linear, 'velocity', ema_v, losses_v)):
      rows = [floor_and_excess(path, kind, net, x0, eps, t) for t in times]
      print(f'{name:16}' + ''.join(f'{r[0]:7.3f}' for r in rows[:3])
            + f'{rows[3][0]:10.3f}{rows[3][1]:8.4f}'
            + f'{losses[-1000:].mean().item():12.3f}{rows[3][2]:9.4f}')
floor, _, gap = floor_and_excess(vp, 'eps', eps_from_velocity, x0, eps,
                                 times[3])
print(f'velocity net read as a noise model on vp: loss {floor + gap:.3f},'
      f' excess {gap:.4f}')
```

```
target, path      t=0.1  t=0.5  t=0.9  t~U(0,1)    s.e.  train loss   excess
noise, vp         1.368  0.477  0.002     0.540  0.0011       0.542   0.0002
velocity, linear  2.351  2.796  9.157     4.906  0.0070       4.899   0.0010
velocity net read as a noise model on vp: loss 0.540, excess 0.0002
```

$t = 0.1$ における下限は、単一モードによる解析的な見積もりと 0.01 以内で一致している。経路に沿って 2 つの下限は逆方向に動く。ノイズの下限は $t = 0$ 付近（小さなノイズがモード内分散と区別できない領域）で最大となり、0.002（$t = 0.9$）まで下がるのに対し、速度の下限は ≈ 9（$t = 0.9$）まで上昇してから $\operatorname{tr}\operatorname{Cov}[\mathbf{x}_0]$（$t = 1$）へと落ち着く（演習問題 3）。$t \sim U(0,1)$ で平均すると、下限には 1 桁の差（ノイズの 0.54 に対して速度は 4.9）が生じる。これは目標のスケーリング c_t^2 と 2 つの経路の時計の違いによるものである。

訓練された両方のネットワークはそれぞれの理論的な下限に到達している。厳密なオラクルに対する平均二乗誤差（excess）はノイズネットワークで 0.0002、速度ネットワークで 0.001 であり、いずれも下限自体のモンテカルロ標準誤差を十分に下回る。さらに、線形経路の速度ネットワークに `eps_from_velocity` を介して VP ノイズ予測器としてクエリを送ると、VP ノイズ目標で直接訓練したネットワークとまったく同じ損失（0.540）および過剰損失（0.0002）を達成する。両ネットワークは背後にある同一の事後平均を学習して収束しているのである（生のスコア目標 $-\epsilon/\sigma_t$ はノイズの下限を σ_t^2 で割るため $t \to 0$ で発散する。スコアネットワークがノイズ除去スコアマッチングの目的関数に σ_t^2 の重みを掛けてノイズ予測に帰着させるのはこのためである）。こうして学習した場からサンプルを生成する方法が 17.3 節の主題となる。

17.2.3 まとめ

ガウス経路 $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ は、$t = 0$ におけるクリーンなデータと $t = 1$ におけるガウスノイズの間を補間し、そのような経路はすべて信号対雑音比の対数である対数信号対雑音比 $\lambda_t = \log(\alpha_t^2/\sigma_t^2)$ によって与えられる標準的な時計を共有する。ノイズレベルにわたるアニーリングは単一レベルのジレンマを解消する。高ノイズレベルが平滑化された谷を越えて大域的なモード重みを確立し、低ノイズレベルが各モードを先鋭化するからである。データ例ごとの条件付き目標（$\mathbf{x}_0$、ϵ、$-\epsilon/\sigma_t$、$\mathbf{v}$、または $\dot\alpha_t \mathbf{x}_0 + \dot\sigma_t \epsilon$）へ回帰することで、$\mathbf{x}_t$ が与えられた下で対応する条件付き期待値を出力するようにネットワークを訓練できる。5 つの目標はすべて $\mathbf{x}_t$ と事後平均 $\hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t]$ のアフィン関数であるため、単一の訓練済み予測器から任意のガウス経路上の 5 つの目標がすべて定まる。最後に、データ例ごとの回帰目的関数はいずれも観測できないノイズの実現値に起因する正の損失の下限 $c_t^2\, \mathbb{E}\operatorname{tr}\operatorname{Cov}[\mathbf{x}_0 \mid \mathbf{x}_t]$ を持つため、予測器の精度は生の訓練損失ではなく、この下限を上回る過剰損失によって測られる。

17.2.4 演習問題

1. **$\mathbf{v}$ から速度へ。** VP 経路上の点 $\mathbf{x} = (0.5, -0.5)$、時刻 $t = 0.5$（$\alpha_t = 0.28$、$\sigma_t = 0.96$）において、本文では厳密な予測値 $\hat{\mathbf{v}} = (-0.15, 0.88)$ が表示された。

1. VP 経路上では $\alpha_t^2 + \sigma_t^2 = 1$ が成り立つ。$\hat{\mathbf{x}}_0 = \alpha_t \mathbf{x} - \sigma_t \hat{\mathbf{v}}$ および $\hat{\epsilon} = \sigma_t \mathbf{x} + \alpha_t \hat{\mathbf{v}}$ を示し、手計算によって本文で表示された $\hat{\mathbf{x}}_0$ と $\hat{\epsilon}$ を復元せよ。入力を 2 桁に丸めているため約 0.01 の誤差が生じる。
2. $\alpha_t = \cos\phi_t$、$\sigma_t = \sin\phi_t$ と書くとき、周辺速度が $\mathbf{u}_t = \dot{\phi}_t\, \mathbf{v}$ となり、したがって $\mathbf{v}$ が単位角度あたりの速度であることを示し、コサイン経路における係数 $2/\pi$ を導出せよ。
3. 本節の VP 経路に対して $\dot{\phi}_t = \beta(t)\,\alpha_t/(2\sigma_t)$ を示せ。これを $t = 0.5$ で評価し、最初の小問の丸め誤差の範囲内で、表示された速度 $(-0.22, 1.29)$ を $\hat{\mathbf{v}}$ から復元せよ。

2. **速度・ノイズ・クリーンな点の回帰目標。** 線形経路上で $\mathbf{x}_t = (1-t)\mathbf{x}_0 + t\epsilon$ の事後平均をとる。
1. 周辺速度が $(\mathbf{x} - \hat{\mathbf{x}}_0)/t$ と $(\hat{\epsilon} - \mathbf{x})/(1-t)$ の両方に等しいことを示せ。
2. 最初の小問より、$\hat{\mathbf{x}}_0$ の誤差は t で割られて速度に伝わり、$\hat{\epsilon}$ の誤差は $1-t$ で割られて伝わる。それぞれの予測が速度の算出元として不適切になるのは経路のどちらの端の近くか。また、このことからノイズモデルが $\alpha = 0$ で決してクエリされない理由をどのように説明できるか（コサイン経路上におけるすべての目標間の誤差伝播については 6（38.7 節）で扱う）。
3. $\mathbf{u}_\theta = (\mathbf{x} - \hat{\mathbf{x}}_{0,\theta})/t$ の速度損失がクリーンな点の損失 $\mathbb{E}\|\hat{\mathbf{x}}_{0,\theta} - \mathbf{x}_0\|^2$ に重み $1/t^2$ を掛けたものに等しく、$\mathbf{u}_\theta = (\hat{\epsilon}_\theta - \mathbf{x})/(1-t)$ の速度損失がノイズ損失に重み $1/(1-t)^2$ を掛けたものに等しいことを示せ。重みなしの 2 つの損失と比べたとき、フローマッチングはそれぞれどの時刻を重視しているか。

3. [code] **損失の下限。** `oracle` と `floor_and_excess` を用いよ。
1. 線形経路上の速度目標の下限と VP 経路上のノイズ目標の下限を、t に対して区間 $(0,1)$ 上のグリッドでプロットせよ。それぞれどこで最大になるか。表示された訓練損失は 1,000 個のミニバッチ（各 512 点）の平均である。その標準誤差を `losses_v` から推定し、表示された訓練損失と下限の差と比較せよ。
2. Σ を混合分布の共分散行列とし、その固有値を κ_1, κ_2 とする。この共分散を持つガウスデータに対して、線形経路上の時刻 t における速度の下限が $\sum_i \kappa_i / \big((1-t)^2\kappa_i + t^2\big)$ になることを示せ。この曲線をプロットに追加し、混合分布の下限を下回り得ない理由を説明せよ。2 つの下限が経路の両端で一致するのはなぜか。
3. スコア目標 $-\epsilon/\sigma_t$ の下限は、ノイズ目標の下限を σ_t^2 で割ったものになる。これを VP 経路上の $t = 0.01$、0.1、0.5 で評価し、t を $(0,1)$ 上の一様分布としてこの目標でネットワークを訓練しない理由を説明せよ。
4. ノイズモデルの訓練損失は速度モデルの約 10 分の 1 である。これはノイズモデルの方が優れていることを意味するか。VP 経路の時刻 $t \in \{0.1, 0.5, 0.9\}$ において、オラクルに対する各ネットワークの誤差を共通の単位 $\mathbb{E}\|\hat{\mathbf{x}}_0 - \hat{\mathbf{x}}_0^\star\|^2$（ただし $\hat{\mathbf{x}}_0^\star$ は `oracle` の出力）で測定せよ。`ema_eps` は `vp.x0_eps` で変換し、`ema_v` は `eps_from_velocity` に続けて $\hat{\mathbf{x}}_0 = (\mathbf{x}_t - \sigma_t\hat{\epsilon})/\alpha_t$ を適用して変換せよ。本文の係数 c_t を用いて、生の損失がこれほど大きく異なる理由を説明せよ。2 つのモデルを正しくランク付けできるのは、生の損失、下限に対する過剰損失、共通単位での誤差のうちどれか。

小問 (d) は CMU 10-799 の Homework 2（2026 年春学期）問 3(a) を改変したものである。

4. **高次元におけるラダーの間隔。** ●データが $\mathbb{R}^d$ の原点にある単一の点であり、したがって $p_{\tilde{\sigma}} = \mathcal{N}(\mathbf{0}, \tilde{\sigma}^2\mathbf{I})$ であるとし、ラダーが公比 $q = \tilde{\sigma}_{i-1}/\tilde{\sigma}_i > 1$ の等比列であるとする。
1. $r = \|\mathbf{x}\|$（$\mathbf{x} \sim p_{\tilde{\sigma}}$）が $\mathbb{E}[r^2] = d\tilde{\sigma}^2$ および $\mathrm{Var}[r^2] = 2d\tilde{\sigma}^4$ を満たすことを示し、r が近似的に $\mathcal{N}(\tilde{\sigma}\sqrt{d}, \tilde{\sigma}^2/2)$（$d$ が大きいとき）に従うことを導け。
2. レベル $i-1$ を $p_{\tilde{\sigma}_{i-1}}$ のサンプルとともに離れた連鎖は、レベル i に入るとき高い確率で区間 $\tilde{\sigma}_{i-1}(\sqrt{d} \pm 3/\sqrt{2})$ 内の半径にある。$p_{\tilde{\sigma}_i}$ がこの区間に割り当てる確率質量が $\Phi\big(\sqrt{2d}(q-1) + 3q\big) - \Phi\big(\sqrt{2d}(q-1) - 3q\big)$ であることを示せ（ただし Φ は標準正規分布の累積分布関数である）。この確率質量がゼロに近いと、ネットワークが訓練入力をほとんど見たことのない領域から連鎖がそのレベルを開始することになるの

はなぜか。
   3. Y. Song and Ermon (2020) はこの確率質量が約 2 分の 1 になるように q を選んでいる。q を CIFAR-10 画像の次元 $d = 3072$ において求め、$\tilde{\sigma} = 50$ から 0.01 までのレベル数を数えよ。これを同論文の CIFAR-10 モデルにおける 232 レベルと比較せよ。
   4. 最上位のレベルはモードを融合させるのに十分な粗さも備えていなければならない。$\pm\mu$ に位置し分散 v を持つ 2 つの等しいガウス成分が単一のモードを持つための必要十分条件は $\mu^2 \le v$ である（4（38.5 節））。この基準を混合分布の下側 2 つのモードに適用して両者が融合するノイズレベルを見積もり、ラダーの最上位レベルである 3 と比較せよ。
5. [code] **厳密な最適解は暗記する**。有限個の訓練セットはそれ自体が 1 つの分布、すなわち点質量の混合分布であり、その厳密な速度場は既知である。Bertrand et al. (2025) は、ノイズからこの場をたどると訓練点がそのまま再現されることを出発点とし、それにもかかわらず訓練されたネットワークがなぜ汎化するのかを問うている。
   1. 混合分布から 64 個の点を引き、それらの点のみから抽出したバッチ上で `TimeMLP(256, 3)` を `regression_loss(linear, 'velocity')` により訓練せよ。
   2. `d2l.GaussianMixture(points.tolist(), [1/64] * 64, 0.0)` は経験分布を表し（`device` も渡すこと）、その `velocity` メソッドは $t > 0$ における厳密な場を与える。標準ガウスノイズから引いた同じ 2,000 個の初期点から、この厳密な場と訓練済みネットワークの双方を、100 回の直線ステップ $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{u}_t(\mathbf{x})/100$（$t = 1, 0.99, \dots, 0.01$）でたどれ。両方のサンプル集合について、最近傍の訓練点までの距離のヒストグラムをプロットせよ。
   3. 訓練セットをそのまま再現するのはどちらの場で、新しい点を生成するのはどちらか。Bertrand らは、ネットワークを汎化させているのは回帰目標に含まれるノイズではないと論じている。訓練点を 1,024 個に増やして最初の 2 つの小問を繰り返し、2 つのヒストグラムがどのように変化するか述べよ。

17.3 積分によるサンプリング

ガウス経路（Gaussian path）に沿って訓練されたネットワークは、すべてのノイズレベル $t \in [0, 1]$ において周辺速度（marginal velocity）$\mathbf{u}_t(\mathbf{x})$（あるいは同値なスコア $\nabla \log p_t(\mathbf{x})$）を予測する。この決定論的なベクトル場を $\mathbf{x}_1 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$（$t = 1$）から $t = 0$ まで逆向きに積分すると、なぜ p_0 に従うサンプルが、固定レベルのランジュバン連鎖が見逃した正確なモード重み $0.5, 0.3, 0.2$ を伴って生成されるのだろうか。

その保証は、確率輸送の**連続の式**（continuity equation）からもたらされる。各条件付きガウス分布の雲 $p_t(\mathbf{x} \mid \mathbf{x}_0)$ はそれぞれの条件付き速度（conditional velocity）のもとで質量保存則を満たすため、$p(\mathbf{x}_0)$ について平均をとれば、周辺密度 p_t が事後平均速度 $\mathbf{u}_t(\mathbf{x})$ によって輸送されることがわかる。Y. Song, Sohl-Dickstein, et al. (2021) はこの**確率フロー ODE**（probability-flow ODE）による決定論的サンプリングを定式化し、一方で Ho et al. (2020) は確率的な逆マルコフ連鎖によってサンプリングした。後に示すように、DDPM の各祖先ステップは、確率フロー ODE のオイラーステップと現在のノイズレベルにおけるランジュバンステップの和へと一次近似で分解される。本節では、これら両方のサンプラを導出し、経路に依存しない $(\bar{\mathbf{x}}, \bar{\sigma})$ 座標系で ODE を表したうえで、3 モード混合分布において訓練済みネットワークを厳密なオラクルと比較する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

各節は独立したノートブックとして動作するため、下のセルでは 17.2 節から必要な要素、すなわち 3 モード混合分布とその 2 つの経路である分散保持（variance-preserving、VP）経路および線形経路を再構築する。また、前節と同様に、任意の点とノイズレベルにおける混合分布の厳密

な事後平均 $(\hat{\mathbf{x}}_0, \hat{\epsilon})$ を `oracle` という関数名で定義する。さらに、本節で扱う 2 つのサンプラに対応して 2 つのネットワークを再訓練する。1 つはフロー用の線形経路上の速度モデルであり、もう 1 つは DDPM の連鎖用の VP 経路上のノイズモデルである。訓練レシピは前節のステップ数を半分にしたもので、2 万ステップで十分である。これを確認するため、直近 1,000 ステップの平均訓練損失を表示し、前節で求めた下限（速度では約 4.9、ノイズでは 0.54）と比較する。2 つの訓練を合わせても、1 基の GPU で約 1 分で完了する。

```
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)
vp, linear = d2l.GaussianPath.vp(), d2l.GaussianPath.linear()

def oracle(x, alpha, sigma):
    """The exact posterior means (x0_hat, eps_hat) of the mixture at x."""
    x0_hat = mix.posterior_mean(x, alpha, sigma)
    a, s = (torch.as_tensor(v, device=x.device).reshape(-1, 1)
            for v in (alpha, sigma))
    return x0_hat, (x - a * x0_hat) / s

num_steps = 20_000
torch.manual_seed(1)
_, ema_v, losses_v = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(linear, 'velocity'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
torch.manual_seed(2)
_, ema_eps, losses_eps = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(vp, 'eps'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
print(f'loss over the last 1,000 steps: velocity '
      f'{losses_v[-1000:].mean().item():.3f}, '
      f'noise {losses_eps[-1000:].mean().item():.3f}')
```

```
loss over the last 1,000 steps: velocity 4.919, noise 0.537
```

どちらの損失も下限に達しており、いずれのネットワークにも回帰からこれ以上学習すべきことはほとんど残っていない。肝心のサンプルにおいてこれらのネットワークがどれだけオラクルに迫れるかは、本節の後半で検証する。

17.3.1 確率質量の輸送

まず、速度場が密度に何をもたらすかから始めよう。p_t に従って分布する点の雲を考え、各点をその位置における速度 $\mathbf{u}_t(\mathbf{x})$ で微小時間だけ動かしてみる。点は生成も消滅もしないため、$\mathbf{x}$ における密度の変化は、流入する質量から流出する質量を引いた差によってのみ生じる。単位時間あたりに小さな窓を通過する質量は、密度と速度の積 $p_t\mathbf{u}_t$ である。この流れが $\mathbf{x}$ の周囲の微小な箱から単位体積あたりに正味で運び出す量は、その発散 $\nabla \cdot (p_t\mathbf{u}_t) = \sum_i \partial(p_t u_{t,i})/\partial x_i$ であり、その場所の密度はまさにその割合で減少する。これが流体力学における**連続の式**である。

$$\frac{\partial p_t(\mathbf{x})}{\partial t} = -\nabla \cdot \big(p_t(\mathbf{x})\,\mathbf{u}_t(\mathbf{x})\big).$$

密度と速度場がこの式を満たすとき、場に従って動く点は時間を順方向に進めても逆方向に進めても、すべての時刻で経路の密度を保つ。では、17.2 節の周辺速度は p_t とともにこの式を満たすだろうか。

まず 1 つのクリーンな点 $\mathbf{x}_0$ を考えよう。これにノイズを加えた点はガウス分布の雲 $\mathcal{N}(\alpha_t \mathbf{x}_0, \sigma_t^2 \mathbf{I})$ をなし、雲の各点は条件付き速度 $\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon$ で動く。ここで $\epsilon = (\mathbf{x} - \alpha_t \mathbf{x}_0)/\sigma_t$ は、その点を位置 $\mathbf{x}$ に置いたノイズである。この場を $\mathbf{u}_t(\mathbf{x} \mid \mathbf{x}_0)$ と呼ぶことにすると、これはこの 1 つの雲を正しく動かす。すなわち、同じノイズ ϵ を保ったまま動く点は各時刻で $\alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ に位置し、常にその雲からのサンプルとなっているため、この雲とその速度場は連続の式を満たす。周辺密度 p_t はデータ全体にわたる雲の平均であり、連続の式は密度に関して線形であるため、各雲の式を平均すると、右辺に平均質量流を持つ p_t の式が得られる。ベイズの定理 $p(\mathbf{x}_0)\, p_t(\mathbf{x} \mid \mathbf{x}_0) = p_t(\mathbf{x})\, p(\mathbf{x}_0 \mid \mathbf{x}_t = \mathbf{x})$ により、この流れは次のようになる。

$$\begin{aligned} \mathbb{E}_{\mathbf{x}_0}\left[p_t(\mathbf{x} \mid \mathbf{x}_0)\, \mathbf{u}_t(\mathbf{x} \mid \mathbf{x}_0)\right] &= p_t(\mathbf{x})\, \mathbb{E}\left[\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon \,\middle|\, \mathbf{x}_t = \mathbf{x}\right] \\ &= p_t(\mathbf{x})\, \mathbf{u}_t(\mathbf{x}). \end{aligned}$$

途中に現れた事後分布による平均は、周辺速度の定義そのものである。したがって、すべての雲が合わせて $\mathbf{x}$ を横切って運ぶ質量は、その位置の密度と周辺速度の積に等しく、p_t と $\mathbf{u}_t$ は連続の式を満たす。次節以降のサンプラはこの結果に基づいている。必要な条件を含めた厳密な証明は 38.7 節で与える。

速度を平均しても軌跡の平均にはならないではないか、と疑問に思う読者もいるだろう。その指摘は正しい。条件付き経路（線形経路では直線）はあらゆる時刻であらゆる点を通過し、いたるところで互いに交差する。これに対し、1 つのベクトル場の 2 本の軌跡が同じ時刻に交わることは決してない。ある時刻のある点における速度はただ 1 つしか存在しないからである。$\mathbf{u}_t$ に従うサンプルは、いかなる条件付き経路もたどらない。Figure 17.6 に、1 次元の 2 成分混合ガウス分布に対する両方の曲線を示す。オレンジ色で描かれたノイズは右側のモードにあるクリーンな点から生じたものだが、その軌跡は左側のモードに行き着く。上の議論は個々の軌跡については何も主張していない。主張しているのは、この速度場があらゆる時刻で**密度**を再現するということであり、サンプラに必要なのはその密度だけである。

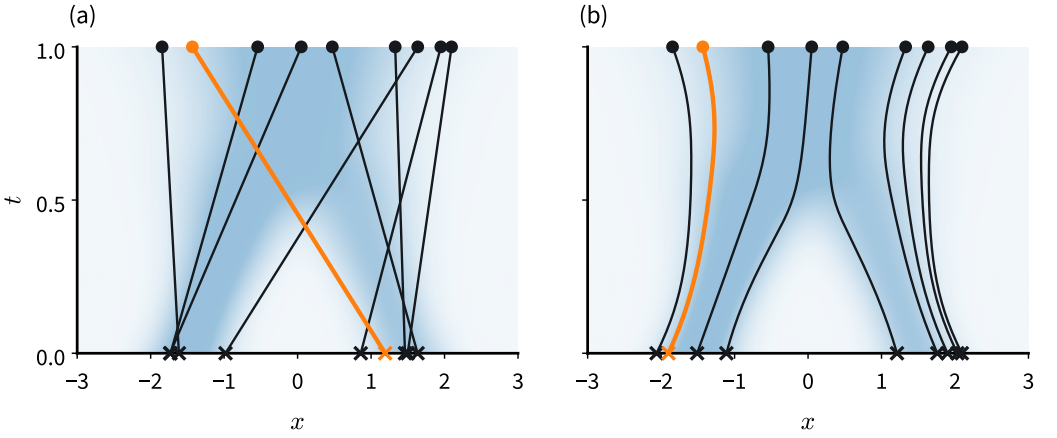


Figure 17.6: 1 次元の 2 成分混合ガウス分布における、線形経路上の条件付き経路と軌跡。時間は $t = 0$ のデータ（下）から $t = 1$ のノイズ（上）へと進み、陰影は密度 p_t を表す。(a) クリーンな点 $\mathbf{x}_0$（バツ印）とノイズ ϵ（点）の 9 組のペアが作る条件付き経路は直線であり、互いに交差する。(b) 同じ 9 個のノイズから出発する周辺速度場 $\mathbf{u}_t$ の軌跡は決して交わらない。オレンジ色のノイズは、条件付き経路に沿えば右側のモードに行き着くが、軌跡に沿えば左側のモードに行き着く。

これで冒頭の問いに答えが出た。厳密に解けば、フローはあらゆる瞬間で質量を保存するため、

モード重みを誤ることはあり得ない。ノイズの半分は必ず重み 1/2 のモードに行き着く。ランジュバン連鎖は、一度データ分布に到達すればそれを定常に保てるものの、谷を越えてそこへ運ぶ仕組みを持たなかった。アニーリングラダーは、ノイズが谷を埋める粗いレベルで谷を越えたが、必要なレベル数やステップ数は手探りで決めるしかなかった。これに対し、フローは既知の密度である標準ガウス分布から出発し、理論的な保証をもってデータへと輸送する。それでも問題が生じ得る箇所は 2 つ残っている。離散ステップは解を近似するにすぎず、ネットワークも速度場を近似するにすぎない点である。本節の残りでは、この両方の誤差を測定する。

以上より、このサンプラは初期点 $\mathbf{x}$ を $p_1 = \mathcal{N}(\mathbf{0}, \mathbf{I})$ から抽出し、次の常微分方程式（ordinary differential equation、ODE）を解く。

$$\frac{d\mathbf{x}}{dt} = \mathbf{u}_t(\mathbf{x}), \qquad \text{from } t = 1 \text{ down to } t = 0. \tag{17.21}$$

すべての時刻においてその解は p_t からのサンプルであり、$t = 0$ ではデータからのサンプルとなる。Y. Song, Sohl-Dickstein, et al. (2021) はこれを**確率フロー ODE**（38.5.4.3 節）と呼んだ。出発後にランダムな要素は一切入らず、サンプルのランダム性はすべてその初期点に集約されている。

17.3.2 確率フロー ODE に対するオイラー法

ODE は、小さな直線ステップを数多く積み重ねることで数値的に解かれる。**オイラー法**（Euler's method、38.2 節）は、時刻 1 から 0 までを K 個の等間隔なステップに分割し、各ステップで点を速度と逆向きに $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{u}_t(\mathbf{x})/K$（$t = 1, 1 - 1/K, \dots, 1/K$）と動かす。では、フローには何ステップ必要なのだろうか。一般のデータについて直接考えるのは難しいため、Nakkiran et al. (2025) にならい、答えが自明なデータから始めて段階的に議論を組み立てていこう。

17.3.2.1 1 点のデータからデータ平均へ

データが 1 つの点 $\mathbf{x}_0$ のみからなる場合を考える。線形経路（$\alpha_t = 1 - t$、$\sigma_t = t$）上では、ある点 $\mathbf{x}$ は時刻 t においてノイズ $\epsilon = (\mathbf{x} - (1-t)\mathbf{x}_0)/t$ を持ち、速度 $\epsilon - \mathbf{x}_0 = (\mathbf{x} - \mathbf{x}_0)/t$ で動く。すべての軌跡は一定速度で $\mathbf{x}_0$ へと向かう直線となり、長さ 1 のオイラーステップ 1 回、すなわち $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{u}_1(\mathbf{x}) = \mathbf{x}_0$ で厳密な解が得られる。データ点が 2 つになると、速度場はそのような 2 つの直線的な速度場を事後確率で重み付け平均したものとなり、事後分布が一方の点から他方の点へと切り替わる場所で軌跡が曲がる（38.7.1.4 節）。

任意の分布は点質量の混合分布とみなせるため、事後分布がいずれの点とも決まらない場所では必ず軌跡が曲がる。$t = 1$ においては、事後分布は完全に未決定である。純粋なノイズの点からは元のクリーンな点について何の手がかりも得られないため、事後分布は事前分布に一致し、次の関係が成り立つ。

$$\mathbf{u}_1(\mathbf{x}) = \mathbb{E}\big[\epsilon - \mathbf{x}_0 \,\big|\, \mathbf{x}_1 = \mathbf{x}\big] = \mathbf{x} - \mathbb{E}[\mathbf{x}_0].$$

したがって、長さ 1 のオイラーステップを 1 回行うと、どの初期点もデータの平均へと送られる。本節の混合分布については、コードを実行する前から結果を予測できる。すなわち $0.5\,(-2.5, -1.5) + 0.3\,(2.5, -1.5) + 0.2\,(0, 2.5) = (-0.5, -0.7)$ であり、これは各モードの間の、最も近いモードからも標準偏差の 4 倍以上離れた点である。ステップ数を増やすとどうなるかを見るため、線形経路上の混合分布の厳密な速度場 `mix.velocity` に対し、原点および原点から距離 1.5 にある 4 点の計 5 個の固定初期点から、1、10、100 ステップでオイラー法を実行してみよう。

```
@torch.no_grad()
def euler(net, x, steps):
    """Euler's method on dx/dt = net(x, t), from t = 1 down to t = 0."""
    for k in range(steps, 0, -1):
        t = torch.full((len(x),), k / steps, device=x.device)
        x = x - net(x, t) / steps
    return x

starts = torch.tensor([[0.0, 0.0], [1.5, 0.0], [-1.5, 0.0], [0.0, 1.5],
                       [0.0, -1.5]], device=device)
ends = torch.stack([starts] + [euler(mix.velocity, starts, steps)
                               for steps in (1, 10, 100)], 1)
print('start         1 step          10 steps        100 steps')
for row in ends.tolist():
    print('  '.join(f'({x:5.2f}, {y:5.2f})' for x, y in row))
```

```
start          1 step          10 steps        100 steps
( 0.00,  0.00)  (-0.50, -0.70)  (-1.91, -1.27)  (-1.98, -1.30)
( 1.50,  0.00)  (-0.50, -0.70)  ( 2.73, -1.34)  ( 2.90, -1.36)
(-1.50,  0.00)  (-0.50, -0.70)  (-2.92, -1.38)  (-3.06, -1.40)
( 0.00,  1.50)  (-0.50, -0.70)  (-0.11,  2.59)  (-0.09,  2.77)
( 0.00, -1.50)  (-0.50, -0.70)  (-1.98, -2.06)  (-2.05, -2.17)
```

1 ステップの列は予測どおりであり、5 つの初期点すべてがデータの存在しない $(-0.50, -0.70)$ に着地している。10 ステップになると、すべての初期点がすでにいずれかのモードの内部に入り、100 ステップの終点から約 0.2 以内の位置に達する。ただし、いずれもデータ平均に近い手前側にとどまっており、粗いステップには 1 ステップの誤差がわずかに残っている。次に、各初期点が**どの**モードに到達するかに注目しよう。原点、その下の初期点、および左の初期点はすべて最も重みの大きいモードに行き着き、右の初期点は右側のモード、上の初期点は上側のモードに行き着く。ここに質量保存則が働いている。標準ガウス分布の半分は重み 1/2 のモードに行き着かなければならないため、そのモードが最も広い初期点の領域（ここではガウス分布の中心を含む）を集めるのである。また、終点がモードの中心に集中するわけでもない。原点は $(-1.98, -1.30)$ に行き着くが、これはモードの中心からモードの標準偏差で約 1 つ分離れた位置である。フローはノイズからデータへの写像であり、それぞれの初期点が固有のサンプルに対応している。

17.3.2.2 すべての経路に共通する座標系

時刻 t に関するオイラーステップは経路に依存する。VP 経路上では、ノイズレベルは $\sigma_t = 0.96$（$t = 0.5$）にとどまり、そこから 0.32（$t = 0.1$）まで低下する（17.2 節）。つまり、前半の時間ではノイズがほとんど除去されず、その 3 分の 2 が次の 5 分の 2 の時間で除去される。t に関するステップは始点における変化率のまま進むため、このように曲がった経路上では大きなステップをとると大きく外れてしまう（演習問題 1）。そこで本書のライブラリのサンプラでは、あらゆるガウス経路が同じ形に見える座標系でステップを進める。経路を α_t で割ってみよう。

$$\bar{\mathbf{x}} = \frac{\mathbf{x}_t}{\alpha_t} = \mathbf{x}_0 + \bar{\sigma}\,\epsilon, \qquad \bar{\sigma} = \frac{\sigma_t}{\alpha_t}.$$

この座標系ではデータのスケールは一定に保たれ、ノイズだけがデータの $\bar{\sigma} = 0$ から純粋なノイズの $\bar{\sigma} = \infty$ まで増大する。あらゆる経路が 17.2 節の分散爆発（variance-exploding、VE）経路（アニーリングラダーが下っていった経路）になり、$\bar{\sigma}$ がそのノイズレベルとなる。Karras et al. (2022) はすべての経路をこの方法でサンプリングしている。ここでの上線（バー）は α_t によ

る除算を表しており、DDPM の $\bar{\alpha}_t = \alpha_t^2$ における二乗の意味ではない（Table 17.1）。1 つの条件付き経路に沿うと、$\bar{\mathbf{x}}$ は $\bar{\sigma}$ に対して傾き ϵ の直線上を動く。周辺速度場はこの傾きの事後平均、すなわちノイズ予測（noise prediction）である。

$$\frac{d\bar{\mathbf{x}}}{d\bar{\sigma}} = \hat{\epsilon}(\bar{\mathbf{x}}).$$

最初の式の事後平均をとると $\bar{\mathbf{x}}_t = \hat{\mathbf{x}}_0 + \bar{\sigma}_t\hat{\epsilon}$ となるため、$\bar{\sigma}_t$ から $\bar{\sigma}_s$ へのオイラーステップは $\bar{\mathbf{x}}_s = \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\hat{\epsilon} = \hat{\mathbf{x}}_0 + \bar{\sigma}_s\hat{\epsilon}$ となり、これに α_s を掛けると次式を得る。

$$\mathbf{x}_s = \alpha_s\,\hat{\mathbf{x}}_0 + \sigma_s\,\hat{\epsilon}.$$

このように、1 回のステップはクリーンな点とノイズを予測し、それらを次のノイズレベルに配置する。この更新式には 17.5 節で DDIM という名前のもとで再び出会うことになる。線形経路上ではこれは t に関するオイラーステップそのものであり、それは $\mathbf{x}_t = (1-t)\hat{\mathbf{x}}_0 + t\hat{\epsilon}$ を $\mathbf{x}_t - (t-s)(\hat{\epsilon} - \hat{\mathbf{x}}_0)$ に代入すれば確かめられる。他の経路上では異なる手法となり、1 点のみからなるデータに対しては任意のステップ幅で厳密になるのに対し、上で見た VP 経路のストライドにおける t のオイラーステップは大きく外れる（演習問題 1）。

`ode_sample` は、レベル $\bar{\sigma}$ のリスト（`sigma_bars` が経路の t グリッドから計算する）に沿ってこのステップを進める。純粋なノイズでは $\bar{\sigma}$ は無限大となり $\bar{\mathbf{x}}$ は定義されないが、ステップに必要なのは $\hat{\mathbf{x}}_0$ と $\hat{\epsilon}$ だけであり、予測器はこれらをノイズそのものから計算できるため、最初のステップは $\mathbf{x}$ から開始する。

```
@d2l.add_to_class(d2l.GaussianPath)
def sigma_bars(self, steps):
    """sigma_bar = sigma / alpha on the grid of steps; inf where alpha = 0."""
    a, s = self.grid(steps)
    return [s_k / a_k if a_k > 0 else math.inf for a_k, s_k in zip(a, s)]

def pred_bar(pred, xb, r):
    """pred at x = alpha x_bar, sigma_bar = r; at r = inf, x_bar is noise."""
    if math.isinf(r):
        return pred(xb, 0.0, 1.0)
    a = 1 / math.sqrt(1 + r ** 2)
    return pred(xb * a, a, r * a)

@torch.no_grad()
def ode_sample(pred, x, sigma_bars):
    """Euler (= DDIM) on d x_bar / d sigma_bar = eps, from x to data; x is the
    noisy point at the first level, unit-variance scale (noise if inf)."""
    r0 = sigma_bars[0]
    xb = x if math.isinf(r0) else x * math.sqrt(1 + r0 ** 2)  # x / alpha
    for r_t, r_s in zip(sigma_bars[:-1], sigma_bars[1:]):
        x0, eps = pred_bar(pred, xb, r_t)
        xb = x0 + r_s * eps                    # the Euler (= DDIM) step
    return xb
```

サンプラは $\hat{\mathbf{x}}_0$ と $\hat{\epsilon}$ を任意の (α, σ) で必要とし、`make_pred` はどの種類の訓練済みネットワークであってもそのような予測器に変換する。本節では各ネットワークをそれ自身の経路上で動かすため、`make_pred` は予測値をこのペアに変換するだけである。後の節では別の経路のグリッド上でネ

ットワークを動かすが、その場合 make_pred はまずネットワーク自身の経路上で同じ対数信号対雑音比（log-SNR）を持つ時刻を（time_at 内の二分法によって）見つけ、点をその経路に合わせて再スケーリングする。なお、ノイズモデルには 1 つ死角がある。純粋なノイズ $\alpha = 0$ ではクリーンな点 $\hat{\mathbf{x}}_0 = (\mathbf{x} - \sigma\hat{\epsilon})/\alpha$ が定義されないため、ノイズモデルは線形経路上でサンプラを開始できない。一方、VP 経路上では α は常に正に保たれ、約 0.007（$t = 1$）となるため、ノイズモデルはそちらで用いる。

```
@d2l.add_to_class(d2l.GaussianPath)
def time_at(self, log_snr):
    """The time with a given log-SNR, by bisection (lambda_t falls in t)."""
    lo, hi = torch.zeros_like(log_snr), torch.ones_like(log_snr)
    for _ in range(50):
        mid = (lo + hi) / 2
        later = self.log_snr(mid) > log_snr      # the answer lies after mid
        lo, hi = torch.where(later, mid, lo), torch.where(later, hi, mid)
    return (lo + hi) / 2

def make_pred(net, path, kind):
    """pred(x, alpha, sigma) -> (x0, eps) from net(x, t) of a kind on path."""
    def pred(x, alpha, sigma):
        log_snr = 2 * math.log(alpha / sigma) if alpha > 0 else -math.inf
        t = path.time_at(torch.tensor(log_snr, dtype=torch.float64))
        x = x * (path.sigma(t).item() / sigma)  # the same point on our path
        t = torch.full((len(x),), t.item(), device=x.device)
        return path.x0_eps(kind, net(x, t), x, t)
    return pred
```

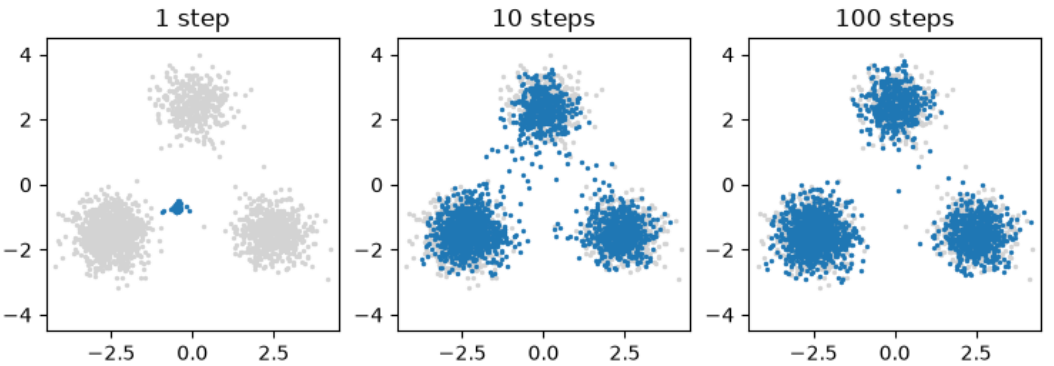
17.3.2.3 学習された速度場

訓練した速度ネットワークは厳密な速度場と同じように振る舞い、1 ステップ後にデータ平均に着地し、10 ステップ後に各モードへ収まるだろうか。標準ガウス分布から抽出した 2,000 個の初期点から、ode_sample を 1、10、100 ステップ実行して確かめよう。数値の意味を読み取りやすくするため、さらに 2 行追加する。データそのものから抽出した 2,000 点（プロットの灰色の点）と、同じ初期点からオラクルで同じ 10 ステップを実行した結果である。最後に、オラクルを用いた ode_sample と、mix.velocity を用いた単純なオイラー法のループを比較する。線形経路上では両者は同一のステップを踏むため、丸め誤差の範囲で一致するはずである。

```
torch.manual_seed(3)
x_T = torch.randn(2000, 2, device=device)
pred_v = make_pred(ema_v, linear, 'velocity')
samples = [ode_sample(pred_v, x_T, linear.sigma_bars(steps))
           for steps in (1, 10, 100)]
data = mix.sample(2000)
print('one step: mean', [round(v, 3) for v in samples[0].mean(0).tolist()],
      'std', [round(v, 3) for v in samples[0].std(0).tolist()])
print(f'{"data":>18}:', mix.summary(data))
for steps, x in zip((10, 100), samples[1:]):
    print(f'{steps:>3} steps, network:', mix.summary(x))
print('  10 steps, oracle:',
      mix.summary(ode_sample(oracle, x_T, linear.sigma_bars(10))))
exact = (ode_sample(oracle, x_T, linear.sigma_bars(100))
```

```
            - euler(mix.velocity, x_T, 100)).abs().max().item()
    print(f'ode_sample against the loop on the exact field: {exact:.0e}')
    d2l.scatter_panels(samples, ['1 step', '10 steps', '100 steps'], data=data);
```

```
one step: mean [-0.511, -0.736] std [0.036, 0.027]
              data: {'weights': [0.517, 0.292, 0.192], 'stray': 0.0145, 'spread': 0.488}
 10 steps, network: {'weights': [0.522, 0.273, 0.206], 'stray': 0.0315, 'spread': 0.447}
100 steps, network: {'weights': [0.512, 0.278, 0.211], 'stray': 0.0125, 'spread': 0.475}
  10 steps, oracle: {'weights': [0.522, 0.273, 0.206], 'stray': 0.019, 'spread': 0.459}
ode_sample against the loop on the exact field: 9e-05
```



1 ステップでは、すべての初期点がデータ平均 $\approx (-0.5, -0.7)$ へと崩壊する（$t = 1$ における小数第 2 位程度の小さなネットワーク近似誤差を除く）。10 回のオイラーステップを実行すると、3 つのモードが $0.5, 0.3, 0.2$ に近い重みで分離されるものの、$\approx 3\%$ の逸脱点が生じ（真のデータでは 1.1%〜1.5%）、モード内の広がりもわずかに狭くなる（0.45 対 0.48〜0.49）。厳密な `oracle` を用いて 10 回のオイラーステップを実行すると、同じモード重み（差は 0.003 以内）と 0.46 の広がりが得られる。厳密な速度場でもネットワークとほぼ同じだけモードが狭まることから、この縮小はソルバに起因している。一方、逸脱点には 2 つの要因がある。オラクルでも 2%〜3% の逸脱点が残り、データよりは多いがネットワークよりは少ないため、ソルバとネットワークの双方が寄与している。100 回のオイラーステップを実行すれば、学習されたネットワークは真のデータの逸脱点の割合（$\approx 1.3\%$）、広がり、およびモード重みにモンテカルロ誤差の範囲内で一致し、厳密なオラクルに対する `ode_sample` も陽的オイラー法のループと単精度浮動小数点の精度（10^{-4}）まで一致する。

17.3.3 祖先連鎖と逆 SDE

DDPM のサンプリング方法はこれとは異なる。Figure 17.4 の逆連鎖を、ノイズを伴うステップを 1 回ずつ踏みながら $t = 1$ から $t = 0$ まで実行するのである。そもそもノイズを含むステップの連鎖でなぜうまくサンプリングできるのか、そしてそれはフローとどのような関係にあるのだろうか。

17.3.3.1 祖先ステップ

小さな順方向ステップの逆過程はほぼガウス分布に従い (Sohl-Dickstein et al. 2015)、クリーンな点が与えられれば厳密にガウス分布となる。$s < t$ に対し、ベイズの定理により $q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$ は $\mathbf{x}_s$ に関する 2 つのガウス分布、すなわち順方向ステップ $q(\mathbf{x}_t \mid \mathbf{x}_s)$ とノイズ付加 $q(\mathbf{x}_s \mid \mathbf{x}_0)$ の積に比例するため、$\mathbf{x}_t$ と $\mathbf{x}_0$ の線形結合を平均に持つガウス分布になる。Ho らのステップは、その平均において $\hat{\mathbf{x}}_0$ を $\mathbf{x}_0$ の代わりに用い、分散 $\beta = 1 - \alpha_t^2/\alpha_s^2$（$s$ から t への順方向ステップ

が加える分散）の新たなノイズを加える。これが**祖先**（ancestral）ステップと呼ばれるのは、グラフィカルモデルの祖先サンプリング（ancestral sampling）と同様に、各 $\mathbf{x}_s$ がその親 $\mathbf{x}_t$ のもとで抽出されるからである。VP 経路上において、$\widehat{\mathbf{x}}_0 = (\mathbf{x}_t - \sigma_t \widehat{\epsilon})/\alpha_t$ とスコア $\nabla \log p_t = -\widehat{\epsilon}/\sigma_t$ を平均の式に代入して整理すると、ステップは次のように書ける。

$$\mathbf{x}_s = \frac{\mathbf{x}_t + \beta \nabla \log p_t(\mathbf{x}_t)}{\sqrt{1-\beta}} + \sqrt{\beta}\,\mathbf{z}, \qquad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{17.22}$$

これは新しいサンプラなのだろうか、それとも先ほどのサンプラにノイズを加えたものなのだろうか。これを見るには、$1/\sqrt{1-\beta} \approx 1 + \beta/2$ と展開し、β に関する一次の項とノイズ項 $\sqrt{\beta}\,\mathbf{z}$ を残すとわかりやすい。するとステップは次の 2 つに分解される。

$$\begin{aligned}\mathbf{x}_s \approx &\left[\mathbf{x}_t + \tfrac{\beta}{2}\nabla \log p_t(\mathbf{x}_t) + \sqrt{\beta}\,\mathbf{z}\right] \\ &+ \tfrac{\beta}{2}(\mathbf{x}_t + \nabla \log p_t(\mathbf{x}_t)).\end{aligned}$$

角括弧の部分は、ランジュバンステップ $\mathbf{x} \leftarrow \mathbf{x} + \frac{h}{2}\nabla \log p + \sqrt{h}\,\xi$（17.1 節）をステップサイズ $h = \beta$ で密度 p_t に適用したものであり、p_t を定常に保ちつつ逸脱した点をそこへ引き戻す。第 2 項は、時刻 t から s へ戻る確率フロー ODE の 1 オイラーステップである。VP 経路上での周辺速度は $\mathbf{u}_t = -\frac{1}{2}\beta(t)\,(\mathbf{x} + \nabla \log p_t)$ となるが、これは 17.2 節のスコア・速度の恒等式において $\dot{\alpha}_t/\alpha_t = -\beta(t)/2$ および $\alpha_t^2 + \sigma_t^2 = 1$ としたものにほかならない。ここで $\beta(t) = 0.1 + 19.9\,t$ は VP スケジュールのレートであり、微小ステップでは $\beta \approx \beta(t)\,(t-s)$ が成り立つ。したがって、t から s へ戻るオイラーステップは点を $-(t-s)\,\mathbf{u}_t = \frac{1}{2}\beta(t)\,(t-s)\,(\mathbf{x}_t + \nabla \log p_t) \approx \frac{\beta}{2}(\mathbf{x}_t + \nabla \log p_t)$ だけ動かし、これが第 2 項に一致する。すなわち、祖先ステップとは、フローのステップに現在のノイズレベルにおけるランジュバン補正を加えたものである。この連鎖は ODE と同じ輸送を行いつつ、ランジュバン項の働きによって点を p_t へと引き戻し、それ以前のステップで生じた誤差の一部を打ち消すことができる。実際にその効果が現れるかどうかは 17.5 節で測定し、そこでは Figure 17.9 において同じ初期点から出発する決定論的軌跡とノイズを含む軌跡を描画する。

17.3.3.2 逆 SDE

連続時間では、この時間反転は厳密になる。本節のコードではこの極限を必要としないが、論文ではサンプラがこの記法で書かれるため、記法を 1 つだけ導入しておく価値がある。**確率微分方程式**（stochastic differential equation、SDE）$d\mathbf{x} = \mathbf{f}(\mathbf{x}, t)\,dt + g(t)\,d\mathbf{W}$ とは、$\Delta t \to 0$ の極限における次のステップの略記である。

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{f}\,\Delta t + g\sqrt{\Delta t}\,\mathbf{z}, \qquad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})\,:$$

すなわち、決定論的な**ドリフト** $\mathbf{f}$ に、タイムステップに比例して分散が増大するガウスノイズを加えたものである。DDPM の順方向連鎖は $\Delta t = 1/T$、$\mathbf{f} = -\frac{1}{2}\beta(t)\,\mathbf{x}$、$g^2 = \beta(t)$ としたこの形のステップになっている（$\sqrt{1-\beta_k} \approx 1 - \beta_k/2$ かつ $\beta_k \approx \beta(t)\,\Delta t$ であるため）。Anderson (1982) は、このような過程を $t=1$ から $t=0$ へと逆向きにたどる過程が、ステップ $\mathbf{x} \leftarrow \mathbf{x} - [\mathbf{f} - g^2\nabla \log p_t(\mathbf{x})]\,\Delta t + g\sqrt{\Delta t}\,\mathbf{z}$ の $\Delta t \to 0$ における極限によって与えられることを示した。この極限において、逆過程は順過程と同じ周辺分布 p_t を持つ。論文では、この反転を時間が逆向きに進む次の SDE として記述する。

$$\begin{aligned} d\mathbf{x} &= \left[\mathbf{f} - g^2 \nabla \log p_t(\mathbf{x})\right] dt + g\, d\bar{\mathbf{W}}, \\ \text{with the same marginals as} \quad \frac{d\mathbf{x}}{dt} &= \mathbf{f} - \tfrac{1}{2} g^2 \nabla \log p_t(\mathbf{x}), \end{aligned} \tag{17.23}$$

ここで、$d\bar{\mathbf{W}}$ は各逆方向ステップで加わる新たなノイズを表す。なぜこれら 2 つの式は同じ周辺分布を共有するのだろうか。両者の逆方向ステップの差は $\frac{1}{2} g^2 \nabla \log p_t \, \Delta t + g\sqrt{\Delta t}\, \mathbf{z}$ であり、これはステップサイズ $h = g^2 \Delta t$ による p_t 上のランジュバンステップにほかならない。ランジュバンステップは p_t を不変に保つため、先ほどの祖先ステップの分解が連続時間でそのまま成り立っているのである。VP 経路の場合、右側の ODE は $\mathbf{u}_t = -\frac{1}{2}\beta(t)\left(\mathbf{x} + \nabla \log p_t\right)$ となり、その経路の確率フロー ODE Equation 17.21 と一致する。一方、逆 SDE のステップは $\mathbf{x} + \frac{1}{2}\beta(t)\Delta t\, \mathbf{x} + \beta(t)\Delta t\, \nabla \log p_t + \sqrt{\beta(t)\Delta t}\, \mathbf{z}$ となり、$\beta = \beta(t)\, \Delta t$ とおいた祖先ステップの一次展開に一致する。すなわち、祖先連鎖は一次近似において、逆 SDE の最も単純な離散化であるオイラー・丸山法（Euler–Maruyama method、38.4.4.1 節）に対応している。ノイズを一切加えない ODE とこの SDE の間には、同じ周辺分布を持つ一連のサンプラ族が存在する。サンプラが注入するノイズが大きいほど、余分な広がりを引き戻すドリフト内のスコア項も強くなる（38.5.5 節、Figure 38.22）。

`ancestral_sample` は任意の経路と任意の予測器 `pred(x, alpha, sigma) -> (x0, eps)` に対し、一次近似ではなく厳密なステップを実行する。具体的には、$\mathbf{x}_s$ の条件付き平均（$\mathbf{x}_t$ と $\hat{\mathbf{x}}_0$ が与えられたもとでの平均）へと移動し、分散 $\sigma_t^2 - (\alpha_t/\alpha_s)^2 \sigma_s^2$（$s$ から t への順方向ステップの分散であり、VP 経路上では β に等しい）のノイズを加える。

```
@torch.no_grad()
def ancestral_sample(pred, x, path, steps, *, t_start=1.0):
    """Ho et al.'s sampler from x at t_start (1 is noise) to data at t = 0."""
    a, s = path.grid(steps, t_start)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        ratio = s[i + 1] * a[i] / (a[i + 1] * s[i])
        x = a[i + 1] * x0 + s[i + 1] * ratio * eps      # the posterior mean
        if s[i + 1] > 0:                                # no noise at t = 0
            var = s[i] ** 2 - (a[i] / a[i + 1] * s[i + 1]) ** 2  # forward step
            x = x + math.sqrt(var) * torch.randn_like(x)
    return x
```

フローと同じ 2,000 個の初期点から、ノイズモデルを用いてこの連鎖を 200 ステップ実行する。Ho らは画像に対して 1,000 ステップを用いたが、2 次元平面上では 200 ステップで十分である。

```
pred_eps = make_pred(ema_eps, vp, 'eps')
print(mix.summary(ancestral_sample(pred_eps, x_T, vp, 200)))
```

```
{'weights': [0.512, 0.297, 0.192], 'stray': 0.013, 'spread': 0.487}
```

この連鎖は、2,000 点の標本誤差の範囲内で 3 つのモードを正しい重みで復元し、逸脱点の割合も 100 ステップのフローと同様に約 1% 以下に収まる。

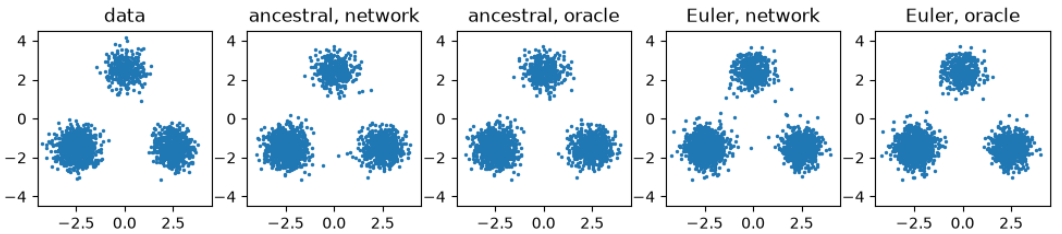
17.3.4 両サンプラからのサンプル

ここまでは各サンプラを 2,000 点に対して 1 回ずつ実行しただけであり、ネットワークの誤差とサンプリングの偶然によるばらつきを見分けるには点数が少なすぎた。そこで、同一の条件のも

とでネットワークとオラクルを比較する。20,000 個の初期点を用意し、ノイズモデルによる 200 回の祖先ステップと速度モデルによる 100 回のオイラーステップを実行したうえで、それぞれネットワークを oracle に置き換え、同じ初期点（連鎖については注入するノイズも同じ）から同じ計算を繰り返す。これにより、ネットワークの行と対応するオラクルの行との差はすべてネットワークに起因するものとなる。このサンプルサイズでは、モード重みの標準誤差は約 0.0035 である。さらに下のセルでは、ネットワークとオラクルが異なるモードへと送った初期点の個数も数える。

```
torch.manual_seed(5)
x_T = torch.randn(20_000, 2, device=device)
runs = {'data': mix.sample(20_000)}
for name, pred in (('network', pred_eps), ('oracle', oracle)):
    torch.manual_seed(6)                        # the same noise for both
    runs[f'ancestral, {name}'] = ancestral_sample(pred, x_T, vp, 200)
for name, pred in (('network', pred_v), ('oracle', oracle)):
    runs[f'Euler, {name}'] = ode_sample(pred, x_T, linear.sigma_bars(100))
for name, x in runs.items():
    print(f'{name:>18}:', mix.summary(x))
print('standard error of a weight:', [round(math.sqrt(w * (1 - w) / 20_000), 4)
                                      for w in mix.weights.tolist()])
mode = lambda x: ((x[:, None] - mix.means) ** 2).sum(-1).argmin(1)
for s in ('ancestral', 'Euler'):
    moved = mode(runs[s + ', network']) != mode(runs[s + ', oracle'])
    print(f'{s}: network and oracle end in different modes for '
          f'{moved.float().mean().item():.2%} of the starts')
d2l.scatter_panels([runs[k][:2000] for k in runs],
                   list(runs), lim=4.5, size=2.4);
```

```
              data: {'weights': [0.504, 0.299, 0.197], 'stray': 0.0103, 'spread': 0.476}
ancestral, network: {'weights': [0.502, 0.296, 0.202], 'stray': 0.0121, 'spread': 0.483}
 ancestral, oracle: {'weights': [0.503, 0.296, 0.201], 'stray': 0.0121, 'spread': 0.482}
    Euler, network: {'weights': [0.499, 0.305, 0.196], 'stray': 0.0127, 'spread': 0.471}
     Euler, oracle: {'weights': [0.499, 0.304, 0.197], 'stray': 0.0107, 'spread': 0.474}
standard error of a weight: [0.0035, 0.0032, 0.0028]
ancestral: network and oracle end in different modes for 0.38% of the starts
Euler: network and oracle end in different modes for 0.21% of the starts
```



どちらのサンプラでも、訓練済みネットワークと厳密なオラクルが異なるモードへ送る初期点は 20,000 個のうちわずか 0.2%〜0.5% にすぎず、ネットワーク実行時の各モード重みも対応するオラクル実行時の値と 0.004 以内で一致する（二項分布の標準誤差 ≈ 0.0035 と同程度である）。また、両サンプラとも真のデータの広がり（誤差 0.01 以内）および逸脱点の割合（$\approx 1.1\%$、すなわち $3s$ を超えるガウス分布の裾）を再現している。

この混合分布において 100〜200 ステップを実行すれば、決定論的な ODE と確率的な祖先連鎖は

実質的に同等の分布の精度を達成する。両者の違いはその構造的性質にある。確率フロー ODE はノイズ $\mathbf{x}_1$ とデータ $\mathbf{x}_0$ の間に決定論的で可逆な全単射を定めるため、潜在空間での補間や厳密な尤度評価が可能になる。一方、祖先連鎖はランジュバンノイズを注入することで、軌跡に沿った離散化誤差やスコア誤差を減衰させることができる（17.5 節）。Table 17.2 に、本章で扱う主要な生成モデルが経路・ターゲット・サンプラをどのように組み合わせているかをまとめる。

Table 17.2: 本章のモデル：ネットワークの訓練に用いるターゲットと経路、およびデータを生成するサンプラ。

モデル	ターゲットと経路	サンプラ	該当箇所
ノイズ除去スコアモデル	単一レベルでのスコア $-\epsilon/\sigma$	ランジュバン動力学	17.1 節
ノイズ条件付きスコアネットワーク	VE 経路上のラダーの各レベルにおけるスコア（重み $\tilde{\sigma}^2$）	アニーリング付きランジュバン動力学	17.2 節
DDPM	VP 経路上の ϵ	祖先連鎖または確率フロー ODE	17.2 節、本節
フローマッチング、Rectified Flow	線形経路上の $\epsilon - \mathbf{x}_0$	確率フロー ODE 上のオイラーステップ	17.2 節、本節
$\mathbf{v}$ 予測	コサイン経路上の $\mathbf{v}$	いずれも可	17.2 節、17.6 節
EDM	VE 経路上のクリーンな点（ノイズレベルごとに再スケーリング）	確率フロー ODE の 2 次ソルバ	17.5 節、17.6 節

17.4 節では 2 次元から 28×28 の画像へと進む。そこでは厳密なオラクルは利用できず、関数評価のたびに畳み込み U-Net の順伝播が必要になる。

17.3.5 まとめ

周辺速度 $\mathbf{u}_t(\mathbf{x}) = \mathbb{E}[\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon \mid \mathbf{x}_t = \mathbf{x}]$ は連続の式 $\partial_t p_t + \nabla \cdot (p_t \mathbf{u}_t) = 0$ を周辺密度 p_t とともに満たすため、確率フロー ODE $d\mathbf{x}/dt = \mathbf{u}_t(\mathbf{x})$ を $t = 1$ から $t = 0$ まで積分すれば、モード重みを保ったまま $\mathcal{N}(\mathbf{0}, \mathbf{I})$ が p_0 へと輸送される。$t = 1$ から $t = 0$ への 1 回のオイラーステップはすべての初期点をデータ平均 $\mathbb{E}[\mathbf{x}_0]$ へと崩壊させてしまうが、再スケーリングされた座標 $\bar{\mathbf{x}} = \mathbf{x}/\alpha_t$ と $\bar{\sigma} = \sigma_t/\alpha_t$ を用いれば、あらゆるガウス経路が共通の常微分方程式 $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$ に帰着し、そのオイラーステップは $\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sigma_s \hat{\epsilon}$ と書ける。ステップ分散 β の一次近似において、DDPM の確率的な祖先ステップはこの確率フローのオイラーステップと p_t 上のランジュバンステップの和へと分解され、これは Anderson の逆時間 SDE に対するオイラー・丸山法による離散化に対応している。

17.3.6 演習問題

1. **2 つの座標系における 1 ステップ。**
   1. 線形経路上において、ステップ $\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sigma_s \hat{\epsilon}$ がオイラーステップ $\mathbf{x}_t - (t - s)\,\mathbf{u}_t(\mathbf{x}_t)$（$t$ に関するステップ）に等しいことを示せ。
   2. 任意のガウス経路上において、データが 1 点 $\mathbf{x}_0 = \mathbf{0}$ のみであるとする。$\hat{\epsilon} = \mathbf{x}/\sigma_t$ となること、$\mathbf{x}_t$ を通る厳密な軌跡が $\mathbf{x}_s = (\sigma_s/\sigma_t)\,\mathbf{x}_t$ であること、そして最初の小問のステップがすべての $s < t$ に対してこの厳密な軌跡を正確にたどることを示せ。
   3. VP 経路上では、このデータの周辺速度は $\mathbf{u}_t(\mathbf{x}) = (\dot{\sigma}_t/\sigma_t)\,\mathbf{x}$ であり、$d\sigma_t^2/dt = \beta(t)\,\alpha_t^2$ かつ $\beta(t) = 0.1 + 19.9\,t$ が成り立つ。$t = 0.5$（$\alpha_t \approx 0.28$、$\sigma_t \approx 0.96$）から $s = 0.1$（$\sigma_s \approx 0.32$）まで 1 ステップ進めよ。t に関するオイラーステップは $\mathbf{x}$ を

何倍にするか。また本来は何倍にすべきか。この結果は、少数ステップで動作するサンプラが用いるべき座標系について何を示唆しているか。

2. **ガウスデータに対する祖先ステップの一次近似。**●データを $\mathcal{N}(\mathbf{0}, v_0\mathbf{I})$ とすると、VP 経路上では $p_t = \mathcal{N}(\mathbf{0}, v_t\mathbf{I})$（ただし $v_t = \alpha_t^2 v_0 + \sigma_t^2$）となる。
   1. Equation 17.22 の祖先ステップを t から s まで、厳密なスコアと $\beta = 1 - \alpha_t^2/\alpha_s^2$ を用いて 1 回行うと、p_t が $\mathcal{N}(\mathbf{0}, (v_s + \delta)\mathbf{I})$（ただし $\delta = \beta^2(1/v_t - 1)/(1-\beta)$）に写されることを示せ。
   2. このステップが厳密になるのはどのようなデータのときか。また、混合分布の 1 成分（$v_0 = 0.25$）のように標準ガウス分布より幅の狭いデータに対しては、得られる分散は大きすぎるか、それとも小さすぎるか。
   3. グリッドを細分化してステップ数を 2 倍にし、各ステップの β を約半分にしたとき、連鎖全体で蓄積される誤差はどのように変化するか。
3. **ステップサイズとガウス反転。**●祖先連鎖はガウス遷移を用いるが、これはステップが小さい場合に正当化される。本問ではどれほど小さければよいかを調べる。$\sigma_{t|s}^2 = \sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2$ を、s から $t > s$ への順方向ステップの分散とする。
   1. 厳密な逆遷移が混合分布 $q(\mathbf{x}_s \mid \mathbf{x}_t) = \mathbb{E}_{\mathbf{x}_0 \mid \mathbf{x}_t}\big[q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)\big]$ であり、その共分散が $\tau^2\mathbf{I} + c^2\,\mathrm{Cov}[\mathbf{x}_0 \mid \mathbf{x}_t]$ となることを示せ。ここで、$\tau^2 = \sigma_{t|s}^2\sigma_s^2/\sigma_t^2$ は $q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$ の分散であり、$c = \alpha_s\sigma_{t|s}^2/\sigma_t^2$ はその平均における $\mathbf{x}_0$ の係数である。
   2. 1 次元においてデータが等確率で ± 1 をとるものとし、$x_t = 0$ とする。逆遷移が $\mathcal{N}(\pm c, \tau^2)$ の等重み混合分布となり、それが二峰性になるのは $\zeta^2 = c^2/\tau^2$ が 1 を超えるとき、かつそのときに限ること、そして $\zeta^2 = \alpha_s^2\sigma_{t|s}^2/(\sigma_t^2\sigma_s^2)$ となることを示せ。また、ζ はステップ幅 $t - s$ を細かくするときどのように縮小するか。
   3. VP 経路上において、ζ を $t = 0.5$ から $s = 0.495$、0.45、0.3 への各ステップ、および $t = 0.3$ から $s = 0$ への 1 ステップについて計算せよ。二峰性になるのはどの逆遷移か。最後のケースにおいて、最適な平均を持つガウス遷移は $x_t = 0$ で何を生成し、本来は何を生成すべきか。
4. [code] **コミットメント時刻。**●経路上のどの時点でサンプルは自身の属するモードを決定するのだろうか。Biroli et al. (2024) はこれをクローン作成によって測定している。すなわち、p_t の 1 点から出発し、独立なノイズを用いて逆連鎖のコピーを 2 本走らせ、両者が同じモードに行き着くかどうかを記録する。
   1. 2,000 個のクリーンな点を抽出し、VP 経路上で $\mathbf{x}_t$（$t = 0.9, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1$）へとノイズを加え、各点から `ancestral_sample(pred_eps, x_t, vp, steps, t_start=t)` のコピーを 2 本ずつ約 $200\,t$ ステップで実行せよ。終点の最近傍平均が一致するペアの割合 $\varphi(t)$ をプロットせよ。φ は $t \to 1$ のときどの値に近づくか、またそれはなぜか。
   2. Λ をデータの共分散行列の最大固有値とする。Biroli らは、第 1 主成分方向の信号がノイズを上回らなくなる時点 $\alpha_t^2\Lambda = 1$ によって種分化時刻を定義している。この混合分布に対してその時刻を計算し、得られた曲線およびモード重みを決定づけた 17.2 節のアニーリングラダーの粗いレベルと比較せよ。演習問題 2（38.7 節）および演習問題 4（38.5 節）では、1 次元の 2 モードに対して同じ問題を扱っている。
5. **同じ比率、同じフロー。**●2 つのガウス経路の係数が $\alpha_t^B = c_t\,\alpha_t^A$ および $\sigma_t^B = c_t\,\sigma_t^A$（$c_t$ は滑らかな正値関数）という関係にあり、すべての時刻で同じ対数信号対雑音比を持つとする。
   1. $p_t^B(\mathbf{y}) = c_t^{-d}\,p_t^A(\mathbf{y}/c_t)$ を示せ。さらに、17.2 節のスコア・速度の恒等式 $\mathbf{u}_t(\mathbf{x}) = (\dot\alpha_t/\alpha_t)\,\mathbf{x} - (\sigma_t\dot\sigma_t - \sigma_t^2\dot\alpha_t/\alpha_t)\nabla\log p_t(\mathbf{x})$ を用いて、$\mathbf{u}_t^B(\mathbf{y}) = (\dot c_t/c_t)\,\mathbf{y} + c_t\,\mathbf{u}_t^A(\mathbf{y}/c_t)$ となることを示せ。ここから、確率フロー ODE の軌跡が $\mathbf{y}_t = c_t\,\mathbf{x}_t$ で関係づけられることを結論づけよ。
   2. $\tau(t)$ を単調増加関数とする。経路 $p_{\tau(t)}$ が速度場 $\dot\tau(t)\,\mathbf{u}_{\tau(t)}$ によって生成され、$\mathbf{u}$ と同

じ軌跡を持つことを示せ。

3. 時間変換 $\tau(t)$ として、コサイン経路 $(\cos\frac{\pi\tau}{2}, \sin\frac{\pi\tau}{2})$ の時刻 $\tau(t)$ における対数信号対雑音比が線形経路の時刻 t における値と一致するものを求め、$c_t = 1/\sqrt{(1-t)^2+t^2}$ となることを示せ。経路上のどこで再スケーリングが最大になるか。
4. コサイン経路の確率フロー ODE は $\mathcal{N}(\mathbf{0}, \mathbf{I})$ に従うデータに対して何をするか、またそれはなぜか。同じデータに対して線形経路の確率フロー ODE は何をするか。

17.4 画像の拡散モデル

17.2 節および 17.3 節で扱った速度回帰、確率フロー ODE（probability-flow ODE）、祖先連鎖（ancestral chain）は、任意の次元 d にそのまま適用できる。平面（$d = 2$）から 28×28 の Fashion-MNIST 画像（$d = 784$）へスケールアップするにあたり、4 つの実践的な構成要素を導入する。

1. **アーキテクチャ：**多重解像度 U-Net (Ronneberger et al. 2015; Ho et al. 2020) を用いる。これは大域的な画像構造を捉える粗い 7×7 のボトルネックと、細かな空間構造を保つフル解像度のスキップ接続（skip connection）を組み合わせたものであり、すべての残差ブロックで時刻とクラスの埋め込みによる条件付けを行う。
2. **訓練レシピ：**モデルパラメータの指数移動平均（exponential moving average、EMA）と、訓練中のランダムな**ラベルドロップアウト**を導入し、単一のネットワークで条件付き速度場と無条件速度場の両方を学習させる。
3. **評価：**ドメイン分類器で抽出した実画像と生成画像の特徴量分布間のフレシェ距離（Fréchet distance、FD）(Heusel et al. 2017) を用い、有限標本における基準下限に照らして較正する。
4. **ガイダンス：**条件付き予測と無条件予測の差に沿って外挿し、サンプル多様性とクラスの典型性（typicality）をトレードオフする**分類器なしガイダンス**（classifier-free guidance、CFG）(Dhariwal and Nichol 2021; Ho and Salimans 2022) を導入する。

本節では、単一の GPU で訓練可能な 120 万パラメータの U-Net を用い、これら 4 つの構成要素すべてを Fashion-MNIST 上で実装・評価する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.4.1 ノイズ除去 U-Net

17.4.1.1 スキップ接続、解像度、時間埋め込み

17.2 節のネットワークである `TimeMLP` は、点と時刻を受け取り、同じサイズの点を返した。画像のネットワークもこの入出力の約束事を維持する。すなわち、ノイズの乗った画像 $\mathbf{x}_t$ と時刻 t を入力とし、同じ形状の予測（ここでは速度）を出力する。画像には畳み込みが適しているが、フル解像度のまま 3×3 畳み込みを積み重ねただけでは、各出力ピクセルは入力の小さな窓にしか依存しない。一方、高ノイズ領域では、ぼやけた塊が靴なのかバッグなのかを画像全体から判断しなければならない。

U-Net は複数の解像度で動作する。エンコーダは解像度を 28×28 から 14×14、さらに 7×7 へと 2 回半分にし、チャネル数を 64 から 128 へと 1 回倍増させることで、最下層の 3×3 畳み込みが画像の大部分を見渡せるようにする。続いてデコーダが 28×28 までアップサンプリングして戻す。ボトルネックだけでは袖の輪郭やスニーカーの縞模様といった細部が失われてしまうため、各解像度のデコーダは同じ解像度のエンコーダの特徴量も受け取り、自身の特徴量と

連結する。これらの**スキップ接続**によって、ネットワークは U 字の形状をとる（Figure 17.7）。Ronneberger らは顕微鏡画像内の構造をセグメンテーションするためにこのアーキテクチャを設計した。セグメンテーションはピクセルごとに 1 つのラベルを求めるタスクであり、画像全体と各ピクセルの細部の両方に依存する入力と同じサイズの出力を必要とする。ノイズ除去器が求めるものもこれと同じである。

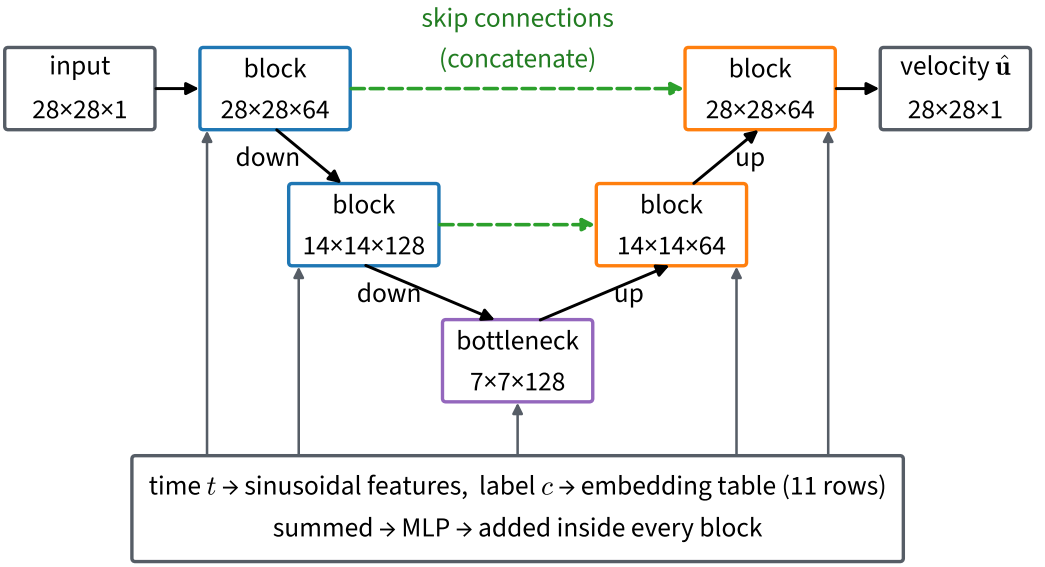


Figure 17.7: 本節の U-Net。2 段階のダウンサンプリング、7 × 7 のボトルネック、2 段階のアップサンプリングからなり、同じ解像度のエンコーダ特徴量をデコーダ特徴量に連結するスキップ接続を備える。時刻とクラスラベルは埋め込まれ、小さな MLP を経てすべてのブロック内部で加算される。

単一のネットワークがあらゆるノイズレベルでノイズ除去を担うが、最適な予測はレベルに応じて変化する。$t = 0$ 付近ではクリーンな画像はほぼ入力そのものであるのに対し、$t = 1$ では多数の画像の平均として推測するほかない。そのため、時刻はすべてのブロックに入力される。17.1 節の `d2l.time_embedding` は、単位時間あたり 1 から 64 ラジアンまで等比的に配置された 64 個の周波数における正弦と余弦へ t を写像する。これは 10.4 節の位置エンコーディングと同じ構成であり、数千までの位置の代わりに 0 から 1 の時刻に合わせて周波数を選ぶことで、近接するノイズレベルにも明確に異なる特徴量を与える。2 層の MLP がこれらを 1 つのベクトルに変換し、各ブロックはそのベクトルの学習された射影をチャネルごとのバイアスとして特徴マップに加算する。

各ブロックはグループ正規化（group normalization）を伴う 2 回の 3×3 畳み込みを適用し、その間に条件付け埋め込みの射影を加算し、両者をまたぐ残差接続を設ける。グループ正規化は 1 枚の画像内のチャネルグループにわたって統計量を計算するため、バッチ正規化とは異なり、異なるノイズレベルにある画像間で統計量が混ざることがない。

```
class UNetBlock(nn.Module):
    """Two 3x3 convolutions with group norm, a residual, and an embedding."""
    def __init__(self, c_in, c_out, emb_dim=None):
        super().__init__()
        self.norm1 = nn.GroupNorm(8, c_in)
```

```
        self.conv1 = nn.Conv2d(c_in, c_out, 3, padding=1)
        self.emb = nn.Linear(emb_dim, c_out) if emb_dim else None
        self.norm2 = nn.GroupNorm(8, c_out)
        self.conv2 = nn.Conv2d(c_out, c_out, 3, padding=1)
        self.skip = (nn.Conv2d(c_in, c_out, 1) if c_in != c_out
                     else nn.Identity())

    def forward(self, x, emb=None):
        h = self.conv1(F.silu(self.norm1(x)))
        if self.emb is not None:
            h = h + self.emb(emb)[:, :, None, None]
        h = self.conv2(F.silu(self.norm2(h)))
        return h + self.skip(x)
```

この U-Net は、ストライド 2 の畳み込み 2 回によって 7×7 まで縮小し、最近傍アップサンプリングによって各スキップ接続のサイズまで拡大して戻す。

```
class UNet(nn.Module):
    """A small U-Net on an image x (n, c, h, w), times t (n,) and labels y.

    y = num_classes is the null label. cond (n, emb_dim), if given, is added
    to the time features; the flow map of the few-step section passes the
    embedding of its step size there. time_freqs: see time_embedding."""
    def __init__(self, ch=64, num_classes=None, in_channels=1,
                 out_channels=1, emb_dim=128, time_freqs=(1.0, 64.0)):
        super().__init__()
        self.emb_dim, self.num_classes = emb_dim, num_classes
        self.time_freqs = time_freqs
        self.label_emb = (nn.Embedding(num_classes + 1, emb_dim)
                          if num_classes else None)
        self.emb_mlp = d2l.make_mlp([emb_dim] * 3, nn.SiLU())
        self.inc = nn.Conv2d(in_channels, ch, 3, padding=1)
        self.down1 = UNetBlock(ch, ch, emb_dim)                    # 28 x 28
        self.pool1 = nn.Conv2d(ch, ch, 3, stride=2, padding=1)     # 14 x 14
        self.down2 = UNetBlock(ch, 2 * ch, emb_dim)
        self.pool2 = nn.Conv2d(2 * ch, 2 * ch, 3, stride=2, padding=1)  # 7 x 7
        self.mid = UNetBlock(2 * ch, 2 * ch, emb_dim)
        self.up1 = UNetBlock(4 * ch, ch, emb_dim)                  # 14 x 14
        self.up0 = UNetBlock(2 * ch, ch, emb_dim)                  # 28 x 28
        self.outc = nn.Sequential(nn.GroupNorm(8, ch), nn.SiLU(),
                                  nn.Conv2d(ch, out_channels, 3, padding=1))

    def forward(self, x, t, y=None, cond=None):
        emb = d2l.time_embedding(t, self.emb_dim, self.time_freqs)
        if y is not None:
            emb = emb + self.label_emb(y)
        if cond is not None:
            emb = emb + cond
        emb = self.emb_mlp(emb)
        up = lambda h, skip: torch.cat([F.interpolate(
            h, size=skip.shape[-2:], mode='nearest-exact'), skip], 1)
        h1 = self.down1(self.inc(x), emb)
```

```
        h2 = self.down2(self.pool1(h1), emb)
        h = self.mid(self.pool2(h2), emb)
        h = self.up1(up(h, h2), emb)  # skip connection
        h = self.up0(up(h, h1), emb)  # skip connection
        return self.outc(h)
```

このようなネットワークでは、パラメータはどこに保持され、計算はどこで費やされるのだろうか。3×3 畳み込みはどの解像度でも同じ数の重みを持つが、ピクセルごとに 1 回実行されるため、同じ層でも 28×28 では 7×7 の 16 倍のコストがかかる。フル解像度で 64 チャネル、10 クラスとしてネットワークを構築し、解像度ごとにパラメータ数と計算量の両方を数えてみよう。

```
torch.manual_seed(0)
dev = d2l.try_gpu()
net = UNet(ch=64, num_classes=10).to(dev)
levels = {28: ['inc', 'down1', 'up0', 'outc'], 14: ['pool1', 'down2', 'up1'],
          7: ['pool2', 'mid']}  # the blocks whose output has each size
for side, names in levels.items():
    params = [p for n in names for p in getattr(net, n).parameters()]
    kernels = sum(p.numel() for p in params if p.dim() == 4)  # convolutions
    print(f'{side} x {side}: {sum(p.numel() for p in params):,} parameters, '
          f'{kernels * side ** 2 / 1e6:.0f}M multiply-adds per image')
print(f'total: {d2l.num_params(net):,} parameters')
```

```
28 x 28: 211,329 parameters, 152M multiply-adds per image
14 x 14: 493,376 parameters, 92M multiply-adds per image
7 x 7: 459,776 parameters, 22M multiply-adds per image
total: 1,198,913 parameters
```

下位 2 つの解像度レベルがパラメータの 4/5 を保持する一方で演算量の 2/5 しか消費しないのに対し、フル解像度レベルはパラメータの 1/6 しか持たないにもかかわらず演算量の半分以上を占める。7×7 のボトルネックは重みの 38% を持ちながら、積和演算（multiply-add）は 8% にすぎない。低解像度では層を深くしても計算コストが安いため、U-Net は最も幅の広い層を最下部に置くのである。ネットワーク 1 回の評価には約 2.5 億回の積和演算を要する。この数値は、1 つのサンプルを生成するのに何回ネットワークを評価するかを数える際に念頭に置いておきたい。

これは意図的に簡素化した U-Net である。その 120 万個のパラメータは、Ho et al. (2020) の CIFAR-10 モデルが持つ 3570 万個の 1/30 にすぎない。同モデルは 4 つの解像度それぞれに 2 つの残差ブロックを備え、16×16 での自己アテンションとドロップアウトを追加している。さらに公開コードでは、初期化時にすべてのブロックの最後の畳み込みをゼロに設定し、各ブロックがスキップ経路そのものとして始まるようにしている。本節ではこれらを一切使わないが、それでも以下で生成するサンプルは衣類として十分に識別できる。2024 年以降、新しい大規模画像生成器の多くは U-Net を圧縮画像のパッチに対する Transformer へと置き換え、クラスではなくテキストプロンプトで条件付けを行っているが（17.7 節）、本節の残りの内容はそのまま通用する。

17.4.1.2 クラスによる条件付け

クラスラベルは追加の入力の 1 つであり、U-Net にはすでにそうした入力を受け取る場所、すなわちすべてのブロックが特徴量に加算する埋め込みが用意されている。クラスごとに 1 つの学習されたベクトルを持つ埋め込みテーブルからラベルのベクトルを取り出し、MLP に通す前に時間特徴量へ加算する。このテーブルには余分な行が 1 つあり、それは「クラス指定なし」を表すヌ

ルラベル $\emptyset$（ここではインデックス 10）である。

訓練中、各ラベルを確率 0.1 でヌルラベルに置き換える。この操作を**ラベルドロップアウト**と呼ぶ。これにより、1 つのネットワークが 2 つの予測を学習する。すなわち、$\mathbf{u}_\theta(\mathbf{x}_t, t, c)$（クラス c の画像の速度）と、$\mathbf{u}_\theta(\mathbf{x}_t, t, \emptyset)$（10 個に 1 個の訓練例で学習される Fashion-MNIST 全体の速度）である。なぜ与えられたラベルを無視するようネットワークに教えるのだろうか。本節の最後で扱うガイダンス（17.4.4.2 節）では、ラベルありの予測に対するベースラインとしてラベルなしの予測が必要になるが、ラベルドロップアウトを使えば同じネットワーク内でほぼ追加コストなしにそれを用意できるからである。Ho and Salimans (2022) は、ドロップアウト率 0.1 や 0.2 が 0.5 よりも良好に機能することを見出した。ベースラインが必要とするデータ量は、それが支える条件付き予測よりも少なくて済むのである。

サンプラは $(\mathbf{x}, t)$ のみのベクトル場を積分する。`conditional` はバッチのラベル c を固定し、ネットワークを場 $(\mathbf{x}, t) \mapsto \mathbf{u}_\theta(\mathbf{x}, t, c)$ へと変換する。

```
def conditional(net, c):
    """The field (x, t) -> net(x, t, c) with the labels c fixed."""
    return lambda x, t: net(x, t, c)
```

17.4.2 訓練とサンプリング

17.4.2.1 損失とコスト

損失は 17.2 節の損失において点を画像に置き換えたものである。クリーンな画像 $\mathbf{x}_0$（値域 $[-1, 1]$）とそのラベル、時刻 t（$[0, 1]$ 上の一様分布）、およびノイズ ϵ を抽出し、線形経路上の $\mathbf{x}_t = (1 - t)\,\mathbf{x}_0 + t\,\epsilon$ を構成する。ネットワークの出力は条件付き速度（conditional velocity）$\epsilon - \mathbf{x}_0$ へと回帰させ、二乗誤差をピクセル全体で平均する。唯一の追加要素はラベルドロップアウトであり、`batch_fn` が 128 枚の画像からなる各バッチに適用する。

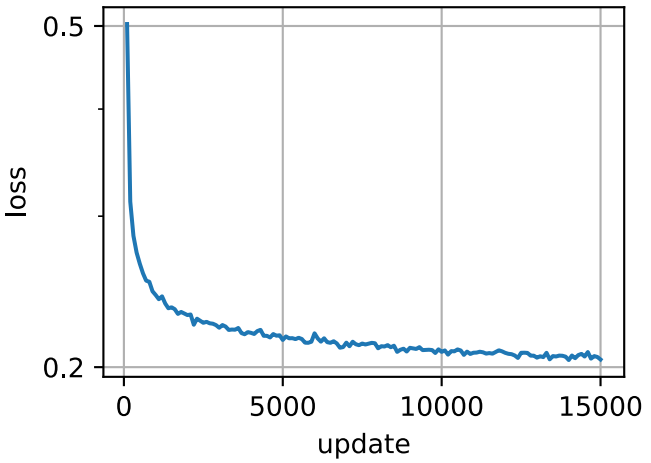
```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()
path = d2l.GaussianPath.linear()

def batch_fn():
    i = torch.randint(0, len(X), (128,), device=X.device)
    drop = torch.rand(128, device=X.device) < 0.1   # label dropout
    return X[i], torch.where(drop, 10, y[i])        # 10 is the null label

def loss_fn(net, batch):
    x0, c = batch
    t, eps = torch.rand(len(x0), device=x0.device), torch.randn_like(x0)
    x_t = path.sample(x0, t, eps)
    return ((net(x_t, t, c) - path.target('velocity', x0, eps, t)) ** 2).mean()
```

17.1 節の `d2l.train_regression` はこの損失に対して Adam を実行し、訓練済みネットワーク、その重みの移動平均、および損失の履歴を返す。ここでは定数の学習率 10^{-3} で 15,000 回の更新を行う。なお Ho et al. (2020) は学習率 $2 \cdot 10^{-4}$ でその 50 倍以上にあたる 800,000 回の更新を行って訓練している。

本節において、訓練はむしろ計算コストの軽いパートである。訓練では延べ $15{,}000 \times 128 \approx 1.9$ 百万枚の画像に対してネットワークの順伝播と逆伝播を実行する。一方、以下で生成するサンプル（一部は 1 サンプルあたり 1,000 ステップを要する）には約 2000 万回のネットワーク評価が必要となり、1 回の逆伝播を順伝播 2 回分と数えても訓練の 3.5 倍の演算量に達する。拡散モデル

（diffusion model）からサンプルを生成するコストは、生成方法をモデルに訓練するコストを上回り得るのであり、本章の残りの大部分はそのコストの削減を扱う。本節のコードは RTX 4090 上では 1 時間未満で完了するが、CPU 上では数時間を要する。GPU のない環境では、5,000 サンプルの代わりに 1,000 サンプルのスコアを計算し、後ほど算出する 1,000 枚時点の基準値と照らし合わせるとよい。

```
num_steps = 15_000
net, ema, losses = d2l.train_regression(net, loss_fn, batch_fn, num_steps)
d2l.plot(torch.arange(100, num_steps + 1, 100),
         losses.reshape(-1, 100).mean(1).cpu(), 'update', 'loss', yscale='log')
```



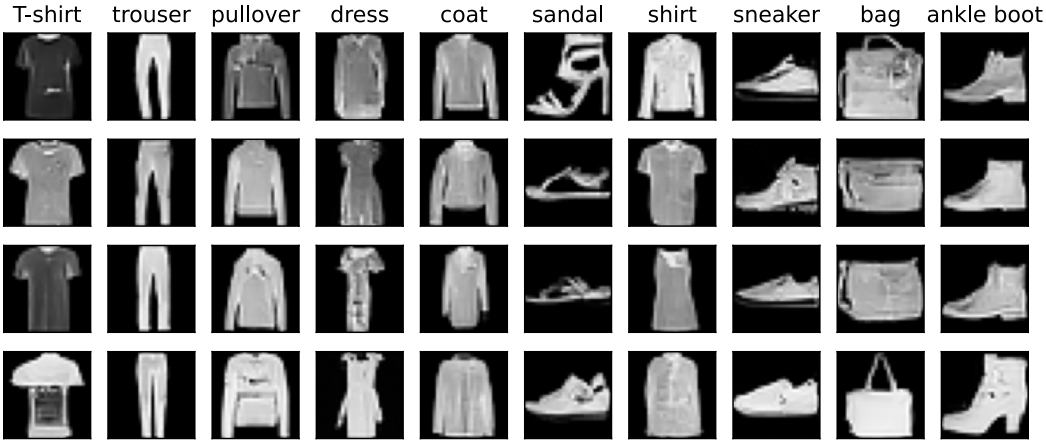
100 回の更新ごとに平均したピクセルあたりの損失は、最初の 1,000 回の更新のうちに約 0.5 から約 0.2 までほぼ下がりきる。損失がゼロよりかなり高い値で頭打ちになるのは、その下限が速度ターゲットの条件付き分散（conditional variance）だからである（17.2 節）。大きな t では、1 枚のノイズ画像と整合するクリーンな画像が多数存在する。

17.4.2.2 サンプルと予測

サンプリングにあたっては、17.3 節の `d2l.make_pred` でベクトル場をクリーンな画像とノイズの予測器へと変換し、`d2l.ode_sample` で確率フロー ODE を $t = 1$ から $t = 0$ まで（ここでは 50 回のオイラーステップで）積分する。`generate` はベクトル場を引数にとり、デフォルトは `conditional` だが、ガイダンスでは別の場を渡すことになる。この関数は n 枚の画像をチャンク単位で生成し、i 番目の画像はクラス $i \bmod 10$ とする。初期ノイズはシード（特に断らない限り 0）から生成するため、同じシードを用いた 2 回の呼び出しは、両者の間で変更した条件のみが異なる。

```
euler50 = lambda pred, x: d2l.ode_sample(pred, x, path.sigma_bars(50))

def generate(model, field=conditional, sampler=euler50, n=5000, chunk=500,
             seed=0):
    """n samples of field(model, c) from the noise of a seed; the i-th
    requests class i % 10."""
    gen = torch.Generator(dev).manual_seed(seed)
    z = torch.randn(n, 1, 28, 28, device=dev, generator=gen)
```

```
    c = torch.arange(n, device=z.device) % 10
    out = [sampler(d2l.make_pred(field(model, c_k), path, 'velocity'), z_k)
           for z_k, c_k in zip(z.split(chunk), c.split(chunk))]
    return torch.cat(out).clamp(-1, 1)

names = d2l.FashionMNIST().text_labels(range(10))
d2l.show_gray(generate(ema, n=40), 4, 10, titles=names + [''] * 30);
```



各列には 1 つのクラスのサンプルが 4 枚ずつ並んでいる。大半は指定したクラスだと認識でき、形状、柄、濃淡も多様である。一部には形の崩れたものもあり、また実画像と同様にシャツ、プルオーバー、コートは見分けにくい。

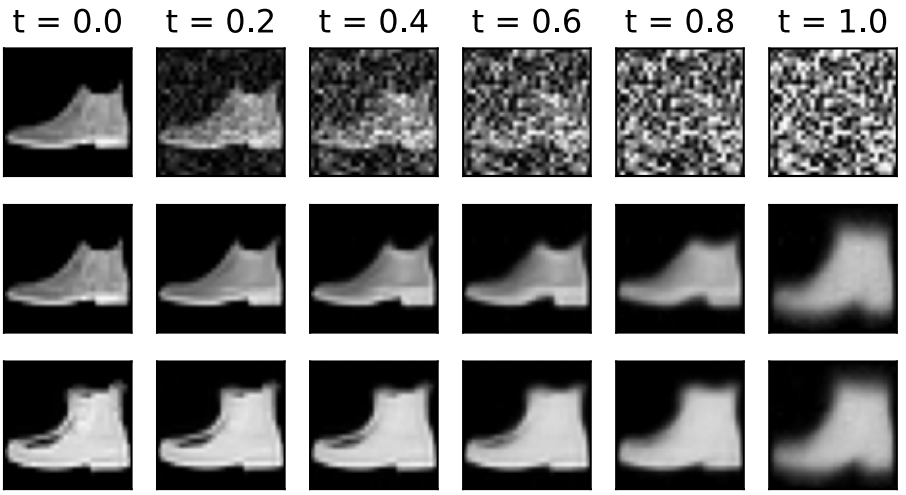
サンプラはあらゆるノイズレベルでネットワークに同じ問いを投げかける。このノイズ画像の背後にあるクリーンな画像はどれか、という問いである。各レベルでネットワークがどこまで答えられるかを見るため、以下の図の 1 行目では、アンクルブーツのテスト画像 1 枚に対し線形経路上の 6 つの時刻でノイズを加えている。2 行目は、ラベルを与えたときに各時刻でネットワークの予測が含意するクリーンな画像 $\hat{\mathbf{x}}_0 = \mathbf{x}_t - t\,\mathbf{u}_\theta(\mathbf{x}_t, t, c)$ を示す。2 行目の両端はあらかじめ予測できる。$t = 0$ では入力は元の画像そのものである。$t = 1$ では入力は純粋なノイズとなり、二乗誤差で訓練されたネットワークは残された情報であるラベルのもとでの事後平均（posterior mean）（17.1 節）、すなわちそのクラスの平均訓練画像を予測する。出力された最初の 2 行の数値は、各 $\hat{\mathbf{x}}_0$ と元の画像およびクラス平均画像とのピクセルあたりの二乗平均平方根（root mean square、RMS）距離である。

3 行目はサンプラ自体の挙動を示す。新たなノイズから出発し、`euler50` の 50 回のオイラーステップでアンクルブーツへ向かって積分を進めながら、途中の $t = 1, 0.8, \ldots, 0.2$ においてネットワークが含意するクリーンな画像を記録している。$t = 0$ の列は完成したサンプルであり、出力の最後の行は完成したサンプルから各時点の予測までの距離を示す。

```
torch.manual_seed(0)
ts = torch.linspace(0, 1, 6, device=dev)
x0, c = X_test[:1].expand(6, -1, -1, -1), y_test[:1].expand(6)
x_t = path.sample(x0, ts, torch.randn_like(x0[:1]).expand_as(x0))
with torch.no_grad():
    x0_hat = path.x0_eps('velocity', ema(x_t, ts, c), x_t, ts)[0]
rms = lambda a, b: ((a - b) ** 2).mean((1, 2, 3)).sqrt().tolist()
```

```
for name, ref in (('image', x0), ('class mean', X[y == c[0]].mean(0))):
    print(f'RMS from the {name}:', ' '.join(f'{e:.2f}' for e in rms(x0_hat, ref)))
x, grid, traj = torch.randn_like(x0[:1]), torch.linspace(1, 0, 51), []
with torch.no_grad():  # one sample of the class: 50 Euler steps from noise
    for i in range(50):
        t = grid[i].item()
        u = ema(x, torch.full((1,), t, device=dev), c[:1])
        if i % 10 == 0:                          # t = 1.0, 0.8, ..., 0.2
            traj.append(x - t * u)               # its predicted clean image
        x = x + (grid[i + 1].item() - t) * u
traj = torch.cat([x] + traj[::-1])               # from t = 0 (the sample) to 1
print('RMS from the sample:', ' '.join(f'{e:.2f}' for e in rms(traj, x)))
d2l.show_gray(torch.cat([x_t, x0_hat, traj]), 3, 6,
              titles=[f't = {t:.1f}' for t in ts.tolist()] + [''] * 12);
```

```
RMS from the image: 0.00 0.07 0.11 0.16 0.24 0.43
RMS from the class mean: 0.46 0.46 0.45 0.43 0.36 0.07
RMS from the sample: 0.00 0.03 0.07 0.12 0.19 0.32
```



両端は予測どおりの結果となる。$t = 1$ では、予測画像は平均的なアンクルブーツから 0.07 以内に位置し、元の画像からは 0.4 以上離れている。その中間では、ネットワークは私たち人間の目よりもよく見えている。$t = 0.6$ ではノイズの中にブーツを見つけるのは難しいが、それでも予測画像はその輪郭をとどめており、平均的なブーツからの距離が 0.4 超であるのに対し、元のブーツからは 0.2 未満に収まっている。細部が最初に消え、輪郭は最後まで残るのである。

サンプラはノイズレベルを逆順にたどり、3 行目はその順序が何をもたらすかを示している。$t = 1$ での予測はぼやけた平均的なブーツであり、最終的に生成されるサンプルからは 0.3〜0.4 離れている。$t = 0.6$ までには最終サンプルから 0.12〜0.14 以内に入り、計算元の画像がまだノイズ 6 対ブーツ 4 の割合であるにもかかわらず、すでに最終的な輪郭を備えている。残りのステップでエッジを鮮鋭化し、陰影と質感を加えていく。つまりサンプルは早い段階で形状を確定させ、遅い段階で表面の細部を仕上げる。演習問題 3 ではこの性質を活用する。

17.4.2.3 移動平均

上のサンプリングは、最終更新時点の重みではなく、重みの移動平均である `ema` から行った。定数学習率のもとでは、Adam の反復点は収束して静止することがない。各更新がミニバッチのノイズを伴うため、反復点は良好な領域の周囲に雲のように漂い、直近の反復点の平均をとる方がその中心の近くに位置する (Polyak and Juditsky 1992)。16.5 節の `d2l.EMA` は指数移動平均を保持し、更新のたびに平均を現在の重みへ向かって少しだけ動かす。

$$\bar{\theta} \leftarrow \bar{\theta} + \left(1 - 2^{-1/n_{1/2}}\right)\left(\theta - \bar{\theta}\right), \tag{17.24}$$

これにより、$n_{1/2}$ 回前の更新における重みは現在の重みの半分の重みを持つようになる。この $n_{1/2}$ を**半減期**（half-life）と呼ぶ。ここでは $n_{1/2} = 1000$ を用い、各更新で差の 0.07% を縮める。論文やライブラリでは代わりに**減衰率**（decay）$2^{-1/n_{1/2}}$（ここでは 0.9993）を指定することが多いため、両者を換算できるようにしておくと便利である。

半減期は訓練の長さに合わせる必要がある。短すぎれば直近ステップのノイズが平均に残り、長すぎれば良好な領域からまだ遠い初期の重みまで引きずってしまう。Ho et al. (2020) は 800,000 回の更新からなる訓練において減衰率 0.9999、すなわち約 7,000 更新分の半減期を用いている。移動平均がどれほど有効であるかは、評価指標を導入した直後にまず測定する。

17.4.3 サンプル品質の測定

17.4.3.1 学習された特徴空間におけるフレシェ距離

40 枚のサンプルを並べたグリッドを見れば、サンプルがもっともらしいかどうかはわかる。しかし、5,000 枚のサンプルが適切な多様性を備えているか、バッグが多すぎてサンダルが少なすぎないか、あるいは見た目にはどちらも良好な 2 つのモデルのどちらが優れているかまでは判断できない。そのためには、サンプルの分布を実画像の分布と比較する数値指標が必要である。

このような比較を行う場として、ピクセル空間は不適切である。スニーカーを 2 ピクセル平行移動させただけで、描かれている内容は変わらないのにユークリッド距離では大きく離れてしまうからである。そこで、内容の似た画像同士が近くに配置される分類器の特徴量空間で画像を比較する。Heusel et al. (2017) の**フレシェ距離**（FD）は、実画像と生成画像の特徴量にそれぞれガウス分布 $\mathcal{N}(\mu_p, \Sigma_p)$ と $\mathcal{N}(\mu_q, \Sigma_q)$ を当てはめ、16.5 節で示した両者間の二乗 2 次ワッサースタイン距離を報告する。

$$\mathrm{FD} = \|\mu_p - \mu_q\|^2 + \mathrm{tr}\Big(\Sigma_p + \Sigma_q - 2\big(\Sigma_p^{1/2}\Sigma_q\Sigma_p^{1/2}\big)^{1/2}\Big). \tag{17.25}$$

第 1 項は、たとえば特定のクラスばかり過剰に生成されるなどして平均的なサンプルが平均的な実画像と異なるときに大きくなる。第 2 項は、たとえば生成されたコートがどれも似通っているなどして分布の広がりが異なるときに大きくなる。両方の項がゼロになるのは、2 つのガウス分布が一致する場合に限られる。

論文等で公表される FID と呼ばれるスコアは、ImageNet で訓練された Inception ネットワークの特徴量を用いる。ここでは 16.5 節と同様、代わりに独自の分類器を訓練する。具体的には、グレースケールのチャネルを 3 回繰り返した Fashion-MNIST 上で 15 エポック訓練した、128 次元の特徴量を持つ小規模な畳み込み分類器 `d2l.FeatureCNN`（オンライン付録「ユーティリティ関数とクラス」）を用いる。特徴量ネットワークは独自のものであるため、本節ではこのスコアを FD と呼び、既存の論文値とは決して比較しない。`fashion_feature_cnn` がこの分類器を訓練し、

fd_features が 1,000 枚ずつ画像の特徴量を計算する。また、ネットワークのヘッドは各画像にクラスを割り当てる。

```
def fashion_feature_cnn(X, y):
    """The feature network that judges Fashion-MNIST samples in this chapter."""
    cnn = d2l.FeatureCNN(resolution=28).to(X.device)
    d2l.train_feature_cnn(cnn, X.expand(-1, 3, -1, -1), y)
    return cnn

def fd_features(cnn, imgs):
    """Features of gray images (n, 1, 28, 28) in [-1, 1], on the CPU."""
    rgb = lambda x: x.clamp(-1, 1).expand(-1, 3, -1, -1)
    return d2l.batched(lambda x: cnn.features(rgb(x)), imgs).cpu()

cnn = fashion_feature_cnn(X, y)
f_test = fd_features(cnn, X_test)
c_test = cnn.head(f_test.to(dev)).argmax(1)
print(f'test accuracy {(c_test == y_test).float().mean():.3f}')
```

```
test accuracy 0.926
```

この特徴量ネットワークはテスト画像の約 93% を正しく分類するため、その特徴量は各クラスを十分に分離している。fd_and_agreement は、基準となる特徴量に対するサンプル集合の FD とあわせて、指定したクラスに特徴量ネットワークが割り当てたサンプルの割合である**クラス一致率**（class agreement）を計算する。ここでは最初の 5,000 枚のテスト画像に対して 5,000 枚のサンプルのスコアを計算する。

```
def fd_and_agreement(cnn, imgs, f_ref):
    """FD to the reference features, and the fraction of samples that the
    feature network assigns to the requested class i % 10."""
    f = fd_features(cnn, imgs)
    cls = cnn.head(f.to(imgs.device)).argmax(1)
    agree = (cls == torch.arange(len(imgs), device=cls.device) % 10)
    return d2l.fid_score(f, f_ref), agree.float().mean().item()

fd_ema, agree_ema = fd_and_agreement(cnn, generate(ema), f_test[:5000])
print(f'5,000 samples: FD {fd_ema:.2f}, class agreement {agree_ema:.3f}')
```

```
5,000 samples: FD 0.82, class agreement 0.880
```

サンプルの FD は約 0.8 であり、特徴量ネットワークはその 90% 近くを指定どおりのクラスに割り当てている。では、0.8 という FD は良い値なのだろうか。この数値は特徴量ネットワークと画像の枚数に依存するため、単独では何も語らない。次項でその基準となる目盛りを与える。

17.4.3.2 フレシェ距離の基準値

実画像からなる 2 つの集合の間でも FD は正の値をとる。標本平均と標本共分散は偶然によって食い違い、Equation 17.25 のプラグイン推定量は（主にトレース項において）上向きの統計的バイアスを持つ。その大きさはおおよそ $1/n$（n は 1 集合あたりの画像枚数）に比例する。これがどれほどの大きさになるかを確かめるため、互いに素な 5 組の実訓練画像集合を、サンプルの基準集合である最初の n 枚のテスト画像に対して $n = 1{,}000$ と $n = 5{,}000$ でスコアリングする。この結果を、その枚数における FD の**基準値**（reference value）と呼ぶ。バイアスが $1/n$ に従う

なら、5,000 枚での値は 1,000 枚での値の約 1/5 になるはずである。その横には Bińkowski et al. (2018) の Kernel Inception Distance（KID）も表示する。これは同じ特徴量上で 3 次カーネルを用いた二乗最大平均値差異（maximum mean discrepancy）（16.5 節）の不偏推定量であり、実画像の 2 集合に対する期待値は任意の n でゼロとなる。

```
f_real = fd_features(cnn, X[:25_000])  # five disjoint sets of 5,000
for n in (1000, 5000):
    sets = [f_real[i * n:(i + 1) * n] for i in range(5)]
    fd = np.array([float(d2l.fid_score(f, f_test[:n])) for f in sets])
    kid = np.array([float(d2l.kid_score(f, f_test[:n])) for f in sets])
    print(f'{n:,} real images, five sets: FD {fd.mean():.2f} '
          f'({fd.min():.2f} to {fd.max():.2f}), KID {kid.mean():.4f} '
          f'({kid.min():.4f} to {kid.max():.4f})')
```

```
1,000 real images, five sets: FD 0.88 (0.82 to 0.96), KID 0.0053 (-0.0070 to 0.0227)
5,000 real images, five sets: FD 0.17 (0.15 to 0.19), KID -0.0018 (-0.0051 to 0.0023)
```

予測は的中している。基準値は 1,000 枚で 0.9 弱、5,000 枚ではその 1/5 にあたる約 0.17 となる。KID にはこのようなバイアスはないが、ばらつきが大きい。1,000 枚では 5 つの値がゼロの両側に数百分の一の幅で散らばるため 1 つの値だけではほとんど判断できず、5,000 枚でも数千分の一の幅でばらつく。多くの論文が FD を報告しているため本章でもそれに倣い、数式による補正ではなく基準値との比較によってバイアスに対処する。

本章の残りの節では、必ず同じ画像枚数かつ同じ特徴量ネットワークのもとでのみ FD を比較する。5,000 枚の画像に対する 2 つの FD の差が、実画像 5 集合のばらつき幅である約 0.05 未満であれば両者は同等とみなす。1,000 枚ではそのばらつき幅は 0.15〜0.2 となる。以降の節では、完全な生成器が平均的に到達するスコアである**下限**（floor）として、各スコアの横にこの基準値を併記する。

生成サンプルの FD は約 0.8 であり、5,000 枚における基準値の約 5 倍高い。この差がモデルの誤差である。ところが、1,000 枚の実画像からなる 2 つの集合同士を比較しても、これと同程度のスコアが出る。画像枚数を伴わないフレシェ距離は無意味である。

基準となる目盛りが手に入ったところで、移動平均がどれほど効くのか、また 1,000 ステップのサンプラが 50 回のオイラーステップを上回るのかを確かめよう。1,000 ステップのサンプラは 17.3 節の祖先連鎖であり、Ho et al. (2020) の分散保持経路上で実行する。このネットワークは線形経路上で訓練されたものだが、問題はない。`make_pred` がネットワーク自身の経路上で同じ信号対雑音比（signal-to-noise ratio）を持つ時刻を見つけて再スケーリングするため、どの経路で訓練されたネットワークでも任意の経路に使えるからである。比較は対応のある形で行う。3 つのサンプラすべてを同じ初期ノイズから出発させることでノイズの引きによる差を排除し、これを 3 通りのノイズ抽出について繰り返す。

```
vp = d2l.GaussianPath.vp()
ancestral1000 = lambda pred, x: d2l.ancestral_sample(pred, x, vp, 1000)
```

共通する 3 つの初期ノイズシードにわたり、EMA 重みと `euler50`、生の最終重み `net` と `euler50`、および EMA 重みと `ancestral1000` のそれぞれから 5,000 枚の画像を生成し、`f_test[:5000]` に対するフレシェ距離を比較する。

```
for seed in range(3):  # three draws of the initial noise
    fd = [fd_and_agreement(cnn, generate(model, sampler=sampler, seed=seed),
```

```
                            f_test[:5000])[0]
        for model, sampler in ((ema, euler50), (net, euler50),
                               (ema, ancestral1000))]
    print(f'noise {seed}: Euler 50 average {fd[0]:.2f}, last weights '
          f'{fd[1]:.2f} ({fd[1] - fd[0]:+.2f}); chain with 1,000 steps '
          f'{fd[2]:.2f} ({fd[2] - fd[0]:+.2f})')
```

```
noise 0: Euler 50 average 0.82, last weights 1.36 (+0.55); chain with 1,000 steps 0.69 (-0.13)
noise 1: Euler 50 average 0.84, last weights 1.46 (+0.63); chain with 1,000 steps 0.66 (-0.18)
noise 2: Euler 50 average 0.83, last weights 1.36 (+0.53); chain with 1,000 steps 0.68 (-0.15)
```

このネットワークでは、いずれのノイズ抽出でも最終更新時の重みは移動平均より FD が 0.4〜0.6 悪化するのに対し、移動平均の FD はノイズ抽出間で数百分の一しか変動しない。どれだけ悪化するかは最後の更新が反復点の雲のどこにたまたま着地したかに左右され、別の訓練ランではまた別の場所に着地するだろう。直近の数千回の更新にわたる平均はそのような運にほとんど左右されないため、サンプリングには移動平均を用いるのである。また 1,000 ステップの連鎖は、やはりすべてのノイズ抽出で 50 回のオイラーステップより FD が 0.1〜0.2 低くなる。ただしこれは経路、サンプラ、ステップ数を同時に変更しているため、3 つの組み合わせの効果を測っているのであって個別の要因ではない。評価回数が 20 倍に増える割には控えめな改善であり、これ以降は 50 回のオイラーステップで評価を行う。

17.4.4 ガイダンス

17.4.4.1 スコア関数に対するベイズの定理

条件付きサンプルはすでに 90% 近くの割合で指定したクラスに属している。しかしそうでないものもあり、シャツ、プルオーバー、コートは見分けにくかった。本章の冒頭で触れたスライダが求めているのはそれ以上のもの、すなわち訓練済みネットワーク以上にラベルへ忠実に従うサンプラである。そのようなサンプラがどこから得られるかを知るため、ラベルがスコア関数にどのような影響を与えるかを見てみよう。時刻 t におけるベイズの定理は $p_t(\mathbf{x} \mid c) = p_t(\mathbf{x})\, p_t(c \mid \mathbf{x}) / p(c)$ と表される。ここで $p_t(c \mid \mathbf{x})$ は、ノイズの乗った画像 $\mathbf{x}$ が時刻 t においてクラス c のクリーンな画像から生じた確率であり、対数の勾配をとると $p(c)$ は消える。

$$\nabla_{\mathbf{x}} \log p_t(\mathbf{x} \mid c) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) + \nabla_{\mathbf{x}} \log p_t(c \mid \mathbf{x}). \tag{17.26}$$

クラスのスコア関数は、データ全体のスコア関数に分類器の対数確率の勾配を足し合わせたものである。**分類器ガイダンス**（classifier guidance）(Dhariwal and Nichol 2021) は、第 2 項を別途用意した分類器 $p_\phi(c \mid \mathbf{x}, t)$ で与え、そこに**ガイダンス強度**（guidance scale）γ を掛ける。

$$\tilde{\mathbf{s}}(\mathbf{x}, t) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) + \gamma \nabla_{\mathbf{x}} \log p_\phi(c \mid \mathbf{x}, t). \tag{17.27}$$

$\gamma = 1$ のとき、これはベイズの定理そのものであるが、Dhariwal and Nichol (2021) は $\gamma > 1$ とすることでサンプルが指定クラスにはるかに良く一致することを見出した。サンプラは勾配を $\mathbf{x}_t$ で評価するが、クリーンな画像用の分類器はそこでは意味のある勾配を返さないため、分類器はノイズ画像上で訓練されていなければならない。付録では 1 次元混合分布上でこの規則を扱い（38.6.4 節）、演習問題 2 では本節の画像に対してこれを実行する。

17.4.4.2 分類器なしガイダンス

分類器ガイダンスにはノイズ画像上で訓練された 2 つ目のネットワークが必要であり、そのネットワーク固有の誤差も入り込む。そこで Ho and Salimans (2022) は、代わりに Equation 17.26 を右辺から左辺へと読んだ。

$$\nabla_{\mathbf{x}} \log p_t(c \mid \mathbf{x}) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x} \mid c) - \nabla_{\mathbf{x}} \log p_t(\mathbf{x}). \tag{17.28}$$

分類器の勾配は条件付きスコア関数と無条件スコア関数の差にほかならず、ラベルドロップアウトを用いて訓練したネットワークはその両方を与える。この 2 つのスコア関数 $\mathbf{s}_c$ と $\mathbf{s}_\emptyset$ を Equation 17.27 に代入すると、ガイド付きスコア関数 $\tilde{\mathbf{s}} = \mathbf{s}_\emptyset + \gamma(\mathbf{s}_c - \mathbf{s}_\emptyset)$ が得られる。すなわち、分類器をネットワーク内部に暗黙的に組み込んだ分類器ガイダンスである。

本節のネットワークが予測するのはスコア関数ではなく速度であるが、固定した点 $(\mathbf{x}, t)$ において速度はスコア関数のアフィン関数である。線形経路上では、17.2 節の変換辞書から次式が成り立つ。

$$\mathbf{u}_t(\mathbf{x}) = -\frac{\mathbf{x} + t\,\nabla_{\mathbf{x}} \log p_t(\mathbf{x})}{1 - t}. \tag{17.29}$$

項 $-\mathbf{x}/(1-t)$ はラベルに依存せず、重み $1-\gamma$ と γ の和は 1 であるため、この項は線形結合をそのまま通過する。したがって、ガイド付き速度も 2 つの速度の同じ結合で表される。

$$\tilde{\mathbf{u}}(\mathbf{x}, t) = \mathbf{u}_\theta(\mathbf{x}, t, \emptyset) + \gamma\big(\mathbf{u}_\theta(\mathbf{x}, t, c) - \mathbf{u}_\theta(\mathbf{x}, t, \emptyset)\big). \tag{17.30}$$

`cfg` はこの規則を計算する。画像をラベル付きで 1 回、ヌルラベル付きで 1 回並べた 2 倍サイズのバッチ 1 つに対してネットワークを評価するため、ガイド付きサンプルは 1 ステップあたり 2 回の評価コストを要する。$\gamma = 1$ ではこの規則は条件付き速度場に帰着するため、`cfg` はバッチの後半を作らずに `conditional(net, c)` をそのまま返す。

```
def cfg(net, c, gamma):
    """Classifier-free guidance: (x, t) -> u + gamma (u_c - u), u unlabeled.

    gamma is a Python number; at gamma = 1 this is conditional(net, c)."""
    if gamma == 1:
        return d2l.conditional(net, c)
    null = torch.full_like(c, net.num_classes)
    def guided(x, t):  # one doubled batch: labeled, then unlabeled
        out_c, out = net(torch.cat([x, x]), torch.cat([t, t]),
                         torch.cat([c, null])).chunk(2)
        return out + gamma * (out_c - out)
    return guided
```

このアフィン性に基づく議論は、変換辞書のすべてのターゲットに対して成り立つ（演習問題 1）。なお Ho and Salimans (2022) はこの規則を $(1+w)\,\mathbf{u}_c - w\,\mathbf{u}_\emptyset$ と表記しているため、同論文の w は本節の $\gamma - 1$ にあたる。Figure 17.8 に本章を通じた例題上でのこの規則を示す。$\gamma > 1$ のとき、ガイド付き予測は無条件予測と条件付き予測を結ぶ直線上で、条件付き予測をさらに越えた先に位置する。

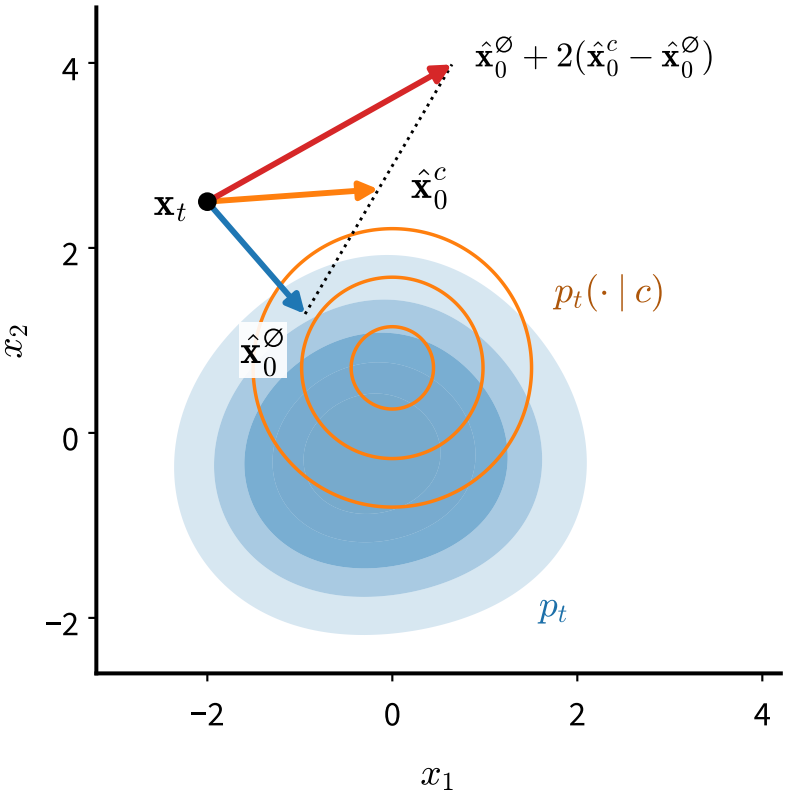


Figure 17.8: ある 1 つのノイズレベルにおける 3 モード混合分布上の分類器なしガイダンス。ノイズの乗った点 $\mathbf{x}_t$ から、データ全体の事後平均 $\hat{\mathbf{x}}_0^{\emptyset}$、上部のモードが与えられたときの事後平均 $\hat{\mathbf{x}}_0^{c}$、および $\hat{\mathbf{x}}_0^{\emptyset} + \gamma(\hat{\mathbf{x}}_0^{c} - \hat{\mathbf{x}}_0^{\emptyset})$（$\gamma = 2$ でのガイド付き予測であり、$\hat{\mathbf{x}}_0^{c}$ のさらに先に位置する）へと矢印が伸びている。等高線は全体のノイズ付き密度と上部モードのノイズ付き密度を表す。

γ がサンプルに与える影響を確かめるため、同じ初期ノイズから 7 通りの強度それぞれで 5,000 枚のサンプルを生成する。FD は実画像の分布をどれだけ満遍なく覆えているかを測り、クラス一致率はどれだけラベルに従っているかを測る。

```
guided = lambda gamma: lambda net, c: cfg(net, c, gamma)
samples = {gamma: generate(ema, guided(gamma))
           for gamma in (0.0, 1.0, 1.25, 1.5, 2.0, 3.0, 4.0)}
for gamma, imgs in samples.items():
    fd, agree = fd_and_agreement(cnn, imgs, f_test[:5000])
    print(f'gamma {gamma:.2f}: FD {fd:5.2f}, class agreement {agree:.3f}')
```

```
gamma 0.00: FD  2.74, class agreement 0.099
gamma 1.00: FD  0.82, class agreement 0.880
gamma 1.25: FD  0.84, class agreement 0.931
gamma 1.50: FD  1.13, class agreement 0.962
gamma 2.00: FD  2.05, class agreement 0.985
gamma 3.00: FD  4.07, class agreement 0.997
gamma 4.00: FD  5.86, class agreement 0.999
```

クラス条件付けを行わない場合（$\gamma = 0$）、クラス一致率はランダム予測の水準である 10% となり、FD は 2.7〜2.8 まで上昇する。これは $\gamma = 1$ における条件付き FD（≈ 0.8、クラス一致率は 90% 近く）の 3 倍を超える。$\gamma = 0$ での FD 上昇のうち、クラス比率の偏りに由来する分と各クラス内の画質劣化に由来する分を切り分けるため、次のコードセルでは Equation 17.25 の平均項 $\|\mu_p - \mu_q\|^2$ を比較し、生成画像のクラスヒストグラムを調べたうえで、5,000 枚の実訓練画像を $\gamma = 0$ のクラス比率に合わせて再サンプリングしたときの FD を計算する。

```
for gamma in (0.0, 1.0):
    f = fd_features(cnn, samples[gamma])
    mean_term = ((f.mean(0) - f_test[:5000].mean(0)) ** 2).sum()
    print(f'gamma {gamma:.1f}: mean term {mean_term:.2f} of FD '
          f'{d2l.fid_score(f, f_test[:5000]):.2f}')
share = lambda cls: torch.bincount(cls, minlength=10) / len(cls)
cls = cnn.head(fd_features(cnn, samples[0.0]).to(dev)).argmax(1)
for name, s_test, s_0 in zip(names, share(c_test), share(cls)):
    print(f'{name:>11}: test images {s_test:.3f}, gamma 0 {s_0:.3f}')
# real training images, 5,000 in the class mix of the gamma = 0 samples
counts, y_real = torch.bincount(cls, minlength=10).tolist(), y[:25_000].cpu()
mixed = torch.cat([torch.nonzero(y_real == k)[:n, 0]
                   for k, n in enumerate(counts)])
for name, f in (('uniform', f_real[:5000]), ('gamma 0', f_real[mixed])):
    mean_term = ((f.mean(0) - f_test[:5000].mean(0)) ** 2).sum()
    print(f'real images, {name} class mix: mean term {mean_term:.2f} of FD '
          f'{d2l.fid_score(f, f_test[:5000]):.2f}')
```

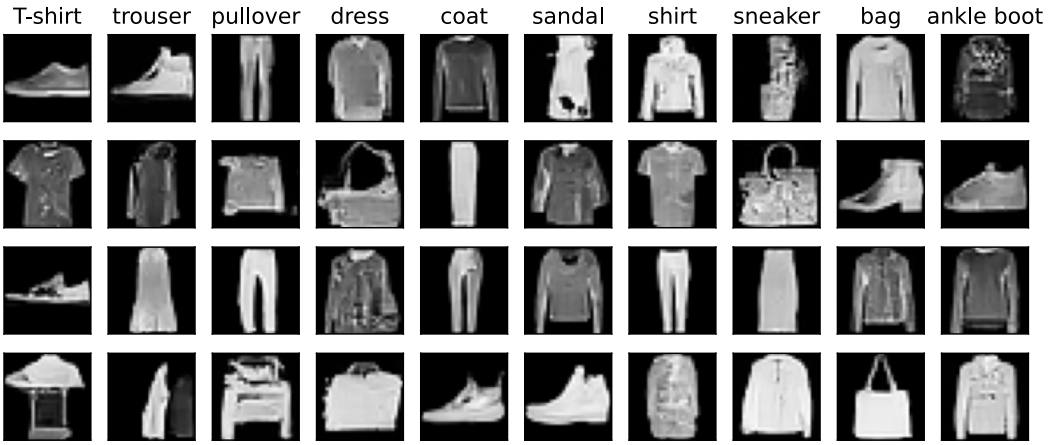
```
gamma 0.0: mean term 0.71 of FD 2.74
gamma 1.0: mean term 0.28 of FD 0.82
    T-shirt: test images 0.101, gamma 0 0.097
    trouser: test images 0.099, gamma 0 0.080
   pullover: test images 0.102, gamma 0 0.144
      dress: test images 0.102, gamma 0 0.097
       coat: test images 0.099, gamma 0 0.100
     sandal: test images 0.099, gamma 0 0.074
```

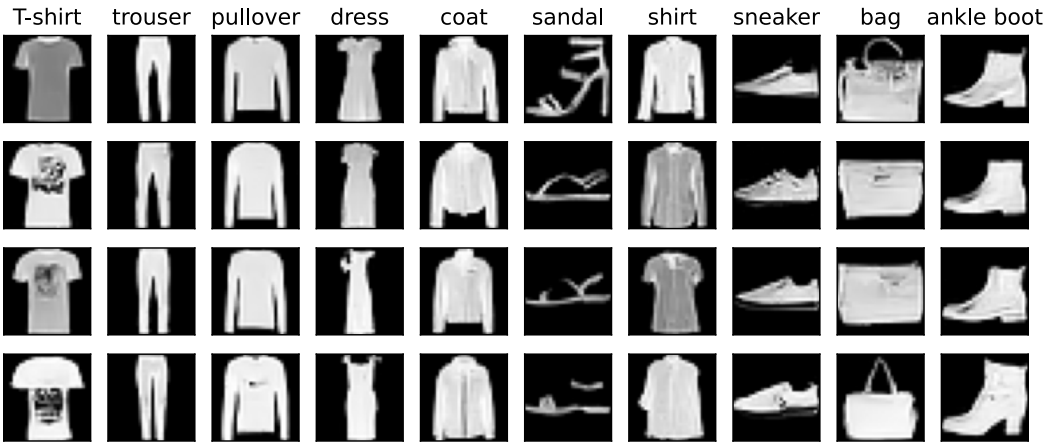
```
      shirt: test images 0.096, gamma 0 0.104
    sneaker: test images 0.103, gamma 0 0.095
        bag: test images 0.100, gamma 0 0.099
 ankle boot: test images 0.098, gamma 0 0.111
real images, uniform class mix: mean term 0.04 of FD 0.19
real images, gamma 0 class mix: mean term 0.13 of FD 0.38
```

無条件サンプラはプルオーバーを過剰に生成し（13%〜14%）、サンダルを過少にしか生成しないが（6%〜7%）、これと同じクラス比率の偏りを実画像に与えても FD は 0.2〜0.3（0.19 から 0.4〜0.5 へ）しか上がらず、FD の差 1.9（$\gamma = 1$ と $\gamma = 0$ の間）の 1/6 未満しか説明しない。残りの差が生じるのは、無条件の速度場 $\mathbf{u}_\theta(\mathbf{x}_t, t, \emptyset)$ が全 10 クラスを同時に扱いながらミニバッチのわずか 10% でしか訓練されておらず、形の崩れた衣類や曖昧な衣類をより多く生み出すためである。Equation 17.30 は $\mathbf{u}_c + (\gamma - 1)(\mathbf{u}_c - \mathbf{u}_\emptyset)$ と書き換えられるため、無条件ベースライン $\mathbf{u}_\emptyset$ の誤差は重み $1 - \gamma$ を伴ってガイド付きサンプルに入り込む。

γ を 1 より大きくすると、多様性と典型性のトレードオフが現れる。穏やかなガイダンス（$\gamma = 1.25$）では、実行ごとのノイズの範囲を超えて FD を悪化させることなく、クラス一致率が約 5 パーセントポイント上昇する（93%〜94%）。$\gamma \geq 1.5$ では、クラス一致率は 96% を超えて上昇する一方（99% を $\gamma = 3$ および 4 で上回り、実テスト画像に対する分類器の正解率よりも高くなる）、クラス内の多様性が失われて高コントラストな典型的プロトタイプへと崩壊していくにつれ、FD は 5.5〜6 まで着実に上昇する。

```
for gamma in (0.0, 4.0):
    d2l.show_gray(generate(ema, guided(gamma), n=40), 4, 10,
                  titles=names + [''] * 30)
```





実際のテキストから画像を生成するシステムでは強いガイダンス強度が使われる。人間のユーザーは 5,000 サンプルからなるバッチの共分散を評価するのではなく、個々の画像をプロンプトへの忠実さと視覚的な明瞭さで評価するからである。

17.4.4.3 ガイド付きサンプラが従う分布

サンプラは $\gamma = 1$ では指定クラスから、$\gamma = 0$ では全画像からサンプリングを行うが、それ以外の強度では Equation 17.30 の規則はいかなる分布から導出されたものでもない。それでも、対応する分布を探してみよう。ガイド付きスコア関数 $\nabla \log p_t(\mathbf{x}) + \gamma \nabla \log p_t(c \mid \mathbf{x})$ は、次の**傾斜**（tilted）密度の対数の勾配になっている。

$$\tilde{p}_t(\mathbf{x}) \propto p_t(\mathbf{x})\, p_t(c \mid \mathbf{x})^{\gamma} \propto p_t(\mathbf{x})^{1-\gamma}\, p_t(\mathbf{x} \mid c)^{\gamma}. \tag{17.31}$$

$\gamma > 1$ のとき、分類器の確率がべき乗されることで、分類器が最も確信を持つ画像へと確率質量が移動する。同値な見方として、無条件密度の一部で割ることにより、そのクラスが他クラスと共有する領域からサンプルが押し出されるとも言える。

この効果はどれほど大きいのだろうか。最小の例を考えれば閉じた形で答えが出る。クラス c の画像が $\mathcal{N}(\mu_c, s^2\mathbf{I})$ に従い、全画像を合わせた分布をより広がりの大きい 1 つのガウス分布 $\mathcal{N}(\mathbf{0}, S^2\mathbf{I})$（$S^2 > s^2$）で戯画化して近似するとしよう。10 クラスの混合分布はガウス分布ではないため、これは本節で最も大胆な仮定であるが、すべての答えを厳密に求められる仮定でもある。ガウス分布の対数は二次関数であるため、クリーンな画像の傾斜対数密度（定数差を除いて $\gamma \log p(\mathbf{x} \mid c) + (1-\gamma)\log p(\mathbf{x})$）もやはりガウス分布の対数となる。二次の項と一次の項をまとめると、逆分散 $1/v$ と平均は次式で与えられる。

$$\frac{1}{v} = \frac{\gamma}{s^2} - \frac{\gamma - 1}{S^2}, \qquad \text{mean} = \frac{\gamma v}{s^2}\,\mu_c. \tag{17.32}$$

クラスの逆分散が γ 回分足され、全画像の逆分散が $\gamma - 1$ 回分引かれる。$\gamma > 1$ のとき、この結果は $1/s^2$ を $(\gamma-1)(1/s^2 - 1/S^2) > 0$ だけ上回り、μ_c の前に掛かる係数も 1 を超える。すなわち、ガイダンスはクラスの広がりを狭めるとともに、全画像の平均から遠ざける方向へ押し出す。1 次元において $\mu_c = 1$、$s = 0.5$、$S^2 = 2$ とすると、強度 $\gamma = 4$ ではクラスの標準偏差が 0.5 から 0.26 へと縮小し、平均は 1 から 1.10 へと移動する。これが多様性と典型性のトレードオフを数式で表した姿である。

では、ガイド付きサンプラは少なくともこの狭まったガウス分布に到達するのだろうか。答えは否であり、その差は大きい。1 次元ではこの例のガイド付き速度は x に関してすべての t でアフィン関数となるため、確率フロー ODE はガウス分布をガウス分布へと運び、その平均と分散を厳密に積分できる。すると、サンプラは傾斜分布そのものすらオーバーシュートすることがわかる。$\gamma = 4$ で純粋なノイズから出発すると、最終的な平均は 1.23、標準偏差は 0.02 となり、クラスの幅は 20 倍以上縮小して傾斜ガウス分布の幅の 1/10 未満にまで潰れてしまう。

$t = 0$ における傾斜分布はサンプラが実際に従う分布ではない。なぜなら、$t > 0$ における傾斜密度はその傾斜分布にノイズを加えたものではなく、他のいかなるクリーンな分布にノイズを加えたものでもないからである。分散 σ_t^2 のガウスノイズを加えて得られる密度は、ノイズ自体より鋭い曲率を持つことができない。すなわち、$-\nabla^2 \log p_t$ のいかなる固有値も $1/\sigma_t^2$ を超えない（証明は 38.6.1.2 節にある）。線形経路上では $\sigma_t = t$ であり、2 つのガウス分布は分散 $(1-t)^2 s^2 + t^2$ と $(1-t)^2 S^2 + t^2$ を持つガウス分布のままである。$t = 0$ のときと同じ計算を行うと、$t = 1$ 付近において $\gamma > S^2/(S^2 - s^2)$（この例では $8/7$）である限り必ず傾斜密度がこの上界を破ることがわかる。$\gamma = 1.5$ では 0.47 より大きいすべての t で、$\gamma = 4$ では 0.24 より大きいすべての t で上界を破る（演習問題 6）。経路の大部分にわたり、ガイド付きサンプラはいかなるノイズ付き分布も持ち得ないほど強く引き寄せるベクトル場に従っており、その引き寄せがステップごとに累積していくのである。付録では 1 次元の 2 クラス混合分布でも同じ現象が生じることを確認している（38.6.4 節）。

問題の原因は高ノイズ領域にあるため、Kynkäänniemi et al. (2024) は高ノイズ領域でガイダンスを切り、中間のノイズレベル範囲でのみガイダンスを適用することで、多様性の損失を抑えつつクラス一致率の向上の大半を維持している。演習問題 3 では本節のモデルでこれを試す。一方、Karras, Aittala, Kynkäänniemi, et al. (2024) は基準側を変更する手法を提案している。すなわち、無条件予測の代わりに、より小規模または訓練の浅い条件付きモデルを用いる。その誤差は誘導対象のモデルの誤差と似ているため、そこから遠ざかる方向に外挿することで多様性ではなく誤差を取り除くことができる。

これらはいずれも計算コストが高い。上記のガイド付きサンプルはそれぞれ 2 倍サイズのバッチによる 50 ステップ、すなわち 100 回のネットワーク評価を要しており、また 1,000 ステップの連鎖も 50 ステップをわずかに上回る程度であった。果たして何回まで評価回数を減らせば十分なのか、それを問うのが 17.5 節である。

17.4.5 まとめ

ノイズ除去 U-Net は、パラメータの大半を保持しつつわずかな演算量で大域的文脈を捉える低解像度のボトルネック層と、鋭い空間的エッジを保持するフル解像度のスキップ接続を組み合わせ、すべての残差ブロックに時刻とクラスの埋め込みを注入する。重みの指数移動平均（EMA）からサンプリングすることで、Adam の最終反復点を用いる場合と比べてフレシェ距離が大幅に低下する。ただし、フレシェ距離自体には正の $O(1/n)$ の有限標本による統計的バイアスがあるため、常に同じサイズ n の互いに素な実データ集合間の基準下限と照らし合わせて読む必要がある。最後に、ランダムなラベルドロップアウトによって単一のネットワーク内で条件付き速度場と無条件速度場を訓練することで、分類器なしガイダンス $\tilde{\mathbf{u}} = \mathbf{u}_\emptyset + \gamma(\mathbf{u}_c - \mathbf{u}_\emptyset)$ による生成の誘導が可能になる。穏やかなガイダンス強度（$\gamma \approx 1.25$）は FD を損なわずにクラス一致率を高める一方、より大きな強度（$\gamma > 1.5$）では高ノイズ領域の傾斜スコア関数がいかなるガウス畳み込み密度よりも鋭く分散を収縮させるため、サンプル多様性と引き換えに典型性が強まる。

17.4.6 演習問題

1. **同値なガイダンス規則**。ネットワークの 2 つの速度が厳密であり、したがって 17.2 節の変換辞書によりスコア関数 $\nabla \log p_t(\mathbf{x} \mid c)$ と $\nabla \log p_t(\mathbf{x})$ のアフィン関数になっているとし、さらに分類器もノイズ画像の厳密なクラス事後分布 $p_t(c \mid \mathbf{x})$ であるとする。

1. 分類器ガイダンス Equation 17.27（強度 γ）と分類器なしガイダンス Equation 17.30（同じ γ）が同一のガイド付き速度を生成することを示せ。
2. 変換辞書のすべてのターゲット（ノイズ、クリーン画像、スコア関数、$\mathbf{v}$、および速度）は、固定した $(\mathbf{x}, t)$ において $a(\mathbf{x}, t) + b(t)\,\hat{\mathbf{x}}_0$ という事後平均 $\hat{\mathbf{x}}_0$ のアフィン関数で表される。いずれか 1 つのターゲットにガイダンス規則を適用してから変換した結果が、他の任意のターゲットに適用した結果と一致することを示せ。この議論は重み $1-\gamma$ と γ のどの性質を用いているか。
3. ネットワークが予測した $\hat{\mathbf{x}}_0$ を $[-1, 1]$ にクリップする場合でも、この規則はすべてのターゲットに対して同じ結果を与えるか。

2. [code] **クリーン画像分類器によるガイダンス。**●分類器ガイダンスにはノイズ画像の分類器が必要であるが、ネットワークによるクリーン画像の推定値に対してクリーン画像の分類器を適用することでも生成を誘導できる。
   1. 特徴量ネットワーク `cnn` のテスト画像に対する正解率を、線形経路上で $\mathbf{x}_t$（$t \in \{0.25, 0.5, 0.75\}$）までノイズを加えた場合について報告せよ。これらの正解率から、本文でノイズ画像上で訓練された分類器が必要だと述べた理由をどのように説明できるか。
   2. 代わりに無条件予測（`cfg(ema, c, 0.0)`）の事後平均 $\hat{\mathbf{x}}_0(\mathbf{x}_t)$ を分類器の入力として用い、無条件スコア関数に $\gamma \nabla_{\mathbf{x}_t} \log p_{\text{cnn}}(c \mid \hat{\mathbf{x}}_0(\mathbf{x}_t))$ を加えて変換辞書により速度へ変換せよ。50 回のオイラーステップを用い、$\gamma \in \{1, 3, 10\}$ で 1,000 枚の画像をサンプリングせよ。
   3. 生成したサンプルを `fd_and_agreement` でスコアリングせよ。このときクラス一致率はサンプラを誘導したのと同じネットワークで測定されている。別のシードで 2 つ目の `d21.FeatureCNN` を訓練し、それを用いてもサンプルのスコアを計算せよ。両者の比較から、同じ分類器を誘導と評価の両方に使うことについて何が言えるか。

3. [code] **限定区間でのガイダンス。** Kynkäänniemi et al. (2024) はノイズレベルのある区間内でのみガイダンスを適用し、それ以外では $\gamma = 1$ を用いる。同論文ではノイズレベルを $\bar{\sigma}_t = \sigma_t / \alpha_t$ で測り、ImageNet モデルの 1 つにおいて $0.28 < \bar{\sigma}_t \le 2.9$ が最良であることを見出した。
   1. 線形経路において、時刻 t のうち $\bar{\sigma}_t$ が 0.28 および 2.9 に等しくなるものを求めよ。
   2. 強度 γ を指定された区間内の t に対してのみ適用する `cfg` を実装せよ。$\gamma = 4$ において、高ノイズ区間、中間区間、低ノイズ区間のそれぞれでガイダンスを行ったときの 5,000 サンプルの FD とクラス一致率を計算し、全ステップでのガイダンスおよび $\gamma = 1$ の場合と比較せよ。
   3. クラス一致率の向上の大半をもたらすのはどの区間であり、FD の上昇の大半をもたらすのはどの区間か。その答えを、サンプルがクラスを確定させる時刻に関する演習問題 4（17.3 節）および本節の予測画像の並びの 3 行目と関連づけて説明せよ。なお、演習問題 5（38.6 節）では 1 次元の 2 クラスについて同じ問いを扱っている。

4. [code] **スキップ接続。** `UNet` から 2 つのスキップ接続を取り除き、`up1` と `up0` がアップサンプリングされた特徴量のみを受け取るようにして入力チャネル幅をそれぞれ `2 * ch` と `ch` に縮小した変種を実装せよ。
   1. 訓練を行う前に、どの時刻 t でスキップ接続が最も重要になり、どの時刻ではほとんど影響しないかを予測せよ。
   2. 両方のネットワークを 5,000 回の更新で訓練せよ。1,000 枚のテスト画像に対し、各ネットワークの速度が含意するクリーン画像 $\hat{\mathbf{x}}_0 = \mathbf{x}_t - t\,\hat{\mathbf{u}}$ の二乗誤差を $t \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ で計算し、両方の曲線をプロットせよ。
   3. 各ネットワークの 5,000 サンプルに対する FD を比較せよ。(b) の誤差曲線からその差を予測できるか。

5. [code] **予測画像のしきい値処理。**●線形経路上では、ガイド付き速度からクリーン画像 $\hat{\mathbf{x}}_0 = \mathbf{x}_t - t\,\tilde{\mathbf{u}}$ が定まる。Saharia et al. (2022) は、大きなガイダンス強度で生じる飽和したサンプルの原因を、データの値域を外れた予測 $\hat{\mathbf{x}}_0$ に求めている。

1. $\gamma \in \{1, 4, 16\}$ に対し、すべてのオイラーステップにおいて $\hat{\mathbf{x}}_0$ のピクセルのうち $[-1, 1]$ の外に出る割合を記録し、t に対してプロットせよ。経路上のどの領域で予測が値域を外れるか。また γ によってそれはどう変化するか。
2. **静的しきい値処理**はステップの前に $\hat{\mathbf{x}}_0$ を $[-1, 1]$ にクリップする。一方、**動的しきい値処理**は画像ごとに m を 1 と $|\hat{\mathbf{x}}_0|$ の 99.5 パーセンタイルのうち大きい方に設定し、$[-m, m]$ にクリップしてから m で割る。しきい値処理後の $\hat{\mathbf{x}}_0$ から速度を構成するようにオイラーステップを書き換え、両方の手法を実装せよ。
3. 画像集合の**飽和率**を、クランプ後に 1 に等しくなるピクセルの割合として定義する。$\gamma \in \{4, 8, 16\}$ において、3 つのサンプラの飽和率、FD、クラス一致率を比較せよ。

6. **ガウスクラスに対するガイダンス。**●本節のガウスクラスの設定、すなわちクラス c が $\mathcal{N}(\mu_c, s^2\mathbf{I})$ に従い、全画像が $\mathcal{N}(\mathbf{0}, S^2\mathbf{I})$（$S^2 > s^2$）に従い、線形経路上でノイズが加えられる場合を考える。
   1. 本文で示した $t = 0$ における傾斜ガウス分布の逆分散と平均を導出せよ。さらに、すべての t においてガイド付きスコア関数 $\gamma \nabla \log p_t(\mathbf{x} \mid c) + (1 - \gamma) \nabla \log p_t(\mathbf{x})$ がガウス分布のスコア関数になることを示してその分散を求め、その分散が t^2 を $t = 1$ 付近で下回る条件が $\gamma > S^2/(S^2 - s^2)$ のとき、かつそのときに限ることを示せ。
   2. 1 次元で $\mu_c = 1$、$s^2 = 0.25$、$S^2 = 2$ とすると、ガイド付き速度は x に関してすべての t でアフィン関数になる。平均と分散に対する ODE を書き下し、$\mathcal{N}(0, 1)$（$t = 0.999$）から $t = 0$ まで $\gamma \in \{1, 2, 4\}$ について積分して、本文の $\gamma = 4$ における終端値（平均 1.23、標準偏差 0.02）を確認せよ。各終端分布を $t = 0$ における傾斜ガウス分布と比較し、本文の上界を用いてその違いを説明せよ。
   3. 演習問題 5（16.6 節）の打ち切りトリックを生成器 $\mu_c + s\mathbf{z}$ に適用する手法や、$p(\mathbf{x} \mid c)^{1/T}$ から温度 $T < 1$ でサンプリングする手法もクラスの広がりを狭める。これら 3 つの手法のうち、クラスの平均も移動させるものはどれか。また、それはなぜか。

17.5 サンプラ、逆変換、編集

拡散サンプラの各ステップではスコアまたは速度のネットワークを 1 回評価するため、画像を生成する計算コストはステップ数に比例して増大する。17.4 節では、確率フロー ODE（probability-flow ODE）の 50 回のオイラーステップによって、1,000 ステップの祖先連鎖（ancestral chain）に近いフレシェ距離（Fréchet distance、FD）を 20 分の 1 のコストで達成した。17.2 節の訓練における回帰は、複数ステップにわたる同時分布ではなく、$\mathbf{x}_t$ の単一の時刻 t における周辺分布のみに依存するため、1 つの訓練済みネットワークで確率的および決定論的な逆過程の連続的な族をサポートできる (J. Song et al. 2021)。

本節では、***ノイズ除去拡散陰的モデル***（denoising diffusion implicit model、DDIM）の族を導出し、その決定論的なメンバー（$\eta = 0$）が $(\bar{\mathbf{x}}, \bar{\sigma})$ 座標系における確率フロー ODE 上のオイラー法と一致することを示す。さらに、1 次（オイラー法 / DDIM）、2 次（ホイン法）、および多段（アダムス・バッシュフォース法 / DPM-Solver++）のソルバを、さまざまなステップ予算と時間グリッドにわたって比較する。最後に、決定論的 ODE の可逆性を、画像からノイズへの逆変換（inversion）、潜在空間での補間、部分ノイズ付加による編集、およびゼロショットインペインティングに用いる。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.5.1 決定論的サンプラと確率的サンプラ

17.3 節の 2 つのサンプラである祖先連鎖とオイラー法は、一方は各ステップで新たなノイズを加え、もう一方はまったく加えないという違いはあるものの、同じ方法でネットワークの出力を読み取る。ガウス経路（Gaussian path）上の点 $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ において、ネットワークはクリーンな点の推定値 $\hat{\mathbf{x}}_0$ とノイズの推定値 $\hat{\epsilon}$ を与え、一方が決まればもう一方も次のように定まる。

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x}_t - \sigma_t \hat{\epsilon}}{\alpha_t}. \tag{17.33}$$

ネットワークが何を予測するように訓練されていようとも、`d2l.make_pred`（17.3 節）はそれを任意のノイズレベルでこの組を返す関数 `pred(x, alpha, sigma)` へと変換し、本節のサンプラはそれ以外の情報を一切参照しない。

17.5.1.1 DDIM 族とオイラー法

時刻 t とそれより前の時刻 $s < t$ をとる。t から s へのサンプラのステップは、$\mathbf{x}_t$ と同じ分布に従う点を $\mathbf{x}_s$ と同じ分布に従う点へと写す必要があり、回帰が課す制約はそれだけである。$\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ と表し、ノイズスケール $\tilde{\sigma}$ を 0 と σ_s の間で選んで、次のようにおく。

$$\mathbf{x}_s = \alpha_s \mathbf{x}_0 + \sqrt{\sigma_s^2 - \tilde{\sigma}^2}\,\epsilon + \tilde{\sigma}\,\xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.34}$$

ここで、ξ は ϵ と独立である。2 つのノイズ分散の和は σ_s^2 になるため、$\mathbf{x}_0$ が与えられたもとで点 $\mathbf{x}_s$ は任意の $\tilde{\sigma}$ に対して正しい分布を持つ。また、$\epsilon = (\mathbf{x}_t - \alpha_t \mathbf{x}_0)/\sigma_t$ であるから、各 $\tilde{\sigma}$ は $\mathbf{x}_t$ から $\mathbf{x}_s$ へのステップを定める。サンプラは、$\mathbf{x}_0$ と ϵ の代わりにネットワークの推定値を用いる。

$$\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sqrt{\sigma_s^2 - \tilde{\sigma}^2}\,\hat{\epsilon} + \tilde{\sigma}\,\xi. \tag{17.35}$$

このステップは、予測されたクリーンな点を予測されたノイズとともにレベル s へ移し、そのノイズの一部を新たなノイズで置き換える。J. Song et al. (2021) は、この新たなノイズの割合を強度 $\eta \geq 0$ によって次のように表している。

$$\tilde{\sigma} = \eta\, \sigma_s \sqrt{1 - \kappa^2}, \qquad \kappa = \frac{\sigma_s \alpha_t}{\alpha_s \sigma_t}. \tag{17.36}$$

ここで $\kappa^2 = \mathrm{SNR}_t / \mathrm{SNR}_s < 1$ であり、$\mathrm{SNR} = \alpha^2/\sigma^2$ である。$\eta = 1$ のとき、$\tilde{\sigma}^2$ は、$\mathbf{x}_s$ が $\mathbf{x}_t$ と $\mathbf{x}_0$ を与えられたときに 17.2 節のノイズ付加連鎖のもとで持つ分散である（演習問題 1）。このときステップは、真の値の代わりに推定値を用いてその連鎖の事後分布からサンプリングする。$\eta = 0$ では新たなノイズは一切入らず、ステップは決定論的になる。これが**ノイズ除去拡散陰的モデル**（DDIM）であり、密度の連鎖ではなくノイズからの写像によってサンプルを定義することから陰的（implicit）と呼ばれる。

`ddim_sample` は、`path.grid` のグリッド上で任意の予測器 `pred(x, alpha, sigma) -> (x0, eps)` を用いて DDIM 族を実行し、レベル間でループがサンプルを操作できるよう `t_end`（デフォルトではデータ）で停止する。

```
@torch.no_grad()
def ddim_sample(pred, x, path, steps, eta=0.0, *, t_start=1.0, t_end=0.0):
    """DDIM from x at t_start (1 is noise) to t_end; eta = 0 is deterministic.
```

```
    eta = 1 draws the posterior variance: not the ancestral sampler's."""
    a, s = path.grid(steps, t_start, t_end)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        ratio = s[i + 1] * a[i] / (a[i + 1] * s[i])     # kappa of the text
        var = (eta * s[i + 1]) ** 2 * (1 - ratio ** 2)   # fresh noise
        x = a[i + 1] * x0 + math.sqrt(s[i + 1] ** 2 - var) * eps
        if var > 0:
            x = x + math.sqrt(var) * torch.randn_like(x)
    return x
```

長い決定論的ストライドはどれほど大きな誤差を生むだろうか。分散保持（variance-preserving）経路上で、データ自体が標準ガウス分布に従う場合を考えよう。このとき $\mathbf{x}_t$ はあらゆる時刻で標準ガウス分布に従うため、厳密なサンプラはどの点も動かさないはずである。厳密な推定値は $\hat{\mathbf{x}}_0 = \alpha_t \mathbf{x}_t$ および $\hat{\epsilon} = \sigma_t \mathbf{x}_t$ となり、$t = 1$ から $t = 0$ への 1 回の決定論的ストライドは $\hat{\mathbf{x}}_0 = \alpha_1 \mathbf{x}_1$ を返す。$\alpha_1 \approx 0.0066$ であるため、点群全体がその平均へと潰れてしまう。厳密な軌跡に沿ってはノイズ推定値が変化するのに対し、このステップはストライド全体にわたってノイズ推定値を固定してしまうからである。

$\eta = 0$ のとき、このステップは $\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sigma_s \hat{\epsilon}$ となり、`ode_sample` が 17.3 節でとるステップに一致する。そこで導出した座標 $\bar{\mathbf{x}} = \mathbf{x}/\alpha_t$ と $\bar{\sigma}_t = \sigma_t/\alpha_t$ では、これは次の式で表される。

$$\bar{\mathbf{x}}_s = \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\hat{\epsilon}, \tag{17.37}$$

これは確率フロー ODE $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$ の 1 回のオイラーステップであり、上の比 κ は単に $\bar{\sigma}_s/\bar{\sigma}_t$ となる。したがって、あらゆるガウス経路において DDIM はオイラー法であり、$\eta = 0$ のとき同じレベル列上の `ddim_sample` と `ode_sample` は 1 つの関数を 2 つの座標系で計算しているにすぎない。38.6.4 節ではこの等価性とともに、2 つの極端な場合を証明する。すなわち、単一のデータ点では各軌跡に沿って $\hat{\epsilon}$ が一定になるため任意の長さのストライドで誤差なく積分される一方、ガウスデータはストライドごとに縮小する（Figure 38.25）。

17.5.1.2 新たなノイズと長いステップ

1 つのネットワークがあらゆる η に対応できるなら、どの強度を使うべきだろうか。$f = \tilde{\sigma}^2/\sigma_s^2$ とおくと、Equation 17.35 のステップは次のようになる。

$$\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sigma_s\Big(\sqrt{1-f}\,\hat{\epsilon} + \sqrt{f}\,\xi\Big), \qquad f = \eta^2\Big(1 - \frac{\mathrm{SNR}_t}{\mathrm{SNR}_s}\Big). \tag{17.38}$$

どのメンバーもクリーン推定値を同じようにレベル s に配置し、付随するノイズ分散のうち割合 f だけが新たなノイズとなる。細かいグリッド上では、f は $\eta = 1$ であっても小さいが、10 ステップのグリッド上では $\eta = 1$ のサンプラは各ステップでノイズの大半を入れ替える。

新たなノイズを加えることにはもっともな根拠がある。17.3 節の祖先ステップは、1 次近似において、現在の分布 p_t を不変に保ちつつ外れた点をそこへ引き戻すランジュバンステップと、確率フロー ODE のオイラーステップとに分解される。ノイズを注入して周辺分布を保つあらゆる逆過程も同様に分解される（38.5.5 節）。したがって、確率的サンプラは状態をモデルの分布へと引き戻し続けるため、あるステップで生じた誤差は次のステップで部分的に忘れ去られる可能性がある。一方、決定論的サンプラはすべての誤差を最後まで持ち越す。

同時に、新たなノイズに反対する同程度にもっともな根拠もある。ランジュバンによる引き戻しは、短いステップについての性質である。長いステップにわたる厳密な逆遷移はガウス分布からは程遠く（演習問題 3、17.3 節）、長い確率的ステップはそれ自体の誤差をつけ加えてしまう。少ない予算では決定論的サンプラが勝るはずであり、ステップが短くなったときに確率的サンプラがそれを追い抜くかどうかは、17.5.2.4 節で画像を用いて測定する。Figure 17.9 に、混合分布上でのこの族の両端を示す。

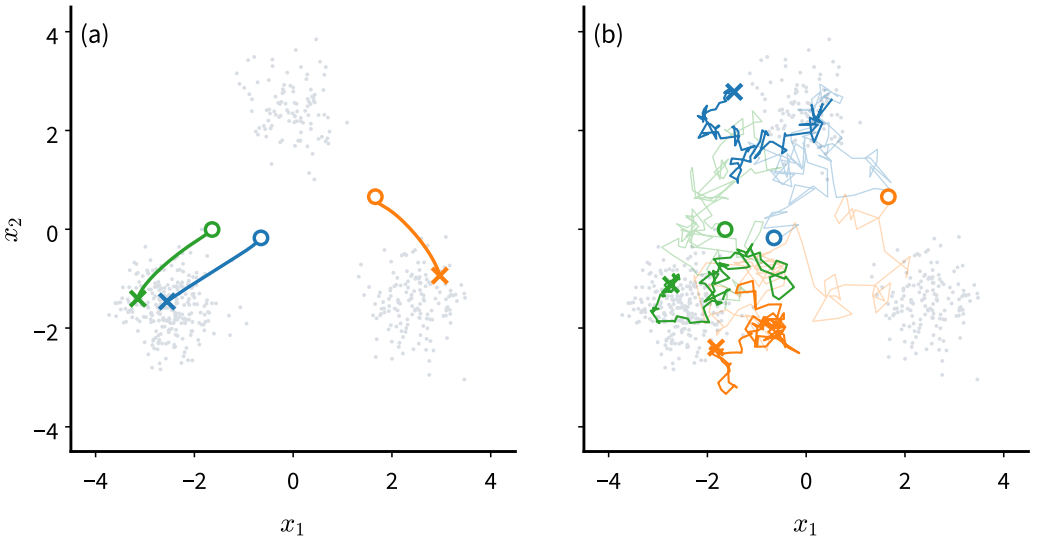


Figure 17.9: 厳密な予測器を用いた 3 モード混合分布上での DDIM 族の両端。分散保持経路上の 200 ステップで、$t = 1$ の 3 つの開始点（丸）から $t = 0$（バツ）まで進めた軌跡であり、訓練データを灰色で示す。(a) $\eta = 0$ では、各開始点は滑らかな曲線をたどってモードに入る。(b) $\eta = 1$ では経路がぎざぎざになり、ここでは 3 つのうち 2 つが (a) とは異なるモードに到達している。ノイズが支配的な各経路の前半は薄く描かれている。どちらも $t = 0$ では同じ分布を生成する。

2 つの確率的サンプラは明確に区別しなければならない。祖先連鎖は順ステップの分散 $\sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2$ を加えるのに対し、$\eta = 1$ はそれより係数 σ_s^2/σ_t^2 だけ小さい事後分散（posterior variance）を加える。どちらも同じ平均へ移動する。したがって、祖先連鎖はこの族のメンバーではない。保持するノイズと追加するノイズの分散の和が σ_s^2 を超えるからである。どちらの分散が正しいのだろうか。一般には、どちらも厳密ではない。データが単一の点であるときは $\mathbf{x}_t$ から $\mathbf{x}_0$ が一意に定まるため事後分散が厳密になり、分散保持経路上の標準ガウスデータに対しては順ステップの分散が厳密になる。座標あたりの分散が 1 未満でノイズより狭いデータでは、最良のガウス分散は両者の間に位置し、より広いデータでは順ステップの分散さえ超えることがある（演習問題 4）。1,000 ステップのグリッド上では、両方の分散がごく小さくなる最後の数ステップを除いてこの係数は 1 に近いため、どちらを選んでもほとんど差はないが、50 ステップでは大きな違いが生じる。

17.5.2 ステップ数の削減

1 次手法はステップサイズに比例する誤差を生じるため、ステップを間引くと予測可能な形で精度が低下する。まずはその低下量を厳密に計算できる設定で測定し、続いてより高次の手法やより適切なステップ配置によってどれだけ回復できるかを調べる。あらかじめ断っておくと、画像上で得られる数値は 1 つの小規模なモデル固有のものであり、別のモデルにはそのまま当てはまらない。しかし、その数値を導き出す手順はそのまま通用する。

17.5.2.1 通し例におけるステップの間引き

ここで 3 モード混合分布 `mix`（17.1 節）に戻ろう。この分布ではあらゆるレベルで事後平均（posterior mean）、ひいては厳密な組 $(\hat{\mathbf{x}}_0, \hat{\epsilon})$ を計算でき、`pred_exact` はその組をサンプラのインターフェースで包んだ関数である。20,000 個の開始点と同数のデータ点を抽出し、データの**広がり**（spread）、すなわち各点から最も近いモード中心までの平均二乗距離（17.1 節）を出力する。各モードは座標あたり 0.25 の分散を持つため、広がりは 0.5 をわずかに下回る。

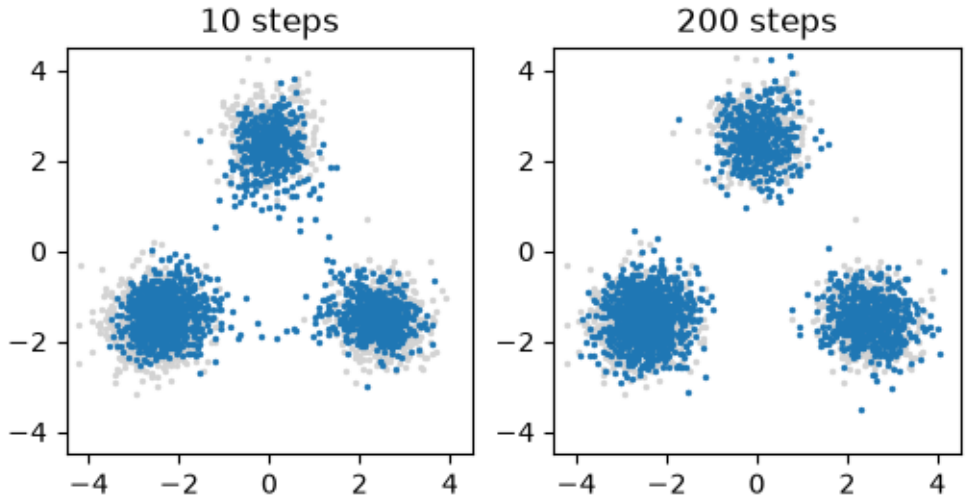
```
mix = d2l.GaussianMixture.three_modes()
vp, lin = d2l.GaussianPath.vp(), d2l.GaussianPath.linear()

def pred_exact(x, alpha, sigma):
    """The mixture's exact (x0, eps) at x = alpha x0 + sigma eps."""
    x0 = mix.posterior_mean(x, alpha, sigma)
    return x0, (x - alpha * x0) / sigma

torch.manual_seed(0)
z, data = torch.randn(20_000, 2), mix.sample(20_000)
print(f'data: spread {mix.summary(data)["spread"]:.3f}')
```

```
data: spread 0.477
```

厳密な予測器を用いる場合、同じ開始点からの 10 ステップのサンプルと 1,000 ステップのサンプルの差はすべてストライドの誤差であり、1 次手法ではステップ数を 2 倍にするごとに半減するはずである。分散保持グリッド上で 5〜1,000 ステップにわたって DDIM を実行し、1,000 ステップのサンプルまでの平均距離と広がりを出力する。$\eta = 0$ の広がりの横には、同じ開始点からの $\eta = 1$ および祖先連鎖の広がりも並べて示す。

```
samples = {n: d2l.ddim_sample(pred_exact, z, vp, n)
           for n in (5, 10, 20, 50, 200, 1000)}
spread = lambda pts: f'{mix.summary(pts)["spread"]:.3f}'
for n, x in samples.items():
    gap = (x - samples[1000]).norm(dim=1).mean()
    eta1 = d2l.ddim_sample(pred_exact, z, vp, n, eta=1.0)
    chain = d2l.ancestral_sample(pred_exact, z, vp, n)
    print(f'{n:4d} steps: distance to 1,000 steps {gap:.3f}, spread '
          f'{spread(x)} (eta = 1: {spread(eta1)}, ancestral: {spread(chain)})')
d2l.scatter_panels([samples[10][:2000], samples[200][:2000]],
                   ['10 steps', '200 steps'], data=data[:2000]);
```

```
   5 steps: distance to 1,000 steps 0.411, spread 0.309 (eta = 1: 0.321, ancestral: 0.386)
  10 steps: distance to 1,000 steps 0.210, spread 0.380 (eta = 1: 0.320, ancestral: 0.528)
  20 steps: distance to 1,000 steps 0.107, spread 0.420 (eta = 1: 0.342, ancestral: 0.520)
  50 steps: distance to 1,000 steps 0.043, spread 0.451 (eta = 1: 0.397, ancestral: 0.502)
 200 steps: distance to 1,000 steps 0.009, spread 0.469 (eta = 1: 0.452, ancestral: 0.480)
1000 steps: distance to 1,000 steps 0.000, spread 0.474 (eta = 1: 0.475, ancestral: 0.479)
```



距離は予測どおりステップ数を 2 倍にするごとに半減し、モード間が約 5 単位離れたスケール上で、5 ステップでは約 0.4、10 ステップでは 0.2、50 ステップでは 0.04 となる。また、誤差には一定の向きがある。10 ステップにおける決定論的サンプルの広がりは約 0.38 で、データの 0.47〜0.48 に対して小さく、左パネルの点群は背後にあるデータよりも目に見えて引き締まっている。1 回のストライドはノイズ推定値を開始時点の値に固定するため、ガウスデータに対しては常にノイズを取り除きすぎてしまう（演習問題 2）。そして、モードを識別できるほどノイズが小さくなると、混合分布の各モードは座標あたり分散 0.25 のガウスデータのように振る舞う。

確率的サンプラは両方向に外れる。10 ステップでは、$\eta = 1$ はモードをさらに収縮させて広がりを約 0.32 にする一方、祖先連鎖はデータ以上に広げて約 0.52 にする。ノイズを加える $\eta = 1$ の方が、ノイズを一切加えない $\eta = 0$ よりも狭く収まるのは一見すると意外である。DDIM 族のどのメンバーも、同じ予算 σ_s^2 を保持ノイズと新たなノイズに振り分ける。ガウスデータではクリーン推定値とノイズ推定値がどちらも $\mathbf{x}_t$ の正の定数倍になるため、保持ノイズはクリーン推定値と同じ向きに揃い、交差項を通じて広がりを押し上げる。新たなノイズは両者と独立であり、それ自体の分散を加えるだけでそれ以上の寄与はない。したがって、粗いグリッド上で $\eta = 1$ が行うように予算を保持ノイズから新たなノイズへ移すと、サンプルは狭くなる（演習問題 2 で証明する）。祖先連鎖は $\eta = 1$ と同じ量のノイズを保持しつつ予算を超える新たなノイズを加えるが、その超過分はノイズと同程度の幅を持つデータに適したものであり、ノイズより狭いモードには過剰である。細かいグリッド上では、3 つのサンプラはすべてデータの広がりに到達する。どちらの向きにもずれる統計量ではその大きさだけでサンプラを順位付けできないため、画像に対してはどちらの向きの誤差にもペナルティを課すスコアを用いる。

17.5.2.2 2 次手法とステップグリッド

オイラー法は 1 ステップの間ずっと傾きを固定するため、実行全体にわたる誤差はステップサイズに比例してしか減少しない。ホイン法（Heun's method、38.2 節）は、オイラーステップを試行ステップとして進み、その試行点で傾きをもう一度評価して、2 つの傾きの平均で移動する。

$$\begin{aligned}\bar{\mathbf{x}}' &= \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\hat{\epsilon}(\bar{\mathbf{x}}_t, \bar{\sigma}_t),\\ \bar{\mathbf{x}}_s &= \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\frac{\hat{\epsilon}(\bar{\mathbf{x}}_t, \bar{\sigma}_t) + \hat{\epsilon}(\bar{\mathbf{x}}', \bar{\sigma}_s)}{2}.\end{aligned}$$

その誤差はステップあたり 2 回の評価で、ステップサイズの二乗に比例して減少する。標準ガウス

データではノイズ推定値は $\hat{\epsilon} = \bar{\sigma}\bar{\mathbf{x}}/(1+\bar{\sigma}^2)$ であり、厳密なフローは $\bar{\mathbf{x}}$ に $\sqrt{(1+\bar{\sigma}_s^2)/(1+\bar{\sigma}_t^2)}$ を掛ける。$\bar{\sigma}_t = 1$ から $\bar{\sigma}_s = 0.5$ への厳密な倍率は 0.791 であるのに対し、オイラーステップは 0.75、ホイン法は 0.80 を与える。誤差はそれぞれ 0.041 と 0.009 であり、2 倍の評価回数で誤差は約 4 分の 1 になる。ただし、2 つのステップでは 2 回目の傾きをとることができない。$\bar{\sigma} = 0$ では推定すべきノイズが残っておらず、$\bar{\sigma} = \infty$ からのステップは無限に長いため $\hat{\mathbf{x}}_0 + \bar{\sigma}_s\hat{\epsilon}$ の形でしか書けない。したがって、$\bar{\sigma} = 0$ への最後のステップと $\bar{\sigma} = \infty$ からの最初のステップはオイラーステップとなり、`heun_sample` が使う評価回数は $2n-1$ 回（n ステップあたり）、線形経路上で純粋なノイズから始める場合は $2n-2$ 回となる。

2 回目の評価が有効かどうかは、ステップをどこに配置するかによって決まる。厳密な速度場を持つ 1 次元混合分布において、均等グリッド上のホイン法は約 40 回未満の評価回数ではオイラー法に劣る一方、データ側にステップを密に寄せたグリッド上ではあらゆる予算でオイラー法を上回り、同じ密な配置はオイラー法の誤差を 2 倍以上に悪化させる（38.7.4 節）。Karras et al. (2022) は、従来のサンプラが一括して決めていた手法の次数とステップ配置という 2 つの選択を切り離し、$\bar{\sigma}^{1/\rho}$ を等間隔に並べることでステップをデータ側に寄せた。

$$\bar{\sigma}_i = \left(\bar{\sigma}_{\max}^{1/\rho} + \frac{i}{n-1}\left(\bar{\sigma}_{\min}^{1/\rho} - \bar{\sigma}_{\max}^{1/\rho}\right)\right)^{\rho}, \qquad i = 0, \dots, n-1, \tag{17.39}$$

これに続けて最後に $\bar{\sigma} = 0$ へのステップをとり、$\bar{\sigma}_{\max} = 80$、$\bar{\sigma}_{\min} = 0.002$、$\rho = 7$ と設定する（画素値が $[-1, 1]$ の画像の場合）。`make_pred` が任意のレベルをネットワーク固有の時刻へと変換するため、グリッドは訓練経路とは独立したサンプラ側の選択肢となる。`edm_grid` はこれらのレベルを返し、`heun_sample` は任意のレベル列に沿ってホインステップを進める。ここでは 3 つの選択肢、すなわち分散保持経路上で時間に関して均等なグリッド、線形経路上で時間に関して均等なグリッド、および Karras らのグリッド（論文 **Elucidating the Design Space of Diffusion-Based Generative Models** にちなんで EDM グリッドと呼ぶ）について、それぞれの 10 ステップグリッドを出力する。

```
def edm_grid(steps, sigma_max=80.0, sigma_min=0.002, rho=7):
    """Karras et al.'s levels sigma_bar: even in sigma_bar^(1/rho), then 0."""
    hi, lo = sigma_max ** (1 / rho), sigma_min ** (1 / rho)
    return [(hi + i / (steps - 1) * (lo - hi)) ** rho
            for i in range(steps)] + [0.0]

@torch.no_grad()
def heun_sample(pred, x, sigma_bars):
    """Heun on d x_bar / d sigma_bar = eps: an Euler trial, then the average
    of the two slopes; from x at the first level to data, as ode_sample."""
    r0 = sigma_bars[0]
    xb = x if math.isinf(r0) else x * math.sqrt(1 + r0 ** 2)  # x / alpha
    for r_t, r_s in zip(sigma_bars[:-1], sigma_bars[1:]):
        x0, eps = d2l.pred_bar(pred, xb, r_t)
        x_next = x0 + r_s * eps                       # the Euler step
        if r_s > 0 and not math.isinf(r_t):           # no slope at 0 or inf
            x_next = xb + (r_s - r_t) * (
                eps + d2l.pred_bar(pred, x_next, r_s)[1]) / 2
        xb = x_next
    return xb

for name, levels in (('VP', vp.sigma_bars(10)), ('linear', lin.sigma_bars(10)),
                     ('EDM', d2l.edm_grid(10))):
```

```
    print(f'{name:7s}', ' '.join(f'{r:7.3g}' for r in levels))
```

```
VP          152    58.8    25.1    11.8     6.1    3.41    2.03    1.23    0.72    0.34       0
linear      inf       9       4    2.33     1.5       1   0.667   0.429    0.25   0.111       0
EDM          80    42.4    21.1    9.72    4.07     1.5    0.47   0.117  0.0204   0.002       0
```

時間に関して均等なグリッドも、$\bar{\sigma}$ で見るとまったく異なる姿をしている。分散保持グリッドは 10 ステップのうち 4 ステップを 152 から 6.1 の間に費やし、0.34 からデータへ到達する。線形グリッドは 9 より上を無限大からの 1 ステップで通過し、4 から 0.11 の間に 7 ステップを費やす。EDM グリッドは 80 から始まり、同じく 4 より上に 4 ステップを費やしたあと、最後の 3 つのレベルを 0.12 未満に密集させる。どの配分がモデルに適しているかはレベルの数値だけからは読み取れないため、以下で画像を用いて測定する。

17.5.2.3 指数積分器と多段ソルバ

ホイン法で 2 次精度を得るには、ステップあたり 2 回目の評価が必要になる。より低コストな道筋はこれまで使ってきた座標系から出発するものであり、実はすでにその道筋の上にいたのである。$\mathbf{x}$ 座標では、確率フロー ODE には $\mathbf{x}$ に関して線形でネットワークに一切依存しない項が含まれている。たとえば分散保持経路では、ドリフトはスコア関数が何であれ $\mathbf{x}$ を速度 $\frac{1}{2}\beta(t)$ で縮小させる。ここで $\beta(t)$ はノイズスケジュールの変化率である。この線形部分の厳密解は指数減衰、ここでは係数 α_t であり、α_t で割ることでこの項が消去される。残る $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$ にはネットワークの項しか含まれない。線形部分を厳密に積分し、ネットワークの項だけを近似するのが**指数積分器**（exponential integrator）の考え方である (Zhang and Chen 2023; Lu, Zhou, et al. 2022)。ステップにわたって $\hat{\epsilon}$ を一定に保つ DDIM はその 1 次のメンバーであり、したがって DDIM をオイラー法に見せた座標変換こそがこれらのソルバの中心的なアイデアだったのである。より高次のメンバーは、ステップ内で $\hat{\epsilon}$ をどう変化させるかという点だけが異なる。DPM-Solver (Lu, Zhou, et al. 2022) は、$\log\bar{\sigma}$ におけるステップの中点で 2 回目の評価を行う（演習問題 3）。

この 2 回目の評価は省くことができる。直前のステップですでに通過したばかりのレベルにおける $\hat{\epsilon}$ が評価済みであるため、**多段**（multistep）法はそれを再利用し、直近 2 回の推定値から現在のステップの中点へと $\hat{\epsilon}$ を線形外挿する。現在のステップ幅を $\Delta = \bar{\sigma}_s - \bar{\sigma}_t$、直前のステップ幅を Δ'、それぞれの開始点における推定値を $\hat{\epsilon}_t$、$\hat{\epsilon}'$ とすると、2 次のアダムス・バッシュフォース（Adams–Bashforth）ステップは次の式で与えられる。

$$\bar{\mathbf{x}}_s = \bar{\mathbf{x}}_t + \Delta\Big(\hat{\epsilon}_t + \frac{\Delta}{2\Delta'}(\hat{\epsilon}_t - \hat{\epsilon}')\Big). \tag{17.40}$$

ノイズ推定値がステップ間で変化しないとき、この更新は DDIM に一致し、変化するときはステップあたり 1 回の評価のままその変化に追随して補正を行う。最初のステップには直前の推定値がないためオイラーステップとなり、$\bar{\sigma} = \infty$ から始めた直後のステップでも Δ' が無限大になるため同様にオイラーステップとなる。DPM-Solver++(2M) (C. Lu et al. 2025) は同じ考え方をクリーン画像の推定値に適用し、$\log\bar{\sigma}$ 上で外挿する手法であり、10〜25 回の評価回数ではこれとその派生手法が標準的な選択肢となっている。`multistep_sample` は任意のレベル列に対して上のステップを実行する。

```
@torch.no_grad()
def multistep_sample(pred, x, sigma_bars):
    """Second-order Adams-Bashforth on d x_bar / d sigma_bar = eps: one
    evaluation per step, the slope extrapolated from the last two."""
    r = sigma_bars
```

```
    xb = x if math.isinf(r[0]) else x * math.sqrt(1 + r[0] ** 2)  # x / alpha
    eps_prev = 0
    for k in range(len(r) - 1):
        x0, eps = d2l.pred_bar(pred, xb, r[k])
        c = ((r[k + 1] - r[k]) ** 2 / (2 * (r[k] - r[k - 1]))
             if k > 0 and math.isfinite(r[k - 1]) else 0.0)  # 0: Euler
        xb = x0 + r[k + 1] * eps + c * (eps - eps_prev)
        eps_prev = eps
    return xb
```

17.5.2.4 画像における評価回数と品質

画像上では、速度場は学習によって求める。ここでは 17.4 節のクラス条件付き U-Net（約 120 万個のパラメータ）を、線形経路の速度に対して 10,000 回の更新で訓練する（GPU 1 基で数分程度）。ラベルの 10 分の 1 はヌルラベルに置き換え、サンプリングには重みの移動平均を用いる。本節の残りの実験はすべてこの 1 つのネットワークで行う。ガイダンスなしのサンプラ（以下の $\gamma = 1$）はすべてのクラスを均等な頻度で生成し（サンプル i に i を 10 で割った余りのクラスを指定）、ガイダンス付きのサンプラ（$\gamma > 1$）はヌルラベルもあわせて評価し、逆変換と編集では各画像にそれぞれのラベルを与える。

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()
loss = d2l.regression_loss(lin, 'velocity')

def batch_fn():
    i = torch.randint(0, len(X), (128,), device=X.device)
    drop = torch.rand(128, device=X.device) < 0.1   # label dropout
    return X[i], torch.where(drop, 10, y[i])        # 10 is the null label

def loss_fn(net, batch):  # the regression loss of the labeled field
    x0, c = batch
    return loss(d2l.conditional(net, c), x0)

torch.manual_seed(0)
net = d2l.UNet(ch=64, num_classes=10).to(X.device)
net, ema, _ = d2l.train_regression(net, loss_fn, batch_fn, 10_000)
```

1,000 枚のサンプルを、1,000 枚のテスト画像に対する `d2l.FeatureCNN` の特徴量空間（17.4 節）におけるフレシェ距離（FD）で評価する。このサンプル数では、実画像の集合どうしでも互いに 1 をわずかに下回るスコアとなり、画像の抽出ごとにスコアは 0.15〜0.2 ほど変動する（17.4.3.2 節）。そこで各測定は 1,000 枚のノイズ画像の 3 回の抽出に対して繰り返し、すべての比較が対になるよう全サンプラで同じノイズを共有したうえで、平均値に範囲の半分（半範囲）をプラスマイナスした形で報告する。フロア（下限値）は、テスト画像を別の 3 組の実画像集合と比較して求める。半範囲内に収まる差は実質的な差ではない。また `fd_runs` は次項で使用するガイダンス強度 γ も引数にとり、$\gamma = 1$ は通常の条件付き速度場を表す。

```
cnn = d2l.fashion_feature_cnn(X, y)
features = lambda imgs: d2l.fd_features(cnn, imgs)
f_test = features(X_test[:1000])
floors = [d2l.fid_score(features(X_test[k:k + 1000]), f_test)
          for k in (1000, 2000, 3000)]
```

```
labels = torch.arange(1000, device=X.device) % 10   # sample i has class i % 10
gen = torch.Generator(X.device).manual_seed(1)
noises = [torch.randn(1000, 1, 28, 28, device=X.device, generator=gen)
          for _ in range(3)]                        # shared by every sampler

def fd_runs(sampler, n, gamma=1.0):
    """FD of sampler(pred, noise, n) on each noise draw, field cfg(gamma)."""
    pred = d2l.make_pred(d2l.cfg(ema, labels, gamma), lin, 'velocity')
    return [d2l.fid_score(features(sampler(pred, z, n).clamp(-1, 1)), f_test)
            for z in noises]

def mean_range(fds):  # the mean, plus or minus half the range of the draws
    return f'{np.mean(fds):5.2f}±{(max(fds) - min(fds)) / 2:.2f}'

print('floor, 1,000 real images against 1,000 others: FD', mean_range(floors))
```

```
floor, 1,000 real images against 1,000 others: FD  0.86±0.15
```

フロアは約 0.9 である。最初の表は、上で未解決のまま残した問いに決着をつけるため、時間に関して均等なステップを持つ分散保持経路上で比較を行う。この設定では、$\eta = 0$ の DDIM は先ほど出力した最初のグリッド上のオイラー法に一致する。もし決定論的サンプラが最後まで持ち越す誤差をノイズが補正するのであれば、ステップが短くなるにつれて $\eta = 1$ と祖先連鎖は $\eta = 0$ に迫り、1,000 ステップに達する前に追い抜くはずである。

```
vp_samplers = {  # the DDIM family and the ancestral chain, even steps on VP
    'DDIM, eta = 0': lambda p, z, n: d2l.ddim_sample(p, z, vp, n),
    'DDIM, eta = 1': lambda p, z, n: d2l.ddim_sample(p, z, vp, n, eta=1.0),
    'ancestral': lambda p, z, n: d2l.ancestral_sample(p, z, vp, n)}
steps = (10, 20, 50, 100, 200, 1000)
torch.manual_seed(2)  # the fresh noise of the stochastic samplers
fd_vp = {name: [fd_runs(f, n) for n in steps]
         for name, f in vp_samplers.items()}
print('steps        ', ''.join(f'{n:>11d}' for n in steps))
for name, row in fd_vp.items():
    print(f'{name:13s}', ''.join(f'{mean_range(v):>11s}' for v in row))
```

```
steps                10         20         50        100        200       1000
DDIM, eta = 0   5.41±0.32  2.95±0.19  1.76±0.12  1.45±0.11  1.33±0.10  1.32±0.10
DDIM, eta = 1   7.81±0.23  4.90±0.16  3.06±0.09  2.32±0.06  1.98±0.08  1.52±0.05
ancestral      28.66±0.26 19.22±0.63  7.85±0.10  3.45±0.14  1.78±0.13  1.34±0.03
```

両者は差を縮めるものの、決して追い抜くことはない。200 ステップまでは、どの予算でも $\eta = 0$ のスコアが $\eta = 1$ より 3 分の 1 からほぼ半分低く、半範囲を大きく超えて優位である。1,000 ステップでは 3 手法とも約 1.3〜1.5 の間に収まり、互いの差は約 0.2 以内となる。このモデルでは新たなノイズが役立つことは一度もなかった。どの予算でも確率的サンプラが半範囲を超えて $\eta = 0$ を上回ることはなく、1,000 ステップでも祖先連鎖は並ぶにとどまる。100 ステップ以下では祖先連鎖が最も大きく劣り、50 ステップで約 8〜11 と、10 ステップの DDIM（5 強）よりも悪い。順ステップの分散はノイズと同程度の幅を持つデータには適しているが、衣類画像はノイズが小さくなればほぼ確定するため、過剰な分散は斑点状のノイズとして現れるはずである。次項末尾の画像列でそれを確認する。DDIM 自体は 100 ステップ以降の変化が約 10 分の 1 にとどまり、50 ステップの時点で 20 分の 1 の評価回数ながら 1,000 ステップの連鎖から約 3 分の 1 以内に迫る。これが J. Song et al. (2021) の中心的な結果である。

新たなノイズに関するこの結論はこのモデル固有のものであり、大規模なモデルにそのまま当てはめてはならない。64×64 ピクセルの ImageNet では、Karras et al. (2022) が各ホインステップの前に調整された少量のノイズを加えることで決定論的サンプラを改善できることを見いだしたほか、Y. Song, Sohl-Dickstein, et al. (2021) の予測子・修正子（predictor-corrector）サンプラはソルバのステップ間にランジュバンステップを挿入して同様の効果を得ており、Restart サンプリング (Xu et al. 2023) は大きな予算で最も効果を発揮する。その仕組みは、上で述べたランジュバンによる引き戻しである。複雑なデータを扱う大規模モデルは後続のノイズでなお補正可能な誤差を生じるため、十分なステップ数があればノイズによる補正効果がノイズ自体の加える分散を上回るのである。

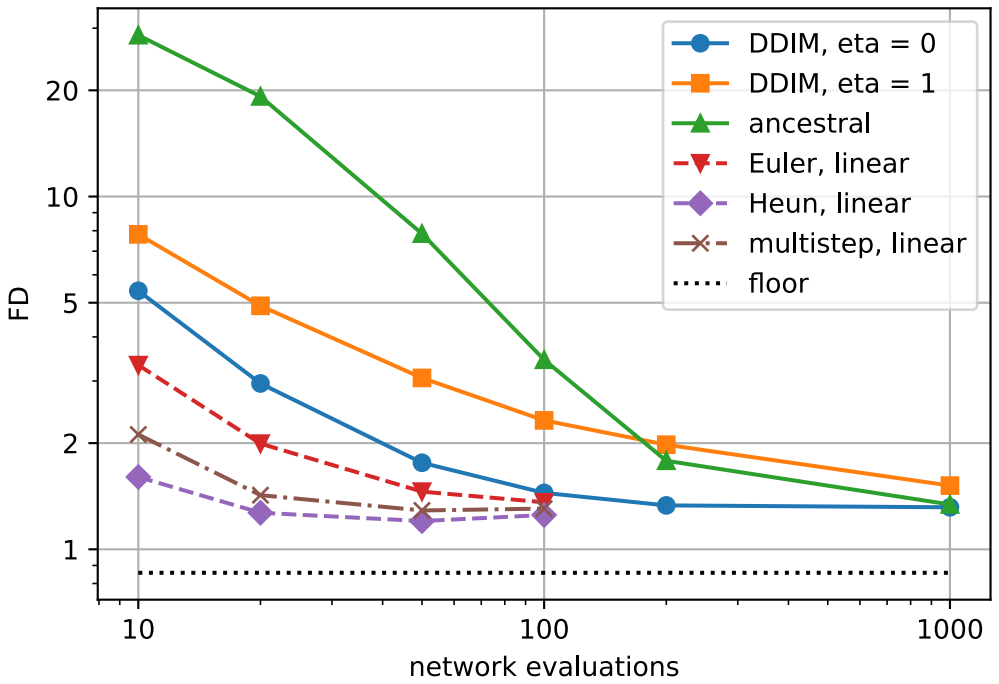
2 つ目の表は、サンプルの実質的なコストであるネットワーク評価回数で比較する。オイラー法、ホイン法、および多段サンプラを、線形グリッド、$\rho = 7$ の Karras らの EDM グリッド、そしてデータ側から高ノイズ側へとレベルを移した $\rho = 3$ の EDM グリッドという 3 つのグリッド上で、約 10〜100 回の評価回数にわたって実行する。ホイン法は約半分のステップ数をとり、有限のレベルから始まる EDM グリッド上では予算より 1 回少ない評価回数となる。

```
grids = {'linear': lin.sigma_bars, 'EDM': d2l.edm_grid,
         'EDM, rho = 3': lambda n: d2l.edm_grid(n, rho=3)}
solvers = {'Euler': d2l.ode_sample, 'Heun': d2l.heun_sample,
           'multistep': multistep_sample}
budgets = (10, 20, 50, 100)  # network evaluations per sample

def on_grid(solver, grid):  # a sampler (pred, noise, steps)
    return lambda p, z, n: solver(p, z, grid(n))

def num_steps(s, g, b):  # Heun skips its second slope only at sigma = inf, 0:
    # b // 2 + 1 steps on the linear grid (b evaluations, the first level is
    # infinite), b // 2 on the EDM grids (b - 1)
    return (b // 2 + (g == 'linear')) if s == 'Heun' else b

fd_ode = {(s, g): [fd_runs(on_grid(solver, grid), num_steps(s, g, b))
                   for b in budgets]
          for s, solver in solvers.items() for g, grid in grids.items()}
print('evaluations            ', ''.join(f'{b:>11d}' for b in budgets))
for (s, g), row in fd_ode.items():
    print(f'{s + ", " + g:23s}', ''.join(f'{mean_range(v):>11s}' for v in row))
```

```
evaluations                       10          20          50         100
Euler, linear              3.33±0.25   1.99±0.15   1.46±0.11   1.36±0.10
Euler, EDM                 6.81±0.38   3.14±0.22   1.86±0.15   1.58±0.14
Euler, EDM, rho = 3        9.74±0.47   4.02±0.25   2.02±0.16   1.62±0.14
Heun, linear               1.60±0.12   1.27±0.07   1.20±0.08   1.25±0.09
Heun, EDM                 11.71±0.29   1.69±0.08   1.32±0.10   1.37±0.11
Heun, EDM, rho = 3        84.52±0.45   8.10±0.26   1.47±0.11   1.39±0.11
multistep, linear          2.12±0.17   1.42±0.11   1.29±0.10   1.30±0.10
multistep, EDM             4.50±0.28   1.96±0.16   1.46±0.12   1.41±0.12
multistep, EDM, rho = 3    6.56±0.35   2.37±0.17   1.46±0.12   1.39±0.12
```

2 つの表はどこで交わるだろうか。次のプロットは、最初の表のサンプラと線形グリッド上の 3 つのソルバを、ネットワーク評価回数という 1 つの横軸上にまとめたものである。

```
curves = {**{name: (steps, row) for name, row in fd_vp.items()},
          **{f'{s}, linear': (budgets, fd_ode[s, 'linear']) for s in solvers}}
d2l.plot([x for x, _ in curves.values()] + [[10, 1000]],
         [[np.mean(v) for v in row] for _, row in curves.values()]
         + [[np.mean(floors)] * 2], 'network evaluations', 'FD',
         legend=list(curves) + ['floor'], xscale='log', yscale='log',
         figsize=(6, 4), fmts=('o-', 's-', '^-', 'v--', 'D--', 'x-.', 'k:'))
```



2 次手法は、モデルの漸近的な FD プラトー（≈ 1.1〜1.3、有限サンプルのフロアより約 0.3 高い水準）に最も早く到達する。線形グリッド上のホイン法は 20 回の関数評価で ≈ 1.2〜1.3 に達し、2 次多段ソルバは 50 回までに到達するのに対し、オイラー法は 50〜100 回を要する。

ステップの配置はソルバの次数と同じくらい重要である。10 回の評価では、オイラー法のスコアは ≈ 3.2（線形グリッド）、≈ 5（VP グリッド）、≈ 6.5（$\rho = 7$ の EDM グリッド）である一方、ホイン法は線形グリッド上で ≈ 1.6 であるのに対し EDM 上では 12〜15 となる。ρ を 3 に下げてステップをデータ側から高ノイズ側へ移すと、10 回および 20 回評価での性能はさらに悪化する。$\bar{\sigma} \approx 4$（ピクセルの標準偏差 ≈ 0.7 の数倍）より上では、ノイズ画像はほぼ純粋なガウスノイズであり、軌跡もほぼ直線になる。線形グリッドは $\bar{\sigma} > 4$ の領域を 2 ステップ（$\infty \to 9 \to 4$）で通過するのに対し、10 ステップの EDM グリッドはそこに 4 ステップを費やし、衣類の構造が形成される中間領域に十分なステップを残せない。

17.5.2.5 積分誤差、ガイダンス、サンプラの選択

ここまでは FD のみでサンプラを評価してきたが、FD が評価するのはサンプル集合全体である。個々のサンプルが、同じノイズ画像から速度場を厳密に積分したときに到達する点からどれだけ離れているかは FD からはわからない。FD では捉えきれないソルバの誤差を見るため、すべてのノイズ画像を 200 回評価のホイン法で積分したものを参照解とし、それに対する各ソルバのピク

セルあたりの二乗平均平方根（root mean square、RMS）距離を測定する。

ガイダンスは積分対象の速度場を変化させるため、この測定を 2 つの速度場に対して行う。分類器なしガイダンス（classifier-free guidance、17.4.4.2 節）はラベルなし予測からラベル付き予測へと $\tilde{\mathbf{u}} = \mathbf{u}_\emptyset + \gamma(\mathbf{u}_c - \mathbf{u}_\emptyset)$ のように外挿するため、$\gamma = 1$ が通常の条件付き速度場であり、ガイダンス付きの評価は 1 回につきネットワークの順伝播を 2 回要する。2 つの構成要素の外側へ外挿する速度場は軌跡に沿った変化も速くなるはずであり、$\gamma = 2$ ではどのソルバもそれに追随するためにより多くの評価回数を必要とすると予測される。

```
  def endpoints(solver, n, gamma):
      """Samples of a solver with n steps on the linear grid, per noise draw."""
      pred = d2l.make_pred(d2l.cfg(ema, labels, gamma), lin, 'velocity')
      return [solver(pred, z, lin.sigma_bars(n)).clamp(-1, 1) for z in noises]

  for gamma in (1.0, 2.0):
      refs = endpoints(d2l.heun_sample, 101, gamma)    # 200 evaluations
      fds = [d2l.fid_score(features(x), f_test) for x in refs]
      print(f'gamma = {gamma:.0f}, reference: FD {mean_range(fds)}; '
            'FD and RMS distance to it')
      print('evaluations', ''.join(f'{b:>18d}' for b in (10, 20, 50)))
      for s, solver in solvers.items():
          cols = []
          for b in (10, 20, 50):
              xs = endpoints(solver, b // 2 + 1 if s == 'Heun' else b, gamma)
              fds = [d2l.fid_score(features(x), f_test) for x in xs]
              rms = np.mean([((x - r) ** 2).mean().sqrt().item()
                             for x, r in zip(xs, refs)])
              cols.append(f'  {mean_range(fds)} {rms:.3f}')
          print(f'  {s:9s}', ''.join(cols))
```

```
gamma = 1, reference: FD  1.29±0.09; FD and RMS distance to it
evaluations                 10                 20                 50
  Euler        3.33±0.25 0.108   1.99±0.15 0.063   1.46±0.11 0.030
  Heun         1.60±0.12 0.127   1.27±0.07 0.044   1.20±0.08 0.014
  multistep    2.12±0.17 0.094   1.42±0.11 0.044   1.29±0.10 0.015
gamma = 2, reference: FD  2.29±0.11; FD and RMS distance to it
evaluations                 10                 20                 50
  Euler        3.88±0.14 0.109   3.07±0.11 0.066   2.57±0.10 0.035
  Heun         3.51±0.15 0.142   2.77±0.10 0.066   2.43±0.10 0.021
  multistep    3.31±0.12 0.099   2.71±0.10 0.051   2.38±0.10 0.021
```

2 つの指標は一致せず、その食い違いから FD の性質が見えてくる。ガイダンスなしの 10 回評価では、ホイン法のサンプルはオイラー法よりも参照解から離れている（約 0.1 に対して距離 0.12〜0.13）にもかかわらず、FD はオイラー法の半分（約 3.2 に対して約 1.6）である。FD は個々のサンプルの誤差ではなくサンプル集合の分布を評価するため、ソルバがあるノイズ画像に対して本来とは異なる衣類を生成しても、集合としては写実的なサンプル群になりうるのである。

予測どおり、ガイダンスを加えると速度場への追随は難しくなる。ガイダンスは写実性と引き換えにクラスとの整合性を高めるため、$\gamma = 2$ では参照解自体のスコアが約 2.1〜2.3 となる。20 回評価ではどのソルバもまだ参照解より 0.4〜1 高く、50 回までに多段サンプラは参照解の範囲内またはその端に達する一方、ホイン法は 0.15〜0.2、オイラー法は約 0.3 上回ったままである。参照解までの距離は、この予測を部分的にのみ裏付ける。20 回評価におけるホイン法の参照解からの距離はガイダンスなしより 3 分の 1 から半分大きくなるが、オイラー法と多段サンプラの変化

は 6 分の 1 以下にとどまり、10 回および 20 回評価では 3 手法の中で多段サンプラが最も参照解に近い。したがってガイダンスのもとでは、50 回評価（ネットワーク順伝播 100 回）までにプラトーへ到達するのは多段サンプラだけであり、ホイン法とオイラー法はさらに多くの評価を必要とする（何回必要かはこの表からはわからない）。

積分誤差は視覚的にも確認できる。以下に示すのは、同じ 8 枚のノイズ画像とラベルから 4 つのサンプラ、すなわち 50 ステップの祖先連鎖、10 ステップおよび 50 ステップの DDIM、そして線形グリッド上で 20 回評価したホイン法によって生成した各 8 枚の画像である。

```
z8 = noises[0][:8]
pred8 = d2l.make_pred(d2l.conditional(ema, labels[:8]), lin, 'velocity')
rows = [d2l.ancestral_sample(pred8, z8, vp, 50),
        d2l.ddim_sample(pred8, z8, vp, 10), d2l.ddim_sample(pred8, z8, vp, 50),
        d2l.heun_sample(pred8, z8, lin.sigma_bars(11))]
d2l.show_gray(torch.cat(rows).clamp(-1, 1), 4, 8);
```

決定論的サンプラはノイズから画像への 1 つの写像を近似するため、決定論的な 3 つの行は列ごとに揃っている。10 ステップでは、50 ステップとホイン法で一致して描かれる柄や輪郭などの細部が平滑化されて消えている。祖先連鎖の行はざらついており、背景や生地に斑点状のノイズが見られる。これが最初の表で確認した過剰な分散を可視化したものである。

では、未測定のモデルに対してはどのようにサンプラを選べばよいだろうか。ノイズ画像の集合と評価スコアを固定し、モデル自身の訓練経路のグリッド上で決定論的ソルバから始めて、スコアの変化が実行間の変動幅以下に収まるまで評価回数を 2 倍ずつ増やしていけばよい。最後に残るのはモデル自体の誤差であり、サンプラにできる最善のことはサンプラ由来の誤差を上乗せしないことである。このモデルでは、プロットの曲線が平坦になる箇所、すなわちホイン法で約 20 回の評価で手順が停止し、新たなノイズが役立つことはない。ガイダンスのもとでは停止が遅くなり、多段サンプラの約 50 回評価となる。多段サンプラはステップあたり 1 回の評価で済み、積分誤差も最小であるため、ガイダンス下ではより優れたデフォルトとなる。論文から持ってきたグリッドで訓練グリッドを置き換える前にも、同じテストを通すべきである。10 回評価ではどのソルバもプラトーに達しておらず、1〜2 ステップでプラトーに到達するには別の種類のモデルが必要になる（17.8 節）。

17.5.3 逆変換と編集

決定論的サンプラは微分方程式を積分するものであり、微分方程式はどちらの向きにも実行できる。これを逆向きに実行すると、各画像に対してノイズ画像、すなわちその画像の**コード**（code）が得られ、サンプラはそのコードから元の画像を再現する。コードを使えば画像間の補間が可能になる。さらに同じサンプラを、部分的にノイズを加えた画像から開始したり各ステップで制約を課したりすることで、再訓練なしに編集やインペインティングを行える。

17.5.3.1 潜在コード

`ddim_invert` は決定論的サンプラを上向きに、すなわち $t = 0$ の画像からノイズへ向けて実行する。各ステップは次のオイラーステップによってレベル t から次のレベル $t' > t$ へと進む。

$$\bar{\mathbf{x}}_{t'} = \bar{\mathbf{x}}_t + (\bar{\sigma}_{t'} - \bar{\sigma}_t)\,\hat{\epsilon}(\bar{\mathbf{x}}_t, \bar{\sigma}_t), \tag{17.41}$$

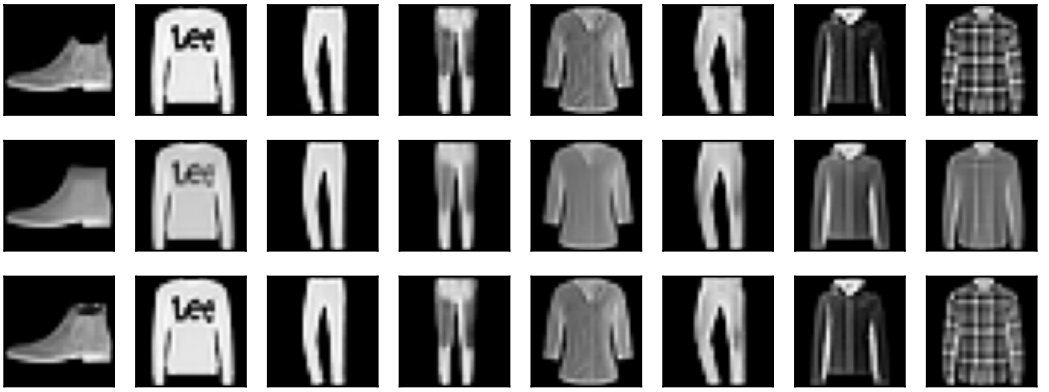
ここでノイズ推定値は下端、すなわちエンコーダが手にしている唯一の点で評価される。予測器が定義されない $\bar{\sigma} = 0$ からの最初のステップでは、代わりに次のレベルのノイズレベルを使って画像を評価する。

```
@torch.no_grad()
def ddim_invert(pred, x, path, steps, t_end=1.0):
    """Deterministic DDIM run backwards, from data x at t = 0 to t_end."""
    a, s = path.grid(steps, 0.0, t_end)
    for i in range(steps):
        j = max(i, 1)  # at t = 0 (sigma = 0) x is read at the next level
        eps = pred(x, a[j], s[j])[1]
        x = a[i + 1] * (x - s[i] * eps) / a[i] + s[i + 1] * eps
    return x
```

デコーダは上端で評価した $\hat{\epsilon}(\bar{\mathbf{x}}_{t'}, \bar{\sigma}_{t'})$ を使って再び下向きにステップを進める。2 つの推定値は 1 ステップにわたる $\hat{\epsilon}$ の変化分だけ異なるため、厳密な速度場を用いたとしてもこの往復は厳密には元に戻らず、その誤差は他の 1 次誤差と同様にステップサイズとともに縮小するはずである。往復では 1 つの速度場を両方向に実行するため、速度場自体が正しいかどうかにかかわらず、生じる誤差はその速度場上でのステップの誤差となる。ただし学習された速度場では、厳密な速度場よりも軌跡に沿って速く変化することがあり、その場合、往復誤差の収束は厳密な速度場を通す場合よりも遅くなる。そこで、それぞれ自身のラベルを持つ 100 枚のテスト画像を線形経路上の 10〜200 ステップでエンコード・デコードし、あわせて混合分布の 20,000 個の点も厳密な速度場による同じ往復に通してみる。

```
x0 = X_test[:100]
pred = d2l.make_pred(d2l.conditional(ema, y_test[:100]), lin, 'velocity')
codes = {n: d2l.ddim_invert(pred, x0, lin, n) for n in (10, 20, 50, 100, 200)}
recon = {n: d2l.ddim_sample(pred, code, lin, n) for n, code in codes.items()}
for n in codes:  # the mixture's points through its exact field, for contrast
    back = d2l.ddim_sample(pred_exact, d2l.ddim_invert(pred_exact, data, lin, n),
                           lin, n)
    print(f'{n:3d} steps: squared error per pixel '
          f'{((recon[n] - x0) ** 2).mean():.4f}, code std {codes[n].std():.2f}; '
          f'mixture {((back - data) ** 2).mean():.4f}')
d2l.show_gray(torch.cat([x0[:8], recon[10][:8], recon[50][:8]]), 3, 8);
```

```
 10 steps: squared error per pixel 0.0347, code std 0.69; mixture 0.3097
 20 steps: squared error per pixel 0.0148, code std 0.76; mixture 0.0878
 50 steps: squared error per pixel 0.0039, code std 0.81; mixture 0.0192
100 steps: squared error per pixel 0.0011, code std 0.84; mixture 0.0074
200 steps: squared error per pixel 0.0002, code std 0.86; mixture 0.0023
```



どちらの往復も厳密には一致せず、どちらの誤差も着実に減少する。10 ステップから 100 ステップの間で二乗平均平方根誤差は両方とも約 6 分の 1 に縮小し（1 次誤差の 10 分の 1 には及ばない）、50 ステップ以降では画像の誤差は少なくとも混合分布と同程度以上の速さで減少する。したがって、学習された速度場は軌跡に沿って厳密な速度場より荒れているわけではなく、残っているのはステップ自体の誤差である。50 ステップにおける誤差はピクセルあたり 0.05〜0.06（-1 から 1 のスケール上）であり、3 行目の再構成画像は元画像に近い。10 ステップでは約 0.19 となり、2 行目では右端のシャツのチェック柄などの細部が失われている。

一方、得られたコードは満足のいくものではない。その標準偏差は 10 ステップで約 0.7、200 ステップでもなお 0.9 未満にとどまるため、画像を再現するコードはサンプラが本来開始点として想定しているノイズよりも幅が狭い。したがって、このコードは標準ガウスノイズからの典型的な抽出サンプルではなく、コード間を補間したりその近傍から新しいコードを抽出したりする際には注意が必要である。

実際の写真をそのコード経由で編集する手法は、これとは別の 2 つ目の困難に直面する。逆変換はある 1 つの速度場の軌跡をたどるのに対し、ガイダンス付きサンプラはそれとは異なる速度場を積分するため、コードからガイダンス付きサンプリングを行うとその軌跡から外れ、元の画像を再現できなくなるのである。ヌルテキスト逆変換（null-text inversion）(Mokady et al. 2023) は、ガイダンス付きサンプラがコードから元画像を再現するよう、各ステップでヌルラベルの埋め込みを最適化することでこの問題を修復する。

サンプラはコードの連続関数であるため、2 枚の画像のコード間を補間することで画像間を連続的に移動できる。しかし、直線は適切な経路ではない。等しいノルム r を持ち角度 ω をなす 2 つのコードの中点ではノルムが $r\cos(\omega/2)$ となり、独立なノイズ画像のようにコードどうしがほぼ直交する場合には約 $\sqrt{2}$ 分の 1 に縮んでしまうが、デコーダはそのように短いコードから開始されたことがない。そこで球面補間（spherical interpolation）では、代わりに 2 つのコードを結ぶ大円に沿って次のように移動する。

$$\mathbf{z}_u = \frac{\sin\left((1-u)\omega\right)\mathbf{z}_0 + \sin(u\omega)\,\mathbf{z}_1}{\sin\omega},$$
$$\cos\omega = \frac{\mathbf{z}_0^\top \mathbf{z}_1}{\|\mathbf{z}_0\|\|\mathbf{z}_1\|},$$

これにより両端点のノルムが保たれる。テスト画像の中からアンクルブーツ 2 枚の 50 ステップのコード間にある 8 点をデコードしてみよう。行全体が同一クラスであるため、デコーダには 1 つのラベルを与えられる。

```
i, j = torch.nonzero(y_test[:100] == y_test[0])[:2, 0]  # one class: one field
z0, z1 = codes[50][i], codes[50][j]
omega = torch.arccos(((z0 * z1).sum() / (z0.norm() * z1.norm())).clamp(-1, 1))
u = torch.linspace(0, 1, 8, device=z0.device).reshape(-1, 1, 1, 1)
zs = (torch.sin((1 - u) * omega) * z0 + torch.sin(u * omega) * z1
      ) / torch.sin(omega)
pred_i = d2l.make_pred(d2l.conditional(ema, y_test[i].repeat(8)), lin,
                       'velocity')
d2l.show_gray(d2l.ddim_sample(pred_i, zs, lin, 50), 1, 8);
```

形状は一方のブーツからもう一方へと徐々に変化するが、中間のフレームは第 3 のブーツというより 2 つのブーツを重ね合わせたように見える。球面補間はコードを適切なノルムに保つものの、2 つのブーツの間にあるコードをデコーダが第 3 のブーツへと写す保証はどこにもないのである。

17.5.3.2 部分ノイズ付加による編集

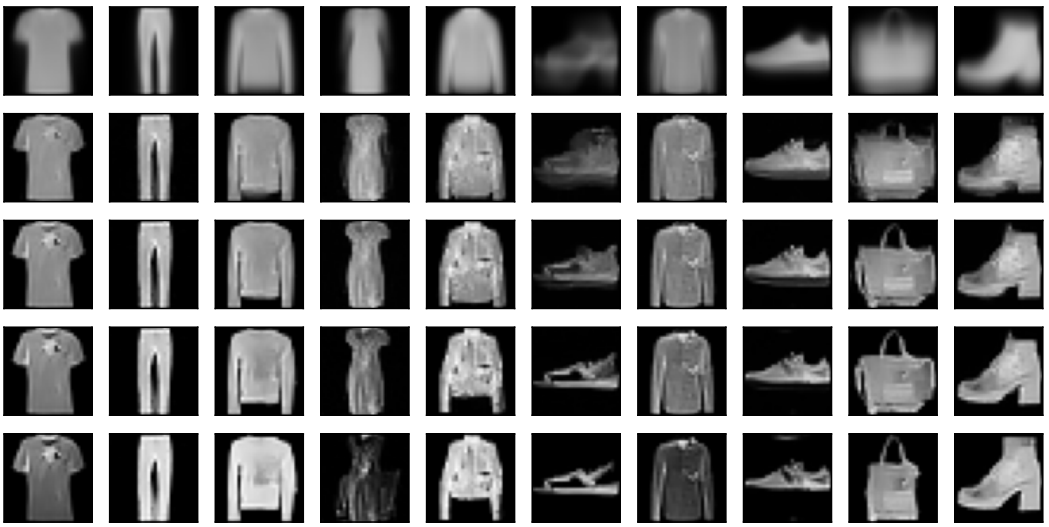
逆変換は、画像を厳密に再現するノイズを求める操作であった。編集において欲しいのはその逆、すなわち入力の細部を忘れつつ大まかな構図だけを保つ開始点である。そこで C. Meng et al. (2022) は、スケッチや粗いコラージュのようなラフな入力 $\mathbf{y}$ に直接新たなノイズを加えて中間時刻 t_0 の状態 $\mathbf{x}_{t_0} = \alpha_{t_0}\mathbf{y} + \sigma_{t_0}\epsilon$ を作り、t_0 からデータ側へとサンプラを実行する。ノイズが実画像にはない入力の不自然な細部を覆い隠し、サンプラがそれを実画像らしい細部に置き換え、大まかな構図だけが生き残る。時刻 t_0 は入力への忠実度（fidelity）と写実性のバランスを決める。このバランスはどれほど良好になりうるだろうか。

ここではラフな入力として、実際のどの衣類にも似ていないぼやけたシルエットである 10 個のクラス平均画像を用い、それぞれに 100 通りずつ $t_0 \in \{0.3, 0.5, 0.7, 1\}$ までノイズを加えて、対応するクラスラベルのもとで 25 回の DDIM ステップにより逆向きに積分する。入力に対するピクセルあたりの二乗距離（忠実度）と 1,000 枚の出力の FD（写実性）を出力する。なお、$t_0 = 0$ の行は入力画像そのものである。

```
means = torch.stack([X[y == c].mean(0) for c in range(10)])
inputs = means.repeat(100, 1, 1, 1)                 # 1,000 inputs, class i % 10
pred = d2l.make_pred(d2l.conditional(ema, labels), lin, 'velocity')
torch.manual_seed(2)
```

```
eps = torch.randn_like(inputs)
edits = {0.0: inputs}
for t0 in (0.3, 0.5, 0.7, 1.0):
    x_t0 = lin.sample(inputs, torch.full((1000,), t0, device=X.device), eps)
    edits[t0] = d2l.ddim_sample(pred, x_t0, lin, 25, t_start=t0).clamp(-1, 1)
for t0, x in edits.items():
    print(f't0 = {t0:.1f}: squared distance to the input '
          f'{((x - inputs) ** 2).mean():.3f}, '
          f'FD {d2l.fid_score(features(x), f_test):.2f}')
d2l.show_gray(torch.cat([x[:10] for x in edits.values()]), 5, 10);
```

```
t0 = 0.0: squared distance to the input 0.000, FD 87.04
t0 = 0.3: squared distance to the input 0.018, FD 30.71
t0 = 0.5: squared distance to the input 0.045, FD 12.19
t0 = 0.7: squared distance to the input 0.087, FD 4.01
t0 = 1.0: squared distance to the input 0.168, FD 1.88
```



入力の FD は 85〜87 であり、t_0 が大きくなるにつれて入力からの距離は増大し、FD は低下する。$t_0 = 0.3$ では出力は平均画像より鮮明になるもののむらがあり、約 30 という FD は 10 ステップの祖先連鎖と同水準で、どの決定論的サンプラにも遠く及ばない。0.5 では奇妙な質感を持つ衣類のように見える（FD は 11〜12）。0.7 になってようやく FD が約 4 となって 10 ステップの決定論的サンプラに近づくが、その時点では入力から生き残っているのは輪郭と姿勢だけである。$t_0 = 1$ では入力は完全に消え去り、各出力はそのクラスの新たなサンプルとなって FD は 2 を下回る。また、同じノイズ画像から開始しているため、0.7 と 1 の行はしばしば似た見た目になる。クラス平均のように粗い入力に対しては、入力がほとんど残らないほどの t_0 に達して初めて写実性が得られるのである。

17.5.3.3 インペインティング

画像の一部（たとえば上半分）が既知であり、残りの部分をモデルに埋めさせたいとしよう。既知のピクセルで 1 をとるマスクを $\mathbf{m}$、そのピクセル値を $\mathbf{y}$ と書く。どのレベルにおいても、$\mathbf{x}_s$ の既知ピクセルがどのような姿であるべきかはわかっており、それは $\alpha_s \mathbf{y}$ にスケール σ_s のノイズを加えたものである。そこで置換インペインティング（replacement inpainting）(Y. Song,

Sohl-Dickstein, et al. 2021; Lugmayr et al. 2022) は、通常どおりサンプラの各ステップを進めたあと、既知ピクセルを新しいレベルで新たにノイズを加えた $\mathbf{y}$ のコピーによって次のように上書きする。

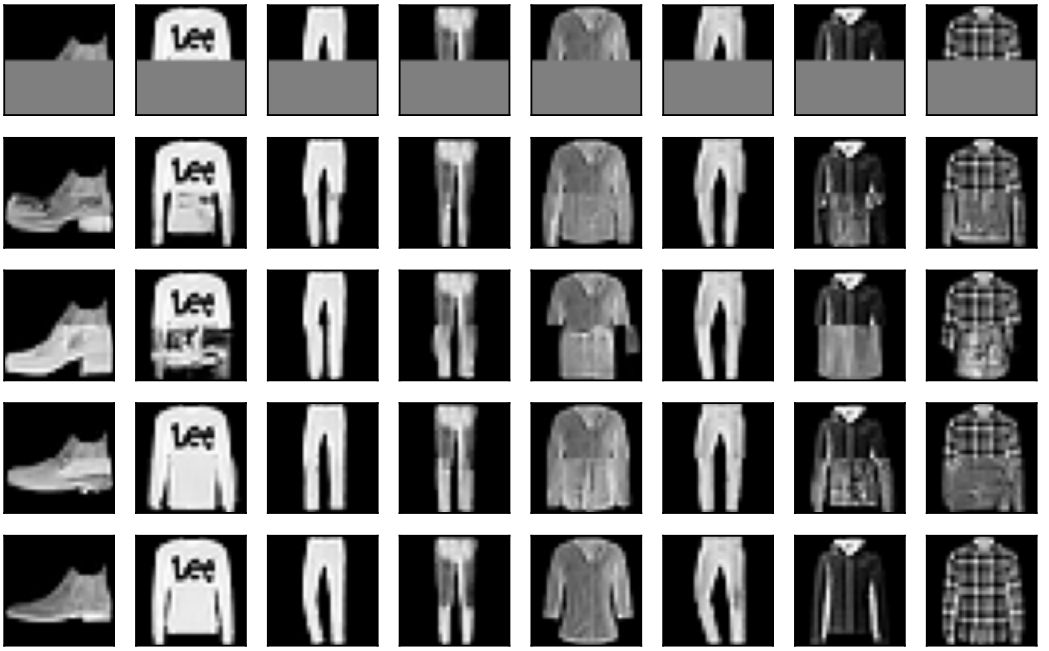
$$\mathbf{x}_s \leftarrow \mathbf{m} \odot \left(\alpha_s \mathbf{y} + \sigma_s \xi\right) + (\mathbf{1} - \mathbf{m}) \odot \mathbf{x}_s. \tag{17.42}$$

この置換は近似である。各ステップは既知ピクセルも自由であるかのように全ピクセルを動かし、そのあとステップがどこへ動かしたかとは無関係なノイズで既知ピクセルを上書きする。未知ピクセルは、ネットワークが次にノイズ画像を見るときに初めて既知ピクセルの情報を知るため、サンプラは既知ピクセルを条件とする正確な分布から未知ピクセルを抽出しているわけではない。`inpaint` はこの上書きを挟めるよう、サンプラを 1 ステップずつ実行する。

```
@torch.no_grad()
def inpaint(pred, known, mask, path, steps):
    """DDIM from noise that resets the known pixels (mask 1) at each level."""
    a, s = path.grid(steps)
    x = torch.randn_like(known)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        x = a[i + 1] * x0 + s[i + 1] * eps            # one DDIM step, eta = 0
        noised = a[i + 1] * known + s[i + 1] * torch.randn_like(x)
        x = mask * noised + (1 - mask) * x            # keep the known pixels
    return x
```

8 枚のテスト画像の上半分を保持し、それぞれの画像のラベルのもとで異なるノイズから 3 通りずつ補完画像を生成する。各行は上から既知の上半分、3 通りの補完画像、そして元画像を示している。

```
mask = torch.zeros(1, 1, 28, 28, device=X.device)
mask[..., :14, :] = 1                                 # the upper half is known
pred = d2l.make_pred(d2l.conditional(ema, y_test[:8].repeat(3)), lin,
                     'velocity')
torch.manual_seed(3)
fills = inpaint(pred, X_test[:8].repeat(3, 1, 1, 1), mask, lin, 50)
d2l.show_gray(torch.cat([X_test[:8] * mask, fills, X_test[:8]]), 5, 8);
```



大半の補完画像は既知の上半分を自然に引き継いでそのクラスらしい衣類を形作っており、1 枚の画像に対する 3 通りの補完結果も互いに異なっている。一方、近似の影響は継ぎ目に現れている。いくつかの補完画像（特にシャツで顕著）では、下半分が上半分とは異なる質感や濃淡になっている（演習問題 5 でこれを測定し修復する）。

本節のすべてのサンプラは同じ訓練済み速度場を積分しており、違いはどのように積分するかという点だけであった。速度場そのものについてはここまで無条件に信頼してきたが、それは選んだターゲットに対する重みなし二乗誤差から得られたものにすぎない。次節では、なぜこの損失が適切なのか、そしてターゲット、経路、重み付けの選択が何を変えるのかを問う。

17.5.4 まとめ

訓練は $\mathbf{x}_t$ の各時刻 t における周辺分布のみに依存するため、1 つの訓練済みネットワークによって、確率性の強度 $\eta \geq 0$ でパラメータ化された DDIM サンプラ族の全体をサポートできる。$\eta = 0$ では DDIM は決定論的になり、確率フロー ODE $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$（再スケーリングされた座標系 $(\bar{\mathbf{x}}, \bar{\sigma}) = (\mathbf{x}/\alpha_t, \sigma_t/\alpha_t)$）上のオイラー法と一致する。本節の Fashion-MNIST モデルでは、少ない予算および中程度の予算において決定論的 ODE ソルバが一貫して確率的サンプラ（$\eta = 1$ および祖先連鎖）を上回り、線形グリッド上の 2 次ホイン法は約 20 回の評価でモデルの FD プラトーに到達し、2 次のアダムス・バッシュフォース多段ソルバはステップあたり 1 回の評価のまま 50 回の評価までにガイダンス下のプラトーに到達する。決定論的 ODE を逆向きに実行する（`ddim_invert`）と任意の画像が潜在ノイズコードへと写され、再訓練なしに球面潜在補間、部分ノイズ付加による画像編集、および置換インペインティングを実現できる。

17.5.5 演習問題

1. **順ステップとマルコフ的なメンバー。**●$\mathbf{x}_0$ が与えられたもとで、$\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ と、$\mathbf{x}_s$ を Equation 17.34 から同じ ϵ を用いて抽出する。
    1. 同時ガウス分布を条件付けることで、$\mathbf{x}_t$ の（$\mathbf{x}_s$ と $\mathbf{x}_0$ が与えられたもとでの）平均と共分散を計算せよ。

2. 平均が $\mathbf{x}_0$ に依存しないのは $\tilde{\sigma}$ が $\eta = 1$ における値をとるときに限ること、またそのとき条件付き分布が順ステップ $\mathbf{x}_t = (\alpha_t/\alpha_s)\mathbf{x}_s + \sqrt{\sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2}\,\nu$（17.2 節のノイズ付加連鎖）に一致することを示し、この連鎖が順過程がマルコフ的になる族の唯一のメンバーであると結論せよ。
3. すべてのステップで $\tilde{\sigma} = 0$ とすると、順過程が $\mathbf{x}_t = \alpha_t\mathbf{x}_0 + \sigma_t\epsilon$（1 つの ϵ をすべての時刻で共有）になることを示せ。なぜ回帰ではこれらの順過程を区別できないのか。

2. **ガウスデータ**。データを分布 $\mathcal{N}(0, v)$ に従う 1 次元データとする。
1. 座標 $(\bar{x}, \bar{\sigma})$ において、厳密なノイズ推定値が $\hat{\epsilon} = \bar{\sigma}\bar{x}/(v + \bar{\sigma}^2)$ であること、$\bar{\sigma}$ から $\bar{\sigma}'$ への 1 回の決定論的ストライドが $\bar{x}$ に $(v + \bar{\sigma}\bar{\sigma}')/(v + \bar{\sigma}^2)$ を掛けること、および厳密なフローがそれに $\sqrt{(v + \bar{\sigma}'^2)/(v + \bar{\sigma}^2)}$ を掛けることを示せ。
2. ストライドの倍率が厳密な倍率を決して超えず、等号成立は $\bar{\sigma}' = \bar{\sigma}$ のときに限ることを示せ。このことは、任意のグリッド上でのガウスデータの決定論的サンプルの分散について何を意味するか。
3. 次に、Equation 17.35 の族から $0 \le \eta \le 1$ かつ厳密な推定値を持つ任意のメンバーをとる。x_s の分散（$x_t \sim \mathcal{N}(0, \alpha_t^2 v + \sigma_t^2)$ から 1 ステップ進んだ後）を $c = \sqrt{\sigma_s^2 - \tilde{\sigma}^2}$ の関数として表し、コーシー・シュワルツの不等式を用いてそれが厳密な $\alpha_s^2 v + \sigma_s^2$ を下回り、かつ η が大きくなるにつれて減少することを示せ。これは本節で出力した広がりのうち、どれを説明するものか。
4. $v = 0.25$（`mix` の 1 つのモードの座標あたりの分散）とする。決定論的サンプルの標準偏差 s を、$x_1 \sim \mathcal{N}(0, 1)$ から始めて 200、50、20、10、5 ステップの分散保持グリッド上で計算せよ。座標あたりこの標準偏差を持つ 2 次元点群の広がりである $2s^2$ を、データの 0.5 および本節で出力した広がりと比較せよ。ガウスモードは、混合分布の収縮と同程度を予測するか、それより大きな収縮を予測するか、それとも小さな収縮を予測するか。

3. [extended] **対数信号対雑音比における 2 次サンプラ**。●●DPM-Solver-2 (Lu, Zhou, et al. 2022) は、$\bar{\sigma}$ から $\bar{\sigma}'$ へのステップを 2 回の評価で行い、2 回目の評価を中点 $\bar{\sigma}_m = \sqrt{\bar{\sigma}\bar{\sigma}'}$（$\log\bar{\sigma}$ 上の中点）にとる。すなわち、$\bar{\mathbf{x}}_m = \bar{\mathbf{x}} + (\bar{\sigma}_m - \bar{\sigma})\,\hat{\epsilon}(\bar{\mathbf{x}}, \bar{\sigma})$ および $\bar{\mathbf{x}}' = \bar{\mathbf{x}} + (\bar{\sigma}' - \bar{\sigma})\,\hat{\epsilon}(\bar{\mathbf{x}}_m, \bar{\sigma}_m)$ である。
1. $h = \log(\bar{\sigma}/\bar{\sigma}')$ とおく。1 ステップにわたる $\bar{\mathbf{x}}$ の厳密な変化量を軌跡に沿った $\hat{\epsilon}$ の積分として表し、この更新の局所誤差が $O(\bar{\sigma}h^3)$ である一方、オイラーステップの局所誤差が $O(\bar{\sigma}h^2)$ であることを示せ。ただし、$\hat{\epsilon}$ は軌跡に沿って $\log\bar{\sigma}$ の滑らかな関数であり、$\mathbf{x} = \bar{\mathbf{x}}/\sqrt{1 + \bar{\sigma}^2}$ に関してリプシッツ連続である（したがって大きな $\bar{\sigma}$ において $\bar{\mathbf{x}}$ に関するリプシッツ定数が $1/\bar{\sigma}$ のオーダーになる）ものと仮定せよ。
2. インターフェース `pred(x, alpha, sigma)` と関係式 $\alpha = 1/\sqrt{1 + \bar{\sigma}^2}$、$\sigma = \bar{\sigma}\alpha$ を用いてこの更新を実装し、$\bar{\sigma} = 0$ への最後のステップはオイラー法で進めよ。これを `ode_sample`、`heun_sample`、`multistep_sample` と約 11、21、51 回の評価回数で比較せよ。比較には、`pred_exact` を用いた混合分布上での 1,000 ステップ参照解までの距離と、画像上での FD を用いよ。
3. $\log\bar{\sigma}$ 上で 80 から 0.002 の間に等間隔に配置したレベルを用いて同じ実験を繰り返せ。各予算において、手法とグリッドのどの組み合わせが最良か。

4. **最適な逆分散**。●t から s への確率的ステップは、事後平均 $\alpha_s\hat{\mathbf{x}}_0 + \sigma_s\kappa\hat{\epsilon}$ へ移動し、座標あたり分散 v のガウスノイズを加える。Ho et al. (2020) は順ステップ分散と事後分散を試し、Bao et al. (2022) は任意のデータに対する最良の v を導出した。厳密な推定値を仮定し、$\bar{e}_t = \mathbb{E}\|\epsilon - \hat{\epsilon}\|^2/d$ を時刻 t（d 次元空間）における厳密なノイズ推定値の座標あたり誤差とする。
1. 演習問題 3a（17.3 節）より、真の逆遷移 $q(\mathbf{x}_s \mid \mathbf{x}_t)$ の共分散は $\tau^2\mathbf{I} + c^2\mathrm{Cov}(\mathbf{x}_0 \mid \mathbf{x}_t)$ となる。ここで c は $\mathbf{x}_0$ の係数（$q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$ の平均における係数）である。その平均を $\alpha_s\mathbf{x}_0 + \sigma_s\kappa\epsilon$（ただし $\epsilon = (\mathbf{x}_t - \alpha_t\mathbf{x}_0)/\sigma_t$）と表すことで、$\tau^2 = \tilde{\sigma}^2$（事後分散）および $c = k = \alpha_s - \sigma_s\kappa\,\alpha_t/\sigma_t$ となることを導け。真の逆遷移の平均は何か。

   2. 分散 v のうち、$q(\mathbf{x}_s \mid \mathbf{x}_t)$ からガウスステップへの期待 KL ダイバージェンスを最小化するものが $v^\star = \tilde{\sigma}^2 + k^2\bar{\sigma}_t^2\bar{e}_t$ であり、かつ $\bar{e}_t = 1 - \sigma_t^2\,\mathbb{E}\|\nabla \log p_t(\mathbf{x}_t)\|^2/d$ が成り立つため $v^\star$ がスコア関数のみから定まることを示せ。
   3. $v^\star$ が、データが 1 点のときは事後分散になり、分散保持経路上の標準ガウスデータのときは順ステップ分散になることを示せ。
   4. （任意・コード課題）$\bar{e}_t$ を、混合分布上で `pred_exact` を用いて 50 ステップの分散保持グリッド上で推定し、3 種類の分散それぞれを用いた確率的ステップの 20,000 サンプルの広がりを比較せよ。
5. [code] **調和化を伴うインペインティング。**●`inpaint` による補完画像は、境界に沿って既知の半分と食い違うことがある。
   1. 継ぎ目を測定せよ。すなわち、補完画像の 13 行目と 14 行目の平均絶対差を計算し、実際のテスト画像における同じ量と比較せよ。
   2. RePaint (Lugmayr et al. 2022) は上下両半分が情報を交換できるようにするため、t から s への各ステップの後に順ステップで結果を再び t までノイズ付加し、そのステップをレベルあたり r 回繰り返す。これを $r \in \{1, 5, 10\}$、50 レベルで実装し、継ぎ目を再度測定せよ。各反復にはどれだけのコストがかかるか。
   3. 各 r について、1 枚のテスト画像の補完結果を 5 通り生成せよ。それらはどの程度多様か。
6. [code] **別クラスへ向けた編集。**本節のクラス条件付きモデルを使用し、`d2l.make_pred(d2l.cfg(ema, c, gamma), lin, 'velocity')` によって誘導せよ（ここで `c` は画像ごとに 1 つのラベルを保持する）。
   1. ズボンのテスト画像 100 枚を $t_0 \in \{0.3, 0.5, 0.7, 0.9\}$ までノイズ付加し、ドレスのラベルへ向けて 25 回の DDIM ステップで逆向きに積分せよ（$\gamma = 1$ および $\gamma = 2$ の両方で実行せよ）。編集のコードセルと同様に、`cnn` がドレスと分類する出力の割合と、入力に対するピクセルあたりの二乗距離を報告せよ。
   2. 純粋なノイズから始め、ズボンのラベルかつ $\gamma = 1$ で $t = 1$ から t^* までサンプリングし、続けてドレスのラベルかつ (a) の 2 つの強度で $t = 0$ までサンプリングせよ（`ddim_sample` は `t_start` と `t_end` を受け取る）。これを $t^* \in \{0.3, 0.5, 0.7, 0.9\}$ について行い、ドレスと分類される割合を t^* に対してプロットし、(a) の t_0 に対する曲線の横に並べて、混合分布における演習問題 4（17.3 節）のコミットメント時刻と両者を比較せよ。
   3. 厳密なネットワークを用いた場合、同じ γ かつ $t_0 = t^*$ における 2 つの実験が同じ分布に従う出力を生成することを示せ。曲線間にはどのような違いが残り、それはどこから生じるのか。

小問 (a) は Berkeley CS180/280A の Project 5A（2025 年秋学期）1.7.3 節を改変したものである。

17.6 尤度、重み付け、設計空間

これまでの節では、ノイズや速度をターゲットとする重みなしの二乗誤差によって拡散モデル（diffusion model）やフローマッチングモデルを訓練し、生成サンプルによって評価してきた。第 16 章の敵対的モデルとは異なり、拡散モデルはデータ点に対して確率密度も割り当てる。すなわち、Sohl-Dickstein et al. (2015) と Ho et al. (2020) は、逆方向マルコフ連鎖が潜在変数モデル（latent-variable model）を定め、そのエビデンス下界（evidence lower bound、ELBO）が各ノイズレベルにおけるノイズ予測回帰の重み付き和に分解されることを示した。一方、確率フロー ODE（probability-flow ODE）は瞬間的な変数変換（instantaneous change of variables）の公式を通じて厳密な対数尤度を定める (Y. Song, Sohl-Dickstein, et al. 2021)。

さらに、Kingma et al. (2021) と Kingma and Gao (2023) は、対数信号対雑音比（log signal-to-noise ratio、log-SNR）$\lambda_t = \log(\alpha_t^2/\sigma_t^2)$ を用いて書き直すと、標準的な拡散モデルおよびフロ

ーマッチングの目的関数がすべて、関数 $w(\lambda)$ で重み付けされたノイズ予測誤差の同じ積分に帰着することを示した。この損失の重み付けと並んで、Karras et al. (2022)（EDM）は入出力の前処理（preconditioning）と訓練時のノイズレベル分布を独立した設計軸として特定した。本節では、離散時間および連続時間の ELBO を導出し、3 モード混合分布上で厳密なフロー尤度と境界を比較し、既存の目的関数が暗黙に定める対数信号対雑音比の重み付け $w(\lambda)$ によってそれらを統一したうえで、Fashion-MNIST 上で統制された 1 要因アブレーション実験を実施する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.6.1 尤度境界

まず 17.2 節の連鎖を生成モデルとして書き下し、その尤度の境界を求めよう。すると、ノイズ回帰が自然に現れる。

17.6.1.1 潜在変数モデルとしての連鎖

順方向連鎖のステップ k は、現在の点を縮小してノイズを加える $\mathbf{x}_k = \sqrt{1-\beta_k}\,\mathbf{x}_{k-1} + \sqrt{\beta_k}\,\epsilon_k$ であり、各ステップを合成すると 1 回のガウス抽出 $\mathbf{x}_k = \alpha_k \mathbf{x}_0 + \sigma_k \epsilon$（ただし $\alpha_k^2 = \prod_{j \le k}(1-\beta_j)$、$\sigma_k^2 = 1 - \alpha_k^2$）になる。これは時刻 $t_k = k/T$ における VP 経路にほかならない。この順方向連鎖の分布を q と書く。生成モデルは、学習されたガウスステップによってこれを逆向きにたどる。

$$
\begin{aligned}
p_\theta(\mathbf{x}_{0:T}) &= p(\mathbf{x}_T) \prod_{k=1}^{T} p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k), \\
p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k) &= \mathcal{N}\!\left(\mu_\theta(\mathbf{x}_k, k),\, \nu_k \mathbf{I}\right),
\end{aligned}
\tag{17.43}
$$

ここで、$p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ であり、平均 μ_θ はネットワークで計算し、分散 ν_k は固定とする（σ_k は引き続き経路のノイズスケールを表す）。

$\mathbf{x}_0$ の密度は、この同時分布を潜在変数 $\mathbf{x}_{1:T}$ について積分したものであるが、これはデータ空間の T 個の直積上の積分であり直接計算できない。しかし、潜在変数上の任意の分布 $q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)$ を用いれば、この積分は期待値 $p_\theta(\mathbf{x}_0) = \mathbb{E}_q\big[p_\theta(\mathbf{x}_{0:T}) / q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)\big]$ に書き換えられる。対数関数は凹関数であるため、平均の対数は対数の平均以上となる（イェンセンの不等式）。これより、負の対数尤度に対して次の上界が得られる。

$$
\begin{aligned}
-\log p_\theta(\mathbf{x}_0) &= -\log \mathbb{E}_q\left[\frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\right] \\
&\le \mathbb{E}_{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\right] =: L(\mathbf{x}_0).
\end{aligned}
\tag{17.44}
$$

これは変分推論（variational inference、36.7 節）における**エビデンス下界**（ELBO）の符号を反転させたものである。多くの潜在変数モデルでは真の事後分布に近づくよう q を学習する必要があるが、ここでは順方向連鎖がそれを与えるため、学習すべきものは何もない。不等式のギャップは $D_{\mathrm{KL}}(q(\mathbf{x}_{1:T} \mid \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{1:T} \mid \mathbf{x}_0))$ であり、$\mathbf{x}_0$ が与えられたもとで学習された逆方向連鎖が順方向連鎖を正確に逆向きにたどるとき、境界はタイトになる。データ全体で平均をとると、この境界は少なくとも交差エントロピー $\mathbb{E}_p[-\log p_\theta(\mathbf{x}_0)] = H(p) + D_{\mathrm{KL}}(p \,\|\, p_\theta)$ 以上となるため、モード欠落を起こすモデルは、そのモードに属するすべてのデータ点で大きなペナルティを受ける。

2 つの連鎖を比較するため、それぞれの分解、すなわち p_θ に対する上記の積と $q(\mathbf{x}_{1:T} \mid \mathbf{x}_0) = \prod_k q(\mathbf{x}_k \mid \mathbf{x}_{k-1})$ を代入すると、境界はステップごとの項に分解される。

$$L(\mathbf{x}_0) = \mathbb{E}_q\Big[-\log p(\mathbf{x}_T) - \sum_{k=1}^{T} \log \frac{p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k)}{q(\mathbf{x}_k \mid \mathbf{x}_{k-1})} \Big]. \tag{17.45}$$

第 k 項は、学習されたステップが $\mathbf{x}_k$ から $\mathbf{x}_{k-1}$ へ進むのに対し、順方向ステップは逆向きに進むため、このままではまだダイバージェンスとして読めない。そこでベイズの定理を使って順方向ステップの向きを反転させる。マルコフ性（Markov property）により、さらに $\mathbf{x}_0$ で条件付けてもこの分布は変わらないため、次式が成り立つ。

$$\begin{aligned} q(\mathbf{x}_k \mid \mathbf{x}_{k-1}) &= q(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{x}_0) \\ &= \frac{q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0)\, q(\mathbf{x}_k \mid \mathbf{x}_0)}{q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)}. \end{aligned}$$

これを $k \geq 2$ に代入すると、周辺分布の比が $\prod_{k=2}^{T} q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)/q(\mathbf{x}_k \mid \mathbf{x}_0) = q(\mathbf{x}_1 \mid \mathbf{x}_0)/q(\mathbf{x}_T \mid \mathbf{x}_0)$ と逐次相殺され、分子が $k = 1$ で残っていた順方向ステップを打ち消す。残った各対数は、期待値をとるとそれぞれダイバージェンスになる。

$$\begin{aligned} L(\mathbf{x}_0) = {} & \underbrace{D_{\mathrm{KL}}\big(q(\mathbf{x}_T \mid \mathbf{x}_0) \,\|\, p(\mathbf{x}_T)\big)}_{L_T} \\ & + \sum_{k=2}^{T} \underbrace{\mathbb{E}_q\, D_{\mathrm{KL}}\big(q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k)\big)}_{L_{k-1}} \\ & + \underbrace{\mathbb{E}_q\big[-\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1) \big]}_{L_0}. \end{aligned} \tag{17.46}$$

事前項（prior term）L_T は連鎖の終端がサンプラの出発点であるガウス分布に一致するかを測り、**ノイズ除去項**（denoising term）L_{k-1} は学習された各ステップを既知のクリーンな点に対するステップ k の逆向き分布である**順事後分布**（forward posterior）と比較し、**再構成項**（reconstruction term）L_0 は最終ステップを評価する。$\mathbf{x}_0$ で条件付けることで順事後分布は解析的に計算できるようになる。すなわち、$\mathbf{x}_{k-1}$ に関する 2 つのガウス分布（順方向ステップと $q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)$）の積であるため、これもガウス分布となる（演習問題 5 で平方完成により導出する）。

$$\begin{aligned} q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) = \mathcal{N}(\tilde{\mu}_k, \tilde{\beta}_k \mathbf{I}), \qquad \tilde{\beta}_k &= \frac{\beta_k\, \sigma_{k-1}^2}{\sigma_k^2}, \\ \tilde{\mu}_k = \frac{\alpha_{k-1}\beta_k}{\sigma_k^2}\, \mathbf{x}_0 + \frac{\sqrt{1-\beta_k}\, \sigma_{k-1}^2}{\sigma_k^2}\, \mathbf{x}_k. & \end{aligned} \tag{17.47}$$

この平均は、クリーンな点とノイズを含む点がそれぞれ示唆する $\mathbf{x}_{k-1}$ の位置の重み付き平均であり、分散は順方向ステップの β_k より小さい。両端を知ることで中間点が絞り込まれるからである。

事前項とノイズ除去項はいずれもガウス分布どうしを比較するため、1 つの公式で計算できる。d 次元の等方性ガウス分布に対して、次式が成り立つ。

$$\begin{aligned} &D_{\mathrm{KL}}(\mathcal{N}(\mathbf{m}_1, v_1\mathbf{I}) \,\|\, \mathcal{N}(\mathbf{m}_2, v_2\mathbf{I})) \\ &\quad = \frac{d}{2}\Big(\frac{v_1}{v_2} - 1 - \log\frac{v_1}{v_2}\Big) + \frac{\|\mathbf{m}_1 - \mathbf{m}_2\|^2}{2v_2}, \end{aligned} \tag{17.48}$$

これは 16.1.4.3 節の公式の特別な場合である。第 1 項は分散のずれに、第 2 項は第 2 の分布の分散を単位として測った平均のずれにペナルティを課す。

各項はどの程度の大きさになるだろうか。最小の例は手計算で確認できる。1 次元座標を考え、データは $\mathcal{N}(0,1)$ から抽出され、連鎖は $T = 2$ ステップで $\beta_1 = \beta_2 = \frac{1}{2}$（したがって $\alpha_1^2 = \frac{1}{2}$、$\alpha_2^2 = \frac{1}{4}$）とし、学習される平均は理論上最良のものとする。事前項は連鎖の終端 $\mathcal{N}(\alpha_2 x_0, \frac{3}{4})$ を $\mathcal{N}(0,1)$ と比較するもので、$x_0^2/8 + 0.019$、平均すると 0.144 に等しい。2 回の大きなステップでは x_0 の情報がかなり残るからである。第 2 ステップの順事後分布は平均 $\frac{\sqrt{2}}{3}(x_0 + x_2)$、分散 $\tilde\beta_2 = \frac{1}{3}$ をもち、最良の学習ステップはその平均に含まれる x_0 を x_2 からの最良の推定値（依然として分散 $\frac{3}{4}$ の不確実性をもつ）で置き換える。逆方向の分散を順方向ステップ自身の分散 $\nu_k = \beta_k$ とすると、境界の期待値は次のようになる。

$$\mathbb{E}\, L = \underbrace{0.144}_{L_2} + \underbrace{0.203}_{L_1} + \underbrace{1.072}_{L_0} = 1.419 = \tfrac{1}{2}\log(2\pi e),$$

これは $\mathcal{N}(0,1)$ の真の負の対数尤度に一致する。ガウスデータとこの分散の選択のもとでは、わずか 2 ステップですでに境界は厳密になる。再構成項が最も確認しやすい。x_1 が与えられたとき、クリーンな点にはまだ分散 $\frac{1}{2}$ の不確実性が残っており、これはちょうど ν_1 に等しいため、$L_0 = \frac{1}{2}\log(2\pi\cdot\frac{1}{2}) + \frac{1}{2} = 1.072$ となる。代わりに $\nu_2 = \tilde\beta_2$ を用いると、L_1 は 0.25 に増え、境界全体も 1.466 に増加する。この理由は記憶に留めておく価値がある。順事後分布は x_0 を知っているため分散が小さいのに対し、x_2 からの学習ステップはそれを知らないため、その不確実性も分散に含めなければならないのである。$T = 1000$ では事前項はほぼ 0 となり、各ステップは微小になって、ノイズ除去項が訓練信号を運ぶ。

17.6.1.2 境界から重み付き回帰へ

各ノイズ除去項はネットワークに依存しない分散をもつ 2 つのガウス分布を比較するため、Equation 17.48 より平均間の距離だけが θ を含む。

$$L_{k-1} = \mathbb{E}_q\Big[\frac{1}{2\nu_k}\big\|\tilde\mu_k(\mathbf{x}_k, \mathbf{x}_0) - \mu_\theta(\mathbf{x}_k, k)\big\|^2\Big] + \text{const}, \tag{17.49}$$

ここで定数項は $\nu_k = \tilde\beta_k$ のとき 0 になる。つまり各ステップは、順事後分布の平均を出力するようネットワークに要求している。

ネットワークはその平均を直接出力する必要はない。$\mathbf{x}_0 = (\mathbf{x}_k - \sigma_k\epsilon)/\alpha_k$ を Equation 17.47 に代入し、$\sigma_k^2 = \beta_k + (1-\beta_k)\sigma_{k-1}^2$ を用いると、$\mathbf{x}_k$ の係数はまとまって $1/\sqrt{1-\beta_k}$ となり、平均は次のように書ける。

$$\tilde{\mu}_k = \frac{1}{\sqrt{1-\beta_k}}\left(\mathbf{x}_k - \frac{\beta_k}{\sigma_k}\,\epsilon\right). \tag{17.50}$$

この式で ϵ 以外の量はすべて、$\mathbf{x}_k$ と k を受け取るネットワークにとって既知である。そこでネットワークにはノイズを予測させ、同じ形で平均を構成する。

$$\mu_\theta(\mathbf{x}_k, k) = \frac{1}{\sqrt{1-\beta_k}}\left(\mathbf{x}_k - \frac{\beta_k}{\sigma_k}\,\hat{\epsilon}_\theta(\mathbf{x}_k, k)\right), \tag{17.51}$$

すると、2 つの平均の差はノイズ誤差の定数倍となる。

$$\begin{aligned} L_{k-1} = \mathbb{E}_{\mathbf{x}_0,\epsilon}\Big[\frac{\beta_k^2}{2\nu_k(1-\beta_k)\sigma_k^2}\left\|\epsilon - \hat{\epsilon}_\theta(\alpha_k\mathbf{x}_0 + \sigma_k\epsilon, k)\right\|^2\Big] \\ + \text{const}. \end{aligned} \tag{17.52}$$

境界の各項は 17.2 節で見た 1 つのノイズレベルにおけるノイズ回帰に重みを掛けたものであり、$\nu_k = \tilde{\beta}_k$ のときその重みは $\beta_k/(2(1-\beta_k)\sigma_{k-1}^2)$ に等しい。ステップ k の間に信号対雑音比 α_k^2/σ_k^2 は $\sigma_k^2/((1-\beta_k)\sigma_{k-1}^2) = 1 + \beta_k/((1-\beta_k)\sigma_{k-1}^2)$ 倍だけ低下するため、$\lambda_k = \log(\alpha_k^2/\sigma_k^2)$ とおくと重みは次のようになる。

$$\frac{\beta_k}{2(1-\beta_k)\sigma_{k-1}^2} = \tfrac{1}{2}(e^{\lambda_{k-1}-\lambda_k} - 1) \approx \tfrac{1}{2}(\lambda_{k-1} - \lambda_k).$$

すなわち、この境界は対数信号対雑音比をどれだけ下げたかに応じて各ステップに重みを課す。次のセルでは、VP 連鎖（$T = 1000$）に沿って 2 通りの重みを対数信号対雑音比のステップ幅の半分と並べて出力する。

```
vp = d2l.GaussianPath.vp()
a, s = vp.grid(1000)
a, s = [float(v) for v in a[::-1]], [float(v) for v in s[::-1]]  # k = 0..T
for k in (2, 10, 100, 500, 1000):
    beta = 1 - (a[k] / a[k - 1]) ** 2
    beta_tilde = beta * s[k - 1] ** 2 / s[k] ** 2
    weight = lambda nu: beta ** 2 / (2 * nu * (1 - beta) * s[k] ** 2)
    dlam = math.log((a[k - 1] * s[k]) ** 2 / (a[k] * s[k - 1]) ** 2)
    print(f'k = {k:4d}: weight {weight(beta):.4f} (nu = beta), '
          f'{weight(beta_tilde):.4f} (nu = beta_tilde), '
          f'half log-SNR step {dlam / 2:.4f}')
```

```
k =    2: weight 0.2708 (nu = beta), 0.5906 (nu = beta_tilde), half log-SNR step 0.3899
k =   10: weight 0.0725 (nu = beta), 0.0848 (nu = beta_tilde), half log-SNR step 0.0783
k =  100: weight 0.0100 (nu = beta), 0.0102 (nu = beta_tilde), half log-SNR step 0.0101
k =  500: weight 0.0055 (nu = beta), 0.0055 (nu = beta_tilde), half log-SNR step 0.0055
k = 1000: weight 0.0101 (nu = beta), 0.0101 (nu = beta_tilde), half log-SNR step 0.0100
```

ステップ幅が小さい領域では、$\nu_k = \tilde{\beta}_k$ に対する重みは対数信号対雑音比の半ステップ幅に一致し、$\nu_k = \beta_k$ に対する重みも先頭付近を除けばそれと変わらない。いずれも対数信号対雑音比が最も急激に低下する最初の数ステップで最大となり、出力された半ステップ幅を 2 倍すると 0.78

($k=2$) および 0.157 ($k=10$) に達する。等間隔のタイムステップからなるどのようなグリッドもこの性質からは逃れられない。定数の β でさえ対数信号対雑音比を少なくとも $\log(k/(k-1))$（ステップ k の値）だけ低下させ、これは $\log 2 \approx 0.69$（$k=2$）、$\log(10/9) \approx 0.105$（$k=10$）になる。本節の β は連鎖に沿って増加するため、最初の数ステップでの低下幅はさらに大きい。最初の数十ステップを過ぎると、どちらの重みも 0.01 付近かそれ以下にとどまる。ここで 17.2 節の単純損失（simple loss）は次式で定義される。

$$L_{\text{simple}}(\theta) = \mathbb{E}_{k,\mathbf{x}_0,\epsilon}\left\|\epsilon - \hat{\epsilon}_\theta(\alpha_k \mathbf{x}_0 + \sigma_k \epsilon, k)\right\|^2,$$

この損失は k を $\{1,\dots,T\}$ 上の一様分布から抽出するため、すべての重みを 1 に設定することになり、境界に比べてノイズの最も小さいステップを軽視する。それにもかかわらず、Ho et al. (2020) はこの損失のほうが良質なサンプルを生成することを見出した。これらのステップは画像の最も細かいディテールを担っており（17.6.3.2 節）、尤度はそれを漏れなく勘定に入れるが、人間の目にはほとんど知覚されないからである。

ステップ幅を限りなく小さくすると、この重み付き和は対数信号対雑音比に関する積分へと収束する。

$$L(\mathbf{x}_0) \to \frac{1}{2}\int_{\lambda_{\min}}^{\lambda_{\max}} \mathbb{E}_\epsilon\left\|\epsilon - \hat{\epsilon}_\theta(\mathbf{x}_\lambda, \lambda)\right\|^2 d\lambda + L_T + L_0, \tag{17.53}$$

ここで $\mathbf{x}_\lambda$ は対数信号対雑音比が λ である経路上の点であり (Kingma et al. 2021)、積分範囲は最小ノイズレベルにおける $\lambda_{\max}$ から経路終端の $\lambda_{\min}$ までである。言葉で言えば、この境界はノイズ誤差を重み 1 で対数信号対雑音比にわたって積分したものに、両端の 2 項を加えたものである。この極限が成り立つには λ に関してステップ幅が小さい必要があるが、離散連鎖のグリッドはデータ側の端でこの条件を満たさない。第 1 ステップから第 2 ステップにかけて対数信号対雑音比は $\log 2$ より大きく低下し、これは T がいくらであっても変わらないからである。その部分で離散連鎖の境界は緩くなる（演習問題 1）。

17.6.1.3 混合分布に対する数値評価

訓練済みネットワークの尤度を測る指標は 2 つある。境界 Equation 17.53 は確率的な逆過程に対応するものである。一方、17.3 節の確率フロー ODE はノイズからデータへの決定論的な写像であり、その密度は変数変換によって求まる。厳密な場に対しては両者は一致するが、訓練されたネットワークでは一致するとは限らない。そこで 17.2 節の混合分布のレシピに従い、VP 経路上でノイズ予測器を 20,000 ステップ訓練する。

```
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)

torch.manual_seed(0)
_, ema, _ = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(vp, 'eps'),
    lambda: mix.sample(512), 20_000, cosine=True, ema_half_life=700)
```

$d\mathbf{x}/dt = \mathbf{u}_t(\mathbf{x})$ の軌跡に沿うと、対数密度は変化率 $-\nabla\cdot\mathbf{u}_t$ で変化する。フローが点を押し広げる場所では密度が薄くなるためである。これは 38.3.3 節の瞬間的な変数変換にほかならない。微小な時刻 $t_{\min}$ におけるデータ点から、密度が標準ガウス分布となる $t=1$ まで積分すると、次式が得られる。

$$\log p(\mathbf{x}_{t_{\min}}) = \log \mathcal{N}(\mathbf{x}_1; \mathbf{0}, \mathbf{I}) + \int_{t_{\min}}^{1} \nabla \cdot \mathbf{u}_t(\mathbf{x}_t)\, dt.$$

`pf_nll` は、場が最も急激に変化するデータ側へ密に配置した 100 ステップ上で、点の位置と累積発散を同時に積分する。数値積分には古典的 4 次ルンゲ・クッタ法（38.2 節）を用いる。これは 17.5 節のホイン法の仲間であり、1 ステップあたり場を 4 回評価する。速度は 17.2 節の変換式を通じてノイズ予測から計算する。発散は 2×2 ヤコビ行列のトレースであり、座標ごとに 1 回のフォワードモード導関数で求まる。d 次元では 1 回の評価につき d 回の導関数計算が必要になるため、画像に対してはより低コストな推定法が必要となる（演習問題 3）。積分は $t_{\min} = 0.001$（ノイズが信号の約 1% となる時刻）から開始し、混合分布の厳密なノイズ予測を基準とする。

```
def eps_exact(x, t):
    """E[eps | x_t] of the mixture, from its posterior mean."""
    x0 = mix.posterior_mean(x, vp.alpha(t), vp.sigma(t))
    return vp.x0_eps('x0', x0, x, t)[1]

def pf_nll(eps_fn, path, x, t_min=1e-3, steps=100):
    """-log p(x) under the probability-flow ODE, RK4 from t_min to 1."""
    def f(x, t):                              # velocity and its divergence
        t = torch.full((len(x),), t, device=x.device)
        def u(x):                             # the velocity, by x0_eps, target
            x0, eps = path.x0_eps('eps', eps_fn(x, t), x, t)
            return path.target('velocity', x0, eps, t)
        e = torch.eye(x.shape[1], device=x.device)
        jvps = [torch.func.jvp(u, (x,), (e_i.expand_as(x),)) for e_i in e]
        return jvps[0][0], sum(j[1][:, i] for i, j in enumerate(jvps))
    ts = (t_min + (1 - t_min) * torch.linspace(0, 1, steps + 1) ** 2).tolist()
    div = torch.zeros(len(x), device=x.device)
    for ta, tb in zip(ts[:-1], ts[1:]):       # from the data to the noise
        h = tb - ta
        k1, d1 = f(x, ta)
        k2, d2 = f(x + h / 2 * k1, ta + h / 2)
        k3, d3 = f(x + h / 2 * k2, ta + h / 2)
        k4, d4 = f(x + h * k3, tb)
        x = x + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
        div = div + h / 6 * (d1 + 2 * d2 + 2 * d3 + d4)
    return (x ** 2).sum(1) / 2 + x.shape[1] / 2 * math.log(2 * math.pi) - div
```

ここで `mix` から 20,000 個のテスト点を抽出し、最初の 4,000 個に対して `pf_nll` を厳密なノイズ予測器 `eps_exact` と訓練済みネットワーク `ema` の両方で実行し、得られた確率フローの負の対数尤度を真の値 `-mix.log_prob(test)` と比較する。

```
torch.manual_seed(1)
test = mix.sample(20_000)
nll = -mix.log_prob(test)
mean_se = lambda v: f'{v.mean():.4f} +- {v.std() / math.sqrt(len(v)):.4f}'
with torch.no_grad():
    pf_exact, pf_net = (pf_nll(f, vp, test[:4000]) for f in (eps_exact, ema))
print(f'true NLL {mean_se(nll[:4000])} nats')
print(f'flow, exact field - true: {mean_se(pf_exact - nll[:4000])}')
```

```
print(f'flow, network - exact field: {mean_se(pf_net - pf_exact)}')
```

```
true NLL 2.4831 +- 0.0171 nats
flow, exact field - true: 0.0001 +- 0.0001
flow, network - exact field: 0.0018 +- 0.0013
```

厳密な場のフローは 1 点あたり約 2.5 nat という真の値を小数点以下約 4 桁まで再現しており、数値積分器の正確さが確認できる。訓練済みネットワークの負の対数尤度はそこから 1000 分の数 nat 以内に収まっており、対応のある差の標準誤差は約 1000 分の 1 nat である。ネットワークは単にサンプリングの方法だけでなく、確率密度そのものを学習しているのである。

境界の計算には両端の 2 項も必要である。事前項は約 $\frac{1}{2}\alpha_1^2\|\mathbf{x}_0\|^2 \approx 2\cdot 10^{-4}$ nat である。最終ステップは $\hat{\mathbf{x}}_0$ の周りに最小ノイズレベルの分散 $e^{-\lambda_{\max}} = \sigma^2/\alpha^2$ をもつガウス分布を出力し、$\mathbf{x}_0 - \hat{\mathbf{x}}_0 = (\sigma/\alpha)(\hat{\epsilon} - \epsilon)$ であるから、次式が成り立つ。

$$L_0 = \frac{d}{2}\log\left(2\pi e^{-\lambda_{\max}}\right) + \frac{1}{2}\,\mathbb{E}\left\|\epsilon - \hat{\epsilon}_\theta(\mathbf{x}_{t_{\min}}, t_{\min})\right\|^2.$$

セルの出力を見る前に、この内訳を見積もっておこう。2 次元でノイズレベルが 0.01 のとき、第 1 項は非常に幅の狭いガウス分布の対数正規化項であり、約 -7.3 nat となる。第 2 項は、これほど小さなノイズをデータ自身の変動と区別できないネットワークの期待二乗誤差の半分であるから約 1 となり、したがって $L_0 \approx -6.3$ となる。境界全体が 2.5 nat 付近にあるなら、積分項単体では約 $2.5 + 6.3 \approx 8.8$ nat でなければならない。どちらの部分も単独では尤度ではなく、その和だけが負の対数尤度の境界となる。

bound は、一様かつ層化した λ を用いてモンテカルロ法により積分を推定する。境界単体を 0.01 nat の精度で確定するには数百万回のネットワーク評価を要する。しかし 2 つの予測器の差であれば、両者に同じデータ点・ノイズレベル・ノイズを入力することで、モンテカルロノイズの大部分が共通となって相殺されるため、はるかに少ない評価回数で済む（36.5.2.6 節）。そのため bound は、共有された乱数抽出上で複数の予測器のリストを一括評価する。

```
@torch.no_grad()
def bound(eps_fns, path, x0, draws, t_min=1e-3):
    """Per-point bound on -log p(x0) for each eps(x, t), on shared draws,
    and its integral over the log-SNR alone."""
    n, d = x0.shape
    a1, s1 = (f(torch.tensor(1.0)).item() for f in (path.alpha, path.sigma))
    lam_max = path.log_snr(torch.tensor(t_min)).item()
    lam_min = 2 * math.log(a1 / s1)               # the end of the path
    scale = (lam_max - lam_min) / (2 * draws)
    prior = (((a1 * x0) ** 2).sum(1) / 2         # KL to N(0, I)
             + d / 2 * (s1 ** 2 - 1 - 2 * math.log(s1)))
    t = torch.full((n,), t_min, device=x0.device)
    eps = torch.randn_like(x0)
    x = path.sample(x0, t, eps)                  # the reconstruction term
    ends = [prior + d / 2 * math.log(2 * math.pi / math.exp(lam_max))
            + ((f(x, t) - eps) ** 2).sum(1) / 2 for f in eps_fns]
    integrals = [torch.zeros(n, device=x0.device) for _ in eps_fns]
    for _ in range(draws):                       # log-SNR uniform, stratified
        u = (torch.randperm(n, device=x0.device)
             + torch.rand(n, device=x0.device)) / n
```

```
            t = path.time_at(lam_max - u * (lam_max - lam_min))
            eps = torch.randn_like(x0)
            x = path.sample(x0, t, eps)
            for i, f in enumerate(eps_fns):
                integrals[i] += scale * ((f(x, t) - eps) ** 2).sum(1)
        return [e + i for e, i in zip(ends, integrals)], integrals

    (b_exact, b_net), (integral, _) = bound([eps_exact, ema], vp, test, 64)
    print(f'exact field: integral {integral.mean():.2f} nats, '
          f'prior and reconstruction {(b_exact - integral).mean():.2f} nats')
    print(f'bound, exact field - true: {mean_se(b_exact - nll)}')
    print(f'bound, network - exact field: {mean_se(b_net - b_exact)}')
```

```
exact field: integral 8.76 nats, prior and reconstruction -6.27 nats
bound, exact field - true: 0.0045 +- 0.0148
bound, network - exact field: 0.0047 +- 0.0005
```

セルの実行結果は予測を裏付けている。積分項は約 8.75 nat、両端の項は約 -6.3 nat である。厳密な場については、全 20,000 個のテスト点にわたって点ごとに対応をとった境界と真の負の対数尤度の差が標準誤差 0.015 の範囲内で 0 となっており、最良の予測器のもとでは連続時間境界にこの精度でのギャップは見られない。ネットワークの境界は厳密な場の境界より約 1000 分の 5 nat だけ大きく、この対応のある差は 10,000 分の 5 nat の精度で分解されている。2 つの指標はいずれも一致しており、この混合分布上でネットワークは最尤法の到達値から 1000 分の 5 nat 以内に迫っている。

17.6.1.4 単純損失と境界の比較

ここまでの結果だけでは、まだ冒頭の問いには答えられない。モデルの表現力に余裕があれば、すべてのノイズレベルを覆うどの重み付けも同じ最小点、すなわち厳密なノイズ予測をもつため、どれを使っても同程度に近づくからである。重み付けが違いを生むのは、ネットワークが小さすぎてあらゆる場所で厳密にはなれないときだけであり、そのとき重み付けは誤差をどこに割り振るかを決める。そこで、単純損失と境界自身の重み付け（λ が一様であり、`bound_loss` が `time_at` を通じて抽出する）の 2 通りで、元の幅と隠れユニット数 32 のそれぞれについて 2 つのシードからノイズ予測器を訓練する。実験の計算量を抑えるため、8 個のモデルはいずれも先ほどの半分の予算である 10,000 ステップ訓練する。結果を見る前に予想を立てておこう。元の幅では 2 つの重み付けはほぼ一致するはずである。一方、32 ユニットでは、境界の重み付けで訓練したモデルのほうが境界の値で優位に立つはずである。それを直接最小化するよう訓練されているからである。他方、単純損失はほぼクリーンな点への抽出を大幅に減らし、すべてのサンプルとフロー軌跡が通過するノイズの大きいレベルにより多くを割くため、そちらでは優位に立つ可能性がある。

```
def bound_loss(path, t_min=1e-3):
    """The noise regression with the log-SNR uniform: the bound's weighting."""
    lam_max, lam_min = (path.log_snr(torch.tensor(t)).item()
                        for t in (t_min, 1.0))
    def loss_fn(net, x0):
        u = torch.rand(len(x0), device=x0.device)
        t = path.time_at(lam_max - u * (lam_max - lam_min))
        eps = torch.randn_like(x0)
        return ((net(path.sample(x0, t, eps), t) - eps) ** 2).sum(1).mean()
    return loss_fn
```

```
losses = {'simple': d2l.regression_loss(vp, 'eps'), 'bound': bound_loss(vp)}
nets = {}
for width in (256, 32):
    for name, loss_fn in losses.items():
        for seed in (0, 1):
            torch.manual_seed(seed)
            nets[width, name, seed] = d2l.train_regression(
                d2l.TimeMLP(width, 3).to(device), loss_fn,
                lambda: mix.sample(512), 10_000, cosine=True,
                ema_half_life=700)[1]
```

各モデルは次の 3 通りで評価する。テスト点上で厳密な場との対応のある差として測った境界およびフロー尤度、そして共有ノイズから祖先連鎖（ancestral chain）で抽出し真の密度のもとで評価した生成サンプルである。サンプルについては、真の密度のもとでの平均負の対数尤度と、どのモードからも標準偏差の 3 倍以上離れた逸脱点（stray）の割合を報告する。データ自身の値は約 2.47 nat および 1% である。

```
b_all, _ = bound([eps_exact, *nets.values()], vp, test, 64)
torch.manual_seed(2)
z = torch.randn(20_000, 2, device=device)        # shared by every model
rows = {}
for (width, name, _), net, b_net in zip(nets, nets.values(), b_all[1:]):
    with torch.no_grad():
        flow = pf_nll(net, vp, test[:4000])
        torch.manual_seed(3)  # the same injected noise for every model
        x = d2l.ancestral_sample(d2l.make_pred(net, vp, 'eps'), z, vp, 1000)
    rows.setdefault((width, name), []).append(
        ((b_net - b_all[0]).mean(), (flow - pf_exact).mean(), mix.nll(x),
         mix.summary(x)['stray']))
print('width loss    bound - exact  flow - exact  samples: NLL  strays'
      '   (seeds 0, 1)')
for (width, name), (r0, r1) in rows.items():
    print(f'{width:5} {name:7}' + ''.join(
        f'{v0:7.4f}{v1:7.4f}' if i < 2 else f'{v0:7.3f}{v1:7.3f}'
        for i, (v0, v1) in enumerate(zip(r0, r1))))
print(f'data {"":36}{mix.nll(test):7.3f}{mix.summary(test)["stray"]:14.3f}')
```

```
width loss    bound - exact  flow - exact  samples: NLL  strays   (seeds 0, 1)
  256 simple  0.0095 0.0080 0.0052 0.0050  2.488  2.502  0.011  0.013
  256 bound   0.0065 0.0065 0.0119 0.0161  2.503  2.498  0.013  0.012
   32 simple  0.0379 0.0308 0.0253 0.0346  2.535  2.534  0.018  0.018
   32 bound   0.0268 0.0201 0.0359 0.0501  2.564  2.552  0.022  0.021
data                                       2.476         0.012
```

元の幅では重み付けの影響はほとんどない。4 つのモデルはいずれも両方の指標で厳密な場から 100 分の 2 nat 以内に収まり、生成サンプルも区別がつかず、逸脱点の割合もデータ自身に近い。もっとも、それぞれの重み付けは自身が対象とする指標ではすでにわずかに優位に立っており、境界では 1000 分の 1〜5 nat、フローでは 1000 分の 5〜10 nat の差がついている。32 ユニットになるとこの差が広がり、サンプル品質にも及ぶ。境界重み付けモデルのほうが境界は 100 分の 1〜2 nat 低い。一方、尤度の境界ではない単純損失のほうがフロー尤度は良好で、負の対数尤度が 100 分の 1〜2 nat 低く、サンプルも優れている。すなわち、サンプルの平均負の対数尤度は 100 分の 2〜6 nat 低く、逸脱点の割合も約 1.5〜1.8%（境界重み付けでは 2.0〜2.3%）と少な

い。この混合分布では、境界で訓練しても小規模ネットワークに最良の尤度はもたらされなかったのである。その理由は予想どおり、訓練時の抽出がどこに向かうかにあると考えられる。λ を $[-10.05, 9.12]$ 上で一様に抽出すると、境界の抽出の約 $7/19$（3 分の 1 超）が、点がほぼクリーンな $\lambda = 2$ より上に費やされるのに対し、VP 経路上で一様な t を抽出する場合はそこには約 10 分の 1 しか割かれない。

したがって、冒頭の問いに対する答えは条件付きの「否」である。単純損失は尤度境界そのものではなく、境界の被積分関数に別の重みを掛けたものである。表現力に余裕があれば、重みの違いがどちらの尤度を動かす幅も高々 100 分の 1 nat 程度にとどまるが、表現力が限られるときには、より良い境界をとるか、それともより良いフロー尤度とサンプルをとるかを重みが左右する。

17.6.2 重み付けとしての目的関数

境界は対数信号対雑音比の単位量あたりに等しい重みを置き、単純損失は連鎖の各ステップに等しい重みを置き、フローマッチングは別の経路上で速度を回帰する。共通の変数で表せば、これらは 3 通りの重み付けをもつ 1 つの目的関数である。

17.6.2.1 対数信号対雑音比による形式

これまで扱ってきた各損失は、時刻 t を抽出して $\mathbf{x}_t$ を作り、ターゲットとの二乗誤差をとる。固定した $\mathbf{x}_t$ において、どのターゲットもノイズ予測のアフィン関数であり、そのスカラー傾きは t のみに依存する。したがって、各二乗誤差は $c(t)\,\|\hat{\epsilon} - \epsilon\|^2$（係数 $c(t)$ はターゲットと経路で決まる）と書ける。変数を t から λ に変換すると、Kingma and Gao (2023) の形式が得られる。

$$\begin{aligned} \mathcal{L}_w &= \frac{1}{2}\int_{\lambda_{\min}}^{\lambda_{\max}} w(\lambda)\,\mathbb{E}\big\|\hat{\epsilon}_\theta(\mathbf{x}_\lambda, \lambda) - \epsilon\big\|^2\, d\lambda, \\ w(\lambda) &= 2\,c(t)\,p(t)\left|\frac{dt}{d\lambda}\right|, \end{aligned} \tag{17.54}$$

ここで $p(t)$ は訓練時刻の密度である。このように、損失は各対数信号対雑音比におけるノイズ誤差をどれだけ重視するかを表す 1 つの関数によって完全に記述される。境界 Equation 17.53 は重み付け $w \equiv 1$ に対応する。

積分内部の期待値は経路に依存しない。同じ λ であれば、2 つの経路は定数倍を除いて同じノイズ付きデータ点を与えるため（17.2 節）、ネットワークは同じ予測問題に直面するからである。したがって、スケジュールが $\mathcal{L}_w$ に関与するのは重み付けと両端 $\lambda_{\min}$、$\lambda_{\max}$ を通じてのみであり、境界については両端が同じスケジュールなら同じ損失を与える (Kingma et al. 2021)。

最小の具体例として、線形経路上で一様な t を用いるノイズ損失（$c = p = 1$）を考えよう。このとき $\sigma_t/\alpha_t = t/(1-t) = e^{-\lambda/2}$ であるから、$t = 1/(1+e^{\lambda/2})$ となり、$|dt/d\lambda| = \frac{1}{8}\,\mathrm{sech}^2(\lambda/4)$ が得られる。重み付け $w(\lambda) = \frac{1}{4}\,\mathrm{sech}^2(\lambda/4)$ は、信号とノイズの強さが等しくなる $\lambda = 0$ を中心とする山型の関数であり、両裾では $e^{-|\lambda|/2}$ のように減衰する。

17.6.2.2 既存の目的関数における重み付け

他の目的関数については 2 つの変換が必要になる。VP 経路上では、Salimans and Ho (2022) の $\mathbf{v}$ ターゲットは $\hat{\mathbf{v}} - \mathbf{v} = (\hat{\epsilon} - \epsilon)/\alpha_t$ を満たすため、$c = 1/\alpha_t^2 = 1 + e^{-\lambda}$ となる。線形経路上の速度も同じ関係式を満たし、そこでは $1/\alpha_t = 1 + e^{-\lambda/2}$ である。一様な t のもとで、定数倍を除いて次の表の重み付けが得られる（演習問題 2 で導出する）。独自にノイズレベルを抽出する EDM の行には、後ほど EDM とともに定義するデータのスケール σ_{data} が現れる。

目的関数	経路とノイズレベル	$w(\lambda)$
境界（ELBO）	任意の経路	1
ノイズ、DDPM	VP、線形 β、t 一様	$\lvert dt/d\lambda \rvert$
ノイズ	線形、t 一様	$\operatorname{sech}^2(\lambda/4)$（$\lambda=0$ に山）
ノイズ	コサイン、t 一様	$\operatorname{sech}(\lambda/2)$
v	コサイン、t 一様	$e^{-\lambda/2}$
速度（フローマッチング）	線形、t 一様	$e^{-\lambda/2}$
EDM	$\log(\sigma_t/\alpha_t) \sim \mathcal{N}(-1.2, 1.2^2)$	$\mathcal{N}(\lambda; 2.4, 2.4^2)\,(e^{-\lambda}+\sigma_{\text{data}}^2)$

この種の式変形は定数倍を誤りやすいため、表の 2 行と DDPM の重み付けのピークをコンピュータで確認しておこう。`weighting` は代数計算を行わずに $w(\lambda)$（一様な t の場合）を求める。係数 $c(t)$ は予測ノイズに対するターゲットの傾きの二乗であり、`x0_eps` と `target` を用いて固定された $\mathbf{x}_t$ 上で計算できる。また $d\lambda/dt$ は `rates` と同様にフォワードモード自動微分で求まる。

```
def weighting(path, kind, lam):
    """w(lambda) = 2 c(t) |dt / dlambda| for t uniform, where c is the squared
    slope of the target in the predicted noise at a fixed x_t."""
    t = path.time_at(lam)
    x, zero, one = (torch.full((len(t), 1), v) for v in (0.0, 0.0, 1.0))
    def target(eps_hat):                        # the target that eps_hat implies
        return path.target(kind, *path.x0_eps('eps', eps_hat, x, t), t)
    c = ((target(one) - target(zero)) ** 2)[:, 0]
    dlam_dt = torch.func.jvp(path.log_snr, (t,), (torch.ones_like(t),))[1]
    return 2 * c / dlam_dt.abs()

lam = torch.linspace(-8, 8, 9)
w_u = weighting(d2l.GaussianPath.linear(), 'velocity', lam)
w_v = weighting(d2l.GaussianPath.cosine(), 'v', lam)
row = lambda name, w: print(f'{name:26}' + ''.join(f'{v:8.3f}' for v in w))
print(f'{"lambda":26}' + ''.join(f'{v:8.0f}' for v in lam.tolist()))
row('velocity, linear path', w_u.tolist())
row('  times e^(lambda / 2)', (w_u * (lam / 2).exp()).tolist())
row('v, cosine path', w_v.tolist())
row('  divided by the row above', (w_v / w_u).tolist())
fine = torch.linspace(-9, 9, 1801)
peak = fine[weighting(vp, 'eps', fine).argmax()]
print(f'noise, VP path: largest weight at lambda = {peak:.1f}')
```

```
lambda                         -8      -6      -4      -2       0       2       4       6       8
velocity, linear path      54.598  20.086   7.389   2.718   1.000   0.368   0.135   0.050   0.018
  times e^(lambda / 2)      1.000   1.000   1.000   1.000   1.000   1.000   1.000   1.000   1.000
v, cosine path             35.034  12.888   4.741   1.744   0.642   0.236   0.086   0.031   0.010
  divided by the row above  0.642   0.642   0.642   0.642   0.642   0.641   0.639   0.622   0.530
noise, VP path: largest weight at lambda = -0.9
```

速度の重み付けは表示されたすべての桁で $e^{-\lambda/2}$ に一致し、**v** の重み付けも同じ曲線を 0.64 倍したもの（$\lambda = -8$ から 4 の範囲）になっている。ノイズ側の端で 0.642 となるこの定数は $2 \cdot 1.008/\pi$ である。本節のコサイン経路は Nichol and Dhariwal (2021) によるもので、$\alpha_t \propto \cos\left(\frac{\pi}{2}(t+0.008)/1.008\right)$ のように小さなオフセット 0.008 を入れて時刻 0 で 1 から始まるよう再スケールしているため、その位相角は $\cos\frac{\pi t}{2}$ よりわずかに遅く進む。このオフセットが効くのは点がほぼクリーンな領域だけであり、そこでは比が 0.62（$\lambda = 6$）、さらに 0.53

($\lambda = 8$) へと低下する。したがって、定数倍とそのクリーン側の端を除けば、コサイン経路上の $\mathbf{v}$ 予測と線形経路上のフローマッチングは同じ目的関数である (Gao et al. 2024)。VP 経路上のノイズ損失は、信号とノイズが等しくなる点よりわずかにノイズ寄りの $\lambda = -0.9$ でピークをとる。17.2 節の速度モデルとノイズモデルは、何を学習するかではなく、その重み付けが異なるのである（Figure 17.10）。

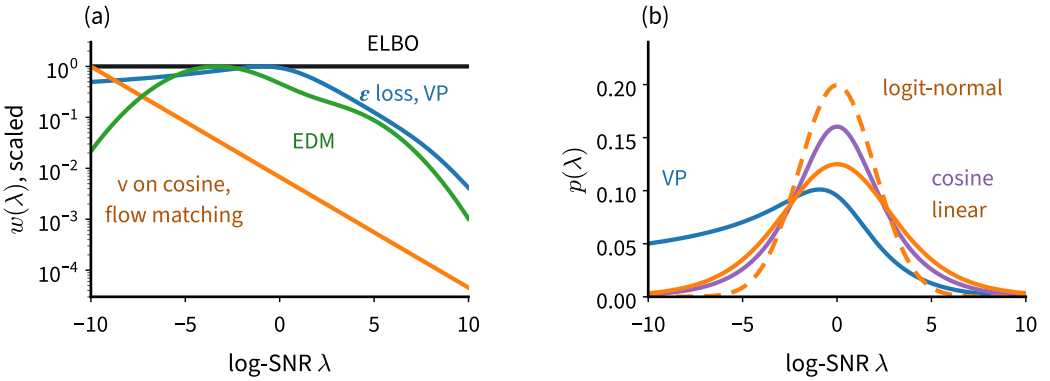


Figure 17.10: 対数信号対雑音比 λ にわたるノイズ誤差の重み付けとしての拡散モデルの目的関数。(a) 対数軸上の $w(\lambda)$。境界は平坦であり、線形 β の VP 経路上のノイズ損失は $\lambda \approx -1$ 付近でピークをとり、$\mathbf{v}$ 予測とフローマッチングは曲線 $e^{-\lambda/2}$ を共有し、EDM の重み付け（論文の値 $\sigma_{\text{data}} = 0.5$ を使用）は山型となる。(b) λ の密度（3 つの経路上で一様な t をとった場合、および線形経路上で Esser et al. (2024) のロジット正規時刻をとった場合）。

これらの重み付けのうち、どれが尤度境界になっているのだろうか。驚くべきことに、1 つだけではない。あるレベル μ 以下で 1、それより上で 0 となる重み付けは、対数信号対雑音比 μ までノイズを加えたデータに対する境界 Equation 17.53 である。そのようなデータは μ から経路を開始するため、境界には μ 以下のレベルだけが残るからである。λ に関して単調減少する重み付けは、こうした階段関数の正の混合として表せる。したがって、そのような $\mathcal{L}_w$ はランダムなレベルのガウスノイズでデータ拡張したデータに対する境界となる (Kingma and Gao 2023)。$w = e^{-\lambda/2}$ の場合、混合密度 $-w'(\mu) = \frac{1}{2}e^{-\mu/2}$ は低い対数信号対雑音比に向かって増大するため、$\mathbf{v}$ 予測とフローマッチングは、ランダムなレベル（主に強いノイズ）までノイズを加えたデータに対する尤度訓練に相当する。これら強いノイズに耐えて残る大まかな構造を重点的に訓練し、尤度を支配する最も細かいディテールにはほとんど重みを置かない。一方、DDPM のノイズ損失と EDM の重み付けはいずれもピークをもち、この種の境界にはまったく当てはまらない。

17.6.2.3 重要度サンプリングとしての時刻分布

訓練時刻は、積 $w(\lambda) = 2c\,p\,|dt/d\lambda|$ を通じてのみ Equation 17.54 に現れる。原理的には、訓練時刻の密度と重み付けは独立した選択であり、λ を範囲全体を覆う任意の密度 $p(\lambda)$ から抽出すれば、次式が成り立つ。

$$\mathcal{L}_w = \mathbb{E}_{\lambda\sim p}\left[\frac{w(\lambda)}{2\,p(\lambda)}\,\mathbb{E}\left\|\hat{\epsilon}_\theta(\mathbf{x}_\lambda, \lambda) - \epsilon\right\|^2\right],$$

これは 36.8.2 節の重要度サンプリング（importance sampling）の恒等式であり、$p(\lambda)$ は推定量の分散を変えるにすぎない。それでもこの選択は重要である。一様な t を線形 β の VP 経路上でとると、抽出のほぼ半分が $\lambda = -3$ 未満に費やされるが（Figure 17.10 のパネル (b)）、そこで

は境界の被積分関数は小さい。その領域では $\mathbf{x}_t$ がほぼノイズそのものであり、そこからノイズを予測するのは容易だからである。この理由から Nichol and Dhariwal (2021) はコサイン経路へ移行した。さらに境界自体を推定する際には、重要度サンプリングによってステップを抽出した（演習問題 1）。

実際には $p(\lambda)$ で割ることは行われないため、時刻分布は形を変えた重み付けとして働く。よく使われるのが Esser et al. (2024) の**ロジット正規分布**（logit-normal）であり、$t = 1/(1+e^{-(m+sn)})$（ただし $n \sim \mathcal{N}(0,1)$）と定める。中心 m は経路に沿って時刻を移動させ、広がり s は集中の度合いを決め、$m = 0$ では $t = \frac{1}{2}$ の周りに時刻を集中させる。$m = 0$ で密度による除算を行わずに用いると、損失の重み付けに経路中央付近を中心とする山型の関数を掛けることになる。

17.6.3 設計空間

拡散モデルの論文はそれぞれ、経路、ターゲット、重み付け、訓練ノイズレベル、ネットワーク、サンプラを固定している。Karras et al. (2022) はこれらの選択を切り分け、その大半を互いに独立に選べることを示した。これにより、各要素を 1 つずつ比較できるようになる。

これらの選択の 1 つであるターゲットは、すべてのターゲットが同じノイズ誤差への重み付けに書き換えられた前項で消え去ったかのように見える。しかし、重み付けの観点が述べるのは損失を最小化する関数が何かということだけであり、有限サイズのネットワークにとってその関数を出力することがどれだけ難しいかや、ある出力を別の出力へ変換するときに誤差がどれだけ拡大するかについては何も語らない。したがって、同じ重み付けをもつ 2 つの損失でも異なるネットワークが訓練されうるのであり（演習問題 2 の最後の小問でその違いを調べる）、ネットワークに何を出力させるかは依然として重要である。

17.6.3.1 EDM の比較表

自身の手法を EDM と呼ぶ Karras et al. の列に沿って整理すると、Table 17.2 の 3 つのモデルと EDM 自体は次のように表される。

手法	訓練：経路、ノイズレベル、重み付け	ネットワーク出力	サンプラ
DDPM	VP、線形 β、t 一様、単純損失	ノイズ	祖先連鎖、1,000 ステップ
NCSN	$\mathbf{x}_0 + \sigma\epsilon$、$\sigma$ の等比列、重み σ^2	スコア関数	アニーリング付きランジュバン動力学
フローマッチング	線形、t 一様	速度	オイラー法または適応的 ODE ソルバ
EDM	$\log\bar{\sigma} \sim \mathcal{N}(-1.2, 1.2^2)$、重み $1/c_{\text{out}}^2$	前処理付きクリーン画像	$\rho = 7$ グリッド上のホイン法

NCSN (Song and Ermon 2019)（38.6.3 節）はデータを一切スケールしない（$\alpha = 1$）。これは**分散爆発**（variance-exploding）の観点である。EDM は、どの経路に対しても $\bar{\mathbf{x}} = \mathbf{x}/\alpha$ とノイズレベル $\bar{\sigma} = \sigma/\alpha$ というこの座標系で、`ode_sample` と同様に動作する。

EDM が新たに加えた列はこのパラメータ化、すなわちネットワークが何を出力し、その入出力をどのようにスケールするかである。これを決める観点は 2 つある。第 1 はスケールである。ネットワーク F はすべてのノイズレベルで重みを共有するため、どのレベルでも単位分散の入力を受け取り、単位分散の出力を生成するのが望ましい。平均 0、標準偏差 σ_{data} のデータに対して、ノイズ除去器（denoiser）を次式で定める。

$$
\begin{aligned}
D(\bar{\mathbf{x}};\bar{\sigma}) &= c_{\text{skip}}\,\bar{\mathbf{x}} + c_{\text{out}}\,F(c_{\text{in}}\,\bar{\mathbf{x}};\,\bar{\sigma}),\\
c_{\text{in}} &= \frac{1}{\sqrt{\bar{\sigma}^2+\sigma_{\text{data}}^2}},\quad c_{\text{skip}} = \frac{\sigma_{\text{data}}^2}{\bar{\sigma}^2+\sigma_{\text{data}}^2},\\
c_{\text{out}} &= \frac{\bar{\sigma}\,\sigma_{\text{data}}}{\sqrt{\bar{\sigma}^2+\sigma_{\text{data}}^2}},
\end{aligned}
\tag{17.55}
$$

そして $\|D-\mathbf{x}_0\|^2$ に重み $1/c_{\text{out}}^2$ を掛けて訓練することで、F のターゲットも単位分散にする（各係数は演習問題 6 で導出する）。低ノイズ領域ではノイズ除去器は入力をそのまま通し（$c_{\text{skip}}\to 1$、$c_{\text{out}}\to 0$）、F はその補正のみを担う。高ノイズ領域では、出力は F の推定値をデータのスケールに合わせたものになる（$c_{\text{skip}}\to 0$、$c_{\text{out}}\to\sigma_{\text{data}}$）。中心化されていないデータに対しても、二乗平均平方根（root mean square）を σ_{data} の代わりに用いれば同じ導出がそのまま成り立つ。導出中のすべての分散が 0 周りの二次モーメント（second moment）に置き換わるためである。Karras et al. は自然画像に対して $\sigma_{\text{data}}=0.5$ を用いているが、以下では本節のデータ上でその値を実際に計測する。

このスケールの議論からは、素のノイズ予測器で何が問題になるかもわかる。素のノイズ予測器は信号係数で割ることで出力をクリーン画像に変換するが（$\hat{\mathbf{x}}_0=(\mathbf{x}_t-\sigma_t\hat{\epsilon})/\alpha_t$）、VP 経路上では $\alpha_1\approx 0.007$ であるため、純粋なノイズ付近ではこの除算によってノイズ誤差が約 150 倍に拡大される。その領域では入力をそのままコピーするだけでもノイズ損失はほぼ満たされてしまうため、除算が必要とする 150 分の 1 未満まで誤差を押し下げる動機が損失には働かない。EDM は小さな数で割ることを一切しない。

第 2 の観点は、有限サイズのネットワークが何を表現できるかである。画像はピクセル数よりはるかに低次元の集合の近くに分布しているのに対し、ノイズはあらゆる次元を埋め尽くす。わずかな信号から画像全体を生成しなければならない高ノイズ領域では、クリーン画像のほうが出力しやすい対象であり、そこでは $c_{\text{skip}}\to 0$ となって F はスケールされたクリーン画像を予測する。一方、低ノイズ領域では F はノイズの定数倍、すなわち入力に対する小さな補正を予測する。Li and He (2026) はこの議論を極限まで推し進め、大きなピクセルパッチを扱う Transformer がノイズ予測や速度予測では失敗する一方、クリーン画像の予測では機能することを示した。

EDM は経路の座標系においてネットワーク $F(\mathbf{x},t)$ をこの前処理で包み（$c_{\text{in}}\bar{\mathbf{x}}=\mathbf{x}/n$、ただし $n^2=\sigma^2+\alpha^2\sigma_{\text{data}}^2$ であり $\alpha=0$ でも有限にとどまる）、クリーン画像を予測する。`edm_loss` は対数正規分布からノイズレベルを抽出し、重みを適用する。なお Karras et al. はノイズレベルを $\frac{1}{4}\log\bar{\sigma}$ としてネットワークに入力しているが、以下の比較で出力だけを変えられるよう、本節の EDM は本章の他のすべてのモデルと同様に経路の時刻 t を渡す。

```
class EDM(nn.Module):
    """Karras et al.'s preconditioned denoiser on a path; returns x0-hat."""
    def __init__(self, net, path, sigma_data=0.5):
        super().__init__()
        self.net, self.path, self.sigma_data = net, path, sigma_data

    def forward(self, x, t):
        a = self.path.alpha(t).reshape(-1, 1, 1, 1)
        s = self.path.sigma(t).reshape(-1, 1, 1, 1)
        sd = self.sigma_data
        n = torch.sqrt(s ** 2 + (a * sd) ** 2)
        return a * sd ** 2 / n ** 2 * x + s * sd / n * self.net(x / n, t)
```

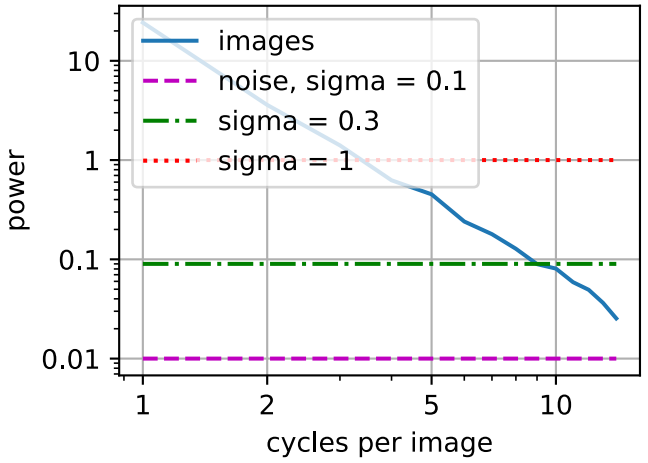
```
    def edm_loss(edm, x0):
        """Log-normal noise levels sigma_bar, loss weight 1 / c_out^2."""
        sigma_bar = torch.exp(-1.2 + 1.2 * torch.randn(len(x0), device=x0.device))
        t = edm.path.time_at(-2 * torch.log(sigma_bar))
        x = edm.path.sample(x0, t, torch.randn_like(x0))
        sd = edm.sigma_data
        weight = (sigma_bar ** 2 + sd ** 2) / (sigma_bar * sd) ** 2
        return (weight.reshape(-1, 1, 1, 1) * (edm(x, t) - x0) ** 2).mean()
```

17.6.3.2 ノイズレベルと画像のスケール

あるノイズレベルは画像から何を消し去るのだろうか。正規直交 2 次元フーリエ変換（Fourier transform）のもとでは、標準偏差 σ の白色ノイズはあらゆる周波数で一様なパワー σ^2 をもつのに対し、画像のパワーは低周波に集中している。そのため、ノイズのパワーが画像のパワーを上回った周波数成分はノイズに埋もれて見えなくなる。次のセルでは、画像あたり r サイクルの半径をもつリング上で平均したテスト画像のパワーを、3 つのノイズレベルと比較してプロットする。

```
    X, y, X_test, _ = d2l.load_fashion_mnist_pm1()
    spec = torch.fft.fft2(X_test[:, 0], norm='ortho')  # noise: power sigma^2
    k = torch.fft.fftfreq(28, 1 / 28, device=X.device)
    ring = torch.sqrt(k[:, None] ** 2 + k ** 2).round().long().flatten()
    power = (torch.bincount(ring, (spec.abs() ** 2).mean(0).flatten())
             / torch.bincount(ring))                  # the mean over each ring
    r, power = torch.arange(1, 15), power[1:15].cpu()
    d2l.plot(r, [power] + [torch.full((14,), s ** 2) for s in (0.1, 0.3, 1.0)],
             'cycles per image', 'power', xscale='log', yscale='log',
             legend=['images', 'noise, sigma = 0.1', 'sigma = 0.3', 'sigma = 1'])
    for s in (0.1, 0.3, 0.5, 1.0, 2.0):
        hidden = (power < s ** 2).nonzero()
        print(f'sigma = {s}: noise dominates from', f'{r[hidden[0]].item()} cycles'
              if len(hidden) else 'no ring below 15')
```

```
sigma = 0.1: noise dominates from no ring below 15
sigma = 0.3: noise dominates from 9 cycles
sigma = 0.5: noise dominates from 6 cycles
sigma = 1.0: noise dominates from 4 cycles
sigma = 2.0: noise dominates from 2 cycles
```



画像のパワーは、画像あたり 1 サイクルから 14 サイクルにかけて 3 桁低下する。$\sigma = 0.1$ のノイズは全領域で画像のパワーを下回るが、$\sigma = 0.3$ では画像あたり約 9 サイクルより外側のリングが隠れ、$\sigma = 1$ では約 4 サイクルから外側が隠れる。したがって、ノイズを加える操作は σ が大きくなるにつれてカットオフ周波数が下がるローパスフィルタとして働き、サンプリングは粗い構造から細かい構造へと進む。高ノイズ側のステップが衣服のシルエットを決め、最後のステップがその質感を決めるのである (Dieleman 2024)。

同じ見方から、なぜスケジュールが画像解像度に依存するかも説明できる。ピクセルあたりのノイズが同じまま 56×56 にすると、各 2×2 ブロックの平均をとることで 28×28 の画像が復元される一方、4 つのノイズ値の平均のノイズ分散は $\sigma^2/4$ に下がる。つまり大まかな構造に対する信号対雑音比が 4 倍に増大するため、ある解像度で調整したスケジュールはより高い解像度では簡単になりすぎるのである。そこで Hoogeboom et al. (2023) は、ピクセル数の比の対数だけ λ_t を下方にシフトさせる（演習問題 4、Figure 17.11）。

ブロック平均はまた、スケジュールの終端がわずかに残した信号をも増幅する。本節の VP 経路は $\lambda \approx -10$ で終わり、そこでは全 784 ピクセルの平均の信号対雑音比は高々 $784\, e^{-10} \approx 0.04$ であり、読み取るには小さすぎる。しかし Stable Diffusion のスケジュールは、対数信号対雑音比が約 -5 のところで止まっており、チャネルあたり 64×64 個の値をもつ潜在表現上ではチャネル平均の信号対雑音比が $4096\, e^{-5} \approx 28$ に達する。モデルはこの生き残った明るさに依存するよう学習してしまい、サンプラは平均 0 のノイズから出発するため、非常に明るい画像や非常に暗い画像を生成できなくなった (Lin, Liu, et al. 2024)。

17.6.3.3 1 要因ずつの比較

いよいよ画像上でこれらの設計選択を 1 つずつ検証しよう。ベースとするのは 17.4 節のレシピ、すなわち線形経路上で t を一様とした速度予測であり、他の各行はそこから 1 つの要素だけを変更する。

行	ベースからの変更点
`t logit-normal`	$m = 0$、$s = 1$ のロジット正規時刻（Esser et al. (2024) の選択）
`t logit-normal, m = -0.5`	同上、中心をデータ寄りの $t \approx 0.38$ に設定
`cosine path, v`	コサイン経路上の $\mathbf{v}$ 予測（重み付けの表よりベースと同じ目的関数）

行	ベースからの変更点
`t at EDM's levels`	EDM の対数正規ノイズレベル（次の行に対する対照群）
`EDM, clean image`	EDM の前処理付きクリーン画像（EDM のノイズレベルと重みを使用）

各モデルはクラス条件なしの 64 チャネル `d2l.UNet` であり、5,000 回の更新（17.4 節の予算の 3 分の 1）を 2 つのシードから訓練する（ただし比較の読み取りが最も微妙なベース、対照群、EDM には 4 シードを用いる）。すべてのモデルについて、同じ 5,000 枚のノイズ画像からホイン法でサンプリングし、5,000 枚のテスト画像に対するフレシェ距離（Fréchet distance、FD）で評価する。ここでサンプラの選択には公平性（fairness）の問題が生じる。EDM の原論文は独自の $\rho = 7$ グリッド上でサンプリングするが、一様な時刻で訓練したモデルに対しては、この計算予算ではそのグリッドが線形グリッドを上回らないことが 17.5 節でわかっている。それでも EDM のノイズレベルで訓練したモデルには適している可能性があり、それを対照群の行で検証できる。したがって単一のグリッド上での比較は特定のレシピに有利に働きかねないため、すべてのモデルを EDM のグリッド（49 回評価）と線形グリッド（48 回評価）の両方でサンプリングする。次のセルではまず、EDM の σ_{data} のためにピクセル値を計測し、あわせて 17.4.3.2 節の下限値、すなわち実画像 2 セット間の FD を求める。

```
sigma_data = X.pow(2).mean().sqrt().item()     # root mean square of a pixel
print(f'pixels: mean {X.mean():.2f}, standard deviation {X.std():.2f}, '
      f'root mean square {sigma_data:.2f}')
cnn = d2l.fashion_feature_cnn(X, y)
f_test = d2l.fd_features(cnn, X_test[:5000])
floor = d2l.fid_score(f_test, d2l.fd_features(cnn, X_test[5000:]))
print(f'FD, real against real: {floor:.2f}')
```

```
pixels: mean -0.43, standard deviation 0.71, root mean square 0.83
FD, real against real: 0.18
```

Fashion-MNIST は背景の大部分が黒であるため、ピクセルの平均は -0.43、二乗平均平方根は 0.83 と論文の値 0.5 を大きく上回っており、セルはこの実測値を `EDM` に渡す。5,000 枚の実画像 2 セット間の FD は 0.2 をわずかに下回る。

`logit_normal_loss` は、`regression_loss` の一様な時刻をロジット正規時刻に置き換えたものである。線形経路上では $\bar{\sigma} = t/(1-t)$ であるから $\log \bar{\sigma} = m + sn$ となり、$m = -1.2$、$s = 1.2$ とおけば速度モデルも EDM の対数正規ノイズレベルで訓練される。したがって、その名前の行と EDM の行との違いはパラメータ化と重みだけである。これはまた、$m = -0.5$ の行よりもさらにデータ寄りの $t \approx 0.23$ を中心とするロジット正規分布でもある。

```
def logit_normal_loss(path, kind, m=0.0, s=1.0):
    """regression_loss with logit-normal times t = sigmoid(m + s N(0, 1))."""
    def loss_fn(net, x0):
        t = torch.sigmoid(m + s * torch.randn(len(x0), device=x0.device))
        eps = torch.randn_like(x0)
        error = net(path.sample(x0, t, eps), t) - path.target(kind, x0, eps, t)
        return (error ** 2).flatten(1).sum(1).mean()
    return loss_fn

linear, cosine = d2l.GaussianPath.linear(), d2l.GaussianPath.cosine()
```

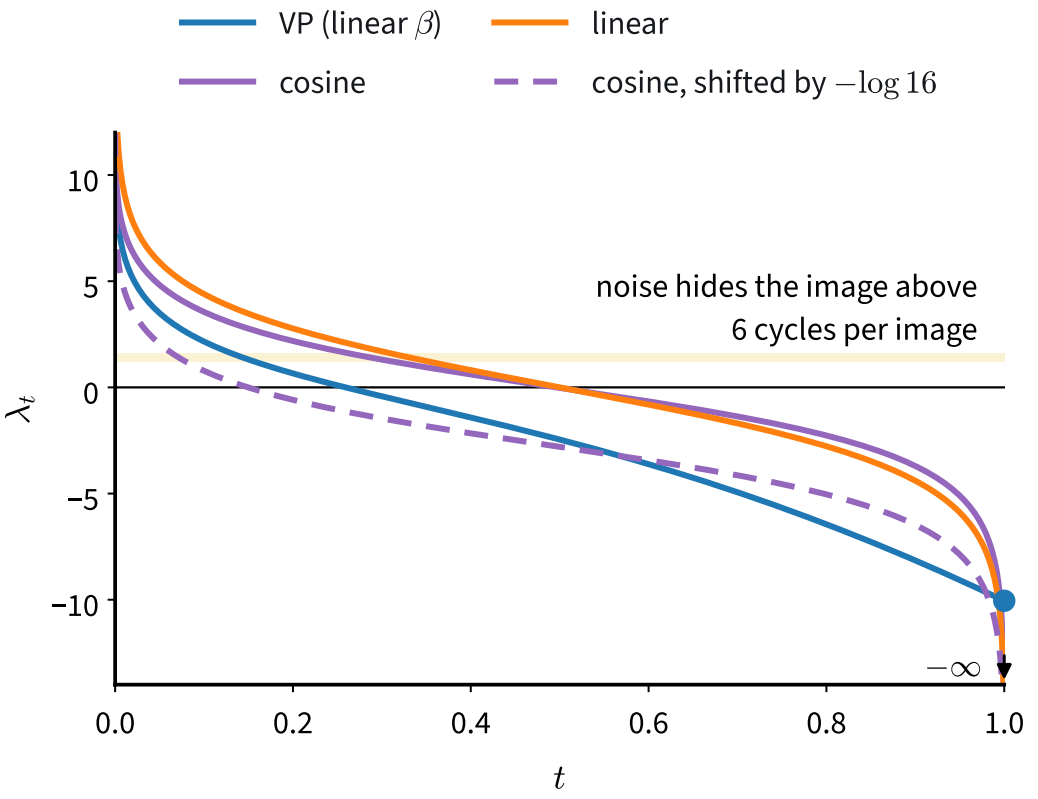


Figure 17.11: 対数信号対雑音比の時計で見たノイズスケジュール。λ_t を t に対してプロットしたもので、線形 β の VP 経路、コサイン経路、線形経路、および 1 辺の長さを 4 倍にした場合のシフト付きコサイン経路（$\lambda_t - \log 16$）を示す。VP 経路は $\lambda \approx -10$ で終わり、他は $-\infty$ で終わる。帯状の領域は、Fashion-MNIST において画像あたり 6 サイクル超の成分がノイズに隠れる対数信号対雑音比を表す。

```
unet = lambda: d2l.UNet(ch=64)
recipes = {  # name: (path, kind of prediction, network, loss)
    'base: velocity, t uniform': (linear, 'velocity', unet,
                                  d2l.regression_loss(linear, 'velocity')),
    't logit-normal': (linear, 'velocity', unet,
                       logit_normal_loss(linear, 'velocity')),
    't logit-normal, m = -0.5': (linear, 'velocity', unet,
                                 logit_normal_loss(linear, 'velocity', -0.5)),
    'cosine path, v': (cosine, 'v', unet, d2l.regression_loss(cosine, 'v')),
    "t at EDM's levels": (linear, 'velocity', unet,  # ln sigma_bar = logit t
                          logit_normal_loss(linear, 'velocity', -1.2, 1.2)),
    'EDM, clean image': (linear, 'x0',
                         lambda: EDM(unet(), linear, sigma_data), edm_loss)}
grids = {'EDM grid': d2l.edm_grid(25),            # 49 network evaluations
         'linear grid': linear.sigma_bars(25)}    # 48 network evaluations
```

次のループは各レシピを訓練し、同じノイズから両方のグリッド上で全モデルをサンプリングして、各行・各グリッドについてシード間の平均 FD と括弧内に各シードの FD を出力する。実行前に 1 つ結果を予測できる。コサイン経路上の $\mathbf{v}$ 損失は別の座標系で表したベースの目的関数そのものであるから、ベースに劣ることはないはずである。

```
batch_fn = lambda: X[torch.randint(0, len(X), (128,), device=X.device)]
torch.manual_seed(1)
z = torch.randn(5000, 1, 28, 28, device=device)  # the same noise for every run
four_seeds = ('base: velocity, t uniform', "t at EDM's levels",
              'EDM, clean image')         # to split EDM's gain
models, samples = {}, {}
print(f'{"FD: mean (each seed)":27} {"EDM grid":28} linear grid')
for name, (path, kind, make_net, loss_fn) in recipes.items():
    fd = {g: [] for g in grids}
    for seed in range(4 if name in four_seeds else 2):
        torch.manual_seed(seed)
        _, ema_img, _ = d2l.train_regression(make_net().to(device), loss_fn,
                                             batch_fn, 5_000)
        pred = d2l.make_pred(ema_img, path, kind)
        for g, levels in grids.items():
            x = torch.cat([d2l.heun_sample(pred, z_k, levels)
                           for z_k in z.split(1000)])
            fd[g].append(d2l.fid_score(d2l.fd_features(cnn, x), f_test))
            if seed == 0 and g == 'EDM grid':
                models[name], samples[name] = ema_img, x[:10]
    edm, lin = (f'{sum(fd[g]) / len(fd[g]):.2f} ('
                + ' '.join(f'{v:.2f}' for v in fd[g]) + ')' for g in grids)
    print(f'{name:27} {edm:28} {lin}')
```

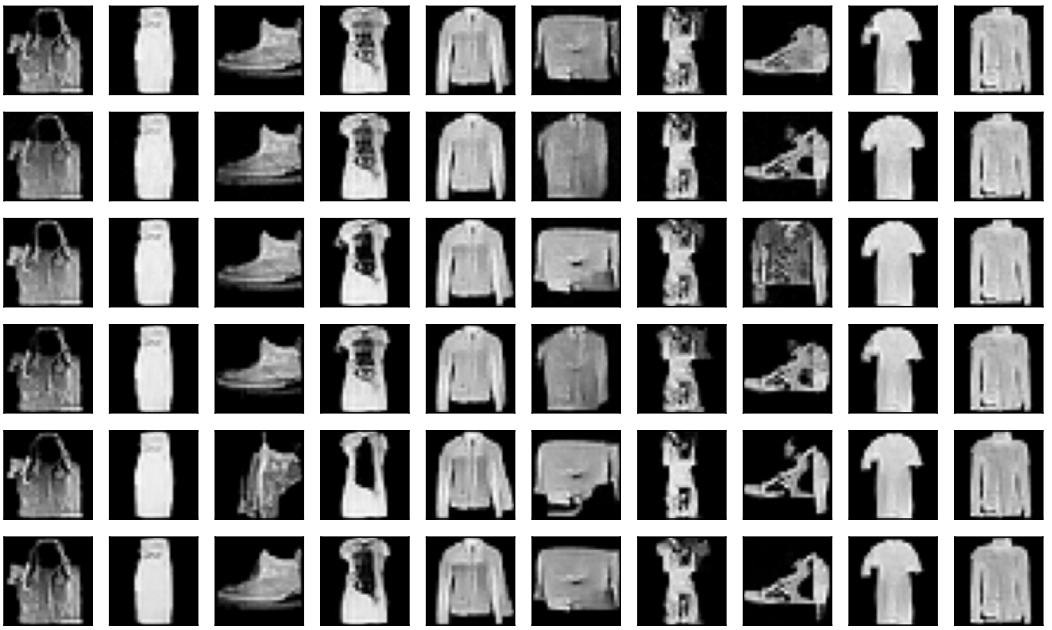
```
FD: mean (each seed)        EDM grid                    linear grid
base: velocity, t uniform   3.71 (3.46 3.73 4.32 3.34)  3.13 (2.97 3.12 3.62 2.82)
t logit-normal              4.81 (4.50 5.12)            3.88 (3.51 4.25)
t logit-normal, m = -0.5    3.29 (3.08 3.50)            2.82 (2.71 2.94)
cosine path, v              3.47 (3.56 3.38)            3.09 (3.19 2.99)
t at EDM's levels           3.19 (2.40 3.11 3.94 3.31)  2.85 (2.13 2.75 3.28 3.25)
EDM, clean image            3.22 (3.17 3.18 3.33 3.20)  3.20 (3.50 3.29 2.97 3.03)
```

5,000 回の更新（20,000 ステップという 17.7 節で潜在拡散（latent diffusion）に用いる予算の 4 分の 1）という訓練初期の段階において、この表は 2 つのサンプリンググリッドと複数のシードにわたってサンプル品質を測定している。

1. **サンプリンググリッドへの感度：**線形経路上で訓練されたすべての速度モデル（EDM のノイズレベルで訓練された対照群を含む）は、EDM グリッドより線形グリッドのほうで FD が 0.3〜1.3 低くなるのに対し、前処理を施した EDM モデルはグリッドの選択に左右されない。EDM のノイズレベルで訓練しただけでは、EDM グリッドが有利になるわけではない。
2. **訓練ノイズレベルの分布：**ロジット正規訓練時刻の中心を $m = 0$（$t = 0.5$）に置くと、EDM グリッド上で FD が 1.1〜1.3（線形グリッド上では 0.5〜0.75）だけ一様な t に比べて悪化する。ロジット正規の中心をデータ側へシフトさせるか（$m = -0.5$、$t \approx 0.38$）、EDM のノイズレベル（$m = -1.2$、$t \approx 0.23$）を用いれば、この劣化は解消する。
3. **ターゲットと前処理：**線形グリッド上では、ベースの速度レシピ、コサイン経路上の $\mathbf{v}$ 予測、および EDM がシード間で重なり合う FD の範囲を達成する。

同じ 10 枚のノイズ画像から生成した、EDM グリッド上における第 1 シードのサンプルを表の順に 1 レシピ 1 行で示す。

```
d2l.show_gray(torch.cat(list(samples.values())), len(samples), 10);
```



ほとんどの列で、6 つのレシピすべてが同じノイズから細部のみ異なる同じ衣服を生成しており、衣服の種類自体が変わるのはごく一部の列に限られる。重み付けの異なる同一の事後平均（posterior mean）から期待されるとおり、各レシピはノイズから画像へのほぼ同じ写像を学習している。

ノイズレベルごとのテスト損失を調べると、中央に寄せたロジット正規モデルの弱点がどこにあるかが特定できる。次のセルでは、同じ 2,000 枚のテスト画像とノイズを用いて、このモデルとベースモデルを個別のノイズレベルで評価する。

```
x0 = X_test[:2000]
torch.manual_seed(3)
eps = torch.randn_like(x0)
with torch.no_grad():
    for t in (0.01, 0.03, 0.1, 0.5, 0.9, 0.98):
        t = torch.full((len(x0),), t, device=device)
        x, target = linear.sample(x0, t, eps), linear.target(
            'velocity', x0, eps, t)
        loss = [((models[name](x, t) - target) ** 2).flatten(1).sum(1).mean()
                for name in ('base: velocity, t uniform', 't logit-normal')]
        print(f't = {t[0]:.2f}: loss {loss[0]:6.1f} (uniform), '
              f'{loss[1]:6.1f} (logit-normal)')
```

```
t = 0.01: loss  422.6 (uniform),  673.0 (logit-normal)
t = 0.03: loss  332.5 (uniform),  408.7 (logit-normal)
t = 0.10: loss  224.7 (uniform),  238.8 (logit-normal)
t = 0.50: loss  131.0 (uniform),  127.3 (logit-normal)
t = 0.90: loss  244.5 (uniform),  245.8 (logit-normal)
t = 0.98: loss  278.6 (uniform),  289.7 (logit-normal)
```

訓練時刻の大半が集まる $t = \frac{1}{2}$ では、ロジット正規モデルはベースをわずかに上回る（損失 127 対 131）が、$t = 0.01$ では大幅に劣る（約 670〜700 対 420〜440）。このモデルは $t = 0.03$ 未満の訓練時刻を 3,000 回に 1 回未満しか抽出しないのに対し、一様な時刻は 100 回に 3 回抽出し、どのサンプラも最終ステップではこれらのレベルを使って細部を仕上げる。Esser et al. ははるかに大きな画像の潜在表現に合わせて中心を選んだが、先ほどのスケールの議論からわかるとおり、28×28 の画像はより少ないノイズで構造が決まるため、中心をシフトした行は訓練レベルをその領域へ移しているのである。

17.6.3.4 レシピの選択

新たなモデルを設計するにあたり、これまでの分析から 3 つの設計指針が導かれる。

1. **純粋なノイズ（$\alpha_1 = 0$）に到達する経路上で、条件の良いターゲットを選ぶ：**線形経路上の速度予測、コサイン経路上の $\mathbf{v}$ 予測、および EDM の前処理は $[0, 1]$ の両端で適切なスケールを保つのに対し、素のノイズ予測は高ノイズ領域で $\alpha_t \to 0$ による除算を行って $\hat{\mathbf{x}}_0$ を再構成しなければならない。
2. **空間構造が形成される領域に訓練ノイズレベルを配置する：**28×28 の画像では、一様な t または低ノイズ側へシフトしたロジット正規分布（$m < 0$）が、$t = 0.5$ を中心とするロジット正規分布を上回る。画像解像度が r 倍に増えるときは、対数信号対雑音比を $\log r^2$ だけ下方にシフトさせる（17.6.3.2 節）。
3. **境界ではなくサンプルのために損失の重み付けを選ぶ：**重み付け $w(\lambda) \equiv 1$ は尤度境界を訓練し、低ノイズの細部に多くの労力を費やす。実際に良質な画像サンプルをもたらしてきたのは、$w(\lambda) = e^{-\lambda/2}$（フローマッチングや $\mathbf{v}$ 予測）のように高い対数信号対雑音比を割り引く重み付けである (Kingma and Gao 2023)。本節の混合分布でも、境界の重み付けは小規模ネットワークに最良のフロー尤度さえもたらさなかった。

本章のデフォルト設定である、線形経路上で一様な t を用いた速度回帰と線形グリッド上の積分は、追加のハイパーパラメータなしでこれらの基準を満たしている。続く 17.7 節では、拡散過程を学習された潜在空間（latent space）へ移すことで、高周波のピクセル細部にネットワーク評価を費やすのを避ける方法を調べる。

17.6.4 まとめ

順方向のノイズ付加連鎖を反転させることで潜在変数モデルが定まり、その負の対数尤度はELBO、すなわちステップごとの対数信号対雑音比の低下量の半分 $\frac{1}{2}(\lambda_{k-1} - \lambda_k)$ で重み付けされたノイズ予測誤差の和によって上から抑えられ、その連続時間極限は $\frac{1}{2}\int \mathbb{E}\|\epsilon - \hat{\epsilon}_\theta\|^2 \, d\lambda$ となる。より一般に、標準的な拡散モデルおよびフローマッチングの目的関数はすべて対数信号対雑音比の座標で $\mathcal{L}_w = \frac{1}{2}\int w(\lambda)\,\mathbb{E}\|\hat{\epsilon}_\theta - \epsilon\|^2 \, d\lambda$ と書け、単調減少する重み付け $w(\lambda)$ はノイズ付加データ上の尤度境界に対応する。特に、線形経路上のフローマッチングとコサイン経路上の $\mathbf{v}$ 予測は $w(\lambda) \propto e^{-\lambda/2}$ を共有する。3 モード混合分布上では、訓練済みネットワークは連続時間 ELBO と確率フロー ODE の変数変換公式の両方のもとで厳密なオラクルの負の対数尤度に 0.005 nat 以内で一致するが、表現力が制限された設定では $w(\lambda) \equiv 1$ がよりタイトな ELBO を与える一方、重みなしのノイズ回帰はより良いフロー尤度と少ない逸脱サンプルをもたらす。最後に、EDM の前処理はネットワークの入力・スキップ接続・ターゲットを全ノイズレベルにわたって単位分散にスケールし、また自然画像に対して白色ノイズはローパスフィルタとして働くため、画像解像度を r 倍にする際には対数信号対雑音比のスケジュールを $2\log r$ だけ下方にシフトさせる必要がある。

17.6.5 演習問題

1. [code] **境界の重み**。●ω_k を Equation 17.52 の重みとする。
    1. ω_k を $\nu_k = \beta_k$ と $\nu_k = \tilde{\beta}_k$ のそれぞれについて VP 連鎖のすべてのステップ $k = 2, \dots, 1000$ で計算し、両者を対数軸上にプロットせよ。それぞれの選択において、$\sum_k \omega_k$ のうちステップ $k \le 20$ が占める割合はいくらか。単純損失がそこに割り当てる割合はいくらか。
    2. 離散連鎖の境界 Equation 17.46 を、混合分布上で厳密なノイズ予測器、$\nu_k = \tilde{\beta}_k$、およびステップ 1 のノイズレベルにおける本文の再構成項を用いて評価せよ。これを真の負の対数尤度および本文の連続時間境界と比較せよ。本文の 2 ステップの例を踏まえると、過剰分の大部分はどのステップから生じると予想されるか、またそれはなぜか。
    3. 離散連鎖の境界のミニバッチ推定では k を一様に抽出する。これに対し Nichol and Dhariwal (2021) は、k を確率 p_k（サンプルごとの L_{k-1} の値の二乗平均平方根に比例）で抽出し、抽出された項を p_k で割る。なぜこれにより推定量の分散が減少するのか。このようなサンプラは混合分布上でどのステップを最も頻繁に抽出するか。
2. **暗黙の重み付け**。@eq-diffusion-weighted-loss を用いよ。
    1. 線形経路上で t を一様とした速度損失が $w(\lambda) \propto e^{-\lambda/2}$ をもち、$\mathbf{v}$ 損失もコサイン経路 $\alpha_t = \cos\frac{\pi t}{2}$、$\sigma_t = \sin\frac{\pi t}{2}$ 上で同じ重み付けをもつことを示せ。
    2. コサイン経路上で t を一様としたノイズ損失が $w(\lambda) \propto \operatorname{sech}(\lambda/2)$ をもつことを示し、さらに w をノイズ損失と $\mathbf{v}$ 損失について線形 β の VP 経路 `vp` 上で計算せよ。
    3. これらの重み付けのうち、λ に関して単調なのはどれか。VP 経路上のノイズ損失は、同じ経路上の $\mathbf{v}$ 損失に比べてどのノイズレベルの重みを小さくしており、最終ステップでは何倍小さくなるか。
    4. （コード）第 1 小問より、コサイン経路上の $\mathbf{v}$ 損失と線形経路上の速度損失は同一の目的関数であるが、本文の比較実験ではこれらによって 2 つの異なるネットワークが訓練される。比較実験の最後のセルと同様に、両モデルの出力を同じ λ におけるノイズ予測へ `make_pred` で変換したうえで、比較実験の第 1 シードの 2 モデルを個別のノイズレベルで評価せよ。どのレベルで両者に差が生じるか。2 つの経路は同じデータ点を異なるスケールでネットワークに入力するが、$t = 1/2$ において各経路の入力はどのスケールをもつか、またその違いから観測結果をどのように説明できるか。
3. [code] **高次元における尤度**。●d 次元において、`pf_nll` の厳密な発散計算は 1 回の評価につき d 回のフォワードモード導関数を必要とする。これを 38.3 節の Hutchinson トレース推

定値 $\mathbf{r}^\top(\partial \mathbf{u}/\partial \mathbf{x})\,\mathbf{r}$ に置き換え、ステップごとに 1 つのラデマッハベクトル $\mathbf{r}$ を用いるようにせよ（必要な導関数計算は 1 回になる (Grathwohl et al. 2019)）。

1. 得られる $\log p$ の推定量が不偏であることを示せ。プローブ 1 個あたりの分散は演習問題 5（38.3 節）の第 1 小問で導出されている。混合分布上で 1 点あたりの推定値の標準偏差を測定し、訓練済みネットワークと厳密な場との間にある 1000 分の数 nat という差と比較せよ。その差を識別するには、1 ステップあたり何個のプローブ、あるいは何個のテスト点が必要か。
2. 100 枚のテスト画像の各ピクセルに 1 階調分の幅 $2/255$ の一様ノイズを加えたうえで、比較実験のベースモデル（線形経路上の速度）にこの推定法を適用せよ。結果を次元あたりビット数 $\big(-\log p(\mathbf{x})/d + \log 127.5\big)/\log 2$（8 ビット画像のピクセルあたりビット数に相当）で報告せよ。256 階調上で一様な密度はちょうど 8 ビットとなる。正常に機能しているモデルはそれよりはるかに小さい数ビット以下の値となり、8 付近またはそれ以上の値は通常、$\log 127.5$ の付け忘れか符号の誤りを意味する。更新回数を 15,000 回にすると結果はどう変わるか。

4. **解像度とスケジュール**。ピクセルあたり単位分散をもつデータに対して $\mathrm{SNR}_t = \alpha_t^2/\sigma_t^2$ と書く。
   1. データを定数 κ 倍すると、$\mathbf{x}_t$ における信号分散とノイズ分散の比が κ^2 倍になる（すべての t において）ことを示せ。
   2. 各画像を $28r \times 28r$ ピクセルへ、各ピクセルを $r \times r$ ブロックに繰り返すことでアップサンプルし、その大きな画像に同じスケジュールでノイズを加える。ノイズ付き画像を各ブロック上で平均すると、比 $r^2\,\mathrm{SNR}_t$ をもつ元の画像が得られることを示せ。
   3. $r = 32$（896×896 画像）のとき、vp に対する $r^2\,\mathrm{SNR}_1$ を計算し、28×28 の経路上で同じ比をもつ時刻を求めよ。純粋なノイズからサンプラを開始する場合、このスケジュールで訓練された 896×896 モデルにとってこの結果は何を意味するか。
   4. λ_t をどのようにシフトさせれば、ブロック平均した大画像が 28×28 モデルと同じ比を通過するようになるか。これを Hoogeboom et al. (2023) のシフトと比較せよ。
   5. （コード）ピクセルを繰り返して Fashion-MNIST を 56×56 にアップサンプルし、比較実験のベースレシピを線形経路そのままの場合と λ_t を $-\log 4$ だけシフトさせた場合の 2 通りで、それぞれ 2 シードから訓練せよ。第 4 小問から予想されるとおり、シード間のばらつきを上回る改善がシフトによって得られるか。

5. **順事後分布**。@eq-diffusion-posterior を導出せよ。
   1. 次式

$$q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) \propto q(\mathbf{x}_k \mid \mathbf{x}_{k-1})\, q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)$$

   を $\mathbf{x}_{k-1}$ に関する 2 つのガウス分布の積として書き、それらの精度（逆分散）を足し合わせ、精度に比例して平均を結合せよ。
   2. $\tilde\beta_k \le \beta_k$ であり、等号成立は $\sigma_{k-1}^2 = 1$ の場合に限られることを示せ。連鎖のどの部分で $\tilde\beta_k$ は β_k よりはるかに小さくなるか。
   3. 本文の 2 ステップの例において、データを $\mathcal{N}(0, a^2)$ とする。境界を最小化する分散 ν_2 が $\mathrm{Var}(x_1 \mid x_2)$ であり、それが $\tilde\beta_2$（$a \to 0$ のとき）および β_2（$a = 1$ のとき）という Ho et al. (2020) の 2 つの選択に一致することを示せ。

6. **EDM の前処理**。●$\mathbf{x}_0$ はノイズと独立で、平均 0、座標ごとの分散 σ_{data}^2 をもつとする。
   1. c_{in} を選び、入力 $c_{\text{in}}(\mathbf{x}_0 + \bar\sigma\epsilon)$（$F$ への入力）が単位分散をもつようにせよ。F の訓練ターゲットを $(\mathbf{x}_0 - c_{\text{skip}}\,\bar{\mathbf{x}})/c_{\text{out}}$ と書いたとき、その分散を最小化するよう c_{skip} を選び、次にその分散を 1 にするよう c_{out} を選べ。結果を Equation 17.55 と照合せよ。さらに平均が 0 でないデータに対して分散を二次モーメントに置き換えて同じ導出を繰り返し、σ_{data} が二乗平均平方根になることを示せ。
   2. 損失 $\|D - \mathbf{x}_0\|^2/c_{\text{out}}^2$ が $\|F - \text{target}\|^2$ に等しく、またノイズ損失として書き直すとノイズ誤差の $(e^{-\lambda} + \sigma_{\text{data}}^2)/\sigma_{\text{data}}^2$ 倍になることを示せ。

3. $\log \bar{\sigma} \sim \mathcal{N}(-1.2, 1.2^2)$ のもとで、EDM の重み付けが $w(\lambda) \propto \mathcal{N}(\lambda; 2.4, 2.4^2)\,(e^{-\lambda}+\sigma_{\text{data}}^2)$ となることを示せ。そのピークはどこにあるか。Kingma and Gao (2023) は、ピークより小さいすべての λ で重みをピーク値に固定することでこれを単調関数にしている。本文の定理によれば、この変更によって EDM の目的関数は何になるか。

17.7 潜在拡散と Transformer

ピクセル空間で拡散モデルを直接実行すると、サンプリングの各ステップですべてのピクセルにわたってノイズ除去ネットワークを評価することになる。28×28 のグレースケール画像に含まれる値は 784 個にすぎないが、512×512 の RGB 写真には 786,432 個もの値が含まれ、しかも 17.6.3.2 節で示したように、こうした高周波次元の大部分は人間の目には知覚できない微細なテクスチャを担っている。**潜在拡散**（latent diffusion）は、知覚圧縮（perceptual compression）と意味的な生成を切り離す手法である (Rombach et al. 2022)。まず変分オートエンコーダ（variational autoencoder）が各画像を**潜在表現**（latents）の低次元空間グリッドへと圧縮し、拡散モデルはこの潜在空間においてステップあたりの FLOPs を大幅に抑えつつ訓練とサンプリングを行い、生成サンプルごとに 1 回だけデコーダを実行する。

第 2 のアーキテクチャ上の転換は、畳み込み U-Net を**拡散 Transformer**（diffusion transformer、DiT）に置き換えることである (Peebles and Xie 2023)。DiT は、適応的層正規化（adaptive layer normalization）の一種である `adaLN-Zero` によって変調されるパッチトークンの系列として、画像や潜在グリッドを処理する。本節では、KL 正則化を施した畳み込みオートエンコーダを Fashion-MNIST 上で訓練し、その $7 \times 7 \times 4$ の潜在表現上で速度フローマッチングを実行して、デコーダの歪み（distortion）と潜在表現のモデル化のしやすさ（modelability）のトレードオフを診断するとともに (Dieleman 2025)、計算規模を変えながら U-Net と DiT のバックボーンを比較する。

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.7.1 知覚圧縮

17.7.1.1 ピクセルのコスト

サンプリングの各ステップでは、画像全体に対してネットワークを評価する。畳み込みはすべてのピクセルにカーネルを適用するため、U-Net のコストはピクセル数に比例して増大する。また、アテンションはあらゆる位置の組を比較するため、そのコストは位置数の二乗で増大する。512×512 のカラー画像は $512 \cdot 512 \cdot 3 = 786{,}432$ 個の数値を保持しており、50 ステップのサンプラはその一つひとつに 50 回アクセスする。これらの数値の大半は細部を記述している。画像をフーリエ基底で表してみよう。画像あたり k サイクルまでの周波数の個数は k^2 のオーダーで増えるため、座標の大半は高周波（細かい周波数）に属する。実際、17.6.3.2 節で見たように、Fashion-MNIST 画像のパワーは画像あたり 1 サイクルから 14 サイクルにかけて 3 桁減少していた。数多くの微細な座標は、画像のパワーをほとんど担っていないのである。

何を捨てるべきかを語る語彙を与えてくれるのが、レート・歪み理論（rate–distortion theory）である。各歪みレベルに対し、レート・歪み関数は符号が必要とする画像あたりの最小ビット数を表す（37.6.3.2 節）。どのビットを削っても見る人が気づかないかは二乗誤差ではなく人間の知覚に関する問いであり、Rombach et al. (2022) は観測によってこれに答えた。自然画像の尤度モデルは、曲線の低歪み側の領域において、誰にも知覚されない細部に大半のビットを費やしているのである。そこで彼らは、この**知覚圧縮**をオートエンコーダに任せ、画像が何を表しているかをモデル化する**意味圧縮**（semantic compression）を拡散モデルに任せた。Stable Diffusion

のオートエンコーダは $512 \times 512 \times 3$ の画像を $64 \times 64 \times 4$ の潜在表現へと写像し、数値の個数を 48 分の 1 に減らす。拡散の各ステップはこの潜在表現上で動作し、デコーダはサンプルごとに 1 回だけ実行される。

良い潜在表現とはどのようなものだろうか。デコーダが画像を復元するのに十分な情報を保ちつつ、拡散モデルにとって学習しやすいものでなければならないが、この 2 つは相反する。すべてのピクセルをそのまま保存する潜在表現は完璧に再構成できるが、モデル化の難しさはピクセルそのものと変わらない。Dieleman (2025) は、このトレードオフを構成する 3 つの量に名前を与えている。**レート**は潜在表現が持つ数値の個数、**歪み**はデコーダが失う情報量、そして**モデル化のしやすさ**は潜在表現の生成モデルがどれだけうまく機能するかを表す。本節のオートエンコーダは、784 個の数値を 4 チャネルの 7×7 グリッド、すなわち 196 個の数値というレートへ写像する（Figure 17.12）。

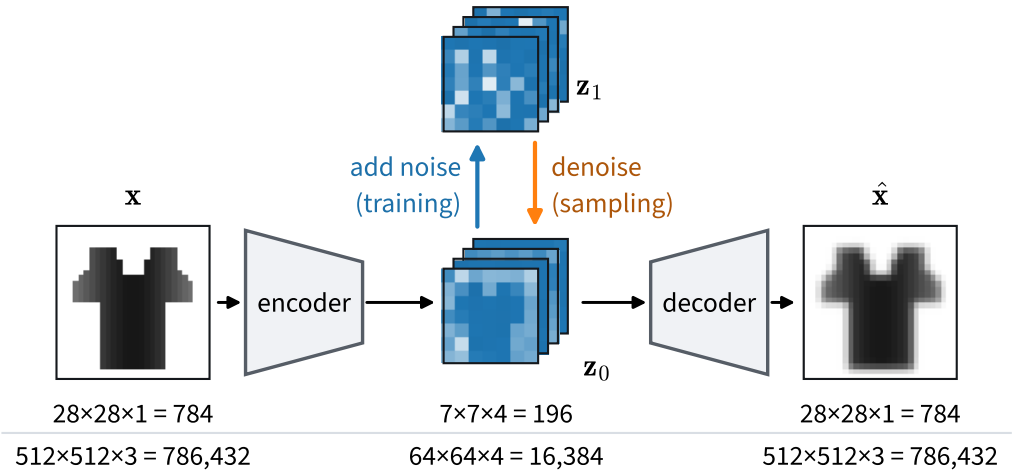


Figure 17.12: 潜在拡散。エンコーダは画像をより小さな潜在表現に写像し、拡散モデルは潜在表現上で訓練とサンプリングを行い、デコーダはサンプリングされた潜在表現を画像に戻す。上：本節のオートエンコーダであり、$28 \times 28 \times 1$（784 個の数値）から $7 \times 7 \times 4$（196 個）へ写像する。下：Stable Diffusion のオートエンコーダであり、$512 \times 512 \times 3$（786,432 個）から $64 \times 64 \times 4$（16,384 個）へ写像する。

17.7.1.2 オートエンコーダ

3 つの量のうち最も自明でないのがモデル化のしやすさであるため、拡散モデルが潜在表現に何を求めるかを整理しておこう。必要なのは、ガウス経路（Gaussian path）のノイズを較正する基準となる既知のスケール、孤立した外れ値がないこと、そしてわずかにずれた潜在表現からももっともらしい画像が得られるよう、近くの潜在表現が似た画像にデコードされることの 3 点である。再構成のみを目的として訓練されたオートエンコーダは、これらのいずれも保証しない。コードを 1,000 のスケールで保持することも、隣接する 2 つのコードを靴とシャツにデコードすることも妨げられないからである。

変分オートエンコーダは、これら 3 つすべてに対する解決策を与える (Kingma and Welling 2014)。エンコーダは平均 μ と標準偏差 $\mathbf{s}$ を 196 個の潜在値それぞれに対して出力し、潜在表現は $\mathbf{z} = \mu + \mathbf{s} \odot \xi$（ただし ξ は標準正規分布に従う）としてサンプリングされ、デコーダは $\mathbf{z}$ を再構成 $\hat{\mathbf{x}}$ に写像する。損失は、再構成誤差に対して、各潜在値を標準正規分布へ引き寄せる次の項を加える。

$$\mathcal{L}(\mathbf{x}) = \frac{1}{784}\,\|\mathbf{x}-\hat{\mathbf{x}}\|^2 + \frac{\beta}{196}\sum_{j=1}^{196}\tfrac{1}{2}\left(\mu_j^2 + s_j^2 - 1 - \log s_j^2\right). \tag{17.56}$$

この和はカルバック・ライブラー・ダイバージェンス（Kullback–Leibler divergence）$D_{\mathrm{KL}}(\mathcal{N}(\mu, \mathrm{diag}(\mathbf{s}^2))\,\|\,\mathcal{N}(\mathbf{0}, \mathbf{I}))$、すなわちエビデンス下界（evidence lower bound）に現れる項であり（36.7 節）、β はその重みである（スケジュールの β_k ではない）。先ほどの 3 つの要件に照らすと、この損失は 2 つの働きをする。各 μ_j を 0 に、各 s_j を 1 に引き寄せることで、スケールを固定し外れ値にペナルティを科す。また、訓練中に $\mathbf{z}$ を大きさ $\mathbf{s}$ のノイズ付きでサンプリングすることで、デコーダには約 $\mathbf{s}$ だけ揺らいだコードが提示されるため、それらを同様にデコードするよう学習する。これが 3 つ目の要件にあたる。

こうした作用がどの程度働くかは β に依存する。本節で選ぶ $\beta = 10^{-4}$ では、再構成が主導権を握ったままである。この重みでは、KL 項はスケールを固定し、エンコーダのノイズが 0 に崩壊するのを防ぐものの、隣接する潜在表現が同様にデコードされるようにするにはあまりに弱すぎる。それが問題になるかどうかは自明ではなく、以下で実際に測定する。

大規模システムのオートエンコーダには、さらなる工夫が加えられている。二乗誤差はもっともらしいテクスチャの候補を平均化してぼやけさせてしまうため（16.7 節）、通常は画像パッチにスコアを付けるクリティックと知覚損失（perceptual loss）を併用して訓練されてきた。ただし、近年の大規模オートエンコーダである ViTok-v2 は、スケールアップに際してクリティックを廃し、固定された知覚損失のみを用いている (Hansen-Estruch et al. 2026)。本節のモデルにはどちらも不要である。Stable Diffusion のオートエンコーダが写真の数値の 48 分の 1 しか保持しないのに対し、本節のモデルは黒い背景上の小さなグレースケール画像の 4 分の 1 の数値を保持するからである。このほか、潜在表現を離散トークンへ量子化するオートエンコーダもあり (Esser et al. 2021; Chang et al. 2022)、その場合は較正すべきスケールのない有界なコードが得られ、マスク付き拡散（masked diffusion）（17.9 節）が扱う形式の入力となる。

本節のオートエンコーダは、埋め込みを持たない U-Net の残差ブロックを幅 64 と 128 で積み重ねたものである。エンコーダは解像度を 2 回半分にして、潜在チャネルごとに平均と対数分散を持つ 7×7 グリッドへと変換し、デコーダはその鏡像の構造をとる。

```
class Autoencoder(nn.Module):
    """Residual convolutional autoencoder from 28x28x1 images to a 7x7
    Gaussian latent and back; returns the reconstruction and the KL term."""
    def __init__(self, latent_channels=4, ch=64, num_blocks=2):
        super().__init__()
        blocks = lambda a, b: [d2l.UNetBlock(a, b)] + [
            d2l.UNetBlock(b, b) for _ in range(num_blocks - 1)]
        down = lambda c: nn.Conv2d(c, c, 3, stride=2, padding=1)
        head = lambda a, b: [nn.GroupNorm(8, a), nn.SiLU(),
                             nn.Conv2d(a, b, 3, padding=1)]
        self.encoder = nn.Sequential(
            nn.Conv2d(1, ch, 3, padding=1), *blocks(ch, ch),
            down(ch), *blocks(ch, 2 * ch),                      # 14 x 14
            down(2 * ch), *blocks(2 * ch, 2 * ch),              # 7 x 7
            *head(2 * ch, 2 * latent_channels))  # mean and log variance
        self.decoder = nn.Sequential(
            nn.Conv2d(latent_channels, 2 * ch, 3, padding=1),
            *blocks(2 * ch, 2 * ch),
            nn.Upsample(scale_factor=2), *blocks(2 * ch, ch),    # 14 x 14
```

```
            nn.Upsample(scale_factor=2), *blocks(ch, ch),        # 28 x 28
            *head(ch, 1))

    def encode(self, x):
        mean, log_var = self.encoder(x).chunk(2, dim=1)
        return mean, log_var.clamp(-30, 20)

    def forward(self, x):
        mean, log_var = self.encode(x)
        z = mean + (0.5 * log_var).exp() * torch.randn_like(mean)
        kl = 0.5 * (mean ** 2 + log_var.exp() - 1 - log_var).mean()
        return self.decoder(z), kl
```

d2l.train_regression（17.1 節）を用いて、20,000 回の更新にわたりこのモデルを訓練する。この関数は損失に対して Adam を実行し、訓練済みモデル、重みの移動平均、および損失の履歴を返す。ただし、ここではコサイン曲線に沿って学習率が 0 まで減衰するため、最終的な重みの平均化は不要である。実行前にコストについて一言述べておくと、本節では 2 つのオートエンコーダと 14 個の拡散モデルをそれぞれ 20,000 回更新するため、1 基の RTX 4090 上でノートブック全体の実行に約 1 時間半かかり、CPU 上ではそれよりはるかに長い時間を要する。

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()
dev = X.device

def batch_fn():
    return X[torch.randint(0, len(X), (128,), device=dev)]

def ae_loss(ae, x, beta=1e-4):
    x_hat, kl = ae(x)
    return ((x_hat - x) ** 2).mean() + beta * kl

torch.manual_seed(1)
ae, _, _ = d2l.train_regression(d2l.Autoencoder().to(dev), ae_loss, batch_fn,
                                20_000, cosine=True)
print(f'{d2l.num_params(ae) / 1e6:.2f}M parameters')
```

```
2.40M parameters
```

1 行目は 8 枚のテスト画像、2 行目はそれらの潜在平均に対するデコーダの出力である。

```
with torch.no_grad():
    x_hat = ae.decoder(ae.encode(X_test[:8])[0])
d2l.show_gray(torch.cat([X_test[:8], x_hat]), 2, 8);
```

再構成画像は元の画像と見分けがつきにくい。これを定量化するため、17.4 節で導入したフレシェ距離（Fréchet distance、FD）を用いる。これは小規模な分類器 `d2l.FeatureCNN` が 2 組の画像集合から抽出した特徴にそれぞれガウス分布を当てはめ、その間の距離を測る指標である。実画像の 2 つの集合であっても偶然による差が生じるため、互いに素な 5,000 枚ずつのテスト画像 2 組の間の FD を**フロア**（下限の目安）として表示する。本節の FD はすべて 5,000 枚の画像を用いて計算する。1,000 枚ではフロアが約 5 倍高くなるからである（17.4.3.2 節）。

```
cnn = d2l.fashion_feature_cnn(X, y)
f_ref = d2l.fd_features(cnn, X_test[:5000])

def fd(imgs):
    """FD of images against the first 5,000 test images."""
    return d2l.fid_score(d2l.fd_features(cnn, imgs), f_ref)

print(f'floor, the other 5,000 test images: FD {fd(X_test[5000:]):.2f}')
```

```
floor, the other 5,000 test images: FD 0.17
```

`fd` を用い、残りの 5,000 枚のテスト画像 `X_test[5000:]` をバッチ単位でエンコードおよびデコードして、ピクセルあたりの二乗再構成誤差と再構成のフレシェ距離 `fd_rec` の両方を測定しよう。

```
x_hat = d2l.batched(lambda x: ae.decoder(ae.encode(x)[0]), X_test[5000:])
fd_rec = fd(x_hat)
print(f'reconstructions: squared error {((x_hat - X_test[5000:]) ** 2).mean():.4f}'
      f' per pixel, FD {fd_rec:.2f}')
```

```
reconstructions: squared error 0.0038 per pixel, FD 0.69
```

再構成画像の FD スコアは、フロアの 0.17 に対して 0.6〜0.8 であり、ピクセルあたりの二乗誤差は約 0.004 である。これがオートエンコーダの歪みであり、同時に理想的な潜在モデルのスコアでもある。もし拡散モデルがエンコーダの生み出す分布どおりに潜在表現をサンプリングできれば、エンコーダの小さなノイズを除いて、デコーダはそれらを再構成画像へと変換するはずだからである。以下のすべての潜在モデルは、この数値を基準として評価する。ただし、後の摂動テストで示す理由により、これは厳密な下界ではなく参照値である。

訓練セットを 1 回エンコードし、潜在表現の標準偏差、エンコーダの $\mathbf{s}$ の平均、および全潜在値の超過尖度（excess kurtosis）を表示する。超過尖度は正規分布では 0 になり、少数の値が大きく外れた位置にあると大きな値をとる。

```
encoded = d2l.batched(lambda x: torch.cat(ae.encode(x), 1), X)
mu, log_var = encoded.chunk(2, dim=1)       # mean and log variance
```

```
s = (0.5 * log_var).exp()
z = mu + s * torch.randn_like(mu)
scale = 1 / z.std().item()
kurtosis = (((z - z.mean()) * scale) ** 4).mean() - 3
print(f'latent std {z.std():.2f} (scale factor {scale:.2f}), encoder std '
      f'{s.mean():.3f}, excess kurtosis {kurtosis:.2f}')
```

```
latent std 0.94 (scale factor 1.06), encoder std 0.026, excess kurtosis -0.20
```

潜在表現の標準偏差は 1 をわずかに下回っているため、約 1.05 の係数 c を掛けて経路のノイズと同じ単位分散に揃える。また、超過尖度は 0 付近または負であり、裾の重い分布にはなっていない。これで 3 つの要件のうち 2 つが満たされた。一方、エンコーダの標準偏差は 0.02〜0.03 であるため、訓練中にデコーダが目にした各コードのぼかしは潜在表現の広がりの約 3% にとどまり、10 分の 1 を超えることはほとんどなかった。わずかにずれた潜在表現をもっともらしい画像にデコードするという 3 つ目の要件は、この弱い KL 項では満たされないのである。

17.7.2 潜在空間での拡散

17.7.2.1 潜在表現に対する同じ回帰

オートエンコーダを固定すれば、データ以外は何も変わらない。前節までに構築した仕組みがすべてそのまま引き継げることこそ、この方式の魅力である。各バッチでは、128 枚の訓練画像の潜在表現 $\mathbf{z}_0 = c\,(\mu + \mathbf{s} \odot \xi)$ を、Rombach et al. (2022) と同様に毎回新たにエンコーダの分布からサンプリングする。訓練は 17.2 節の線形経路上で行い、$\mathbf{z}_t = (1-t)\,\mathbf{z}_0 + t\,\epsilon$ を構成して、ネットワークの出力を速度 $\epsilon - \mathbf{z}_0$ へと回帰させる。

スケール係数が重要になるのは、この経路がデータを単位分散のノイズと混合するためである。標準偏差が 5 の潜在表現をそのまま使うと、あらゆる時刻で対数信号対雑音比（log-SNR）が $\log 25$ だけシフトしてしまう（演習問題 1）。Stable Diffusion が潜在表現に 0.18215 を掛けているのも同じ理由による。ネットワークには 4 つの入力チャネルと出力チャネルを持つ 17.4 節の U-Net を用い、画像と潜在表現の両方で同一の `loss_fn` を共用する。

```
path = d2l.GaussianPath.linear()
loss_fn = d2l.regression_loss(path, 'velocity')  # for images or latents

def latent_batch_fn():
    i = torch.randint(0, len(mu), (128,), device=dev)
    return (mu[i] + s[i] * torch.randn_like(mu[i])) * scale

torch.manual_seed(3)
_, unet_latent, _ = d2l.train_regression(
    d2l.UNet(ch=64, in_channels=4, out_channels=4).to(dev), loss_fn,
    latent_batch_fn, 20_000)
```

比較基準として、同じ U-Net をピクセル上でも訓練する。どちらの実行も更新回数は 20,000 回と、17.5 節の予算の 2 倍に設定する。後述する Transformer にはより長い訓練が必要であり、このピクセルモデルはその比較基準も兼ねるからである。

```
torch.manual_seed(4)
_, unet_pixel, _ = d2l.train_regression(d2l.UNet(ch=64).to(dev), loss_fn,
                                        batch_fn, 20_000)
```

sample は固定されたノイズから d2l.ode_sample（17.3 節）のオイラー法 50 ステップで 5,000 個のサンプルを生成し、decode はスケール係数を元に戻して潜在表現をデコードする。下の図の 1 行目はピクセルモデルのサンプル、2 行目は潜在モデルのデコード済みサンプルである。

```
def sample(net, shape, n=5000):
    """n samples by 50 Euler steps from fixed noise, 1,000 at a time."""
    gen = torch.Generator(dev).manual_seed(0)
    noise = torch.randn(n, *shape, device=dev, generator=gen)
    pred = d2l.make_pred(net, path, 'velocity')
    return d2l.batched(lambda x: d2l.ode_sample(pred, x, path.sigma_bars(50)),
                       noise)

def decode(z):
    return d2l.batched(lambda z_k: ae.decoder(z_k / scale), z)

latent_shape, image_shape = (4, 7, 7), (1, 28, 28)
x_pixel = sample(unet_pixel, image_shape)
x_latent = decode(sample(unet_latent, latent_shape))
fd_pixel, fd_latent = fd(x_pixel), fd(x_latent)
print(f'FD: pixel U-Net {fd_pixel:.2f}, latent U-Net {fd_latent:.2f}')
d2l.show_gray(torch.cat([x_pixel[:8], x_latent[:8]]), 2, 8);
```

```
FD: pixel U-Net 1.54, latent U-Net 2.07
```

どちらの行にも、それとわかる衣類が並んでいる。FD スコアは潜在モデルが約 2（2.0〜2.2）、ピクセルモデルが約 1.5 である。本節のすべてのモデルと同様、これらはどちらも無条件モデルであるため、17.4 節のクラス条件付きモデルのスコアである 0.8 とは直接比較できない。

では、潜在モデルによって何が節約されたのだろうか。train_step_gflops は、128 個の入力に対する順伝播と逆伝播からなる 1 訓練ステップの浮動小数点演算数を、10 億単位（GFLOP）で計測する。

```
def train_step_gflops(net, loss_fn, shape, batch_size=128):
    """GFLOP of one training step, forward and backward, on a batch of zeros
    of the given shape."""
    # A local import, so that d2l needs nothing beyond its header's imports
    from torch.utils.flop_counter import FlopCounterMode
    x = torch.zeros(batch_size, *shape, device=next(net.parameters()).device)
    with FlopCounterMode(display=False) as counter:
        loss_fn(net, x).backward()
    return counter.get_total_flops() / 1e9
```

`train_step_gflops` を 4 チャネルの潜在 U-Net（`latent_shape`）と 1 チャネルのピクセル U-Net（`image_shape`）で評価し、両者のステップあたりの演算コストを直接比較しよう。

```
gflops_latent = d2l.train_step_gflops(
    d2l.UNet(ch=64, in_channels=4, out_channels=4).to(dev), loss_fn,
    latent_shape)
gflops_pixel = d2l.train_step_gflops(d2l.UNet(ch=64).to(dev), loss_fn,
                                     image_shape)
print(f'GFLOP per training step: latent U-Net {gflops_latent:.0f}, '
      f'pixel U-Net {gflops_pixel:.0f}')
```

```
GFLOP per training step: latent U-Net 15, pixel U-Net 203
```

潜在 U-Net がステップあたりに必要とする演算量は約 12 GFLOP であるのに対し、ピクセル U-Net が要する GFLOP 数はおよそ次の値に達する。200. 訓練時および各サンプリングステップのいずれにおいても、潜在ネットワークの評価 1 回あたりの演算量は 10 分の 1 未満で済む。

17.7.2.2 デコーダの耐性とモデル誤差

このように、潜在モデルは計算コストがはるかに低い一方で、品質はやや劣る。その FD はピクセルモデルの約 1.4 倍、再構成画像の約 3 倍である。Dieleman の用語で言えば、196 個の数値というレートには 0.6〜0.8 の歪みが伴い、そこから生成サンプルまでの隔たりがモデル化のしやすさを測る尺度となる。この差はどこから生じるのだろうか。考えられる原因は 2 つある。1 つは、潜在モデルがエンコーダの出力とは異なる潜在表現をサンプリングしてしまうこと、すなわち潜在モデル自身の誤差である。もう 1 つは、潜在モデルがわずかにずれただけの潜在表現をサンプリングしているのに、広がりの約 3% しか揺らがないコードで訓練されたデコーダがそれを質の低い画像に変えてしまうことである。KL 項が弱いことを踏まえると、2 つ目の原因はいかにもありそうに思える。

この 2 つを切り分けるため、誤ったコードに対する耐性を持つデコーダを構築し、実際に耐性があることを確認したうえで、その潜在表現上で潜在モデルを訓練してみよう。耐性の低さが差の原因であれば、生成サンプルは再構成画像に近づくはずである。候補とするのは、10 倍の KL 重みで訓練した 2 つ目のオートエンコーダである。耐性の確認では、残りの 5,000 枚のテスト画像の潜在平均に対し、潜在表現の標準偏差の σ 倍にあたるガウスノイズを加えることで、意図的にずらした潜在表現をデコードする。先に結果を予測しておこう。2 つ目のエンコーダはコードをより強くぼかすため、そのデコーダはより大きな誤差に耐える一方、再構成の品質は悪化するはずである。

```
torch.manual_seed(9)
ae_10, _, _ = d2l.train_regression(
    d2l.Autoencoder().to(dev), lambda ae, x: ae_loss(ae, x, beta=1e-3),
    batch_fn, 20_000, cosine=True)
```

`perturbed_decodes` は、潜在表現の標準偏差を単位とするエンコーダの標準偏差、再構成誤差と FD、そして各 σ に対するデコード画像の平均二乗変化量と FD を表示する。

```
def perturbed_decodes(ae, x, sigmas=(0.1, 0.3, 0.5)):
    """Decode the latent means of x, and the means plus Gaussian noise of
    sigma times their standard deviation."""
    mu, log_var = d2l.batched(lambda x: torch.cat(ae.encode(x), 1),
```

```
                                x).chunk(2, dim=1)
        x_rec = d2l.batched(ae.decoder, mu)
        print(f'  encoder std {(0.5 * log_var).exp().mean() / mu.std():.3f}, '
              f'reconstructions: error {((x_rec - x) ** 2).mean():.4f}, '
              f'FD {fd(x_rec):.2f}')
        gen = torch.Generator(dev).manual_seed(0)
        eps = torch.randn(mu.shape, device=dev, generator=gen)
        for sigma in sigmas:
            x_noisy = d2l.batched(ae.decoder, mu + sigma * mu.std() * eps)
            print(f'  sigma {sigma}: change {((x_noisy - x_rec) ** 2).mean():.4f} '
                  f'per pixel, FD {fd(x_noisy):.2f}')

    for beta, model in ((1e-4, ae), (1e-3, ae_10)):
        print(f'beta {beta:g}')
        perturbed_decodes(model, X_test[5000:])
```

```
beta 0.0001
  encoder std 0.028, reconstructions: error 0.0038, FD 0.69
  sigma 0.1: change 0.0023 per pixel, FD 0.55
  sigma 0.3: change 0.0205 per pixel, FD 0.71
  sigma 0.5: change 0.0545 per pixel, FD 3.46
beta 0.001
  encoder std 0.194, reconstructions: error 0.0041, FD 0.96
  sigma 0.1: change 0.0027 per pixel, FD 0.87
  sigma 0.3: change 0.0232 per pixel, FD 0.54
  sigma 0.5: change 0.0619 per pixel, FD 1.33
```

2 つの予測はどちらも的中している。重みを 10 倍にしたことで、エンコーダは各コードを潜在表現の広がりの約 5 分の 1（以前の 7 倍）だけぼかすようになり、再構成のスコアは約 0.3 悪化した。広がりの半分にあたるノイズを加えると、弱い重みのデコーダでは FD が 3 を超えて跳ね上がるのに対し、耐性の高いデコーダでは 1.3 付近にとどまる。それでいて、デコード画像の変化量はピクセルあたり 0.053〜0.055 対 0.061〜0.062 と、両者でほぼ変わらない。強い KL 項は、誤ったコードを無視するようデコーダに教えたのではない。誤ったコードを、元とは異なるがもっともらしい別の衣類へとデコードするよう教えたのであり、フレシェ距離が高く評価するのはまさにその点である。

出力結果には小さな驚きも含まれている。どちらのオートエンコーダでも、わずかなノイズを加えると、デコード画像は元の画像から遠ざかるにもかかわらず、FD が再構成画像の値を下回るのである（両者とも $\sigma = 0.1$ で低下し、耐性の高い方では $\sigma = 0.3$ でさらに低下する）。考えられる理由は、各コードの平均をデコードすると訓練中にデコーダが目にしたコード群の平均のような少し滑らかすぎる画像になり、そこにわずかなノイズが加わることで実画像の多様性が一部回復するからだというものである（本節のコードセルではこの仮説までは検証しない）。いずれにせよ、出力された数値から重要な結論が確定する。再構成 FD はあくまで参照値であり、いかなる潜在モデルも超えられない下界ではないということである。

では、耐性の高いデコーダを使えば、より良いサンプルが得られるだろうか。`latent_unet_fd` は、与えられたオートエンコーダの潜在表現に対して先ほどと同じ手順（潜在 U-Net を 20,000 回更新して訓練し、オイラー法 50 ステップでサンプリングする）を実行し、デコードされたサンプルの FD を返す。オートエンコーダごとに 2 つのシードから 2 モデルずつ訓練する。

```
def latent_unet_fd(ae, seed):
    """FD of a latent U-Net trained on the latents of ae as above."""
    mu, log_var = d2l.batched(lambda x: torch.cat(ae.encode(x), 1),
                              X).chunk(2, dim=1)
    s = (0.5 * log_var).exp()
    c = 1 / (mu + s * torch.randn_like(mu)).std()   # the scale factor

    def latent_batch():
        i = torch.randint(0, len(mu), (128,), device=dev)
        return (mu[i] + s[i] * torch.randn_like(mu[i])) * c

    torch.manual_seed(seed)
    _, net, _ = d2l.train_regression(
        d2l.UNet(ch=64, in_channels=4, out_channels=4).to(dev), loss_fn,
        latent_batch, 20_000)
    z = sample(net, latent_shape)
    return fd(d2l.batched(lambda z_k: ae.decoder(z_k / c), z))

for beta, model in ((1e-4, ae), (1e-3, ae_10)):
    fds = [latent_unet_fd(model, seed) for seed in (10, 11)]
    print(f'beta {beta:g}: latent U-Net FD {fds[0]:.2f} and {fds[1]:.2f}')
```

```
beta 0.0001: latent U-Net FD 2.04 and 2.02
beta 0.001: latent U-Net FD 2.78 and 2.68
```

結果は否である。10 倍の重みを用いた潜在 U-Net はいずれも、弱い重みを用いたどのモデルよりも約 0.5 ポイント（0.4〜0.8）悪いスコアとなった。これはどちらの設定における 2 シード間のばらつきと比べても少なくとも 2.5 倍以上の差であり、しかも 10 倍の重みのオートエンコーダによるサンプルは、それ自身の再構成 FD からもさらに大きく離れている。デコーダの耐性の低さが原因であったなら、耐性の高いデコーダによってこの差は縮まっていたはずだが、どの実行でもそうはならなかった。摂動テストは 2 つ目の手がかりも与えてくれる。潜在表現の広がりの半分にあたるノイズを加えても耐性の高いデコーダの FD は約 1.3 にしかならないのに、その潜在表現上で訓練した潜在モデルのスコアは 2.6〜2.8 に達している。つまり、これらのモデルが犯している誤りが何であれ、それは正しい潜在表現に乗った小さなランダム誤差ではなく、それより大きな誤差か、あるいは別種の誤差なのである。本節の規模において、この差の正体は潜在モデル自身の誤差である。

だからといって、オートエンコーダに責任がないわけではない。同じ手順で訓練した同じ U-Net が、一方のエンコーダの潜在表現では他方よりも明確に悪い結果を出した以上、潜在モデルの仕事の難しさを決めているのはエンコーダである。これこそが Dieleman のいうモデル化のしやすさであり、訓練した拡散モデルによってのみ測定できるオートエンコーダの性質にほかならない。したがって KL 重みは実質的な設計上の選択であり、ここでは小さい値を選ぶのが正しい。強い KL 項は再構成と生成サンプルの両方を悪化させたからである。演習問題 3 では、この重みをさらに大小両方向へ振って調べる。

では、モデル化しやすい潜在表現とは何だろうか。2024 年以降の研究は、KL 重みよりも、潜在表現が何を表しているかに注目している。REPA (S. Yu et al. 2025) は、拡散モデルの隠れ状態を DINOv2 のような事前学習済み画像エンコーダの特徴と整合させる損失を加えることで、所定のサンプル品質に数倍速く到達する。表現オートエンコーダ（representation autoencoder）(Zheng et al. 2026) はさらに踏み込み、KL 正則化エンコーダ自体を廃して、凍結した事前学習済みエンコーダから潜在表現を供給し、それを逆変換するデコーダのみを訓練する。このような潜在表現は圧縮型のものよりはるかに多くのチャネルを持つため、先ほどの議論からすれば不利に

なるはずだが、実際には拡散モデルはこれをより容易に学習する。こうした潜在表現では、近くの点が似たピクセルではなく似た物体を表す。単に小さく滑らかであることよりも、意味によって整理された潜在表現であることの方が重要であるらしい。

28×28 の解像度では、このトレードオフに魅力はない。ピクセルモデルはもともと安価であるうえ、オートエンコーダには独自の訓練が必要になり、しかも潜在モデルのサンプルはスコアで劣るからである。

17.7.3 Transformer バックボーン

U-Net には、近くのピクセルが最も重要であり、同じ特徴が画像内のどこにでも現れうるという畳み込みネットワークの仮定が組み込まれている。11.5 節のビジョン Transformer はそうした仮定をより少なく置いており、画像分類では少量のデータで訓練すると畳み込みネットワークに後れを取るが、大量のデータで事前学習するとそれを追い抜く。Peebles and Xie (2023) は、拡散においても同じことが成り立つかを問うた。

17.7.3.1 パッチとトークン

DiT はビジョン Transformer と同じ方法で入力を読み取る。画像を $p \times p$ のパッチに切り分け、それぞれを幅 d のベクトルへと線形変換（カーネルサイズとストライドが p の畳み込み）で写像し、学習される位置埋め込み（position embedding）を加えて、Pre-Norm の Transformer ブロック（11.1 節）にトークンを通す。$p = 4$ のとき、28×28 の画像はそれぞれ 16 ピクセルを持つ 49 個のトークンになる。一方、本節の潜在表現に $p = 1$ を適用すると、49 個のグリッド位置のそれぞれが 4 個の数値からなるトークンとなる。

出力は分類器とは異なる。ノイズ除去器は入力と同じ形状の予測を返すため、DiT は最終の線形層によって各トークンをそのパッチの値へと写像し、パッチを組み直して元の形に戻す。各トークンはアテンションを通じて他のすべてのトークンから情報を受け取りながら、自身のパッチの速度を予測する。ピクセルごとに 1 トークンを割り当てるとトークン数は 16 倍になり、アテンションのコストは 256 倍に膨れ上がる。

17.7.3.2 適応的層正規化

Transformer には時刻 t、さらに場合によってはラベルも入力する必要がある。17.4 節の U-Net では、埋め込まれた時刻とラベルの学習された射影を各ブロックの特徴マップにチャネルごとのシフトとして加算することでこれを解決していた。適応的層正規化はこのシフトを残したうえで、スケールとゲートを追加する。

通常の層正規化は各トークンを標準化し、学習されたスケールとシフトを適用する。**適応的層正規化**では、スケール $\mathbf{a}$ とシフト $\mathbf{b}$ を条件 $\mathbf{c}$（時刻埋め込みとラベル埋め込みの和）から計算し、さらにこの条件によって各サブ層の出力に対する**ゲート** $\mathbf{g}$ も設定する。1 つのブロックは次の計算を行う。

$$\mathbf{X} \leftarrow \mathbf{X} + \mathbf{g}_1 \odot \mathrm{Attention}\big(\mathrm{LN}(\mathbf{X}) \odot (1 + \mathbf{a}_1) + \mathbf{b}_1\big),$$
$$\mathbf{X} \leftarrow \mathbf{X} + \mathbf{g}_2 \odot \mathrm{MLP}\big(\mathrm{LN}(\mathbf{X}) \odot (1 + \mathbf{a}_2) + \mathbf{b}_2\big),$$

ここで、LN 自体は固有のスケールやシフトを持たず、6 つのベクトルは $\mathrm{SiLU}(\mathbf{c})$ に適用される 1 つの線形層から得られ、1 枚の画像内の全トークンで共有される。したがって、画像あたり 1 回の小さな行列積によって、各サブ層の各特徴量がノイズレベルごとに異なる強さで増減される。

この線形層はゼロで初期化されるため、**adaLN-Zero** と呼ばれる。初期状態ではすべてのゲートが 0 になるため、各ブロックは恒等写像を計算し、同じくゼロで初期化された最終層は 0 を出力

する。訓練はいたるところで速度 0 を予測するネットワークから始まり、ゲートが大きくなるにつれて各ブロックが働き始める（Figure 17.13）。このブロックのあらゆる要素のなかで最も記憶に留めておくべき細部は、ゼロから始まるゲートである。Peebles and Xie (2023) は、この設計を、条件を追加トークンとして末尾に加える方法、条件に対するクロスアテンションを用いる方法、およびゲートなしの適応的層正規化と比較し、4 つのなかで adaLN-Zero が最良のサンプルを与えることを示した。演習問題 4 では、ゼロからの開始が最初の 2,000 回の更新にどのような影響を与えるかを確認する。

時刻やラベルは画像あたり 1 つのベクトルであるため、適応的層正規化に適している。一方、キャプションは凍結されたテキストエンコーダが出力するトークンごとのベクトルの系列であり、これにはクロスアテンションが自然な選択肢となる。**クロスアテンション**では、m 個の画像トークンがクエリを供給し、n 個のキャプショントークンがキーと値を供給するため、出力は画像トークンごとに 1 行、すなわち画像と同じ形状になる (Rombach et al. 2022)（これに対し、Esser et al. (2024) のマルチモーダル DiT は画像トークンとキャプショントークンを 1 つの系列にまとめる）。訓練中にキャプションをランダムに脱落させれば、空のキャプションが分類器なしガイダンス（classifier-free guidance）（17.4.4.2 節）におけるヌルラベルの役割を果たす。

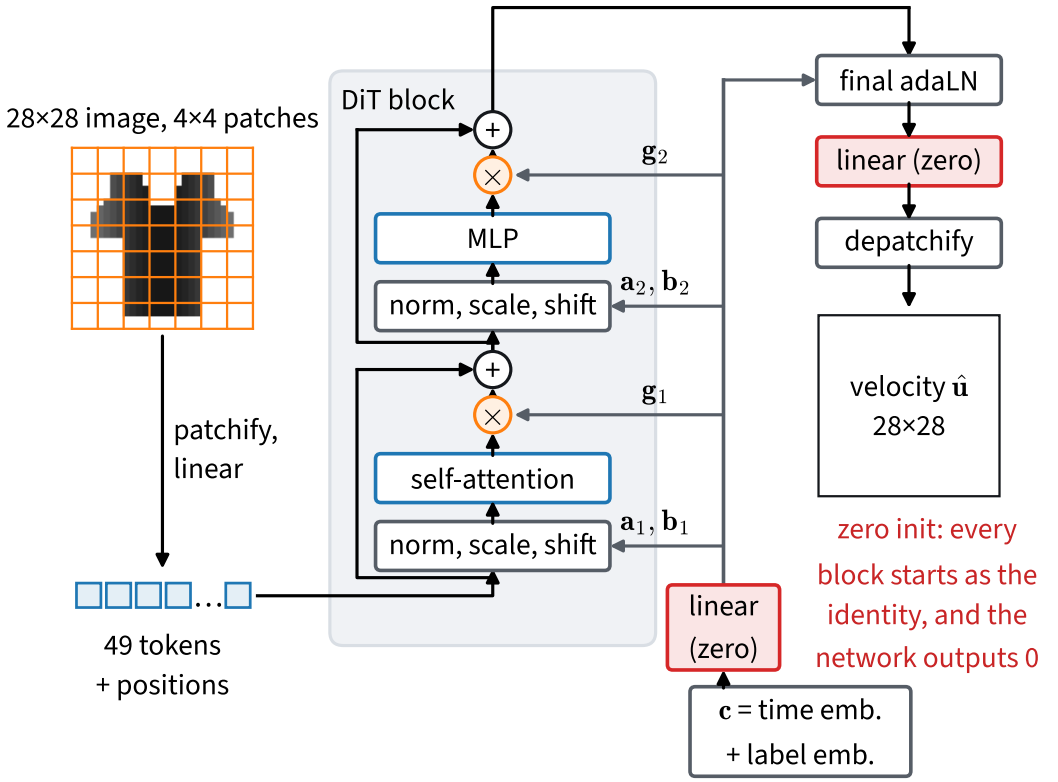


Figure 17.13: 拡散 Transformer。左：4×4 のパッチが、学習される位置を持つ 49 個のトークンになる。中央：DiT ブロック。時刻埋め込みとラベル埋め込みの和である条件 $\mathbf{c}$ が各サブ層のシフト、スケール、ゲートを設定し、ゲートはゼロから始まる。右：最終の適応的層正規化とゼロ初期化された線形層が、各トークンを対応するパッチへ写像する。

`DiTBlock` は 11.5 節の `ViTBlock` を基にこのブロックを構築し、線形変換として `ada` を持つ。

```
class DiTBlock(d2l.ViTBlock):
    """ViTBlock with adaLN-Zero: the condition c sets a shift, a scale and a
    gate for each sublayer; the gates start at zero."""
    def __init__(self, num_hiddens, mlp_num_hiddens, num_heads):
        super().__init__(num_hiddens, mlp_num_hiddens, num_heads, dropout=0.0,
                         use_bias=True)
        ln = lambda: nn.LayerNorm(num_hiddens, elementwise_affine=False,
                                  eps=1e-6)
        self.norm1, self.norm2 = ln(), ln()
        self.ada = nn.Sequential(nn.SiLU(),
                                 nn.Linear(num_hiddens, 6 * num_hiddens))
        nn.init.zeros_(self.ada[1].weight), nn.init.zeros_(self.ada[1].bias)

    def forward(self, X, c):
        shift1, scale1, gate1, shift2, scale2, gate2 = (
            self.ada(c)[:, None].chunk(6, dim=-1))
        X_mod = self.norm1(X) * (1 + scale1) + shift1
        X = X + gate1 * self.attention(X_mod, X_mod, X_mod)
        return X + gate2 * self.ffn(self.norm2(X) * (1 + scale2) + shift2)
```

DiT はパッチ埋め込み（patch embedding）、学習される位置、時刻埋め込みとラベル埋め込み、および最終の適応的層正規化と線形層を追加する。

```
class DiT(nn.Module):
    """Diffusion transformer conditioned on a time t in [0, 1] and, with
    num_classes, on a label y; label num_classes means no label."""
    def __init__(self, img_size=28, patch_size=4, num_channels=1, width=256,
                 depth=6, num_heads=4, num_classes=None):
        super().__init__()
        self.width, self.num_classes = width, num_classes
        self.p, self.g = patch_size, img_size // patch_size
        self.patch = d2l.PatchEmbedding(img_size, patch_size, width,
                                        num_channels)
        self.pos = nn.Parameter(0.02 * torch.randn(1, self.g ** 2, width))
        self.time_mlp = d2l.make_mlp([width] * 3, nn.SiLU())
        self.label_emb = (nn.Embedding(num_classes + 1, width)
                          if num_classes else None)
        self.blks = nn.ModuleList([DiTBlock(width, 4 * width, num_heads)
                                   for _ in range(depth)])
        self.ln = nn.LayerNorm(width, elementwise_affine=False, eps=1e-6)
        self.ada = nn.Sequential(nn.SiLU(), nn.Linear(width, 2 * width))
        self.out = nn.Linear(width, patch_size ** 2 * num_channels)
        for m in (self.ada[1], self.out):
            nn.init.zeros_(m.weight), nn.init.zeros_(m.bias)

    def forward(self, x, t, y=None):
        c = self.time_mlp(d2l.time_embedding(t, self.width))
        if y is not None:
            c = c + self.label_emb(y)
        X = self.patch(x) + self.pos
        for blk in self.blks:
            X = blk(X, c)
```

```
        shift, scale = self.ada(c)[:, None].chunk(2, dim=-1)
        X = self.out(self.ln(X) * (1 + scale) + shift)  # (n, g*g, p*p*c)
        n, g, p = len(x), self.g, self.p  # tokens back to pixels
        X = X.reshape(n, g, g, p, p, -1).permute(0, 5, 1, 3, 2, 4)
        return X.reshape(n, -1, g * p, g * p)
```

幅 256、深さ 6、ヘッド数 4 の DiT を構築し、8 枚の画像と時刻を入力してみよう。

```
dit = d2l.DiT().to(dev)       # width 256, depth 6, 4 heads, 4 x 4 patches
x, t = X[:8], torch.rand(8, device=dev)
with torch.no_grad():
    tokens, out = dit.patch(x), dit(x, t)
print('images', tuple(x.shape), '-> tokens', tuple(tokens.shape),
      '-> output', tuple(out.shape))
print(f'parameters: DiT {d2l.num_params(dit) / 1e6:.2f}M, '
      f'U-Net {d2l.num_params(d2l.UNet(ch=64)) / 1e6:.2f}M')
print('output at initialization all zero:', bool((out == 0).all()))
```

```
images (8, 1, 28, 28) -> tokens (8, 49, 256) -> output (8, 1, 28, 28)
parameters: DiT 7.39M, U-Net 1.20M
output at initialization all zero: True
```

予告どおり、未訓練の出力は厳密に 0 である。この DiT は U-Net の 6 倍にあたる 739 万個のパラメータを持つが、以下で行うスケーリング実験では、ステップあたりの演算量は両者でほぼ同等であることがわかる。違いは重みの使われ方にある。畳み込みは少数の重みを多数の位置すべてに適用するのに対し、DiT は多数の重みをわずか 49 個のトークンに適用するのである。

17.7.3.3 計算量に対するスケーリング

Transformer は U-Net より優れているのだろうか。どちらの種類であっても大きいネットワークは小さいネットワークに勝るため、単一のサイズだけを見ても一般的な答えは出ない。重要なのは、規模を大きくしたときにそれぞれがどう改善するかである。Peebles and Xie (2023) は 4 種類のサイズの DiT を、それぞれ 3 種類のパッチサイズで訓練した。訓練ステップ数を固定すると、ネットワークを大きくした場合でもパッチを小さくした場合でも、順伝播あたりの演算量が増えるにつれて FID は着実に低下した。700 万ステップ訓練された最大のモデルは、256×256 のクラス条件付き ImageNet において FID の最良記録を更新した。

同じ測定を本節の規模で両方のバックボーンに対して行おう。2 つの DiT は原論文の大半のモデルと同様にヘッドあたり 64 次元を保ちながら幅と深さをともに拡大したもので、DiT-S（幅 128、深さ 4、ヘッド数 2）および先ほど構築した DiT-M である。2 つの U-Net はチャネル数を `ch=32` から先ほどのピクセルモデルである `ch=64` へと倍増させたものである。いずれもピクセル上で同じ損失と手順により 20,000 回の更新にわたり、異なるシードから 2 回ずつ訓練し、1 訓練ステップの FLOPs によってその規模を測る。

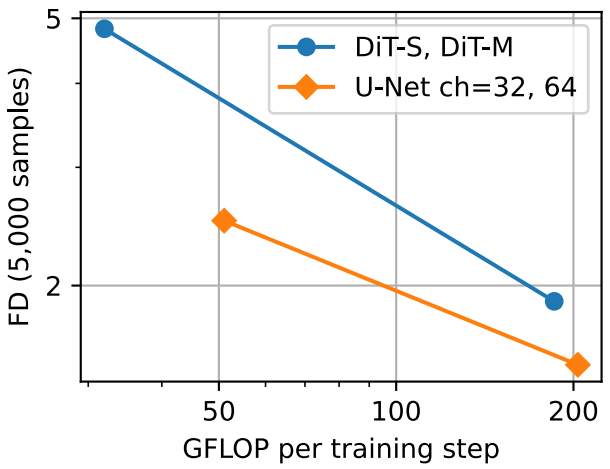
```
sizes = {'DiT-S': dict(width=128, depth=4, num_heads=2),
         'DiT-M': dict(width=256, depth=6, num_heads=4)}
nets = {'DiT-S': lambda: d2l.DiT(**sizes['DiT-S']),
        'DiT-M': lambda: d2l.DiT(**sizes['DiT-M']),
        'U-Net 32': lambda: d2l.UNet(ch=32),
        'U-Net 64': lambda: d2l.UNet(ch=64)}
seeds = {'DiT-S': (5, 12), 'DiT-M': (6, 13), 'U-Net 32': (7, 14),
```

```
            'U-Net 64': (15,)}  # the first U-Net 64 is the pixel model above
    runs = {name: dict(fd=[]) for name in nets}
    runs['U-Net 64'].update(fd=[fd_pixel], imgs=x_pixel[:8])
    for name, make_net in nets.items():
        for seed in seeds[name]:
            torch.manual_seed(seed)
            net = make_net().to(dev)
            _, ema, _ = d2l.train_regression(net, loss_fn, batch_fn, 20_000)
            imgs = sample(ema, image_shape)
            runs[name]['fd'].append(fd(imgs))
            runs[name].setdefault('imgs', imgs[:8])
        runs[name].update(params=d2l.num_params(net), gflops=d2l.train_step_gflops(
            net, loss_fn, image_shape))
    for name, run in runs.items():
        print(f'{name:8} {run["params"] / 1e6:5.2f}M parameters '
              f'{run["gflops"]:4.0f} GFLOP per step   FD '
              + ', '.join(f'{v:.2f}' for v in run['fd']))
```

```
DiT-S     1.26M parameters   32 GFLOP per step   FD 4.82, 4.83
DiT-M     7.39M parameters  185 GFLOP per step   FD 1.89, 1.89
U-Net 32  0.34M parameters   51 GFLOP per step   FD 2.47, 2.52
U-Net 64  1.20M parameters  203 GFLOP per step   FD 1.54, 1.51
```

各モデルの 2 回の実行における平均 FD を、ステップあたりの FLOPs に対して両対数軸でプロットする。

```
    curves = [[runs['DiT-S'], runs['DiT-M']], [runs['U-Net 32'], runs['U-Net 64']]]
    d2l.plot([[run['gflops'] for run in curve] for curve in curves],
             [[sum(run['fd']) / 2 for run in curve] for curve in curves],
             'GFLOP per training step', 'FD (5,000 samples)',
             legend=['DiT-S, DiT-M', 'U-Net ch=32, 64'], xscale='log',
             yscale='log', fmts=('o-', 'D-'))
```



以下で述べる差はいずれも、関係する各モデルの 2 回の実行間のばらつきの少なくとも 2 倍以上あるため、3 つの読み取りは確かなものである。1 つ目は意外な結果である。DiT-S は ch=64 の

U-Net と同程度のパラメータ数（126 万個対 120 万個）を持ちながら FD はその 3 倍に達し、6 倍のパラメータを持つ DiT-M でさえまだ及ばない。28×28 の衣類画像においては、畳み込みに組み込まれた仮定の方が、パラメータ数を 6 倍に増やすことよりも価値が大きいのである。

計算量で測っても、差は縮まるものの、依然として両方の水準で U-Net が優位に立つ。小規模側では、ステップあたり約 50 GFLOP の ch=32 の U-Net が約 2.5 のスコアであるのに対し、演算量がそれより 3 分の 1 少ない DiT-S は 4.5〜4.9 である。200 GFLOP 付近では、U-Net が約 1.5、DiT-M が 1.7〜1.9 となる。それでも、規模の拡大に伴う改善幅は Transformer の方が大きい。計算量を 5.8 倍にすると DiT の FD は 2.5〜2.8 分の 1 に下がるのに対し、計算量を 4 倍にしたときの U-Net の FD は約 1.65 分の 1 にとどまる。プロット上でより急な傾きを示しているのは Transformer の直線である。

下の各行は、同じノイズから生成した DiT-S、DiT-M、および 2 つの U-Net の最初の 8 サンプルを示している。

```
d2l.show_gray(torch.cat([run['imgs'] for run in runs.values()]), 4, 8);
```

4 つのモデルはいずれもノイズ画像を 1 枚の画像へと送る常微分方程式（ODE）を積分しているため、ほとんどの列で同じ衣類を描き出している（1〜2 列では衣類の種類自体が変化している）。この 8 列を眺めただけでは、最上段の FD が最下段の 3 倍も悪いことはほとんど読み取れないだろう。フレシェ距離は分類器の特徴を通じて何千ものサンプルを比較するが、私たちの目は 8 枚しか比較していないからである。

本節の 4 点だけでは、大規模領域で Transformer を主流のバックボーンへと押し上げた主張、すなわち幅と深さを増やすだけの単純な手順で何桁もの計算量にわたり改善し続けるという事実までは示せない。その傾向を示したのは Peebles と Xie の実験であり、本節の実験ではない。ここでのプロットにはより急な傾きは現れているものの、逆転までは至っていない。このスケーリング傾向に加えて Transformer を魅力的にしているのは、その演算が大規模な行列積として実行されるため、言語モデルを大規模に訓練するための手法がそのまま流用できる点である。

17.7.4 表現とバックボーンの選択

本節の前半と後半のテーマは、最後の 1 つの実験で交差する。潜在空間において Transformer は有効だろうか。サイズ 1 のパッチで潜在表現上に DiT-M を訓練してみよう。これにより、ピクセル上のときと同じ 49 個のトークンを扱うことになり、ステップあたりにも同じ演算量（潜在 U-Net の 12〜17 倍）を費やす。

```
torch.manual_seed(8)
net = d2l.DiT(img_size=7, patch_size=1, num_channels=4,
              **sizes['DiT-M']).to(dev)
_, dit_latent, _ = d2l.train_regression(net, loss_fn, latent_batch_fn, 20_000)
gflops = d2l.train_step_gflops(net, loss_fn, latent_shape)
print(f'latent DiT-M, {gflops:.0f} GFLOP per step: FD '
      f'{fd(decode(sample(dit_latent, latent_shape))):.2f}; latent U-Net '
      f'{fd_latent:.2f}; reconstructions {fd_rec:.2f}')
```

```
latent DiT-M, 185 GFLOP per step: FD 1.89; latent U-Net 2.07; reconstructions 0.69
```

潜在 DiT-M の FD スコアは約 2（1.8〜2.2）と、潜在 U-Net（1.9〜2.2）とほぼ同じであり、それらと同様に再構成 FD の約 3 倍の位置に着地する。また、ここでも潜在表現が Transformer を助けることはない。同一の計算量で比べると潜在 DiT-M はピクセル DiT-M を上回らず、最良のピクセルモデルが本節のあらゆる潜在モデルに勝るからである。

では、オートエンコーダを導入してまで潜在表現を使う価値があるのはどのような場合だろうか。潜在表現による節約効果はピクセル数とともに増大するが、その代償であるオートエンコーダの歪みとモデル化の難化はそうではない。512×512 の解像度では、Stable Diffusion の潜在表現上で動作する U-Net が扱う位置数は 64 分の 1 になり、演算量もおよそその比率で減少する（本節でも、16 倍の位置数を扱うピクセル U-Net は潜在 U-Net の 13〜18 倍の演算量を要していた）。この節約は訓練の各更新と 50 回のサンプリングステップのすべてに効いてくるため、その画像サイズになればオートエンコーダの代償を上回る。

Transformer の場合、計算量の事情は異なる。$64 \times 64 \times 4$ の潜在表現上でサイズ 2 のパッチを使うと DiT が扱うトークン数は 1,024 個になるが、ピクセル上でサイズ 16 のパッチを使っても、ほぼ同じコストで同じ 1,024 個のトークンを扱うことになる（演習問題 2）。オートエンコーダは Transformer のトークン数を減らさない。その役割は、$16 \cdot 16 \cdot 3 = 768$ 個の生のピクセル値の線形埋め込みを、学習された非線形な埋め込みに置き換えることである。これほど多くの値の線形埋め込みはノイズへの対処が不得手である。画像がはるかに低次元の集合の近くに横たわっているのに対し、ノイズは 768 次元すべてを埋め尽くすからである。近年の 2 つの設計は、この問題を中心に据えてピクセル空間の Transformer を構築している。Li and He (2026) は大きなパッチを維持したままノイズや速度ではなくクリーンな画像を予測し（演習問題 6）、砂時計型 Transformer（hourglass transformer）(Crowson et al. 2024) は U-Net の多重解像度の形状を取り入れて細かい解像度では局所的にのみアテンションを行うことで、コストがピクセル数に対して線形にしか増えないようにしている。

以上をまとめよう。ピクセル空間での拡散が計算予算（compute budget）に収まる低解像度画像では、U-Net を用いてピクセルを直接拡散すればよい。高解像度画像では、オートエンコーダを再構成の歪みと潜在表現のモデル化のしやすさの両方を左右する生成パイプラインの中核部分として位置づけたうえで、軽く KL 正則化したオートエンコーダの潜在表現を拡散させ、大きな計算予算にわたってスケールさせる際には DiT バックボーンを用いるのがよい。

17.7.5 まとめ

潜在拡散は、KL 正則化オートエンコーダによって画像を低次元の空間グリッドへと圧縮し、潜在表現を単位標準偏差に再スケーリングしたうえで、潜在空間において標準的な拡散またはフローマッチングの回帰を実行する。オートエンコーダはレート（潜在次元数）、歪み（再構成 FD）、およびモデル化のしやすさ（拡散モデルが潜在分布をどれだけ容易に学習できるか）のバランスをとる。KL 重み β を大きくするとランダムな潜在摂動に対するデコーダの耐性は高まるものの、再構成品質と生成サンプルの FD はともに悪化する。ノイズ除去バックボーン自体については、拡散 Transformer（DiT）が画像や潜在グリッドをパッチへとトークン化し、ゼロ初期化された残差ゲートを持つ適応的層正規化である `adaLN-Zero` を通じて条件付けを注入する。28×28 画像における小さな計算予算では、同一 FLOPs で比較すると畳み込み U-Net が DiT を上回る一方、ステップあたりの計算量を拡大すると DiT の方がより急な改善の傾きを示す。

17.7.6 演習問題

1. **潜在表現のスケール**。線形経路はデータを単位分散のノイズと混合する。
   1. 各潜在値の標準偏差が 1 ではなく ς であるとする。時刻 t における信号対雑音比が $\varsigma^2(1-t)^2/t^2$ となり、したがってあらゆる時刻で対数信号対雑音比が $\log \varsigma^2$ だけシフトすることを示せ。
   2. $\varsigma = 5$ の潜在表現が、$\varsigma = 1$ の潜在表現における $t = \frac{1}{2}$ での信号対雑音比と同じ値を持つのはどの時刻か。そのとき、一様にサンプリングされた訓練時刻のうち、信号が依然としてノイズを上回る領域に落ちる割合はどれだけか。
   3. [code] 潜在表現に $5c$ を掛けたもの（c の代わり）を用いて潜在 U-Net を訓練し、デコード後の FD を比較せよ。この結果を、解像度に伴うスケジュールのシフト（17.6.3.2 節）と関連づけて説明せよ。
2. **高解像度におけるトークン数**。Stable Diffusion のオートエンコーダは $512 \times 512 \times 3$ の画像を $64 \times 64 \times 4$ に写像する。
   1. 潜在表現上でサイズ 2 のパッチを用いた場合と、ピクセル上でサイズ 16 のパッチを用いた場合とで、DiT が扱うトークン数はそれぞれいくつか。また、この 2 つの場合において各トークンは何個の数値を保持するか。
   2. 幅 d の DiT ブロックは約 $12d^2$ 個の重みを持ち、1 訓練ステップには重みとトークンの組あたり約 6 FLOPs と、さらにアテンションのコストがかかる。サイズ 2 のパッチを用いた潜在表現上で、バッチサイズ 256 に対し幅 1,152・深さ 28 の DiT を 1 訓練ステップ実行するときの FLOPs を見積もれ。そのうちアテンションが占める部分はどの程度の大きさか。
   3. ピクセル上でサイズ 16 のパッチを用いた場合について同様の見積もりを行え。オートエンコーダは Transformer のステップあたりのコストをどれだけ節約するか。また、位置数に比例してコストが増大する U-Net の場合はどれだけ節約するか。
3. [code] **オートエンコーダの幅と重み**。レートと KL 重みはオートエンコーダの 2 つの設定項目である。
   1. 潜在チャネル数を 2、4、8 としたオートエンコーダをそれぞれ訓練し、各モデルの再構成 FD と、その潜在表現上で訓練した潜在 U-Net のデコード後の FD を報告せよ。2 つの数値のうちより大きく動くのはどちらか。また、潜在表現が大きくなるにつれて両者の差は縮まるか。この結果は、Stable Diffusion のオートエンコーダが 4 チャネルであることや、後続のモデルが 16 チャネル以上を用いていることについて何を示唆するか。
   2. 本節の摂動潜在表現テストを $\beta \in \{0, 10^{-2}\}$ に拡張し、潜在表現の標準偏差、エンコーダの標準偏差、および超過尖度もあわせて表示せよ。$\beta = 0$ では潜在表現のスケールを 1 に縛るものがなくなるが、それでもスケール係数 c によって利用可能になるのはなぜか。

3. 新たな 2 つのオートエンコーダの両方について潜在 U-Net を訓練せよ。KL 重みが小さいほど良いサンプルが得られるという本文の順序関係は、$\beta = 0$ でも維持されるか。

4. [code] **なぜゼロから始めるのか。** `ada` 層のゼロ初期化（`d2l.DiTBlock` および `d2l.DiT` 内）を、フレームワークのデフォルトの初期化に置き換えよ。
   1. 両方の初期化について、未訓練ネットワークの出力の大きさを表示せよ。
   2. 両方の DiT-M モデルを 2,000 回の更新にわたり訓練し、損失をプロットせよ。初期化時点での恒等写像の性質を通じて、その違いを説明せよ。

5. **参照としてのクロスアテンション。** 画像トークン $\mathbf{Y} \in \mathbb{R}^{m \times d}$ がキャプショントークン $\mathbf{X} \in \mathbb{R}^{n \times d}$ にクエリを送る状況を考える。クエリを $\mathbf{Q} = \mathbf{Y}\mathbf{W}_q$、キーを $\mathbf{K} = \mathbf{X}\mathbf{W}_k$、値を $\mathbf{V} = \mathbf{X}\mathbf{W}_v$ とすると、この層は $\mathrm{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d})\,\mathbf{V}$ を出力する。
   1. 各画像位置がキャプションの値ベクトルの凸結合を受け取ることを示せ。また、1 トークンからなる空のキャプションを用いた場合、すべての位置が同じベクトルを受け取り、したがってガイダンスにおける無条件予測にとってクロスアテンションが学習された定数シフトとして働くことを示せ。
   2. 仮に値を画像側から $\mathbf{V} = \mathbf{Y}\mathbf{W}_v$ として計算したとする。$m \neq n$ のとき形状にどのような不整合が生じるか。また、$m = n$ のときそのような層は何を計算することになるか。
   3. クロスアテンションではキャプションのキーと値は $\mathbf{x}_t$ に依存しないが、マルチモーダル DiT ではすべてのブロックでキャプショントークンが画像トークンによって更新される。77 トークンのキャプション、1,024 個の画像トークン、ガイダンス付き 50 ステップの条件において、クロスアテンションモデルがサンプルごとに 1 回だけ実行すれば済むキャプション側の計算はどれか。また、マルチモーダル DiT のトークン計算のうちキャプションが占める割合はどれだけか。

   小問 (b) は CMU 10-423/623「Generative AI」Homework 4, Question 2.3（2026 年春学期）を改変したものである。

6. [code] **大きなパッチとクリーン画像のターゲット。●** `DiT` を `patch_size=7`（16 個の各トークンが 49 個のピクセル値を保持する設定）でピクセル上に訓練せよ。幅 32（ヘッド数 1）と幅 256 の 2 通りについて、それぞれ 10,000 回の更新を行うこと。さらに各幅に対し、同じ速度損失のもとで 2 通りの出力パラメータ化を与えよ。すなわち、ネットワークが速度を直接出力する場合と、$\hat{\mathbf{x}}_0$ を出力して速度を $(\mathbf{x}_t - \hat{\mathbf{x}}_0)/t$（ただし t の下限を 0.05 にクランプする）として構成する場合である。
   1. 4 つのモデルの FD を 2×2 の表にまとめて報告せよ。
   2. トークンが保持する値の個数がネットワークのチャネル数を上回る幅 32 において、クリーン画像の出力は有効か。また、幅 256 ではその差は縮まるか。
   3. 画像はピクセル数よりはるかに低次元の集合の近くに横たわる一方、ノイズはすべての次元を埋め尽くすという 17.6 節の議論を用いて結果を説明せよ。さらに Li and He (2026) の知見と比較せよ。

17.8 少数ステップ生成

拡散モデルは、ネットワークを何度も繰り返し呼び出すことでサンプルを生成する。17.5 節のサンプラは祖先サンプリング連鎖における 1,000 回の呼び出しを約 50 回に減らし、17.7 節の潜在モデルは 1 回あたりの呼び出しコストを下げたが、それでも 50 回は 50 回であり、GAN（第 16 章）の生成器に必要な呼び出しは 1 回だけであった。では、1 回の長いステップで済ませてはどうだろうか。これはうまくいかない。しかも、良いネットワークがないからではない。厳密な速度場（velocity field）を用いたとしても、確率フロー ODE（probability-flow ODE）の 1 回のオイラーステップはあらゆる出発点をデータの平均へと送り込んでしまうからである（17.3 節）。3 つの塊からなる混合分布では、その点は塊と塊の間のデータがまったく存在しない場所にある。

速度がサンプラに教えるのはどの方向へ進むかであり、1 ステップで生成するには最終的にどこへ到達するかを知る必要がある。そこで、常微分方程式（ordinary differential equation、ODE）の右辺ではなく解そのもの、すなわちある時刻の点からその軌跡がより早い時刻に位置する点への写像をネットワークに学習させる。Song et al. (2023) は、訓練済みの拡散モデルである**教師**（teacher）を新しいネットワークである**生徒**（student）へと蒸留することでデータへのジャンプを学習させ、その生徒を**一貫性モデル**（consistency model）と呼んだ。Geng et al. (2025) は、データのみから任意の区間にわたるジャンプを学習させ、この手法を **MeanFlow** と呼んだ。訓練は単一のネットワークによる回帰のままであり、同じノイズからは常に同じサンプルが得られるため、生徒と教師を画像ごとに比較できる。これに対し 16.7 節ではクリティックを用いて 1 ステップ生成器を訓練したが、その路線には本節の最後で立ち返る。まずは 1 回あるいは数回のオイラーステップが何を誤るのかを確認し、続いて両手法に共通する 1 つの損失を導出して、これまでの例題と Fashion-MNIST で生徒を訓練する。

```
import copy
```

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.8.1 長いオイラーステップ

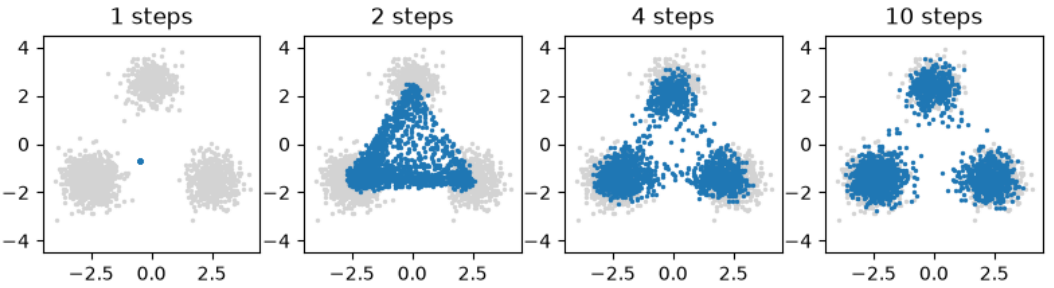
厳密な速度場において、長いオイラーステップが犯す誤差はただ 1 つ、軌跡が曲がっているのに接線に沿って進んでしまうことである。$t = 1$ ではこの誤差が最大になる。どの出発点からも速度はデータの平均を向いており、長さ 1 のステップはそこに着地する（17.3 節）。それ以降のステップはいずれかのモードへ向かい始めた点から出発するため狙いは定まってくるが、それでも各ステップは曲線を横切ってショートカットしてしまう。

17.8.1.1 厳密な速度場でのオイラーステップ

17.3 節では 1、10、100 ステップの結果を見たが、2 ステップと 4 ステップではどうなるだろうか。`mix.velocity` は混合分布の厳密な速度場を評価し、`d2l.make_pred` はそれを `d2l.ode_sample` が積分する予測器へと変換する。ここでは同じ 20,000 個のガウス初期点から、1、2、4、10 ステップでオイラー法を実行する。2 ステップと 4 ステップの行が、本節の生徒モデルが上回るべきベースラインとなる。

```
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)
lin = d2l.GaussianPath.linear()
pred_exact = d2l.make_pred(mix.velocity, lin, 'velocity')
torch.manual_seed(0)
z = torch.randn(20_000, 2, device=device)
euler = {n: d2l.ode_sample(pred_exact, z, lin.sigma_bars(n))
         for n in (1, 2, 4, 10)}
for n, x in euler.items():
    s = mix.summary(x)
    print(f'{n:2d} Euler steps: weights {s["weights"]}, '
          f'stray {s["stray"]:.1%}')
d2l.scatter_panels([x[:2000] for x in euler.values()],
                   [f'{n} steps' for n in euler], data=mix.sample(2000),
                   size=2.4);
```

```
 1 Euler steps: weights [1.0, 0.0, 0.0], stray 100.0%
 2 Euler steps: weights [0.644, 0.217, 0.14], stray 25.1%
 4 Euler steps: weights [0.54, 0.277, 0.183], stray 6.5%
10 Euler steps: weights [0.512, 0.291, 0.197], stray 2.2%
```



1 ステップではすべてのサンプルが平均 $(-0.5, -0.7)$ に集まり、`summary` はこれを逸脱サンプル（stray）として数えつつ、最も近い最大重みのモードに割り当てるため、重みは 1/0/0 となる。2 ステップでは各モードに到達するものの、サンプルの約 4 分の 1 がモード間に取り残され、重みも本来の 0.5/0.3/0.2 に対して約 0.64/0.22/0.14 となる。最初の半ステップが依然として最大重みのモードに最も近い平均を狙うため、そのモードの割合が増えてしまうのである。4 ステップでも約 6.5%、10 ステップでも約 2% のサンプルが逸脱したままであるのに対し、データ自体の逸脱率は約 1% である。速度場が厳密であるため、この超過分はすべてステップ幅に起因する誤差である。

17.8.1.2 Reflow

そもそもなぜ軌跡は曲がるのだろうか。各訓練例はデータ点と独立に抽出されたノイズベクトルを直線分で結ぶが、異なる訓練例の線分どうしは交差する。交差する場所では速度場がそれらの方向を平均化するため、互いに交差できない軌跡は互いを避けるように曲がるのである（38.7.2 節）。この曲率は損失ではなく、ノイズとデータのペアリングから生じている。

Reflow (Liu et al. 2023) はこのペアリングを変更する。各ノイズベクトルを、訓練済みモデルがそこから到達するサンプルとペアにし、そのペアを用いてネットワークを再訓練するのである。新しい各線分は元のモデルの 1 本の軌跡に対する弦（chord）であり、元の軌跡どうしは決して交差しないため、それらの弦もめったに交差しない。38.7.2.3 節では、厳密な速度場を用いた Reflow がデータ分布を保ち、輸送コストを決して増加させないことを証明しており、2 つの三日月（two-moons）データでは Reflow 後のモデルの 1 回のオイラーステップが元のモデルの多数ステップと同等のスコアを達成する。

しかし、訓練された速度場ではこの保証が弱まる。新しいペアの終点は実データではなくモデルの生成サンプルであるため、Reflow 後のモデルは元のモデルの誤差をそのまま引き継ぐ。2 つの三日月データでは欠陥も含めて元のモデルのサンプルが再現され（38.7.2.3 節）、画像モデルでも同様に生成サンプルのあらゆる欠陥が受け継がれる。さらに 1 ラウンドごとに 1 回分の訓練コストがかかるうえ、Liu et al. (2023) における最良の 1 ステップ結果も結局は Reflow 後のモデルを蒸留して得られたものであった。そこで本節の残りでは、データのペアリング方法ではなく、ネットワークが何を出力するかの方を変更する。

17.8.2 フロー写像

速度モデルは軌跡がどちらを向いているかを表し、フロー写像（flow map）は軌跡がどこに到着するかを表す。

17.8.2.1 速度の積分

確率フロー ODE $d\mathbf{x}_\tau/d\tau = \mathbf{u}_\tau(\mathbf{x}_\tau)$ を、点 $\mathbf{x}$（時刻 t）からより早い時刻 $s \le t$ まで逆向きに走らせる。その到着点が**フロー写像** $\Phi_{t\to s}(\mathbf{x})$ である。速度場が $\mathbf{x}$ についてリプシッツ連続であれば、厳密なフロー写像は全単射（bijection）となり（38.1.1 節）、求めたいサンプラは $\Phi_{1\to 0}$ である。

軌跡 $\mathbf{x}_\tau$（$\mathbf{x}_t$ を通る解）に沿ってこの ODE を積分すると、フロー写像は次のように変位として表される。

$$\Phi_{t\to s}(\mathbf{x}_t) = \mathbf{x}_t - \int_s^t \mathbf{u}_\tau(\mathbf{x}_\tau)\, d\tau = \mathbf{x}_t - (t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t), \tag{17.57}$$

ここで、$\bar{\mathbf{u}}(\mathbf{x}_t, s, t)$ は 2 つの時刻間における軌跡上の**平均速度**（average velocity）である。オイラー法はこれを始点での速度 $\mathbf{u}_t(\mathbf{x}_t)$ で置き換えるため、ステップサイズとともに増大する誤差が生じる（38.2 節）。長さ 0 の区間では $\bar{\mathbf{u}}(\mathbf{x}, t, t) = \mathbf{u}_t(\mathbf{x})$ となって両者は一致するため、$\bar{\mathbf{u}}$ を表すネットワークは、速度モデルに終点時刻というもう 1 つの入力を加えたものになる。

最小の例を見れば、瞬間速度にはわからず平均速度にはわかっていることが明らかになる。データを平均 m、標準偏差 ς の 1 次元ガウス分布とする。このとき x_t は平均 $(1-t)\,m$、標準偏差 $k_t = \sqrt{(1-t)^2\varsigma^2 + t^2}$ のガウス分布に従う。1 次元では軌跡どうしが追い越すことはできないため、フローは各点の分位点（quantile）を保ち、ガウス分布における分位点は標準化された位置 $(x - (1-t)m)/k_t$ によって決まる。したがって、フロー写像は次のアフィン写像になる。

$$\Phi_{t\to s}(x) = (1-s)\,m + \frac{k_s}{k_t}(x - (1-t)\,m). \tag{17.58}$$

$m = 2$、$\varsigma = 0.5$ とすると、全区間のジャンプ $\Phi_{1\to 0}(x) = 2 + 0.5\,x$ は標準ガウスノイズをデータ分布 $\mathcal{N}(2, 0.5^2)$ へと写し、その平均速度は $\bar{u}(x, 0, 1) = 0.5\,x - 2$ となる。始点での速度は $u_1(x) = x - 2$ であり、そのオイラーステップは定数 2 をあらゆる x に対して返す。データの広がりとしてノイズの広がりの半分を残す必要があるため、平均速度はノイズの半分しか取り除かないのである。

誰もその値を教えてくれないなかで、ネットワークはどうやって Φ を学習できるのだろうか。それは、真のフロー写像だけが満たす恒等式を損失へと変換することによって可能になる。軌跡に沿って 2 つの時刻を動かす方法ごとに 1 つの恒等式が存在する（Figure 17.14）。$\mathbf{x}_\tau$ を点 $\mathbf{x}$（時刻 t）を通る軌跡とし、$s \le r \le t$ とする。

合成（compositional）恒等式は、2 回のジャンプが 1 回のジャンプに等しいこと、すなわち $\Phi_{r\to s}(\Phi_{t\to r}(\mathbf{x})) = \Phi_{t\to s}(\mathbf{x})$ を表す。これに基づく損失は導関数を必要とせず、中間時刻におけるモデルの評価だけで済み、ターゲットの精度はそれを構成する短いジャンプの精度で決まる。漸進的蒸留（progressive distillation）の手法 (Salimans and Ho 2022) やショートカットモデル (Frans et al. 2025) がこれを用いている。

ラグランジュ型（Lagrangian）恒等式は、始点を固定して終点を動かす。すると写像された点は粒子のように速度場に従って動き、$\partial_s \Phi_{t\to s}(\mathbf{x}) = \mathbf{u}_s(\Phi_{t\to s}(\mathbf{x}))$ を満たす。これに基づく損失には、ジャンプの着地点における速度が必要になる。訓練例から条件付き速度（conditional velocity）が得られるのは自身の $\mathbf{x}_t$ においてのみであるため、着地点での速度は教師モデルか、あるいはフロー写像自身が供給しなければならない（演習問題 3）。

オイラー型（Eulerian）恒等式は、終点を固定して始点を軌跡に沿って滑らせる。1 本の軌跡上のどの点も同じ場所へと写像されるため、次式が成り立つ。

$$\frac{d}{dt}\Phi_{t\to s}(\mathbf{x}_t) = \partial_t \Phi_{t\to s}(\mathbf{x}_t) + \nabla_{\mathbf{x}} \Phi_{t\to s}(\mathbf{x}_t)\, \mathbf{u}_t(\mathbf{x}_t) = \mathbf{0}. \tag{17.59}$$

これに基づく損失が必要とするのは 1 回のヤコビ行列・ベクトル積（Jacobian-vector product）と始点での速度だけであり、始点には訓練例の速度が存在する。さらにターゲットに含まれるのは 1 組の時刻だけで、中間時刻もステップサイズも不要である。

ガウス分布の例では、3 つとも手計算で確かめられる。比 k_s/k_t は合成則を満たし、点 $\Phi_{t\to s}(x)$ を s を動かしながら追跡すると標準化された位置が保たれるためフローの点と同じように動き、さらにその標準化された位置は軌跡に沿って一定である。

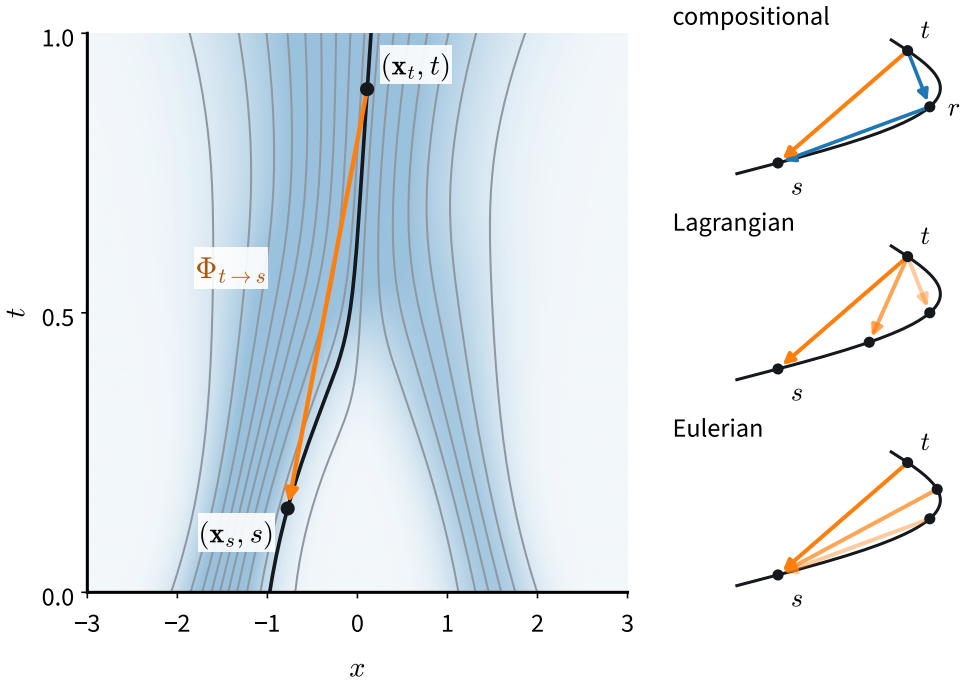


Figure 17.14: フロー写像。左：2 つのガウス分布からなる 1 次元混合分布に対する確率フロー ODE の軌跡であり、$t = 1$ のノイズ（上）から $t = 0$ のデータ（下）へと向かう。$(\mathbf{x}_t, t)$ から $(\mathbf{x}_s, s)$ への弦がフロー写像の 1 回のジャンプを表す。右（1 本の軌跡上）：r を経由する 2 回のジャンプが 1 回のジャンプになる（合成）、始点を t に固定して終点を動かす（ラグランジュ型）、終点を s に固定して始点を動かす（オイラー型）。

写像が恒等写像となり $s = t$ において s に関する傾きが速度に一致するという条件と組み合わせれば、どの恒等式からもフロー写像が一意に定まる。違いは、損失が何を計算しなければならないかだけである。合成恒等式が最も単純であり、中間時刻もジャンプの着地点における速度も必要としないのはオイラー型恒等式だけであるが、その代わりに 1 回のヤコビ行列・ベクトル積を必要とする。一貫性モデルと MeanFlow はどちらもオイラー型恒等式を用いており、Boffi et al. (2025) は課す恒等式によって少数ステップ手法を分類している。

17.8.2.2 一貫性モデル

最初のフロー写像手法は、データへのジャンプのみを保持するものであった。**一貫性モデル** (Song et al. 2023) とは、ネットワーク $f(\mathbf{x}, t)$ を用いてあらゆる時刻で $\Phi_{t\to 0}(\mathbf{x})$ を出力するよう訓練したモデルである。その訓練条件は、入力が軌跡に沿って動いても出力が変化しないこと、すなわち $\frac{d}{dt} f(\mathbf{x}_t, t) = \mathbf{0}$ かつ $f(\mathbf{x}, 0) = \mathbf{x}$ であり、これはオイラー型恒等式 Equation 17.59 において $s = 0$ としたものにほかならない。

一貫性蒸留（consistency distillation）において、Song らは時間グリッドの隣接する時刻間で教師に 1 回の ODE ステップを踏ませ、その 2 点における生徒の出力が一致することを要請した。より早い時刻の点での出力がターゲットとなり、これは固定されたうえで生徒の重みの移動平均を用いて計算される。グリッドを細かくしていくと、この出力差をステップ幅で割ったものは上記の導関数に収束し、Lu and Song (2025) はこれを直接用いている。**一貫性訓練**（consistency training）は教師を必要とせず、教師の速度を各訓練例の直線分の速度 $\epsilon - \mathbf{x}_0$、すなわちフローマッチング（flow matching）の条件付き速度で置き換える。

一貫性モデルを 1 回評価すればサンプルが 1 つ得られる。しかし、このモデルは軌跡の終点しか知らないため、2 回目の評価を行う際に $t = 1$ から $t = 0.5$ へジャンプすることはできない。その代わりに**再ノイズ化**（re-noising）を行う。すなわち、新たなノイズを用いてサンプルを $t = 0.5$ へと戻し（$\mathbf{x}_{0.5} = 0.5\,\mathbf{x}_0 + 0.5\,\epsilon'$）、軌跡の曲がりがより小さいその点から再びデータへと写像するのである。始点と終点の両方の時刻を持つフロー写像であれば再ノイズ化は不要であり、2 回の評価は 1 本の軌跡に沿った 2 回のジャンプとなって、サンプルは初期ノイズのみに依存したまま保たれる。

17.8.2.3 平均速度と MeanFlow 恒等式

フロー写像を Equation 17.57 のように平均速度で表すと、オイラー型恒等式は回帰目標（regression target）へと変わる。平均速度に区間の長さを掛けて速度の積分とし、軌跡に沿って t で微分する（ただし s は固定する）。

$$\frac{d}{dt}\Big[(t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t)\Big] = \frac{d}{dt}\int_s^t \mathbf{u}_\tau(\mathbf{x}_\tau)\,d\tau = \mathbf{u}_t(\mathbf{x}_t). \tag{17.60}$$

左辺に積の微分法則を適用すると、Geng et al. (2025) の **MeanFlow 恒等式**が得られる。

$$\bar{\mathbf{u}}(\mathbf{x}_t, s, t) = \mathbf{u}_t(\mathbf{x}_t) - (t-s)\,\frac{d}{dt}\bar{\mathbf{u}}(\mathbf{x}_t, s, t). \tag{17.61}$$

平均速度は、始点での速度に対して、始点が軌跡上を滑るときに平均速度が変化する速さで補正を加えたものになっている。Equation 17.57 を Equation 17.59 に代入すれば、2 つの恒等式が同一であることがわかる。全微分（total derivative）は $\partial_t \bar{\mathbf{u}} + (\nabla_{\mathbf{x}} \bar{\mathbf{u}})\,\mathbf{u}_t$ であり、これはネットワーク $(\mathbf{x}, t) \mapsto \bar{\mathbf{u}}(\mathbf{x}, s, t)$ と接ベクトル $(\mathbf{u}_t, 1)$ とのヤコビ行列・ベクトル積にほかならない。フォワードモード微分を用いれば、この量は出力と同時に逆伝播 1 回分程度のコストで計算できる。したがって、1 回の訓練ステップにかかるコストは、速度モデルのステップに比べて逆伝播約 1 回分多くなり、教師がある場合にはそこに教師の評価 1 回分が加わる。

ネットワーク自身から計算して固定した右辺へとネットワークを回帰させることで、この恒等式は損失になる。周辺速度（marginal velocity）$\mathbf{u}_t$ は直接利用できないが、2 つの代替手段がある。教師があれば、その推定値 $\hat{\mathbf{u}}$（$\mathbf{u}_t(\mathbf{x}_t)$ の近似）が得られる。教師がなければ、フローマッチングが機能したのと同じ理由により、訓練例の条件付き速度 $\epsilon - \mathbf{x}_0$ がその役割を果たす。ターゲットは直接的にもヤコビ行列・ベクトル積の接ベクトルを通じても速度に関してアフィンである。し

たがって、$\mathbf{x}_t$ を生じ得た訓練例全体で平均をとれば、条件付き速度を $\mathbf{u}_t(\mathbf{x}_t)$ に置き換えたターゲットに一致し、二乗誤差はその平均によって最小化される。用いる速度を $\mathbf{u}_{\text{ref}}$ と書き、これを**参照速度**（reference velocity）と呼ぶと、ある訓練点における損失は次のようになる。

$$\left\| \bar{\mathbf{u}}_\theta(\mathbf{x}_t, s, t) - \text{sg}\left[\mathbf{u}_{\text{ref}} - (t-s)\, \frac{d}{dt} \bar{\mathbf{u}}_\theta(\mathbf{x}_t, s, t) \right] \right\|^2, \tag{17.62}$$

ここで sg は**勾配停止**（stop-gradient）であり、損失を微分する際に定数として扱う量を表す。これは先ほどの平均化の議論に不可欠な操作である。その議論は、ネットワークが予測値のみを通じて損失に入る場合に限り成り立つからである。もしターゲット側まで微分を通せば、ネットワークは恒等式を満たす代わりにターゲット内部のヤコビ行列を変形させて訓練例間のターゲットのばらつきを変えることで、損失を下げられてしまう（演習問題 4）。勾配停止の呼び出し 1 つがあるかないかだけの違いしかないため、導出の全ステップのなかで再実装時に最も誤りやすい箇所がここである。

この呼び出しは `.detach()` である。

この損失には 2 つの選択肢が残されている。$\mathbf{u}_{\text{ref}}$ をどこから得るか、そして終点時刻 s を自由にするか 0 に固定するかである。主要な手法はこの 4 通りの組み合わせに分類される。ただし、この損失以外の方法で恒等式を課す手法もあり、それぞれに 17.10 節で述べる独自の工夫が加えられている。

Table 17.3: Diffusion Flow Map Methods

速度 $\mathbf{u}_{\text{ref}}$	1 時刻（$s = 0$）	2 時刻（$s \le t$）
教師の $\hat{\mathbf{u}}$	一貫性蒸留	フロー写像蒸留 (Kim et al. 2024)
訓練例の $\epsilon - \mathbf{x}_0$	一貫性訓練	MeanFlow

$s = 0$ では Equation 17.57 が $f(\mathbf{x}, t) = \mathbf{x} - t\,\bar{\mathbf{u}}(\mathbf{x}, 0, t)$ となるため、左上のセルは Lu and Song (2025) による連続時間形式の一貫性蒸留にあたる。`flow_map_error` は 4 手法すべてに対して線形経路上の訓練例ごとの誤差 Equation 17.62 を実装している。ネットワーク `u` が $\bar{\mathbf{u}}$ を計算し、`teacher=None` のときは条件付き速度を選び、呼び出し側が時刻の抽出と平均化を行う。

```
def flow_map_error(u, teacher, x0, eps, s, t):
    """Per-example squared error of the average velocity u(x_t, s, t) against
    the stop-gradient target v - (t - s) du/dt; v from teacher, or eps - x0."""
    x_t = d2l.GaussianPath.linear().sample(x0, t, eps)
    if teacher is None:
        v = eps - x0
    else:
        with torch.no_grad():
            v = teacher(x_t, t)
    u_t, du_dt = torch.func.jvp(lambda x, r: u(x, s, r), (x_t, t),
                                (v, torch.ones_like(t)))
    gap = (t - s).reshape(-1, *[1] * (x0.dim() - 1))
    target = (v - gap * du_dt).detach()
    return ((u_t - target) ** 2).flatten(1).sum(1)
```

t に関する微分はネットワークの時間特徴量を通過するため、フロー写像では本章の他のモデルよりも低周波の特徴量を用いる。特徴量 $\sin(\omega t)$ の導関数は $\omega \cos(\omega t)$ であるから、各特徴量はその周波数が掛かった状態でターゲットに入る。他のネットワークは t を最大 64 までの周波数

で受け取るが（17.1 節）、フロー写像では `time_freqs=slow`、すなわち 10^{-4} から 1 までの周波数（下のコードで `slow` として定義）を渡す。高い周波数は導関数に大きな係数を生じさせ（たとえば $1000\,t$ の埋め込みを用いるコードでは最大 1,000 倍される）、これは Lu and Song (2025) が一貫性モデルにおいて報告している不安定性の原因となる。

フロー写像は $\mathbf{x}_t$ を $\mathbf{x}_s$ へと 1 回のジャンプ $\mathbf{x}_s = \mathbf{x}_t - (t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t)$ で写すため、少数ステップのサンプラは $[0,1]$ を等間隔の区間に分割し、各区間を順にジャンプする。

```
@torch.inference_mode()
def flow_map_sample(u, x, steps, t_start=1.0):
    """Jump from x at t_start (1 is noise) to data at t = 0 in equal steps."""
    ts = torch.linspace(t_start, 0, steps + 1, device=x.device)
    for t, s in zip(ts[:-1], ts[1:]):
        x = x - (t - s) * u(x, s.expand(len(x)), t.expand(len(x)))
    return x
```

速度 $\mathbf{u}_t$ を $\bar{\mathbf{u}}$ の代わりに渡せば同じ関数がそのままオイラー法になるため、1 つのサンプラを教師と生徒の両方に使える。ただし、ジャンプは一方向のみである。本節のすべての損失は $s \le t$ として時刻を引くため、17.5 節で ODE を逆向きに解いたようにサンプルを反転するには、逆方向にも写像を訓練する必要がある。

次節末尾の画像実験では、生徒を教師の U-Net のコピーから出発させる。時間幅 $t-s$ は専用の時間埋め込みとゼロ初期化した線形変換を通して U-Net の埋め込みに加算されるため、訓練開始時点では $\bar{\mathbf{u}}(\mathbf{x}, s, t) = \hat{\mathbf{u}}(\mathbf{x}, t)$ となって教師の速度と一致する。

```
class UNetFlowMap(nn.Module):
    """u(x, s, t): a velocity U-Net, plus a zero-initialized input for the
    gap t - s. The caller passes a copy of the teacher's U-Net."""
    def __init__(self, unet):
        super().__init__()
        # the teacher is an EMA copy with gradients off; the student trains
        self.unet = unet.requires_grad_(True)
        self.gap = nn.Linear(unet.emb_dim, unet.emb_dim)
        nn.init.zeros_(self.gap.weight), nn.init.zeros_(self.gap.bias)

    def forward(self, x, s, t):
        gap = self.gap(d2l.time_embedding(t - s, self.unet.emb_dim,
                                          self.unet.time_freqs))
        return self.unet(x, t, cond=gap)
```

17.8.3 フロー写像の訓練

17.8.3.1 3 モード混合分布

この損失は実際に機能するのだろうか。また、参照速度をどこから得るかは結果に影響するのだろうか。この混合分布ならその両方に答えられる。厳密な速度場が既知であるため、完璧なサンプラがどのような数値を出力するかがわかるからである。教師は 17.2 節と同様に訓練した速度モデル `TimeMLP` であり、入力点と t の正弦特徴量を幅 256 の層に通すネットワークを、ここでは 20,000 ステップ訓練する。

```
torch.manual_seed(1)
_, teacher, _ = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(lin, 'velocity'),
    lambda: mix.sample(512), 20_000, cosine=True)
```

3 種類の生徒が Table 17.3 の 3 つのセルを埋める。いずれも `TimeMLP` として始点 t と区間幅 $t-s$ を受け取り、単一実行の偶然と手法固有の性質を見分けられるよう、教師と同じ設定でそれぞれ 6 回ずつ（乱数シード 2〜7）訓練する。一貫性モデルの生徒は t を一様に引き、$s=0$ に固定して教師の速度を用いる。フロー写像蒸留（flow-map distillation）の生徒は一様乱数で 2 つの時刻を引いて小さい方を s とし、同様に教師の速度を用いる。MeanFlow の生徒も同じ方法で時刻を引くが、教師は用いない。4 つ目のセルである一貫性訓練には専用の生徒を用意する必要がない。その 2 つの選択（1 時刻と訓練例の速度）は、それぞれ他の生徒によって検証されるからである。

損失はどの程度の値になると予想されるだろうか。蒸留される生徒のターゲットは位置と時刻のみの関数であるため、その損失は 0 に近づき得る。一方、MeanFlow のターゲットには各訓練例固有の速度が $\mathbf{u}_{\text{ref}}$ として 1 回、接ベクトルの方向としてもう 1 回の計 2 回含まれるため、同じ位置と時刻における訓練例間のターゲットのばらつき（分散）より下に損失が下がることはない。

```
class MLPFlowMap(d2l.TimeMLP):
    """u(x, s, t): a TimeMLP on the start t and the interval t - s."""
    def forward(self, x, s, t):
        return super().forward(x, t, t - s)

slow = (1e-4, 1.0)  # time features of a flow map, whose loss has a d/dt

def consistency_loss(u, x0):  # s = 0, the teacher's velocity
    t = torch.rand(len(x0), device=device)
    return d2l.flow_map_error(u, teacher, x0, torch.randn_like(x0),
                              torch.zeros_like(t), t).mean()

def distillation_loss(u, x0):  # s < t, the teacher's velocity
    a, b = torch.rand(2, len(x0), device=device)
    s, t = torch.minimum(a, b), torch.maximum(a, b)
    return d2l.flow_map_error(u, teacher, x0, torch.randn_like(x0), s,
                              t).mean()

def meanflow_loss(u, x0):  # s < t, no teacher
    a, b = torch.rand(2, len(x0), device=device)
    s, t = torch.minimum(a, b), torch.maximum(a, b)
    return d2l.flow_map_error(u, None, x0, torch.randn_like(x0), s, t).mean()

students = {}
for name, loss in (('consistency', consistency_loss),
                   ('distillation', distillation_loss),
                   ('MeanFlow', meanflow_loss)):
    final = []
    for seed in range(2, 8):
        torch.manual_seed(seed)
        _, students[name, seed], losses = d2l.train_regression(
            MLPFlowMap(256, 3, num_times=2, time_freqs=slow).to(device),
            loss, lambda: mix.sample(512), 20_000, cosine=True)
        final.append(float(losses[-1000:].mean()))
```

```
    print(f'{name:>12}, seeds 2 to 7: mean loss of the last 1,000 updates '
          f'{min(final):.3f} to {max(final):.3f}')
```

```
 consistency, seeds 2 to 7: mean loss of the last 1,000 updates 0.073 to 0.083
distillation, seeds 2 to 7: mean loss of the last 1,000 updates 0.030 to 0.037
    MeanFlow, seeds 2 to 7: mean loss of the last 1,000 updates 76.136 to 81.287
```

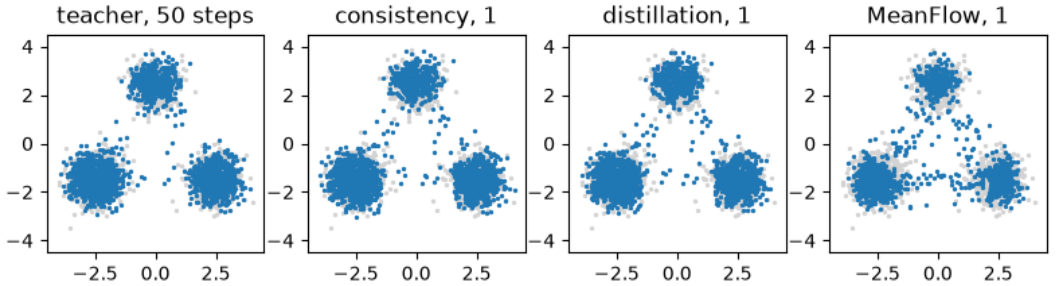
蒸留された生徒の損失は 0 付近で収束し、2 時刻の生徒で約 0.03、一貫性モデルの生徒で 0.06〜0.08 となる。これに対し MeanFlow は 6 回の実行で 76〜85 となり、一貫性モデルの生徒の約 1,000 倍の損失で終わる。その大部分は、いかなるネットワークも取り除けない訓練例間の不一致、すなわち損失の下限（フロア）である可能性がある。このコードセルではその下限を測定していないため、損失の数値自体から生徒の出来はほとんどわからない。MeanFlow の実行結果は損失ではなく生成サンプルによって評価する。

次に、すべてのモデルをオイラー法の実験と同じ 20,000 個の初期点から出発させる。教師は 50 オイラーステップ、一貫性モデルの生徒は 1 回ジャンプまたは $t = 0.5$ で再ノイズ化する 2 回ジャンプ、2 時刻の生徒は 1、2、4 回ジャンプで評価する。混合分布から新たに抽出したサンプルは、完璧なサンプラが出力する基準値を示す。表の各行には、まず各実行における 3 つのモード重みの最大誤差が並ぶ（生徒はシード順に 6 個、データと教師は 1 個）。続いて実行間の範囲として、すべてのモード中心から標準偏差の 3 倍を超えて離れたサンプルの割合である逸脱率と、逸脱しなかったサンプルから所属モードの中心までの平均二乗距離である広がりを示す（17.1 節）。

```
def renoised_sample(u, z):
    """Two evaluations of a one-time map: jump, re-noise to t = 0.5, jump."""
    x = d2l.flow_map_sample(u, z, 1)
    half = torch.full((len(z),), 0.5, device=device)
    return d2l.flow_map_sample(u, lin.sample(x, half, torch.randn_like(x)), 1,
                               t_start=0.5)

def span(values, fmt):
    """'lo-hi' over the runs, or the one value if they print alike."""
    lo, hi = format(min(values), fmt), format(max(values), fmt)
    return lo if lo == hi else f'{lo}-{hi}'

def report(name, xs):
    """Each run's largest weight error; the range of strays and spreads."""
    stats, w_true = [mix.summary(x) for x in xs], mix.weights.tolist()
    err = ' '.join(
        f'{max(abs(w - w0) for w, w0 in zip(s["weights"], w_true)):.3f}'
        for s in stats)
    print(f'{name:>24}  {err:35}  {span([s["stray"] for s in stats], ".1%"):9}'
          f'  {span([s["spread"] for s in stats], ".3f")}')

torch.manual_seed(4)
samples = {'data': [mix.sample(len(z))],
           'teacher, 50 steps': [d2l.flow_map_sample(
               lambda x, s, t: teacher(x, t), z, 50)]}
for (name, seed), u in students.items():
    if name == 'consistency':  # a one-time map re-noises for a second step
        new = {'1': d2l.flow_map_sample(u, z, 1),
               '2 re-noised': renoised_sample(u, z)}
    else:
```

```
            new = {str(k): d2l.flow_map_sample(u, z, k) for k in (1, 2, 4)}
        for k, x in new.items():
            samples.setdefault(f'{name}, {k}', []).append(x)
    print(f'{"":24}  {"largest weight error, each run":35}  {"stray":9}  spread')
    for name, xs in samples.items():
        report(name, xs)
    names = ['teacher, 50 steps', 'consistency, 1', 'distillation, 1',
             'MeanFlow, 1']
    d2l.scatter_panels([samples[k][0][:2000] for k in names], names,
                       data=mix.sample(2000), size=2.4);
```

```
                        largest weight error, each run       stray      spread
                  data  0.005                                0.9%       0.474
       teacher, 50 steps  0.003                                1.6%       0.461
          consistency, 1  0.003 0.003 0.003 0.003 0.003 0.003  2.0%-2.1%  0.457-0.461
consistency, 2 re-noised  0.003 0.002 0.004 0.004 0.003 0.003  1.2%-1.5%  0.472-0.482
         distillation, 1  0.003 0.003 0.003 0.003 0.003 0.003  1.9%-2.1%  0.446-0.453
         distillation, 2  0.004 0.004 0.004 0.003 0.004 0.004  1.0%-1.1%  0.464-0.473
         distillation, 4  0.004 0.003 0.003 0.003 0.004 0.003  0.8%-1.0%  0.457-0.465
             MeanFlow, 1  0.007 0.002 0.004 0.002 0.006 0.002  4.0%-4.4%  0.314-0.339
             MeanFlow, 2  0.003 0.003 0.007 0.009 0.008 0.007  1.3%-1.7%  0.364-0.379
             MeanFlow, 4  0.005 0.005 0.007 0.010 0.023 0.007  0.7%-0.9%  0.360-0.376
```



1 回の評価後、どの生徒のどの実行でも、サンプルは誤差 0.007 以内で 0.5/0.3/0.2 の重みをもつ 3 つのモード上に配置される。新たに抽出した 20,000 点でも 0.004〜0.005 のずれが生じるため、1 回のジャンプでサンプリングノイズと同程度の精度まで重みを再現できており、蒸留した 2 時刻の生徒は 2 回および 4 回のジャンプでもその精度を維持する。新規抽出データの逸脱率は各ガウス分布の標準偏差の 3 倍を超える裾に相当する約 1% であり、50 ステップの教師では約 1.5% である。蒸留した 2 種類の生徒はいずれも 1 回の評価後の逸脱率が約 2% で、広がりも 0.45〜0.46（データは 0.47）とデータの広がりを保っている。2 回目の評価を行うと、再ノイズ化する場合（1.2%〜1.5%）でも再ジャンプする場合（1.0%〜1.1%）でも、逸脱率は教師と同等以下まで下がる。この混合分布では、再ノイズ化による悪化はこれらの数値からは見られない。2 回目のジャンプが保持しているのは、各サンプルと初期ノイズとの対応関係である。一方、MeanFlow は 1 回ジャンプ後の逸脱率が約 4% であり、図のパネルからはもう 1 つの欠陥も見て取れる。モードが狭すぎるのである。広がりは 1 回ジャンプ後で 0.31〜0.34、2 回および 4 回ジャンプ後でも 0.35〜0.40 にとどまる。4 回ジャンプ時の逸脱率がデータ自体と同程度に低いのも、このモード収縮が理由である。

6 回の実行を比較すると、1 回だけの実行では見誤る点も浮かび上がる。蒸留された生徒の重み誤差は、すべての実行において 2 回および 4 回のジャンプで 0.006 以内に収まる。しかし MeanFlow はそうではない。4 回ジャンプ後には 6 回中 1〜2 回の実行で誤差が 0.013〜0.023（新規抽出のず

れの 2〜6 倍）に達する一方、残りの実行では 0.010 以内に収まる。このように、複数回ジャンプ後の MeanFlow の重みは訓練実行ごとにばらつくため、1 回の実行結果だけでは判断できないが、蒸留された生徒にはそのような依存性がない。

このタブで重みがずれているのはシード 6（表の 5 列目）の実行であり、4 回ジャンプ後に 0.023 ずれている。

この比較から欠陥の原因が特定できる。2 時刻の 2 つの生徒はネットワーク、時刻のペア、更新回数をすべて共有しており、ターゲット内部の速度だけが異なる。それにもかかわらず、両者の広がりの範囲は一度も重ならない。すべての蒸留実行はすべての MeanFlow 実行よりも、1 回ジャンプ後で 0.1 以上、2 回および 4 回ジャンプ後でも少なくとも 0.06 上回っているのに対し、同一手法の 6 回の実行間の幅は高々 0.013（蒸留）および約 0.04（MeanFlow）にすぎない。したがってこの実験において、モードの収縮は訓練例の速度で訓練したことに起因している。ノイジーなターゲットで訓練した結果、ノイジーな写像ではなく狭すぎる写像に行き着く理由までは、この実験からは断定できない。Equation 17.61 の補正項を一部しか学習できなかった生徒は部分的にオイラーステップとして振る舞い、オイラー法のストライドはまさにこのようにモードを収縮させるが（17.5 節）、これはもっともらしい解釈であって証明された結論ではない。少なくともこの混合分布においては教師を用意する価値がある。データのみから訓練すれば教師の訓練は省けるが、生徒ははるかにノイジーなターゲットを平均化しなければならなくなる。

17.8.3.2 Fashion-MNIST

画像の教師モデルは、無条件の速度 U-Net である `UNet(ch=64)`（パラメータ数は約 120 万個）であり、生徒がそのコピーから出発するため、低周波の時間特徴量を用いて 10,000 回更新して訓練する。`train_regression` は訓練後のネットワークとともに重みの移動平均を返し、その平均化されたネットワークを教師とする。

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()

def batch_fn():
    return X[torch.randint(0, len(X), (128,), device=X.device)]

torch.manual_seed(0)
_, teacher, _ = d2l.train_regression(  # the student copies these time features
    d2l.UNet(ch=64, time_freqs=slow).to(X.device),
    d2l.regression_loss(lin, 'velocity'), batch_fn, 10_000)
```

1,000 枚の生成サンプルを、17.4 節のフレシェ距離（Fréchet distance、FD）を用いて 1,000 枚のテスト画像と比較しスコア化する。これは Fashion-MNIST 分類器が両画像集合から抽出した特徴量に当てはめたガウス分布間の距離であり、互いに素な 1,000 枚ずつのテスト画像 2 組の間で計算した下限を基準に読み取る。

```
cnn = d2l.fashion_feature_cnn(X, y)
features = lambda imgs: d2l.fd_features(cnn, imgs)
f_test = features(X_test[:1000])
floor = d2l.fid_score(features(X_test[1000:2000]), f_test)
print(f'floor, 1,000 real images against 1,000 others: FD {floor:.2f}')
```

```
floor, 1,000 real images against 1,000 others: FD 0.81
```

生徒は先ほど定義した `UNetFlowMap` に教師のコピーを渡したものであり、教師の速度を用い、2 つの時刻をともに自由に動かして訓練する。時刻ペアの半分では $s = t$ とし、ターゲットを教師の

速度そのものにする。生徒は教師から出発するため、ジャンプを学習するあいだに瞬間速度を忘れないようにつなぎ留める狙いがある。残りの半分では混合分布のときと同様に $s < t$ となるよう時刻を引く。訓練済みの重みから出発し、学習率 $2 \cdot 10^{-4}$ で 5,000 回の更新（1 回あたり 128 枚、教師の 10,000 回の半分）にわたって学習させる。結果がどの程度実行ごとのばらつきによるものかを確かめるため、シード 2 と 3 を用いて同じ教師から生徒を 2 つ訓練する。

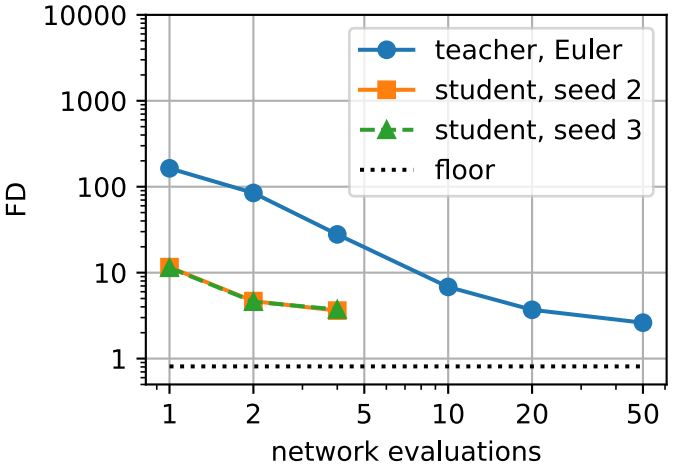
```
def student_loss(u, x0):
    a, b = torch.rand(2, len(x0), device=x0.device)
    s, t = torch.minimum(a, b), torch.maximum(a, b)
    # half the pairs have s = t: the teacher's velocity as target
    s = torch.where(torch.rand(len(x0), device=x0.device) < 0.5, t, s)
    return d2l.flow_map_error(u, teacher, x0, torch.randn_like(x0), s,
                              t).mean()

image_students = {}
for seed in (2, 3):
    torch.manual_seed(seed)
    _, image_students[seed], _ = d2l.train_regression(
        d2l.UNetFlowMap(copy.deepcopy(teacher)).to(X.device), student_loss,
        batch_fn, 5_000, lr=2e-4, ema_half_life=500,
        compiled=False)  # jvp through GroupNorm won't compile
```

同じ 1,000 枚のノイズ画像から、教師は 1〜50 オイラーステップ、各生徒は 1、2、4 回のジャンプを行い、ネットワーク評価回数に対して FD をプロットする。教師の 1 ステップは平均画像に着地するはずであり、これは混合分布での失敗のピクセル版にあたるため、そこでの FD は他のすべての点を圧倒的に上回るはずである。

```
torch.manual_seed(3)
noise = torch.randn(1000, 1, 28, 28, device=X.device)
velocity = lambda x, s, t: teacher(x, t)
ns_teacher, ns_student = (1, 2, 4, 10, 20, 50), (1, 2, 4)
images = {('teacher', n): d2l.flow_map_sample(velocity, noise, n)
          for n in ns_teacher}
images |= {(f'student {seed}', n): d2l.flow_map_sample(u, noise, n)
           for seed, u in image_students.items() for n in ns_student}
fd = {key: d2l.fid_score(features(x), f_test) for key, x in images.items()}
print('FD at evaluations' + ''.join(f'{n:8d}' for n in ns_teacher))
for name in ['teacher', *(f'student {seed}' for seed in image_students)]:
    print(f'{name:17}' + ''.join(f'{fd[name, n]:8.2f}' for n in ns_teacher
                                  if (name, n) in fd))
d2l.plot([ns_teacher, ns_student, ns_student, [1, 50]],
         [[fd['teacher', n] for n in ns_teacher],
          *[[fd[f'student {seed}', n] for n in ns_student]
            for seed in image_students], [floor, floor]],
         'network evaluations', 'FD', ylim=(0.5, 1e4), xscale='log',
         yscale='log', fmts=('o-', 's-', '^--', 'k:'),
         legend=['teacher, Euler', 'student, seed 2', 'student, seed 3',
                 'floor'], legend_loc='upper right')
```

```
FD at evaluations       1       2       4      10      20      50
teacher            163.74   84.70   27.87    6.82    3.70    2.62
student 2           11.58    4.66    3.62
```

```
student 3           11.40    4.59    3.73
```

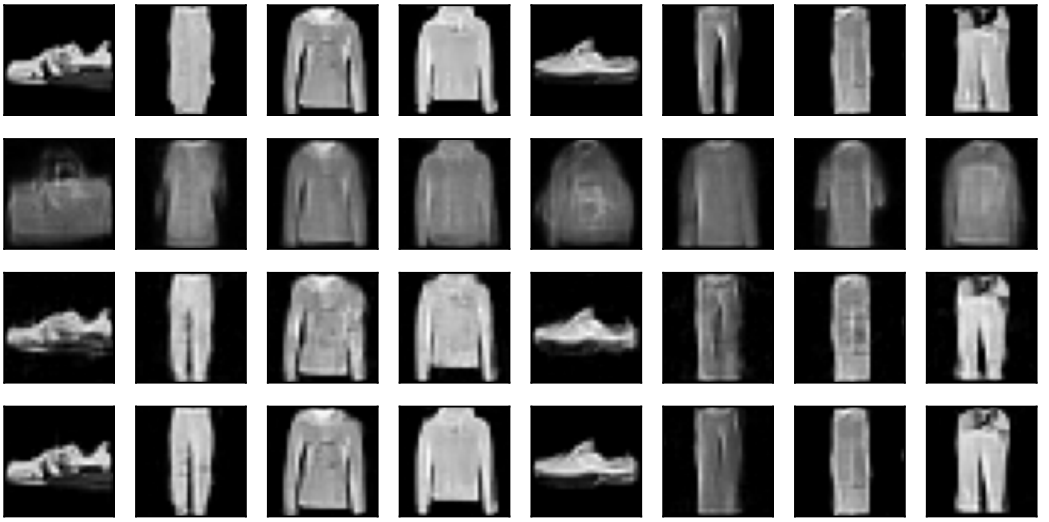


教師の FD は、1 オイラーステップでの約 145〜170、2 ステップでの約 80 から、10 ステップで約 7、50 ステップで約 2.6 まで低下する（下限は約 0.8）。少ない評価予算では、生徒はこの曲線のずっと下に位置する。1 回ジャンプのスコアは 11〜12（教師の 50 ステップ FD の 4〜5 倍）、2 回ジャンプはほぼ 2 倍、4 回ジャンプは 1.4〜1.5 倍となり、教師が 20 ステップで到達する水準に迫る。プロットを横方向に読むと、生徒の 2 回ジャンプは教師の 10〜20 オイラーステップの間に、1 回ジャンプは 4〜10 ステップの間に相当する。2 つの生徒はどの評価予算でも FD の 10 分の 1 以内で一致しているため、これらの結果はどちらの生徒にも当てはまる。

蒸留された生徒にできるのは最良でも教師の ODE を再現することまでであるため、その到達目標は下限ではなく教師の厳密な ODE がもつ FD（50 オイラーステップが上から近づく値）である。残る差のうちどれだけが訓練量に起因するのかは、今回の実行からはわからない。各生徒が見た画像は延べ 640,000 枚であるが、公開されている論文の生徒モデルははるかに多くの画像で訓練されており、1 ステップで教師にずっと近い性能に達している (Song et al. 2023; Lu and Song 2025)。

下の各行は同じ 8 枚のノイズ画像から生成したものであり、上から 50 オイラーステップの教師、2 オイラーステップの教師、そしてシード 2 の生徒による 1 回ジャンプと 2 回ジャンプである。

```
rows = [('teacher', 50), ('teacher', 2), ('student 2', 1), ('student 2', 2)]
d2l.show_gray(torch.cat([images[r][:8].clamp(-1, 1) for r in rows]), 4, 8);
```



各行は列ごとに対応している。教師の 2 オイラーステップは衣類の灰色の平均像を描き出すにとどまり（混合分布での失敗のピクセル版）、50 ステップが描く衣類の種類にすらなっていない列もある。これに対し生徒は、同じノイズからわずか 1 回の評価で、ほぼすべての列において教師と同じ衣類を描き出す。1 回のジャンプで失われるのは細部であり、柄や縞模様がにじみ、輪郭がところどころギザギザになり、わずかな染みが現れる。2 回目のジャンプを行うと、ほとんどの列が教師の 50 ステップに近い仕上がりになる。

17.8.4 分布マッチング

フロー写像の性能は、学習元となる軌跡によって上限が決まる。蒸留された生徒は教師の ODE によって制限され、データのみから訓練された生徒は厳密な速度場をどれだけうまく学習できるかによって制限される（混合分布では広がりの再現が不十分であった）。

もう 1 つの路線は、軌跡ではなく分布に一致させるように 1 ステップ生成器 $g_\theta(\mathbf{z})$ を訓練する方法である。このような生成器を二乗誤差で教師のサンプルに当てはめると、平均ぼけした画像を描いてしまう。教師の長い計算を 1 回の順伝播で再現できないネットワークにとっては複数のノイズベクトルが区別しづらく見え、二乗誤差のもとではそれらの画像の平均が最適になるからである（16.7 節）。同節では 2 つの解決策も示した。実画像に対するクリティックを用いる方法 (Sauer, Lorenz, et al. 2023) と、クリティックを必要としない分布マッチング蒸留（distribution matching distillation）である (Yin, Gharbi, Zhang, et al. 2024)。後者は生成器のサンプルにノイズを加えて $\mathbf{x}_t = (1-t)\, g_\theta(\mathbf{z}) + t\,\epsilon$ とし、その分布 $q_{\theta,t}$ とノイズ付きデータ分布 p_t とのカルバック・ライブラー・ダイバージェンスを下げる。その勾配は次のようになる。

$$\begin{aligned}&\nabla_\theta D_{\mathrm{KL}}(q_{\theta,t} \,\|\, p_t) \\ &\qquad = \mathbb{E}\Big[\big(\nabla \log q_{\theta,t}(\mathbf{x}_t) - \nabla \log p_t(\mathbf{x}_t)\big)^\top \frac{\partial \mathbf{x}_t}{\partial \theta}\Big].\end{aligned} \tag{17.63}$$

この勾配は生徒のスコアから教師のスコアを引いたものであるため、降下ステップはノイズを加えた各サンプルを、生徒が割り当てている以上にデータの確率質量が多い領域へと動かす。線形経路上ではスコアが速度のアフィン関数になるため教師が $\nabla \log p_t$ を供給でき（17.2 節）、生徒自身のサンプルを用いて並行して訓練する第 2 の拡散モデルが $\nabla \log q_{\theta,t}$ を供給する（Figure 17.15）。このような生徒の上限は教師の軌跡ではなく教師の分布によって決まる。敵対的蒸留におけるク

リティックや、分布マッチングにクリティックを追加する手法 (Yin, Gharbi, Park, et al. 2024) のように実画像を損失に含めた場合に限り、生徒は教師を上回ることができる（16.8 節）。どちらの変種も ODE の軌跡に沿って訓練されるわけではないため、あるノイズベクトルに対する生成器のサンプルは教師のサンプルと一致するとは限らず、ODE を介した反転や編集はもはや適用できない。

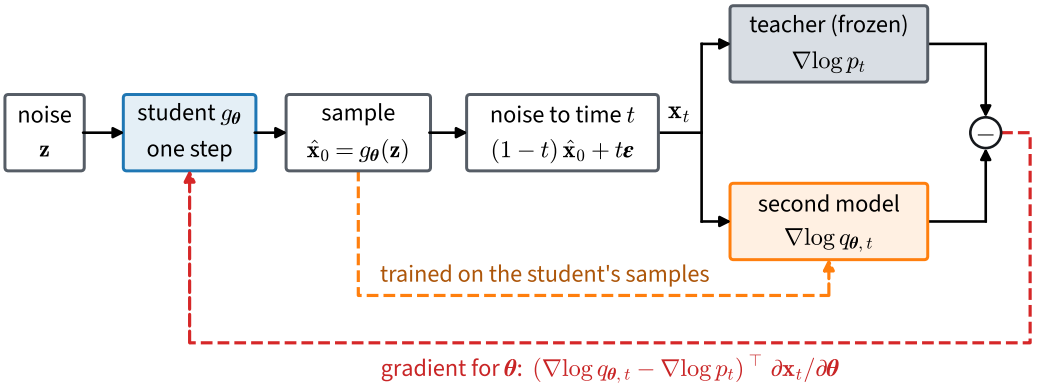


Figure 17.15: 分布マッチング蒸留。生徒はノイズ $\mathbf{z}$ を 1 ステップでサンプルへと写し、そのサンプルに時刻 t までノイズを加える。凍結された教師はノイズ付きデータのスコア $\nabla \log p_t$ を $\mathbf{x}_t$ において供給する。生徒のサンプル上で並行して訓練される第 2 の拡散モデル（オレンジ）は、ノイズを加えた生成サンプルのスコア $\nabla \log q_{\theta,t}$ を供給する。2 つのスコアの差が生徒に伝わる勾配となる（赤）。

どの手法を選ぶべきかは計算予算から決まる。50 回の評価が可能であれば、優れたソルバと組み合わせた教師モデルで十分であり、2 回目の訓練は不要である。レイテンシが重要な場合には、教師から 2 時刻のフロー写像を蒸留するとよい。Fashion-MNIST では 2 回のジャンプで教師の 10〜20 オイラーステップに相当する品質が得られた。教師がない場合、MeanFlow はデータのみからフロー写像を訓練できるが、混合分布において本節で用いた単純な二乗誤差ではモードが狭くなりすぎた（公開されたレシピでは損失に重み付けを導入している。演習問題 4）。クリティックの有無にかかわらず分布マッチングも 1 ステップ生成を実現するが、反転や編集に必要なノイズからサンプルへの対応写像は失われる。

いずれの路線も同じ対象、すなわち速度やスコアが定義される連続的な経路から出発している。しかし、$\mathbf{x}$ がトークンの系列である場合にはそのどちらも存在しない。これが 17.9 節の主題である。

17.8.5 まとめ

確率フロー ODE を $t = 1$ から $t = 0$ まで 1 回のオイラーステップで進めると、曲がった軌跡の初期接線に沿って進むため、すべての出発点がデータの平均 $\mathbb{E}[\mathbf{x}_0]$ へと潰れてしまう。これに対しフロー写像 $\Phi_{t\to s}(\mathbf{x}_t) = \mathbf{x}_t - (t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t)$ は、平均速度 $\bar{\mathbf{u}}$ を介して ODE 軌跡に沿った積分変位を予測するものであり、これを t で微分するとオイラー型恒等式すなわち MeanFlow 恒等式 $\bar{\mathbf{u}} = \mathbf{u}_t - (t-s)\,\frac{d}{dt}\bar{\mathbf{u}}$ が得られ、1 回のフォワードモードによるヤコビ行列・ベクトル積と勾配停止ターゲットを用いて訓練できる。参照速度 $\mathbf{u}_{\text{ref}}$ の選択（事前学習済みの教師 $\hat{\mathbf{u}}$ か条件付き速度 $\boldsymbol{\epsilon} - \mathbf{x}_0$ か）と時間領域の選択（$s = 0$ か $s \le t$ か）を組み合わせることで、一貫性蒸留、一貫性訓練、フロー写像蒸留、および MeanFlow が統一的に理解できる。3 モード混合分布および Fashion-MNIST において、教師からのフロー写像蒸留はモード内の広がりを保ちつつ教師の

10〜20 オイラーステップに相当する品質を生徒の 2〜4 回のジャンプで達成する一方、ノイジーな訓練例ごとの速度に対して単純な二乗誤差で訓練した教師なし MeanFlow は混合分布のモードを収縮させる。

17.8.6 演習問題

1. **1 ステップと 2 ステップのオイラー法**。
   1. 純粋なノイズから $t = 0$ への決定論的サンプラの 1 ステップは、経路によらずどの出発点からも $\mathbb{E}[\mathbf{x}_0]$ を返す（17.3 節）。t に関する 1 回のオイラーステップ $\mathbf{x} - \mathbf{u}_1(\mathbf{x})$ をコサイン経路 $\alpha_t = \cos\frac{\pi t}{2}$、$\sigma_t = \sin\frac{\pi t}{2}$ 上でとった場合にも同じことが成り立つか。このステップは何を返すか。また、t に関するオイラーステップが経路に依存する一方で、$\bar{\sigma}$ に関するステップが経路に依存しないのはなぜか。
   2. 線形経路上の 1 次元データを、等確率の 2 点 $\pm a$ とする。$\mathbb{E}[x_0 \mid x_t = x] = a\tanh\left(a(1-t)x/t^2\right)$ となること、および長さ $\frac{1}{2}$ の 2 回のオイラーステップが初期点 x を $a\tanh(ax)$ へ写すことを示せ。この例の厳密なフローは 38.7.1.4 節にある。
   3. $a = 2$ のとき、2 ステップのサンプルのうちデータ点から距離 0.1 以内に入る割合はいくらか。また、厳密なフロー写像 $\Phi_{1\to 0}$ は x をどこへ送るか。
2. [code] **混合分布における Reflow**。●Reflow (Liu et al. 2023) は、ノイズベクトルと現在のモデルがそこから生成するサンプルのペアを用いて速度モデルを再訓練する。
   1. 20,000 個のノイズベクトルを引き、本節の教師を 50 オイラーステップで積分し、得られたペア上で速度損失と教師と同じ設定を用いて新しい `TimeMLP` を訓練せよ。
   2. 新しいモデルの 1、2、4、10 オイラーステップにおけるモード重みと逸脱率、および 50 ステップの軌跡とその弦との平均距離を、教師の同じ量と並べて報告せよ。
   3. Reflow は教師の誤差のうちどれを取り除き、どれを引き継ぎ、どれを新たに付け加えるか。2 ラウンド目を行うとそれぞれどうなるか。
3. **ラグランジュ型損失**。●Boffi et al. (2025) はラグランジュ型恒等式を用いてフロー写像を訓練している。
   1. Equation 17.57 を $\partial_s \Phi_{t\to s}(\mathbf{x}) = \mathbf{u}_s(\Phi_{t\to s}(\mathbf{x}))$ に代入し、ジャンプの出力点における速度を含む $\bar{\mathbf{u}}$ の恒等式を導け。
   2. この恒等式ではなぜ条件付き速度 $\epsilon - \mathbf{x}_0$ を用いることができず、速度モデルが必要になるのか。また、フロー写像自身の速度 $\bar{\mathbf{u}}(\cdot, s, s)$ は利用できるため、単一のネットワークで自己訓練できることを示せ。
   3. [code] このような生徒を混合分布上で訓練し、逸脱率と 1 ステップあたりの訓練コストについて MeanFlow の生徒と比較せよ。
4. [code] **勾配停止と重み付け**。 Equation 17.62 の最小点が何になるかは、2 つの細部によって決まる。
   1. `flow_map_error` をセルにコピーし、ターゲットに対する勾配停止を取り除け。この損失で混合分布上に MeanFlow の生徒を訓練し、1 ステップでのモード重み、逸脱率、広がりを報告せよ。ターゲット側も微分した場合の損失の勾配を書き下し、勾配停止が取り除いている項を特定せよ。ネットワークは Equation 17.61 を満たさずにこの損失を下げるため、何を変えることができるか。
   2. Song らは代わりに生徒の重みの移動平均を用いてターゲットを計算している。これを `d2l.EMA` で実装し、比較せよ。
   3. MeanFlow の ImageNet レシピ (Geng et al. 2025) は、各訓練例の二乗誤差 e を固定した $e + c$（c は小さい定数）で割ることで、勾配が $\log(e + c)$ の勾配と一致するようにしている。二乗誤差はターゲットの条件付き平均によって最小化されるが、重み付き損失は最小点をどの値の近くに置くか。教師がない場合、混合分布のノイジーな点におけるターゲットは 3 つのモードに広がる。1 回ジャンプのサンプルのモード重みを予測したうえで、この重み付けと $c = 10^{-3}$ を用いて MeanFlow の生徒を訓練し、結果を確認せよ。

4. 28×28 の画像では、e は 784 個の二乗誤差の和となる。このとき同じ (s, t) における訓練例間のそのばらつきは 2 次元の場合よりはるかに小さくなり、重み付けは主として時刻間の重み付けとして働くことを論ぜよ。混合分布での実験から画像向けレシピについて何がわかり、何がわからないか。

5. [code] **ガイダンス付き生徒**。●17.4.4.2 節の分類器なしガイダンスは、1 ステップごとにネットワークを 2 回評価する。Meng et al. (2023) はガイダンス付き速度を 1 つのネットワークへと蒸留し、ステップ数を減らす前の段階ですべてのステップのコストを半減させている。本節の画像教師と同様に 17.4 節のクラス条件付き U-Net を `time_freqs=slow` で訓練し、参照速度 $\mathbf{u}_{\text{ref}}$ をガイダンス付き速度 $\tilde{\mathbf{u}}$ すなわち `d2l.cfg(teacher, c, 2.0)`（c はランダムなラベル）として、ラベルも入力に受け取る 2 時刻の生徒へと蒸留せよ。
   1. 50 回のガイダンス付きオイラーステップを用いた教師と、1、2、4 回のジャンプを用いた生徒について、FD および予測クラスと指定クラスの一致率を比較せよ。
   2. 各場合において、サンプル 1 枚あたり何回ネットワークを評価するか。
   3. Meng らはガイダンス強度も生徒への入力として与えている。訓練中にガイダンス強度をどのようにサンプリングすべきか。また、$\gamma = 2$ に固定して蒸留した生徒にはないどのような利点がその生徒から得られるか。

6. ***ノイズ付き分布のマッチング***。●分布マッチング蒸留 (Yin, Gharbi, Zhang, et al. 2024) は、1 ステップ生成器 $\mathbf{x} = g_\theta(\mathbf{z})$ を勾配 Equation 17.63 に沿って訓練する。ここで $q_{\theta,t}$ はノイズを加えた生成サンプル $\mathbf{x}_t$ の分布である。
   1. Equation 17.63 を導出せよ。$q_{\theta,t}$ 自身の θ への依存から生じる項が消えるのはなぜか。
   2. 1 次元において、生成器を $m + bz$（ただし $b > 0$）、データを $\mathcal{N}(\mu, v)$ とする。線形経路上の $D_{\text{KL}}(q_t \,\|\, p_t)$ を閉じた式で表し、$m = \mu$、$b^2 = v$ が任意の $t < 1$ における唯一の最小点であることを示せ。
   3. データを $\frac{1}{2}\mathcal{N}(-2, 0.25) + \frac{1}{2}\mathcal{N}(2, 0.25)$ とする。$D_{\text{KL}}(q_t \,\|\, p_t)$ を (m, b) のグリッド上で数値積分により $t \in \{0.1, 0.3, 0.5, 0.9\}$ のそれぞれについて評価し、各 t における最小点を求めよ。最小値が $\log 2$ 付近にとどまる（小さい t において）のはなぜか。この結果を GAN におけるモード欠落（16.1.3.3 節）およびノイズ下でのモードの合流（演習問題 4、38.5 節）と関連付けよ。大きい t に向けてノイズレベルに重みを付けるとガウス生成器にとって何が得られ、どのような重み付けによっても得られないものは何か。

17.9 離散拡散

前節までのモデルはすべて、$\mathbb{R}^2$ 上の点、ピクセル強度、あるいはオートエンコーダの潜在表現といった連続値ベクトルにガウスノイズを加えることに依拠していた。しかし、テキストトークン、ソースコード、量子化された画像トークンのような離散系列には、微小な実数を加算する摂動を与えられない。また、自己回帰モデル（11.2 節）が左から右へ L 回の逐次的な順伝播によって L 個のトークンを生成するのに対し、離散拡散モデル（discrete diffusion model）は双方向文脈（bidirectional context）を用い、1 回のネットワーク評価あたり複数のトークンを並列に生成する。

有限語彙上の拡散を定義するため、Hoogeboom et al. (2021) と Austin et al. (2021) はガウス摂動を、遷移行列 $\mathbf{Q}_t$ に従う離散状態マルコフ連鎖に置き換えた。テキストや離散符号において最も有効な破損過程は**吸収（マスキング）カーネル**（absorbing kernel）であり、各トークンを独立にマスク記号 $[\text{M}]$ へ置き換えていき、最終的に系列全体を消去する。この吸収連鎖のもとでは、17.6 節の変分下界（variational lower bound）は重み付きマスク交差エントロピー目的関数へと簡約される (Sahoo et al. 2024; Shi et al. 2024)。本節では、吸収マルコフ連鎖とその変分上界を導出し、3 モード混合分布を量子化した 2 トークン版および 2 値化 Fashion-MNIST 画像において複数トークンを並列にマスク解除する際に生じる独立性誤差（independence error）を解析したうえで、マスク拡散をマスク言語モデルおよび自己回帰モデル（11.4 節、11.2 節）と結びつけ

る。

```
from scipy.special import ndtr, xlogy
```

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.9.1 トークンの破損

トークンの順過程（forward process）に求める条件は、17.2 節のガウス連鎖が備えていたものと同じである。すなわち、周辺分布と事後分布が閉形式で得られ、終端状態がデータに関する情報を一切持たないことである。

17.9.1.1 遷移行列

まず記法を変更する。離散拡散の文献に合わせ、$t = 1, \dots, T$ はステップ数を数える整数とする。これはガウス連鎖において k が担っていた役割である。ネットワークへの入力は引き続き、本章の時計における時刻 t/T とする。

トークンは K 通りの値のいずれかをとるものとし、ワンホット行ベクトル $\mathbf{x} \in \{0,1\}^K$ で表す。トークンに対するノイズとはその値をランダムに変化させることであり、各行の和が 1 となる**遷移行列**（transition matrix）$\mathbf{Q}_t \in [0,1]^{K \times K}$ で与えられる。ここで $[\mathbf{Q}_t]_{ij}$ は、値 i から値 j へステップ t で遷移する確率である。これはマルコフ連鎖モンテカルロ法（36.8.3 節）における行確率行列であり、分布は行ベクトルとして右から $\mathbf{Q}_t$ を掛けることで更新される。したがって、次式が成り立つ。

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathrm{Cat}(\mathbf{x}_t;\, \mathbf{x}_{t-1}\mathbf{Q}_t), \tag{17.64}$$

ここで、$\mathrm{Cat}(\mathbf{x}; \mathbf{p})$ は確率ベクトル $\mathbf{p}$ を持つカテゴリカル分布であり、この場合は現在の値によって選ばれる $\mathbf{Q}_t$ の行がその確率ベクトルとなる。長さ L のトークン系列は、すべての位置で独立にこの連鎖を走らせることで破損させる。2 つの遷移の合成はそれぞれの行列の積になるため、t に関する帰納法により、t ステップ後の周辺分布は次のようになる。

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathrm{Cat}(\mathbf{x}_t;\, \mathbf{x}_0\bar{\mathbf{Q}}_t), \qquad \bar{\mathbf{Q}}_t = \mathbf{Q}_1\mathbf{Q}_2 \cdots \mathbf{Q}_t. \tag{17.65}$$

両端が与えられたときの 1 ステップの事後分布は、ベイズの定理とマルコフ性（Markov property）から次のように導かれる。

$$\begin{aligned} q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) &= \frac{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)} \\ &= \mathrm{Cat}\left(\mathbf{x}_{t-1};\, \frac{\mathbf{x}_t\mathbf{Q}_t^\top \odot \mathbf{x}_0\bar{\mathbf{Q}}_{t-1}}{\mathbf{x}_0\bar{\mathbf{Q}}_t\mathbf{x}_t^\top}\right), \end{aligned} \tag{17.66}$$

ここで、$\odot$ は要素ごとの積である。とり得る各直前状態は、初期状態のもとでの確率と、観測された $\mathbf{x}_t$ へ至る確率の両方で重み付けされる。

各ステップで確率 0.1 で反転する 2 値トークンを考えると、この事後分布の働きがよくわかる。ここでは $\mathbf{Q}_1 = \mathbf{Q}_2 = \left(\begin{smallmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{smallmatrix}\right)$ である。1 番目の値 $\mathbf{x}_0 = (1, 0)$ から出発し、2 ステップ後に 2 番

目の値 $\mathbf{x}_2 = (0, 1)$ が観測されたとしよう。これが起こる確率は $0.9 \cdot 0.1 + 0.1 \cdot 0.9 = 0.18$ であり、これが分母となる。中間の状態については、$\mathbf{x}_2 \mathbf{Q}_2^\top = (0.1, 0.9)$ かつ $\mathbf{x}_0 \bar{\mathbf{Q}}_1 = (0.9, 0.1)$ である。これらの要素ごとの積は $(0.09, 0.09)$ となり、0.18 で割ると $(1/2, 1/2)$ が得られる。トークンは 1 度反転しており、両端が与えられたもとでは、その反転が 1 ステップ目で起きた確率と 2 ステップ目で起きた確率は等しい。

17.9.1.2 吸収カーネル

変分上界には $\bar{\mathbf{Q}}_t$ の閉形式が必要であり、サンプラには既知の初期状態が必要である。Austin et al. (2021) の**吸収カーネル**はその両方を与える。このカーネルは語彙に**マスク** $[\mathrm{M}]$ という 1 つの値を追加し、各トークンを確率 β_t でステップ t においてそこへ遷移させる。一度マスクされたトークンはマスクされたままである。

$$\mathbf{Q}_t = (1 - \beta_t)\, \mathbf{I} + \beta_t\, \mathbf{1}\mathbf{e}_{\mathrm{M}}^\top, \tag{17.67}$$

ここで、$\mathbf{1}$ は全要素が 1 の列ベクトル、$\mathbf{e}_{\mathrm{M}}$ はマスクに対応するワンホット列ベクトルであり、$\mathbf{1}\mathbf{e}_{\mathrm{M}}^\top$ のすべての行は確率質量をすべてマスクに置く。

このような行列の積は簡潔にまとまる。$\mathbf{P} = \mathbf{1}\mathbf{e}_{\mathrm{M}}^\top$ と書こう。$\mathbf{e}_{\mathrm{M}}^\top \mathbf{1} = 1$ より $\mathbf{P}^2 = \mathbf{P}$ が成り立ち、したがって $(a\mathbf{I} + b\mathbf{P})(c\mathbf{I} + d\mathbf{P}) = ac\,\mathbf{I} + (ad + bc + bd)\,\mathbf{P}$ となる。$a + b = c + d = 1$ のとき第 2 項の係数は $1 - ac$ に等しいため、$\mathbf{I}$ の係数は連鎖に沿ってそのまま掛け合わされる。そこで、$\bar{\alpha}_t = \prod_{s \le t} (1 - \beta_s)$ とおく。これは 17.2 節のガウス連鎖と同じ積である。ガウス連鎖ではクリーンな点の重みの二乗であったが、ここではより直接的な意味、すなわちトークンが t ステップの間マスクされずに残る生存確率を表す。すると、次式が得られる。

$$\bar{\mathbf{Q}}_t = \bar{\alpha}_t\, \mathbf{I} + (1 - \bar{\alpha}_t)\, \mathbf{1}\mathbf{e}_{\mathrm{M}}^\top. \tag{17.68}$$

この連鎖はあるトークンを別のトークンと取り違えることは決してなく、ただ消去するだけである。$\bar{\alpha}_T = 0$ となるスケジュールは、ガウス連鎖が純粋なノイズに到達するのと同様に、データに関する情報を一切持たない全マスク系列に到達する（Figure 17.16）。もう 1 つのよく使われる選択肢である**一様カーネル**（uniform kernel）は、語彙から一様にトークンを再サンプリングする (Hoogeboom et al. 2021)（演習問題 1）。

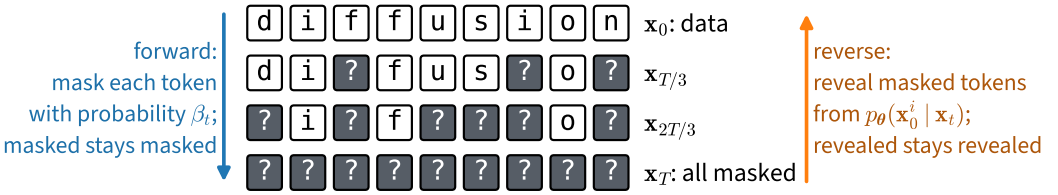


Figure 17.16: 9 個のトークンに対する吸収連鎖。順方向では、各トークンはステップごとに確率 β_t で独立にマスク（「?」として図示）へ置き換えられ、その後もマスクされたままとなり、T ステップ後にはすべてのトークンがマスクされる。逆方向では、モデルが部分的にマスクされた系列からマスクトークンを予測し、一度マスク解除されたトークンは決して変更されない。

以降は吸収カーネルを用いる。Austin らの比較実験においてテキスト上で最良の尤度を達成したうえ、事後分布がこれ以上ないほど単純になるからである。一方で失うものもある。逆方向では連鎖はマスクを解くことしかしないため、左から右へ生成するモデルと同様に、サンプラは一度

書き込んだすべてのトークンをそのまま確定させることになる。一様カーネルのようにトークンを再サンプリングする連鎖や、ReMDM (G. Wang et al. 2025) のように一度マスク解除したトークンを再びマスクするサンプラであれば、早い段階での誤りを修正できる（17.10 節）。

17.9.2 マスク拡散の変分上界

逆連鎖を訓練するには損失が必要であり、17.6 節の変分上界がその損失を与える。

17.9.2.1 同じ上界、異なるターゲット

逆モデルは全マスク系列から出発するマルコフ連鎖 $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ である。17.6 節では、このような連鎖の負の対数尤度を次の式で上から抑えた。

$$
\begin{aligned}
-\log p_\theta(\mathbf{x}_0) \le L_{\text{VLB}} &= D_{\text{KL}}\big(q(\mathbf{x}_T \mid \mathbf{x}_0) \,\|\, p(\mathbf{x}_T)\big) \\
&\quad + \sum_{t=2}^{T} \mathbb{E}_q\Big[D_{\text{KL}}\big(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)\big)\Big] \\
&\quad + \mathbb{E}_q\big[-\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)\big].
\end{aligned}
$$

これは順連鎖を変分分布としたときの 36.7 節のエビデンス下界（evidence lower bound）に負号をつけたものである。その導出はマルコフ性とベイズの定理しか使っていないため、トークンに対してもそのまま成り立つ。

変わるのはターゲットである。ガウスノイズの場合、ネットワークはノイズ、速度、またはクリーンな点を予測しており、これらはいずれも 1 つの事後平均（posterior mean）$\mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t]$ のアフィン関数であった（17.1 節）。ワンホットトークンに対するこの事後平均は、クリーンなトークンがとり得る各値の確率ベクトルになるため、その分布を予測することが本章の出発点であったノイズ除去器（denoiser）の離散版にあたる。Austin et al. (2021) に従い、ネットワークは破損した系列全体が与えられたもとで、各位置 i についてクリーンな値の上の分布 $p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)$ を出力し、逆遷移（reverse transition）はこの予測に関して真の事後分布を平均化する。

$$
p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \sum_{\hat{\mathbf{x}}_0} q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \hat{\mathbf{x}}_0)\, p_\theta(\hat{\mathbf{x}}_0 \mid \mathbf{x}_t), \tag{17.69}
$$

この計算は位置ごとに行い、マスクされていない位置での予測は観測されたトークンそのものとする。単一のトークンであれば、予測が完璧なときこれは真の逆遷移と一致する（演習問題 3a）。しかし系列に対しては 1 つの点で真の逆遷移に及ばない。真の逆遷移が同じステップで元に戻る複数の位置を同時分布からサンプリングするのに対し、この逆モデルはそれらを互いに独立にサンプリングするのである。本節の大部分は、この不足がもたらすコストを扱う。

17.9.2.2 吸収連鎖の事後分布

命題（吸収連鎖の事後分布） $\beta_1 > 0$（**したがって** $\bar{\alpha}_t < 1$ **がすべての** $t \ge 1$ **で成り立つ**）**である吸収カーネルのもとで、クリーンな値** $\mathbf{x}_0 \neq [\mathbf{M}]$ **を持つ 1 つの位置を考える。** $\mathbf{x}_t = \mathbf{x}_0$ **ならば、確率 1 で** $\mathbf{x}_{t-1} = \mathbf{x}_0$ **である。** $\mathbf{x}_t = [\mathbf{M}]$ **ならば、次式が成り立つ。**

$$
\begin{aligned}
q(\mathbf{x}_{t-1} = \mathbf{x}_0 \mid \mathbf{x}_t = [\mathrm{M}], \mathbf{x}_0) &= \frac{\bar{\alpha}_{t-1} - \bar{\alpha}_t}{1 - \bar{\alpha}_t}, \\
q(\mathbf{x}_{t-1} = [\mathrm{M}] \mid \mathbf{x}_t = [\mathrm{M}], \mathbf{x}_0) &= \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t},
\end{aligned}
\tag{17.70}
$$

これ以外の値が正の確率を持つことはない。

証明 Equation 17.66 を Equation 17.67 および Equation 17.68 を用いて評価する。ベクトル $\mathbf{x}_0 \bar{\mathbf{Q}}_{t-1}$ は確率質量 $\bar{\alpha}_{t-1}$ を $\mathbf{x}_0$ に置き、$1 - \bar{\alpha}_{t-1}$ をマスクに置くため、とり得る直前状態はこの 2 つだけである。$\mathbf{x}_t = \mathbf{x}_0$ ならば、マスクがトークンに戻ることは決してないため、ベクトル $\mathbf{x}_t \mathbf{Q}_t^\top$ は成分 $1 - \beta_t$ を $\mathbf{x}_0$ に持ち、成分 0 をマスクに持つので、事後分布は点質量となる。$\mathbf{x}_t = [\mathrm{M}]$ ならば、成分は β_t が $\mathbf{x}_0$ に、1 がマスクに対応するため、正規化前の事後分布は $\bar{\alpha}_{t-1}\beta_t$ を $\mathbf{x}_0$ に、$1 - \bar{\alpha}_{t-1}$ をマスクに置く。それらの和 $1 - \bar{\alpha}_{t-1}(1 - \beta_t) = 1 - \bar{\alpha}_t$ が分母 $q(\mathbf{x}_t = [\mathrm{M}] \mid \mathbf{x}_0)$ であり、$\bar{\alpha}_{t-1}\beta_t = \bar{\alpha}_{t-1} - \bar{\alpha}_t$ となる。■

1 ステップ遡ると、マスクトークンはマスクされたままであるか、クリーンな値に戻るかのいずれかである。したがって逆モデル Equation 17.69 は、$\mathbf{x}_t$ のマスクされていない位置をそのままコピーし、マスクされた各位置を互いに独立に、元に戻る確率で $p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)$ からのサンプルに置き換える。

17.9.2.3 重み付き交差エントロピー

逆モデルは見えているトークンをコピーしてマスクのみを埋めるため、変分上界においてコストを生じ得るのはマスクされた位置だけである。この形で Sahoo et al. (2024) と Shi et al. (2024) によって示された次の結果が、本節の中心となる命題である。すなわち、そのコストは交差エントロピーになる。

命題（マスク拡散の変分上界）$\bar{\alpha}_0 = 1$、$\bar{\alpha}_t < 1$ $(t \geq 1)$、$\bar{\alpha}_T = 0$ を満たす吸収カーネルと、上で述べた逆モデルに対して、次式が成り立つ。

$$
L_{\mathrm{VLB}} = \sum_{t=1}^{T} \frac{\bar{\alpha}_{t-1} - \bar{\alpha}_t}{1 - \bar{\alpha}_t} \, \mathbb{E}_{q(\mathbf{x}_t \mid \mathbf{x}_0)} \Big[\sum_{i \,:\, \mathbf{x}_t^i = [\mathrm{M}]} - \log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t) \Big].
\tag{17.71}
$$

証明事前項（prior term）は 0 になる。$\bar{\alpha}_T = 0$ により $q(\mathbf{x}_T \mid \mathbf{x}_0)$ は全マスク系列上の点質量となり、これは $p(\mathbf{x}_T)$ と一致するからである。また、$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ と $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ はともに位置ごとに分解されるため、各ダイバージェンスは位置ごとのダイバージェンスの和となる。マスクされていない位置では両分布は同じ点質量となるため、ダイバージェンスは 0 である。マスクされた位置では $w_t = (\bar{\alpha}_{t-1} - \bar{\alpha}_t)/(1 - \bar{\alpha}_t)$ とおく。真の事後分布は確率 $1 - w_t$ をマスクに、w_t を $\mathbf{x}_0^i$ に割り当て、モデルは確率 $1 - w_t$ をマスクに、$w_t \, p_\theta(\cdot \mid \mathbf{x}_t)$ を各トークンに割り当てる。したがって、次式が成り立つ。

$$
\begin{aligned}
D_{\mathrm{KL}} &= (1 - w_t) \log \frac{1 - w_t}{1 - w_t} + w_t \log \frac{w_t}{w_t \, p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)} \\
&= -w_t \log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t).
\end{aligned}
$$

以上で $t = 2, \ldots, T$ の項が得られる。再構成項（reconstruction term）は $\mathbb{E}[-\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)]$ である。$w_1 = (1 - \bar{\alpha}_1)/(1 - \bar{\alpha}_1) = 1$ であるから、モデルはステップ 1 でマスクされているす

べての位置を $p_\theta(\cdot \mid \mathbf{x}_1)$ からのサンプルで置き換え、残りをコピーする。ゆえにこの項はマスクされた位置にわたる $-\log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_1)$ の和に等しく、総和の $t=1$ の項と一致する。■

これはマスク言語モデルの損失に 2 つの変更を加えたものである。第 1 に、マスク率が 1 つの値に固定されず、ゼロから全マスクまですべてのレベルにわたって変化する。第 2 に、各レベルが固有の重みを持つ。この上界を $\log 2$ とトークン数で割ると、トークンあたりビット数で表した符号長（code length）となる（37.2 節）。画像実験ではこの単位で値を読み取る。

線形スケジュール（linear schedule）$\bar{\alpha}_t = 1 - t/T$ のもとでは、トークンの割合 t/T がステップ t でマスクされ、重みは $w_t = (1/T)/(t/T) = 1/t$ となる。多くがマスクされた系列での予測誤差は、わずかしかマスクされていない系列での同じ誤差よりも小さく評価される。マスクされる位置の個数は t に比例して増えるため、重み付きマスク位置数の期待値はどのステップでも同じであり、L/T（トークン数 L の場合）となる。ステップ間の違いは、マスクトークンの予測がどれだけ難しいかだけである。

この総和はリーマン和（Riemann sum）でもある。$\ell(\mathbf{x}_0, \mathbf{x})$ を $\mathbf{x}$ のマスク位置における交差エントロピー、すなわち Equation 17.71 の内側の和とし、$T \to \infty$ の極限を、$\bar{\alpha}_t = \bar{\alpha}(t/T)$（本章の時計における減少関数で、時刻 0 がデータ、時刻 1 が全マスク系列に対応する）のもとでとる。いま t はステップ数を表すため、時刻を $\tau = t/T$ と書く。すると $w_t \approx -\bar{\alpha}'(\tau)/(T(1-\bar{\alpha}(\tau)))$ となる。この因子 $1/T$ は 1 ステップの幅 $d\tau$ であるから、変分上界は次の積分へ収束する。

$$\int_0^1 \frac{-\bar{\alpha}'(\tau)}{1-\bar{\alpha}(\tau)}\, \mathbb{E}\big[\ell(\mathbf{x}_0, \mathbf{x}_\tau)\big]\, d\tau.$$

ここでマスク確率 $u = 1 - \bar{\alpha}(\tau)$ へ変数変換しよう（本節における u は速度ではなく確率である）。$du = -\bar{\alpha}'(\tau)\, d\tau$ より、積分は次の形になる。

$$\int_0^1 \frac{1}{u}\, \mathbb{E}_u\big[\ell(\mathbf{x}_0, \mathbf{x})\big]\, du,$$

ここで、$\mathbb{E}_u$ は各トークンを確率 u で独立にマスクする期待値である。時刻ではなくマスク確率を入力として受け取るネットワークにとって、この極限はスケジュールに依存しない。これはガウス拡散の上界が持つスケジュール不変性（17.6 節）の離散版である (Kingma et al. 2021)。

実は、最適なネットワークにはそのどちらも不要である。どの位置をマスクするかは、トークンの中身を一切見ないコイン投げによって決まる。入力によってどの位置がマスクされ、残りの位置にどのトークンがあるかが示された時点で、時刻は隠されたトークンについて何の追加情報ももたらさない。したがって、マスクされた位置における最良の予測、すなわち可視トークンが与えられたもとでのそのトークンの条件付き分布は、時刻に一切依存しない (Ou et al. 2025)（演習問題 4）。

近年の論文では、この極限は別の形で登場する。すなわち、残っている各トークンをレート $-\bar{\alpha}'(\tau)/\bar{\alpha}(\tau)$ でマスクする**連続時間マルコフ連鎖**（continuous-time Markov chain）(Campbell et al. 2022) を考え、1 トークンだけ異なる系列間の確率比を用いて時間反転する定式化である (Lou et al. 2024)。吸収カーネルでは、この確率比は既知の時間関数と本節のネットワークが予測するクリーンな条件付き分布との積になるため (Ou et al. 2025)、どちらの見方でも同じ対象を訓練することになる。

関数 `masked_loss` は、系列ごとに 1 つのステップ t を用い、スケジュール `alpha_bar`（値 $\bar{\alpha}_0, \dots, \bar{\alpha}_T$）とマスクのインデックス `mask_id` を受け取って変分上界を推定する。係数 T により、一様にサンプリングしたステップから全ステップの和の不偏推定量が得られる。総和は最初の軸以降のすべての軸にわたってとるため、トークンのペアでも画像でも同じ関数を共有できる。本章のインタ

ーフェースを統一するためネットワークには引き続き t/T を入力するが、訓練されたペアモデルがこの時刻入力をほとんど使っていないことは演習問題 4 で確認する。

```
def masked_loss(model, x0, t, alpha_bar, mask_id):
    """One-draw estimate of the masked bound per sequence, in nats."""
    T = len(alpha_bar) - 1
    shape = (-1,) + (1,) * (x0.dim() - 1)      # broadcast over positions
    masked = (torch.rand(x0.shape, device=x0.device)
              < (1 - alpha_bar[t]).view(shape))  # with prob. 1 - abar_t
    logits = model(torch.where(masked, mask_id, x0), t / T)
    nll = -torch.log_softmax(logits, -1).gather(-1, x0[..., None])[..., 0]
    w = (alpha_bar[t - 1] - alpha_bar[t]) / (1 - alpha_bar[t])  # 1/t if linear
    return (T * w.view(shape) * nll * masked).flatten(1).sum(1)
```

17.9.3 マスク解除によるサンプリング

訓練済みのネットワークは、マスクされたすべての位置で分布を与える。生成においては、どの位置をいつ埋めるかを決めなければならない。

17.9.3.1 並列マスク解除と独立性誤差

生成は全マスク系列から始まる。ステップ t では、1 回の順伝播によってマスクされたすべての位置で $p_\theta(\cdot \mid \mathbf{x}_t)$ が得られ、各マスク位置は候補トークンをサンプリングして確率 w_t で採択する。17.5 節のガウスサンプラと同様に、T 回ものネットワーク評価は避けたいため、レベル $T = t_0 > t_1 > \cdots > t_S = 0$ を選んでそれらの間をジャンプする。レベルを飛び越えることはカーネルの合成にあたり、上で示した簡約により吸収カーネルの積は再び吸収カーネルとなる。したがって粗視化した連鎖も各レベルで周辺分布 Equation 17.68 を持ち、先ほどの命題がそのまま適用できる。すなわち、関数 unmask は t_k から t_{k+1} へジャンプする際、各マスクトークンを確率 $(\bar\alpha_{t_{k+1}} - \bar\alpha_{t_k})/(1 - \bar\alpha_{t_k})$ でマスク解除する。

マスク解除の判定はネットワークを一切参照しないため、各トークンがいつ現れるかは事前にわかる。マスクされたまま残る確率 $(1-\bar\alpha_{t_{k+1}})/(1-\bar\alpha_{t_k})$ はジャンプに沿って掛け合わされて逐次相殺されるため、レベル t_k でトークンがまだマスクされている確率は順連鎖と同じく $1-\bar\alpha_{t_k}$ である。したがって各トークンは、ジャンプ k $(k = 0, \dots, S-1)$ において確率 $\Delta_k = \bar\alpha_{t_{k+1}} - \bar\alpha_{t_k}$ で、他の位置とは独立にマスク解除される。

```
@torch.inference_mode()
def unmask(model, shape, steps, alpha_bar, mask_id):
    """Reveal tokens from the all-mask state in `steps` jumps."""
    T = len(alpha_bar) - 1
    levels = torch.linspace(T, 0, steps + 1).round().long().tolist()
    z = torch.full(shape, mask_id, device=alpha_bar.device)
    for t, s in zip(levels[:-1], levels[1:]):
        logits = model(z, torch.full(shape[:1], t / T, device=z.device))
        draw = torch.distributions.Categorical(logits=logits).sample()
        p = (alpha_bar[s] - alpha_bar[t]) / (1 - alpha_bar[t])  # reveal prob.
        reveal = (z == mask_id) & (torch.rand(shape, device=z.device) < p)
        z = torch.where(reveal, draw, z)
    return z
```

ネットワークの予測は位置ごとの積になっている。そのため、同じジャンプでマスク解除されるトークンはそれぞれの予測分布から互いに独立にサンプリングされる一方、異なるジャンプでマスク解除されるトークンは、すでにマスク解除されたトークンで条件付けられながら順次サンプリングされる。この独立性にはどれだけのコストがかかるだろうか。コストが生じ得る最小の例である 2 トークンの場合から始めよう。

本章で一貫して用いている例からこの 2 トークンを作ることができる。それは平面上の 3 つのガウス分布の混合分布であり、平均は $(-2.5, -1.5)$、$(2.5, -1.5)$、$(0, 2.5)$、重みは 0.5、0.3、0.2、標準偏差は 0.5 であった（17.1 節）。平面を 32×32 のグリッド（範囲 $[-5, 5]^2$）へ量子化すると、各サンプルは 2 つの軸に沿ったセルのインデックスというトークンのペアになる。ペアの確率テーブル p には、各モードに対応する 3 つの塊（ブロブ）がある。周辺分布 p^1 と p^2 は、第 1 軸に沿っては 3 つの山を持つが、2 つのモードが同じ高さを共有しているため第 2 軸に沿っては 2 つの山しか持たない。周辺分布の積 $p^1 \otimes p^2$ は 2 つのトークンを独立にサンプリングするため、一方の山と他方の山が交差するすべての場所にブロブを置く。その結果 6 つのブロブが生じ、そのうち 3 つはデータが存在しない場所にある。

ここで、マスクされた各位置での予測が、マスクされていないトークンが与えられたもとでの真の条件付き分布と一致する**厳密な**ネットワークを考えよう。線形スケジュールのもとで S 回の等間隔ジャンプを行うと、すべてのジャンプで $\Delta_k = 1/S$ となるため、両方のトークンが同じジャンプでマスク解除される確率は $S \cdot (1/S)^2 = 1/S$ であり、この事象ではペアは $p^1 \otimes p^2$ からサンプリングされる。それ以外の場合は、一方のトークンが周辺分布から、もう一方が後から条件付き分布からサンプリングされるため、ペアは p から得られる。したがってサンプラは次の混合分布からサンプリングする。

$$\left(1 - \tfrac{1}{S}\right) p + \tfrac{1}{S}\, p^1 \otimes p^2.$$

2 つの確率テーブルの差の絶対値和の半分である全変動（total variation）で測ると、この混合分布は $\mathrm{TV}(p^1 \otimes p^2, p)/S$ だけ p から離れている。誤差はステップ数の逆数に比例して減少し、トークン間の依存性が強いほど大きくなる。

1 ステップでの誤差は計算する前から予測できる。積分布の 3 つの偽のブロブは確率質量のうち $0.5 \cdot 0.2 + 0.3 \cdot 0.2 + 0.2 \cdot 0.8 = 0.32$ を受け取り、もしブロブ同士に重なりがなければこれがそのまま全変動になる。各セルの確率は 1 次元のガウス積分に分解できるため、以下のコードセルでは p を厳密に計算し、そのエントロピー、2 トークン間の相互情報量、および 4 通りのステップ数に対する厳密なネットワークのサンプラ誤差を求める。

```
mix = d2l.GaussianMixture.three_modes()
means, weights = np.float64(mix.means), np.float64(mix.weights)
K, T = 32, 32                 # vocabulary of each token; number of steps
edges = np.linspace(-5, 5, K + 1)
cdf = ndtr((edges - means[:, :, None]) / mix.std)  # (modes, axes, K + 1)
cdf[:, :, 0], cdf[:, :, -1] = 0, 1      # edge cells extend to infinity
cell = np.diff(cdf, axis=2)             # mass of each cell, per mode, axis
table = np.einsum('m,mi,mj->ij', weights, cell[:, 0], cell[:, 1])
p1, p2 = table.sum(1), table.sum(0)     # marginals of the two tokens
product = np.outer(p1, p2)
H = lambda p: -xlogy(p, p).sum()        # entropy in nats
tv = lambda p, q: np.abs(p - q).sum() / 2
mutual = H(p1) + H(p2) - H(table)
print(f'entropy {H(table):.3f} nats, mutual information {mutual:.3f} nats')
for S in (1, 2, 8, 32):
```

```
    print(f'{S:>2} steps, exact network: TV {tv(product, table) / S:.3f}')
```

```
entropy 4.840 nats, mutual information 0.479 nats
 1 steps, exact network: TV 0.314
 2 steps, exact network: TV 0.157
 8 steps, exact network: TV 0.039
32 steps, exact network: TV 0.010
```

1 ステップのサンプラは確率質量の 0.314 を誤った場所に配置する。隣接するブロブがわずかに重なっているため予測値の 0.32 より少し小さい値となっており、32 ステップでは誤差は 0.010 まで下がる。

17.9.3.2 確信度順のマスク解除

実用上の多くのサンプラはランダムにマスク解除するわけではない。MaskGIT (Chang et al. 2022) や拡散言語モデル LLaDA (Nie, Zhu, You, et al. 2025) は、マスクされたすべての位置で候補トークンをサンプリングし、それぞれの予測分布のもとで最も確率の高い候補だけを残して、残りを再びマスクする。この**確信度再マスキング**（confidence remasking）は、サンプリングした値に基づいてどのトークンをマスク解除するかを選ぶ。これはランダムな順序より優れているのだろうか。2 トークンの確率テーブルを使えば、一切の訓練なしに答えが出る。2 トークンの場合、確信度再マスキングは両方を周辺分布 $A \sim p^1$ と $B \sim p^2$ からサンプリングし、自身の周辺分布のもとでより確率の高い方を残して、もう一方を残したトークンが与えられた条件付き分布から引き直す。その出力テーブルは次のようになる。

$$\begin{aligned} q(a,b) = {} & p^1(a)\, P\big(p^2(B) \le p^1(a)\big)\, p(b \mid a) \\ & + p^2(b)\, P\big(p^1(A) < p^2(b)\big)\, p(a \mid b), \end{aligned}$$

ここで第 1 項は第 1 トークンのサンプル a を残す場合、第 2 項は第 2 トークンのサンプル b を残す場合である。次のコードセルでは、このテーブルを 1 ステップ、2 回のランダムジャンプ、および固定順序（第 1 トークンを周辺分布から、第 2 トークンを条件付き分布からサンプリング）のテーブルと比較する。

```
c2 = table / p1[:, None]                # p(x2 | x1); each row sums to one
c1 = table / p2[None, :]                # p(x1 | x2); each column sums to one
keep1 = p1[:, None] >= p2[None, :]      # the draw of x1 is the likelier one
confidence = ((product * keep1).sum(1)[:, None] * c2
              + (product * ~keep1).sum(0)[None, :] * c1)
centers = (edges[:-1] + edges[1:]) / 2
grid = np.stack(np.meshgrid(centers, centers, indexing='ij'), -1)
mode = ((grid[:, :, None] - means) ** 2).sum(-1).argmin(-1)  # nearest mode
orders = {'one step': product, 'two random jumps': (product + table) / 2,
          'fixed order': p1[:, None] * c2, 'confidence': confidence}
print(f'{"order":>16} {"TV":>6} {"entropy":>8}  mass near the modes')
for name, q in orders.items():
    masses = ' '.join(f'{q[mode == k].sum():.3f}' for k in range(3))
    print(f'{name:>16} {tv(q, table):6.3f} {H(q):8.3f}  {masses}')
```

```
           order     TV  entropy  mass near the modes
       one step  0.314    5.319  0.482 0.321 0.197
two random jumps  0.157    5.216  0.491 0.310 0.199
```

```
fixed order  0.000    4.840  0.500 0.300 0.200
 confidence  0.179    4.406  0.606 0.322 0.072
```

条件付き分布が厳密であれば、左から右であれランダムであれ、トークンをサンプリングする前に固定された順序で 1 ステップに 1 トークンずつマスク解除するサンプラは、固定順序の結果が示すとおり厳密になる。2 回のランダムジャンプでは半分の確率で両トークンが同時にマスク解除され、1 ステップ誤差は半分になる。驚くべきことに、同じく 1 ステップに 1 トークンずつマスク解除しながらサンプリングした値から順序を選ぶ確信度再マスキングは、2 回のランダムジャンプよりも p から遠ざかり、0.179 対 0.157 となる。上の式からその理由がわかる。小さなモードからのサンプルは周辺確率が低いため、比較に負けて引き直される側になりやすく、勝った側で条件付けた分布から引き直しても小さなモードへ戻ることはめったにない。つまりこのサンプラはモード探索的であり、最小モードの質量は 0.2 から約 0.07 へ落ち込み、エントロピーも 4.84 から 4.41 nat へ低下する。テキストではエントロピーが低い方が流暢に読めることが多く、また不完全なネットワークでは簡単な位置から先にマスク解除することが有効な場合もある。しかし変分上界はこのサンプラを記述しておらず、そのバイアスを検出できない。変分上界はランダムマスクのもとで予測を評価するものであり、言い換えればランダム順のサンプラを記述しているからである。

17.9.4 訓練されたノイズ除去器

ここまでの数値はすべて厳密な確率テーブルから求めたものであった。ここからはネットワークを実際に訓練する。まずはあらゆる誤差の原因を切り分けられるトークンペアを扱い、続いて画像を扱う。

17.9.4.1 量子化された混合分布

ペア用のネットワークは、インデックス K にある 33 番目の値としてのマスクを含む両トークンのワンホットエンコーディングと時刻 t/T を受け取り、各位置について 32 通りのクリーンな値に対するロジットを出力する。スケジュールは $T = 32$ ステップの線形スケジュールとする。

```
class MaskedMLP(nn.Module):
    """Logits of both clean tokens of a pair, given the masked pair and t."""
    def __init__(self, K, hidden=256):
        super().__init__()
        self.K = K
        self.net = d2l.make_mlp([2 * (K + 1) + 1, hidden, hidden, 2 * K],
                                nn.SiLU())

    def forward(self, z, t):  # z: (n, 2) in {0, ..., K}, where K is the mask
        h = F.one_hot(z, self.K + 1).float().flatten(1)
        return self.net(torch.cat([h, t[:, None]], 1)).view(-1, 2, self.K)

def quantize(x):  # points in the plane to pairs of tokens in {0, ..., K - 1}
    return ((x + 5) / 10 * K).floor().long().clamp(0, K - 1)

alpha_bar = 1 - torch.arange(T + 1) / T     # the linear schedule
```

このネットワークを `d2l.train_regression`（17.1 節）により、混合分布から毎回新たにサンプリングした 1 バッチあたり 512 ペアと一様にサンプリングしたステップを用い、6,000 回の更新にわたってコサイン減衰する学習率で訓練する。

```
torch.manual_seed(0)
batch_fn = lambda: (quantize(mix.sample(512)), torch.randint(1, T + 1, (512,)))
loss_fn = lambda model, batch: masked_loss(model, *batch, alpha_bar,
                                           mask_id=K).mean()
_, pair_model, _ = d2l.train_regression(MaskedMLP(K), loss_fn, batch_fn,
                                        6000, cosine=True)
```

位置が 2 つだけなので、変分上界の期待値は厳密に計算できる。各ステップにおいて、少なくとも 1 つのマスクを含むマスクパターンは 3 通りある。m 個のマスクを持つパターンの確率は $u^m(1-u)^{2-m}$（ただし $u = 1-\bar{\alpha}_t$）であり、1,024 通りのペアを確率テーブルで重み付けする。この平均はテーブルのエントロピー、すなわちいかなるモデルも下回ることのできない平均負の対数尤度の最小値を下回らない（37.1 節）。ネットワークはどこまでこの値に近づけるだろうか。

テーブルからサンプリングされたペアの 2 トークンを X^1、X^2 と書き、厳密なネットワークについてこれを計算してみよう。変分上界は、逆連鎖が各トークンをマスク解除するステップにおいてそのトークンを評価する。実際、重み w_t と、確率 $1-\bar{\alpha}_t$（ステップ t でトークンがマスクされている確率）の積は $\bar{\alpha}_{t-1} - \bar{\alpha}_t$ であり、これは逆連鎖がステップ t でそのトークンをマスク解除する確率に等しい。2 つのトークンが異なるステップでマスク解除されるとき、最初のトークンは周辺分布から、2 番目のトークンは最初のトークンが与えられた条件付き分布から予測されるため、合計のコストは $H(X^1) + H(X^2 \mid X^1) = H(X^1, X^2)$ となり、余分なコストは生じない。一方、同じステップでマスク解除される 2 つのトークンはそれぞれの周辺分布から独立に予測されるため、コストは $H(X^1) + H(X^2)$ となり、$H(X^1, X^2)$ より相互情報量 $I(X^1; X^2)$（37.6 節）だけ余分にかかる。線形スケジュールでは各トークンがあるステップでマスク解除される確率は $1/T$ なので、両方が同じステップでマスク解除される確率は $T \cdot (1/T)^2 = 1/T$ となり、厳密なネットワークは次の値を達成する。

$$H(X^1, X^2) + \frac{I(X^1; X^2)}{T}$$

（演習問題 5 では、この議論を任意のスケジュールに対する証明へと一般化する。）各位置を独立に予測するいかなるネットワークも、この下限 $4.840 + 0.479/32 \approx 4.855$ nat を下回ることはできない。$T \to \infty$ では 2 つのトークンが同じステップでマスク解除されることはなくなり、下限はエントロピーまで下がる。

```
@torch.inference_mode()
def exact_bound(model, table, alpha_bar, mask_id):
    """The bound averaged over the table, summed exactly over t and masks."""
    T, K = len(alpha_bar) - 1, len(table)
    pairs = torch.cartesian_prod(torch.arange(K), torch.arange(K))
    p, total = torch.tensor(table.flatten(), dtype=torch.float32), 0.0
    for t in range(1, T + 1):
        u = 1 - alpha_bar[t]
        w = (alpha_bar[t - 1] - alpha_bar[t]) / u
        for masked in torch.tensor([[1, 0], [0, 1], [1, 1]], dtype=torch.bool):
            z, m = torch.where(masked, mask_id, pairs), masked.sum()
            logp = torch.log_softmax(model(z, torch.full((K * K,), t / T)), -1)
            nll = -(logp.gather(-1, pairs[..., None])[..., 0] * masked).sum(1)
            total += u ** m * (1 - u) ** (2 - m) * w * (p @ nll)
    return float(total)
```

```
bound = exact_bound(pair_model, table, alpha_bar, mask_id=K)
print(f'exact bound {bound:.3f} nats; '
      f'entropy {H(table):.3f}, floor {H(table) + mutual / T:.3f}')
```

```
exact bound 4.856 nats; entropy 4.840, floor 4.855
```

上界はエントロピーを 0.02 nat 未満しか上回っておらず、モデルは各トークンの周辺分布と他方が与えられたときの条件付き分布を学習できている。差のほとんどすべては下限によるものであり、下限に対するネットワーク自身の超過分は 0.001 nat のオーダーにすぎない。この下限は単なる上界の緩さではない。厳密なネットワークの 32 ステップサンプラがモデル化するのは $(1-1/32)\,p+(1/32)\,p^1\otimes p^2$ であって、p ではないからである。何が失われるかについて、変分上界とサンプラは一致している。

訓練されたサンプラも同じだけ失うのだろうか。2 トークンであれば、出力テーブルはサンプリングではなく総和によって厳密に計算できる。各トークンはジャンプ k において確率 Δ_k（コード中の `pi`）で、もう一方とは独立にマスク解除される。同時にマスク解除されるトークンはそのレベルにおけるネットワークの 2 つの周辺分布からサンプリングされ、後からマスク解除されるトークンは先に現れたトークンが与えられた条件付き分布からサンプリングされる。`conditionals` はペアモデルからこれらの予測を読み取り、`sampler_table` は全ジャンプにわたって和をとる。これに真の周辺分布と条件付き分布を渡せば、先に表示した厳密なネットワークの誤差が再現されるはずである。

```
@torch.inference_mode()
def conditionals(model, tau):
    """A pair model's predictions at time tau: both marginals with both
    tokens masked, then c2[a, b] = q(x2 = b | x1 = a), c1[a, b] = q(x1 = a |
    x2 = b)."""
    K = model.K                                  # also the mask index
    a, m = torch.arange(K), torch.full((K,), K)
    z = torch.cat([torch.tensor([[K, K]]),       # both masked
                   torch.stack([a, m], 1),       # x1 = a revealed
                   torch.stack([m, a], 1)])      # x2 = a revealed
    q = torch.softmax(model(z, torch.full((2 * K + 1,), tau)), -1)
    q = q.double().numpy()                       # (2K + 1, 2, K)
    return q[0, 0], q[0, 1], q[1:K + 1, 1], q[K + 1:, 0].T
```

関数 `cond`（`conditionals` と同じシグネチャを持つ関数）を与えると、`sampler_table` はすべてのジャンプの組み合わせにわたって和をとり、S ステップのマスク解除サンプラの厳密な出力分布を構成する。これを真の同時分布 `table` と比較し、`pair_model` と `exact` の両方について $S\in\{32,8,2,1\}$ にわたり評価する。

```
def sampler_table(cond, steps, alpha_bar):
    """Exact output table of `unmask` on a pair; cond(t / T) returns the
    predictions at level t in the order of `conditionals`."""
    T = len(alpha_bar) - 1
    levels = [round(l) for l in np.linspace(T, 0, steps + 1)]
    pi = np.float64([float(alpha_bar[s] - alpha_bar[t])  # P(revealed in jump)
                     for t, s in zip(levels[:-1], levels[1:])])[:, None]
    q1, q2, c2, c1 = map(np.stack, zip(*[cond(t / T) for t in levels[:-1]]))
    before1 = np.cumsum(pi * q1, 0) - pi * q1    # revealed in earlier jumps
    before2 = np.cumsum(pi * q2, 0) - pi * q2
```

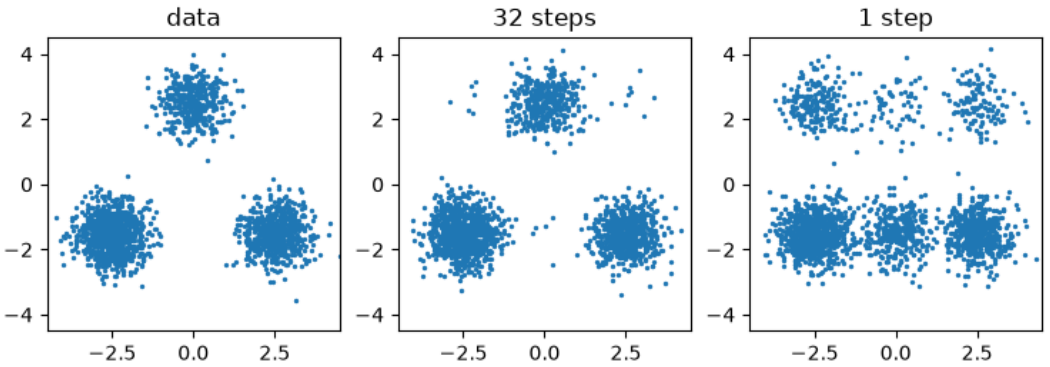
```
        return (np.einsum('j,ja,jb->ab', pi[:, 0] ** 2, q1, q2)   # same jump
                + np.einsum('ja,jab->ab', pi * before1, c2)       # x1 first
                + np.einsum('jb,jab->ab', pi * before2, c1))      # x2 first

    exact = lambda tau: (p1, p2, c2, c1)  # the true marginals and conditionals
    print('TV to the table: trained and exact network; TV between the two')
    for S in (32, 8, 2, 1):
        q = sampler_table(lambda tau: conditionals(pair_model, tau), S, alpha_bar)
        q_exact = sampler_table(exact, S, alpha_bar)
        print(f'{S:>2} steps: trained {tv(q, table):.3f}, exact '
              f'{tv(q_exact, table):.3f}, between {tv(q, q_exact):.3f}')
```

```
TV to the table: trained and exact network; TV between the two
32 steps: trained 0.020, exact 0.010, between 0.013
 8 steps: trained 0.045, exact 0.039, between 0.013
 2 steps: trained 0.159, exact 0.157, between 0.012
 1 steps: trained 0.316, exact 0.314, between 0.012
```

厳密なネットワークの列は $0.314/S$ を再現し、訓練されたサンプラもすべてのステップ数でそれに追随している。訓練されたテーブルと厳密なテーブルの距離であるネットワーク自身の誤差は、約 0.01 でどの S でもほぼ変わらない。この誤差は 32 ステップでは距離を 0.020 対 0.010 と 2 倍にするが、1 ステップや 2 ステップではほとんど問題にならない。ステップ数を増やしてもこの誤差は消えず、より良いネットワークによってしか減らせない。

以下のパネルは、2,000 個のデータペアと、各サンプラから 2,000 個ずつ unmask でサンプリングしたペアを、それぞれのセル内のランダムな位置に配置して示したものである。1 ステップサンプラは、あるモードの列と別のモードの行が交差する 3 つの偽のブロブを埋めてしまう。

```
torch.manual_seed(1)
pairs = [quantize(mix.sample(2000))] + [
    unmask(pair_model, (2000, 2), S, alpha_bar, mask_id=K) for S in (32, 1)]
decode = lambda z: (z + torch.rand(len(z), 2)) / K * 10 - 5  # inside cell
d2l.scatter_panels([decode(z) for z in pairs], ['data', '32 steps', '1 step']);
```



17.9.4.2 2 値トークンとしての画像

2 トークンで仕組みがわかったので、画像でスケールさせた場合を見よう。画像に対するマスク拡散は、学習されたトークナイザの離散符号上で動作する (Chang et al. 2022)。ここではトークナ

イザを必要としない 2 値ピクセルをそのような符号の代わりとして用いる。Fashion-MNIST を半分の輝度でしきい値処理すると、すべてのピクセルが 2 値をとるトークンとなり、マスクされたピクセルが 3 番目の値を持つ。これにより画像はグリッド上の 784 個のトークン系列となるため、ネットワークには本章の U-Net（17.4 節）をそのまま使える。ワンホットエンコーディングを 3 つの入力チャネルとして受け取り、クリーンなピクセルのロジットを 2 つの出力チャネルとして返すのである。損失とサンプラはペアの場合と同じものを用い、$T = 100$ ステップの線形スケジュールとし、マスクのインデックスを 2 とする。

```
X, _, X_test, _ = d2l.load_fashion_mnist_pm1(binary=True)  # (n, 28, 28)
T_img = 100                                    # steps of the image chain
alpha_bar_img = 1 - torch.arange(T_img + 1, device=X.device) / T_img

class MaskedUNet(nn.Module):
    """d2l.UNet on one-hot pixels (black, white, mask); clean-pixel logits."""
    def __init__(self):
        super().__init__()
        self.unet = d2l.UNet(in_channels=3, out_channels=2)

    def forward(self, z, t):  # z: (n, 28, 28) in {0, 1, 2}
        x = F.one_hot(z, 3).float().permute(0, 3, 1, 2)
        return self.unet(x, t).permute(0, 2, 3, 1)  # (n, 28, 28, 2)
```

訓練は 6,000 回の更新を 128 枚の画像バッチ上で行い、定数の学習率と、半減期 1,000 回更新の重みの移動平均を用いる。

```
torch.manual_seed(2)
def batch_fn():
    i = torch.randint(0, len(X), (128,), device=X.device)
    return X[i], torch.randint(1, T_img + 1, (128,), device=X.device)

loss_fn = lambda model, batch: masked_loss(  # per pixel
    model, *batch, alpha_bar_img, mask_id=2).mean() / (28 * 28)
_, unet, _ = d2l.train_regression(MaskedUNet().to(X.device), loss_fn,
                                  batch_fn, 6000)
```

1,000 枚のホールドアウト画像に対し、各画像・各レベルにつき 1 つのランダムマスクを用いて変分上界を推定し、訓練セットにおける各位置の白ピクセルの出現頻度を用いてピクセルを独立とみなすモデルの交差エントロピーと比較する。

```
X_eval = X_test[:1000]
with torch.inference_mode():
    bound = sum(masked_loss(unet, X_eval, torch.full_like(X_eval[:, 0, 0], t),
                            alpha_bar_img, mask_id=2).mean()
                for t in range(1, T_img + 1)) / T_img
white = X.float().mean(0).clamp(1e-6, 1 - 1e-6)  # frequency of white
indep = -torch.where(X_eval == 1, white.log(), (1 - white).log()).sum((1, 2))
bits = lambda nats: f'{nats / math.log(2) / (28 * 28):.3f} bits per pixel'
print(f'bound {bits(bound)}, independent pixels {bits(indep.mean())}')
```

```
bound 0.170 bits per pixel, independent pixels 0.702 bits per pixel
```

変分上界はピクセルあたり約 0.17 ビットであるのに対し、独立ピクセルモデルでは約 0.70 ビッ

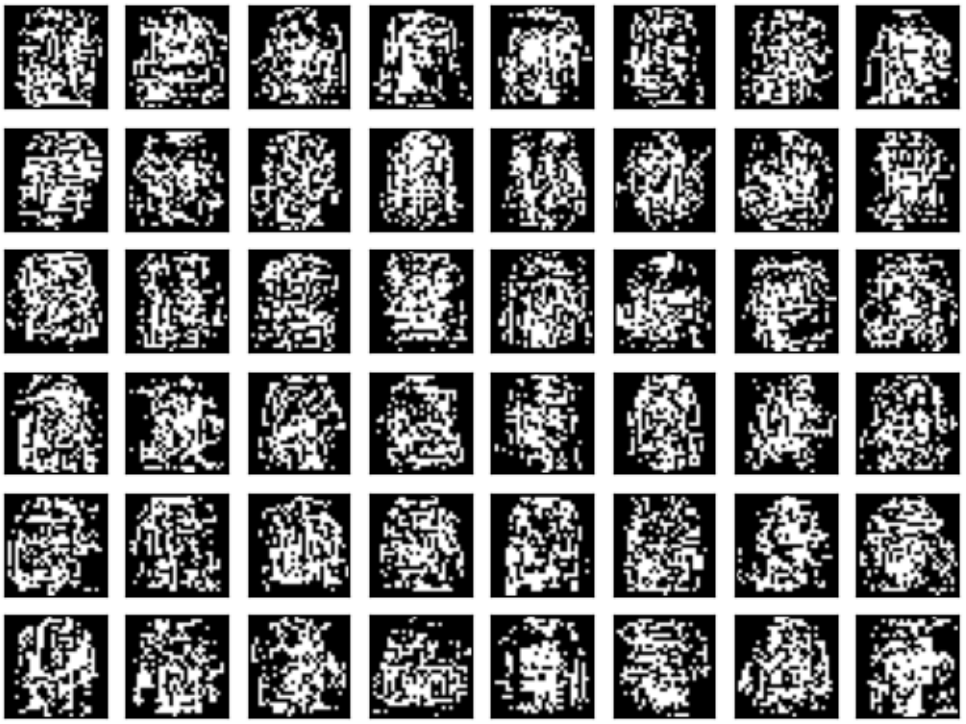
トである。符号長として読めば、ホールドアウトされた衣類画像 1 枚あたり約 135 ビットであり、独立ピクセルの約 550 ビットから大きく短縮されている。もっとも、隣接ピクセルを見るモデルならほぼどれでも大差で勝てるため、独立ピクセルが弱い基準であることは確かである。この実行が示しているのはその差の大きさである。ピクセル間の依存性をモデル化すれば独立ピクセルの符号長の少なくとも 4 分の 3 を節約でき、ネットワークはその節約を実現するのに十分な依存性を学習している。本来対等な比較相手となるのは、ラスタ順に 1 ピクセルずつマスク解除する同規模のモデルであろう。ここではそれを訓練しないが、テキスト上でのマスク拡散と左から右へのモデルの比較については 17.9.5.1 節で紹介する。

サンプラは 1,000 枚の画像を 100 ステップで生成し、グリッドには最初の 48 枚を示す。

```
torch.manual_seed(3)
z100 = unmask(unet, (1000, 28, 28), T_img, alpha_bar_img, mask_id=2)
d2l.show_gray(z100[:48], 6, 8, scale=0.8);
```

サンプルの大半は識別可能な 2 値の衣類になっているが、前節までのグレースケール画像のサンプルに比べると粗く、5 枚に 1 枚ほどは断片的であるかほぼ空白である。6,000 回の更新では U-Net は訓練不足のままであるが、ここでは問題ない。この実験の目的は変分上界とステップ数の関係を調べることであり、最良の衣類画像を生成することではないからである。100 回の各ジャンプでは、それ以前のジャンプでマスク解除されたすべてのピクセルを参照する予測から約 8 ピクセルずつがマスク解除される。次のグリッドは、1 回あたり約 200 ピクセルをマスク解除する 4 回のジャンプを用いた結果である。

```
z4 = unmask(unet, (1000, 28, 28), 4, alpha_bar_img, mask_id=2)
d2l.show_gray(z4[:48], 6, 8, scale=0.8);
```



衣類の形はまったく残らない。最初のジャンプが全マスク画像に対する予測（衣類がどのあたりに置かれやすいかしか知らない分布）から各ピクセルを独立にサンプリングし、後続のすべてのジャンプがこのノイズに条件付けられるためである。残るのは特定の衣類というより平均的な衣類の輪郭であり、その内部がテクスチャで埋め尽くされている。

4 ステップで何が失われるかを測るため、上下左右 4 つの隣接ピクセルすべてと値が異なるピクセルである**スペックル**を各種 1,000 枚の画像で数え、その割合と 1,000 枚にわたる標準誤差を表示する。さらに 100 ステップでもまだ何が失われているかを見るため、3 つ目のバッチでは平均して 1 ジャンプあたり 1 ピクセルとなる 784 回のジャンプを行う。元の連鎖の 100 レベルではこれだけのジャンプ数を担えない。unmask はレベルを連鎖の整数レベルへ丸めるため、同じレベル間のジャンプでは何もマスク解除されないからである。そこでこのジャンプには独自の 784 レベルの線形スケジュールを用いる。その時刻 $k/784$ の大半はネットワークが訓練時に見た 100 個の時刻の間に入り（最後の 7 個は最小時刻より下にある）、いずれにせよマスクされているピクセルの個数自体がネットワークに実質的な時刻を教えている。

```
def speckles(z):
    """Per image, the fraction of interior pixels unlike all four neighbors."""
    c = z[:, 1:-1, 1:-1]
    unlike = ((c != z[:, :-2, 1:-1]) & (c != z[:, 2:, 1:-1])
              & (c != z[:, 1:-1, :-2]) & (c != z[:, 1:-1, 2:]))
    return unlike.float().mean((1, 2))

alpha_bar_784 = 1 - torch.arange(785, device=X.device) / 784  # 784 levels
z784 = unmask(unet, (1000, 28, 28), 784, alpha_bar_784, mask_id=2)
for name, z in (('data', X[:1000]), ('784 steps', z784), ('100 steps', z100),
                ('4 steps', z4)):
```

```
    per_image = speckles(z)
    print(f'{name:>9}: speckles {per_image.mean():.4f}, standard error over '
          f'the 1,000 images {per_image.std() / len(z) ** 0.5:.5f}')
```

```
     data: speckles 0.0037, standard error over the 1,000 images 0.00016
784 steps: speckles 0.0047, standard error over the 1,000 images 0.00016
100 steps: speckles 0.0075, standard error over the 1,000 images 0.00021
  4 steps: speckles 0.0330, standard error over the 1,000 images 0.00027
```

データには内部ピクセル 1,000 個あたり約 4 個のスペックルがあり、4 ステップのサンプルにはその約 9 倍ある。これは 1 ステップのペアにおける偽のブロブの画像版である。100 ステップのサンプルはデータの約 2 倍であり、そこには 2 つの原因が混在している。1 ジャンプあたりまだ約 8 ピクセルが同時に現れることと、6,000 回の更新後ではネットワークが不完全なことである。784 ステップの行がこの 2 つを切り分ける。ペアの場合と同様、ジャンプ数を増やすと独立性誤差は縮まりネットワーク自身の誤差はそのまま残るが、1 ジャンプあたり 1 ピクセルにすることで 100 ステップの過剰分の 3 分の 2 から 4 分の 3 が解消される。この減少は画像のサンプルの偶然だけで生じ得るだろうか。両方の行は同じネットワークから生成されているため、比較における偶然の要素はそれぞれの 1,000 枚の画像だけである。各値は独立な画像にわたる平均であり、標準誤差は 0.00016〜0.0002、2 つの平均の差の標準誤差は約 0.00026 であるのに対し、実際の減少量は約 10 倍に達する。したがって、100 ステップで失われていたものの大半はネットワークではなく独立性誤差である。残りの部分（784 ステップのサンプルになおデータの約 1.3 倍、差の標準誤差にして約 4 倍分のスペックルを残している要因）は、ネットワーク自身の誤差と、ピクセルが隣接ピクセルとまれに同じジャンプを共有するときにわずかに残る独立性誤差の和である。演習問題 2e ではこの 2 つをさらに分離する。

17.9.5 隣接するモデルと振り返り

本節ではここまで、マスク拡散をそれ自身の枠組みで扱ってきた。本書においてこれに隣接するモデルは 11.2 節や 11.4 節の言語モデルであり、系譜上の祖先は本章の前節までのモデルである。

17.9.5.1 拡散と自己回帰

11.2 節の言語モデルを訓練した読者なら、そのモデルと本節のモデルがどう関係するのか気になるだろう。1 ステップあたりにマスク解除するトークン数によってサンプラを整理すれば、2 トークンの確率テーブルがその答えを与えてくれる。

左から右へ固定された順序で 1 ステップに 1 トークンずつマスク解除すると、サンプラは 11.2 節の言語モデルと同じ形になる。その条件付き分布は、マスクされていないプレフィックスとマスクされた残りからなる入力に対するネットワークの予測である。ペアの実験では、これは 4 つの順序を比較したテーブルの固定順序の行にあたり、距離 0.000 で p と一致する。条件付き分布が厳密であれば、1 トークンずつマスク解除する連鎖は厳密であり、その損失はエントロピーに等しい。

一様ランダムな順序で 1 ステップに 1 トークンずつマスク解除すると、サンプラは**順序非依存な自己回帰モデル**（order-agnostic autoregressive model）(Uria et al. 2014) となり、その損失はあらゆる順序にわたる負の対数尤度の平均になる。Hoogeboom et al. (2022) は、時刻を無視するネットワークにおいてこの損失が 17.9.2.3 節の連続時間上界と一致することを示し（演習問題 6 ではベータ積分を用いてこれを導出する）、Ou et al. (2025) は、マスクトークンに対する最良の予測がすでにマスク解除されたトークンのみに依存するため、時刻を無視しても何も失われないことを示した。したがってマスク拡散モデルは、すべての順序にわたる言語モデルとして同時に訓練されていることになる。厳密なネットワークではどの順序も同じ同時尤度、すなわちデー

タのエントロピーを与えるため、連続時間上界はタイトである。これはペアにおける下限を順序の側から見たものにほかならない。$T \to \infty$ では 2 つのトークンが同時にマスク解除されることはなくなり、超過分 I/T は消える。

1 ステップに複数のトークンをマスク解除すると、サンプラは独立性誤差を被る。ペアの場合、S 回のジャンプでは確率質量の $1/S$ が周辺分布の積からサンプリングされ、T ステップの上界は I/T だけ上昇する。その見返りとして、S 回のネットワーク評価で L 個のトークンを生成できるのに対し、左から右へのモデルは L 回の評価を必要とする。

もっとも、この 2 種類の評価の計算規模は同じではない。キーと値をキャッシュする左から右へのモデル（11.3 節）は、1 ステップにつき新しい 1 トークンだけを処理する。これに対しノイズ除去器には因果マスクがないため、1 つのトークンがマスク解除されるたびにすべての位置の活性化値が変化し、何もキャッシュできず、各評価は全 L 位置にわたる順伝播となる。トークン通過回数で数えれば、コストは約 L 対 SL である。したがって並列デコーディングは逐次ステップ数を減らし、それに伴ってレイテンシを削減する一方、総演算量は通常より多くなる。Fast-dLLM (Wu et al. 2026) はステップ間で近似キャッシュを再利用することでその一部を取り戻し、ブロック拡散（block diffusion）(Arriola et al. 2025) は完了したブロック上に厳密なキャッシュを持ちながらブロック単位で順次デコードすることで削減を図る。

対極に位置するのが BERT (Devlin et al. 2018) であり、テキスト中のトークンの 15% を隠して両側から予測する。11.4 節のマスク言語モデルも同様で、各バイトを確率 15% でマスクする。この単一のマスクレベルに制限すると、損失 Equation 17.71 は（BERT が一部のマスクをランダムなトークンや元のトークンに置き換える点を除けば）隠されたトークンの負の対数尤度の和というそれらのマスクトークン目的関数に帰着する。すなわちマスク言語モデルは 1 ステップの離散拡散モデルである (Austin et al. 2021)。ただし訓練時に 1 つのマスクレベルしか見ていないため、全マスク系列から出発するサンプラに入力すると、訓練時とはかけ離れた入力を見せることになる。

これら両極端の間に位置するのが今日の拡散言語モデルであり、そのノイズ除去器は因果マスクを持たない Transformer に、$[\mathrm{M}]$ 用の語彙項目を追加し、時刻入力を省いたものである (Nie, Zhu, You, et al. 2025)。これらは何をもたらすのだろうか。データが豊富にある状況での尤度ではない。常に新鮮なデータで訓練する場合、マスク拡散が自己回帰モデルと同じ損失に到達するには定数倍多くの計算量が必要になる (Nie, Zhu, Du, et al. 2025)。一方、データが限られていて何度も繰り返し学習する場合にはマスク拡散が上回ることがあり (Prabhudesai et al. 2025)、著者らはランダムマスクがデータ拡張として働くためだと説明している。これらのモデルの強みは、1 回のネットワーク評価で多数のトークンを書き込むことで逐次ステップ数を減らせる並列デコーディングと、空白の両側のテキストに条件付けられる穴埋め生成（infilling）にある。

17.9.5.2 拡散モデルの振り返り

関連文献の節を除けば、本章はここで終わり、「発展」パートもここで完結する。最後に一度全体を振り返っておこう。すべての節を貫いていたのは 1 つの着想である。あらゆるノイズレベルでノイズを含む点からクリーンな点を予測するネットワークはそれ自体が生成器であり、サンプラはその予測を積分することでデータへと変換する。トークンにおいては、ノイズを含む点は一部が消去された系列となり、予測は消去された値の上の分布となり、積分は段階的なマスク解除の列となったが、変分上界はそのまま引き継がれた。

この解答の各要素にはそれぞれ弱点があり、実務家はそれらすべてに直面する。密度は上界としてしか得られず、最良の上界を与える重み付けは最良のサンプルを与える重み付けとは一致しない（17.6 節）。第 16 章の敵対的ゲームは回帰に置き換わるものの、サンプリングではネットワークを何度も呼び出す必要がある。元の DDPM 連鎖では 1,000 回、優れたソルバを使っても数十回（17.5 節）かかり、数回で済むのは生徒モデルを蒸留する 2 回目の訓練を行った後だけである（17.8 節）。潜在空間を導入すれば大きな画像も扱えるようになるが、最初に訓練してから凍結す

るオートエンコーダという 2 つ目のモデルが増える（17.7 節）。そしてテキストでは、1 回の評価あたり多数のトークンを生成しようとすれば独立性誤差が生じ、しかもサンプラは一度書き込んだトークンを取り消せない。

未解決の問いもいくつか残っている。教師モデルの軌跡に沿って訓練された少数ステップの生徒モデルは、依然として教師に及ばない。教師に並ぶかそれを上回る生徒モデルは、分布マッチングにクリティックを加えており、ノイズからサンプルへの決定論的な写像を手放している（17.8 節）。また、最良の尤度と最良のサンプルの両方を同時に達成する単一の重み付けは知られていない。テキストにおけるマスク拡散と自己回帰の計算量ギャップが最大規模のスケールでも残るかどうかは、本書で扱ういかなる実験よりもはるかに大規模な実験によって決着がつきつつある。「言語モデル」パートでは、概観するアーキテクチャの 1 つとして再び拡散言語モデルを取り上げる（31.8 節（英語版））。

17.9.6 まとめ

離散拡散は行確率遷移行列 $\mathbf{Q}_t$ によってカテゴリトークンを破損させる。吸収カーネルのもとでは、各トークンは t ステップ後まで確率 $\bar{\alpha}_t = \prod_{s \le t}(1-\beta_s)$ で生存し、そうでなければ $[\mathrm{M}]$ に置き換えられる。負の対数尤度に対する変分上界は、重み $w_t = (\bar{\alpha}_{t-1} - \bar{\alpha}_t)/(1-\bar{\alpha}_t)$（線形スケジュールでは $1/t$）を持つマスクトークン交差エントロピーの重み付き和へと簡約され、連続時間極限は $\int_0^1 \frac{1}{u}\, \mathbb{E}_u[\ell]\, du$ となる。このとき独立なマスキングにより、マスク位置における最適なクリーントークン分布は時刻非依存になる。逆モデルが各ステップ内で位置ごとに分解されるため、複数のトークンを同時にマスク解除すると独立性誤差が生じる。すなわち 2 トークン分布 p に対して、S 回の等間隔線形ジャンプは $(1-1/S)p + (1/S)p^1 \otimes p^2$ からサンプリングし、$1/S$ に比例する全変動誤差を生じる。また T ステップの上界の下限は、エントロピー $H(X^1, X^2)$ を $I(X^1; X^2)/T$ だけ上回る。マスク解除スケジュールという観点から見ると、マスク拡散は古典的な系列モデルを結びつける。左から右へ 1 ステップに 1 トークンずつマスク解除すれば自己回帰モデル（11.2 節）が得られ、ランダムな順序で 1 ステップに 1 トークンずつマスク解除すれば順序非依存な自己回帰が得られ、単一のマスク率 $u = 0.15$ で訓練すれば BERT（11.4 節）が得られる。

17.9.7 演習問題

1. **一様カーネル**。順連鎖がすべてのステップで一様カーネル $\mathbf{Q}_t = (1-\beta_t)\,\mathbf{I} + \beta_t\, \frac{1}{K}\mathbf{1}\mathbf{1}^\top$（ただし $\beta_t > 0$）を用いるとする。
   1. $\bar{\mathbf{Q}}_t = \bar{\alpha}_t\,\mathbf{I} + (1-\bar{\alpha}_t)\,\frac{1}{K}\mathbf{1}\mathbf{1}^\top$ となること、および $\mathbf{x}_t = \mathbf{x}_0$ となる確率が $\bar{\alpha}_t + (1-\bar{\alpha}_t)/K$ であることを示せ。したがって、ステップ t である値を示しているトークンは再サンプリングされたものである可能性がある。吸収カーネルでは答える必要がないが、このカーネルのもとではすべての位置についてネットワークが答えなければならない問いは何か。
   2. 事後分布 $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ を Equation 17.66 から、$\mathbf{x}_t = \mathbf{x}_0$ の場合と $\mathbf{x}_t \neq \mathbf{x}_0$ の場合に分けて導出せよ。また、$t \ge 2$ では K 個の値すべてが正の確率を持つことを示し、2 点事後分布 Equation 17.70 との違いを確認せよ。
   3. 変分上界のダイバージェンス項は、すべての位置でこの事後分布と逆モデルを比較する。これらを評価するコストが Equation 17.71 の被加数よりも高くなるのはなぜか。
2. [code] **同時にマスク解除されるペア**。サンプラがレベル $T = t_0 > t_1 > \cdots > t_S = 0$ を、$\bar{\alpha}_0 = 1$ かつ $\bar{\alpha}_T = 0$ を満たすスケジュールのもとで用いるとし、$\Delta_k = \bar{\alpha}_{t_{k+1}} - \bar{\alpha}_{t_k}$ を $\bar{\alpha}$ のジャンプ k（$k = 0, \dots, S-1$）における増加量とする。

1. レベル t_k 上の粗視化した連鎖では、トークンはジャンプ k で確率 Δ_k により、他の位置とは独立にマスク解除される。2 つのトークンが同じジャンプでマスク解除される確率が $\rho = \sum_k \Delta_k^2$ となること、および和が 1 になる S 個の増分のうち等間隔の増分 $\Delta_k = 1/S$ が ρ を最小化することを示せ。
2. 厳密なネットワークを用いたとき、サンプラが量子化混合分布の 2 トークンを $(1-\rho)\,p + \rho\, p^1 \otimes p^2$ からサンプリングすることを示せ。
3. 長さ L のトークン系列と等間隔増分に対して、同じジャンプでマスク解除されるトークンペア数の期待値を計算せよ。$L = 1024$ のとき、この期待値を 1 未満に抑えるには何ステップ必要か。また、テキストにおいてこの個数がサンプラの誤差を過大評価しているのはなぜか。
4. 1 つもトークンをマスク解除しないジャンプでは、次のジャンプへの入力は変化しない。したがって、演習問題 4a により厳密なネットワークがそうであるように、ネットワークが時刻を無視するサンプラでは、先にマスク解除の判定をすべて引き、少なくとも 1 つのトークンをマスク解除するジャンプでのみネットワークを評価できる。線形スケジュールのもとで S 回の等間隔ジャンプを行うとき、評価回数の期待値が $S\,(1-(1-1/S)^L)$（トークン数 L の場合）となることを示せ。これをペアの $S = 32$、2 値画像の $S = 100$ および $S = 784$、ならびに $L = S = 1024$ について計算せよ。
5. 784 ステップによる 17.9.4.2 節のサンプルには、100 ステップにおける過剰スペックルの 4 分の 1 から 3 分の 1 が残っている。S 回の等間隔ジャンプにおいて、内部ピクセルが 4 つの隣接ピクセルの少なくとも 1 つと同じジャンプを共有する確率が $1-(1-1/S)^4$ であることを示し、$S = 4$、100、784 についてその値を計算せよ。次に `unet` を 2 倍の更新回数だけ訓練し、100 回および 784 回のジャンプで再びスペックルを数えよ。どちらの個数が減少し、そこから残存していた過剰分のうちネットワークに起因する割合について何が言えるか。

3. **2 通りの逆パラメータ化。**●逆モデル Equation 17.69 はネットワークの予測に関して事後分布を平均化する。これに対し Austin et al. (2021) は次式のように置いた。

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) \propto \sum_{\hat{\mathbf{x}}_0} q(\mathbf{x}_{t-1}, \mathbf{x}_t \mid \hat{\mathbf{x}}_0)\, \tilde{p}_\theta(\hat{\mathbf{x}}_0 \mid \mathbf{x}_t) \tag{17.72}$$

ここで $\tilde{p}_\theta$ はネットワーク出力であり、各候補を $q(\mathbf{x}_t \mid \hat{\mathbf{x}}_0)$ で再重み付けする形になっている。ほとんどのカーネルではこれら 2 つの形式は異なる。データから出発した順過程の条件付き分布を $q(\mathbf{x}_0 \mid \mathbf{x}_t)$ および $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ と書くことにする。

1. 単一のトークンを考える。Equation 17.69 の予測が $q(\mathbf{x}_0 \mid \mathbf{x}_t)$ に等しいならば、あらゆるカーネルに対して逆モデルは真の逆遷移 $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ に一致することを示せ。
2. 吸収カーネルのもとでは、マスクされた位置において重み $q(\mathbf{x}_t \mid \hat{\mathbf{x}}_0)$ がすべての候補に対して等しくなり、したがって 2 つの形式が一致することを示せ。

4. [code] **時刻に依存しない予測。**●本節では、吸収カーネルのもとでマスク位置における最良の予測が t に依存しないことを論じた (Ou et al. 2025)。

1. この議論を短い証明として書き下せ。吸収順過程のどの性質を用いているか。また、一様カーネルでも同じことが成り立つか。
2. 11.4 節のマスク言語モデルは各トークンを独立に確率 15% でマスクする。(a) により、その厳密な予測はそのマスク密度のパターンにおいてサンプラが必要とする予測と一致する。長さ $L = 512$ トークンの系列に対して、そのようなモデルが訓練中に見るマスク割合の範囲（たとえば 15% の前後 3 標準偏差以内）を求めよ。線形スケジュールのもとで $S = 512$ 個の等間隔レベルを用いて全マスク系列から生成を行うとき、

全ステップのうちこの範囲のマスク割合をネットワークに見せるステップの割合はいくらか。

3. `MaskedMLP` は時刻を入力として受け取る。第 1 トークンがマスクされ第 2 トークンが観測されているパターンを固定し、`pair_model` の予測を第 2 トークンの全 32 通りの値について $t \in \{1, 16, 31\}$（すなわち時刻入力 $t/32$）で計算せよ。次に、それぞれ第 2 トークンの周辺分布で平均した 2 種類のカルバック・ライブラー・ダイバージェンスを計算せよ。すなわち、$t = 16$ および $t = 31$ における予測の、$t = 1$ における予測からのダイバージェンスと、各 t における予測の、`table` の厳密な条件付き分布からのダイバージェンスである（確率は 10^{-12} でクランプせよ）。これら 2 種類の値の大きさから、訓練済みネットワークによる時刻入力の使われ方について何が言えるか。

5. [code] **厳密なネットワークの下限。**●$\bar{\alpha}_0 = 1$ かつ $\bar{\alpha}_T = 0$ を満たすスケジュールの増分を $\Delta_t = \bar{\alpha}_{t-1} - \bar{\alpha}_t$ とし、量子化混合分布の 2 トークンを考える。

   1. 真のテーブルにわたって平均した厳密なネットワークの上界 Equation 17.71 が次式となることを示せ。

$$H(X^1, X^2) + I(X^1; X^2) \sum_{t=1}^{T} \Delta_t^2. \tag{17.73}$$

   ステップ t の項を 3 つのマスクパターンとその確率を用いて書き下せば、$(1-\bar{\alpha}_t)^2$ の望遠鏡和によって t にわたる和を計算できる。線形スケジュールでは $H(X^1, X^2) + I(X^1; X^2)/T$ となることを確かめよ。

   2. スケジュール $\bar{\alpha}_t = 1 - (t/T)^2$ に対してこの和を閉形式および $T = 32$ で評価し、線形スケジュールの下限と比較せよ。$T \to \infty$ の極限でスケジュールの選択が結果に影響しなくなるのはなぜか。

   3. `MaskedMLP` と同じ呼び出しシグネチャを持ち、`table` から計算した厳密な条件付き分布（および両トークンがマスクされている場合は周辺分布）の対数を返す関数を書け（確率は 10^{-12} でクランプせよ）。これを `exact_bound` に、2 つのスケジュールをそれぞれ `alpha_bar` として渡せ。

   4. `MaskedMLP` を `masked_loss` により、(b) のスケジュールのもとで `pair_model` と同じ設定を用いて訓練し、その厳密な上界をその下限および `pair_model` の厳密な上界と比較せよ。

6. **順序非依存な上界。**●演習問題 4a により厳密なネットワークがそうであるように、ネットワークが時刻を無視するとし、$\ell(i, U) = -\log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_0^U)$ を位置 i における損失（集合 U に属する位置だけがちょうどマスク解除されているとき）とする。このとき 17.9.2.3 節の上界の連続時間極限をマスク確率 u に関する積分として書くと、$\ell(i, U)$ の $i \notin U$ にわたる重み付き和になる。Hoogeboom et al. (2022) と Ou et al. (2025) は、これが順序非依存な自己回帰モデル (Uria et al. 2014) の期待負の対数尤度

$$\mathbb{E}_\sigma \Big[\sum_{k=1}^{L} -\log p_\theta(\mathbf{x}_0^{\sigma(k)} \mid \mathbf{x}_0^{\sigma(1)}, \ldots, \mathbf{x}_0^{\sigma(k-1)}) \Big], \tag{17.74}$$

に等しいことを示した。ここで σ は L 個の位置の一様ランダムな置換である。

   1. 積分における $\ell(i, U)$ の重みを $|U| = L - m$ のときに求めよ。評価にはベータ積分 $\int_0^1 u^{m-1}(1-u)^{L-m}\, du = (m-1)!\,(L-m)!/L!$ を用いることができる。

2. ランダムな順序が最初に U の位置をマスク解除し、次に位置 i をマスク解除する確率を求め、2 つの式が一致することを結論せよ。
3. 厳密なネットワークではどの順序も同じ値を与える。それはどの値か。演習問題 5a の $T \to \infty$ の場合と比較せよ。
4. イェンセンの不等式を用いて、この式が一様ランダムに順序を引いてからその順序でトークンをサンプリングするモデルの負の対数尤度の上界にもなることを示せ。

7. [extended] **マスク拡散言語モデル**。●●Prabhudesai et al. (2025) は、マスク拡散モデルが自己回帰モデルよりも繰り返しデータをうまく活用すると報告している。この問題では意図的にデータを繰り返す。文字レベルのウィンドウ `d2l.TimeMachine(batch_size=64, num_steps=128, tokenization='char', num_train=100000, num_val=3000)`（8.2 節）を用いよ。その 100,000 個の訓練ウィンドウは、最初の 100,000 文字位置（『The Time Machine』の冒頭）から始まるため、2,000 回の更新を 64 個の 128 文字ウィンドウからなるミニバッチで行うと、これらの位置を約 160 回通過することになる。

   1. `GPT`（11.2 節）をこれらのウィンドウ上で 2,000 回更新にわたりその訓練レシピ `LMOptimizer` で訓練し、訓練損失と検証損失を 200 回更新ごとに記録せよ。検証損失はどの更新回数で最小に達し、その後 2 つの損失はどう振る舞うか。
   2. ノイズ除去器を `d2l.GPT` のサブクラスとして構築せよ。トークン数を `vocab_size + 1`（最後のトークンがマスク）とし、本節の議論が許すとおり時刻を受け取って無視する順伝播を持たせること。（ヒント：因果マスクを外すには、入れ子クラス `CausalAttention` をオーバーライドし、フレームワークの融合アテンションをマスクなしで呼び出せばよい。）これを `masked_loss` と線形の `alpha_bar`（$T = 128$ ステップ）を用いて 2,000 回更新にわたり訓練せよ。
   3. 200 回更新ごとに、検証ウィンドウ上で 1 ウィンドウあたり複数のマスクを用いて文字あたりビット数での上界を推定せよ。これを文字あたりビット数で表した `GPT` の検証損失と並べてプロットし、各曲線の最良値とそれに到達する更新回数を比較せよ。ノイズ除去器の上界は、`GPT` の損失よりも、`GPT` が過学習を始める前のすべての更新回数において十分上に留まるはずである。この差はスケーリング研究が測定しているコストそのものであり、バグではない。
   4. ノイズ除去器の訓練を 8,000 回更新まで継続せよ（2,000 回更新の時点でまだ上界が下がり続けている場合）。どちらのモデルの方が遅く過学習を始めるか。また、ランダムにマスクされたウィンドウ上で訓練することは、何度も見たウィンドウからノイズ除去器が暗記できる内容をどのように変えるか。
   5. $S \in \{128, 32, 8\}$ ステップでウィンドウをサンプリングし、`GPT` のサンプルと比較せよ。少数ステップのサンプルはどこで破綻するか。その破綻を演習問題 2c と関連付けて説明せよ。

17.10 関連文献

本節では、本章で展開した手法を発展させる専門書、チュートリアル、参照実装、および一次文献をまとめる。

17.10.1 専門書、講義、コード

- **The Principles of Diffusion Models** (Lai et al. 2025) は、変分・スコアベース・フローベースの各経路に沿って拡散モデルを導出し、それらがどこで合流するかを示している。
- MIT の講義 6.S184 の講義ノート、講義、ラボ (Holderrieth and Erives 2025) は、フローマッチング、潜在拡散 Transformer、および離散拡散（discrete diffusion）を扱っている。

- **Diffusion Meets Flow Matching** (Gao et al. 2024) は、スケジュールの再スケーリングのもとで DDIM とオイラー法のフローマッチングサンプラが等価であることを、対話的な図を用いて解説している。
- **Flow Matching Guide and Code** (Lipman et al. 2024) とそのライブラリは、17.2 節に対する完全な証明と参照コードを提供している。
- **Step-by-Step Diffusion** (Nakkiran et al. 2025) は、確率解析（stochastic calculus）を用いず初等確率論のみから、決定論的サンプラと確率的サンプラ（17.3 節および 17.5 節）を導出している。
- Sander Dieleman のブログでは、周波数空間でのノイズ除去 (Dieleman 2024)、潜在表現 (Dieleman 2025)、フロー写像 (Dieleman 2026b)、および言語に対する連続拡散と離散拡散の比較 (Dieleman 2026a) が検討されている。
- 参照実装としては、DDPM 公式リポジトリ (Ho et al. 2020)、Score SDE リポジトリ (Y. Song, Sohl-Dickstein, et al. 2021)、および MDLM リポジトリ (Sahoo et al. 2024) がある。
- **Deep Generative Models** 講義 (Kuleshov 2023)、Yang Song のスコアベースモデルのチュートリアル (Song 2021)、Lilian Weng の **What Are Diffusion Models?** (Weng 2021)、および Luo (2022) は、スコアマッチングと変分拡散に関する平易な解説を提供している。

一次論文のなかでも、次の 5 編はとりわけ基礎をなすものである。Karras et al. (2022)（EDM）はサンプラ、前処理（preconditioning）、ノイズスケジュールを切り分け、Esser et al. (2024)（Stable Diffusion 3）はレクティファイドフローとマルチモーダル拡散 Transformer を組み合わせた。また、C. Lu et al. (2025)（DPM-Solver++）はガイダンス下での高次ソルバを解析し、Lu and Song (2025)（sCM）は大規模環境における連続時間一貫性モデル（consistency model）を安定化し、Sahoo et al. (2024)（MDLM）はマスク付き離散拡散の見通しのよい定式化を与えている。隣接するトピックは 16.8 節（敵対的蒸留および分布整合蒸留、FID 評価）ならびに 38.8 節（指数積分器、スコア誤差の上界、シュレーディンガーブリッジ）で扱う。

17.10.2 エネルギーベースモデルとスコアマッチング

エネルギーベースモデルの訓練：@Song.Kingma.2021 は、正規化されていない密度に対する最尤法による訓練を概観している。コントラスティブダイバージェンスはデータ点を初期値として短い MCMC 連鎖を開始するため (Hinton 2002)、離れたモード間での混合が妨げられるが（17.1 節）、持続的コントラスティブダイバージェンス（persistent contrastive divergence）はパラメータ更新をまたいで連鎖の状態を保持する (Tieleman 2008; Du and Mordatch 2019)。

ノイズ対照推定：微分によって正規化定数を消去する代わりに、Gutmann and Hyvärinen (2012) は正規化定数を学習可能なパラメータとして扱い、既知のノイズ分布に対するロジスティック回帰によってモデルを訓練する。

スコアマッチングの限界：低密度の谷を隔てたモードの重みに対してフィッシャーダイバージェンスが不感になる性質（17.1 節）は Wenliang and Kanagawa (2020) によって解析されており、Koehler et al. (2022) は等周定数（isoperimetric constant）を用いてスコアマッチングの統計的効率に境界を与えている。

ノイズラダーのヒューリスティクス：@Song.Ermon.2020 は、幾何ノイズラダー（17.2 節）に対して 4 つの実践的な規則を与えた。すなわち、最大ノイズスケールを訓練データ間の最大ペアワイズ距離に設定すること、高確率シェルが重なり合うほど隣接レベルを十分に近づけて選ぶこと（演習問題 4、17.2 節）、ランジュバン法のステップサイズを調整すること、および重みに指数移動平均を適用することである。

経験ベイズ：トゥイーディーの公式は経験ベイズ推定（empirical Bayes estimation）に端を発しており (Robbins 1956)、Efron (2011) が高次元推定における選択バイアスの補正にこれを用いる手法を広めた。

17.10.3 画像ネットワークとガイダンス

大規模な U-Net アーキテクチャ：@Dhariwal.Nichol.2021（ADM）は、DDPM の U-Net に多重解像度アテンション、残差リサンプリング、および適応グループ正規化を追加した。Karras, Aittala, Lehtinen, et al. (2024)（EDM2）は、活性化値と重みの大きさを保つように U-Net を再設計し、周期的なチェックポイントからの事後的な EMA 再構成を導入した。

飽和のないガイダンス：大きなガイダンス強度はピクセル強度を飽和させることがある（演習問題 5、17.4 節）。Sadat et al. (2025) は、ガイダンスによる更新を条件付き予測に平行な成分と直交する成分へ分解し、平行成分を減衰させることで飽和を防いでいる。

空間的条件付け：ControlNet (Zhang et al. 2023) は、ゼロ初期化された層を介して接続したエンコーダの訓練可能なコピーを用いることで、事前学習済みのテキスト画像生成モデルを空間マップ（エッジ、深度、姿勢）で条件付ける。

17.10.4 評価、暗記、汎化

プロンプトアライメント：フレシェ距離はプロンプトへの忠実度ではなく分布の類似度を測るため、CLIPScore (Hessel et al. 2021) は共通埋め込みにおける画像とキャプションの類似度を測定し、一方 GenEval (Ghosh et al. 2023) は物体検出器を用いて物体の数、位置、属性を検証する。

暗記：@Carlini.Hayes.Nasr.ea.2023 は大規模なテキスト画像生成モデルから訓練画像とほぼ同一の重複画像を抽出しており、Biroli et al. (2024) は逆過程においてクラスの確定（種分化、speciation）と訓練データ点の暗記（崩壊、collapse）に対応する異なる力学的レジームを特定している。

ノイズ除去器の汎化：ノイズ除去目的関数の厳密な経験最小点は有限の訓練セットを暗記してしまうが、Bertrand et al. (2025) は目標ノイズだけでは汎化を説明できないことを示している。互いに素な部分集合で訓練されたノイズ除去器はほぼ同一のスコア場を学習し (Kadkhodaie et al. 2024)、それは主として畳み込みの局所性と同変性によって規定される (Kamb and Ganguli 2025)。サーベイについては Massias and Bertrand (2026) を参照のこと。

17.10.5 サンプラ、編集、逆問題

ガイダンス付きサンプリングのためのソルバ：大きなガイダンス強度は高次のノイズ予測ソルバを不安定にする（演習問題 3、17.5 節）。そこで DPM-Solver++ (C. Lu et al. 2025) は代わりにクリーン画像の推定値を用いて多段更新をパラメータ化し、UniPC (W. Zhao et al. 2023) は追加計算コストのない修正子（corrector）ステップを加えている。

ODE ソルバにおける確率性：@Karras.Aittala.Aila.ea.2022 はホイン法の各ステップの前に少量のノイズを注入し、一方 Restart サンプリング (Xu et al. 2023) は決定論的な ODE 積分とより大きな順方向ノイズのジャンプを交互に繰り返すことで、蓄積した離散化誤差を縮小させる。

厳密な反転：DDIM 反転（DDIM inversion、17.5 節）は分類器なしガイダンスのもとで離散化誤差を蓄積するため、ヌルテキスト反転（null-text inversion）(Mokady et al. 2023) はガイダンスなしの軌跡の周囲で無条件埋め込みを最適化し、一方 EDICT (Wallace et al. 2023) は結合されたアフィン変換を用いて厳密な可逆性を実現する。

再訓練なしの逆問題：ブラー除去や超解像などの線形逆問題に対し、Diffusion Posterior Sampling（DPS）(Chung et al. 2023) は事後平均 $\hat{\mathbf{x}}_0(\mathbf{x}_t)$ において測定尤度の勾配を近似し、DDRM (Kawar et al. 2022) は測定作用素のスペクトル領域で動作する。

17.10.6 スケジュール、重み付け、尤度

学習される逆過程の分散：コサインスケジュール（17.2 節）と並び、Nichol and Dhariwal (2021) は逆過程の分散を $\tilde{\beta}_k$ と β_k の間の学習される補間としてパラメータ化し、変分境界を用いて訓練している。

Min-SNR 重み付け：@Hang.Gu.Li.ea.2023 は、クリーン画像の二乗誤差に $\min(\mathrm{SNR}_t, \gamma)$ の重みを掛けることで（等価的にノイズ誤差へ $\min(1, \gamma e^{-\lambda})$ の重みを掛けることに相当）、低ノイズのステップが勾配更新を支配するのを防いでいる。

尤度最適化：変分拡散モデル（variational diffusion model、VDM）(Kingma et al. 2021) は ELBO 推定量の分散を最小化するようにノイズスケジュールを最適化する。また、Y. Song, Durkan, et al. (2021) は連続時間スコアマッチングに対する尤度重み付けスケジュールを導出し、Zheng et al. (2023) は確率フロー ODE に対する厳密な尤度訓練と評価を改善している。

17.10.7 潜在表現、テキスト、動画

テキスト条件付け：大規模なテキスト画像生成モデルは、凍結した CLIP (Radford et al. 2021) や T5 (Raffel et al. 2020) のテキストエンコーダを用い、クロスアテンション（17.7 節）または画像・テキスト結合自己アテンション (Esser et al. 2024) を介して条件付けを行う。Saharia et al. (2022) は、テキストエンコーダをスケールする方が、U-Net をスケールするよりもプロンプトアライメントを大きく向上させることを示した。

表現アライメントとオートエンコーダ：REPA (S. Yu et al. 2025) は、中間の隠れ状態を事前学習済みの視覚エンコーダと整合させることで潜在 DiT の訓練を高速化する。Deep Compression Autoencoders（DC-AE）(Chen et al. 2025) は空間方向のダウンサンプリングを $8\times$ 超へと拡大し、表現オートエンコーダ（representation autoencoder）(Zheng et al. 2026) は凍結したセマンティックエンコーダ（DINO など）と学習されるデコーダを組み合わせる。

動画生成：動画拡散モデルは、3D 時空間オートエンコーダと、空間および時間にわたる分解型または結合型のアテンションを適用する (Ho et al. 2022; Blattmann et al. 2023; Polyak et al. 2024)。

17.10.8 少数ステップ生成

漸進的蒸留とガイダンス蒸留：漸進的蒸留（progressive distillation）(Salimans and Ho 2022) は教師モデルのステップ数を反復的に半減させる手法であり、低 SNR における数値安定性のために $\mathbf{v}$ 予測を導入した。Meng et al. (2023) は、分類器なしガイダンスをガイダンス強度 γ で条件付けられた単一パスへと蒸留している。

大規模環境における一貫性とフロー写像：改良型一貫性訓練（improved consistency training、iCT）(Song and Dhariwal 2024) は離散時間において教師モデルを不要にし、一方 sCM (Lu and Song 2025) は三角関数経路と接線正規化（tangent normalization）を用いて ImageNet スケールでの連続時間一貫性訓練を安定化する。Improved MeanFlow (Geng et al. 2026) は瞬間速度へ直接回帰し、Boffi et al. (2025) はオイラー形式、ラグランジュ形式、および合成形式のフロー写像恒等式にわたる自己蒸留を解析している。

17.10.9 離散拡散

代替カーネルと再マスキング：離散フローマッチング (Gat et al. 2024) は、単体上の確率経路を通じて離散生成を定式化する。一様破壊カーネルを用いれば任意のステップでトークンを修正でき、Sahoo et al. (2025) は一様離散拡散をしきい値処理されたガウス拡散と結びつけている。

ReMDM (G. Wang et al. 2025) は吸収型拡散に再マスキングを導入し、Kim et al. (2025) は数独などの推論タスクにおいて確信度順のアンマスキングが高い効果を発揮することを実証している。

拡散言語モデル：LLaDA (Nie, Zhu, You, et al. 2025) は 8B のマスク付き拡散言語モデルをゼロから訓練し、一方 Dream (J. Ye et al. 2025) は自己回帰モデルのチェックポイントから初期化する。ブロック拡散（block diffusion）(Arriola et al. 2025) はブロック間の自己回帰生成とブロック内の並列拡散を組み合わせ、Fast-dLLM (Wu et al. 2026) は KV キャッシュの再利用（11.3 節）を追加している。なお K. Zheng, Chen, Mao, et al. (2025) は、32 ビットの Gumbel-max サンプリングがサンプリング温度を人為的に低下させ、サンプルエントロピーを犠牲にして報告される生成パープレキシティを実態より良く見せてしまうことに警鐘を鳴らしている。

17.10.10 事後学習と視覚・テキスト以外の応用

強化学習と選好最適化：DDPO (Black et al. 2024) と Flow-GRPO (Jie Liu et al. 2025) は、拡散モデルおよびフローモデルの確率的な逆遷移に対して方策勾配更新（15.8 節）を適用する。Diffusion-DPO (Wallace et al. 2024) は、計算困難なモデルの対数尤度を拡散モデルの ELBO で置き換えることにより、直接選好最適化を適合させている。

音声、ロボティクス、タンパク質：DiffWave (Kong et al. 2021) は生の音声波形に拡散を適用する。拡散方策（diffusion policy）(C. Chi et al. 2023) は視覚観測に条件付けられた多峰性のロボット行動軌跡をモデル化し、直接的な MSE 方策回帰におけるモード平均化の破綻を回避する。マルチモーダルなタンパク質モデル (Campbell et al. 2024) は、連続的な 3D 原子座標と離散的なアミノ酸配列を同時に拡散させる。

参考文献

Abadi, Martín, Paul Barham, Jianmin Chen, et al. 2016. "TensorFlow: A System for Large-Scale Machine Learning." **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)**, 265–83. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Achiam, Joshua. 2017. **Advanced Policy Gradient Methods**. Lecture, UC Berkeley CS 294-112: Deep Reinforcement Learning, https://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf.

Achiam, Joshua. 2018. **Spinning up in Deep RL**. Online, OpenAI; PyTorch rewrite Jan 2020, https://spinningup.openai.com/.

Agarwal, Alekh, Nan Jiang, Sham M Kakade, and Wen Sun. 2019. **Reinforcement Learning: Theory and Algorithms**. Working draft monograph, continuously revised, https://rltheorybook.github.io/.

Agarwal, Alekh, Sham M. Kakade, Jason D. Lee, and Gaurav Mahajan. 2021. "On the Theory of Policy Gradient Methods: Optimality, Approximation, and Distribution Shift." **Journal of Machine Learning Research** 22 (98): 1–76.

Agarwal, Rishabh, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. 2021. "Deep Reinforcement Learning at the Edge of the Statistical Precipice." **Advances in Neural Information Processing Systems** 34. https://arxiv.org/abs/2108.13264.

Agarwal, Rishabh, Nino Vieillard, Yongchao Zhou, et al. 2024. "On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes." **International Conference on Learning Representations**. https://arxiv.org/abs/2306.13649.

Agrawal, Shipra, and Navin Goyal. 2012. "Analysis of Thompson Sampling for the Multi-Armed Bandit Problem." **Proceedings of the 25th Annual Conference on Learning Theory**, Proceedings of machine learning research, vol. 23: 39.1–26. https://proceedings.mlr.press/v23/agrawal12.html.

Ahmadian, Arash, Chris Cremer, Matthias Gallé, et al. 2024. "Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs." **Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics**. https://arxiv.org/abs/2402.14740.

Ahn, Kwangjun, Byron Xu, Natalie Abreu, et al. 2025. “Dion: Distributed Orthonormalized Updates.” **arXiv Preprint arXiv:2504.05295**.

Ainslie, Joshua, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints.” **Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing**, 4895–901.

Ainsworth, Samuel K., Jonathan Hayase, and Siddhartha Srinivasa. 2023. “Git Re-Basin: Merging Models Modulo Permutation Symmetries.” **International Conference on Learning Representations**.

Akyürek, Ekin, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2023. “What Learning Algorithm Is in-Context Learning? Investigations with Linear Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2211.15661.

Alayrac, Jean-Baptiste, Jeff Donahue, Pauline Luc, et al. 2022. “Flamingo: A Visual Language Model for Few-Shot Learning.” **ArXiv:2204.14198**. https://arxiv.org/abs/2204.14198.

Albergo, Michael S., Nicholas M. Boffi, and Eric Vanden-Eijnden. 2023. “Stochastic Interpolants: A Unifying Framework for Flows and Diffusions.” **arXiv Preprint arXiv:2303.08797**.

Amari, Shun-ichi. 1998. “Natural Gradient Works Efficiently in Learning.” **Neural Computation** 10 (2): 251–76.

Amdahl, Gene M. 1967. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” **Proceedings of the AFIPS Spring Joint Computer Conference** 30: 483–85.

Ameisen, Emmanuel, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, et al. 2025. **Circuit Tracing: Revealing Computational Graphs in Language Models**. Transformer Circuits Thread, https://transformer-circuits.pub/2025/attribution-graphs/methods.html.

Amsel, Noah, David Persson, Christopher Musco, and Robert M. Gower. 2026. “The Polar Express: Optimal Matrix Sign Methods and Their Application to the Muon Algorithm.” **International Conference on Learning Representations**.

Anderson, Brian D. O. 1982. “Reverse-Time Diffusion Equation Models.” **Stochastic Processes and Their Applications** 12 (3): 313–26.

Anderson, Brian D. O., and John B. Moore. 2007. **Optimal Control: Linear Quadratic Methods**. Dover Publications.

Andrychowicz, Marcin, Anton Raichuk, Piotr Stańczyk, et al. 2021. “What Matters for on-Policy Deep Actor-Critic Methods? A Large-Scale Study.” **International Conference on Learning Representations**. https://arxiv.org/abs/2006.05990.

Ansel, Jason, Edward Yang, Horace He, et al. 2024. “PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation.” **29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**, 929–47. https://doi.org/10.1145/3620665.3640366.

Arjovsky, Martin, and Léon Bottou. 2017. “Towards Principled Methods for Training Generative Adversarial Networks.” **International Conference on Learning Representations**.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou. 2017. “Wasserstein Generative Adversarial Networks.” **Proceedings of the 34th International Conference on Machine Learning**, 214–23.

Arora, Sanjeev, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. 2017. “Generalization and Equilibrium in Generative Adversarial Nets (GANs).” **International Conference on Machine Learning**, 224–32.

Arora, Simran, Sabri Eyuboglu, Aman Timalsina, et al. 2024. “Zoology: Measuring and Improving Recall in Efficient Language Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2312.04927.

Arora, Simran, Sabri Eyuboglu, Michael Zhang, et al. 2024. “Simple Linear Attention Language Models Balance the Recall-Throughput Tradeoff.” **International Conference on Machine Learning**.

Arriola, Marianne, Aaron Gokaslan, Justin T. Chiu, et al. 2025. “Block Diffusion: Interpolating Between Autoregressive and Diffusion Language Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2503.09573.

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. “Finite-Time Analysis of the Multiarmed Bandit Problem.” **Machine Learning** 47 (2-3): 235–56.

Austin, Jacob, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. 2021. “Structured Denoising Diffusion Models in Discrete State-Spaces.” **Advances in Neural Information Processing Systems** 34.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2015. “Neural Machine Translation by Jointly Learning to Align and Translate.” **International Conference on Learning Representations**. https://doi.org/10.48550/arXiv.1409.0473.

Baird, Leemon. 1995. “Residual Algorithms: Reinforcement Learning with Function Approximation.” **International Conference on Machine Learning**, 30–37.

Bakker, Bram. 2001. “Reinforcement Learning with Long Short-Term Memory.” **Advances in Neural Information Processing Systems** 14.

Ball, Philip J., Laura Smith, Ilya Kostrikov, and Sergey Levine. 2023. “Efficient Online Reinforcement Learning with Offline Data.” **International Conference on Machine Learning**, 1577–94.

Bao, Fan, Chongxuan Li, Jun Zhu, and Bo Zhang. 2022. “Analytic-DPM: An Analytic Esti-

mate of the Optimal Reverse Variance in Diffusion Probabilistic Models." **International Conference on Learning Representations**. https://arxiv.org/abs/2201.06503.

Barbero, Federico, Álvaro Arroyo, Xiangming Gu, et al. 2025. "Why Do LLMs Attend to the First Token?" **Conference on Language Modeling (COLM)**. https://arxiv.org/abs/2504.02732.

Barbero, Federico, Alex Vitvitskyi, Christos Perivolaropoulos, Razvan Pascanu, and Petar Veličković. 2025. "Round and Round We Go! What Makes Rotary Positional Encodings Useful?" **International Conference on Learning Representations**. https://arxiv.org/abs/2410.06205.

Barto, Andrew G, Richard S Sutton, and Charles W Anderson. 1983. "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems." **IEEE Transactions on Systems, Man, and Cybernetics**, no. 5: 834–46.

Beck, Maximilian, Korbinian Pöppel, Phillip Lippe, et al. 2025. "xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference." **arXiv Preprint arXiv:2503.13427**.

Beck, Maximilian, Korbinian Pöppel, Markus Spanring, et al. 2024. "xLSTM: Extended Long Short-Term Memory." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2405.04517.

Behrouz, Ali, Peilin Zhong, and Vahab Mirrokni. 2025. "Titans: Learning to Memorize at Test Time." **arXiv Preprint arXiv:2501.00663**.

Bellemare, Marc G, Will Dabney, and Rémi Munos. 2017. "A Distributional Perspective on Reinforcement Learning." **International Conference on Machine Learning**, 449–58.

Bellemare, Marc G., Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. 2016. "Unifying Count-Based Exploration and Intrinsic Motivation." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/1606.01868.

Bellman, Richard. 1952. "On the Theory of Dynamic Programming." **Proceedings of the National Academy of Sciences** 38 (8): 716–19. https://doi.org/10.1073/pnas.38.8.716.

Bellman, Richard. 1957a. "A Markovian Decision Process." **Journal of Mathematics and Mechanics** 6 (5): 679–84. http://www.jstor.org/stable/24900506.

Bellman, Richard. 1957b. **Dynamic Programming**. Princeton University Press.

Beltagy, Iz, Matthew E Peters, and Arman Cohan. 2020. "Longformer: The Long-Document Transformer." **ArXiv:2004.05150**. https://arxiv.org/abs/2004.05150.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent. 2013. "Representation Learning: A Review and New Perspectives." **IEEE Transactions on Pattern Analysis and Machine Intelligence** 35 (8): 1798–828.

Bergsma, Shane, Nolan Dey, Gurpreet Gosal, Gavia Gray, Daria Soboleva, and Joel Hestness. 2025. "Power Lines: Scaling Laws for Weight Decay and Batch Size in LLM Pre-Training."

Advances in Neural Information Processing Systems. https://arxiv.org/abs/2505.13738.

Bergstra, James, Olivier Breuleux, Frédéric Bastien, et al. 2010. "Theano: A CPU and GPU Math Compiler in Python." **Proc. 9th Python in Science Conference** 1: 3–10. https://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf.

Bernstein, Jeremy. 2025. **Deriving Muon**. Https://jeremybernste.in/writing/deriving-muon.

Bernstein, Jeremy, and Laker Newhouse. 2024. "Old Optimizer, New Norm: An Anatomy." **arXiv Preprint arXiv:2409.20325**.

Bertrand, Quentin, Anne Gagneux, Mathurin Massias, and Rémi Emonet. 2025. "On the Closed-Form of Flow Matching: Generalization Does Not Arise from Target Stochasticity." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2506.03719.

Bertsekas, Dimitri P. 2025. **A Course in Reinforcement Learning**. 2nd ed. Athena Scientific.

Besiroglu, Tamay, Ege Erdil, Matthew Barnett, and Josh You. 2024. "Chinchilla Scaling: A Replication Attempt." **arXiv Preprint arXiv:2404.10102**.

Bhatt, Aditya, Daniel Palenicek, Boris Belousov, et al. 2024. "CrossQ: Batch Normalization in Deep Reinforcement Learning for Greater Sample Efficiency and Simplicity." **International Conference on Learning Representations**.

Bick, Aviv, Kevin Y. Li, Eric P. Xing, J. Zico Kolter, and Albert Gu. 2024. "Transformers to SSMs: Distilling Quadratic Knowledge to Subquadratic Models." **Advances in Neural Information Processing Systems**.

Bińkowski, Mikołaj, Danica J. Sutherland, Michael Arbel, and Arthur Gretton. 2018. "Demystifying MMD GANs." **International Conference on Learning Representations**. https://arxiv.org/abs/1801.01401.

Biroli, Giulio, Tony Bonnaire, Valentin de Bortoli, and Marc Mézard. 2024. "Dynamical Regimes of Diffusion Models." **Nature Communications** 15: 9957. https://arxiv.org/abs/2402.18491.

Black Forest Labs. 2024. **FLUX.1 [Schnell] Model Card**. Https://huggingface.co/black-forest-labs/FLUX.1-schnell.

Black, Kevin, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. 2024. "Training Diffusion Models with Reinforcement Learning." **International Conference on Learning Representations**. https://arxiv.org/abs/2305.13301.

Blattmann, Andreas, Tim Dockhorn, Sumith Kulal, et al. 2023. "Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets." **arXiv Preprint arXiv:2311.15127**. https://arxiv.org/abs/2311.15127.

Blau, Yochai, and Tomer Michaeli. 2018. “The Perception-Distortion Tradeoff.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 6228–37.

Blelloch, Guy E. 1990. **Prefix Sums and Their Applications**. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University. https://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html.

Boffi, Nicholas M., Michael S. Albergo, and Eric Vanden-Eijnden. 2025. “How to Build a Consistency Model: Learning Flow Maps via Self-Distillation.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2505.18825.

Bordelon, Blake, Lorenzo Noci, Mufan Bill Li, Boris Hanin, and Cengiz Pehlevan. 2024. “Depthwise Hyperparameter Transfer in Residual Networks: Dynamics and Scaling Limit.” **International Conference on Learning Representations**. https://arxiv.org/abs/2309.16620.

Bradley, Ralph Allan, and Milton E Terry. 1952. “Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons.” **Biometrika** 39 (3/4): 324–45. https://doi.org/10.1093/biomet/39.3-4.324.

Brandfonbrener, David, Alberto Bietti, Jacob Buckman, Romain Laroche, and Joan Bruna. 2022. “When Does Return-Conditioned Supervised Learning Work for Offline Reinforcement Learning?” **Advances in Neural Information Processing Systems** 35.

Bricken, Trenton, Adly Templeton, Joshua Batson, et al. 2023. **Towards Monosemanticity: Decomposing Language Models with Dictionary Learning**. Transformer Circuits Thread, https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Brock, Andrew, Jeff Donahue, and Karen Simonyan. 2019. “Large Scale GAN Training for High Fidelity Natural Image Synthesis.” **International Conference on Learning Representations**. https://arxiv.org/abs/1809.11096.

Brown, Noam, and Tuomas Sandholm. 2017. “Libratus: The Superhuman AI for No-Limit Poker.” **IJCAI**, 5226–28. https://doi.org/10.24963/ijcai.2017/772.

Brown, Tom, Benjamin Mann, Nick Ryder, et al. 2020. “Language Models Are Few-Shot Learners.” **Advances in Neural Information Processing Systems** 33: 1877–901. https://arxiv.org/abs/2005.14165.

Bubeck, Sébastien, and Nicolò Cesa-Bianchi. 2012. “Regret Analysis of Stochastic and Nonstochastic Multi-Armed Bandit Problems.” **Foundations and Trends in Machine Learning** 5 (1): 1–122. https://doi.org/10.1561/2200000024.

Burda, Yuri, Harrison Edwards, Amos Storkey, and Oleg Klimov. 2019. “Exploration by Random Network Distillation.” **International Conference on Learning Representations**. https://arxiv.org/abs/1810.12894.

Campbell, Andrew, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. 2022. “A Continuous Time Framework for Discrete Denoising Models.” **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/22

05.14987.

Campbell, Andrew, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. 2024. “Generative Flows on Discrete State-Spaces: Enabling Multimodal Flows with Applications to Protein Co-Design.” **International Conference on Machine Learning**.

Cesista, Franz Louis, Jiacheng You, and Keller Jordan. 2025. **Squeezing 1–2% Efficiency Gains Out of Muon by Optimizing the Newton–Schulz Coefficients**. Http://leloykun.github.io/ponder/muon-opt-coeffs/.

Chang, Huiwen, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. 2022. “MaskGIT: Masked Generative Image Transformer.” **IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 11315–25. https://arxiv.org/abs/2202.04200.

Chen, Charlie, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. “Accelerating Large Language Model Decoding with Speculative Sampling.” **arXiv Preprint arXiv:2302.01318**.

Chen, Junyu, Han Cai, Junsong Chen, et al. 2025. “Deep Compression Autoencoder for Efficient High-Resolution Diffusion Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2410.10733.

Chen, Lili, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, et al. 2021. “Decision Transformer: Reinforcement Learning via Sequence Modeling.” **Advances in Neural Information Processing Systems** 34.

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. “Evaluating Large Language Models Trained on Code.” **arXiv Preprint arXiv:2107.03374**.

Chen, Shouyuan, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. “Extending Context Window of Large Language Models via Positional Interpolation.” **arXiv Preprint arXiv:2306.15595**.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. “Training Deep Nets with Sublinear Memory Cost.” **arXiv Preprint arXiv:1604.06174**.

Chen, Xinyue, Che Wang, Zijian Zhou, and Keith Ross. 2021. “Randomized Ensembled Double Q-Learning: Learning Fast Without a Model.” **International Conference on Learning Representations**. https://arxiv.org/abs/2101.05982.

Cheng, Jianpeng, Li Dong, and Mirella Lapata. 2016. “Long Short-Term Memory-Networks for Machine Reading.” **Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing**, 551–61. https://doi.org/10.18653/v1/d16-1053.

Chi, Cheng, Siyuan Feng, Yilun Du, et al. 2023. “Diffusion Policy: Visuomotor Policy Learning via Action Diffusion.” **Robotics: Science and Systems**. https://arxiv.org/abs/2303.04137.

Chi, Ta-Chung, Ting-Han Fan, Alexander I. Rudnicky, and Peter J. Ramadge. 2023. “Dissecting Transformer Length Extrapolation via the Lens of Receptive Field Analysis.” **Pro-**

ceedings of the 61st Annual Meeting of the Association for Computational Linguistics. https://arxiv.org/abs/2212.10356.

Child, Rewon, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. "Generating Long Sequences with Sparse Transformers." **ArXiv:1904.10509**. https://arxiv.org/abs/1904.10509.

Cho, Kyunghyun, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches." **ArXiv:1409.1259**. https://arxiv.org/abs/1409.1259.

Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, et al. 2014. "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation." **Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing**, 1724–34. https://arxiv.org/abs/1406.1078.

Chong, Min Jin, and David Forsyth. 2020. "Effectively Unbiased FID and Inception Score and Where to Find Them." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 6070–79. https://arxiv.org/abs/1911.07023.

Choromanski, Krzysztof, Valerii Likhosherstov, David Dohan, et al. 2021. "Rethinking Attention with Performers." **International Conference on Learning Representations**.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, et al. 2022. "PaLM: Scaling Language Modeling with Pathways." **ArXiv:2204.02311**. https://arxiv.org/abs/2204.02311.

Christiano, Paul, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." **Advances in Neural Information Processing Systems** 30.

Christodoulou, Petros. 2019. "Soft Actor-Critic for Discrete Action Settings." **arXiv Preprint arXiv:1910.07207**.

Chu, Casey, Andrey Zhmoginov, and Mark Sandler. 2017. "CycleGAN, a Master of Steganography." **arXiv Preprint arXiv:1712.02950**. https://arxiv.org/abs/1712.02950.

Chua, Kurtland, Roberto Calandra, Rowan McAllister, and Sergey Levine. 2018. "Deep Reinforcement Learning in a Handful of Trials Using Probabilistic Dynamics Models." **Advances in Neural Information Processing Systems** 31.

Chung, Hyungjin, Jeongsol Kim, Michael T. Mccann, Marc L. Klasky, and Jong Chul Ye. 2023. "Diffusion Posterior Sampling for General Noisy Inverse Problems." **International Conference on Learning Representations**. https://arxiv.org/abs/2209.14687.

Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." **ArXiv:1412.3555**. https://arxiv.org/abs/1412.3555.

Cohen, Jeremy M., Behrooz Ghorbani, Shankar Krishnan, et al. 2022. "Adaptive Gradient Methods at the Edge of Stability." **arXiv Preprint arXiv:2207.14484**. https://arxiv.org/ab

s/2207.14484.

Cohen, Jeremy M., Simran Kaur, Yuanzhi Li, J. Zico Kolter, and Ameet Talwalkar. 2021. "Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability." **International Conference on Learning Representations**.

Conmy, Arthur, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. "Towards Automated Circuit Discovery for Mechanistic Interpretability." **Advances in Neural Information Processing Systems** 36. https://arxiv.org/abs/2304.14997.

Crowson, Katherine, Stefan Andreas Baumann, Alex Birch, Tanishq Mathew Abraham, Daniel Z. Kaplan, and Enrico Shippole. 2024. "Scalable High-Resolution Pixel-Space Image Synthesis with Hourglass Diffusion Transformers." **Proceedings of the 41st International Conference on Machine Learning**, Proceedings of machine learning research, vol. 235: 9550–75.

Cunningham, Hoagy, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. 2024. "Sparse Autoencoders Find Highly Interpretable Features in Language Models." **International Conference on Learning Representations**. https://arxiv.org/abs/2309.08600.

D'Angelo, Francesco, Maksym Andriushchenko, Aditya Varre, and Nicolas Flammarion. 2024. "Why Do We Need Weight Decay in Modern Deep Learning?" **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2310.04415.

D'Oro, Pierluca, Max Schwarzer, Evgenii Nikishin, Pierre-Luc Bacon, Marc G. Bellemare, and Aaron Courville. 2023. "Sample-Efficient Reinforcement Learning by Breaking the Replay Ratio Barrier." **International Conference on Learning Representations**.

Dahl, George E., Frank Schneider, Zachary Nado, et al. 2023. "Benchmarking Neural Network Training Algorithms." **arXiv Preprint arXiv:2306.07179**.

Dai, Damai, Chengqi Deng, Chenggang Zhao, et al. 2024. "DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models." **arXiv Preprint arXiv:2401.06066**.

Damian, Alex, Eshaan Nichani, and Jason D. Lee. 2023. "Self-Stabilization: The Implicit Bias of Gradient Descent at the Edge of Stability." **International Conference on Learning Representations**.

Dao, Tri. 2024. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." **International Conference on Learning Representations**. https://arxiv.org/abs/2307.08691.

Dao, Tri, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness." **Advances in Neural Information Processing Systems**.

Dao, Tri, and Albert Gu. 2024. "Transformers Are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality." **International Conference on Ma-**

chine Learning, 10041–71. https://arxiv.org/abs/2405.21060.

Daskalakis, Constantinos, Andrew Ilyas, Vasilis Syrgkanis, and Haoyang Zeng. 2018. "Training GANs with Optimism." **International Conference on Learning Representations**.

Dauphin, Yann N., Angela Fan, Michael Auli, and David Grangier. 2017. "Language Modeling with Gated Convolutional Networks." **International Conference on Machine Learning**, 933–41.

Dauphin, Yann N., Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization." **Advances in Neural Information Processing Systems** 27: 2933–41.

De, Soham, Samuel L. Smith, Anushan Fernando, et al. 2024. "Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models." **arXiv Preprint arXiv:2402.19427**.

Dean, Jeffrey, Greg S Corrado, Rajat Monga, et al. 2012. "Large Scale Distributed Deep Networks." **Proceedings of the 25th International Conference on Neural Information Processing Systems, Volume 1**, 1223–31. https://doi.org/10.5555/2999134.2999271.

DeepSeek-AI. 2024. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model." **arXiv Preprint arXiv:2405.04434**.

DeepSeek-AI. 2025. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." **Nature**, ahead of print. https://doi.org/10.1038/s41586-025-09422-z.

DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, et al. 2024. "DeepSeek LLM: Scaling Open-Source Language Models with Longtermism." **arXiv Preprint arXiv:2401.02954**.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, et al. 2024. "DeepSeek-V3 Technical Report." **arXiv Preprint arXiv:2412.19437**.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, et al. 2025. "DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models." **arXiv Preprint arXiv:2512.02556**.

Defazio, Aaron, Xingyu Alice Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. 2024. "The Road Less Scheduled." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2405.15682.

Dehghani, Mostafa, Josip Djolonga, Basil Mustafa, et al. 2023. "Scaling Vision Transformers to 22 Billion Parameters." **International Conference on Machine Learning**, 7480–512.

Deshpande, Ishan, Ziyu Zhang, and Alexander G. Schwing. 2018. "Generative Modeling Using the Sliced Wasserstein Distance." **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**.

Dettmers, Tim, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022. "8-Bit Optimizers

via Block-Wise Quantization." **International Conference on Learning Representations**. https://arxiv.org/abs/2110.02861.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." **ArXiv:1810.04805**. https://arxiv.org/abs/1810.04805.

Dey, Nolan, Quentin Anthony, and Joel Hestness. 2024. **The Practitioner's Guide to the Maximal Update Parameterization**. Https://blog.eleuther.ai/mutransfer/.

Dey, Nolan, Gurpreet Gosal, Zhiming Chen, et al. 2023. "Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster." **arXiv Preprint arXiv:2304.03208**.

Dey, Nolan, Bin Claire Zhang, Lorenzo Noci, et al. 2025. "Don't Be Lazy: CompleteP Enables Compute-Efficient Deep Transformers." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2505.01618.

Dhariwal, Prafulla, and Alex Nichol. 2021. "Diffusion Models Beat GANs on Image Synthesis." **Advances in Neural Information Processing Systems** 34.

Dieleman, Sander. 2024. **Diffusion Is Spectral Autoregression**. Blog post, 2 September 2024, https://sander.ai/2024/09/02/spectral-autoregression.html.

Dieleman, Sander. 2025. **Generative Modelling in Latent Space**. Blog post, 15 April 2025, https://sander.ai/2025/04/15/latents.html.

Dieleman, Sander. 2026a. **Continuous Diffusion Language Models**. Blog post, 24 August 2026, https://sander.ai/2026/08/24/continuous-dlms.html.

Dieleman, Sander. 2026b. **Learning the Integral of a Diffusion Model**. Blog post, 6 May 2026, https://sander.ai/2026/05/06/flow-maps.html.

Ding, Ming, Zhuoyi Yang, Wenyi Hong, et al. 2021. "CogView: Mastering Text-to-Image Generation via Transformers." **Advances in Neural Information Processing Systems** 34.

Ding, Yiran, Li Lyna Zhang, Chengruidong Zhang, et al. 2024. "LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens." **International Conference on Machine Learning**, Proceedings of machine learning research, vol. 235: 11091–104. https://arxiv.org/abs/2402.13753.

Dinh, Laurent, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. 2017. "Sharp Minima Can Generalize for Deep Nets." **Proceedings of the 34th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 70: 1019–28. https://proceedings.mlr.press/v70/dinh17b.html.

Dong, Juechu, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2025. "Flex Attention: A Programming Model for Generating Optimized Attention Kernels." **Proceedings of Machine Learning and Systems (MLSys)**. https://arxiv.org/abs/2412.05496.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, et al. 2021. “An Image Is Worth 16 x 16 Words: Transformers for Image Recognition at Scale.” **International Conference on Learning Representations**. https://openreview.net/forum?id=YicbFdNTTy.

Dowson, D. C., and B. V. Landau. 1982. “The Fréchet Distance Between Multivariate Normal Distributions.” **Journal of Multivariate Analysis** 12 (3): 450–55.

Draxler, Felix, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. 2018. “Essentially No Barriers in Neural Network Energy Landscape.” **Proceedings of the 35th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 80: 1309–18.

Du, Yilun, and Igor Mordatch. 2019. “Implicit Generation and Modeling with Energy Based Models.” **Advances in Neural Information Processing Systems** 32. https://arxiv.org/abs/1903.08689.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.” **Journal of Machine Learning Research** 12: 2121–59. https://jmlr.org/papers/v12/duchi11a.html.

Efron, Bradley. 2011. “Tweedie’s Formula and Selection Bias.” **Journal of the American Statistical Association** 106 (496): 1602–14.

Elhage, Nelson, Tristan Hume, Catherine Olsson, et al. 2022. “Toy Models of Superposition.” **Transformer Circuits Thread**.

Elhage, Nelson, Neel Nanda, Catherine Olsson, et al. 2021. “A Mathematical Framework for Transformer Circuits.” **Transformer Circuits Thread**.

Elsayed, Mohamed, Gautham Vasan, and A Rupam Mahmood. 2024. “Streaming Deep Reinforcement Learning Finally Works.” **arXiv Preprint arXiv:2410.14606**.

Emmons, Scott, Benjamin Eysenbach, Ilya Kostrikov, and Sergey Levine. 2022. “RvS: What Is Essential for Offline RL via Supervised Learning?” **International Conference on Learning Representations**.

Engstrom, Logan, Andrew Ilyas, Shibani Santurkar, et al. 2020. “Implementation Matters in Deep RL: A Case Study on PPO and TRPO.” **International Conference on Learning Representations**.

Entezari, Rahim, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. 2022. “The Role of Permutation Invariance in Linear Mode Connectivity of Neural Networks.” **International Conference on Learning Representations**.

Espeholt, Lasse, Hubert Soyer, Rémi Munos, et al. 2018. “IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.” **International Conference on Machine Learning**, 1407–16.

Esser, Patrick, Sumith Kulal, Andreas Blattmann, et al. 2024. “Scaling Rectified Flow Transformers for High-Resolution Image Synthesis.” **Proceedings of the 41st International**

Conference on Machine Learning, Proceedings of machine learning research, vol. 235: 12606–33. https://arxiv.org/abs/2403.03206.

Esser, Patrick, Robin Rombach, and Björn Ommer. 2021. "Taming Transformers for High-Resolution Image Synthesis." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**.

Even-Dar, Eyal, and Yishay Mansour. 2003. "Learning Rates for Q-Learning." **Journal of Machine Learning Research** 5: 1–25.

Fan, Angela, Mike Lewis, and Yann Dauphin. 2018. "Hierarchical Neural Story Generation." **Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics**, 889–98.

Farebrother, Jesse, Jordi Orbay, Quan Vuong, et al. 2024. "Stop Regressing: Training Value Functions via Classification for Scalable Deep RL." **International Conference on Machine Learning**. https://arxiv.org/abs/2403.03950.

Fedus, William, Barret Zoph, and Noam Shazeer. 2022. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." **Journal of Machine Learning Research** 23 (120): 1–39. https://arxiv.org/abs/2101.03961.

Feng, Leo, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadeghi. 2024. "Were RNNs All We Needed?" **arXiv Preprint arXiv:2410.01201**.

Foret, Pierre, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. 2021. "Sharpness-Aware Minimization for Efficiently Improving Generalization." **International Conference on Learning Representations**. https://arxiv.org/abs/2010.01412.

Frans, Kevin, Danijar Hafner, Sergey Levine, and Pieter Abbeel. 2025. "One Step Diffusion via Shortcut Models." **International Conference on Learning Representations**. https://arxiv.org/abs/2410.12557.

Frostig, Roy, Matthew James Johnson, and Chris Leary. 2018. "Compiling Machine Learning Programs via High-Level Tracing." In **Proceedings of Systems for Machine Learning**. https://mlsys.org/Conferences/2018/doc/19.pdf.

Fu, Daniel Y., Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. 2023. "Hungry Hungry Hippos: Towards Language Modeling with State Space Models." **International Conference on Learning Representations**. https://arxiv.org/abs/2212.14052.

Fujimoto, Scott, and Shixiang Shane Gu. 2021. "A Minimalist Approach to Offline Reinforcement Learning." **Advances in Neural Information Processing Systems** 34.

Fujimoto, Scott, Herke van Hoof, and David Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods." **International Conference on Machine Learning**, 1587–96.

Fujimoto, Scott, David Meger, and Doina Precup. 2019. "Off-Policy Deep Reinforcement

Learning Without Exploration." **International Conference on Machine Learning**, 2052–62.

Gallici, Matteo, Mattie Fellows, Benjamin Ellis, et al. 2025. "Simplifying Deep Temporal Difference Learning." **International Conference on Learning Representations**. https://arxiv.org/abs/2407.04811.

Gao, Leo, John Schulman, and Jacob Hilton. 2023. "Scaling Laws for Reward Model Overoptimization." **International Conference on Machine Learning**. https://arxiv.org/abs/2210.10760.

Gao, Ruiqi, Emiel Hoogeboom, Jonathan Heek, Valentin De Bortoli, Kevin P. Murphy, and Tim Salimans. 2024. **Diffusion Meets Flow Matching: Two Sides of the Same Coin**. Blog post, https://diffusionflow.github.io/.

Garg, Shivam, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. "What Can Transformers Learn in-Context? A Case Study of Simple Function Classes." **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2208.01066.

Garipov, Timur, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P. Vetrov, and Andrew Gordon Wilson. 2018. "Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs." **Advances in Neural Information Processing Systems** 31: 8789–98.

Gat, Itai, Tal Remez, Neta Shaul, et al. 2024. "Discrete Flow Matching." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2407.15595.

Gemma Team. 2024. "Gemma 2: Improving Open Language Models at a Practical Size." **arXiv Preprint arXiv:2408.00118**.

Gemma Team. 2025. "Gemma 3 Technical Report." **arXiv Preprint arXiv:2503.19786**.

Geng, Zhengyang, Mingyang Deng, Xingjian Bai, J. Zico Kolter, and Kaiming He. 2025. "Mean Flows for One-Step Generative Modeling." **arXiv Preprint arXiv:2505.13447**.

Geng, Zhengyang, Yiyang Lu, Zongze Wu, Eli Shechtman, J. Zico Kolter, and Kaiming He. 2026. "Improved Mean Flows: On the Challenges of Fastforward Generative Models." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/2512.02012.

Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. 2000. "Learning to Forget: Continual Prediction with LSTM." **Neural Computation** 12 (10): 2451–71. https://doi.org/10.1162/089976600300015015.

Ghadimi, Saeed, and Guanghui Lan. 2013. "Stochastic First- and Zeroth-Order Methods for Nonconvex Stochastic Programming." **SIAM Journal on Optimization** 23 (4): 2341–68. https://arxiv.org/abs/1309.5549.

Gholami, Amir, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. "AI and Memory Wall." **IEEE Micro** 44 (3): 33–39. https://doi.org/10.1109/MM.2024.3373763.

Ghorbani, Behrooz, Shankar Krishnan, and Ying Xiao. 2019. “An Investigation into Neural Net Optimization via Hessian Eigenvalue Density.” **Proceedings of the 36th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 97: 2232–41.

Ghosh, Dhruba, Hanna Hajishirzi, and Ludwig Schmidt. 2023. “GenEval: An Object-Focused Framework for Evaluating Text-to-Image Alignment.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2310.11513.

Gidel, Gauthier, Reyhane Askari Hemmat, Mohammad Pezeshki, et al. 2019. “Negative Momentum for Improved Game Dynamics.” **Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics**, Proceedings of machine learning research, vol. 89: 1802–11.

Givens, Clark R., and Rae Michael Shortt. 1984. “A Class of Wasserstein Metrics for Probability Distributions.” **Michigan Mathematical Journal** 31 (2): 231–40.

Godbole, Varun, George E. Dahl, Justin Gilmer, Christopher J. Shallue, and Zachary Nado. 2023. **Deep Learning Tuning Playbook**. Https://github.com/google-research/tuning_playbook.

Goel, Karan, Albert Gu, Chris Donahue, and Christopher Ré. 2022. “It’s Raw! Audio Generation with State-Space Models.” **International Conference on Machine Learning**. https://arxiv.org/abs/2202.09729.

Goodfellow, Ian. 2016. “NIPS 2016 Tutorial: Generative Adversarial Networks.” **arXiv Preprint arXiv:1701.00160**. https://arxiv.org/abs/1701.00160.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, et al. 2014. “Generative Adversarial Nets.” **Advances in Neural Information Processing Systems**, 2672–80. https://doi.org/10.5555/2969033.2969125.

Goyal, Priya, Piotr Dollár, Ross Girshick, et al. 2017. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.” **ArXiv:1706.02677**. https://arxiv.org/abs/1706.02677.

Grathwohl, Will, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. 2019. “FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models.” **International Conference on Learning Representations**.

Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. “The Llama 3 Herd of Models.” **arXiv Preprint arXiv:2407.21783**.

Graves, Alex. 2013. “Generating Sequences with Recurrent Neural Networks.” **ArXiv:1308.0850**. https://arxiv.org/abs/1308.0850.

Grazzi, Riccardo, Julien Siems, Jörg K. H. Franke, Arber Zela, Frank Hutter, and Massimiliano Pontil. 2025. “Unlocking State-Tracking in Linear RNNs Through Negative Eigenvalues.” **International Conference on Learning Representations**. https://arxiv.org/abs/2411.12537.

Greensmith, Evan, Peter L Bartlett, and Jonathan Baxter. 2004. “Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning.” **Journal of Machine Learning Research** 5: 1471–530.

Gretton, Arthur, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. “A Kernel Two-Sample Test.” **Journal of Machine Learning Research** 13: 723–73.

Griewank, Andreas, and Andrea Walther. 2008. **Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation**. 2nd ed. Society for Industrial; Applied Mathematics (SIAM). https://doi.org/10.1137/1.9780898717761.

Gu, Albert. 2023. “Modeling Sequences with Structured State Spaces.” PhD thesis, Stanford University.

Gu, Albert, and Tri Dao. 2023. “Mamba: Linear-Time Sequence Modeling with Selective State Spaces.” **arXiv Preprint arXiv:2312.00752**.

Gu, Albert, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. 2020. “HiPPO: Recurrent Memory with Optimal Polynomial Projections.” **Advances in Neural Information Processing Systems** 33: 1474–87.

Gu, Albert, Karan Goel, and Christopher Ré. 2022. “Efficiently Modeling Long Sequences with Structured State Spaces.” **International Conference on Learning Representations**. https://arxiv.org/abs/2111.00396.

Gu, Albert, Ankit Gupta, Karan Goel, and Christopher Ré. 2022. “On the Parameterization and Initialization of Diagonal State Space Models.” **Advances in Neural Information Processing Systems** 35: 35971–83.

Gu, Albert, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. 2023. “How to Train Your HiPPO: State Space Models with Generalized Orthogonal Basis Projections.” **International Conference on Learning Representations**. https://arxiv.org/abs/2206.12037.

Gu, Xiangming, Tianyu Pang, Chao Du, et al. 2025. “When Attention Sink Emerges in Language Models: An Empirical View.” **International Conference on Learning Representations**. https://arxiv.org/abs/2410.10781.

Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. “Improved Training of Wasserstein GANs.” **Advances in Neural Information Processing Systems** 30.

Guo, Han, Songlin Yang, Tarushii Goel, Eric P. Xing, Tri Dao, and Yoon Kim. 2025. “Log-Linear Attention.” **arXiv Preprint arXiv:2506.04761**.

Gupta, Vineet, Tomer Koren, and Yoram Singer. 2018. “Shampoo: Preconditioned Stochastic Tensor Optimization.” **Proceedings of the 35th International Conference on Machine Learning (ICML)**, 1842–50.

Gustafson, John L. 1988. “Reevaluating Amdahl’s Law.” **Communications of the ACM** 31 (5): 532–33.

Gutmann, Michael U., and Aapo Hyvärinen. 2012. “Noise-Contrastive Estimation of Unnormalized Statistical Models, with Applications to Natural Image Statistics.” **Journal of Machine Learning Research** 13: 307–61. https://www.jmlr.org/papers/v13/gutmann12a.html.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.” **International Conference on Machine Learning**, 1861–70.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, et al. 2018. “Soft Actor-Critic Algorithms and Applications.” **arXiv Preprint arXiv:1812.05905**.

Hafner, Danijar, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2025. “Mastering Diverse Control Tasks Through World Models.” **Nature** 640 (8059): 647–53. https://doi.org/10.1038/s41586-025-08744-2.

Hägele, Alexander, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro von Werra, and Martin Jaggi. 2024. “Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2405.18392.

Hansen, Nicklas, Hao Su, and Xiaolong Wang. 2024. “TD-MPC2: Scalable, Robust World Models for Continuous Control.” **International Conference on Learning Representations**. https://arxiv.org/abs/2310.16828.

Hansen-Estruch, Philippe, Jiahui Chen, Vivek Ramanujan, et al. 2026. “ViTok-V2: Scaling Native Resolution Auto-Encoders to 5 Billion Parameters.” **arXiv Preprint arXiv:2605.05331**.

Hansen-Estruch, Philippe, David Yan, Ching-Yao Chung, et al. 2025. “Learnings from Scaling Visual Tokenizers for Reconstruction and Generation.” **arXiv Preprint arXiv:2501.09755**.

Hasselt, Hado van. 2010. “Double Q-Learning.” **Advances in Neural Information Processing Systems** 23: 2613–21.

Hasselt, Hado van, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. 2018. “Deep Reinforcement Learning and the Deadly Triad.” **arXiv Preprint arXiv:1812.02648**.

Haviv, Adi, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. 2022. “Transformer Language Models Without Positional Encodings Still Learn Positional Information.” **Findings of the Association for Computational Linguistics: EMNLP 2022**. https://arxiv.org/abs/2203.16634.

He, Horace. 2022. **Making Deep Learning Go Brrrr from First Principles**. Https://horace.io/brrr_intro.html.

He, Kaiming, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. "Masked Autoencoders Are Scalable Vision Learners." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 16000–16009. https://doi.org/10.1109/cvpr52688.2022.01553.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." **Proceedings of the IEEE International Conference on Computer Vision**, 1026–34. https://doi.org/10.1109/iccv.2015.123.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. "Deep Residual Learning for Image Recognition." **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 770–78. https://doi.org/10.1109/cvpr.2016.90.

He, Pengcheng, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. "DeBERTa: Decoding-Enhanced BERT with Disentangled Attention." **International Conference on Learning Representations**.

Heimersheim, Stefan, and Neel Nanda. 2024. "How to Use and Interpret Activation Patching." **arXiv Preprint arXiv:2404.15255**.

Henderson, Peter, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. "Deep Reinforcement Learning That Matters." **Proceedings of the AAAI Conference on Artificial Intelligence** 32: 3207–14.

Hendrycks, Dan, and Kevin Gimpel. 2016. "Gaussian Error Linear Units (GELUs)." **ArXiv:1606.08415**. https://arxiv.org/abs/1606.08415.

Hennessy, John L, and David A Patterson. 2011. **Computer Architecture: A Quantitative Approach**. Elsevier. https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8.

Henry, Alex, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. 2020. "Query-Key Normalization for Transformers." **Findings of the Association for Computational Linguistics: EMNLP 2020**.

Hessel, Jack, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. 2021. "CLIPScore: A Reference-Free Evaluation Metric for Image Captioning." **Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing**. https://arxiv.org/abs/2104.08718.

Hessel, Matteo, Joseph Modayil, Hado van Hasselt, et al. 2018. "Rainbow: Combining Improvements in Deep Reinforcement Learning." **Proceedings of the AAAI Conference on Artificial Intelligence** 32: 3215–22.

Heusel, Martin, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium." **Advances in Neural Information Processing Systems** 30. https://arxiv.org/abs/1706.08500.

Hillis, W. Daniel, and Guy L. Steele Jr. 1986. “Data Parallel Algorithms.” **Communications of the ACM** 29 (12): 1170–83.

Hinton, Geoffrey E. 2002. “Training Products of Experts by Minimizing Contrastive Divergence.” **Neural Computation** 14 (8): 1771–800.

Hiraoka, Takuya, Takahisa Imagawa, Taisei Hashimoto, Takashi Onishi, and Yoshimasa Tsuruoka. 2022. “Dropout Q-Functions for Doubly Efficient Reinforcement Learning.” **International Conference on Learning Representations**.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel. 2020. “Denoising Diffusion Probabilistic Models.” **Advances in Neural Information Processing Systems** 33: 6840–51. https://arxiv.org/abs/2006.11239.

Ho, Jonathan, and Tim Salimans. 2022. “Classifier-Free Diffusion Guidance.” **arXiv Preprint arXiv:2207.12598**.

Ho, Jonathan, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J. Fleet. 2022. “Video Diffusion Models.” **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2204.03458.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. “Long Short-Term Memory.” **Neural Computation** 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Hockney, Roger W. 1994. “The Communication Challenge for MPP: Intel Paragon and Meiko CS-2.” **Parallel Computing** 20 (3): 389–98. https://doi.org/10.1016/S0167-8191(06)80021-9.

Hoeffding, Wassily. 1963. “Probability Inequalities for Sums of Bounded Random Variables.” **Journal of the American Statistical Association** 58 (301): 13–30.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, et al. 2022. “Training Compute-Optimal Large Language Models.” **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2203.15556.

Holderrieth, Peter, and Ezra Erives. 2025. “An Introduction to Flow Matching and Diffusion Models.” **arXiv Preprint arXiv:2506.02070**. https://arxiv.org/abs/2506.02070.

Holtzman, Ari, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. “The Curious Case of Neural Text Degeneration.” **International Conference on Learning Representations**. https://arxiv.org/abs/1904.09751.

Hoogeboom, Emiel, Alexey A. Gritsenko, Jasmijn Bastings, Ben Poole, Rianne van den Berg, and Tim Salimans. 2022. “Autoregressive Diffusion Models.” **International Conference on Learning Representations**.

Hoogeboom, Emiel, Jonathan Heek, and Tim Salimans. 2023. “Simple Diffusion: End-to-End Diffusion for High Resolution Images.” **International Conference on Machine Learning**. https://arxiv.org/abs/2301.11093.

Hoogeboom, Emiel, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. 2021. "Argmax Flows and Multinomial Diffusion: Learning Categorical Distributions." **Advances in Neural Information Processing Systems** 34. https://arxiv.org/abs/2102.05379.

Horowitz, Mark. 2014. "1.1 Computing's Energy Problem (and What We Can Do about It)." **IEEE International Solid-State Circuits Conference (ISSCC)**, 10–14. https://doi.org/10.1109/ISSCC.2014.6757323.

Howard, Ronald A. 1960. **Dynamic Programming and Markov Processes**. The Technology Press of the Massachusetts Institute of Technology; John Wiley & Sons.

Hu, Shengding, Yuge Tu, Xu Han, et al. 2024. "MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies." **ArXiv:2404.06395**. https://arxiv.org/abs/2404.06395.

Huang, Cheng-Zhi Anna, Ashish Vaswani, Jakob Uszkoreit, et al. 2018. "Music Transformer: Generating Music with Long-Term Structure." **International Conference on Learning Representations**. https://arxiv.org/abs/1809.04281.

Huang, Gao, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. 2017. "Snapshot Ensembles: Train 1, Get M for Free." **International Conference on Learning Representations**. https://arxiv.org/abs/1704.00109.

Huang, Shengyi, Rousslan F J Dossa, Chang Ye, et al. 2022. "CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms." **Journal of Machine Learning Research** 23 (274): 1–18.

Huang, Shengyi, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. 2022. **The 37 Implementation Details of Proximal Policy Optimization**. ICLR Blog Track, 25 March 2022, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Huang, Xun, and Serge Belongie. 2017. "Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization." **Proceedings of the IEEE International Conference on Computer Vision**, 1501–10. https://arxiv.org/abs/1703.06868.

Huang, Yanping, Youlong Cheng, Ankur Bapna, et al. 2019. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." **Advances in Neural Information Processing Systems** 32. https://arxiv.org/abs/1811.06965.

Huang, Yiwen, Aaron Gokaslan, Volodymyr Kuleshov, and James Tompkin. 2024. "The GAN Is Dead; Long Live the GAN! A Modern GAN Baseline." **Advances in Neural Information Processing Systems** 37.

Hyvärinen, Aapo. 2005. "Estimation of Non-Normalized Statistical Models by Score Matching." **Journal of Machine Learning Research** 6: 695–709.

Ilic, Aleksandar, Frederico Pratas, and Leonel Sousa. 2014. "Cache-Aware Roofline Model: Upgrading the Loft." **IEEE Computer Architecture Letters** 13 (1): 21–24. https://doi.or

g/10.1109/L-CA.2013.6.

Inan, Hakan, Khashayar Khosravi, and Richard Socher. 2017. “Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling.” **International Conference on Learning Representations**.

Isola, Phillip, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. “Image-to-Image Translation with Conditional Adversarial Networks.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 1125–34. https://doi.org/10.1109/cvpr.2017.632.

Ivanov, Andrei, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. “Data Movement Is All You Need: A Case Study on Optimizing Transformers.” **Proceedings of Machine Learning and Systems** 3. https://arxiv.org/abs/2007.00072.

Jaakkola, Tommi, Michael I Jordan, and Satinder P Singh. 1994. “On the Convergence of Stochastic Iterative Dynamic Programming Algorithms.” **Neural Computation** 6 (6): 1185–201.

Jacobs, Robert A., Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. “Adaptive Mixtures of Local Experts.” **Neural Computation** 3 (1): 79–87.

Jacot, Arthur, Franck Gabriel, and Clément Hongler. 2018. “Neural Tangent Kernel: Convergence and Generalization in Neural Networks.” **Advances in Neural Information Processing Systems** 31. https://doi.org/10.1145/3406325.3465355.

Jaegle, Andrew, Sebastian Borgeaud, Jean-Baptiste Alayrac, et al. 2022. “Perceiver IO: A General Architecture for Structured Inputs & Outputs.” **International Conference on Learning Representations**.

Jaegle, Andrew, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. 2021. “Perceiver: General Perception with Iterative Attention.” **International Conference on Machine Learning**, 4651–64.

Jain, Paras, Ajay Jain, Aniruddha Nrusimha, et al. 2020. “Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization.” **Proceedings of Machine Learning and Systems** 2. https://arxiv.org/abs/1910.02653.

Jain, Sarthak, and Byron C. Wallace. 2019. “Attention Is Not Explanation.” **Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics**, 3543–56.

Jamba Team. 2024. “Jamba-1.5: Hybrid Transformer-Mamba Models at Scale.” **arXiv Preprint arXiv:2408.12570**.

Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. “When to Trust Your Model: Model-Based Policy Optimization.” **Advances in Neural Information Processing Systems** 32.

Jelassi, Samy, David Brandfonbrener, Sham M. Kakade, and Eran Malach. 2024. “Repeat

After Me: Transformers Are Better Than State Space Models at Copying." **International Conference on Machine Learning**, 21502–21. https://arxiv.org/abs/2402.01032.

Jiang, Albert Q., Alexandre Sablayrolles, Arthur Mensch, et al. 2023. "Mistral 7B." **arXiv Preprint arXiv:2310.06825**.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux, et al. 2024. "Mixtral of Experts." **arXiv Preprint arXiv:2401.04088**.

Jiang, Nan, Alex Kulesza, Satinder Singh, and Richard Lewis. 2015. "The Dependence of Effective Planning Horizon on Model Accuracy." **International Conference on Autonomous Agents and Multiagent Systems**.

Jin, Chi, Zeyuan Allen-Zhu, Sébastien Bubeck, and Michael I. Jordan. 2018. "Is Q-Learning Provably Efficient?" **Advances in Neural Information Processing Systems** 31: 4863–73. https://arxiv.org/abs/1807.03765.

Jin, Ying, Zhuoran Yang, and Zhaoran Wang. 2021. "Is Pessimism Provably Efficient for Offline RL?" **International Conference on Machine Learning**, Proceedings of machine learning research, vol. 139: 5084–96.

Johnson, Rie, and Tong Zhang. 2013. "Accelerating Stochastic Gradient Descent Using Predictive Variance Reduction." **Advances in Neural Information Processing Systems** 26: 315–23.

Jolicoeur-Martineau, Alexia. 2019. "The Relativistic Discriminator: A Key Element Missing from Standard GAN." **International Conference on Learning Representations**.

Jolicoeur-Martineau, Alexia. 2020. "On Relativistic f-Divergences." **Proceedings of the 37th International Conference on Machine Learning**, 4931–39.

Jordan, Keller. 2026. **Modded-NanoGPT Optimization Benchmark**. Https://github.com/KellerJordan/modded-nanogpt/tree/master/records/track_3_optimization.

Jordan, Keller, and contributors. 2024. **Modded-Nanogpt: Speedrunning the NanoGPT Baseline**. Https://github.com/KellerJordan/modded-nanogpt.

Jordan, Keller, Yuchen Jin, Vlado Boza, et al. 2024. **Muon: An Optimizer for Hidden Layers in Neural Networks**. Https://kellerjordan.github.io/posts/muon/.

Jouppi, Norman P, Cliff Young, Nishant Patil, et al. 2017. "In-Datacenter Performance Analysis of a Tensor Processing Unit." **2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)**, 1–12. https://doi.org/10.1145/3140659.3080246.

Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. 2015. "An Empirical Exploration of Recurrent Network Architectures." **International Conference on Machine Learning**, 2342–50. https://proceedings.mlr.press/v37/jozefowicz15.html.

Kadkhodaie, Zahra, Florentin Guth, Eero P. Simoncelli, and Stéphane Mallat. 2024. “Generalization in Diffusion Models Arises from Geometry-Adaptive Harmonic Representations.” **International Conference on Learning Representations**. https://arxiv.org/abs/2310.02557.

Kaelbling, Leslie Pack, Michael L Littman, and Anthony R Cassandra. 1998. “Planning and Acting in Partially Observable Stochastic Domains.” **Artificial Intelligence** 101 (1–2): 99–134. https://doi.org/10.1016/S0004-3702(98)00023-X.

Kahng, Minsuk, Nikhil Thorat, Duen Horng Chau, Fernanda B. Viégas, and Martin Wattenberg. 2018. “GAN Lab: Understanding Complex Deep Generative Models Using Interactive Visual Experimentation.” **IEEE Transactions on Visualization and Computer Graphics** 25 (1): 310–20.

Kakade, Sham, and John Langford. 2002. “Approximately Optimal Approximate Reinforcement Learning.” **International Conference on Machine Learning**, 267–74.

Kalman, Rudolf E. 1960. “A New Approach to Linear Filtering and Prediction Problems.” **Journal of Basic Engineering** 82 (1): 35–45.

Kalra, Dayal Singh, and Maissam Barkeshli. 2024. “Why Warmup the Learning Rate? Underlying Mechanisms and Improvements.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2406.09405.

Kamb, Mason, and Surya Ganguli. 2025. “An Analytic Theory of Creativity in Convolutional Diffusion Models.” **Proceedings of the 42nd International Conference on Machine Learning**, Proceedings of machine learning research, vol. 267: 28795–831.

Kaplan, Jared, Sam McCandlish, Tom Henighan, et al. 2020. “Scaling Laws for Neural Language Models.” **ArXiv:2001.08361**. https://arxiv.org/abs/2001.08361.

Karras, Tero, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. “Progressive Growing of GANs for Improved Quality, Stability, and Variation.” **International Conference on Learning Representations**. https://arxiv.org/abs/1710.10196.

Karras, Tero, Miika Aittala, Timo Aila, and Samuli Laine. 2022. “Elucidating the Design Space of Diffusion-Based Generative Models.” **Advances in Neural Information Processing Systems** 35.

Karras, Tero, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. 2020. “Training Generative Adversarial Networks with Limited Data.” **Advances in Neural Information Processing Systems** 33.

Karras, Tero, Miika Aittala, Tuomas Kynkäänniemi, Jaakko Lehtinen, Timo Aila, and Samuli Laine. 2024. “Guiding a Diffusion Model with a Bad Version of Itself.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2406.02507.

Karras, Tero, Miika Aittala, Samuli Laine, et al. 2021. “Alias-Free Generative Adversarial Networks.” **Advances in Neural Information Processing Systems** 34: 852–63. https://arxiv.org/abs/2106.12423.

Karras, Tero, Miika Aittala, Jaakko Lehtinen, Janne Hellsten, Timo Aila, and Samuli Laine. 2024. “Analyzing and Improving the Training Dynamics of Diffusion Models.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**.

Karras, Tero, Samuli Laine, and Timo Aila. 2019. “A Style-Based Generator Architecture for Generative Adversarial Networks.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 4401–10. https://arxiv.org/abs/1812.04948.

Karras, Tero, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. 2020. “Analyzing and Improving the Image Quality of StyleGAN.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 8110–19. https://arxiv.org/abs/1912.04958.

Kasimbeg, Priya, Frank Schneider, Runa Eschenhagen, et al. 2025. “Accelerating Neural Network Training: An Analysis of the AlgoPerf Competition.” **International Conference on Learning Representations**. https://arxiv.org/abs/2502.15015.

Katharopoulos, Angelos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. “Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention.” **International Conference on Machine Learning**.

Kawar, Bahjat, Michael Elad, Stefano Ermon, and Jiaming Song. 2022. “Denoising Diffusion Restoration Models.” **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2201.11793.

Kazemnejad, Amirhossein, Milad Aghajohari, Eva Portelance, et al. 2025. “VinePPO: Refining Credit Assignment in RL Training of LLMs.” **International Conference on Machine Learning**. https://arxiv.org/abs/2410.01679.

Kazemnejad, Amirhossein, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. 2023. “The Impact of Positional Encoding on Length Generalization in Transformers.” **Advances in Neural Information Processing Systems** 36. https://arxiv.org/abs/2305.19466.

Kidambi, Rahul, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. 2020. “MOReL: Model-Based Offline Reinforcement Learning.” **Advances in Neural Information Processing Systems** 33.

Kim, Dongjun, Chieh-Hsin Lai, Wei-Hsiang Liao, et al. 2024. “Consistency Trajectory Models: Learning Probability Flow ODE Trajectory of Diffusion.” **International Conference on Learning Representations**. https://arxiv.org/abs/2310.02279.

Kim, Gyu Yeol, and Min-hwan Oh. 2026. “Convergence of Muon with Newton–Schulz.” **International Conference on Learning Representations**.

Kim, Jaeyeon, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. 2025. “Train for the Worst, Plan for the Best: Understanding Token Ordering in Masked Diffusions.” **International Conference on Machine Learning**.

Kimi Team. 2025a. “Kimi K2: Open Agentic Intelligence.” **arXiv Preprint arXiv:2507.20534**.

Kimi Team. 2025b. “Kimi Linear: An Expressive, Efficient Attention Architecture.” **arXiv Preprint arXiv:2510.26692**.

Kingma, Diederik P, and Jimmy Ba. 2015. “Adam: A Method for Stochastic Optimization.” **International Conference on Learning Representations**. https://arxiv.org/abs/1412.6980.

Kingma, Diederik P., and Ruiqi Gao. 2023. “Understanding Diffusion Objectives as the ELBO with Simple Data Augmentation.” **Advances in Neural Information Processing Systems** 36. https://arxiv.org/abs/2303.00848.

Kingma, Diederik P., Tim Salimans, Ben Poole, and Jonathan Ho. 2021. “Variational Diffusion Models.” **Advances in Neural Information Processing Systems** 34.

Kingma, Diederik P., and Max Welling. 2014. “Auto-Encoding Variational Bayes.” **International Conference on Learning Representations (ICLR)**. https://arxiv.org/abs/1312.6114.

Kitaev, Nikita, Lukasz Kaiser, and Anselm Levskaya. 2020. “Reformer: The Efficient Transformer.” **International Conference on Learning Representations**.

Kocmi, Tom, Christian Federmann, Roman Grundkiewicz, Marcin Junczys-Dowmunt, Hitokazu Matsushita, and Arul Menezes. 2021. “To Ship or Not to Ship: An Extensive Evaluation of Automatic Metrics for Machine Translation.” **Proceedings of the Sixth Conference on Machine Translation**, 478–94.

Koehler, Frederic, Alexander Heckett, and Andrej Risteski. 2022. “Statistical Efficiency of Score Matching: The View from Isoperimetry.” **arXiv Preprint arXiv:2210.00726**. https://arxiv.org/abs/2210.00726.

Komatsuzaki, Aran, Joan Puigcerver, James Lee-Thorp, et al. 2023. “Sparse Upcycling: Training Mixture-of-Experts from Dense Checkpoints.” **International Conference on Learning Representations**.

Konda, Vijay R, and John N Tsitsiklis. 2000. “Actor-Critic Algorithms.” **Advances in Neural Information Processing Systems** 12.

Kong, Jungil, Jaehyeon Kim, and Jaekyoung Bae. 2020. “HiFi-GAN: Generative Adversarial Networks for Efficient and High Fidelity Speech Synthesis.” **Advances in Neural Information Processing Systems** 33.

Kong, Zhifeng, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. 2021. “DiffWave: A Versatile Diffusion Model for Audio Synthesis.” **International Conference on Learning Representations**. https://arxiv.org/abs/2009.09761.

Kool, Wouter, Herke van Hoof, and Max Welling. 2019. “Buy 4 REINFORCE Samples, Get a Baseline for Free!” **ICLR 2019 Workshop on Deep Reinforcement Learning Meets Structured Prediction**. https://mlanthology.org/iclrw/2019/kool2019iclrw-buy/.

Korbak, Tomasz, Ethan Perez, and Christopher L Buckley. 2022. “RL with KL Penalties Is

Better Viewed as Bayesian Inference." **Findings of the Association for Computational Linguistics: EMNLP 2022**. https://arxiv.org/abs/2205.11275.

Korpelevich, G. M. 1976. "The Extragradient Method for Finding Saddle Points and Other Problems." **Ekonomika i Matematicheskie Metody** 12: 747–56.

Korthikanti, Vijay, Jared Casper, Sangkug Lym, et al. 2022. "Reducing Activation Recomputation in Large Transformer Models." **arXiv Preprint arXiv:2205.05198**.

Kosson, Atli, Bettina Messmer, and Martin Jaggi. 2024. "Rotational Equilibrium: How Weight Decay Balances Learning Across Neural Networks." **International Conference on Machine Learning**. https://arxiv.org/abs/2305.17212.

Kostrikov, Ilya, Ashvin Nair, and Sergey Levine. 2022. "Offline Reinforcement Learning with Implicit Q-Learning." **International Conference on Learning Representations**. https://arxiv.org/abs/2110.06169.

Kovalev, Dmitry. 2025. "Understanding Gradient Orthogonalization for Deep Learning via Non-Euclidean Trust-Region Optimization." **arXiv Preprint arXiv:2503.12645**.

Krakovna, Victoria, Jonathan Uesato, Vladimir Mikulik, et al. 2020. **Specification Gaming: The Flip Side of AI Ingenuity**. DeepMind blog, 21 April 2020, https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/.

Kramár, János, Tom Lieberum, Rohin Shah, and Neel Nanda. 2024. "AtP*: An Efficient and Scalable Method for Localizing LLM Behaviour to Components." **arXiv Preprint arXiv:2403.00745**.

Kramers, Hendrik A. 1940. "Brownian Motion in a Field of Force and the Diffusion Model of Chemical Reactions." **Physica** 7 (4): 284–304.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." **Advances in Neural Information Processing Systems**, 1097–105. https://doi.org/10.5555/2999134.2999257.

Kuleshov, Volodymyr. 2023. **Deep Generative Models: Lecture Slides (Lectures 10–13: Advanced Topics in GANs, Energy-Based Models, Score-Based Generative Models, Diffusion Models)**. Course materials, Cornell Tech. Course website: https://kuleshov-group.github.io/dgm-website/; slides: https://github.com/kuleshov-group/dgm-resources.

Kullback, Solomon. 1959. **Information Theory and Statistics**. John Wiley & Sons.

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine. 2020. "Conservative Q-Learning for Offline Reinforcement Learning." **Advances in Neural Information Processing Systems** 33.

Kung, Sun Yuan. 1988. "VLSI Array Processors." **Prentice Hall**.

Kunstner, Frederik, Jacques Chen, Jonathan Wilder Lavington, and Mark Schmidt. 2023. "Noise Is Not the Main Factor Behind the Gap Between SGD and Adam on Transformers,

but Sign Descent Might Be." **International Conference on Learning Representations**. https://arxiv.org/abs/2304.13960.

Kunstner, Frederik, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. 2024. "Heavy-Tailed Class Imbalance and Why Adam Outperforms Gradient Descent on Language Models." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2402.19449.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, et al. 2023. "Efficient Memory Management for Large Language Model Serving with PagedAttention." **Proceedings of the 29th Symposium on Operating Systems Principles**, 611–26. https://doi.org/10.1145/3600006.3613165.

Kynkäänniemi, Tuomas, Miika Aittala, Tero Karras, Samuli Laine, Timo Aila, and Jaakko Lehtinen. 2024. "Applying Guidance in a Limited Interval Improves Sample and Distribution Quality in Diffusion Models." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2404.07724.

Kynkäänniemi, Tuomas, Tero Karras, Samuli Laine, Jaakko Lehtinen, and Timo Aila. 2019. "Improved Precision and Recall Metric for Assessing Generative Models." **Advances in Neural Information Processing Systems** 32. https://arxiv.org/abs/1904.06991.

Lahoti, Aakash, Kevin Y. Li, Berlin Chen, et al. 2026. "Mamba-3: Improved Sequence Modeling Using State Space Principles." **arXiv Preprint arXiv:2603.15569**.

Lai, Chieh-Hsin, Yang Song, Dongjun Kim, Yuki Mitsufuji, and Stefano Ermon. 2025. "The Principles of Diffusion Models." **arXiv Preprint arXiv:2510.21890**. https://arxiv.org/abs/2510.21890.

Lai, Tze Leung, and Herbert Robbins. 1985. "Asymptotically Efficient Adaptive Allocation Rules." **Advances in Applied Mathematics** 6 (1): 4–22. https://doi.org/10.1016/0196-8858(85)90002-8.

Lambert, Nathan. 2026. **Reinforcement Learning from Human Feedback**. Online book, continuously updated since 2024; arXiv:2504.12501; print edition Manning/Simon & Schuster, https://rlhfbook.com/.

Lattimore, Tor, and Csaba Szepesvári. 2020. **Bandit Algorithms**. Cambridge University Press. https://tor-lattimore.com/downloads/book/book.pdf.

Lee, Jason D., Max Simchowitz, Michael I. Jordan, and Benjamin Recht. 2016. "Gradient Descent Only Converges to Minimizers." **29th Annual Conference on Learning Theory**, Proceedings of machine learning research, vol. 49: 1246–57.

Lee, Sang-gil, Wei Ping, Boris Ginsburg, Bryan Catanzaro, and Sungroh Yoon. 2023. "BigVGAN: A Universal Neural Vocoder with Large-Scale Training." **International Conference on Learning Representations**.

Lee, Yongjoon, and Jung-Woo Choi. 2026. "RAF: Relativistic Adversarial Feedback for Universal Speech Synthesis." **Interspeech**.

Lepikhin, Dmitry, HyoukJoong Lee, Yuanzhong Xu, et al. 2021. "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding." **International Conference on Learning Representations**.

Leviathan, Yaniv, Matan Kalman, and Yossi Matias. 2023. "Fast Inference from Transformers via Speculative Decoding." **International Conference on Machine Learning**, 19274–86.

Levine, Sergey. 2018. "Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review." **arXiv Preprint arXiv:1805.00909**.

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu. 2020. "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems." **arXiv Preprint arXiv:2005.01643**.

Li, Chun-Liang, Wei-Cheng Chang, Yu Cheng, Yiming Yang, and Barnabás Póczos. 2017. "MMD GAN: Towards Deeper Understanding of Moment Matching Network." **Advances in Neural Information Processing Systems** 30.

Li, Gen, Yuting Wei, Yuejie Chi, Yuantao Gu, and Yuxin Chen. 2021. "Softmax Policy Gradient Methods Can Take Exponential Time to Converge." **Proceedings of the 34th Conference on Learning Theory**, Proceedings of machine learning research, vol. 134: 3107–10.

Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. "Visualizing the Loss Landscape of Neural Nets." **Advances in Neural Information Processing Systems** 31: 6389–99.

Li, Ke, and Jitendra Malik. 2018. "Implicit Maximum Likelihood Estimation." **arXiv Preprint arXiv:1809.09087**.

Li, Mu, David G Andersen, Jun Woo Park, et al. 2014. "Scaling Distributed Machine Learning with the Parameter Server." **11th Symposium on Operating Systems Design and Implementation (OSDI 14)**, 583–98. https://doi.org/10.1145/2640087.2644155.

Li, Shen, Yanli Zhao, Rohan Varma, et al. 2020. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training." **Proceedings of the VLDB Endowment** 13 (12): 3005–18. https://doi.org/10.14778/3415478.3415530.

Li, Tianhong, and Kaiming He. 2026. "Back to Basics: Let Denoising Generative Models Denoise." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 36115–25. https://arxiv.org/abs/2511.13720.

Li, Weiwei, and Emanuel Todorov. 2004. "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems." **Proceedings of the First International Conference on Informatics in Control, Automation and Robotics**, 222–29.

Li, Yujia, Kevin Swersky, and Richard Zemel. 2015. "Generative Moment Matching Networks." **Proceedings of the 32nd International Conference on Machine Learning**, 1718–27.

Li, Zichong, Liming Liu, Chen Liang, Weizhu Chen, and Tuo Zhao. 2026. “NorMuon: Making Muon More Efficient and Scalable.” **International Conference on Machine Learning**.

Lieber, Opher, Barak Lenz, Hofit Bata, et al. 2024. “Jamba: A Hybrid Transformer-Mamba Language Model.” **arXiv Preprint arXiv:2403.19887**.

Lightman, Hunter, Vineet Kosaraju, Yuri Burda, et al. 2024. “Let’s Verify Step by Step.” **International Conference on Learning Representations**.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, et al. 2016. “Continuous Control with Deep Reinforcement Learning.” **International Conference on Learning Representations**. https://arxiv.org/abs/1509.02971.

Lim, Jae Hyun, and Jong Chul Ye. 2017. “Geometric GAN.” **arXiv Preprint arXiv:1705.02894**.

Lin, Long-Ji. 1992. “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching.” **Machine Learning** 8: 293–321.

Lin, Shanchuan, Bingchen Liu, Jiashi Li, and Xiao Yang. 2024. “Common Diffusion Noise Schedules and Sample Steps Are Flawed.” **Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision**, 5404–11. https://arxiv.org/abs/2305.08891.

Lin, Shanchuan, Anran Wang, and Xiao Yang. 2024. “SDXL-Lightning: Progressive Adversarial Diffusion Distillation.” **arXiv Preprint arXiv:2402.13929**.

Lin, Shanchuan, Xin Xia, Yuxi Ren, Ceyuan Yang, Xuefeng Xiao, and Lu Jiang. 2025. “Diffusion Adversarial Post-Training for One-Step Video Generation.” **Proceedings of the 42nd International Conference on Machine Learning**.

Lin, Zhouhan, Minwei Feng, Cicero Nogueira dos Santos, et al. 2017. “A Structured Self-Attentive Sentence Embedding.” **ArXiv:1703.03130**. https://arxiv.org/abs/1703.03130.

Lipman, Yaron, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. 2023. “Flow Matching for Generative Modeling.” **International Conference on Learning Representations**.

Lipman, Yaron, Marton Havasi, Peter Holderrieth, et al. 2024. “Flow Matching Guide and Code.” **arXiv Preprint arXiv:2412.06264**. https://arxiv.org/abs/2412.06264.

Little, John D. C. 1961. “A Proof for the Queuing Formula: $L = \lambda W$.” **Operations Research** 9 (3): 383–87. https://doi.org/10.1287/opre.9.3.383.

Liu, Bo, Rui Wang, Lemeng Wu, Yihao Feng, Peter Stone, and Qiang Liu. 2024. “Longhorn: State Space Models Are Amortized Online Learners.” **arXiv Preprint arXiv:2407.14207**.

Liu, Dong C, and Jorge Nocedal. 1989. “On the Limited Memory BFGS Method for Large Scale Optimization.” **Mathematical Programming** 45 (1): 503–28. https://doi.org/10.1007/bf01589116.

Liu, Jie, Gongye Liu, Jiajun Liang, et al. 2025. “Flow-GRPO: Training Flow Matching Models via Online RL.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2505.05470.

Liu, Jingyuan, Jianlin Su, Xingcheng Yao, et al. 2025. “Muon Is Scalable for LLM Training.” **arXiv Preprint arXiv:2502.16982**.

Liu, Nelson F., Kevin Lin, John Hewitt, et al. 2024. “Lost in the Middle: How Language Models Use Long Contexts.” **Transactions of the Association for Computational Linguistics** 12: 157–73. https://arxiv.org/abs/2307.03172.

Liu, Xingchao, Chengyue Gong, and Qiang Liu. 2023. “Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow.” **International Conference on Learning Representations**.

Liu, Yinhan, Myle Ott, Naman Goyal, et al. 2019. “RoBERTa: A Robustly Optimized BERT Pretraining Approach.” **ArXiv:1907.11692**. https://arxiv.org/abs/1907.11692.

Liu, Yue, Yunjie Tian, Yuzhong Zhao, et al. 2024. “VMamba: Visual State Space Model.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2401.10166.

Liu, Ze, Yutong Lin, Yue Cao, et al. 2021. “Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows.” **Proceedings of the IEEE/CVF International Conference on Computer Vision**, 10012–22. https://doi.org/10.1109/iccv48922.2021.00986.

Liu, Zhuang, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. “A ConvNet for the 2020s.” **ArXiv:2201.03545**. https://arxiv.org/abs/2201.03545.

Liu, Zichen, Changyu Chen, Wenjun Li, et al. 2025. “Understanding R1-Zero-Like Training: A Critical Perspective.” **Second Conference on Language Modeling**. https://arxiv.org/abs/2503.20783.

Loshchilov, Ilya, and Frank Hutter. 2016. “SGDR: Stochastic Gradient Descent with Warm Restarts.” **ArXiv:1608.03983**. https://arxiv.org/abs/1608.03983.

Loshchilov, Ilya, and Frank Hutter. 2019. “Decoupled Weight Decay Regularization.” **International Conference on Learning Representations**. https://arxiv.org/abs/1711.05101.

Lou, Aaron, Chenlin Meng, and Stefano Ermon. 2024. “Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution.” **Proceedings of the 41st International Conference on Machine Learning**, Proceedings of machine learning research, vol. 235: 32819–48. https://arxiv.org/abs/2310.16834.

Lu, Cheng, and Yang Song. 2025. “Simplifying, Stabilizing and Scaling Continuous-Time Consistency Models.” **International Conference on Learning Representations**.

Lu, Cheng, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. “DPM-

Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps." **Advances in Neural Information Processing Systems** 35.

Lu, Cheng, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2025. "DPM-Solver++: Fast Solver for Guided Sampling of Diffusion Probabilistic Models." **Machine Intelligence Research** 22 (4): 730–51. https://doi.org/10.1007/s11633-025-1562-4.

Lu, Enzhe, Zhejun Jiang, Jingyuan Liu, et al. 2025. "MoBA: Mixture of Block Attention for Long-Context LLMs." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2502.13189.

Lucic, Mario, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. 2018. "Are GANs Created Equal? A Large-Scale Study." **Advances in Neural Information Processing Systems** 31. https://arxiv.org/abs/1711.10337.

Lugmayr, Andreas, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. 2022. "RePaint: Inpainting Using Denoising Diffusion Probabilistic Models." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 11461–71. https://arxiv.org/abs/2201.09865.

Luo, Calvin. 2022. "Understanding Diffusion Models: A Unified Perspective." **arXiv Preprint arXiv:2208.11970**.

Luo, Weile, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture." **arXiv Preprint arXiv:2402.13499**.

Lyle, Clare, Pablo Samuel Castro, and Marc G. Bellemare. 2019. "A Comparative Analysis of Expected and Distributional Reinforcement Learning." **Proceedings of the AAAI Conference on Artificial Intelligence** 33: 4504–11.

Mack, Yue-Pok, and Bernard W Silverman. 1982. "Weak and Strong Uniform Consistency of Kernel Regression Estimates." **Zeitschrift für Wahrscheinlichkeitstheorie Und Verwandte Gebiete** 61 (3): 405–15. https://doi.org/10.1007/bf00539840.

Makoviychuk, Viktor, Lukasz Wawrzyniak, Yunrong Guo, et al. 2021. "Isaac Gym: High Performance GPU-Based Physics Simulation for Robot Learning." **Advances in Neural Information Processing Systems, Datasets and Benchmarks Track**. https://arxiv.org/abs/2108.10470.

Malladi, Sadhika, Kaifeng Lyu, Abhishek Panigrahi, and Sanjeev Arora. 2022. "On the SDEs and Scaling Rules for Adaptive Gradient Algorithms." **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2205.10287.

Mao, Xudong, Qing Li, Haoran Xie, Raymond Y. K. Lau, Zhen Wang, and Stephen Paul Smolley. 2017. "Least Squares Generative Adversarial Networks." **Proceedings of the IEEE International Conference on Computer Vision**, 2794–802.

Martens, James. 2010. "Deep Learning via Hessian-Free Optimization." **Proceedings of the 27th International Conference on Machine Learning**, 735–42.

Martens, James, and Roger Grosse. 2015. “Optimizing Neural Networks with Kronecker-Factored Approximate Curvature.” **Proceedings of the 32nd International Conference on Machine Learning (ICML)**, 2408–17.

Massias, Mathurin, and Quentin Bertrand. 2026. **Diffusion and Flow-Matching: From Memorization to Generalization & Beyond**. Tutorial at the International Conference on Machine Learning, https://icml.cc/virtual/2026/75374.

Mayne, David Q. 1966. “A Second-Order Gradient Method for Determining Optimal Trajectories of Non-Linear Discrete-Time Systems.” **International Journal of Control** 3 (1): 85–95. https://doi.org/10.1080/00207176608921369.

McCandlish, Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. “An Empirical Model of Large-Batch Training.” **arXiv Preprint arXiv:1812.06162**.

Mei, Jincheng, Bo Dai, Chenjun Xiao, Csaba Szepesvári, and Dale Schuurmans. 2021. “Understanding the Effect of Stochasticity in Policy Optimization.” **Advances in Neural Information Processing Systems** 34.

Mei, Jincheng, Chenjun Xiao, Csaba Szepesvári, and Dale Schuurmans. 2020. “On the Global Convergence Rates of Softmax Policy Gradient Methods.” **Proceedings of the 37th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 119: 6820–29.

Meng, Chenlin, Yutong He, Yang Song, et al. 2022. “SDEdit: Guided Image Synthesis and Editing with Stochastic Differential Equations.” **International Conference on Learning Representations**. https://arxiv.org/abs/2108.01073.

Meng, Chenlin, Robin Rombach, Ruiqi Gao, et al. 2023. “On Distillation of Guided Diffusion Models.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 14297–306.

Meng, Kevin, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. “Locating and Editing Factual Associations in GPT.” **Advances in Neural Information Processing Systems** 35. https://arxiv.org/abs/2202.05262.

Merity, Stephen, Caiming Xiong, James Bradbury, and Richard Socher. 2016. “Pointer Sentinel Mixture Models.” **ArXiv:1609.07843**. https://arxiv.org/abs/1609.07843.

Merrill, William, Shane Arora, Dirk Groeneveld, and Hannaneh Hajishirzi. 2025. “Critical Batch Size Revisited: A Simple Empirical Approach to Large-Batch Language Model Training.” **arXiv Preprint arXiv:2505.23971**.

Merrill, William, Jackson Petty, and Ashish Sabharwal. 2024. “The Illusion of State in State-Space Models.” **International Conference on Machine Learning**. https://arxiv.org/abs/2404.08819.

Mescheder, Lars, Andreas Geiger, and Sebastian Nowozin. 2018. “Which Training Methods for GANs Do Actually Converge?” **International Conference on Machine Learning**, 3481–90. https://arxiv.org/abs/1801.04406.

Meta AI. 2025. **The Llama 4 Herd: The Beginning of a New Era of Natively Multimodal AI Innovation**. Https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

Michel, Paul, Omer Levy, and Graham Neubig. 2019. "Are Sixteen Heads Really Better Than One?" **Advances in Neural Information Processing Systems** 32.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, et al. 2018. "Mixed Precision Training." **International Conference on Learning Representations**.

Micikevicius, Paulius, Dusan Stosic, Neil Burgess, et al. 2022. "FP8 Formats for Deep Learning." **arXiv Preprint arXiv:2209.05433**.

Milakov, Maxim, and Natalia Gimelshein. 2018. "Online Normalizer Calculation for Softmax." **arXiv Preprint arXiv:1805.02867**.

MiniMax. 2025. "MiniMax-01: Scaling Foundation Models with Lightning Attention." **arXiv Preprint arXiv:2501.08313**.

Mirza, Mehdi, and Simon Osindero. 2014. "Conditional Generative Adversarial Nets." **ArXiv:1411.1784**. https://arxiv.org/abs/1411.1784.

Mishchenko, Konstantin, Ahmed Khaled, and Peter Richtárik. 2020. "Random Reshuffling: Simple Analysis with Vast Improvements." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2006.05988.

Mistral AI. 2023. **Mistral-7B-Instruct-v0.2 Model Card**. Https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2.

Miyasawa, Koichi. 1961. "An Empirical Bayes Estimator of the Mean of a Normal Population." **Bulletin of the International Statistical Institute** 38 (4): 181–88.

Miyato, Takeru, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. "Spectral Normalization for Generative Adversarial Networks." **International Conference on Learning Representations**.

Miyato, Takeru, and Masanori Koyama. 2018. "CGANs with Projection Discriminator." **International Conference on Learning Representations**. https://arxiv.org/abs/1802.05637.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, et al. 2016. "Asynchronous Methods for Deep Reinforcement Learning." **International Conference on Machine Learning**, 1928–37.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2013. "Playing Atari with Deep Reinforcement Learning." **ArXiv:1312.5602**.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." **Nature** 518 (7540): 529–33. https://doi.org/10.1038/nature14236.

Mohan, Jayashree, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. “Analyzing and Mitigating Data Stalls in DNN Training.” **Proceedings of the VLDB Endowment** 14 (5): 771–84. https://doi.org/10.14778/3446095.3446100.

Mokady, Ron, Amir Hertz, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. 2023. “NULL-Text Inversion for Editing Real Images Using Guided Diffusion Models.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 6038–47.

Murphy, Kevin P. 2025. “Reinforcement Learning: An Overview.” **arXiv Preprint arXiv:2412.05265**.

Nadaraya, Elizbar A. 1964. “On Estimating Regression.” **Theory of Probability & Its Applications** 9 (1): 141–42. https://doi.org/10.1137/1109020.

Nado, Zachary, Justin M. Gilmer, Christopher J. Shallue, Rohan Anil, and George E. Dahl. 2021. “A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes.” **arXiv Preprint arXiv:2102.06356**.

Nagabandi, Anusha, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. 2018. “Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning.” **IEEE International Conference on Robotics and Automation (ICRA)**, 7559–66. https://doi.org/10.1109/ICRA.2018.8463189.

Nakamoto, Mitsuhiko, Yuexiang Zhai, Anikait Singh, et al. 2023. “Cal-QL: Calibrated Offline RL Pre-Training for Efficient Online Fine-Tuning.” **Advances in Neural Information Processing Systems** 36.

Nakkiran, Preetum, Arwen Bradley, Hattie Zhou, and Madhu Advani. 2025. “Step-by-Step Diffusion: An Elementary Tutorial.” **Foundations and Trends in Computer Graphics and Vision** 17 (1): 1–75. https://doi.org/10.1561/0600000114.

Nanda, Neel. 2023. **Attribution Patching: Activation Patching at Industrial Scale**. Blog post, https://www.neelnanda.io/attribution-patching.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, et al. 2021. “Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM.” **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**, 1–15. https://doi.org/10.1145/3458817.3476209.

Nauman, Michal, Mateusz Ostaszewski, Krzysztof Jankowski, Piotr Miłoś, and Marek Cygan. 2024. “Bigger, Regularized, Optimistic: Scaling for Compute and Sample-Efficient Continuous Control.” **Advances in Neural Information Processing Systems** 37.

Nemirovski, Arkadi, and David Yudin. 1983. **Problem Complexity and Method Efficiency in Optimization**. Wiley.

Nesterov, Yu. 2018. **Lectures on Convex Optimization**. Springer. https://doi.org/10.1007/978-3-319-91578-4.

Nesterov, Yurii. 1983. “A Method of Solving a Convex Programming Problem with Conver-

gence Rate $O(1/k^2)$." **Soviet Mathematics Doklady** 27 (2): 372–76.

Ng, Andrew Y, Daishi Harada, and Stuart Russell. 1999. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping." **International Conference on Machine Learning**, 278–87.

Nguyen, Eric, Michael Poli, Marjan Faizi, et al. 2023. "HyenaDNA: Long-Range Genomic Sequence Modeling at Single Nucleotide Resolution." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2306.15794.

Nguyen, XuanLong, Martin J. Wainwright, and Michael I. Jordan. 2009. "On Surrogate Loss Functions and f-Divergences." **The Annals of Statistics** 37 (2): 876–904.

Nichol, Alexander Quinn, and Prafulla Dhariwal. 2021. "Improved Denoising Diffusion Probabilistic Models." **Proceedings of the 38th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 139: 8162–71. https://arxiv.org/abs/2102.09672.

Nie, Shen, Fengqi Zhu, Chao Du, et al. 2025. "Scaling up Masked Diffusion Models on Text." **International Conference on Learning Representations**. https://arxiv.org/abs/2410.18514.

Nie, Shen, Fengqi Zhu, Zebin You, et al. 2025. "Large Language Diffusion Models." **Advances in Neural Information Processing Systems** 38. https://arxiv.org/abs/2502.09992.

Nikishin, Evgenii, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. 2022. "The Primacy Bias in Deep Reinforcement Learning." **International Conference on Machine Learning**. https://arxiv.org/abs/2205.07802.

Nocedal, Jorge, and Stephen J. Wright. 2006. **Numerical Optimization**. 2nd ed. Springer.

Nota, Chris, and Philip S. Thomas. 2020. "Is the Policy Gradient a Gradient?" **Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems**.

Nowozin, Sebastian, Botond Cseke, and Ryota Tomioka. 2016. "F-GAN: Training Generative Neural Samplers Using Variational Divergence Minimization." **Advances in Neural Information Processing Systems** 29.

NVIDIA. 2025. "Nemotron-H: A Family of Accurate and Efficient Hybrid Mamba-Transformer Models." **arXiv Preprint arXiv:2504.03624**.

Odena, Augustus, Vincent Dumoulin, and Chris Olah. 2016. "Deconvolution and Checkerboard Artifacts." **Distill**. http://distill.pub/2016/deconv-checkerboard.

Odena, Augustus, Christopher Olah, and Jonathon Shlens. 2017. "Conditional Image Synthesis with Auxiliary Classifier GANs." **Proceedings of the 34th International Conference on Machine Learning**, 2642–51. https://arxiv.org/abs/1610.09585.

Olsson, Catherine, Nelson Elhage, Neel Nanda, et al. 2022. "In-Context Learning and Induc-

tion Heads." **Transformer Circuits Thread**.

OpenAI. 2025. "Gpt-Oss-120b & Gpt-Oss-20b Model Card." **arXiv Preprint arXiv:2508.10925**.

Oquab, Maxime, Timothée Darcet, Théo Moutakanni, et al. 2024. "DINOv2: Learning Robust Visual Features Without Supervision." **Transactions on Machine Learning Research**. https://arxiv.org/abs/2304.07193.

Osband, Ian, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. "Deep Exploration via Bootstrapped DQN." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/1602.04621.

Oswald, Johannes von, Eyvind Niklasson, Ettore Randazzo, et al. 2023. "Transformers Learn in-Context by Gradient Descent." **International Conference on Machine Learning**, Proceedings of machine learning research, vol. 202. https://arxiv.org/abs/2212.07677.

Ou, Jingyang, Shen Nie, Kaiwen Xue, et al. 2025. "Your Absorbing Discrete Diffusion Secretly Models the Conditional Distributions of Clean Data." **International Conference on Learning Representations**. https://arxiv.org/abs/2406.03736.

Ouyang, Long, Jeff Wu, Xu Jiang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." **ArXiv:2203.02155**. https://arxiv.org/abs/2203.02155.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. "BLEU: A Method for Automatic Evaluation of Machine Translation." **Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics**, 311–18. https://doi.org/10.3115/1073083.1073135.

Pardo, Fabio, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. 2018. "Time Limits in Reinforcement Learning." **International Conference on Machine Learning**, 4045–54. https://arxiv.org/abs/1712.00378.

Parikh, Ankur P, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. 2016. "A Decomposable Attention Model for Natural Language Inference." **ArXiv:1606.01933**. https://arxiv.org/abs/1606.01933.

Park, Taesung, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. 2019. "Semantic Image Synthesis with Spatially-Adaptive Normalization." **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 2337–46. https://doi.org/10.1109/cvpr.2019.00244.

Parmar, Gaurav, Richard Zhang, and Jun-Yan Zhu. 2022. "On Aliased Resizing and Surprising Subtleties in GAN Evaluation." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 11410–20. https://arxiv.org/abs/2104.11222.

Paster, Keiran, Sheila A. McIlraith, and Jimmy Ba. 2022. "You Can't Count on Luck: Why Decision Transformers and RvS Fail in Stochastic Environments." **Advances in Neural Information Processing Systems** 35.

Patarasuk, Pitch, and Xin Yuan. 2009. “Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations.” **Journal of Parallel and Distributed Computing** 69 (2): 117–24. https://doi.org/10.1016/j.jpdc.2008.09.002.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. “Curiosity-Driven Exploration by Self-Supervised Prediction.” **International Conference on Machine Learning**. https://arxiv.org/abs/1705.05363.

Peebles, William, and Saining Xie. 2023. “Scalable Diffusion Models with Transformers.” **Proceedings of the IEEE/CVF International Conference on Computer Vision**, 4195–205. https://arxiv.org/abs/2212.09748.

Peng, Bowen, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2024. “YaRN: Efficient Context Window Extension of Large Language Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2309.00071.

Peng, Bo, Ruichong Zhang, Daniel Goldstein, et al. 2025. “RWKV-7 ”Goose” with Expressive Dynamic State Evolution.” **arXiv Preprint arXiv:2503.14456**.

Perez, Ethan, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. 2018. “FiLM: Visual Reasoning with a General Conditioning Layer.” **Proceedings of the AAAI Conference on Artificial Intelligence** 32. https://arxiv.org/abs/1709.07871.

Pethick, Thomas, Wanyun Xie, Kimon Antonakopoulos, Zhenyu Zhu, Antonio Silveti-Falls, and Volkan Cevher. 2025. “Training Deep Learning Models with Norm-Constrained LMOs.” **International Conference on Machine Learning**.

Poli, Michael, Stefano Massaroli, Eric Nguyen, et al. 2023. “Hyena Hierarchy: Towards Larger Convolutional Language Models.” **International Conference on Machine Learning**. https://arxiv.org/abs/2302.10866.

Polyak, Adam, Amit Zohar, Andrew Brown, et al. 2024. “Movie Gen: A Cast of Media Foundation Models.” **arXiv Preprint arXiv:2410.13720**. https://arxiv.org/abs/2410.13720.

Polyak, Boris T. 1964. “Some Methods of Speeding up the Convergence of Iteration Methods.” **USSR Computational Mathematics and Mathematical Physics** 4 (5): 1–17. https://doi.org/10.1016/0041-5553(64)90137-5.

Polyak, Boris T., and Anatoli B. Juditsky. 1992. “Acceleration of Stochastic Approximation by Averaging.” **SIAM Journal on Control and Optimization** 30 (4): 838–55.

Pomerleau, Dean A. 1989. “ALVINN: An Autonomous Land Vehicle in a Neural Network.” In **Advances in Neural Information Processing Systems 1**, edited by David S Touretzky. Morgan Kaufmann.

Pomerleau, Dean A. 1991. “Efficient Training of Artificial Neural Networks for Autonomous Navigation.” **Neural Computation** 3 (1): 88–97.

Pontryagin, Lev S., Vladimir G. Boltyanskii, Revaz V. Gamkrelidze, and Evgenii F. Mishchenko. 1962. **The Mathematical Theory of Optimal Processes**. Interscience.

Popović, Maja. 2015. “ChrF: Character n-Gram F-Score for Automatic MT Evaluation.” **Proceedings of the Tenth Workshop on Statistical Machine Translation**, 392–95. https://doi.org/10.18653/v1/W15-3049.

Post, Matt. 2018. “A Call for Clarity in Reporting BLEU Scores.” **Proceedings of the Third Conference on Machine Translation: Research Papers**, 186–91. https://doi.org/10.18653/v1/W18-6319.

Prabhudesai, Mihir, Mengning Wu, Amir Zadeh, Katerina Fragkiadaki, and Deepak Pathak. 2025. “Diffusion Beats Autoregressive in Data-Constrained Settings.” **arXiv Preprint arXiv:2507.15857**. https://arxiv.org/abs/2507.15857.

Press, Ofir, Noah A. Smith, and Mike Lewis. 2022. “Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation.” **International Conference on Learning Representations**. https://arxiv.org/abs/2108.12409.

Press, Ofir, and Lior Wolf. 2017. “Using the Output Embedding to Improve Language Models.” **Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics**, 157–63.

Puterman, Martin L. 1994. **Markov Decision Processes: Discrete Stochastic Dynamic Programming**. John Wiley & Sons. https://doi.org/10.1002/9780470316887.

Qiu, Zihan, Zekun Wang, Bo Zheng, et al. 2025. “Gated Attention for Large Language Models: Non-Linearity, Sparsity, and Attention-Sink-Free.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/2505.06708.

Qwen Team. 2025. **Qwen3-Next: Towards Ultimate Training and Inference Efficiency**. Model release, https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Instruct.

Rabe, Markus N., and Charles Staats. 2021. “Self-Attention Does Not Need $O(n^2)$ Memory.” **arXiv Preprint arXiv:2112.05682**.

Rabiner, Lawrence, and Biing-Hwang Juang. 1993. **Fundamentals of Speech Recognition**. Prentice-Hall.

Radford, Alec, Jong Wook Kim, Chris Hallacy, et al. 2021. “Learning Transferable Visual Models from Natural Language Supervision.” **International Conference on Machine Learning**, 8748–63. https://proceedings.mlr.press/v139/radford21a.html.

Radford, Alec, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. “Robust Speech Recognition via Large-Scale Weak Supervision.” **International Conference on Machine Learning**. https://arxiv.org/abs/2212.04356.

Radford, Alec, Luke Metz, and Soumith Chintala. 2015. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” **ArXiv:1511.06434**. https://arxiv.org/abs/1511.06434.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. “Improving Language Understanding by Generative Pre-Training.” **OpenAI**. https://cdn.openai.c

om/research-covers/language-unsupervised/language_understanding_paper.pdf.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." **OpenAI Blog** 1 (8): 9. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Rafailov, Rafael, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." **Advances in Neural Information Processing Systems** 36.

Raffel, Colin, Noam Shazeer, Adam Roberts, et al. 2020. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." **Journal of Machine Learning Research** 21: 1–67. https://arxiv.org/abs/1910.10683.

Raffin, Antonin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." **Journal of Machine Learning Research** 22 (268): 1–8.

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." **SC20: International Conference for High Performance Computing, Networking, Storage and Analysis**, 1–16.

Rajbhandari, Samyam, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning." **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. https://arxiv.org/abs/2104.07857.

Ramapuram, Jason, Federico Danieli, Eeshan Dhekane, et al. 2025. "Theory, Analysis, and Best Practices for Sigmoid Self-Attention." **International Conference on Learning Representations**. https://arxiv.org/abs/2409.04431.

Ramdas, Aaditya, Sashank J. Reddi, Barnabás Póczos, Aarti Singh, and Larry Wasserman. 2015. "On the Decreasing Power of Kernel and Distance Based Nonparametric Hypothesis Tests in High Dimensions." **Proceedings of the AAAI Conference on Artificial Intelligence** 29.

Ramsauer, Hubert, Bernhard Schäfl, Johannes Lehner, et al. 2021. "Hopfield Networks Is All You Need." **International Conference on Learning Representations**. https://arxiv.org/abs/2008.02217.

Randløv, Jette, and Preben Alstrøm. 1998. "Learning to Drive a Bicycle Using Reinforcement Learning and Shaping." **Proceedings of the Fifteenth International Conference on Machine Learning**, 463–71.

Rashidinejad, Paria, Banghua Zhu, Cong Ma, Jiantao Jiao, and Stuart Russell. 2021. "Bridging Offline Reinforcement Learning and Imitation Learning: A Tale of Pessimism." **Advances in Neural Information Processing Systems** 34.

Rawlings, James B., David Q. Mayne, and Moritz Diehl. 2017. **Model Predictive Control:**

Theory, Computation, and Design. 2nd ed. Nob Hill Publishing.

Reddi, Sashank J, Satyen Kale, and Sanjiv Kumar. 2018. "On the Convergence of Adam and Beyond." **International Conference on Learning Representations**. https://arxiv.org/abs/1904.09237.

Rei, Ricardo, Craig Stewart, Ana C. Farinha, and Alon Lavie. 2020. "COMET: A Neural Framework for MT Evaluation." **Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing**, 2685–702. https://doi.org/10.18653/v1/2020.emnlp-main.213.

Ren, Jie, Samyam Rajbhandari, Reza Yazdani Aminabadi, et al. 2021. "ZeRO-Offload: Democratizing Billion-Scale Model Training." **2021 USENIX Annual Technical Conference (USENIX ATC 21)**, 551–64. https://www.usenix.org/conference/atc21/presentation/ren-jie.

Robbins, Herbert. 1952. "Some Aspects of the Sequential Design of Experiments." **Bulletin of the American Mathematical Society** 58 (5): 527–35. https://doi.org/10.1090/S0002-9904-1952-09620-8.

Robbins, Herbert. 1956. "An Empirical Bayes Approach to Statistics." **Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1**, 157–63.

Robbins, Herbert, and Sutton Monro. 1951. "A Stochastic Approximation Method." **The Annals of Mathematical Statistics** 22 (3): 400–407.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. "High-Resolution Image Synthesis with Latent Diffusion Models." **IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 10684–95.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. "U-Net: Convolutional Networks for Biomedical Image Segmentation." **Medical Image Computing and Computer-Assisted Intervention (MICCAI)**, Lecture notes in computer science, vol. 9351: 234–41. https://arxiv.org/abs/1505.04597.

Ross, Stéphane, and J Andrew Bagnell. 2010. "Efficient Reductions for Imitation Learning." **Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics**, Proceedings of machine learning research, vol. 9: 661–68.

Ross, Stéphane, Geoffrey J Gordon, and J Andrew Bagnell. 2011. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." **International Conference on Artificial Intelligence and Statistics**, 627–35.

Rouhani, Bita Darvish, Ritchie Zhao, Ankit More, et al. 2023. "Microscaling Data Formats for Deep Learning." **arXiv Preprint arXiv:2310.10537**.

Rubinstein, Reuven Y., and Dirk P. Kroese. 2004. **The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning**. Springer. https://doi.org/10.1007/978-1-4757-4321-0.

Rudin, Nikita, David Hoeller, Philipp Reist, and Marco Hutter. 2022. “Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning.” **Conference on Robot Learning**. https://arxiv.org/abs/2109.11978.

Rummery, Gavin A, and Mahesan Niranjan. 1994. **On-Line Q-Learning Using Connectionist Systems**. CUED/F-INFENG/TR 166. Cambridge University Engineering Department.

Sadat, Seyedmorteza, Otmar Hilliges, and Romann M. Weber. 2025. “Eliminating Oversaturation and Artifacts of High Guidance Scales in Diffusion Models.” **International Conference on Learning Representations**. https://arxiv.org/abs/2410.02416.

Saharia, Chitwan, William Chan, Saurabh Saxena, et al. 2022. “Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding.” **ArXiv:2205.11487**. https://arxiv.org/abs/2205.11487.

Sahoo, Subham Sekhar, Marianne Arriola, Yair Schiff, et al. 2024. “Simple and Effective Masked Diffusion Language Models.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2406.07524.

Sahoo, Subham Sekhar, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin Chiu, and Volodymyr Kuleshov. 2025. “The Diffusion Duality.” **International Conference on Machine Learning**.

Salimans, Tim, and Jonathan Ho. 2022. “Progressive Distillation for Fast Sampling of Diffusion Models.” **International Conference on Learning Representations**.

Samuel, Arthur L. 1959. “Some Studies in Machine Learning Using the Game of Checkers.” **IBM Journal of Research and Development** 3 (3): 210–29. https://doi.org/10.1147/rd.33.0210.

Sardana, Nikhil, Jacob Portes, Sasha Doubov, and Jonathan Frankle. 2024. “Beyond Chinchilla-Optimal: Accounting for Inference in Language Model Scaling Laws.” **International Conference on Machine Learning**.

Sauer, Axel, Frederic Boesel, Tim Dockhorn, Andreas Blattmann, Patrick Esser, and Robin Rombach. 2024. “Fast High-Resolution Image Synthesis with Latent Adversarial Diffusion Distillation.” **SIGGRAPH Asia 2024 Conference Papers**.

Sauer, Axel, Kashyap Chitta, Jens Müller, and Andreas Geiger. 2021. “Projected GANs Converge Faster.” **Advances in Neural Information Processing Systems** 34.

Sauer, Axel, Tero Karras, Samuli Laine, Andreas Geiger, and Timo Aila. 2023. “StyleGAN-T: Unlocking the Power of GANs for Fast Large-Scale Text-to-Image Synthesis.” **Proceedings of the 40th International Conference on Machine Learning**.

Sauer, Axel, Dominik Lorenz, Andreas Blattmann, and Robin Rombach. 2023. “Adversarial Diffusion Distillation.” **arXiv Preprint arXiv:2311.17042**.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 2016. “Prioritized Experience Replay.” **International Conference on Learning Representations**. https:

//arxiv.org/abs/1511.05952.

Schiff, Yair, Chia-Hsiang Kao, Aaron Gokaslan, Tri Dao, Albert Gu, and Volodymyr Kuleshov. 2024. “Caduceus: Bi-Directional Equivariant Long-Range DNA Sequence Modeling.” **International Conference on Machine Learning**. https://arxiv.org/abs/2403.03234.

Schlag, Imanol, Kazuki Irie, and Jürgen Schmidhuber. 2021. “Linear Transformers Are Secretly Fast Weight Programmers.” **International Conference on Machine Learning**.

Schmidt, Robin M., Frank Schneider, and Philipp Hennig. 2021. “Descending Through a Crowded Valley — Benchmarking Deep Learning Optimizers.” **Proceedings of the 38th International Conference on Machine Learning**, 9367–76. https://arxiv.org/abs/2007.01547.

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, et al. 2020. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.” **Nature** 588 (7839): 604–9. https://doi.org/10.1038/s41586-020-03051-0.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. “Trust Region Policy Optimization.” **International Conference on Machine Learning**, 1889–97.

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2016. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” **International Conference on Learning Representations**.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. “Proximal Policy Optimization Algorithms.” **arXiv Preprint arXiv:1707.06347**.

Schuster, Mike, and Kuldip K Paliwal. 1997. “Bidirectional Recurrent Neural Networks.” **IEEE Transactions on Signal Processing** 45 (11): 2673–81. https://doi.org/10.1109/78.650093.

Semenov, Andrei, Matteo Pagliardini, and Martin Jaggi. 2025. “Benchmarking Optimizers for Large Language Model Pretraining.” **arXiv Preprint arXiv:2509.01440**.

Sergeev, Alexander, and Mike Del Balso. 2018. “Horovod: Fast and Easy Distributed Deep Learning in TensorFlow.” **ArXiv:1802.05799**. https://arxiv.org/abs/1802.05799.

Shah, Ishaan, Anthony M. Polloreno, Karl Stratos, et al. 2025. “Practical Efficiency of Muon for Pretraining.” **arXiv Preprint arXiv:2505.02222**.

Shah, Jay, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. “FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2407.08608.

Shallue, Christopher J., Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. “Measuring the Effects of Data Parallelism on Neural Network Training.” **Journal of Machine Learning Research** 20 (112): 1–49. https://arxiv.org/abs/

1811.03600.

Shannon, Matt, Ben Poole, Soroosh Mariooryad, et al. 2020. “Non-Saturating GAN Training as Divergence Minimization.” **arXiv Preprint arXiv:2010.08029**.

Shao, Zhihong, Peiyi Wang, Qihao Zhu, et al. 2024. “DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models.” **arXiv Preprint arXiv:2402.03300**.

Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani. 2018. “Self-Attention with Relative Position Representations.” **ArXiv:1803.02155**. https://arxiv.org/abs/1803.02155.

Shazeer, Noam. 2019. “Fast Transformer Decoding: One Write-Head Is All You Need.” **arXiv Preprint arXiv:1911.02150**.

Shazeer, Noam. 2020. “GLU Variants Improve Transformer.” **ArXiv:2002.05202**. https://arxiv.org/abs/2002.05202.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, et al. 2017. “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.” **International Conference on Learning Representations**.

Shazeer, Noam, and Mitchell Stern. 2018. “Adafactor: Adaptive Learning Rates with Sublinear Memory Cost.” **International Conference on Machine Learning**. https://arxiv.org/abs/1804.04235.

Shen, Lingfeng, Aayush Mishra, and Daniel Khashabi. 2024. “Position: Do Pretrained Transformers Learn in-Context by Gradient Descent?” **International Conference on Machine Learning**, Proceedings of machine learning research, vol. 235: 44712–40. https://arxiv.org/abs/2310.08540.

Shi, Jiaxin, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. 2024. “Simplified and Generalized Masked Diffusion for Discrete Data.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2406.04329.

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.” **ArXiv:1909.08053**. https://arxiv.org/abs/1909.08053.

Siems, Julien, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. 2025. “DeltaProduct: Improving State-Tracking in Linear RNNs via Householder Products.” **arXiv Preprint arXiv:2502.10297**.

Silver, David, Aja Huang, Chris J Maddison, et al. 2016. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” **Nature** 529 (7587): 484–89. https://doi.org/10.1038/nature16961.

Silver, David, Thomas Hubert, Julian Schrittwieser, et al. 2018. “A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play.” **Science** 362 (6419): 1140–44. https://doi.org/10.1126/science.aar6404.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. "Deterministic Policy Gradient Algorithms." **Proceedings of the 31st International Conference on Machine Learning**, Proceedings of machine learning research, vol. 32: 387–95. https://proceedings.mlr.press/v32/silver14.html.

Silver, David, Julian Schrittwieser, Karen Simonyan, et al. 2017. "Mastering the Game of Go Without Human Knowledge." **Nature** 550 (7676): 354–59. https://doi.org/10.1038/nature24270.

Silverman, B. W. 1986. **Density Estimation for Statistical and Data Analysis**. Chapman; Hall.

Smith, Jimmy T. H., Andrew Warrington, and Scott W. Linderman. 2023. "Simplified State Space Layers for Sequence Modeling." **International Conference on Learning Representations**. https://arxiv.org/abs/2208.04933.

Smith, Samuel L., Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. "Don't Decay the Learning Rate, Increase the Batch Size." **International Conference on Learning Representations**. https://arxiv.org/abs/1711.00489.

Sohl-Dickstein, Jascha, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. "Deep Unsupervised Learning Using Nonequilibrium Thermodynamics." **International Conference on Machine Learning**, 2256–65. https://proceedings.mlr.press/v37/sohl-dickstein15.html.

Sønderby, Casper Kaae, Jose Caballero, Lucas Theis, Wenzhe Shi, and Ferenc Huszár. 2017. "Amortised MAP Inference for Image Super-Resolution." **International Conference on Learning Representations**.

Song, Jiaming, Chenlin Meng, and Stefano Ermon. 2021. "Denoising Diffusion Implicit Models." **International Conference on Learning Representations**.

Song, Yang. 2021. **Generative Modeling by Estimating Gradients of the Data Distribution**. Blog post, https://yang-song.net/blog/2021/score/.

Song, Yang, and Prafulla Dhariwal. 2024. "Improved Techniques for Training Consistency Models." **International Conference on Learning Representations**. https://arxiv.org/abs/2310.14189.

Song, Yang, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. 2023. "Consistency Models." **Proceedings of the 40th International Conference on Machine Learning**.

Song, Yang, Conor Durkan, Iain Murray, and Stefano Ermon. 2021. "Maximum Likelihood Training of Score-Based Diffusion Models." **Advances in Neural Information Processing Systems** 34.

Song, Yang, and Stefano Ermon. 2019. "Generative Modeling by Estimating Gradients of the Data Distribution." **Advances in Neural Information Processing Systems** 32. https://arxiv.org/abs/1907.05600.

Song, Yang, and Stefano Ermon. 2020. “Improved Techniques for Training Score-Based Generative Models.” **Advances in Neural Information Processing Systems** 33. https://arxiv.org/abs/2006.09011.

Song, Yang, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. 2021. “Score-Based Generative Modeling Through Stochastic Differential Equations.” **International Conference on Learning Representations**. https://arxiv.org/abs/2011.13456.

Stein, George, Jesse C. Cresswell, Rasa Hosseinzadeh, et al. 2023. “Exposing Flaws of Generative Model Evaluation Metrics and Their Unfair Treatment of Diffusion Models.” **Advances in Neural Information Processing Systems** 36. https://arxiv.org/abs/2306.04675.

Stich, Sebastian U. 2019. “Local SGD Converges Fast and Communicates Little.” **International Conference on Learning Representations**. https://arxiv.org/abs/1805.09767.

Strehl, Alexander L., and Michael L. Littman. 2008. “An Analysis of Model-Based Interval Estimation for Markov Decision Processes.” **Journal of Computer and System Sciences** 74 (8): 1309–31. https://doi.org/10.1016/j.jcss.2007.08.009.

Su, Jianlin, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2024. “RoFormer: Enhanced Transformer with Rotary Position Embedding.” **Neurocomputing** 568: 127063. https://arxiv.org/abs/2104.09864.

Sun, Mingjie, Xinlei Chen, J. Zico Kolter, and Zhuang Liu. 2024. “Massive Activations in Large Language Models.” **Conference on Language Modeling (COLM)**. https://arxiv.org/abs/2402.17762.

Sun, Ruoyu, Tiantian Fang, and Alexander Schwing. 2020. “Towards a Better Global Loss Landscape of GANs.” **Advances in Neural Information Processing Systems** 33.

Sun, Yu, Xinhao Li, Karan Dalal, et al. 2024. “Learning to (Learn at Test Time): RNNs with Expressive Hidden States.” **arXiv Preprint arXiv:2407.04620**.

Sun, Yutao, Li Dong, Shaohan Huang, et al. 2023. “Retentive Network: A Successor to Transformer for Large Language Models.” **arXiv Preprint arXiv:2307.08621**.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le. 2014. “Sequence to Sequence Learning with Neural Networks.” **Advances in Neural Information Processing Systems**, 3104–12. https://doi.org/10.5555/2969033.2969173.

Sutton, Richard S. 1988. “Learning to Predict by the Methods of Temporal Differences.” **Machine Learning** 3 (1): 9–44. https://doi.org/10.1007/BF00115009.

Sutton, Richard S. 1990. “Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming.” **Proceedings of the Seventh International Conference on Machine Learning**, 216–24.

Sutton, Richard S, and Andrew G Barto. 2018. **Reinforcement Learning: An Introduction**.

2nd ed. MIT Press. http://incompleteideas.net/book/RLbook2020.pdf.

Sutton, Richard S, David McAllester, Satinder Singh, and Yishay Mansour. 2000. “Policy Gradient Methods for Reinforcement Learning with Function Approximation.” **Advances in Neural Information Processing Systems** 12.

Szepesvári, Csaba. 2010. **Algorithms for Reinforcement Learning**. Synthesis Lectures on Artificial Intelligence and Machine Learning 9. Morgan; Claypool. https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf.

Tallec, Corentin, and Yann Ollivier. 2018. “Can Recurrent Neural Networks Warp Time?” **International Conference on Learning Representations**. https://arxiv.org/abs/1804.11188.

Tang, Yunhao, and Rémi Munos. 2025. “On a Few Pitfalls in KL Divergence Gradient Estimation for RL.” **arXiv Preprint arXiv:2506.09477**.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. “Efficient Transformers: A Survey.” **ArXiv:2009.06732**. https://arxiv.org/abs/2009.06732.

Team OLMo. 2025. “Olmo 3.” **arXiv Preprint arXiv:2512.13961**.

Team OLMo, Pete Walsh, Luca Soldaini, et al. 2025. “2 OLMo 2 Furious.” **arXiv Preprint arXiv:2501.00656**.

Tedrake, Russ. 2023. **Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation**. MIT. https://underactuated.csail.mit.edu/.

Tesauro, Gerald. 1995. “Temporal Difference Learning and TD-Gammon.” **Communications of the ACM** 38 (3): 58–68. https://doi.org/10.1145/203330.203343.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. 2005. “Optimization of Collective Communication Operations in MPICH.” **The International Journal of High Performance Computing Applications** 19 (1): 49–66. https://doi.org/10.1177/1094342005051521.

Theis, Lucas, Aäron van den Oord, and Matthias Bethge. 2016. “A Note on the Evaluation of Generative Models.” **International Conference on Learning Representations**.

Thomas, Philip. 2014. “Bias in Natural Actor-Critic Algorithms.” In **Proceedings of the 31st International Conference on Machine Learning**, edited by Eric P. Xing and Tony Jebara, vol. 32. Proceedings of Machine Learning Research 1.

Thompson, William R. 1933. “On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples.” **Biometrika** 25 (3/4): 285–94.

Thrun, Sebastian, and Anton Schwartz. 1993. “Issues in Using Function Approximation for Reinforcement Learning.” **Proceedings of the 4th Connectionist Models Summer School**, 255–63.

Tieleman, Tijmen. 2008. “Training Restricted Boltzmann Machines Using Approximations to the Likelihood Gradient.” **Proceedings of the 25th International Conference on Machine Learning**, 1064–71.

Tieleman, Tijmen, and Geoffrey Hinton. 2012. “Divide the Gradient by a Running Average of Its Recent Magnitude.” In **COURSERA: Neural Networks for Machine Learning, Lecture 6.5-Rmsprop**. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Tillet, Philippe, H. T. Kung, and David Cox. 2019. “Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations.” **3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)**, 10–19. https://doi.org/10.1145/3315508.3329973.

Tobin, Josh, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. “Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World.” **IEEE/RSJ International Conference on Intelligent Robots and Systems**. https://arxiv.org/abs/1703.06907.

Touvron, Hugo, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. “Training Data-Efficient Image Transformers & Distillation Through Attention.” **International Conference on Machine Learning**, 10347–57. https://proceedings.mlr.press/v139/touvron21a.html.

Towers, Mark, Ariel Kwiatkowski, Jordan Terry, et al. 2024. “Gymnasium: A Standard Interface for Reinforcement Learning Environments.” **arXiv Preprint arXiv:2407.17032**.

Tsitsiklis, John N. 1994. “Asynchronous Stochastic Approximation and Q-Learning.” **Machine Learning** 16: 185–202. https://doi.org/10.1007/BF00993306.

Tsitsiklis, John N, and Benjamin Van Roy. 1997. “An Analysis of Temporal-Difference Learning with Function Approximation.” **IEEE Transactions on Automatic Control** 42 (5): 674–90. https://doi.org/10.1109/9.580874.

Uria, Benigno, Iain Murray, and Hugo Larochelle. 2014. “A Deep and Tractable Density Estimator.” **Proceedings of the 31st International Conference on Machine Learning**, Proceedings of machine learning research, vol. 32: 467–75.

Van Hasselt, Hado, Arthur Guez, and David Silver. 2016. “Deep Reinforcement Learning with Double Q-Learning.” **Proceedings of the AAAI Conference on Artificial Intelligence** 30.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. “Attention Is All You Need.” **Advances in Neural Information Processing Systems**, 5998–6008. https://doi.org/10.5555/3295222.3295349.

Vincent, Pascal. 2011. “A Connection Between Score Matching and Denoising Autoencoders.” **Neural Computation** 23 (7): 1661–74.

Voita, Elena, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. “Analyz-

ing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned." **Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics**. https://arxiv.org/abs/1905.09418.

Volkov, Vasily. 2016. "Understanding Latency Hiding on GPUs." PhD thesis, University of California, Berkeley.

Vries, Harm de, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron Courville. 2017. "Modulating Early Visual Processing by Language." **Advances in Neural Information Processing Systems** 30. https://arxiv.org/abs/1707.00683.

Vyas, Nikhil, Depen Morwani, Rosie Zhao, et al. 2024. "SOAP: Improving and Stabilizing Shampoo Using Adam." **arXiv Preprint arXiv:2409.11321**.

Waleffe, Roger, Wonmin Byeon, Duncan Riach, et al. 2024. "An Empirical Study of Mamba-Based Language Models." **arXiv Preprint arXiv:2406.07887**.

Wallace, Bram, Meihua Dang, Rafael Rafailov, et al. 2024. "Diffusion Model Alignment Using Direct Preference Optimization." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 8228–38. https://arxiv.org/abs/2311.12908.

Wallace, Bram, Akash Gokul, and Nikhil Naik. 2023. "EDICT: Exact Diffusion Inversion via Coupled Transformations." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**.

Wang, Guanghan, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. 2025. "Remasking Discrete Diffusion Models with Inference-Time Scaling." **Advances in Neural Information Processing Systems**.

Wang, Hongyu, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. 2024. "DeepNet: Scaling Transformers to 1,000 Layers." **IEEE Transactions on Pattern Analysis and Machine Intelligence** 46: 6761–74. https://arxiv.org/abs/2203.00555.

Wang, Jie, Tao Ji, Yuanbin Wu, et al. 2024. "Length Generalization of Causal Transformers Without Position Encoding." **Findings of the Association for Computational Linguistics: ACL 2024**, 14024–40. https://doi.org/10.18653/v1/2024.findings-acl.834.

Wang, Junxiong, Daniele Paliotta, Avner May, Alexander M. Rush, and Tri Dao. 2024. "The Mamba in the Llama: Distilling and Accelerating Hybrid Models." **Advances in Neural Information Processing Systems**.

Wang, Ke Alexander, Jiaxin Shi, and Emily B. Fox. 2025. "Test-Time Regression: A Unifying Framework for Designing Sequence Models with Associative Memory." **arXiv Preprint arXiv:2501.12352**.

Wang, Kevin, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. 2023. "Interpretability in the Wild: A Circuit for Indirect Object Identification in GPT-2 Small." **International Conference on Learning Representations**. https://arxiv.org/abs/2211.00593.

Wang, Lean, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. 2024. “Auxiliary-Loss-Free Load Balancing Strategy for Mixture-of-Experts.” **arXiv Preprint arXiv:2408.15664**.

Wang, Sinong, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. “Linformer: Self-Attention with Linear Complexity.” **arXiv Preprint arXiv:2006.04768**.

Wang, Xi, and Laurence Aitchison. 2025. “How to Set AdamW’s Weight Decay as You Scale Model and Dataset Size.” **International Conference on Machine Learning**. https://arxiv.org/abs/2405.13698.

Wang, Zhendong, Jonathan J. Hunt, and Mingyuan Zhou. 2023. “Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning.” **International Conference on Learning Representations**. https://arxiv.org/abs/2208.06193.

Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. “Dueling Network Architectures for Deep Reinforcement Learning.” **International Conference on Machine Learning**, 1995–2003.

Warner, Benjamin, Antoine Chaffin, Benjamin Clavié, et al. 2024. “Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference.” **arXiv Preprint arXiv:2412.13663**.

Watkins, Christopher J C H. 1989. “Learning from Delayed Rewards.” PhD thesis, King’s College, University of Cambridge. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Watkins, Christopher JCH, and Peter Dayan. 1992. “Q-Learning.” **Machine Learning** 8 (3–4): 279–92. https://doi.org/10.1007/bf00992698.

Watson, Geoffrey S. 1964. “Smooth Regression Analysis.” **Sankhyā: The Indian Journal of Statistics, Series A**, 359–72. https://doi.org/10.1007/bf02868765.

Weed, Jonathan, and Francis Bach. 2019. “Sharp Asymptotic and Finite-Sample Rates of Convergence of Empirical Measures in Wasserstein Distance.” **Bernoulli** 25 (4A): 2620–48.

Weinberger, Kilian Q., and Gerald Tesauro. 2007. “Metric Learning for Kernel Regression.” **Proceedings of the 11th International Conference on Artificial Intelligence and Statistics**, Proceedings of machine learning research, vol. 2: 612–19.

Welch, Lloyd R. 1974. “Lower Bounds on the Maximum Cross Correlation of Signals.” **IEEE Transactions on Information Theory** 20 (3): 397–99. https://doi.org/10.1109/TIT.1974.1055219.

Wen, Kaiyue, David Hall, Tengyu Ma, and Percy Liang. 2025. “Fantastic Pretraining Optimizers and Where to Find Them.” **arXiv Preprint arXiv:2509.02046**.

Wen, Kaiyue, Zhiyuan Li, Jason Wang, David Hall, Percy Liang, and Tengyu Ma. 2024. “Understanding Warmup-Stable-Decay Learning Rates: A River Valley Loss Landscape Perspective.” **arXiv Preprint arXiv:2410.05192**.

Weng, Lilian. 2021. **What Are Diffusion Models?** Blog post, https://lilianweng.github.io/posts/2021-07-11-diffusion-models/.

Wenliang, Li K., and Heishiro Kanagawa. 2020. "Blindness of Score-Based Methods to Isolated Components and Mixing Proportions." **arXiv Preprint arXiv:2008.10087**. https://arxiv.org/abs/2008.10087.

Widrow, Bernard, and Marcian E. Hoff. 1960. "Adaptive Switching Circuits." **IRE WESCON Convention Record** 4: 96–104.

Wiegreffe, Sarah, and Yuval Pinter. 2019. "Attention Is Not Not Explanation." **Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing**, 11–20.

Wigner, Eugene P. 1958. "On the Distribution of the Roots of Certain Symmetric Matrices." **Ann. Math.**, 325–27. https://doi.org/10.2307/1970079.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." **Machine Learning** 8: 229–56.

Williams, Samuel, Andrew Waterman, and David Patterson. 2009. **Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures**. Lawrence Berkeley National Lab. https://doi.org/10.2172/1407078.

Wortsman, Mitchell, Gabriel Ilharco, Samir Yitzhak Gadre, et al. 2022. "Model Soups: Averaging Weights of Multiple Fine-Tuned Models Improves Accuracy Without Increasing Inference Time." **International Conference on Machine Learning**.

Wu, Chengyue, Hao Zhang, Shuchen Xue, et al. 2026. "Fast-dLLM: Training-Free Acceleration of Diffusion LLM by Enabling KV Cache and Parallel Decoding." **International Conference on Learning Representations**.

Xiao, Guangxuan, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. "Efficient Streaming Language Models with Attention Sinks." **International Conference on Learning Representations**.

Xiong, Ruibin, Yunchang Yang, Di He, et al. 2020. "On Layer Normalization in the Transformer Architecture." **International Conference on Machine Learning**, 10524–33. https://proceedings.mlr.press/v119/xiong20b.html.

Xiong, Wenhan, Jingyu Liu, Igor Molybog, et al. 2024. "Effective Long-Context Scaling of Foundation Models." **Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics**, 4643–63. https://doi.org/10.18653/v1/2024.naacl-long.260.

Xu, Yilun, Mingyang Deng, Xiang Cheng, Yonglong Tian, Ziming Liu, and Tommi Jaakkola. 2023. "Restart Sampling for Improving Generative Processes." **Advances in Neural Information Processing Systems** 36.

Xu, Yuanzhong, HyoukJoong Lee, Dehao Chen, et al. 2021. "GSPMD: General and Scalable Parallelization for ML Computation Graphs." **ArXiv:2105.04663**. https://arxiv.org/abs/

2105.04663.

Yang, An, Anfeng Li, Baosong Yang, et al. 2025. “Qwen3 Technical Report.” **arXiv Preprint arXiv:2505.09388**.

Yang, Charlene, Yunsong Wang, Steven Farrell, Thorsten Kurth, and Samuel Williams. 2020. “Hierarchical Roofline Performance Analysis for Deep Learning Applications.” **arXiv Preprint arXiv:2009.05257**.

Yang, Greg, and Edward J. Hu. 2021. “Tensor Programs IV: Feature Learning in Infinite-Width Neural Networks.” **International Conference on Machine Learning**, 11727–37.

Yang, Greg, Edward J. Hu, Igor Babuschkin, et al. 2022. “Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer.” **arXiv Preprint arXiv:2203.03466**.

Yang, Greg, James B. Simon, and Jeremy Bernstein. 2023. “A Spectral Condition for Feature Learning.” **arXiv Preprint arXiv:2310.17813**.

Yang, Greg, Dingli Yu, Chen Zhu, and Soufiane Hayou. 2024. “Tensor Programs VI: Feature Learning in Infinite-Depth Neural Networks.” **International Conference on Learning Representations**. https://arxiv.org/abs/2310.02244.

Yang, Songlin, Jan Kautz, and Ali Hatamizadeh. 2025. “Gated Delta Networks: Improving Mamba2 with Delta Rule.” **International Conference on Learning Representations**. https://arxiv.org/abs/2412.06464.

Yang, Songlin, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. 2024. “Gated Linear Attention Transformers with Hardware-Efficient Training.” **International Conference on Machine Learning**, 56501–23. https://arxiv.org/abs/2312.06635.

Yang, Songlin, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. 2024. “Parallelizing Linear Transformers with the Delta Rule over Sequence Length.” **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2406.06484.

Ye, Jiacheng, Zhihui Xie, Lin Zheng, et al. 2025. “Dream 7B: Diffusion Large Language Models.” **arXiv Preprint arXiv:2508.15487**. https://arxiv.org/abs/2508.15487.

Ye, Tianzhu, Li Dong, Yuqing Xia, et al. 2025. “Differential Transformer.” **International Conference on Learning Representations**. https://arxiv.org/abs/2410.05258.

Yin, Tianwei, Michaël Gharbi, Taesung Park, et al. 2024. “Improved Distribution Matching Distillation for Fast Image Synthesis.” **Advances in Neural Information Processing Systems** 37.

Yin, Tianwei, Michaël Gharbi, Richard Zhang, et al. 2024. “One-Step Diffusion with Distribution Matching Distillation.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 6613–23. https://arxiv.org/abs/2311.18828.

You, Yang, Igor Gitman, and Boris Ginsburg. 2017. “Large Batch Training of Convolutional

Networks." **ArXiv:1708.03888**. https://arxiv.org/abs/1708.03888.

Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. "Orca: A Distributed Serving System for Transformer-Based Generative Models." **16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)**, 521–38. https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, Qiying, Zheng Zhang, Ruofei Zhu, et al. 2025. "DAPO: An Open-Source LLM Reinforcement Learning System at Scale." **arXiv Preprint arXiv:2503.14476**.

Yu, Sihyun, Sangkyung Kwak, Huiwon Jang, et al. 2025. "Representation Alignment for Generation: Training Diffusion Transformers Is Easier Than You Think." **International Conference on Learning Representations**. https://arxiv.org/abs/2410.06940.

Yu, Tianhe, Garrett Thomas, Lantao Yu, et al. 2020. "MOPO: Model-Based Offline Policy Optimization." **Advances in Neural Information Processing Systems** 33.

Yuan, Jingyang, Huazuo Gao, Damai Dai, et al. 2025. "Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention." **Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)**, 23078–97. https://doi.org/10.18653/v1/2025.acl-long.1126.

Zaheer, Manzil, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. 2018. "Adaptive Methods for Nonconvex Optimization." **Advances in Neural Information Processing Systems**, 9793–803. https://proceedings.neurips.cc/paper/2018/hash/90365351ccc7437a1309dc64e4db32a3-Abstract.html.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." **Advances in Neural Information Processing Systems** 32.

Zhang, Fred, and Neel Nanda. 2024. "Towards Best Practices of Activation Patching in Language Models: Metrics and Methods." **International Conference on Learning Representations**. https://arxiv.org/abs/2309.16042.

Zhang, Han, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. 2019. "Self-Attention Generative Adversarial Networks." **Proceedings of the 36th International Conference on Machine Learning**, 7354–63. https://arxiv.org/abs/1805.08318.

Zhang, Hanlin, Depen Morwani, Nikhil Vyas, et al. 2024. "How Does Critical Batch Size Scale in Pre-Training?" **arXiv Preprint arXiv:2410.21676**.

Zhang, Jingzhao, Sai Praneeth Karimireddy, Andreas Veit, et al. 2020. "Why Are Adaptive Methods Good for Attention Models?" **Advances in Neural Information Processing Systems** 33.

Zhang, Lvmin, Anyi Rao, and Maneesh Agrawala. 2023. "Adding Conditional Control to Text-to-Image Diffusion Models." **Proceedings of the IEEE/CVF International Conference on Computer Vision**, 3836–47. https://arxiv.org/abs/2302.05543.

Zhang, Qinsheng, and Yongxin Chen. 2023. "Fast Sampling of Diffusion Models with Ex-

ponential Integrator." **International Conference on Learning Representations**. https://arxiv.org/abs/2204.13902.

Zhang, Susan, Stephen Roller, Naman Goyal, et al. 2022. "OPT: Open Pre-Trained Transformer Language Models." **ArXiv:2205.01068**. https://arxiv.org/abs/2205.01068.

Zhang, Yushun, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. 2024. "Why Transformers Need Adam: A Hessian Perspective." **Advances in Neural Information Processing Systems** 37. https://arxiv.org/abs/2402.16788.

Zhao, Shengyu, Zhijian Liu, Ji Lin, Jun-Yan Zhu, and Song Han. 2020. "Differentiable Augmentation for Data-Efficient GAN Training." **Advances in Neural Information Processing Systems** 33.

Zhao, Wenliang, Lujia Bai, Yongming Rao, Jie Zhou, and Jiwen Lu. 2023. "UniPC: A Unified Predictor-Corrector Framework for Fast Sampling of Diffusion Models." **Advances in Neural Information Processing Systems** 36. https://arxiv.org/abs/2302.04867.

Zhao, Yanli, Andrew Gu, Rohan Varma, et al. 2023. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel." **Proceedings of the VLDB Endowment** 16 (12): 3848–60. https://doi.org/10.14778/3611540.3611569.

Zheng, Boyang, Nanye Ma, Shengbang Tong, and Saining Xie. 2026. "Diffusion Transformers with Representation Autoencoders." **International Conference on Learning Representations**. https://arxiv.org/abs/2510.11690.

Zheng, Chujie, Shixuan Liu, Mingze Li, et al. 2025. "Group Sequence Policy Optimization." **arXiv Preprint arXiv:2507.18071**.

Zheng, Kaiwen, Yongxin Chen, Huayu Chen, et al. 2025. "Direct Discriminative Optimization: Your Likelihood-Based Visual Generative Model Is Secretly a GAN Discriminator." **Proceedings of the 42nd International Conference on Machine Learning**.

Zheng, Kaiwen, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. 2025. "Masked Diffusion Models Are Secretly Time-Agnostic Masked Models and Exploit Inaccurate Categorical Sampling." **International Conference on Learning Representations**.

Zheng, Kaiwen, Cheng Lu, Jianfei Chen, and Jun Zhu. 2023. "Improved Techniques for Maximum Likelihood Estimation for Diffusion ODEs." **International Conference on Machine Learning**. https://arxiv.org/abs/2305.03935.

Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks." **Proceedings of the IEEE International Conference on Computer Vision**, 2223–32. https://doi.org/10.1109/iccv.2017.244.

Zhu, Lianghui, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. 2024. "Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model." **International Conference on Machine Learning**. https://arxiv.org/abs/

2401.09417.

Ziebart, Brian D, Andrew Maas, J Andrew Bagnell, and Anind K Dey. 2008. “Maximum Entropy Inverse Reinforcement Learning.” **Proceedings of the AAAI Conference on Artificial Intelligence**, 1433–38.