Dive into Deep Learning

BASICS

Alexander J Smola

Aston Zhang · Zachary C Lipton · Mu Li · Alexander J Smola (1st edition)

Dive into Deep Learning（2026.11.27 版）。

Table of contents

基礎 **1**

1 準備 **3**
1.1 データ操作 . . . 3
1.1.1 テンソルの生成 . . . 4
1.1.2 形状変換とインデックス参照 . . . 6
1.1.3 演算 . . . 11
1.1.4 メモリと変換 . . . 14
1.1.5 まとめ . . . 16
1.1.6 演習問題 . . . 16
1.2 データの前処理 . . . 17
1.2.1 読み込みと確認 . . . 18
1.2.2 クリーニングとエンコーディング . . . 20
1.2.3 スケーリングと変換 . . . 23
1.2.4 まとめ . . . 25
1.2.5 演習問題 . . . 25
1.3 線形代数 . . . 26
1.3.1 スカラー、ベクトル、行列、テンソル . . . 27
1.3.2 要素ごとの演算とリダクション . . . 29
1.3.3 積 . . . 31
1.3.4 ノルム . . . 33
1.3.5 固有ベクトルと固有値 . . . 36
1.3.6 まとめ . . . 38
1.3.7 演習問題 . . . 38
1.4 微分積分 . . . 40
1.4.1 導関数 . . . 41
1.4.2 偏微分と勾配 . . . 44
1.4.3 連鎖律 . . . 49
1.4.4 まとめ . . . 50
1.4.5 演習問題 . . . 50
1.5 自動微分 . . . 52
1.5.1 勾配の計算 . . . 52
1.5.2 グラフの制御 . . . 55
1.5.3 プログラムの微分 . . . 56
1.5.4 フォワードモードとリバースモード . . . 57
1.5.5 まとめ . . . 58
1.5.6 演習問題 . . . 58

1.6 確率と統計 59
1.6.1 サンプリング 59
1.6.2 事象と確率変数 61
1.6.3 期待値と分散 63
1.6.4 推定誤差 64
1.6.5 条件付き確率 66
1.6.6 まとめ 68
1.6.7 演習問題 69

2 線形回帰 **71**
2.1 線形回帰 71
2.1.1 モデルと損失 72
2.1.2 損失の最小化 75
2.1.3 二乗損失とガウスノイズ 80
2.1.4 ニューラルネットワークとしての線形回帰 83
2.1.5 まとめ 84
2.1.6 演習問題 84
2.2 合成データ 85
2.2.1 データの生成 86
2.2.2 データの観察 87
2.2.3 パラメータの復元 87
2.2.4 まとめ 88
2.2.5 演習問題 88
2.3 線形回帰のスクラッチ実装 89
2.3.1 モデル、損失、勾配 89
2.3.2 スクラッチからの訓練 91
2.3.3 実装の検証 92
2.3.4 まとめ 95
2.3.5 演習問題 96
2.4 実装のためのオブジェクト指向設計 97
2.4.1 ユーティリティ 97
2.4.2 モデル 99
2.4.3 データ 101
2.4.4 訓練 102
2.4.5 線形回帰の再訓練 106
2.4.6 まとめ 107
2.4.7 演習問題 108
2.5 線形回帰の簡潔な実装 108
2.5.1 全結合層 109
2.5.2 損失関数 109
2.5.3 オプティマイザ 110
2.5.4 フレームワークの構成要素を用いた訓練 110
2.5.5 まとめ 111
2.5.6 演習問題 111
2.6 汎化 112
2.6.1 訓練誤差と汎化誤差 112
2.6.2 学習不足と過学習 113
2.6.3 モデル選択 119
2.6.4 まとめ 123
2.6.5 演習問題 123

2.7 重み減衰 . . . 125
2.7.1 ノルムペナルティ . . . 125
2.7.2 重み減衰のスクラッチ実装 . . . 127
2.7.3 固有ベクトルに沿った縮小 . . . 130
2.7.4 簡潔な実装 . . . 134
2.7.5 まとめ . . . 136
2.7.6 演習問題 . . . 136

3 線形分類 139
3.1 ソフトマックス回帰 . . . 140
3.1.1 モデル . . . 140
3.1.2 交差エントロピー損失 . . . 143
3.1.3 正解率と確信度 . . . 145
3.1.4 まとめ . . . 147
3.1.5 演習問題 . . . 147
3.2 画像分類データ . . . 149
3.2.1 データセットの読み込み . . . 150
3.2.2 ミニバッチの読み込み . . . 151
3.2.3 画像の確認 . . . 152
3.2.4 まとめ . . . 153
3.2.5 演習問題 . . . 153
3.3 ソフトマックス回帰の実装 . . . 154
3.3.1 ソフトマックス . . . 154
3.3.2 損失と正解率 . . . 155
3.3.3 モデルと損失 . . . 157
3.3.4 訓練と評価 . . . 158
3.3.5 簡潔な実装 . . . 162
3.3.6 まとめ . . . 164
3.3.7 演習問題 . . . 165
3.4 分類器の評価 . . . 166
3.4.1 正解率の限界 . . . 167
3.4.2 較正 . . . 168
3.4.3 まとめ . . . 171
3.4.4 演習問題 . . . 171
3.5 分類における汎化 . . . 173
3.5.1 テストセット . . . 173
3.5.2 テストセットの再利用 . . . 175
3.5.3 統計学習理論 . . . 177
3.5.4 まとめ . . . 178
3.5.5 演習問題 . . . 179
3.6 環境と分布シフト . . . 180
3.6.1 分布シフトの種類 . . . 180
3.6.2 共変量シフトの補正 . . . 183
3.6.3 ラベルシフトの補正 . . . 188
3.6.4 概念シフトとフィードバック . . . 189
3.6.5 まとめ . . . 190
3.6.6 演習問題 . . . 190

4 多層パーセプトロン 193
4.1 多層パーセプトロン . . . 194

4.1.1 線形モデルの限界：XOR 問題 . . . 195
4.1.2 隠れ層 . . . 196
4.1.3 表現力 . . . 198
4.1.4 活性化関数 . . . 199
4.1.5 まとめ . . . 201
4.1.6 演習問題 . . . 201
4.2 実装 . . . 202
4.2.1 スクラッチ実装 . . . 202
4.2.2 簡潔な実装 . . . 205
4.2.3 ソフトマックス回帰との比較 . . . 206
4.2.4 まとめ . . . 209
4.2.5 演習問題 . . . 209
4.3 誤差逆伝播法 . . . 210
4.3.1 計算グラフ . . . 210
4.3.2 MLP における誤差逆伝播法 . . . 211
4.3.3 メモリと計算量 . . . 215
4.3.4 まとめ . . . 215
4.3.5 演習問題 . . . 215
4.4 数値安定性と初期化 . . . 217
4.4.1 勾配消失と勾配爆発 . . . 217
4.4.2 パラメータの初期化 . . . 219
4.4.3 隠れユニット間の対称性 . . . 222
4.4.4 まとめ . . . 223
4.4.5 演習問題 . . . 223
4.5 深層学習における汎化 . . . 225
4.5.1 記憶と容量 . . . 225
4.5.2 二重降下 . . . 228
4.5.3 勾配降下法の暗黙的バイアス . . . 230
4.5.4 早期停止 . . . 232
4.5.5 まとめ . . . 233
4.5.6 演習問題 . . . 234
4.6 ドロップアウト . . . 235
4.6.1 隠れ層におけるノイズ . . . 236
4.6.2 実装 . . . 236
4.6.3 ラベルノイズ下におけるドロップアウトの効果 . . . 239
4.6.4 まとめ . . . 240
4.6.5 演習問題 . . . 241
4.7 Kaggle での住宅価格予測 . . . 242
4.7.1 データセットと log-RMSE 指標 . . . 242
4.7.2 データリークのない前処理 . . . 243
4.7.3 K 分割交差検証によるモデル選択 . . . 244
4.7.4 予測の提出 . . . 247
4.7.5 まとめ . . . 248
4.7.6 演習問題 . . . 248

5 計算 **251**
参考資料と関連文献 . . . 252
5.1 モジュールとモデル構築 . . . 253
5.1.1 モジュールの抽象化 . . . 253
5.1.2 モジュールコンテナ . . . 256

5.1.3 順伝播の計算 . . . 258
5.1.4 遅延初期化 . . . 260
5.1.5 設定からのモデル構築 . . . 261
5.1.6 まとめ . . . 263
5.1.7 演習問題 . . . 263
5.2 パラメータ、状態、メモリ . . . 264
5.2.1 状態木 . . . 265
5.2.2 訓練時のメモリ . . . 267
5.2.3 パラメータの共有と凍結 . . . 269
5.2.4 まとめ . . . 273
5.2.5 演習問題 . . . 273
5.3 初期化 . . . 274
5.3.1 イニシャライザの適用 . . . 274
5.3.2 カスタムイニシャライザと直接編集 . . . 276
5.3.3 残差ネットワークの初期化 . . . 276
5.3.4 まとめ . . . 280
5.3.5 演習問題 . . . 281
5.4 カスタム層とカスタム関数 . . . 282
5.4.1 パラメータをもたない層 . . . 282
5.4.2 パラメータをもつ層 . . . 283
5.4.3 固定状態をもつ層 . . . 285
5.4.4 カスタム勾配をもつ関数 . . . 287
5.4.5 まとめ . . . 290
5.4.6 演習問題 . . . 291
5.5 数値形式 . . . 291
5.5.1 浮動小数点形式 . . . 292
5.5.2 dtype の昇格とモデルのキャスト . . . 295
5.5.3 混合精度訓練 . . . 296
5.5.4 数値計算の失敗モード . . . 301
5.5.5 まとめ . . . 302
5.5.6 演習問題 . . . 303
5.6 保存、読み込み、事前学習済みの重み . . . 303
5.6.1 モデルの状態 . . . 304
5.6.2 Pickle と safetensors . . . 305
5.6.3 訓練チェックポイント . . . 307
5.6.4 事前学習済みの重み . . . 311
5.6.5 まとめ . . . 312
5.6.6 演習問題 . . . 313
5.7 GPU、デバイス、メモリ . . . 314
5.7.1 デバイスと配置 . . . 315
5.7.2 非同期実行 . . . 317
5.7.3 GPU メモリ . . . 320
5.7.4 デバイス対応の Trainer . . . 325
5.7.5 まとめ . . . 327
5.7.6 演習問題 . . . 327
5.8 再現性と検査 . . . 328
5.8.1 シードと乱数生成器 . . . 329
5.8.2 決定論的な算術演算 . . . 332
5.8.3 フックによるモデルの検査 . . . 333
5.8.4 まとめ . . . 337

5.8.5 演習問題 337

6 畳み込みニューラルネットワーク 339
参考資料と関連文献 340
6.1 全結合層から畳み込みへ 341
6.1.1 同変性と不変性 342
6.1.2 全結合層への制約 343
6.1.3 畳み込みと相互相関 346
6.1.4 チャネル 347
6.1.5 まとめ 348
6.1.6 演習問題 348
6.2 画像に対する畳み込み 349
6.2.1 相互相関 350
6.2.2 畳み込み層 351
6.2.3 行列積としての畳み込み 354
6.2.4 特徴マップと受容野 355
6.2.5 まとめ 356
6.2.6 演習問題 356
6.3 パディング、ストライド、ダイレーション 359
6.3.1 パディング 359
6.3.2 ストライド 362
6.3.3 ダイレーション 364
6.3.4 まとめ 365
6.3.5 演習問題 366
6.4 複数の入力チャネルと複数の出力チャネル 367
6.4.1 複数の入力チャネル 367
6.4.2 複数の出力チャネル 368
6.4.3 1×1 畳み込み 370
6.4.4 まとめ 372
6.4.5 演習問題 372
6.5 プーリング 373
6.5.1 最大プーリングと平均プーリング 373
6.5.2 小さなシフトに対する耐性 374
6.5.3 グローバルプーリング 376
6.5.4 パディングとストライド 376
6.5.5 複数チャネル 377
6.5.6 まとめ 378
6.5.7 演習問題 378
6.6 LeNet 379
6.6.1 LeNet のアーキテクチャ 380
6.6.2 各層の形状 381
6.6.3 パラメータと積和演算 383
6.6.4 訓練 384
6.6.5 まとめ 388
6.6.6 演習問題 388

7 現代の畳み込みニューラルネットワーク 391
7.1 AlexNet と学習された画像表現 392
7.1.1 学習された特徴、データ、ハードウェア 392
7.1.2 AlexNet のアーキテクチャ 393

7.1.3 Fashion-MNIST での訓練 . . . 397
7.1.4 まとめ . . . 400
7.1.5 演習問題 . . . 400
7.2 ブロック、ボトルネック、ブランチ：VGG、NiN、GoogLeNet . . . 401
7.2.1 VGG：設計単位としてのブロック . . . 402
7.2.2 NiN：1 × 1 畳み込みとグローバル平均プーリング . . . 406
7.2.3 GoogLeNet：マルチブランチブロック . . . 410
7.2.4 ネットワークの比較 . . . 413
7.2.5 まとめ . . . 414
7.2.6 演習問題 . . . 415
7.3 正規化層 . . . 415
7.3.1 バッチ正規化 . . . 416
7.3.2 ゼロからの実装 . . . 418
7.3.3 バッチ正規化を用いた LeNet . . . 420
7.3.4 バッチ統計量を使わない正規化 . . . 423
7.3.5 まとめ . . . 425
7.3.6 演習問題 . . . 425
7.4 残差ネットワーク . . . 426
7.4.1 関数クラスと深さ . . . 427
7.4.2 残差ブロック . . . 428
7.4.3 プレーンネットワークと残差ネットワークの比較 . . . 430
7.4.4 ResNet-18 . . . 433
7.4.5 ResNeXt と DenseNet . . . 436
7.4.6 まとめ . . . 440
7.4.7 演習問題 . . . 441
7.5 効率的な畳み込みネットワーク . . . 442
7.5.1 デプスワイズ分離可能ネットワーク . . . 443
7.5.2 スケーリングと探索 . . . 453
7.5.3 構造的再パラメータ化 . . . 455
7.5.4 まとめ . . . 458
7.5.5 演習問題 . . . 458
7.6 畳み込みネットワークの現代的な訓練レシピ . . . 459
7.6.1 2015 年から 2022 年にかけての訓練レシピ . . . 460
7.6.2 各構成要素の実装 . . . 461
7.6.3 1 つのネットワークに対する 2 つのレシピ . . . 466
7.6.4 まとめ . . . 469
7.6.5 演習問題 . . . 469
7.7 ConvNeXt：ResNet の段階的な現代化 . . . 470
7.7.1 現代化のロードマップ . . . 471
7.7.2 実装 . . . 473
7.7.3 Fashion-MNIST での小規模ロードマップ . . . 478
7.7.4 限界と未解決問題 . . . 480
7.7.5 まとめ . . . 481
7.7.6 演習問題 . . . 481
7.8 畳み込みネットワークの設計空間 . . . 482
7.8.1 AnyNet 設計空間 . . . 482
7.8.2 設計空間の比較 . . . 486
7.8.3 RegNet . . . 490
7.8.4 まとめ . . . 494
7.8.5 演習問題 . . . 494

8 系列モデル **497**
8.1 系列の扱い . . . 498
8.1.1 系列データ . . . 498
8.1.2 自己回帰モデル . . . 499
8.1.3 マルコフモデルと定常性 . . . 500
8.1.4 ノイズを含む正弦波の予測 . . . 501
8.1.5 まとめ . . . 508
8.1.6 演習問題 . . . 508
8.2 テキストからトークンへ . . . 509
8.2.1 データセットの読み込み . . . 509
8.2.2 文字、単語、バイト . . . 510
8.2.3 バイトペアエンコーディング . . . 514
8.2.4 特殊トークン . . . 522
8.2.5 実世界のトークナイザ . . . 522
8.2.6 まとめ . . . 526
8.2.7 演習問題 . . . 526
8.3 言語モデル . . . 527
8.3.1 n グラム言語モデル . . . 528
8.3.2 パープレキシティと平滑化 . . . 534
8.3.3 系列の分割 . . . 540
8.3.4 まとめ . . . 542
8.3.5 演習問題 . . . 542
8.4 リカレントニューラルネットワーク . . . 544
8.4.1 隠れ状態を持つリカレントニューラルネットワーク . . . 544
8.4.2 RNN 言語モデルのゼロからの実装 . . . 548
8.4.3 勾配クリッピング . . . 550
8.4.4 訓練と評価 . . . 551
8.4.5 テキスト生成 . . . 557
8.4.6 簡潔な実装 . . . 558
8.4.7 まとめ . . . 560
8.4.8 演習問題 . . . 560
8.5 通時的誤差逆伝播法 . . . 563
8.5.1 勾配消失と勾配爆発 . . . 563
8.5.2 打ち切り型通時的誤差逆伝播法 . . . 568
8.5.3 まとめ . . . 568
8.5.4 演習問題 . . . 569
8.6 デコーディングと生成 . . . 570
8.6.1 デコーディング問題 . . . 570
8.6.2 貪欲デコーディング . . . 573
8.6.3 ビームサーチ . . . 574
8.6.4 サンプリング . . . 577
8.6.5 評価とコスト . . . 582
8.6.6 まとめ . . . 584
8.6.7 演習問題 . . . 584

参考文献 **587**

基礎

イントロダクションでは、機械学習が人手で書かれたルールを、データ例から直接学習するパラメータ化されたプログラムに置き換える仕組みを見た。ここで、衣類を写した小さなグレースケール写真を数万枚手渡され、それぞれの画像に写っているのが T シャツ、スニーカー、コート、ハンドバッグのどれかを識別するシステムを構築するよう求められたとする。生のピクセル強度を衣類のカテゴリへ確実に対応づける明示的な `if-else` ルールをどう書けばよいか、わかるプログラマはいない。しかし、本パートの最初の 3 章を終える頃には、まさにそれを実現する実用的な分類器を実際に構築し、訓練し終えているはずである。さらに本パートを終える頃には、複雑な視覚シーンを認識する深い畳み込みネットワークや、英語の散文をゼロから生成することを学習するリカレントネットワークにまで進んでいるだろう。

「基礎」パート全体を通じたアプローチは、シンプルな原則に基づいている。深層学習を真に理解するには、ブラックボックスのライブラリ呼び出しに頼る前に、その中核となる仕組みを自らの手で構築すべきだということである。これら 8 つの章を通して、基礎となるほぼすべてのモデル、層、損失関数、オプティマイザを 2 回ずつ実装する。最初は**ゼロから**、基本的なテンソル演算と自動微分（automatic differentiation）のみに頼り、すべての数学的演算がコード上で見える形で実装する。次に、現代の深層学習フレームワーク（PyTorch、JAX、TensorFlow、MXNet）の高水準な抽象化を用いて**簡潔に**実装する。各コンポーネントが土台からどのように動作するかを確認した後は、以降の章ではその簡潔なフレームワーク実装を構成要素として用いる。

本パートの 8 つの章は、次の 3 つの段階で展開される。

- **ツールキットと基本レシピ（第 1 章〜第 3 章）：**まずは、完全に解析できるほどシンプルなモデルから始める。第 1 章では、多次元配列（**テンソル**）へのデータの格納と操作、表形式データの前処理、コードによる線形代数と微分積分の適用、**autograd** による導関数の自動計算、そして確率を用いた不確実性の扱いといった、深層学習に不可欠な実用的ツールを身につける。続く第 2 章と第 3 章では、線形モデルを題材に機械学習の普遍的な訓練ループを導入する。第 2 章では、線形回帰、二乗誤差損失、ミニバッチ確率的勾配降下法を用いて連続値のターゲットにモデルを適合させ、統計学の中心的な課題である**汎化**（generalization）に向き合う。すなわち、訓練データに過度に適合させると過学習を招く理由と、重み減衰などの正則化手法がいかにモデルを健全に保つかを考察する。第 3 章では連続値から離散的なカテゴリへと対象を移し、ソフトマックス回帰と交差エントロピー損失を構築して Fashion-MNIST ベンチマーク上の衣類写真を分類する。あわせて、実世界のデータ分布が訓練セットからずれたときに、デプロイされた分類器がどのように振る舞うかも検証する。
- **深層化とソフトウェアエンジンの習得（第 4 章〜第 5 章）：**線形モデルに引けるのは、クラス間の平坦な決定境界だけである。第 4 章では、非線形な活性化関数を備えたユニットの隠れ層を積み重ねて**多層パーセプトロン**（multilayer perceptron、MLP）を構築し、この限界を打破する。ここでは、誤差逆伝播法（backpropagation）によって深いネットワーク内を信号と勾配が順方向・逆方向に流れる仕組みをたどり、不用意に初期化すると深いネットワークが勾配消失や勾配爆発に陥る理由を探る。さらに、容量の大きいネットワークに

対する強力な正則化手法として**ドロップアウト**を導入する。続く第 5 章では、現代の深層学習フレームワークのソフトウェアアーキテクチャを解き明かす。再利用可能なモジュール式のブロックから複雑なネットワークを組み立てる方法、パラメータの共有・検査・初期化、カスタム層と順伝播の定義、訓練済みの重みの保存と読み込み、そして GPU 上への計算の移行を扱う。

- **ドメイン構造のアーキテクチャへの組み込み（第 6 章〜第 8 章）：**全結合ネットワークはすべての入力特徴量を対称に扱うため、入力が 2 次元の画像グリッド上に並んだピクセルなのか、文の中に順序よく並んだ単語なのかを区別しない。最後の 3 章では、データが本来持つ対称性をネットワークのアーキテクチャに直接組み込むことで、統計的効率と計算効率が劇的に向上することを示す。画像に対しては、第 6 章で平行移動不変性と空間的局所性の原理から**畳み込みニューラルネットワーク**（convolutional neural network、CNN）を展開し、パディング、ストライド、複数の入力・出力チャネル、プーリングを導入して古典的な LeNet アーキテクチャを構築する。続く第 7 章では、コンピュータビジョンを現代へと押し上げたアーキテクチャ上のマイルストーンとして、AlexNet、繰り返し可能なブロック設計（VGG）、1×1 畳み込み（NiN）、複数分岐の並列ブロック（GoogLeNet）、**バッチ正規化**（batch normalization）、ResNet の**残差接続**（residual connection）、DenseNet をたどり、原理に基づいたアーキテクチャ設計空間（AnyNet と RegNet）へと至る。最後に第 8 章ではテキストや時系列などの系列データに目を向け、自己回帰言語モデリング、バイトレベル BPE トークン化、n-gram モデルとパープレキシティ、**リカレントニューラルネットワーク**（RNN）、通時的誤差逆伝播法（backpropagation through time）、デコーディングを構築し、H. G. Wells の『The Time Machine』風の新しい文章を生成する BPE レベルの言語モデルを訓練してこれらすべてを検証する。

最初の 5 章（第 1 章〜第 5 章）は一貫した背骨をなしており、順番どおりに読み進めるのが最もよい。これらを終えた後は、コンピュータビジョンの章（第 6 章と第 7 章）と系列モデリングの章（第 8 章）は互いに依存しないため、どちらの順序で学んでも構わない。これら全体を通じて、「発展」パートが土台とする概念的・工学的な習慣が築かれる。

Chapter 1

準備

深層学習の世界に飛び込む準備として、いくつか欠かせない必須スキルを身につけておく必要がある。具体的には、(i) 多次元データをメモリに格納して操作する技法（1.1 節）、(ii) 欠損値やカテゴリ型のテキスト項目を含む生ファイルから表形式データを取り込み、クリーニングするツール（1.2 節）、(iii) 高次元表現を変換し、その大きさを測る線形代数の演算を自在に扱う力（1.3 節）、(iv) 損失関数（loss function）を減少させるために各パラメータをどの方向へ調整すべきかを定める微分法（1.4 節）、(v) 数百万個のパラメータにわたる連鎖律（chain rule）をフレームワークに処理させる自動微分（automatic differentiation）の仕組み（1.5 節）、そして (vi) 不確実性のもとで推論し、サンプリングノイズから真のシグナルを見分けるための主要な言語である確率と統計の実践的な理解（1.6 節）である。

古典的な線形代数、微分積分、確率に深層学習が付け加えるものは**規模**（scale）である。手計算など到底不可能な、数百万から数十億もの変数に対してこれらを適用する。これを支えるのが、以降のすべての章で前提となる 2 つのソフトウェアの柱、すなわち CPU や GPU 上で数値配列全体にコンパイル済み演算を並列適用する**テンソルライブラリ**と、任意の数値計算プログラムを機械的に微分する**自動微分**である。

本章では、これらの前提知識を手を動かしながら短時間で身につける速習コースを提供する。NumPy、線形代数、多変数の微分積分にすでに慣れている場合は既知の部分を流し読みしてもよいが、自動微分を扱う 1.5 節と、標準誤差（standard error）および統計的推定を扱う 1.6 節は、以降の章で絶えず用いられるため一読を勧める。完全な証明と導出を備えたより深い理論的基礎については、付録の**深層学習のための数学**で線形代数（第 33 章）、微分積分（第 34 章）、確率と統計（第 36 章）を詳細に展開している。

1.1 データ操作

機械学習で何かを実現するには、メモリ上でデータを効率よく格納し操作する手段が欠かせない。まずは、深層学習において**テンソル**と呼ばれる n 次元配列から始めよう。

専用の配列コンテナが不可欠である理由を見るため、256 枚のグレースケール写真（解像度 28×28 ピクセル）からなる 1 つの**ミニバッチ**を考えよう。ここには $256 \times 28 \times 28 = 200{,}704$ 個の数値が含まれる。画像ファイルは各ピクセルを $\{0, \dots, 255\}$ の整数として格納しており、これを $[0, 1]$ にスケーリングするため 255 で割る。素の Python では、3 重の入れ子ループによってインタプリタ上で 200,704 回のスカラー除算を個別に実行することになる。一方、テンソルライブラリでは `images / 255` という 1 つの式を書くだけでコンパイル済みの C++ や CUDA のカー

ネルが呼び出され、200,704 個の数値すべてが CPU や GPU 上で並列に除算される。深層学習では一貫して、Python のループで個々のスカラー要素を反復処理するのではなく、テンソル全体に対して一度に演算を適用する。

現代のあらゆる深層学習フレームワークにおいて、テンソルクラスは NumPy の `ndarray` に酷似しているが (Harris et al. 2020)、次の 2 つの重要な機能が加わっている。

1. **ハードウェアアクセラレーション：**NumPy が CPU 上で動作するのに対し、フレームワークのテンソルは GPU や専用の AI アクセラレータ上で実行される。
2. **自動微分**（automatic differentiation）：フレームワークはテンソル上の計算を追跡し、誤差逆伝播法（backpropagation）によって勾配（gradient）を自動的に計算する（1.5 節）。

PyTorch では、テンソルクラスは `torch.Tensor` である。PyTorch（`torch`）をインポートする。

```
import torch
```

1.1.1 テンソルの生成

1.1.1.1 形状

テンソルは、1 つ以上の**軸**（axis）に沿って数値を並べたものである。軸が 1 つのテンソルは**ベクトル**、2 つのテンソルは**行列**、$k > 2$ 個の軸を持つテンソルは k^{th} **階テンソル**である（軸の数をそのテンソルの**階数**（order）といい、コード上では `ndim` として参照できる）。例として挙げたグレースケール画像のミニバッチは 4^{th} 階テンソルであり、その形状（shape）は $(256, 1, 28, 28)$ となる。軸 0 はバッチ内の画像、軸 1 はカラーチャネル（グレースケールでは 1、RGB では 3）、軸 2 は縦方向のピクセル行、軸 3 は横方向のピクセル列をそれぞれ指定する。

`arange(n)` を呼び出すと、0（含む）から n（含まない）まで等間隔に並んだ値の 1 次元ベクトルが生成される。新しく生成されたテンソルはデフォルトで CPU 計算用のメインメモリ上に配置される。ここでは 32 ビット浮動小数点（floating point）の値（`float32`）を指定する。

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

テンソル内の各値は**要素**（element）と呼ばれる。`x` に含まれる要素の総数（**サイズ**）は次のように取得できる。

PyTorch では、要素の総数は `numel()` によって返される。

```
x.numel()
```

```
12
```

次に、`x.shape` によってテンソルの**形状**タプルを確認する。これは各軸に沿った長さを並べたものである（ここでは 12 要素のベクトルに対する 1 要素タプル `(12,)` となる）。

```
x.shape
```

```
torch.Size([12])
```

Figure 1.1 に、階数が 0 から 3 までのテンソルを示す。単一の数値（**スカラー**）は、空の形状 `()` を持つ 0^{th} 階テンソルである。どのようなテンソルでも、そのサイズは形状タプルに含まれる長さの積に等しい（先の画像ミニバッチでは $256 \times 1 \times 28 \times 28 = 200{,}704$）。

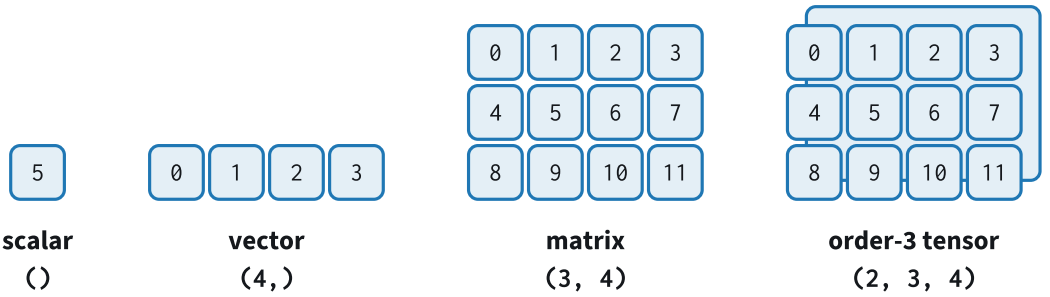


Figure 1.1: 軸が 0、1、2、3 個のテンソル。形状は各軸の長さを並べたものであり、サイズはそれらの長さの積（それぞれ 1、4、12、24 要素）である。

1.1.1.2 データ型

すべてのテンソルは、各要素のバイナリ表現を定める**データ型**（data type、`dtype`）を持つ。深層学習モデルは算術演算の大半を 32 ビット浮動小数点（`float32`、1 要素あたり 4 バイト、10 進で ≈ 7 桁の精度）または 16 ビット形式（`float16` / `bfloat16`、1 要素あたり 2 バイト、5.5 節を参照）で実行する。生の画像ピクセルは通常 8 ビット符号なし整数（`uint8`、各ピクセル 1 バイトで $\{0, \ldots, 255\}$ を表す）として格納されるため、この $200{,}704$ ピクセルのミニバッチは `uint8` では 0.2 MB、`float32` では 0.8 MB を占める。

整数引数に対する `torch.arange` のデフォルトは `int64` であり、キャストには `x.to(torch.int32)` を使う。

1.1.1.3 初期値

関数 `zeros` と `ones` は、それぞれ 0 または 1 で満たされた任意の形状のテンソルを生成する（バイアスやバッファに使われる）。

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

ニューラルネットワークの重み（weight）は通常、確率分布から各要素を独立にサンプリングして初期化（initialization）する（4.4 節）。以下では、3×4 行列を標準ガウス分布（平均 0、標準偏差 1）から抽出する。

```
torch.randn(3, 4)
```

```
tensor([[ 0.2763, -0.8037, -1.2709,  0.2572],
        [ 0.3519, -1.3422, -0.7240,  0.4446],
        [ 1.1666,  0.8201,  0.8971,  0.9919]])
```

コンピュータが生成する乱数は**擬似乱数**（pseudorandom）である。決定論的なアルゴリズムが整数の**シード**から計算するため、同じシードからやり直せばまったく同じ乱数系列が再現される（以下では `a` と `c` は一致するが、`b` は異なる）。

PyTorch では、`torch.manual_seed` によってグローバルな乱数生成器をリセットする。

```
torch.manual_seed(42)
a = torch.randn(3)
b = torch.randn(3)  # The generator has moved on
torch.manual_seed(42)
c = torch.randn(3)  # Same seed: c repeats a
a, b, c
```

```
(tensor([0.3367, 0.1288, 0.2345]),
 tensor([ 0.2303, -1.1229, -0.1863]),
 tensor([0.3367, 0.1288, 0.2345]))
```

小さなテンソルの各要素があらかじめわかっている場合は、入れ子になった Python のリストから直接生成できる（ここでは 3×4 行列を生成しており、外側のリストが 3 つの行を、内側の各リストが 4 つの列を与える）。

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

最後に、本節を通して用いる合成ミニバッチ `images` を、形状 `(256, 1, 28, 28)`・データ型 `uint8` として、$\{0, \dots, 255\}$ のランダムな整数で満たして生成する。

```
# 256 grayscale images of 28 x 28 pixels, each pixel an integer in [0, 255]
images = torch.randint(0, 256, (256, 1, 28, 28), dtype=torch.uint8)
images.shape, images.dtype
```

```
(torch.Size([256, 1, 28, 28]), torch.uint8)
```

1.1.2 形状変換とインデックス参照

1.1.2.1 形状変換

`reshape` を呼び出すと、要素数とメモリ上の行優先順（row-major order）を保ったまま、テンソルの軸を再構成する（Figure 1.2）。ここでは、12 要素のベクトル `x` を 3×4 行列 `X` に形状変換する。

```
X = x.reshape(3, 4)
X
```

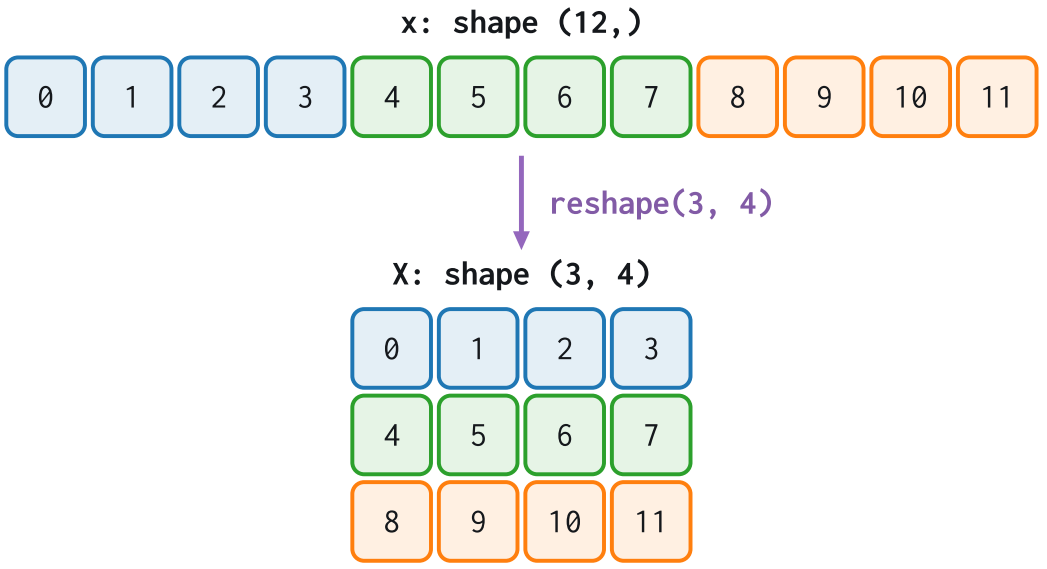


Figure 1.2: 12 要素のベクトルから 3×4 行列への形状変換。要素の順序は保たれ、行列を 1 行ずつ埋めていく。

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

全体のサイズは固定されているため、1 つの次元に `-1` を渡すと、その長さがフレームワークによって自動的に決まる。たとえば、$(256, 1, 28, 28)$ の画像ミニバッチを、1 行あたり 1 枚の画像を持つ $(256, 784)$ 行列へと平坦化（flatten）できる。

```
images.reshape(256, -1).shape
```

```
torch.Size([256, 784])
```

`None` を使ってインデックス参照すると、その位置に長さ 1 の新しい単一軸（singleton axis）が挿入され、一方で `squeeze` は長さ 1 の軸を取り除く。

PyTorch では、`image.unsqueeze(0)` でも位置 `0` に単一軸を挿入できる。

```
image = images[0]
image[None].shape, image.squeeze(0).shape
```

```
(torch.Size([1, 1, 28, 28]), torch.Size([28, 28]))
```

形状変換は要素の順序を保ったままグループ分けを変えるだけなので、2 つの軸を入れ替えることは決してできない。ライブラリによっては、チャネル軸を末尾に置いた $(256, 28, 28, 1)$ の形式で画像ミニバッチを受け取るものがある。ここまでの $(256, 1, 28, 28)$ のミニバッチからこの形式を得るには、軸 1 を末尾へ移す必要がある。これが**転置**（transpose）であり、元の軸を新しい順序（ここでは $(0, 2, 3, 1)$）で指定する。行列における軸の並べ替えは行と列の入れ替えの 1 通りしかなく、`X.T` と書く。これを同じ形状への形状変換と比較してみよう。

PyTorch では、すべての軸を並べ替えるメソッドは `permute` と呼ばれ、`transpose(i, j)` は `i` と `j` の 2 つの軸だけを入れ替える。

```
images.permute(0, 2, 3, 1).shape, X.reshape(4, 3), X.T
```

```
(torch.Size([256, 28, 28, 1]),
 tensor([[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.],
         [ 9., 10., 11.]]),
 tensor([[ 0.,  4.,  8.],
         [ 1.,  5.,  9.],
         [ 2.,  6., 10.],
         [ 3.,  7., 11.]]))
```

どちらの行列も 4×3 であるが、形状変換の方は依然として行方向に $0, 1, 2, \ldots$ と並んでいるのに対し、転置の方は `X` の行が列へと変わっている。形状は一致するため、転置すべき箇所に誤って形状変換を書いてもエラーは発生しない。正しい数値を誤った配置のまま次の層へ渡してしまうだけである。

1.1.2.2 インデックス参照とスライス

要素や部分テンソルには、軸ごとにカンマで区切った 0 始まりの整数インデックス参照（負のインデックスは末尾から逆向きに数える）および半開区間のスライス `start:stop`（`start` を含み、`stop` を含まない）でアクセスする。

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

形状の違いに注目しよう（Figure 1.3）。整数インデックス（`X[-1]`）はその軸をつぶして形状 `(4,)` のベクトルを返すのに対し、スライス（`X[1:3]`）は軸を保ったまま形状 `(2, 4)` の行列を返す。

形状 $(256, 1, 28, 28)$ の画像ミニバッチに対して、`images[0]` は最初の画像 $(1, 28, 28)$ を取り出し、`images[:32]` は最初の 32 枚の画像 $(32, 1, 28, 28)$ をスライスし、`images[:, 0, 14, 14]` は全 256 枚の画像にわたって中央のピクセル $(256,)$ を取り出す。

```
images[0].shape, images[:32].shape, images[:, 0, 14, 14].shape
```

```
(torch.Size([1, 28, 28]), torch.Size([32, 1, 28, 28]), torch.Size([256]))
```

整数インデックステンソル（integer index tensor）を使ってテンソルをインデックス参照することもできる（**ギャザー**、gather）。2 つの 1 次元整数テンソル `rows = (0, 1, 2)` と `cols = (3, 0, 2)` を渡すと、要素 `(0, 3)`、`(1, 0)`、`(2, 2)` が選択される。

```
rows, cols = torch.arange(3), torch.tensor([3, 0, 2])
X[rows, cols]
```

```
tensor([ 3.,  4., 10.])
```

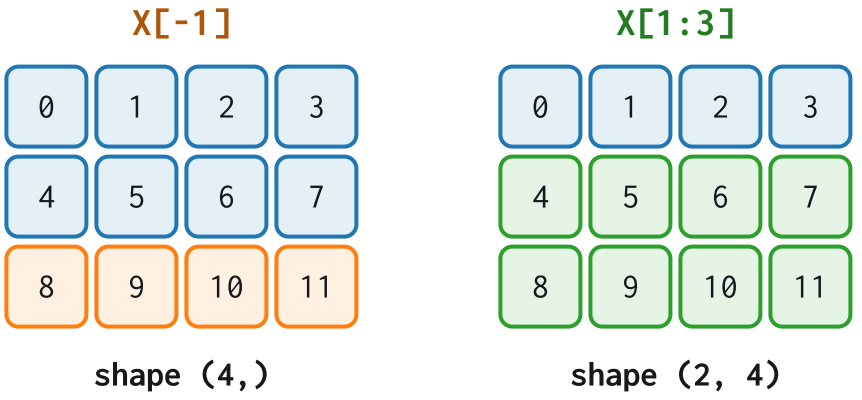


Figure 1.3: 3×4 行列のインデックス参照。整数インデックス `X[-1]` は最後の行を選択して軸 0 を取り除き、形状 $(4,)$ を残す。スライス `X[1:3]` は行 1 と行 2 を選択して軸 0 を保持し、形状 $(2, 4)$ を残す。

同様に、要素ごとの比較からは `True` と `False` の値からなる**ブールマスク**が得られる（`&`、`|`、`~` で組み合わせ可能）。`X[mask]` でインデックス参照すると、`mask` が `True` であるすべての要素が平坦化された 1 次元ベクトルとして取り出される。

```
mask = (X > 2) & (X < 8)
X[mask]
```

```
tensor([3., 4., 5., 6., 7.])
```

これに対し、3×4 という `X` の形状を保ちたい場合は、`where(mask, X, 0)` を使うことで、`X` のうち `mask` が `True` の位置の要素を選択し、残りのすべての位置を `0` で埋めることができる。

```
torch.where(mask, X, 0)
```

```
tensor([[0., 0., 0., 3.],
        [4., 5., 6., 7.],
        [0., 0., 0., 0.]])
```

Figure 1.4 に、これら 3 つの選択方法を並べて示す。違いは返される形状にあり、インデックステンソルは各行から 1 つの要素を返し、マスクはデータに応じた長さのベクトルを返し、`where` は `X` と同じ形状のテンソルを返す。

1.1.2.3 要素の書き込み

テンソルをインプレースで上書きできるかどうかは、フレームワークの設計に依存する。

PyTorch のテンソルは**ミュータブル**であり、`X[i, j] = val` はメモリをインプレースで上書きする。さらに、`reshape` は元のバッファを共有するゼロコピーの**ビュー**を返しているため、`X` を変更すると `x` も変更される。17 を `X[1, 2]`（1 次元インデックスでは 6）に代入し、`X` と `x` の両方を確認しよう。

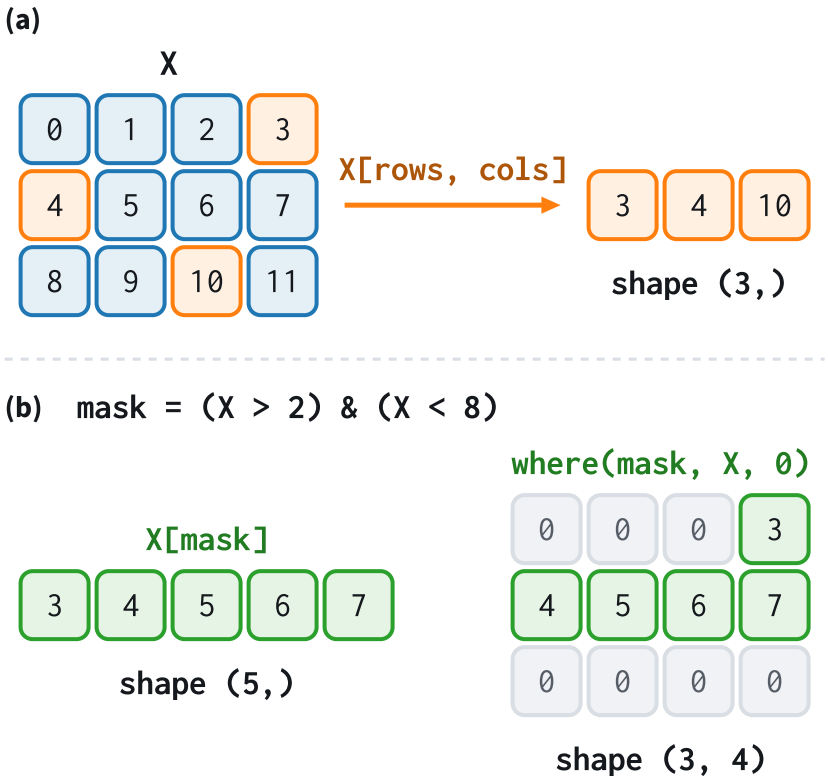


Figure 1.4: 3×4 行列 `X` からの要素の選択。(a) インデックステンソル `rows = (0, 1, 2)` と `cols = (3, 0, 2)` は各行から 1 つの要素を選択する。(b) マスク `(X > 2) & (X < 8)` は 2 より大きく 8 より小さい 5 つの要素をベクトルとして取り出すのに対し、`where` は `X` の形状を保ち、それ以外のすべての位置に 0 を書き込む。

```
X[1, 2] = 17
X, x
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5., 17.,  7.],
         [ 8.,  9., 10., 11.]]),
 tensor([ 0.,  1.,  2.,  3.,  4.,  5., 17.,  7.,  8.,  9., 10., 11.]))
```

PyTorch では、`x[6]` が `17.` に変化している。`reshape` が `x` とストレージを共有するビューを返したからである。

代入の左辺でスライスを使うと、ブロック全体に一度に値を代入できる。

```
X[:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

1.1.3 演算

1.1.3.1 要素ごとの演算

標準的な二項算術演算子（`+`、`-`、`*`、`/`、`**`）はスカラーの算術演算を拡張し、同じ形状を持つ 2 つのテンソル `x` と `y` の対応する要素の組ごとに独立して作用する。

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2.0, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

`exp` のような単項数学関数も同様に要素ごとに適用され、e^{x_i} を `x` の各要素に対して計算する。

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

`uint8` 型のミニバッチ `images` を `255` で割ると、全 200,704 ピクセルが一度にスケーリングされ、結果の型は `float32` に昇格（type promotion）する。

```
scaled = images / 255
print(scaled.dtype, images[0, 0, 0, :4], scaled[0, 0, 0, :4], sep='\n')
```

```
torch.float32
tensor([102, 121, 210, 214], dtype=torch.uint8)
tensor([0.4000, 0.4745, 0.8235, 0.8392])
```

`x == y` のような関係演算子も要素ごとに作用し、`True` を `x` と `y` が一致する位置に持つブールテンソルを返す（ここでは両方とも `2.0` に等しいインデックス 1 のみ）。

```
x == y
```

```
tensor([False,  True, False, False])
```

`sum()` でテンソルの総和をとると、すべての要素が 1 つのスカラーに縮約される（$1+2+4+8 = 15$ が `x` の総和）。`True` は 1、`False` は 0 として数えられるため、ブールマスクの総和 `(x == y).sum()` によって一致する要素の個数を数えられる。

```
x.sum(), (x == y).sum()
```

```
(tensor(15.), tensor(1))
```

1.1.3.2 ブロードキャスト

4 要素のバイアスベクトル `v` を 3×4 行列 `X` のすべての行に加える場合のように、**異なる**形状のテンソル同士を組み合わせることは多い。

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
v = torch.tensor([100.0, 200, 300, 400])
X + v
```

```
tensor([[100., 201., 302., 403.],
        [104., 205., 306., 407.],
        [108., 209., 310., 411.]])
```

テンソルライブラリは物理的なコピーを実体化する代わりに、2 段階の**ブロードキャスト**規則を適用する。

1. **形状を右揃えにする：**両方の形状タプルが同じ長さになるまで、階数の低いテンソルの形状タプルの左側に 1 を補う。揃えられた 2 つの次元は、長さが等しいか、または片方が 1 であれば互換性がある。
2. **単一軸を引き伸ばす：**メモリをコピーすることなく、長さ 1 の軸をもう一方のテンソルの長さ $k > 1$ に合わせて仮想的に複製し、その上で要素ごとの演算を適用する。

両方のテンソルが同時に引き伸ばされる様子を見るため（Figure 1.5）、3×1 の列ベクトル `a` と 1×2 の行ベクトル `b` を作ってみよう。

```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

`a + b` を計算すると、`a` が 2 つの列に、`b` が 3 つの行に仮想的に複製され、3×2 行列としてすべての対の和 $a_{i0} + b_{0j}$ が生成される。

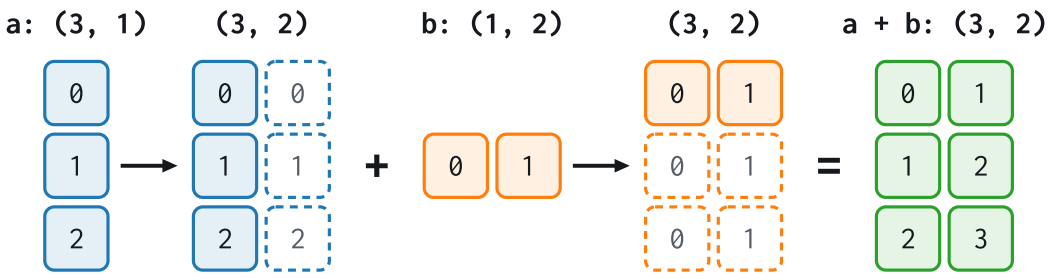


Figure 1.5: ブロードキャストは長さ 1 の軸を引き伸ばす。a（形状 $(3,1)$ の列）と b（形状 $(1,2)$ の行）は、要素ごとの加算の前にどちらも形状 $(3,2)$ へと拡張される（破線は仮想的なコピー）。

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

揃えられた次元の組のうち、長さが異なりかつどちらも 1 でないものがある場合（たとえば `(3, 2)` と `(2, 3)`）、ブロードキャストはエラーを送出する。

```
try:
    torch.ones((3, 2)) + torch.ones((2, 3))
except Exception as e:
    print(e)
```

```
The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 1
```

したがって、$(28,28)$ の平均画像を `images`（形状 $(256,1,28,28)$）から引くと $(1,1,28,28)$ として問題なくブロードキャストされる一方、画像ごとの値を持つ形状 $(256,)$ のベクトルを平坦化された画像 $(256,784)$ に足す演算は、列 $(256,1)$ に形状変換しない限り失敗する。

エラーメッセージが出るのはむしろ幸運な場合である。形状 $(3,)$ のラベルを形状 $(3,1)$ の予測から引く演算は同じ規則のもとで正当とみなされ、本来欲しかった 3 個の誤差ではなく、すべての対の差からなる 3×3 行列を返してしまう（Figure 1.6）。その行列全体で平均をとった損失もやはり 1 つの数値になるため、モデルが学習に失敗するまで異常に気づきにくい。

1.1.3.3 連結

`concatenate`（または `cat` / `concat`）は、**既存**の軸に沿ってテンソルを端から端へ連結（concatenation）する（それ以外のすべての軸は長さが一致していなければならない）。2 つの 3×4 行列 `X` と `Y` を `axis=0` に沿って連結すると行が積み重なって形状 `(6, 4)` となり、`axis=1` に沿って連結すると列がつながって形状 `(3, 8)` となる。

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

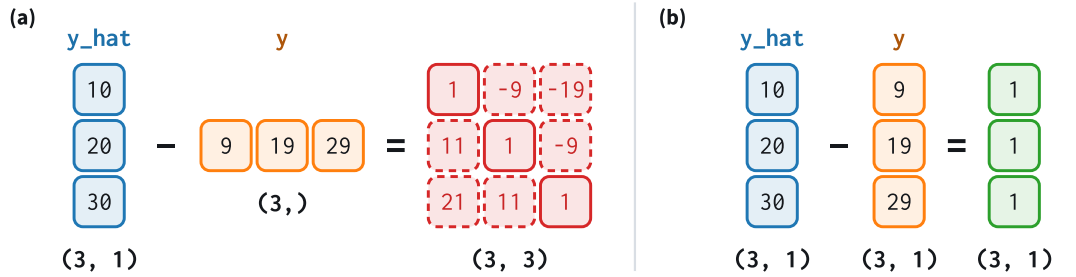


Figure 1.6: エラーメッセージの出ないブロードキャスト。(a) 予測 y_hat（形状 $(3, 1)$）からラベル y（形状 $(3,)$）を引くと、すべての対の差 9 個が得られ、対角成分（実線）だけが本来欲しかった 3 個の誤差を保持する。(b) y を $(3, 1)$ に形状変換すれば、同じ減算からその 3 個の誤差だけが返される。

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

これに対し、stack は同じ形状のテンソルを**新しい**軸に沿って結合し、X と Y を束ねて 3^{rd} 階テンソルを作る。形状はデフォルトで (2, 3, 4)（axis=0）となり、(3, 4, 2) になるのは axis=2 に沿って重ねた場合である。

```
torch.stack((X, Y)).shape, torch.stack((X, Y), dim=2).shape
```

```
(torch.Size([2, 3, 4]), torch.Size([3, 4, 2]))
```

1.1.4 メモリと変換

1.1.4.1 メモリ確保とインプレース更新

Y = Y + X のような標準的な式は新しいバッファを確保し、Y をその新しいアドレスに再束縛する。これは Python の id() で確認できる。

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

数ギガバイトものパラメータを持つモデルでは、不要なメモリ確保はメモリ帯域幅（memory bandwidth）を浪費し、他の参照が古いバッファを指したままになる原因にもなる（ただし順伝播（forward pass）の内部では、自動微分でまだ必要とされる活性化値（activation）を上書きしてはならない。1.5 節を参照）。

MXNet と PyTorch では、スライス代入 `Z[:] = X + Y` によって事前に確保された既存のバッファ `Z` に結果が書き込まれるため、`id(Z)` は変化しない。

```
Z = torch.zeros_like(Y)
before = id(Z)
Z[:] = X + Y
id(Z) == before
```

```
True
```

`X` の古い値がもはや不要なとき、累積代入 `X += Y` は和を `X` 自体に書き込むよう要求する。実際にそうできたかどうかは `id(X)` で確認できる。

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

`id(X) == before` は `True` であり、`X` はインプレースで更新された。

1.1.4.2 他の Python オブジェクトへの変換

描画ライブラリやデータライブラリとの境界では、フレームワークのテンソルと NumPy 配列の相互変換がよく使われるが、変換時にメモリを共有するかコピーするかはフレームワークによって異なる。

CPU 上では、`X.numpy()` と `torch.from_numpy(A)` は**まったく同じ基底のメモリバッファを共有する**ため、一方をインプレースで変更するともう一方も即座に変更される。以下の書き込みの後、`X[0, 0]` と `B[0, 0]` はともに `100.0` に変わる。

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

`A`、`B`、`X` がメモリを共有しているかどうかは、NumPy 配列上で `A[0, 0] = 100` と上書きしてから `X[0, 0]` と `B[0, 0]` を調べることで確認できる。

```
A[0, 0] = 100  # Write to the NumPy array
X[0, 0], B[0, 0]
```

```
(tensor(100.), tensor(100.))
```

単一要素の（0^{th} 階）テンソルを Python の `float` や `int` として取り出すには（たとえばスカラーの損失をログに記録するとき）、`.item()`、`float()`、または `int()` を呼び出す。ただし、これによって実行待ちの GPU カーネルと CPU との同期が強制される点に注意しよう。

```
a = torch.tensor(3.5)
a, a.item(), float(a), int(a)
```

```
(tensor(3.5000), 3.5, 3.5, 3)
```

ヒント：フレームワーク API の調べ方。見慣れないモジュールや関数に出会ったら、`dir(obj)`（または Tab 補完）で利用可能な属性を一覧し、Jupyter 上で `help(fn)` や `fn?` を使ってシグネチャや docstring を確認し（Python のソースコードを見るには `fn??`）、手計算で期待される出力を予測できるごく小さなテスト入力で理解を確かめよう。

1.1.5 まとめ

テンソルは、入力、パラメータ、活性化値、勾配を、`shape` と `dtype` によって特徴づけられる n 次元配列として格納する。深層学習のコードはテンソル全体に対する計算を一度に記述し、ミニバッチの全 $200{,}704$ ピクセルのスケーリングもコンパイル済みの 1 回の演算（`images / 255`）で処理する。軸の再構成には `reshape`、部分集合の抽出や変更には整数インデックス・スライス・ブールマスク（`where`）、テンソルの結合には `concatenate` と `stack` を用い、形状の異なるテンソル同士は**ブロードキャスト**（形状を右揃えにして単一軸を引き伸ばす仕組み）によって組み合わせる。最後に、テンソルがミュータブルである場合はインプレース更新（`Z[:] = ...`、`X += Y`）によってメモリを再利用し、そうでない場合はミュータブルなコンテナやコンパイラがその役割を担う。

1.1.6 演習問題

1. [code] **形状変換における軸の自動決定**。n 枚のグレースケール画像（解像度 28×28）からなるミニバッチが形状 $(n, 1, 28, 28)$ のテンソルとして与えられ、各画像を 784 個のピクセル値からなる行ベクトルへ平坦化しなければならないとする。以下の各予測をコードで検証せよ。

    1. ミニバッチを `(-1, 784)` に形状変換した場合と `(256, -1)` に形状変換した場合の出力形状を、まず $n = 256$ 枚の通常のミニバッチについて、次にエポック最後の $n = 100$ 枚の小さなミニバッチについて予測せよ。2 つの目標形状のうち、**任意**のバッチサイズ n に対して確実に動作するのはどちらか。
    2. 画像が誤って 3 チャネルの RGB 形式（形状 $(n, 3, 28, 28)$）で読み込まれたとする。このテンソルを `(-1, 784)` に形状変換するとどうなるか。`reshape(-1, 784)` はチャネル数の誤りを検出するか、それともエラーを出さずにバッチ次元をかき混ぜてしまうか。

2. [code] **ブロードキャストの形状と暗黙のバグ**。ブロードキャストは 2 つの形状タプルを右から揃える。

    1. すべての軸で長さが異なるにもかかわらずブロードキャスト可能な 2 つの 3^{rd} 階テンソル `a` と `b` を構成せよ。`a + b` の形状を手計算で予測し、コードを実行して予測を検証せよ。
    2. ある回帰モデルが、5 個のデータ例に対して予測の列 `y_hat`（形状 $(5, 1)$）を出力し、正解ターゲット `y` が形状 $(5,)$ の 1 次元ベクトルとして格納されているとする。残差 `y_hat - y` の形状を予測せよ。次に、`y` に `[0.0, 1.0, 2.0, 3.0, 4.0]` を格納し、すべての予測が真のラベルと完全に一致するよう `y_hat = y.reshape(5, 1)` と設定せよ。`y_hat - y` の各要素の二乗の平均を計算し、予測が完璧であるにもかかわらずこの「平均二乗誤差」が 0 にならない理由を説明せよ。さらに、各予測が自身のラベルとのみ比較されるように `y`（または `y_hat`）を形状変換する方法を示せ。

3. [code] **データ型の算術的な限界**。`images` の生のピクセルは 8 ビット符号なし整数（`uint8`）として格納される一方、ニューラルネットワークの層は浮動小数点（`float32`）で計算する。

    1. `100` を、`uint8` テンソルの値 `200` を保持する要素に加算したときの数値結果を予測し、コードで検証せよ。このようなラップアラウンド（桁あふれによる回り込み）が起き

る理由を説明し、整数のオーバーフローを起こさずに `images` の最初の画像を輝度 100 だけ明るくする方法（クランプまたは型の昇格による方法）を示せ。

2. IEEE 754 の `float32` 数値は仮数部に 24 ビットの精度を割り当てている。`float32` が $2^{24} = 16{,}777{,}216$ までのすべての連続する整数を正確に表現できる理由を説明し、`float32(2**24) + float32(1)` を評価しても値が $16{,}777{,}216$ のまま変化しないことをコードで確認せよ。訓練ループで処理したデータ例またはトークンの総数を `float32` のスカラーカウンタ（毎回 1 ずつ加算）に累積していくと、何個のデータ例を処理した時点でカウンタが完全に停止するか。また、代わりにどのデータ型を使うべきか。

4. [code] **インプレース演算の合成。**●`(A + B) * (-A / 2)` を、同じ形状を持つ 2 つの `float32` テンソル `A` と `B` に対して**中間テンソルを一切確保せずに**評価せよ。ただし、PyTorch のインプレースメソッド（`add_`、`div_`、`neg_`、`mul_`）または `out=` キーワード引数のみを用いること。`A.data_ptr()` と `B.data_ptr()` が変化しないこと、および最終結果が非インプレースの式 `(A + B) * (-A / 2)` と一致することを確認せよ。さらに、`A[:] = A + B` と書くだけでは一時領域の確保ゼロという要件を**満たさない**理由を説明せよ。

   Nicolas P. Rougier「100 numpy exercises」問題 35 を改変。

5. [code] **Python のループとブロードキャスト。**●200 要素の 1 次元ベクトル `a` と 300 要素の 1 次元ベクトル `b` が与えられたとき、すべての対の絶対差からなる 200×300 行列 D（要素は $d_{ij} = |a_i - b_j|$）を計算したい。

   1. まず 2 重の入れ子になった Python の `for` ループを使って D を計算し、次に `a` を `(200, 1)` に、`b` を `(1, 300)` に形状変換した上で 1 回のブロードキャスト減算によって計算せよ。両方の行列が数値的に一致することを確認し、各手法の実経過時間（ウォールクロック時間）を計測して高速化倍率を報告せよ。
   2. ブロードキャストは入力 `a` と `b` のコピーを回避するが、出力行列 D は依然としてメモリ上に実体化されなければならない。D を `float32` で保持するとき、`a` と `b` がそれぞれ $10{,}000$ 要素を含むならば何メガバイトを占めるか。この結果から、ブロードキャストを用いた外積型の演算におけるピーク時のメモリ使用量についてどのような教訓が得られるか。

1.2 データの前処理

これまでは、望みどおりの形状とデータ型を備えた既製のテンソルとして与えられる合成データ（synthetic data）を扱ってきた。しかし、実世界で深層学習を応用するには、任意のファイル形式に保存された雑多なデータを取り出し、整った数値テンソルへと前処理しなければならない。生のレコードの表をモデルに入力できるテンソルへと変換する作業には、**前処理**（preprocessing）という控えめな言葉から想像される以上に多くの統計的な判断が伴う。

本節を通して用いる例として、10 件の住宅販売記録からなる小さな表を考えよう。この表には、部屋数（`NumRooms`）、屋根材（`RoofType`）、平方フィート単位の床面積（`Area`）、ドル単位の販売価格（`Price`）が記録されている。小さいながらも、この表には生の表形式データをそのままニューラルネットワークに入力することを阻む 3 つの障害が現れている。

1. **欠損値：**10 件の住宅のうち 3 件で部屋数が欠けており、4 件で屋根材が欠けている。
2. **カテゴリ型のテキスト項目：**屋根材は単語（`Slate` または `Tile`）として記録されているが、ニューラルネットワークは数値のみで計算する。
3. **スケールの不一致：**平方フィート単位の床面積は、数値として部屋数の約 500 倍も大きい。

いずれの障害に対しても統計的な選択が求められる。不完全な行を破棄すれば表の大半が失われ、順序のない屋根材を `0`, `1`, `2` と符号化すれば架空の順序関係を作り出すことになり、床面積をスケーリングしないままにすれば距離や勾配が床面積に支配されてしまう。Python における表形

式データの標準ツールは pandas ライブラリである (McKinney 2010)。以下では、Figure 1.7 に示す前処理パイプラインに沿って、生の CSV ファイルから float32 の特徴量テンソルとターゲットテンソルを作成する手順を順に見ていく。

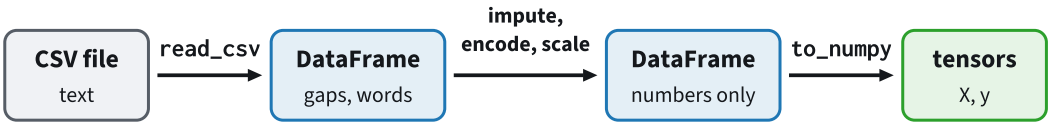


Figure 1.7: ファイルからテンソルへ。read_csv は欠損やテキスト列を含むテキストファイルを DataFrame に読み込む。補完、エンコーディング、スケーリングによってすべての要素を数値に変換し、to_numpy がその結果をテンソル X および y としてフレームワークに渡す。

まず、pandas を慣例的な別名 pd としてインポートし、あわせてファイルパス操作のための Python の os モジュールと、深層学習フレームワーク（整えた表を最後のステップでテンソルに変換する際に必要となる）をインポートする。

```
import os
import pandas as pd
import torch
```

1.2.1 読み込みと確認

1.2.1.1 CSV ファイル

カンマ区切り値（comma-separated values、CSV）ファイルは、1 行目に列見出し、2 行目以降にカンマで区切られたレコードを 1 行ずつ並べることで、表形式データをプレーンテキストとして保存する。ここでは 10 件の住宅データセットを ../data/house_tiny.csv に書き出す。連続するカンマの間に挟まれた空のフィールド（,Tile,2100,290000 など）は、記録されていない値を表す。

```
os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Area,Price
3,Slate,1500,210000
,Tile,2100,290000
2,,850,127500
4,Slate,1940,258000
,,1200,168000
3,Tile,1650,225000
5,Slate,2600,375000
2,,900,142000
,Tile,1750,240000
4,,2050,295000''')
```

pd.read_csv を呼び出すと、ファイルが解析されて pandas の **DataFrame** に読み込まれる。これは名前と型を持つ列からなる 2 次元の表であり、左端には行インデックス（0 から 9）が付いている。

```
data = pd.read_csv(data_file)
data
```

```
   NumRooms RoofType  Area   Price
0       3.0    Slate  1500  210000
1       NaN     Tile  2100  290000
2       2.0      NaN   850  127500
3       4.0    Slate  1940  258000
4       NaN      NaN  1200  168000
5       3.0     Tile  1650  225000
6       5.0    Slate  2600  375000
7       2.0      NaN   900  142000
8       NaN     Tile  1750  240000
9       4.0      NaN  2050  295000
```

pandas は、すべての空フィールド（および NA のような標準的なトークン）を、NaN（**非数**、not a number）へと自動的に置き換えている点に注目しよう。これは**欠損値**（missing value）を表すセンチネル値である。

1.2.1.2 データ型と要約

新しく読み込んだデータセットに変更を加える前に、data.info() を使って行数・列数、非欠損の要素数、および自動判定されたデータ型を確認しよう。

```
data.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   NumRooms  7 non-null      float64
 1   RoofType  6 non-null      str
 2   Area      10 non-null     int64
 3   Price     10 non-null     int64
dtypes: float64(1), int64(2), str(1)
memory usage: 481.0 bytes
```

全 10 行のうち、Area と Price には欠損がなく（int64）、NumRooms には 7 個の観測値があり（float64）、RoofType には 6 個の観測値がある（str、古いバージョンの pandas では object）。NumRooms は整数の部屋数を表すにもかかわらず、float64 に格上げされている点に注目しよう。IEEE 754 規格では NaN は浮動小数点のビットパターンとして定義されているため、標準的な整数配列では欠損要素を表現できないからである。

数値列については、data.describe() によって観測値の個数、平均、標本標準偏差（std）、最小値、四分位数、最大値が表示される。

```
data.describe()
```

```
       NumRooms         Area          Price
count  7.000000    10.000000      10.000000
mean   3.285714  1654.000000  233050.000000
```

```
std    1.112697   556.421104   76128.710455
min    2.000000   850.000000  127500.000000
25%    2.500000  1275.000000  178500.000000
50%    3.000000  1700.000000  232500.000000
75%    4.000000  2022.500000  282000.000000
max    5.000000  2600.000000  375000.000000
```

NumRooms は 2 から 5（平均 3.29）の範囲にあるのに対し、Area は 850 から 2,600 平方フィート（平均 1,654）、Price は $127,500 から $375,000 の範囲にわたっている。

1.2.1.3 入力とターゲットの分離

教師あり学習（supervised learning）では、指定された**ターゲット**（または**ラベル**）列である Price を、残りの**入力**列、すなわち**特徴量**（feature）列（NumRooms、RoofType、Area）から予測する。最初に特徴量をターゲットから分離しておけば、後続の特徴量変換がターゲットに手を加えたりターゲットの情報を漏洩させたりするのを防げる。.iloc による 0 始まりの位置スライシングを使うと、data.iloc[:, :3] で最初の 3 列が選択され、data.iloc[:, 3] で 4 列目が選択される。

```
inputs, targets = data.iloc[:, :3], data.iloc[:, 3]
list(inputs.columns), targets.name
```

```
(['NumRooms', 'RoofType', 'Area'], 'Price')
```

位置によるスライシングは簡潔だが、実運用のパイプラインでは CSV の列が挿入されたり並び替えられたりすると、data.iloc[:, 3] が気づかないうちに誤ったターゲットを取得してしまう。列名で選択すれば（targets = data['Price'] および inputs = data.drop(columns='Price')）、このリスクを回避できる（演習問題 1）。

1.2.2 クリーニングとエンコーディング

1.2.2.1 欠損値

センサの脱落やアンケートの未回答、過去の記録漏れなどにより、欠損値は日常的に発生する。.isna() に .sum() をつなげて呼び出すと、各列の欠損要素数が集計される（True が 1 として数えられる）。

```
inputs.isna().sum()
```

```
NumRooms    3
RoofType    4
Area        0
dtype: int64
```

これには 3 つの標準的な対処法があり、それぞれ異なる統計的トレードオフを持つ（Figure 1.8）。

- **削除**（deletion）は、欠損値を含むすべての行（または列）を破棄する手法である。実装はきわめて簡単だが、データが失われるうえ、値が完全にランダムに欠損しているのでない限り（たとえばワンルームマンションで部屋数の記載漏れが偏って多い場合など）、残った標本に統計的バイアスが生じる。
- **補完**（imputation）は、各 NaN を列の平均や中央値などの推定値で置き換える手法である。すべての行を保持できる反面、近似値で代用するため、その列の見かけ上のばらつきが小さくなる。

- **欠損性インジケータ**（missingness indicator）は、各要素がもともと欠損していた**かどうか**を記録する追加の二値（$0/1$）列を補完と組み合わせる手法である。補完した列ごとに特徴量が 1 つ増える引き換えに、情報を持つ欠損性（たとえば部屋数の未報告が主に高級邸宅で生じる場合など）をモデルが活用できるようになる。

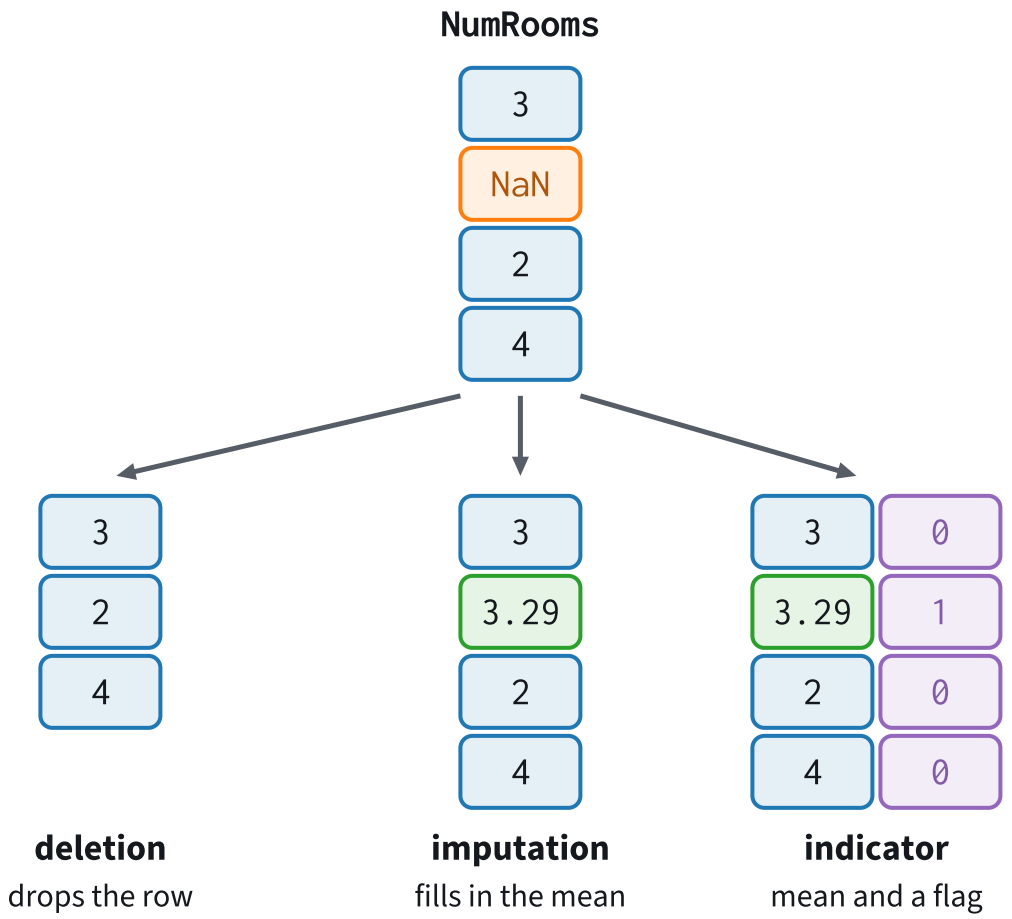


Figure 1.8: NumRooms の最初の 4 行に対する、部屋数の欠損への 3 通りの対処法。削除はその行を除外する。補完は観測された 7 個の値の平均 $23/7 \approx 3.29$ で埋める。欠損性インジケータは同じ平均で埋めたうえで、どこに欠損があったかを記録する列を追加する。

この 10 件の住宅の表では、行の削除（inputs.dropna()）を実行するとデータがほとんど残らない。

```
len(inputs.dropna()), len(inputs)
```

```
(4, 10)
```

不完全な行を除外すると 6 件（全 10 件中、データセットの 60%）の住宅が破棄されてしまうため、代わりに補完を用いる。inputs.fillna(inputs.mean(numeric_only=True)) を呼び出すと、数値列の各 NaN がその列の観測値の平均で置き換えられ（外れ値によって平均が歪む場合は中央値の方が望ましい）、テキスト列である RoofType は次のカテゴリ特徴量のステップへとそのまま残

される。

```
inputs = inputs.fillna(inputs.mean(numeric_only=True))
inputs
```

```
   NumRooms RoofType  Area
0  3.000000    Slate  1500
1  3.285714     Tile  2100
2  2.000000      NaN   850
3  4.000000    Slate  1940
4  3.285714      NaN  1200
5  3.000000     Tile  1650
6  5.000000    Slate  2600
7  2.000000      NaN   900
8  3.285714     Tile  1750
9  4.000000      NaN  2050
```

行 1、4、8 には、観測された 7 件の部屋数の平均である $23/7 \approx 3.285714$ が格納された。現実の住宅に 3.29 部屋というものは存在しないが、補完された平均は中立的な統計的プレースホルダとして機能する。

1.2.2.2 カテゴリ特徴量

欠損要素を独立した 1 つの状態として含めると、RoofType は順序のない 3 つのカテゴリ（Slate、Tile、または NaN）に分類される。Slate $= 0$、Tile $= 1$、missing $= 2$ のように単一の整数コードを割り当てると、特徴量のあらゆる線形結合において Tile が Slate と未記録の屋根の中間として扱われることになり、人工的な距離と順序関係を押し付けてしまう。

ワンホットエンコーディング、すなわち**ダミー変数エンコーディング**（dummy variable encoding）は、k 個のカテゴリを持つ列を k 本の二値（$0/1$）インジケータ列に置き換え、該当するカテゴリの要素を 1.0、残りの $k-1$ 個の要素を 0.0 に設定することでこの落とし穴を回避する。pandas では、pd.get_dummies が数値列をそのまま残しつつすべての文字列列・カテゴリ列をエンコードする。dummy_na=True を指定すると明示的な欠損性インジケータである RoofType_nan が追加され、dtype=float を指定するとブール値ではなく $0.0/1.0$ の浮動小数点数が格納される。

```
inputs = pd.get_dummies(inputs, dummy_na=True, dtype=float)
inputs
```

```
   NumRooms  Area  RoofType_Slate  RoofType_Tile  RoofType_nan
0  3.000000  1500             1.0            0.0           0.0
1  3.285714  2100             0.0            1.0           0.0
2  2.000000   850             0.0            0.0           1.0
3  4.000000  1940             1.0            0.0           0.0
4  3.285714  1200             0.0            0.0           1.0
5  3.000000  1650             0.0            1.0           0.0
6  5.000000  2600             1.0            0.0           0.0
7  2.000000   900             0.0            0.0           1.0
8  3.285714  1750             0.0            1.0           0.0
9  4.000000  2050             0.0            0.0           1.0
```

どの行でも、RoofType_Slate、RoofType_Tile、RoofType_nan の和は 1.0 になる（そのため切片バイアスと併用すると 1 列が冗長になる。同様に、欠損値のない 2 状態の列には $0/1$ インジケータが 1 列あれば足りる）。ワンホットエンコーディングはカテゴリ数が少ない場合に適しているが、

数万ものカテゴリを持つ列（郵便番号、ユーザー ID、語彙の単語など）は、代わりにコンパクトで学習可能な**埋め込み**（embedding）へと写像する（8.4 節）。

1.2.3 スケーリングと変換

1.2.3.1 標準化

これで `inputs` のすべての列が数値になったが、`NumRooms` と `Area` の測定スケールは大きく食い違っている。生の単位のままでは、2 部屋の差は数値上 3 平方フィートよりも小さい。床面積をスケーリングしないままにしておくと、ユークリッド距離（Euclidean distance）や重みのペナルティが床面積に支配されてしまう。さらに 2.3 節および第 9 章で見るように、何桁も異なるスケール差は損失ランドスケープ（loss surface）を細長い谷へと引き伸ばし、勾配降下法（gradient descent）の収束を遅らせる。

標準化（standardization）、すなわち **z スコア正規化**（z-score normalization）は、連続値の特徴量を共通のスケールに揃えるため、各要素 x を次の式へと変換する。

$$\frac{x - \mu}{\sigma}, \tag{1.1}$$

ここで、μ はその列の標本平均、$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$ は標本標準偏差である（Bessel の分母 $n-1$ は 1.6 節で導出する）。標準化されたすべての列は平均が 0、標準偏差が 1 となり、床面積が平方フィートで記録されていようと平方メートルで記録されていようと、観測値は無次元の標準偏差を単位として測られる。ワンホットインジケータはすでに $\{0, 1\}$ の値をとるため、ここでは `['NumRooms', 'Area']` のみを標準化する。

```
numerical = ['NumRooms', 'Area']
mu, sigma = inputs[numerical].mean(), inputs[numerical].std()
inputs[numerical] = (inputs[numerical] - mu) / sigma
inputs
```

```
   NumRooms      Area  RoofType_Slate  RoofType_Tile  RoofType_nan
0 -0.314485 -0.276769             1.0            0.0           0.0
1  0.000000  0.801551             0.0            1.0           0.0
2 -1.415185 -1.444949             0.0            0.0           1.0
3  0.786214  0.513999             1.0            0.0           0.0
4  0.000000 -0.815929             0.0            0.0           1.0
5 -0.314485 -0.007189             0.0            1.0           0.0
6  1.886913  1.700151             1.0            0.0           0.0
7 -1.415185 -1.355089             0.0            0.0           1.0
8  0.000000  0.172531             0.0            1.0           0.0
9  0.786214  0.711691             0.0            0.0           1.0
```

行 6（5 部屋、2,600 平方フィート）は $(1.89, 1.70)$ となり、どちらの次元でも平均より標準偏差の約 1.7 から 1.9 倍だけ上に位置している。一方、平均で補完された行 1、4、8 の 3 つの部屋数は 0.0 になっている。ここで、Figure 1.9 に示す副作用に注目しよう。補完された 3 行を μ に置くと、σ の分子にはゼロしか加わらない一方で n が 7 から 10 に増えるため、部屋数の標準偏差が 1.11 から 0.91 へと縮小し、観測値の z スコアが（1.54 から 1.89 へと、行 6 において）膨らむのである。

データセットを**訓練セット**（training set）と**テストセット**に分割する際には、方法論上の 1 つの原則が何より重要となる。すなわち、**補完に用いる平均や中央値、カテゴリの語彙、標準化の**

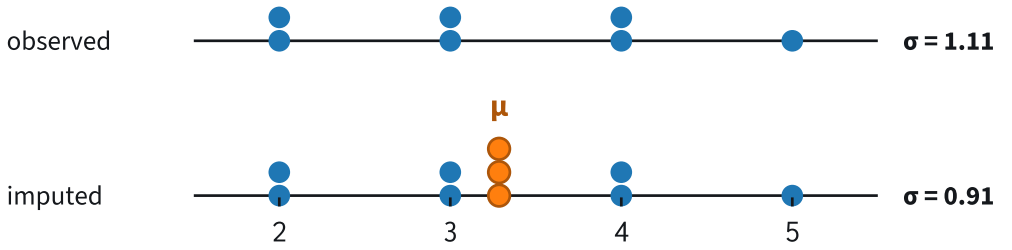


Figure 1.9: 平均補完の前後の部屋数。補完された 3 つの値（オレンジ）はすべて、観測された 7 つの値の平均 μ 上に重なるため、この列は元のデータよりもばらつきが小さくなり、標準偏差は 1.11 から 0.91 へと低下する。

(μ, σ) **を含むすべての前処理統計量は、訓練セットのみから算出しなければならず**、それを変更せずにそのままテストセットへ適用する（Figure 1.10）。分割前のデータセット全体で統計量を算出すると、評価用に確保したテストデータから訓練へと情報が漏洩し（**データリーケージ**）、テストスコアが不当に高く出てしまう。4.7 節では、本格的な不動産ベンチマークに対してこの原則を徹底する（演習問題 4 も参照）。

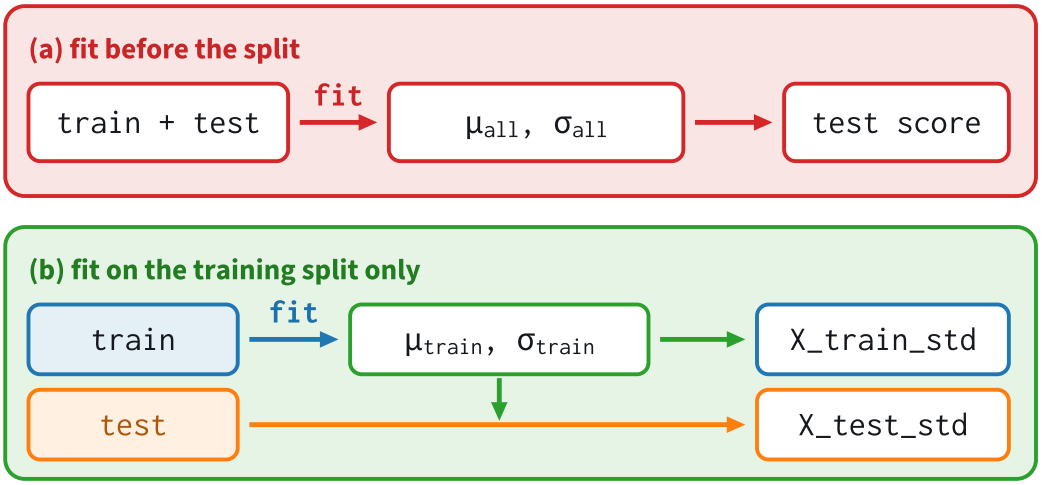


Figure 1.10: 統計量の算出とデータ分割の順序。(a) すべての行で算出した統計量はすでにテストセットを見ているため、テストスコアが楽観的になりすぎる。(b) 訓練データ分割のみで算出した統計量を変更せずに両方の分割へ適用すれば、テストセットは未知のまま保たれる。

1.2.3.2 テンソルへの変換

最後に、`.to_numpy(dtype='float32')` を使って整形済みの列を連続した 32 ビット浮動小数点の NumPy 配列として取り出し（pandas のデフォルトである `float64` からダウンキャストする）、フレームワークのテンソルコンストラクタに渡せる状態にする（1.1 節）。

```
X = torch.tensor(inputs.to_numpy(dtype='float32'))
y = torch.tensor(targets.to_numpy(dtype='float32'))
X.shape, X.dtype, y.shape, y.dtype
```

```
(torch.Size([10, 5]), torch.float32, torch.Size([10]), torch.float32)
```

特徴行列 `X` の形状は `(10, 5)`（住宅 10 件に対し、標準化された数値特徴量 2 列と屋根材インジケータ 3 列）、ターゲットベクトル `y` の形状は `(10,)` であり、いずれもデータ型は `float32` である。実運用のパイプラインでは、この段階に至る前に複数のリレーショナルテーブルを結合し、破損レコードやセンチネルコード（不明な座標を表す `-9999` など）を検査することが多い。こうして計画行列（design matrix）$\mathbf{X}$ にまとめられたデータは、次節で扱う線形代数の演算へと引き渡される。

1.2.4 まとめ

実世界の表形式データセットには、ほぼ例外なく欠損値やカテゴリ型のテキスト項目、大きく異なるスケールで測定された数値列が含まれているが、ニューラルネットワークには整った `float32` テンソルが必要である。pandas を用いれば、`read_csv` で CSV ファイルを DataFrame に読み込み、`info()` と `describe()` で列の型や欠損値の数、値の分布を確認し、最初の段階で入力特徴量からターゲット列を分離できる。欠損値は、削除、補完（数値の欠損を訓練セットの平均や中央値で埋めるなど）、または欠損性インジケータ列によって対処できる。順序のないカテゴリ列は、架空の数値的順序を作り出さないようワンホットエンコーディング（`get_dummies`）によって二値（$0/1$）のインジケータ列へと変換し、連続値の数値列は訓練データ分割のみで計算した統計量を用いて平均 0 と分散 1 へと標準化する（$x \mapsto (x - \mu)/\sigma$）。最後に、`.to_numpy(dtype='float32')` によって整形済みの DataFrame を深層学習フレームワークのテンソルへと直接橋渡しする。

1.2.5 演習問題

1. [code] **列名による選択と位置による選択の比較**。本節では、`data` を `data.iloc[:, :3]` と `data.iloc[:, 3]` を用いて整数の位置により特徴量とターゲットに分割した。ここで、`house_tiny.csv` の更新版において `YearBuilt` という新しい列が `Area` と `Price` の間に挿入されたとする。
    1. 更新されたファイルに対して位置による分割 `data.iloc[:, :3]` と `data.iloc[:, 3]` を実行すると、`inputs` と `targets` にはそれぞれ何が入るか。本節の後続のコード行で例外が発生するか、それともパイプラインは気づかないうちに誤ったターゲットを予測するモデルを訓練してしまうか。
    2. 列の順序や新たに追加された特徴量にかかわらず、列名を使って（たとえば `data['Price']` と `data.drop(...)` により）`targets` が常に `Price` を抽出し、`inputs` が他のすべての列を保持するよう、特徴量とターゲットの分割を書き換えよ。詳細については、データのインデックス参照と選択に関する pandas ユーザーガイドを参照せよ。
2. [code] **補完戦略の比較**。`NumRooms` の欠損要素を次の 3 つの方式で補完せよ。(i) 平均による補完、(ii) 中央値による補完、(iii) 平均による補完に二値の `NumRooms_isna` インジケータ列を追加する方式。
    1. 平均による補完と中央値による補完のそれぞれについて、補完後の `NumRooms` 列の標本平均と標本標準偏差を計算し、元の 7 個の観測値の平均および標準偏差と比較したうえで、なぜ標準偏差がその方向に変化するのかを説明せよ。
    2. より大規模な不動産データセットにおいて、`NumRooms` の未報告が主に高級邸宅で発生しているとする。平均による補完のみを用いた場合、モデルにはそれらの邸宅の部屋

数がいくつとして見えるか。また、線形モデル（入力列の重み付き和にバイアスを加えて価格を予測するモデル）が、追加された `NumRooms_isna` インジケータ列をどのように利用してその系統的な過小評価を補正できるか説明せよ。

3. 3 つの対処法（削除、平均による補完のみ、インジケータを伴う補完）のそれぞれが、値の欠損を引き起こすメカニズムについてどのような統計的仮定を置いているかを明確に述べよ。

Kaggle Learn の Intermediate Machine Learning 演習問題 2 を改変。

3. **高カーディナリティのカテゴリ特徴量。**ワンホットエンコーディングは、k 個の異なるカテゴリを持つカテゴリ列を k 本の二値列（`dummy_na=True` の場合は $k+1$ 本）に展開する。

   1. $n = 10^6$ 件の住宅からなる表に、欠損値のない $k = 10^4$ 通りの郵便番号を持つ `ZipCode` 列が含まれているとする。ワンホットエンコーディング後の行列を 64 ビット浮動小数点（`float64`）の密行列として保存した場合に何 GB を占めるかを計算し、非ゼロ要素の割合を答えよ。
   2. さらにこの表に、すべての行に対して一意な識別子文字列を持つ `ListingID` 列も含まれているとする。`ListingID` を扱う 3 つの方法、すなわち (i) ワンホットエンコーディング、(ii) 各 ID を学習可能な埋め込みベクトルに写像する方法、(iii) その列を完全に除外する方法を比較せよ。それぞれの方法のもとで、モデルは `ListingID` から、`ListingID` が訓練セットに一度も現れなかった新規登録の住宅へ汎化するような何を学習できるか。また、どの方法を選ぶべきか。

   Kaggle Learn の Intermediate Machine Learning 演習問題 3 を改変。

4. [code] **データリーケージの定量化。**本文では、`NumRooms` の欠損要素の補完と `NumRooms` および `Area` の標準化に、全 10 行にわたって計算した標本平均と標準偏差を用いた。ここでは `data` を、最初の 8 行からなる訓練セット（`data.iloc[:8]`）と最後の 2 行からなるテストセット（`data.iloc[8:]`）に分割せよ。

   1. 補完の平均および標準化パラメータ（μ と σ）を**訓練用の 8 行のみから**計算するように前処理パイプラインを再実行し、その訓練パラメータを変更せずに用いて訓練データ分割とテストデータ分割の両方を補完・標準化せよ。得られた標準化後のテスト行を、μ と σ を全 10 行から計算したときの値と比較せよ。
   2. 表全体を使う手順のもとでは、評価用に確保したテスト住宅に関するどのような情報が訓練特徴量へ漏洩したのかを正確に説明せよ。
   3. あるカテゴリ特徴量または数値特徴量が、たまたま訓練行のすべてで一定の値をとり（したがって訓練データの標準偏差が $\sigma = 0$ となり）、テスト行では変化するとする。訓練データのみに基づく標準化の式 $(x-\mu)/\sigma$ は、訓練データ分割とテストデータ分割のそれぞれで何を生成するか。また、堅牢な前処理パイプラインは分散ゼロの特徴量をどのように扱うべきか。

1.3 線形代数

画像のミニバッチを 255 で割る操作は各ピクセルを個別にスケーリングするだけだが、予測モデルでは数百から数千もの入力特徴量を組み合わせて判断を下さなければならない。**線形代数**（linear algebra）はそのような重み付き結合を記述する数学の言語であり、ミニバッチ全体に対する 1 つの層の順伝播（forward pass）を簡潔な 1 本の式で表し、出力の形状や浮動小数点演算のコストを実行前に読み取ることを可能にする。

28×28 のグレースケール衣類画像（第 3 章）を 10 個のカテゴリに分類する線形画像分類器（image classifier）を考えよう。各写真を平坦化すると $28 \times 28 = 784$ ピクセルになる。平坦化した 256 枚の画像のミニバッチをデータ行列 $\mathbf{X} \in \mathbb{R}^{256\times 784}$ の行として積み重ね、784 個ずつ

の重みを 10 個のカテゴリそれぞれについて $\mathbf{W} \in \mathbb{R}^{784\times10}$ の列として並べ、10 個のカテゴリごとのオフセットを $\mathbf{b} \in \mathbb{R}^{10}$ にまとめると（パラメータ数は合計 7,850 個）、ミニバッチ全体の $256 \times 10 = 2{,}560$ 個のカテゴリスコアは次の式で与えられる。

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}, \tag{1.2}$$

ここで、$\mathbf{b}$ はブロードキャストによってすべての行に加算される（Figure 1.11）。

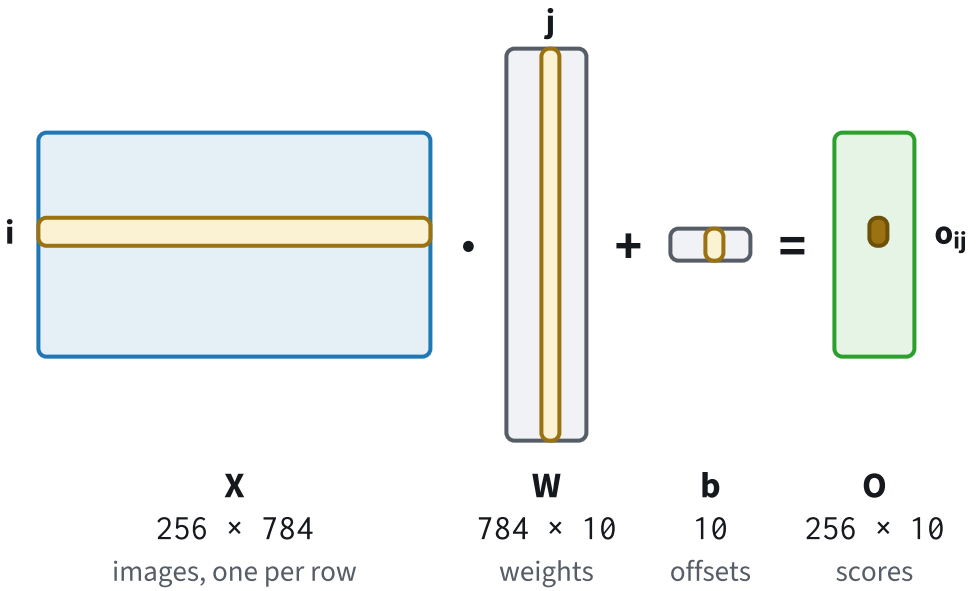


Figure 1.11: 256 枚の画像と 10 個のカテゴリからなるミニバッチに対する線形分類器のスコア。$\mathbf{X}$ の第 i 行は画像 i を保持し、$\mathbf{W}$ の第 j 列はカテゴリ j の重みを保持し、この 2 つからスコア o_{ij} が定まる。

本節では、スカラー、ベクトル、行列、テンソルから出発し、リダクション、ドット積、行列積を経て $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ へと積み上げ、最後に**ノルム**と**固有値**（eigenvalue）を扱う（記法の規約は記法にまとめられている）。

```
import torch
```

1.3.1 スカラー、ベクトル、行列、テンソル

0^{th}、1^{st}、2^{nd}、4^{th} 階の配列の作成と形状変更はすでに 1.1 節で扱った。ここでは、各階数（コードでは `ndim`）を標準的な数学上の名称および記法と対応づける。

1.3.1.1 スカラー

単一の数値は**スカラー**と呼ばれ、通常の小文字イタリック体（x, y, z）で表される。$x \in \mathbb{R}$ と書けば、x が実数であることを表す。コードでは、スカラーは 0^{th} 階のテンソルであり、空の形状 `()` を持つ。

```
x = torch.tensor(3.0)
x, x.shape
```

```
(tensor(3.), torch.Size([]))
```

1.3.1.2 ベクトル

ベクトルは固定長のスカラーの 1 次元配列（スカラーはその**要素**または**成分**）であり、太字の小文字（$\mathbf{x}, \mathbf{y}, \mathbf{w}$）で表される。$\mathbf{x} \in \mathbb{R}^n$ は n 次元の実ベクトルを表し、特に断らない限り**列ベクトル**（column vector）として扱う。

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \tag{1.3}$$

文中にインラインで書く場合は $\mathbf{x} = (x_1, \dots, x_n)$ とも表記する。コードでは、ベクトルは 1^{st} 階のテンソルである。

```
x = torch.arange(3, dtype=torch.float32)
x
```

```
tensor([0., 1., 2.])
```

数学の記法では 1 始まりのインデックス（$x_1, \dots, x_n$）を使うのに対し、Python では 0 始まりのインデックスを使うため、3 番目の成分 x_3 には `x[2]` でアクセスする。

```
x[2]
```

```
tensor(2.)
```

ベクトルの**次元数**（dimensionality）n（成分の個数）は、Python の組み込み関数 `len(x)` によって、あるいは要素数 1 のタプルである形状 `x.shape` の唯一の要素として得られる。

```
len(x), x.shape
```

```
(3, torch.Size([3]))
```

「次元」という言葉が曖昧に使われることによる混乱を防ぐため、本書ではテンソルの軸の数（`ndim`）には**階数**（order）、ベクトルの成分の数には**次元数**という用語を充てる。

1.3.1.3 行列

行列は 2^{nd} 階のテンソルであり、太字の大文字（$\mathbf{A}, \mathbf{X}, \mathbf{W}$）で表される。$\mathbf{A} \in \mathbb{R}^{m \times n}$ と書けば m 行 n 列であることを示し（$m = n$ のとき $\mathbf{A}$ は**正方行列**（square matrix）である）、成分 a_{ij} が第 i 行・第 j 列に並ぶ。

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{1.4}$$

表形式データや平坦化したミニバッチの行列 $\mathbf{X} \in \mathbb{R}^{n \times d}$ では、1 行につき 1 つのデータ例、1 列につき 1 つの特徴量を格納する。以下では、2×3 行列 `A` を構築する。

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
A
```

```
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

行列の行と列を入れ替えると、その**転置**（transpose）$\mathbf{A}^\top \in \mathbb{R}^{n \times m}$（成分は $[\mathbf{A}^\top]_{ij} = a_{ji}$）が得られ、コードでは `.T` でアクセスする。

```
A.T
```

```
tensor([[0., 3.],
        [1., 4.],
        [2., 5.]])
```

正方行列は、$\mathbf{S} = \mathbf{S}^\top$（$s_{ij} = s_{ji}$ がすべての i, j で成り立つ）のとき**対称行列**（symmetric matrix）である。

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
(S == S.T).all()
```

```
tensor(True)
```

1.3.1.4 高階テンソル

3 つ以上の軸を持つテンソルは、サンセリフ体の大文字（`X`, `Y`, `Z`）で表される。以下では、合成した 4 次元の画像ミニバッチ `images`（形状は $(256, 1, 28, 28)$、要素は $[0, 1]$ の範囲）を生成し、各画像を平坦化して 2 次元行列 $\mathbf{X} \in \mathbb{R}^{256 \times 784}$ を作る。

```
images = torch.rand(256, 1, 28, 28)
X = images.reshape(256, -1)  # One row of 784 pixels per image
images.shape, X.shape
```

```
(torch.Size([256, 1, 28, 28]), torch.Size([256, 784]))
```

1.3.2 要素ごとの演算とリダクション

1.3.2.1 要素ごとの演算とアダマール積

同じ形状を持つ 2 つの行列 $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ の要素ごとの積は**アダマール積**（Hadamard product）$\mathbf{A} \odot \mathbf{B}$ と呼ばれ、その成分は $[\mathbf{A} \odot \mathbf{B}]_{ij} = a_{ij} b_{ij}$ であり、コードでは `*` で計算する。

```
A * A
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

1.3.2.2 軸に沿ったリダクション

`axis=0` を `.sum()` に渡すと行方向にわたって和をとり列和のベクトル（$\sum_{i=1}^{m} a_{ij}$）を生成するのに対し、`axis=1` を渡すと列方向にわたって和をとり行和のベクトル（$\sum_{j=1}^{n} a_{ij}$）を生成する。`axis` を省略すると全要素の和をとる（Figure 1.12）。

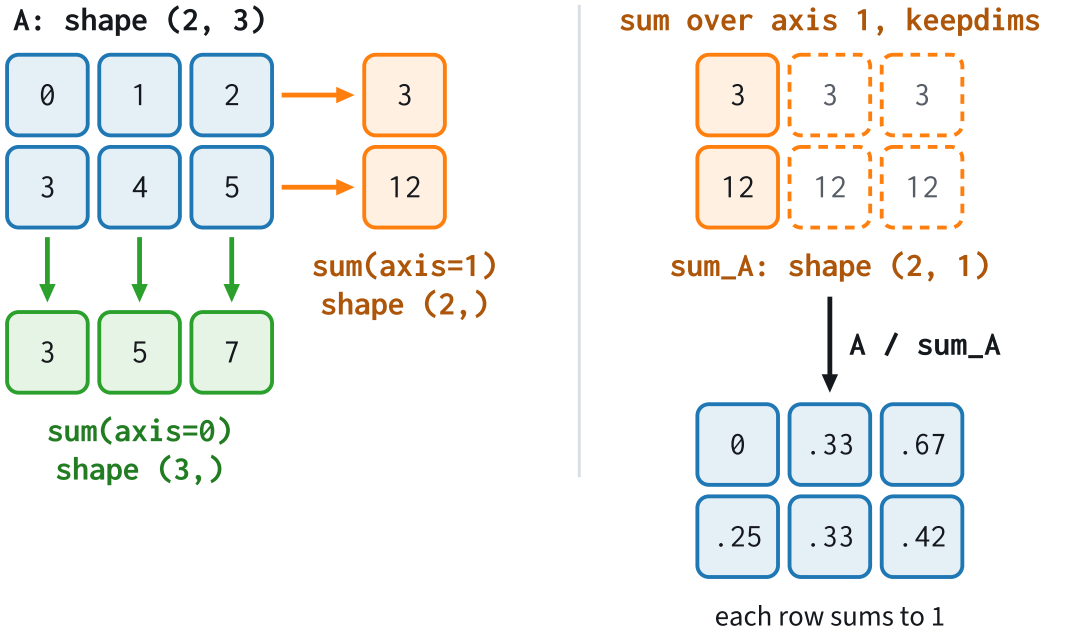


Figure 1.12: 2×3 行列のリダクション。軸 0 にわたって和をとると列ごとに 1 つの和が残り、軸 1 にわたって和をとると行ごとに 1 つの和が残る（リダクションされた軸は形状から消える）。`keepdims=True` を指定すると行和は形状 $(2, 1)$ を保つため、ブロードキャストによって 3 つの列へと引き伸ばされ、除算 `A / sum_A` によって各行の和が 1 になるようスケーリングされる。

```
A.sum(), A.sum(axis=0), A.sum(axis=1)
```

```
(tensor(15.), tensor([3., 5., 7.]), tensor([ 3., 12.]))
```

1.3.2.3 ブロードキャストのためのリダクション軸の保持

`A`（形状 `(2, 3)`）の各行をその行和で正規化しようとしても、`A.sum(axis=1)`（形状 `(2,)`）は `(2, 3)` に対してブロードキャストできない。末尾の次元 2 と 3 が衝突するためである。`keepdims=True` を渡すとリダクションされた軸が長さ 1 のまま保持され、`sum_A` の形状は `(2, 1)` になる。

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

PyTorch では `dim`・`keepdim` と、NumPy 互換の別名である `axis`・`keepdims` の両方を受け付ける。

`sum_A` の形状が `(2, 1)` であるため、`A / sum_A` で割ると各行の和が 3 つの列すべてにブロードキャストされ、得られる 2×3 行列のどの行も和が 1 になる。

```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

256×784 のミニバッチ $\mathbf{X}$ において、ピクセルごとの列平均（`axis=0`、形状 `(784,)`）を引く操作はそのままブロードキャストされるが、画像ごとの行平均（`axis=1`）を引くには `keepdims=True`（形状 `(256, 1)`）が必要である。

```
col_means = X.mean(axis=0)
row_means = X.mean(axis=1, keepdims=True)
col_means.shape, row_means.shape, (X - row_means).shape
```

```
(torch.Size([784]), torch.Size([256, 1]), torch.Size([256, 784]))
```

注意すべき点として、もし $\mathbf{X}$ がたまたま正方行列（784×784）であった場合、`keepdims=True` を `X - X.mean(axis=1)` で省略してもエラーは発生せず、画像 j の平均がすべての画像の第 j 列から暗黙のうちに引かれてしまう（演習問題 2）。

1.3.3 積

1.3.3.1 ドット積

2 つのベクトル $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ が与えられたとき、それらの**ドット積**、すなわち**内積**（inner product）$\mathbf{x}^\top \mathbf{y}$（$\langle \mathbf{x}, \mathbf{y} \rangle$ とも書く）は、対応する成分同士の積の総和である。

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i. \tag{1.5}$$

コードでは、2 つの 1 次元ベクトル `x` と `y` のドット積をフレームワークの `dot` 関数で直接計算する。

```
y = torch.ones(3)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

同じ計算は、`x * y` で要素ごとに掛けてから得られた積を `sum` で足し合わせることでも実行できる（$\sum_i x_i = 0 + 1 + 2 = 3$ が $\mathbf{y} = \mathbf{1}$ のときに得られる）。

```
torch.sum(x * y)
```

```
tensor(3.)
```

1.3.3.2 行列・ベクトル積

$\mathbf{A} \in \mathbb{R}^{m \times n}$ を m 個の行ベクトル $\mathbf{a}_1^\top, \dots, \mathbf{a}_m^\top \in \mathbb{R}^n$ に分割すると、**行列・ベクトル積**（matrix–vector product）$\mathbf{Ax}$（ただし $\mathbf{x} \in \mathbb{R}^n$）は、各行とのドット積を並べた m 次元ベクトルになる（Figure 1.13）。

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix} \in \mathbb{R}^m.$$

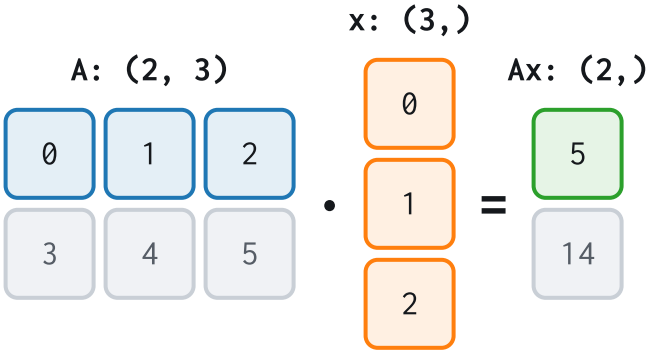


Figure 1.13: 行列・ベクトル積 $\mathbf{Ax}$ は $\mathbf{A}$ の行ごとに 1 つのドット積をとるため、2×3 行列は長さ 3 のベクトルを長さ 2 のベクトルに写す。1 行目からは $0 \cdot 0 + 1 \cdot 1 + 2 \cdot 2 = 5$ が得られる。

Python では、中置演算子 `@` によって行列・ベクトル積と行列・行列積の両方を計算できる。

```
A.shape, x.shape, A @ x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]))
```

列ごとに見れば、$\mathbf{Ax} = \sum_{j=1}^{n} x_j (\mathbf{Ae}_j)$ は $\mathbf{A}$ の列ベクトルの線形結合であり、ここで j^{th} 列 $\mathbf{Ae}_j$ は j^{th} 標準基底ベクトル（basis vector）$\mathbf{e}_j$ の像である。$\mathbb{R}^2$ において、2×2 行列は $\mathbf{e}_1, \mathbf{e}_2$ が張る単位正方形を、その列ベクトルが張る平行四辺形へと写す（Figure 1.14）。

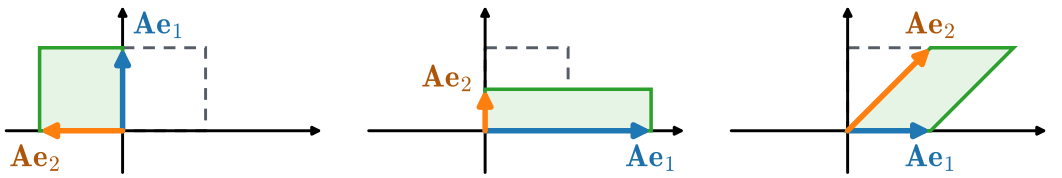


Figure 1.14: 2×2 行列は単位正方形（破線）を、その列ベクトル $\mathbf{Ae}_1$ と $\mathbf{Ae}_2$ が張る平行四辺形に写す。左から順に、90° の回転、2 倍（$\mathbf{e}_1$ 方向）と $1/2$ 倍（$\mathbf{e}_2$ 方向）のスケーリング、せん断。

1.3.3.3 行列・行列積

$\mathbf{A} \in \mathbb{R}^{m \times k}$ と $\mathbf{B} \in \mathbb{R}^{k \times n}$ が与えられたとき、それらの行列積 $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times n}$ の (i, j) 成分は、$\mathbf{A}$ の第 i 行と $\mathbf{B}$ の第 j 列のドット積に等しい。

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}. \tag{1.6}$$

```
B = torch.ones(3, 4)
A @ B
```

```
tensor([[ 3.,  3.,  3.,  3.],
        [12., 12., 12., 12.]])
```

これで、10 クラスの線形分類器 $\mathbf{O} = \mathbf{XW} + \mathbf{b}$ を 256×784 のミニバッチ `X` に対して評価し、`argmax` を軸 1 に沿って適用することで各画像のスコアが最も高いカテゴリを選べる。

```
W = 0.01 * torch.randn(784, 10)  # Column j: category j
b = torch.zeros(10)              # Entry j: category j
O = X @ W + b                    # Row i: image i
O.shape, O.argmax(dim=1).shape
```

```
(torch.Size([256, 10]), torch.Size([256]))
```

$m \times k$ 行列と $k \times n$ 行列の積では、mn 個の長さ k のドット積を計算するため、コストは $\approx 2mkn$ FLOPs となる（$2 \times 256 \times 784 \times 10 \approx 4 \times 10^6$ FLOPs が `X @ W` にかかり、また $2n^3$ FLOPs がメモリ上のわずか $3n^2$ 個の要素に対して 2 つの $n \times n$ 行列の積で実行される。第 13 章を参照）。線形代数では 2 つの特別な正方行列が繰り返し登場する。1 つは**単位行列**（identity matrix）$\mathbf{I} \in \mathbb{R}^{n \times n}$（主対角成分が 1 でそれ以外が 0）であり、任意のベクトルを変化させない（$\mathbf{Ix} = \mathbf{x}$）。もう 1 つは**逆行列**（inverse）$\mathbf{A}^{-1}$ であり、可逆行列（invertible matrix）$\mathbf{A}$ に対して $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ によって定義される（33.2.2.3 節）。

後の章で扱うモデルでは、ミニバッチ内のデータ例ごとに 1 つずつ用意された行列のスタック全体を一度に掛け合わせる。`@` 演算子はこれをループなしで実行する。すなわち、3 つ以上の軸を持つテンソルに対しては最後の 2 つの軸で行列積を計算し、先頭側の軸にわたってブロードキャストする（1.1.3.2 節）。`@` では対応できない場合でも、`einsum` を使えば Equation 1.6 の和を添字ごとに書き下せる。文字列 `'bil,lj->bij'` は各因子と結果の軸に名前を付けたものであり、結果に含まれない添字（ここでは `l`）について和がとられる。`A` の定数倍を 3 つ積み重ね、それらすべてに `B` を掛けてみよう。

```
As = torch.stack([A, 2 * A, 3 * A])  # Three 2 x 3 matrices, shape (3, 2, 3)
C = As @ B                           # B is broadcast across the batch
C.shape, (torch.einsum('bil,lj->bij', As, B) == C).all()
```

```
(torch.Size([3, 2, 4]), tensor(True))
```

結果にはスタック内の行列ごとに 1 つずつ、計 3 個の 2×4 の積が格納されており、`einsum` の結果はすべての成分で `@` と一致する。

1.3.4 ノルム

ノルムはベクトルや行列の大きさを測る量であり、機械学習では予測誤差 $\hat{\mathbf{y}} - \mathbf{y}$ を測る**損失関数**（loss function）としても、重みの大きさを制約する**正則化ペナルティ**（regularization penalty）としても用いられる。

1.3.4.1 ユークリッド長と角度

ℓ_2 **ノルム**（または**ユークリッドノルム**）は、$\mathbf{x} \in \mathbb{R}^n$ に対して次の式で定義される。

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top \mathbf{x}} = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{1.7}$$

これはしばしば $\|\mathbf{x}\|$ と略記される。平面上の 2 つのベクトル $\mathbf{x}, \mathbf{y}$ が角度 θ をなすとき、余弦定理 $\|\mathbf{x}-\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\|\mathbf{x}\|\|\mathbf{y}\|\cos\theta$ と双線形展開 $(\mathbf{x}-\mathbf{y})^\top(\mathbf{x}-\mathbf{y}) = \|\mathbf{x}\|^2 - 2\mathbf{x}^\top\mathbf{y} + \|\mathbf{y}\|^2$ を等置すると、次式が得られる（Figure 1.15）。

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\|\cos\theta. \tag{1.8}$$

任意の次元 d において、Equation 1.8 を逆に読むことで、非ゼロベクトル間の**コサイン類似度**（cosine similarity）を次のように定義する。

$$\cos\theta = \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}, \tag{1.9}$$

この量は向きのみに依存し、$+1$（同じ向き）から 0（直交）を経て -1（反対向き）までの値をとる（Figure 1.16）。

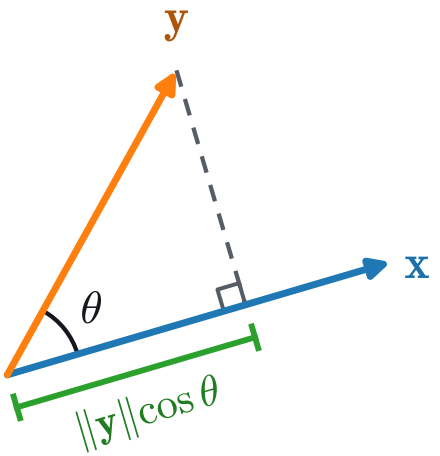


Figure 1.15: ドット積 $\mathbf{x}^\top\mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\|\cos\theta$ は、$\mathbf{x}$ の長さと符号付き長さ $\|\mathbf{y}\|\cos\theta$（$\mathbf{y}$ を $\mathbf{x}$ を通る直線へ射影した長さ）の積である。

コーシー・シュワルツの不等式（Cauchy–Schwarz inequality）により $|\mathbf{x}^\top\mathbf{y}| \le \|\mathbf{x}\|\|\mathbf{y}\|$ が保証される（したがって $\cos\theta \in [-1, 1]$）。実際、単位ベクトル $\mathbf{u} = \mathbf{x}/\|\mathbf{x}\|$ と $\mathbf{v} = \mathbf{y}/\|\mathbf{y}\|$ に対して $0 \le \|\mathbf{u} \mp \mathbf{v}\|^2 = 2 \mp 2\mathbf{u}^\top\mathbf{v}$ を展開すれば $|\mathbf{u}^\top\mathbf{v}| \le 1$ が得られる。

```
def cosine(x, y):
    return torch.dot(x, y) / (torch.linalg.vector_norm(x) *
```

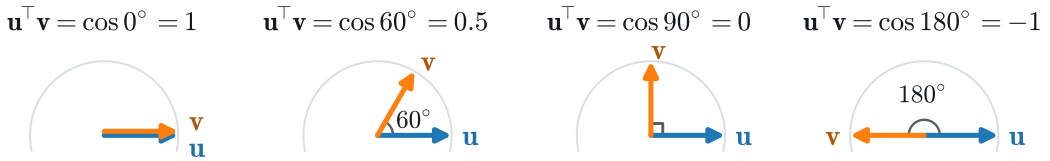


Figure 1.16: 角度 θ をなす 2 つの単位ベクトル。それらのドット積は $\cos\theta$ に等しく、1（同じ向き）から 0（垂直）を経て -1（反対向き）までの値をとる。

```
                                torch.linalg.vector_norm(y))

x, y = torch.tensor([1.0, 1, 1, 1]), torch.tensor([1.0, 1, 1, -1])
cosine(x, y), cosine(2 * x, y), torch.acos(cosine(x, y))
```

```
(tensor(0.5000), tensor(0.5000), tensor(1.0472))
```

1.3.4.2 ℓ_1 ノルム、ℓ_p ノルム、フロベニウスノルム

ℓ_1 ノルム（または**マンハッタン距離**、Manhattan distance）は、各成分の絶対値の総和をとる（Figure 1.17）。

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|. \tag{1.10}$$

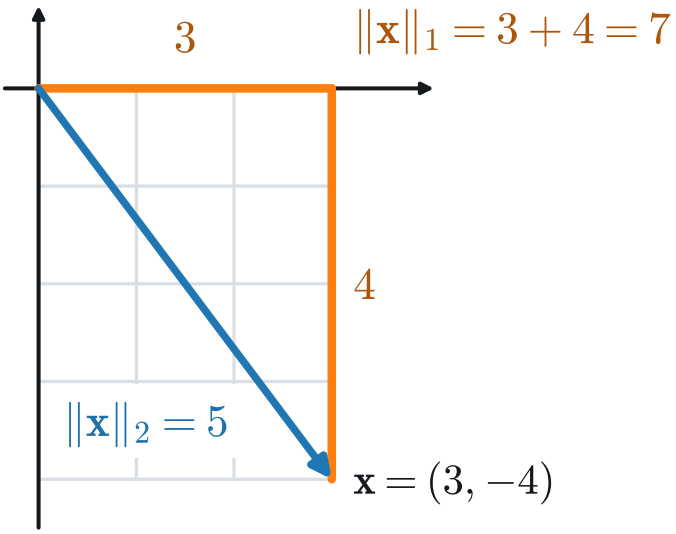


Figure 1.17: $\mathbf{x}$ の ℓ_2 ノルムは $\mathbf{x}$ への直線経路の長さであり、ℓ_1 ノルムは格子に沿った経路の長さである。

```
x = torch.tensor([3.0, -4.0])
torch.linalg.vector_norm(x), torch.abs(x).sum()
```

```
(tensor(5.), tensor(7.))
```

ℓ_2 は成分を二乗するのに対し ℓ_1 は二乗しないため、ℓ_1 損失は大きな外れ値（outlier）の誤差に対してより頑健であり、ℓ_1 の重みペナルティは重みを厳密な 0 へと導く（スパース性、sparsity）。どちらも ℓ_p **ノルム**の族 $\|\mathbf{x}\|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$（$p \geq 1$）に属し、$\|\mathbf{x}\|_\infty = \max_i |x_i|$ が $p \to \infty$ の極限に対応する。あらゆる正当なノルムは、（i）絶対斉次性 $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$、（ii）三角不等式（triangle inequality）$\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$、（iii）正定値性（$\|\mathbf{x}\| > 0$ が $\mathbf{x} \neq \mathbf{0}$ に対して成り立つ）を満たす。

$m \times n$ 行列 $\mathbf{A}$ に対して、**フロベニウスノルム**は ℓ_2 ノルムをその mn 個の成分全体について計算したものである。

$$\|\mathbf{A}\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}. \tag{1.11}$$

```
torch.linalg.matrix_norm(torch.ones(4, 9))
```

```
tensor(6.)
```

PyTorch では `torch.linalg.vector_norm` と `torch.linalg.matrix_norm` が分かれており、後者は 3 次元テンソルを 2 次元行列のバッチとして扱う。

1.3.5 固有ベクトルと固有値

50 層のネットワークが入力ベクトル $\mathbf{x}$ に行列を繰り返し掛けると何が起きるだろうか。非ゼロベクトル $\mathbf{v} \neq \mathbf{0}$ が $\mathbf{A} \in \mathbb{R}^{n\times n}$ のスカラー**固有値** λ に対応する**固有ベクトル**（eigenvector）であるとは、$\mathbf{A}$ を掛けても $\mathbf{v}$ が回転せず単に定数倍されることをいう。

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \quad \Longrightarrow \quad \mathbf{A}^k\mathbf{v} = \lambda^k\mathbf{v}. \tag{1.12}$$

先ほどの対称行列 $\mathbf{S} = \left[\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right]$ では、$(1,1)$ は $\lambda = 3$ に対応する固有ベクトルであり、$(-1,1)$ は $\lambda = 1$ に対応する固有ベクトルである（Figure 1.18）。

実対称行列の固有値と単位固有ベクトルは `linalg.eigh` を使って計算する。

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
eigenvalues, eigenvectors = torch.linalg.eigh(S)
eigenvalues, eigenvectors
```

```
(tensor([1., 3.]),
 tensor([[-0.7071,  0.7071],
         [ 0.7071,  0.7071]]))
```

スペクトル定理（spectral theorem、33.3 節）により、すべての $n \times n$ 実対称行列 $\mathbf{A}$ は n 個の実固有値と、固有ベクトルからなる完全な正規直交基底（orthonormal basis）$\mathbf{v}_1, \ldots, \mathbf{v}_n$ を持つ。任意のベクトルを $\mathbf{x} = \sum_{i=1}^{n} c_i\mathbf{v}_i$（ただし $c_i = \mathbf{v}_i^\top\mathbf{x}$）と展開すると、次式が得られる。

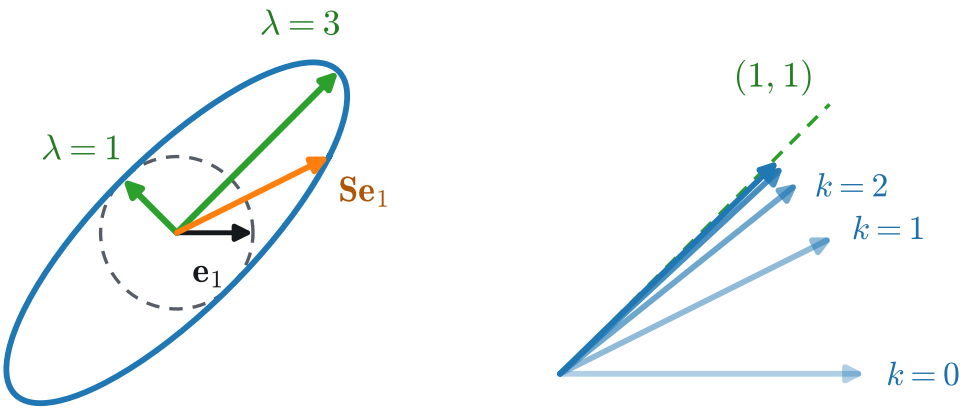


Figure 1.18: 左：$\mathbf{S}$ は単位円（破線）を、固有ベクトルを主軸とし $\lambda = 3$ 倍および $\lambda = 1$ 倍に引き伸ばされた楕円へと写し、$\mathbf{e}_1$ は $\mathbf{Se}_1 = (2, 1)$ へと向きを変える。右：$\mathbf{S}^k\mathbf{e}_1$（$k = 0, \dots, 4$）の向きは、最大固有値の固有ベクトル $(1, 1)$ に近づいていく。

$$\mathbf{A}^k\mathbf{x} = \sum_{i=1}^{n} c_i \lambda_i^k \mathbf{v}_i. \tag{1.13}$$

k が大きくなるにつれて $|\lambda_i|$ が最大の項が支配的になり、$\mathbf{A}^k\mathbf{x}$ は主固有ベクトルの向きに揃いながら 1 ステップあたり $\max_i |\lambda_i|$ 倍で成長する（**べき乗法**、power iteration）。対称行列では、この成長率は任意のベクトルをその行列が引き伸ばせる最大の倍率、すなわち**スペクトルノルム** $\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq\mathbf{0}} \|\mathbf{Ax}\|/\|\mathbf{x}\|$ にも一致する。

```
norm = torch.linalg.vector_norm
v = torch.tensor([1.0, 0.0])
for _ in range(10):
    v_prev, v = v, S @ v
growth = norm(v) / norm(v_prev)
growth, v / norm(v)
```

```
(tensor(3.0000), tensor([0.7071, 0.7071]))
```

50 層の思考実験に戻ろう。$\mathbf{M} = \left[\begin{smallmatrix} 1 & 0.1 \\ 0.1 & 1 \end{smallmatrix}\right]$ の固有値 $\lambda_1 = 1.1$ と $\lambda_2 = 0.9$ は 1 からわずか 10% しか離れていないが、その 2 つの単位固有ベクトルに $\mathbf{M}$ を 50 回掛けると、1 つ目は $1.1^{50} \approx 117.4$ 倍に増幅され、2 つ目は $0.9^{50} \approx 0.0052$ 倍に縮小する。

```
M = torch.tensor([[1.0, 0.1], [0.1, 1.0]])
# The eigenvectors (1, 1) and (-1, 1) of M, as columns of length 1
V = torch.tensor([[1.0, -1.0], [1.0, 1.0]]) / 2 ** 0.5
for _ in range(50):  # 50 "layers"
    V = M @ V
torch.linalg.vector_norm(V, dim=0)
```

```
tensor([1.1739e+02, 5.1538e-03])
```

ネットワークの重み行列は対称行列ではなく、その大半は正方行列ですらない。そのような行列がベクトルをどれだけ引き伸ばすかも、固有値からわかるのだろうか。答えは否である。$\mathbf{A} = \left[\begin{smallmatrix} 0 & 2 \\ 0 & 0 \end{smallmatrix}\right]$ の固有値はどちらも 0 であり $\mathbf{A}^2 = \mathbf{0}$ となるが、それでも $\mathbf{A}$ は $\mathbf{e}_2$ を $2\mathbf{e}_1$ に写すため長さを 2 倍にする。$\|\mathbf{Ax}\|_2^2 = \mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}$ であるから、ベクトルの伸縮を支配するのは対称行列 $\mathbf{A}^\top\mathbf{A}$ である。その固有値の平方根 $\sigma_i = \sqrt{\lambda_i(\mathbf{A}^\top\mathbf{A})}$ が $\mathbf{A}$ の**特異値**（singular value）であり、正方行列か否かにかかわらずすべての行列に対して存在する。$\mathbf{S}$ と $\mathbf{A}$ について計算してみよう。

```
A = torch.tensor([[0.0, 2.0], [0.0, 0.0]])  # Both eigenvalues are 0
svdvals = torch.linalg.svdvals
(svdvals(S), svdvals(A), torch.linalg.eigvalsh(A.T @ A).sqrt(),
 torch.linalg.matrix_rank(A))
```

```
(tensor([3., 1.]), tensor([2., 0.]), tensor([0., 2.]), tensor(1))
```

対称行列 $\mathbf{S}$ の特異値は（丸め誤差を除けば）3 と 1 であり、固有値の絶対値に等しい。すなわち、対称行列ではこれら 2 つの概念は一致する。$\mathbf{A}$ の特異値は 2 と 0 であり、$\mathbf{A}^\top\mathbf{A}$ の固有値からも同じ 2 つの数値が昇順で得られる。任意の行列において、最大の特異値はそのスペクトルノルムに等しい。また、非ゼロの特異値の個数は行列の**ランク**、すなわち積によって潰れずに残る独立な方向の数であり、$\mathbf{A}$ では 1 である。これら 2 つの事実はいずれも特異値分解（singular value decomposition、33.5 節）から導かれる。

各層の重み行列がベクトルの長さを平均して 1 の近くに保たない限り、活性化値や勾配は深さとともに指数関数的に**爆発**（explode）または**消失**（vanish）する（4.4 節）。

1.3.6 まとめ

線形代数はミニバッチ全体を扱うモデルを簡潔に表現する。平坦化した 256 枚の画像を $\mathbf{X} \in \mathbb{R}^{256\times 784}$ として積み重ねれば、全 2,560 個のカテゴリスコアを $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ によって一括計算できる。リダクションは指定した軸に沿って値を集約する（`keepdims=True` はブロードキャストのために軸を長さ 1 として保持する）。ドット積 $\mathbf{x}^\top\mathbf{y} = \sum_i x_i y_i$ は行列・ベクトル積 $\mathbf{Ax}$ と行列・行列積 $\mathbf{AB}$（計算量は $\approx 2mkn$ FLOPs）を構成する。ノルム（ℓ_2、ℓ_1、ℓ_p、フロベニウス、スペクトル）はベクトルや行列の大きさを測り、正規化したドット積はコサイン類似度 $\cos\theta = \mathbf{x}^\top\mathbf{y}/(\|\mathbf{x}\|\|\mathbf{y}\|)$ を与え、固有ベクトル（$\mathbf{Av} = \lambda\mathbf{v}$）は行列積の反復によって各成分が λ^k 倍にスケーリングされる理由を説明する。対称行列でない行列に対しては特異値がベクトルの伸縮を測り、その最大値がスペクトルノルムとなる。

1.3.7 演習問題

1. **転置と行列積**。次の代数恒等式を証明せよ。
   1. 行列積の転置が因子の順序を逆転させること、すなわち $(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top$ が（$\mathbf{AB}$ が定義される限り常に）成り立つことを示せ。（両辺の (i,j) 成分を Equation 1.6 を用いて書き出せ。）
   2. **任意の**長方行列 $\mathbf{X} \in \mathbb{R}^{m\times n}$ に対して、積 $\mathbf{X}\mathbf{X}^\top$ は常に正方行列かつ対称行列になることを示せ（$\mathbf{X}^\top\mathbf{X}$ についても同様）。本節の 256×784 の画像ミニバッチ $\mathbf{X}$ に対して、$\mathbf{X}\mathbf{X}^\top$ の形状はどうなるか。また、その (i,j) 成分は何を測っているか。
2. [code] **リダクションとブロードキャストの形状予測**。コードを一切実行せずに、`images`（形状 $(256, 1, 28, 28)$）と `X`（形状 $(256, 784)$）に対する以下の各式の結果の形状（またはエラー）を予測し、その後コードで答えを確認せよ。

1. `len(images)` は何を返すか。`images.sum(axis=(2, 3))` の形状は `keepdims=False` の場合と `keepdims=True` の場合でそれぞれどうなるか。ブロードキャストによって `images` をそのまま割れるのはどちらか。また、`images / images.sum(axis=(2, 3), keepdims=True)` は何を計算しているか。
2. `X - X.mean(axis=1)` を 256×784 行列 `X` に対して評価すると何が起きるか。また、`keepdims=True` によってそれがどのように修正されるか。
3. ここで、ミニバッチがたまたま $n = 784$ 枚の画像を含んでおり、`X` が 784×784 の正方行列であるとする。`X - X.mean(axis=1)` が**エラーを一切出さずに**実行されることを示せ。`X - X.mean(axis=1)` の (i, j) 成分が実際に何を計算しているかを `X` の成分を用いて書き出し、正方行列における暗黙のブロードキャストバグが形状エラーよりもはるかに危険である理由を説明せよ。

3. **ベクトルの集合は互いにどこまで鈍角になれるか。**以下の設問では、k 個のベクトル間のすべての対のドット積が同時にどこまで負になれるかを探る。

   1. 平面上の 3 つのベクトル $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^2$ は、$\mathbf{u}^\top \mathbf{v} < 0$、$\mathbf{v}^\top \mathbf{w} < 0$、$\mathbf{u}^\top \mathbf{w} < 0$ を同時に満たせるか。具体的な数値例を挙げるか、存在しないことを証明せよ。
   2. $k \geq 2$ 個の単位ベクトル $\mathbf{u}_1, \dots, \mathbf{u}_k \in \mathbb{R}^d$（$d \geq k - 1$）が対称に配置され、異なるすべての対がまったく同じドット積 $\mathbf{u}_i^\top \mathbf{u}_j = c$（すべての $i \neq j$）を持つとする。$0 \leq \|\mathbf{u}_1 + \cdots + \mathbf{u}_k\|_2^2$ を展開することで、$c \geq -\frac{1}{k-1}$ を証明せよ。10 クラス分類器における単位長のクラステンプレート 10 個がすべて等しい角度で離れているとき、この下界は 10 個のテンプレート間の対ごとのコサイン類似度について何を意味するか。

   MIT 18.06、問題セット 1、1.2 節、問題 28 より改題。

4. [code] **複数のパラメータテンソルにわたるノルム。**本節で導入したベクトルノルムのルーチンは、高階テンソルにも適用できる。

   1. 形状 $(2, 3, 4)$ の 3 次元テンソルに対してデフォルト引数でベクトルノルムを評価し、どのような数式を計算しているかを言葉で述べ、各要素を手動で二乗・総和・平方根をとることで答えを確認せよ。
   2. 深層ニューラルネットワークは、学習可能なパラメータをさまざまな階数と形状を持つ重みテンソルおよびバイアステンソルのリストに分けて保持する。訓練の監視や勾配クリッピング（8.4.2 節）の際には、全パラメータを 1 つの巨大なベクトルに連結したときの単一の全体 ℓ_2 ノルムがしばしば必要になる。テンソルを平坦化したり連結したりせずに、この全体 ℓ_2 ノルムを個々のテンソルノルムを用いて表し、形状 $(3, 4)$、$(4,)$、$(2, 3, 5)$ の 3 つのテンソルに対してコードで式を検証せよ。

5. **行列積の結合法則と計算コスト。**行列積は結合法則を満たし、数学的には $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ が成り立つが、連鎖する行列積に括弧を付ける順序によって計算コストは何桁も変わり得る。$\mathbf{A} \in \mathbb{R}^{1000 \times 50000}$、$\mathbf{B} \in \mathbb{R}^{50000 \times 20}$、$\mathbf{C} \in \mathbb{R}^{20 \times 10000}$ が与えられたとき、$(\mathbf{AB})\mathbf{C}$ と $\mathbf{A}(\mathbf{BC})$ の両方について、総 FLOP 数と中間の積行列の要素数を求めよ。2 つの評価順序で、演算量と中間メモリ量はそれぞれ何倍異なるか。

6. [code] **ループなしの全対ユークリッド距離行列。**●50 個ずつの $\mathbb{R}^8$ のベクトルからなる 2 つの集合が $\mathbf{X} \in \mathbb{R}^{50 \times 8}$ と $\mathbf{Y} \in \mathbb{R}^{50 \times 8}$ の行として積み重ねられているとき、Python のループを使わずに二乗ユークリッド距離の 50×50 行列 $\mathbf{D}$（成分 $D_{ij} = \|\mathbf{x}_i - \mathbf{y}_j\|_2^2$）を計算したい。

   1. まず 3 次元ブロードキャストによって $\mathbf{D}$ を計算せよ。すなわち、$\mathbf{X}$ を $(50, 1, 8)$ に、$\mathbf{Y}$ を $(1, 50, 8)$ に形状変更し、差をとって二乗し、最後の軸に沿って和をとれ。次に、$\mathbf{D}$ を二次式の展開 $\|\mathbf{x}_i - \mathbf{y}_j\|_2^2 = \|\mathbf{x}_i\|_2^2 + \|\mathbf{y}_j\|_2^2 - 2\mathbf{x}_i^\top \mathbf{y}_j$ によって計算せよ。これには行ノルムのリダクション、2 次元ブロードキャスト、および 1 回の行列積 $\mathbf{X}\mathbf{Y}^\top$ しか必

要ない。両者の結果が浮動小数点精度の範囲で一致することを確認せよ。

2. $\mathbf{X}$ と $\mathbf{Y}$ がそれぞれ $10{,}000$ 個の $\mathbb{R}^{128}$ のベクトルを含むとき、3 次元ブロードキャスト法が確保しなければならない中間の浮動小数点数の個数は、行列積を用いる方法と比べていくつになるか。大規模データではどちらの方法を使うべきか。
3. $\|\mathbf{x}_i\|_2^2 + \|\mathbf{y}_j\|_2^2 - 2\mathbf{x}_i^\top \mathbf{y}_j$ における浮動小数点の丸め誤差によって、ごく小さな負の値（-10^{-7} など）が $\mathbf{x}_i = \mathbf{y}_j$ のときに生じ得る理由を説明し、平方根をとる前に出力をクランプする方法を示せ。

Stanford CS231n、課題 1（k 近傍距離）、および fast.ai、Deep Learning from the Foundations、Lesson 8 より改題。

7. [code] **べき乗法と収束率。●**$\mathbf{v}_{k+1} = \mathbf{A}\mathbf{v}_k / \|\mathbf{A}\mathbf{v}_k\|_2$ をランダムな単位ベクトル $\mathbf{v}_0$ から繰り返し更新する正規化べき乗法を、次の 3×3 対称行列に対して実装せよ。

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 3 & 4 & 5 \end{bmatrix}. \tag{1.14}$$

1. 最初の反復回数 k（連続する反復ベクトル間のコサイン類似度の絶対値 $|\mathbf{v}_k^\top \mathbf{v}_{k-1}|$ が 0.999 を初めて超える時点）を記録せよ。収束したベクトル $\mathbf{v}_k$ を `eigh(A)` が返す主固有ベクトルと比較し、2 つのベクトルが全体の符号 ± 1 だけ異なり得る理由を説明せよ。
2. 各ステップにおいて、$\mathbf{v}_k$ と真の主固有ベクトル $\mathbf{u}_{\max}$ がなす角の正弦 $\sin\theta_k = \sqrt{1 - (\mathbf{v}_k^\top \mathbf{u}_{\max})^2}$ を測定し、1 ステップあたりの縮小比 $\sin\theta_{k+1} / \sin\theta_k$ を計算せよ。この漸近的な比を $\mathbf{A}$ の固有値を用いて Equation 1.13 から表し、べき乗法がごく少数のステップで収束した理由を説明せよ。
3. 次に、シフトした行列 $\mathbf{B} = \mathbf{A} - 9\mathbf{I}$ に対してべき乗法を実行せよ。べき乗法は $\mathbf{A}$ のどの固有値および固有ベクトルに収束するか。また、連続する反復ベクトル $\mathbf{v}_k$ と $\mathbf{v}_{k+1}$ の符号はどうなるか、その理由とともに答えよ。

MIT 18.06、問題セット 1、2.1 節、問題 29〜30 より改題。

1.4 微分積分

古代ギリシャにおいて、Archimedes は辺の数を $n \to \infty$ と増やした正多角形を円に内接させることで円の面積を計算した（Figure 1.19）。三角形が細くなるにつれて、その高さは r に近づき、底辺の合計は円周 $2\pi r$ に近づくため、面積は $\frac{1}{2} r (2\pi r) = \pi r^2$ となる。微小なステップの極限において曲がった量を単純な線形の断片で近似するという考え方は、積分法と**微分法**（differential calculus）の両方の根底にある。

深層学習において、微分法は**最適化**（optimization）の原動力である。モデルの訓練とは、モデルが訓練例にどれだけ当てはまっていないかを測るスカラー値の損失関数を最小化するように、パラメータを調整することを意味する。数百万から数十億個のパラメータがある場合、試行錯誤で各パラメータを探ろうとすれば、更新ごとに数百万回の順伝播が必要になるだろう。その代わりに、**導関数**（derivative）は各パラメータに関する損失の瞬間的な変化率を与える。すべての偏微分を**勾配**（gradient）ベクトルにまとめれば最急上昇の方向がわかり、さらに**連鎖律**（chain rule）を用いれば入れ子になった層全体の勾配を 1 回の逆伝播で組み立てられる。

本節では、すべての数値を手計算で確かめられるほど小さな線形回帰（linear regression）の例を一貫して用いながら、導関数、勾配、連鎖律を展開する。1.5 節では、深層学習フレームワークが任意のプログラムに対してこれらの計算をどのように自動化するかを学ぶ。

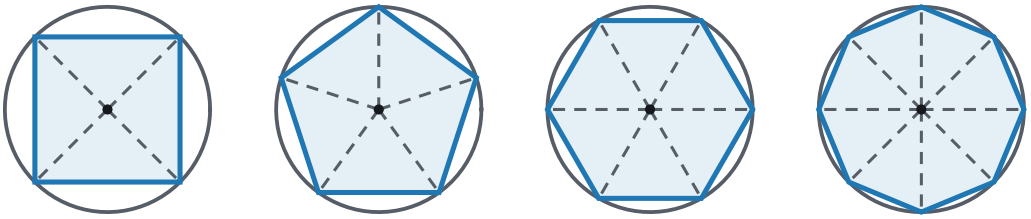


Figure 1.19: 円に内接する辺の数 4、5、6、8 の正多角形。辺の数が増えるにつれて、多角形の面積は円の面積に近づく。

1.4.1 導関数

1.4.1.1 極限としての導関数

単一パラメータのモデルがすべてのデータ例に対して定数値 w を予測するとし、ラベルが 1、1、0 である 3 つの訓練例でこのモデルを評価する。二乗誤差（squared error）の和をとると、次のようになる。

$$L(w) = (w-1)^2 + (w-1)^2 + (w-0)^2 = 3w^2 - 4w + 2. \tag{1.15}$$

$w = 1$ における損失は $L(1) = 1$ である。

```
def loss(w):
    return 3 * w ** 2 - 4 * w + 2
```

$L(w)$ が微小な摂動 $h \neq 0$ に対してどのように変化するかを見るため、差分商（difference quotient）$\frac{L(1+h)-L(1)}{h}$ を、h が 10^{-1} から 10^{-5} まで小さくなるときについて計算してみよう。

```
for h in 10.0 ** -np.arange(1, 6):
    print(f'h = {h:.0e}   ratio = {(loss(1 + h) - loss(1)) / h:.5f}')
```

```
h = 1e-01   ratio = 2.30000
h = 1e-02   ratio = 2.03000
h = 1e-03   ratio = 2.00300
h = 1e-04   ratio = 2.00030
h = 1e-05   ratio = 2.00003
```

$L(1+h) = 1 + 2h + 3h^2$ を展開すると、$\frac{L(1+h)-L(1)}{h} = 2 + 3h \to 2$（$h \to 0$ のとき）となることがわかる。厳密には、$f : \mathbb{R} \to \mathbb{R}$ に対し、関数 f の点 x における**導関数**は次の極限で定義される。

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.16}$$

幾何学的には、差分商は点 $P = (x, f(x))$ と $Q = (x+h, f(x+h))$ を通る**割線**（secant line）の傾きである。$h \to 0$ につれて、この割線は回転し、点 P における傾き $f'(x)$ の**接線**（tangent line）へと近づく（Figure 1.20）。

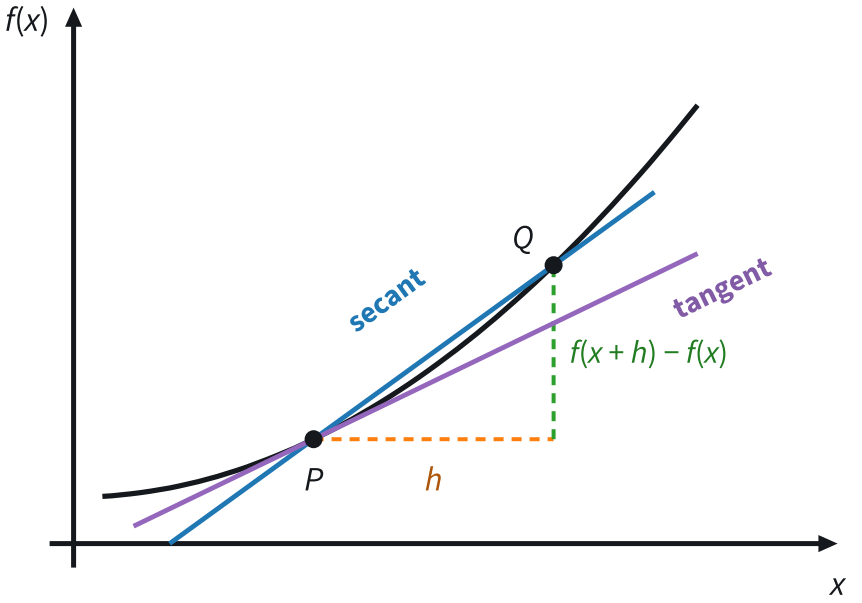


Figure 1.20: 差分商 $(f(x+h)-f(x))/h$ は P と Q を通る割線の傾きである。$h \to 0$ につれて Q は P に向かって移動し、割線は P における接線へと変化して、その傾きは $f'(x)$ となる。

すべての関数がいたるところで微分可能とは限らない。たとえば $|x|$ は $x=0$ に角を持つ（左からの傾きは -1、右からの傾きは $+1$）。さらに悪いことに、離散的な分類の正解率はほとんどいたるところで平坦（$f'(x)=0$）であり、予測が反転する箇所で不連続に跳び上がるため、勾配信号をまったく与えない。分類器を訓練する際に、交差エントロピー（cross-entropy）のような滑らかな代理損失（surrogate loss）を最小化するのはこのためである（Figure 1.21）。

1.4.1.2 微分の公式

$y=f(x)$ が与えられたとき、導関数を $f'(x)=y'=\frac{dy}{dx}=\frac{df}{dx}=\frac{d}{dx}f(x)=Df(x)$ と書く。よく使われる初等関数の導関数は次の通りである。

$$\frac{d}{dx}C=0, \quad \frac{d}{dx}x^n=nx^{n-1}, \quad \frac{d}{dx}e^x=e^x, \quad \frac{d}{dx}\log x=\frac{1}{x} \quad (x>0),$$

また、微分可能な関数 f, g の組み合わせは次の法則に従う。

$$\frac{d}{dx}[Cf(x)]=Cf'(x), \qquad \frac{d}{dx}[f(x)+g(x)]=f'(x)+g'(x),$$

$$\frac{d}{dx}[f(x)g(x)]=f'(x)g(x)+f(x)g'(x), \qquad \frac{d}{dx}\frac{f(x)}{g(x)}=\frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}.$$

Figure 1.22 は、積の微分公式を $f(x)\times g(x)$ の長方形の面積によって幾何学的に示したもので

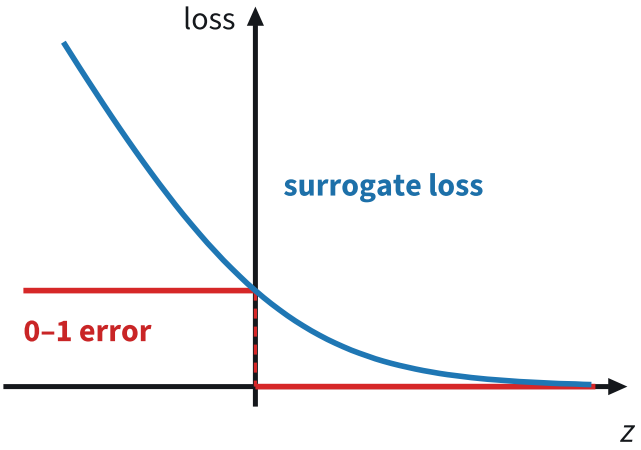


Figure 1.21: 0-1 誤差（誤予測で 1、正解で 0）と滑らかな代理損失を、分類器が正しいときに正となるスコア z に対してプロットしたもの。0-1 誤差は不連続点の両側で平坦であるため、その傾きからは何もわからない。一方、代理損失はいたるところで右下がりの傾きを持ち、パラメータをどちらへ動かすべきかを教えてくれる。

ある（完全な証明は 34.1 節を参照）。これらの公式を $L(w) = 3w^2 - 4w + 2$ に適用すると直ちに $L'(w) = 6w - 4$ が得られ、したがって $L'(1) = 2$ となる。

1.4.1.3 一次線形近似と最小値

$\frac{f(x+h)-f(x)}{h} \approx f'(x)$ の両辺に h を掛けると、次の**一次線形近似**（first-order linear approximation）が得られる。

$$f(x+h) \approx f(x) + f'(x)\,h. \tag{1.17}$$

$L'(1) = 2 > 0$ であるため、w を減らせば損失は小さくなる。導関数と逆向きに $w \leftarrow w - \eta L'(w)$（ステップサイズ $\eta = 0.1$）と進むと、w は 1 から $1 - 0.1(2) = 0.8$ へ移動し、L は 1 から $L(0.8) = 0.72$ へと下がる。これは接線による予測値 $1 + 2(-0.2) = 0.60$ に近い（0.12 の差は無視した二次の曲率 $3h^2$ によるものである）。滑らかな極小値（local minimum）では必ず接線が水平（$f'(x) = 0$）になる。そこで $L'(w) = 6w - 4 = 0$ と置くと、唯一の大域的最小点（global minimizer）として $w^* = 2/3$、すなわち 3 つのラベル $1, 1, 0$ の標本平均（sample mean）が得られる。

```
w = np.linspace(0, 2, 101)
d2l.plot(w, [loss(w), 2 * w - 1], 'w', 'L(w)',
         legend=['L(w)', 'tangent at w = 1'])
```

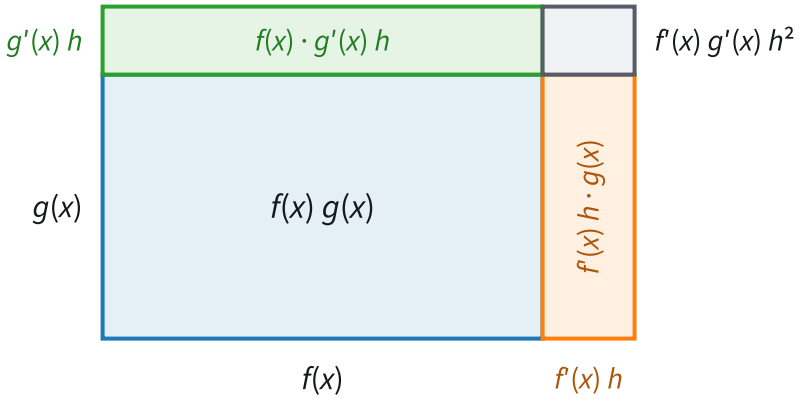
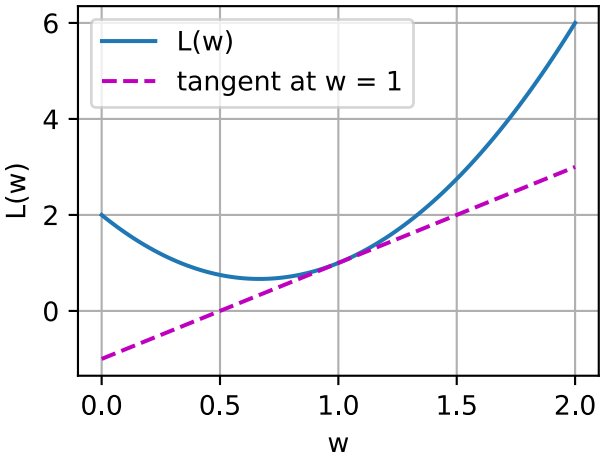


Figure 1.22: 長方形の面積として見た積の微分公式。x が h だけ増えると、辺 $f(x)$ と $g(x)$ はそれぞれ約 $f'(x)h$ と $g'(x)h$ だけ伸び、面積は 2 つの細い帯状の部分と面積 $f'(x)g'(x)h^2$ の角の部分だけ増えるが、この角は小さい h では無視できる。



1.4.2 偏微分と勾配

1.4.2.1 偏微分

次に、先ほどの 3 つのデータ例の問題にスカラー入力 0、1、2（対応するラベルはそれぞれ 1、1、0）を与え、アフィン直線 $w_1 + w_2 \cdot \text{input}$ を当てはめてみよう。

$$L(w_1, w_2) = (w_1 - 1)^2 + (w_1 + w_2 - 1)^2 + (w_1 + 2w_2 - 0)^2. \tag{1.18}$$

多変数関数 $y = f(x_1, \dots, x_n)$ において、x_i に関する**偏微分**（partial derivative）は、他のすべての変数を固定したまま座標 i のみを微小変化させる。

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \tag{1.19}$$

$(w_1, w_2) = (1, 0)$ において、$w_2 = 0$ に固定すると $L(w_1, 0) = 3w_1^2 - 4w_1 + 2$ となるため $\frac{\partial L}{\partial w_1}(1, 0) = 2$ である。一方、$w_1 = 1$ に固定すると $L(1, w_2) = 5w_2^2 + 4w_2 + 1$ となり、その導関数は $10w_2 + 4$ であるため $\frac{\partial L}{\partial w_2}(1, 0) = 4$ となる（Figure 1.23）。

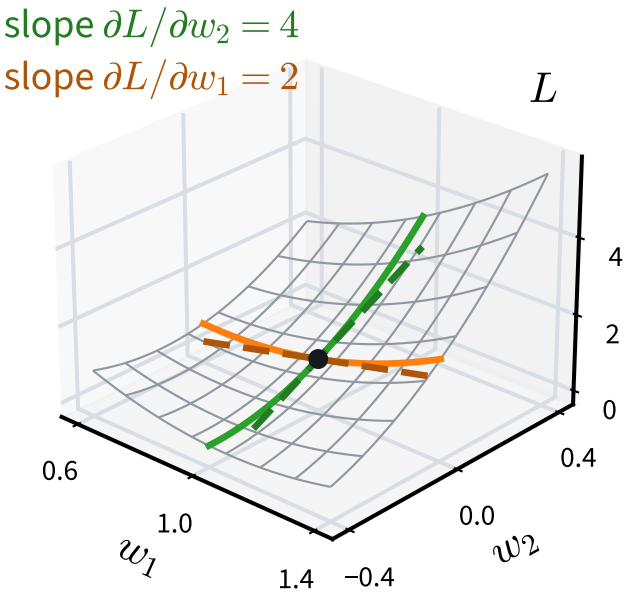


Figure 1.23: 直線モデルの損失 $L(w_1, w_2)$ を $\mathbf{w} = (1, 0)$（黒点）の周辺で示したもの。$w_2 = 0$ に固定すると定数モデルの損失であるオレンジ色の曲線が切り出され、$w_1 = 1$ に固定すると緑色の曲線が切り出される。黒点におけるそれぞれの接線（破線）の傾きは $\partial L/\partial w_1 = 2$ と $\partial L/\partial w_2 = 4$ である。

行列・ベクトル形式（1.3 節）では、各データ例を $(1, \text{input}_i)$ という $\mathbf{X}$ の行として書くと、次のようになる。

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2. \tag{1.20}$$

これらの偏微分を数値的に確かめるには、`line_loss(w)` と有限差分（finite difference）の補助関数 `numerical_gradient(f, w, h=1e-6)` を定義すればよい。この補助関数は、`w` の座標を 1 つずつ $h = 10^{-6}$ だけ `float64` で摂動させる。

```
X = np.array([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows [1, input]
y = np.array([1.0, 1.0, 0.0])

def line_loss(w):
    return np.sum((X @ w - y) ** 2)

def numerical_gradient(f, w, h=1e-6):
```

```
    """Finite differences of f at w, moving one coordinate at a time."""
    grad = np.zeros_like(w)
    for i in range(len(w)):
        step = np.zeros_like(w)
        step[i] = h
        grad[i] = (f(w + step) - f(w)) / h
    return grad
```

`numerical_gradient(line_loss, w)` を $\mathbf{w} = (1, 0)$ で評価すると、解析的に求めた偏微分 $(2, 4)$ が $O(h) \approx 10^{-6}$ の打ち切り誤差（truncation error）の範囲内で再現される。

```
w = np.array([1.0, 0.0])
numerical_gradient(line_loss, w)
```

```
array([2.000003, 4.000005])
```

なお、`numerical_gradient` は n 回余分に f を評価する必要があり（パラメータ数が n の場合）、$n = 10^8$ ともなると非現実的になる。

1.4.2.2 有限差分における数値精度の限界

また、有限差分では h が小さすぎるときにも浮動小数点の桁落ちという問題が生じる。有限精度のもとでは、計算機イプシロンが $\epsilon \approx 2.2 \times 10^{-16}$（`float64`）や $\epsilon \approx 1.2 \times 10^{-7}$（`float32`）であるため、ほぼ等しい数値の差 $f(x + h) - f(x)$ をとると上位の有効桁が失われる**桁落ち**（catastrophic cancellation）が起き、$\sim \epsilon$ 程度の丸め残差が h で割られることになる。したがって、**打ち切り誤差**（$\propto h$）と**丸め誤差**（rounding error、$\propto \epsilon/h$）が競合する。

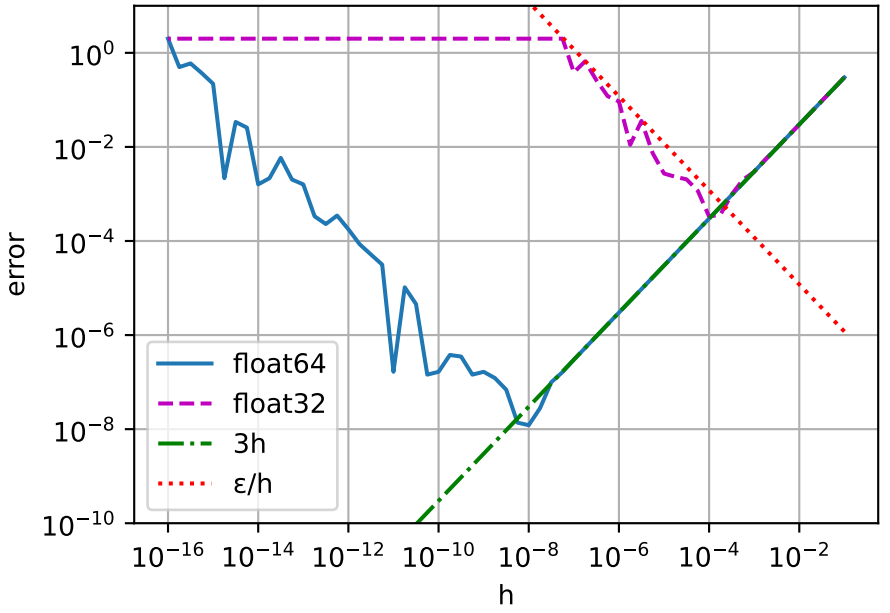
```
def quotient_error(f, x, df, h):
    """Error of (f(x + h) - f(x)) / h as an estimate of f'(x) = df."""
    return abs((f(x + h) - f(x)) / h - df)

h = np.logspace(-1, -16, 61)  # 61 values of h, evenly spaced on a log scale
```

64 ビットの NumPy 演算を、深層学習でデフォルトとして使われる 32 ビット浮動小数点形式（floating-point format）と比較するため、ステップのグリッド `h` を `float32` のフレームワークテンソル `h32` にキャストする。

```
h32 = torch.tensor(h, dtype=torch.float32)
```

ここで、`quotient_error(loss, 1, 2, ...)` を $w = 1$（厳密な導関数の値は $L'(1) = 2$）において全 61 個のステップサイズにわたり `float64`（`h`）と `float32`（`h32`）の両方で評価し、打ち切り誤差の直線 $3h$ と `float32` の丸め誤差の漸近線 ϵ/h を重ねて描画する。

```
eps32 = np.finfo(np.float32).eps
d2l.plot(h, [quotient_error(loss, 1, 2, h), quotient_error(loss, 1, 2, h32),
             3 * h, eps32 / h], 'h', 'error', xscale='log', yscale='log',
         ylim=(1e-10, 1e1), legend=['float64', 'float32', '3h', 'ε/h'],
         figsize=(5, 3.5))
```



$3h \approx \epsilon/h$ を釣り合わせると最適なステップサイズは $h^* \approx \sqrt{\epsilon/3}$ となり、その形式が持つ有効桁数の半分が失われる（最小誤差は $\sim 10^{-8}$ に達するのが $h \approx 10^{-8}$ の `float64` であり、$\sim 10^{-4}$ に達するのが $h \approx 10^{-4}$ の `float32` である）。連鎖律を用いた解析的な微分は、$O(n)$ の計算コストと $\sqrt{\epsilon}$ への精度低下の両方を回避できるため、`float64` による有限差分は単体テストでの勾配チェックの用途に残されている。

1.4.2.3 勾配と最急降下

全 n 個の偏微分（関数 $f : \mathbb{R}^n \to \mathbb{R}$ の各変数に関するもの）を連結すると、**勾配**ベクトル $\nabla_{\mathbf{x}} f(\mathbf{x}) = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})^\top$（$\mathbf{x}$ と同じ形状）が得られる。これにより、Equation 1.17 は同時摂動 $\mathbf{h} \in \mathbb{R}^n$ へと一般化される。

$$f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h}. \tag{1.21}$$

ステップ $\mathbf{h} = t\mathbf{d}$（長さ $t > 0$、単位ベクトル $\mathbf{d}$ は $\|\mathbf{d}\|_2 = 1$）に対して、Equation 1.8 より $f(\mathbf{x}+t\mathbf{d}) - f(\mathbf{x}) \approx t\,\|\nabla f(\mathbf{x})\| \cos\theta$ となる。したがって、$\nabla f(\mathbf{x})$ は**最急上昇**（steepest ascent）の方向（$\theta = 0°$）を指し、$-\nabla f(\mathbf{x})$ は**最急降下**（steepest descent）の方向（$\theta = 180°$）を指す。また、$\nabla f(\mathbf{x})$ は点 $\mathbf{x}$ を通る等位集合（level set）と直交する（$\theta = 90°$、Figure 1.24）。

これが**勾配降下法**（gradient descent）$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$ の動機となる。$\mathbf{w} = (1, 0)$ から学習率（learning rate）$\eta = 0.1$ で 1 ステップ進めると、$(1,0) - 0.1(2,4) = (0.8, -0.4)$ へ移動し、損失は 1.0 から 0.4 に減少する。なぜ Figure 1.24 の最急降下は谷をジグザグに進み、大域的最小値 $\mathbf{w}^* = (7/6, -1/2)$ をまっすぐ目指さないのだろうか。それは、「最急」がユークリッド距離でのステップ長 $\|\mathbf{h}\|_2$ の単位あたりで測られ、各座標の単位に依存するからである。1 つの特徴量の軸を引き伸ばすと等高線が潰れ、$-\nabla L$ が最小点から逸れる方向に傾く。実務において特徴量の標準化（1.2 節）や曲率の前処理（preconditioning、第 9 章）が極めて重要になるのはこのためである。

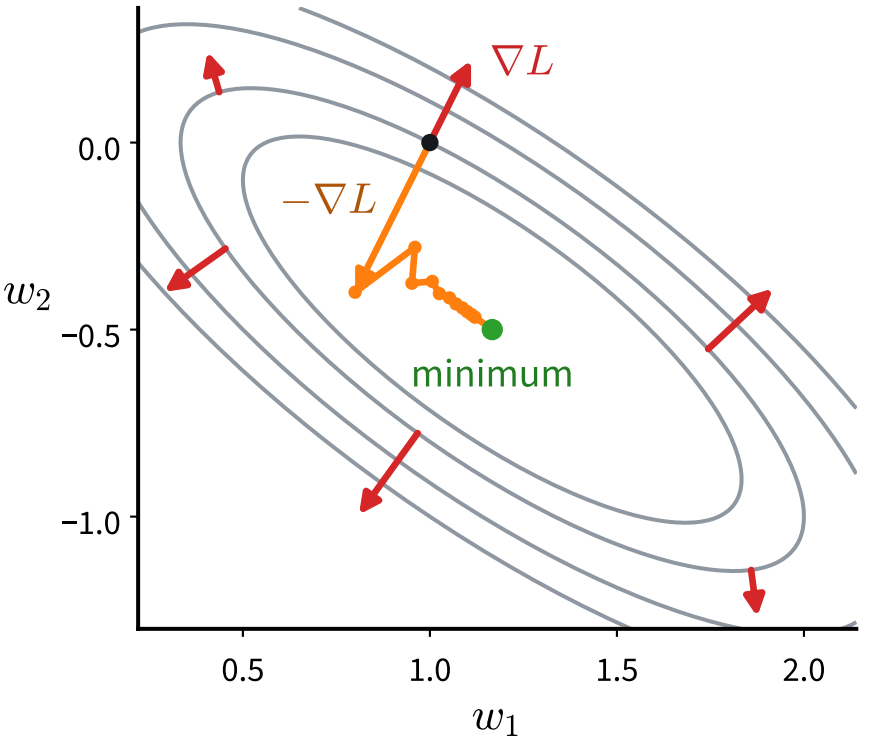


Figure 1.24: 直線モデルの損失 $L(w_1, w_2)$ の等位集合。勾配（赤）はそれぞれの始点を通る等位集合に直交し、上り方向を指す。$\mathbf{w} = (1, 0)$（黒点）から出発すると、$\eta = 0.1$ の勾配降下法（オレンジ）は最小点を直接指していない $-\nabla L$ に沿って進み始め、ジグザグに最小点へと向かう。

1.4.2.4 行列微分の恒等式

ベクトル勾配に関する 3 つの恒等式（34.4 節）を用いると、座標を書き下すことなくベクトル表現を微分できる。

$$\nabla_{\mathbf{x}}\,(\mathbf{a}^{\top}\mathbf{x}) = \mathbf{a}, \qquad \nabla_{\mathbf{x}}\,(\mathbf{x}^{\top}\mathbf{A}\mathbf{x}) = (\mathbf{A}+\mathbf{A}^{\top})\mathbf{x}, \\ \nabla_{\mathbf{x}}\|\mathbf{x}\|^{2} = 2\mathbf{x}.$$

$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w}-\mathbf{y}\|^{2} = \mathbf{w}^{\top}(\mathbf{X}^{\top}\mathbf{X})\mathbf{w} - 2(\mathbf{X}^{\top}\mathbf{y})^{\top}\mathbf{w} + \mathbf{y}^{\top}\mathbf{y}$ を展開してこれらの公式を適用すると、次の式が得られる。

$$\nabla L(\mathbf{w}) = 2\mathbf{X}^{\top}\mathbf{X}\mathbf{w} - 2\mathbf{X}^{\top}\mathbf{y} = 2\mathbf{X}^{\top}(\mathbf{X}\mathbf{w}-\mathbf{y}). \tag{1.22}$$

言葉で言い換えると、勾配の各成分は、$\mathbf{X}$ の対応する特徴量の列と残差（residual）ベクトル $\mathbf{r} = \mathbf{X}\mathbf{w}-\mathbf{y}$ とのドット積（dot product）の 2 倍である。最適解 $\nabla L(\mathbf{w}^{*}) = \mathbf{0}$ においては、予測誤差はすべての特徴量と直交する。

1.4.3 連鎖律

深層ネットワークは、入れ子になった多数の層 $y = f_{L}(f_{L-1}(\cdots f_{1}(\mathbf{x})\cdots))$ を合成したものである。合成関数を展開せずに微分するには**連鎖律**を用いる。

スカラー関数 $z = f(u)$ と $u = g(x)$ では、変化率は $\frac{dz}{dx} = \frac{dz}{du}\frac{du}{dx}$ のように掛け合わされる。より一般に、$z = f(u_{1},\dots,u_{m})$ であり各変数が $u_{j} = g_{j}(x_{1},\dots,x_{n})$ で与えられるとき、変化率は各経路 $x_{i} \to u_{j} \to z$ に沿って掛け合わされ、すべての並列経路にわたって足し合わされる。

$$\frac{\partial z}{\partial x_{i}} = \sum_{j=1}^{m} \frac{\partial z}{\partial u_{j}}\frac{\partial u_{j}}{\partial x_{i}} \quad\Longrightarrow\quad \nabla_{\mathbf{x}} z = \mathbf{J}^{\top}\nabla_{\mathbf{u}} z, \tag{1.23}$$

ここで、$\mathbf{J} \in \mathbb{R}^{m\times n}$ は $\mathbf{u}$ の $\mathbf{x}$ に関する**ヤコビ行列**（Jacobian matrix）であり、その成分は $J_{ji} = \frac{\partial u_{j}}{\partial x_{i}}$ である。アフィン写像 $\mathbf{u} = \mathbf{A}\mathbf{x}-\mathbf{y}$ の場合、ヤコビ行列は単に $\mathbf{J} = \mathbf{A}$ となる。

小さな例を手計算で追ってみよう。$z = u^{2} + v$（ただし $u = x_{1}x_{2}$、$v = x_{2}^{2}$）とし、$(x_{1},x_{2}) = (1,2)$ で評価する。入力 x_{2} から z へは、u を通る経路と v を通る経路の 2 つの経路がある（Figure 1.25）。それぞれの経路は局所的な導関数の積をもたらし、2 つの寄与を足し合わせることで $\partial z/\partial x_{2} = 4\cdot 1 + 1\cdot 4 = 8$ が得られる。

直線モデルの損失 $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w}-\mathbf{y}\|^{2}$ を 2 つの段階、すなわち（1）内側の残差 $\mathbf{r} = \mathbf{X}\mathbf{w}-\mathbf{y}$（ヤコビ行列は $\mathbf{J} = \mathbf{X}$）と（2）外側の二乗ノルム $L = \|\mathbf{r}\|^{2}$（勾配は $\nabla_{\mathbf{r}}L = 2\mathbf{r}$）に分解すると、多変数の連鎖律によって $\nabla_{\mathbf{w}}L = \mathbf{X}^{\top}(2\mathbf{r})$ が 2 回のパスで復元される。

```
r = X @ w - y    # forward: errors, from which the loss is r @ r
grad_r = 2 * r   # backward: gradient of the loss with respect to r
X.T @ grad_r     # backward: gradient with respect to w
```

```
array([2., 4.])
```

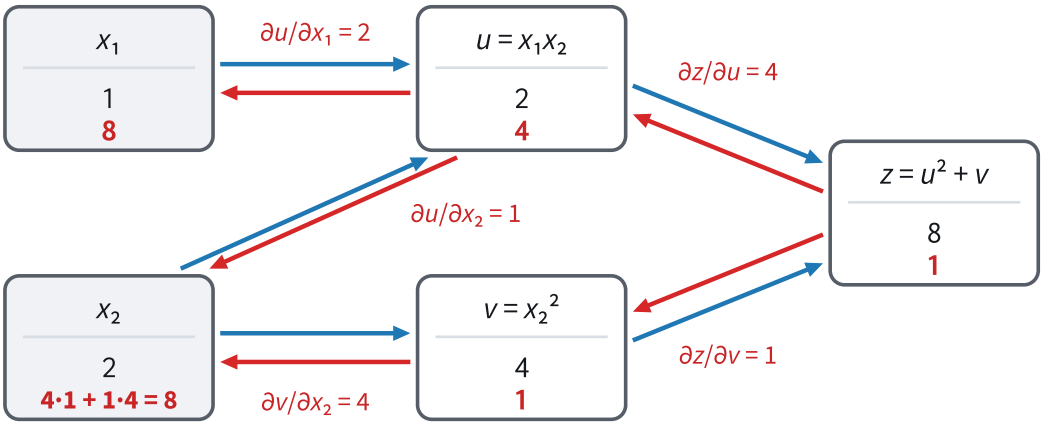


Figure 1.25: $z = u^2 + v$（$u = x_1 x_2$、$v = x_2^2$）の $(x_1, x_2) = (1, 2)$ における連鎖律。各ボックスは変数の値（黒）と z のその変数に関する導関数（赤）を示し、各赤い矢印は局所的な導関数を表す。導関数は経路に沿って掛け合わされ、x_2 のように 2 つの経路が合流する箇所で足し合わされる。

順伝播（forward pass）では $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$ と $L = \mathbf{r}^\top \mathbf{r}$ を計算する。一方、**逆伝播**（backward pass）、すなわち**誤差逆伝播法**（backpropagation）では $2\mathbf{r}$ に $\mathbf{X}^\top$ を掛け、おおよそ 1 回の順伝播評価と同じコストですべてのパラメータに関する厳密な勾配を計算する。1.5 節では、この機械的な逆伝播をフレームワークの自動微分（automatic differentiation）エンジンに委ねる。

1.4.4 まとめ

導関数 $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$ は接線の傾きと局所線形近似 $f(x + h) \approx f(x) + f'(x)h$ を与える。$f : \mathbb{R}^n \to \mathbb{R}$ に対しては、勾配 $\nabla_{\mathbf{x}} f(\mathbf{x})$ がすべての偏微分をまとめたものとなり、$f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h}$ を満たし、等位集合に直交して最急上昇の方向を指すことから、勾配降下法 $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$ の根拠となる。$O(n)$ 回の評価コストがかかり桁落ちによって精度が失われる（$\sim \sqrt{\epsilon}$）有限差分とは異なり、多変数の連鎖律 $\nabla_{\mathbf{x}} z = \mathbf{J}^\top \nabla_{\mathbf{u}} z$ は、1 回の逆伝播（**誤差逆伝播法**）によって損失からすべてのパラメータへ厳密な勾配を逆方向に伝播させる。

1.4.5 演習問題

1. **底と指数の両方が変化するときの x^x の微分**。$x > 0$ に対し、関数 $f(x) = x^x$（x が底と指数の両方に現れる）を考える。
    1. べき乗の微分公式のみを適用すること（指数を定数とみなして $x \cdot x^{x-1} = x^x$ を得ること）と、指数関数の微分公式のみを適用すること（底を定数とみなして $x^x \log x$ を得ること）がいずれも誤りである理由を説明せよ。
    2. $x^x = e^{x \log x}$ と書き換え、連鎖律と積の微分公式を適用して $f'(x)$ を求めよ。真の導関数が小問 1 の 2 つの単純な答えの**和**になっていることに注目し、なぜそうなるのかを、$g(u, v) = u^v$ に対して $u = x, v = x$ に沿って 2 変数の連鎖律を適用することで説明せよ。
2. **なぜ二乗誤差からは平均が得られ、絶対誤差からは中央値が得られるのか**。単一パラメータのモデルが定数値 $w \in \mathbb{R}$ を、n 個の訓練例（スカラーラベルは $y_1, \ldots, y_n$）に対して予測するとする。

   1. 二乗誤差の和 $L_2(w) = \sum_{i=1}^{n} (w - y_i)^2$ を w に関して微分し、導関数をゼロと置いて、唯一の最小点が標本**平均** $w^{*} = \frac{1}{n}\sum_{i=1}^{n} y_i$ であることを証明せよ。
   2. 次に、絶対誤差の和 $L_1(w) = \sum_{i=1}^{n} |w - y_i|$ を考える。任意の点 w（どのラベル y_i とも等しくない点）において、導関数 $L_1'(w)$ は w より真に小さいラベルの個数から w より真に大きいラベルの個数を引いた値に等しいことを示せ。w がどの値をとるとき $L_1'(w)$ は存在しないか。以上から、$L_1(w)$ は $\{y_1, \dots, y_n\}$ の任意の標本**中央値**によって最小化されることを導け。
   3. 4 つのラベル $\{1, 1, 0, 10\}$ に対し、L_2 の最小点（平均）と L_1 の最小点（中央値）を計算せよ。外れ値のラベル 10 が破損して 100 になった場合、それぞれの最小点はどうなるか。

3. **ユークリッドノルムの勾配**。勾配 $\nabla_{\mathbf{x}} f(\mathbf{x})$ を、二乗していないユークリッドノルム $f(\mathbf{x}) = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$ に対して任意の非ゼロベクトル $\mathbf{x} \neq \mathbf{0}$ で計算し、勾配が存在するすべての点で $\|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 = 1$ となることを確かめよ。原点 $\mathbf{x} = \mathbf{0}$ では勾配はどうなるか。また、それはなぜか。

4. **勾配降下法のステップサイズ安定性**。等方的な二次関数 $f(\mathbf{x}) = \|\mathbf{x}\|_2^2$ を、勾配降下法の更新式 $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$（学習率 $\eta > 0$）によって最小化することを考える。

   1. $\mathbf{x}_0 = (1, 1)^{\top}$ から出発し、1 ステップ後の $\mathbf{x}_1$ と新しい目的関数値 $f(\mathbf{x}_1)$ を、3 通りの学習率 $\eta = 0.1$、$\eta = 1.0$、$\eta = 2.0$ について計算せよ。
   2. $\mathbf{x}_{k+1}$ を $\mathbf{x}_k$ と η を用いた閉じた式で表せ。学習率 $\eta > 0$ がどの範囲にあるとき、勾配降下法の 1 ステップによって $f(\mathbf{x})$ は任意の非ゼロの初期点 $\mathbf{x}_k \neq \mathbf{0}$ から真に減少するか。また、η がどの特別な値のとき 1 ステップで大域的最小値 $\mathbf{x}^{*} = \mathbf{0}$ へ直接到達するか。さらに、$\eta > 1$ のときは何が起きるか。
   3. 勾配降下法のステップでパラメータベクトルが変化しない停留点 $\mathbf{x}$（$\mathbf{x}_{k+1} = \mathbf{x}_k$ を満たす点）を、$f(\mathbf{x}) = \|\mathbf{x}\|_2^2$ の場合と一般の微分可能な目的関数 f の場合の両方について特徴づけよ。

5. [code] **中心差分と前進差分**。●本節で解析した片側の**前進差分** $\frac{f(x+h) - f(x)}{h}$ は $O(h)$ の打ち切り誤差を持つ。対称な**中心差分**

$$\frac{f(x + h) - f(x - h)}{2h} \tag{1.24}$$

   は関数 f を $x + h$ と $x - h$ で評価することで、$O(h)$ となる二次の項が相殺されるようにしている。中心差分版の `quotient_error` を実装し、対数グリッド上の h の値にわたって `float64` で評価せよ。

   1. 二次の損失 $L(w) = 3w^2 - 4w + 2$ の $w = 1$ において、前進差分と中心差分の誤差曲線を同じ両対数軸上にプロットせよ。二次関数では、なぜ中心差分の誤差が計算機イプシロン（$\sim 10^{-16}$）まで、大きな $h = 10^{-1}$ であっても一気に下がるのか。
   2. 次に、二次関数ではない $f(x) = e^{x}$ の $x = 0$（$f'(0) = 1$）において同じ比較を繰り返せ。大きい h における中心差分曲線の打ち切り誤差領域の傾きを、傾き 1 の前進差分曲線と比較し、各手法における最適なステップサイズ h^{*} と達成可能な最小誤差を求め、テイラー展開 $e^{\pm h} = 1 \pm h + \frac{h^2}{2} \pm \frac{h^3}{6} + \cdots$ を用いてその両方を説明せよ。

6. [code] **`float64` と `float32` における勾配チェック**。●有限差分は訓練には遅すぎるものの、中心有限差分は手計算で導出した勾配やカスタムカーネルの勾配を単体テストする際の定番手法である。

1. 関数 `central_gradient(f, x, h=1e-5)` を実装し、中心差分によって勾配 $\nabla f(\mathbf{x})$（$f : \mathbb{R}^n \to \mathbb{R}$）を推定せよ。これを `float64` で用い、二次形式の勾配の恒等式 $\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ をランダムな**非対称** 5×5 行列 $\mathbf{A}$ とランダムなベクトル $\mathbf{x} \in \mathbb{R}^5$ に対してテストし、5 つの座標にわたる最大相対誤差を求めよ。
2. 次に、**バグのある**式 $2\mathbf{A}\mathbf{x}$（$\mathbf{A}$ が対称でないことを忘れたときによくある誤り）に対して同じチェックを実行せよ。このとき相対誤差はどのくらい大きくなるか。
3. 最後に、正しい式 $(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ のチェックを、32 ビット `float32` のフレームワークテンソルを用いて $h \in \{10^{-1}, 10^{-2}, \dots, 10^{-6}\}$ にわたって繰り返せ。どの h が `float32` で最小の相対誤差を達成するか。また、それが前進差分の最適値 $h^* \approx 10^{-4}$ と異なるのはなぜか。さらに、`float32` において正しい勾配実装とバグのある実装を明確に切り分けられる相対誤差のしきい値はどの程度か。

1.5 自動微分

1.4 節では、2 つのパラメータをもつ直線モデルの勾配 $\nabla_{\mathbf{w}} L = \mathbf{X}^\top(2\mathbf{r}) = (2, 4)$ を手計算で容易に導出できたが、数十の層と数十億個のパラメータにわたって手計算で導出するのは絶望的である。現代の深層学習フレームワークは、この計算を**自動微分**（automatic differentiation、**autograd**）によって自動化する。プログラムがスカラーの損失を評価する際、フレームワークは基本演算をトレースして**計算グラフ**（computational graph）を構築し、そのグラフを逆向きにたどることで（**誤差逆伝播法**、backpropagation）、連鎖律（chain rule）を機械的に適用する。驚くべきことに、モデルのパラメータが 2 個であっても 10 億個であっても、1 回の逆方向スイープだけで**すべて**のパラメータに関する厳密な勾配を、順伝播（forward pass）のコストのわずかな定数倍で計算できる。

```
import torch
```

1.5.1 勾配の計算

1.5.1.1 順伝播

まず、1.4 節で扱った直線モデル $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ を $\mathbf{w} = (1, 0)$ においてフレームワークのテンソルで定義し、順伝播のみを記述する。

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # [1, input]
y = torch.tensor([1.0, 1.0, 0.0])
w = torch.tensor([1.0, 0.0])

def loss(w):
    return ((X @ w - y) ** 2).sum()
```

演算の記録と中間活性化値の保存はメモリを消費するため、フレームワークは勾配計算の対象として明示的に指定されたパラメータに依存する演算のみを追跡する。

`w.requires_grad_(True)` を呼び出すと、`w` がインプレースで指定され、PyTorch は `w` を含む演算を追跡して得られた勾配を `w.grad` に格納する。

```
w.requires_grad_(True)
```

```
tensor([1., 0.], requires_grad=True)
```

次に `L = loss(w)` を評価すると、PyTorch は `w` から `L` までの演算の系列を自動的に記録する。

```
L = loss(w)
L
```

```
tensor(1., grad_fn=<SumBackward0>)
```

`grad_fn=<SumBackward0>` が `L` に付与されていることに注目しよう。これは計算グラフの最終演算を記録したものである。

記録されたトレースは Figure 1.26 に示す**計算グラフ**を形成し、損失を $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$、$\mathbf{r} = \hat{\mathbf{y}} - \mathbf{y}$、$\mathbf{s} = \mathbf{r} \odot \mathbf{r}$、$L = \sum_i s_i$ という 4 つの基本演算に分解する。

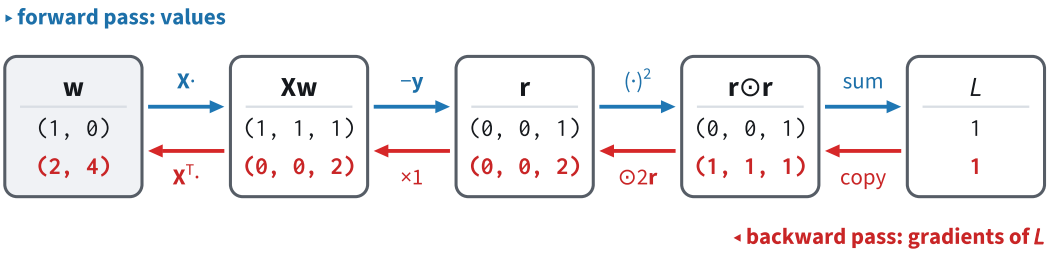


Figure 1.26: $L = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ の $\mathbf{w} = (1, 0)$ における計算グラフ。$\mathbf{w}$ およびそこから計算される各値に 1 つずつノードがあり、定数 $\mathbf{X}$ と $\mathbf{y}$ は省略されている。また、$\mathbf{r} \odot \mathbf{r}$ は要素ごとの二乗を表す。各ノードにはその値（黒）と、それに関する L の勾配（赤）を示す。順伝播（青）は左から右へ進み、逆伝播（赤）は 1 を L に与えて右から左へ各演算の局所規則を適用する。

1.5.1.2 逆伝播

逆伝播（backward pass）を実行すると、出力における $\partial L/\partial L = 1$ から各ノードの局所微分規則を通じて $\mathbf{w}$ へと感度が逆向きに伝播する。

$$\nabla_{\mathbf{s}} L = \mathbf{1} \to \nabla_{\mathbf{r}} L = 2\mathbf{r} \to \nabla_{\hat{\mathbf{y}}} L = 2\mathbf{r} \to \nabla_{\mathbf{w}} L = \mathbf{X}^\top (2\mathbf{r}), \tag{1.25}$$

これにより、先ほどと同じ $(2, 4)$ が得られる。

`L.backward()` は逆伝播を実行し、勾配を `w.grad` に格納する。

```
L.backward()
w.grad
```

```
tensor([2., 4.])
```

逆方向の局所規則では順伝播の中間活性化値（intermediate activation、ここでは $\mathbf{r}$ や $\mathbf{X}$ など）が必要になるため、訓練は推論（inference）よりも大幅に多くのメモリを消費する（Figure 1.27）。

1.5.1.3 逆伝播の繰り返し

同じパラメータベクトル $\mathbf{w}$ で勾配をもう一度評価するとどうなるだろうか。

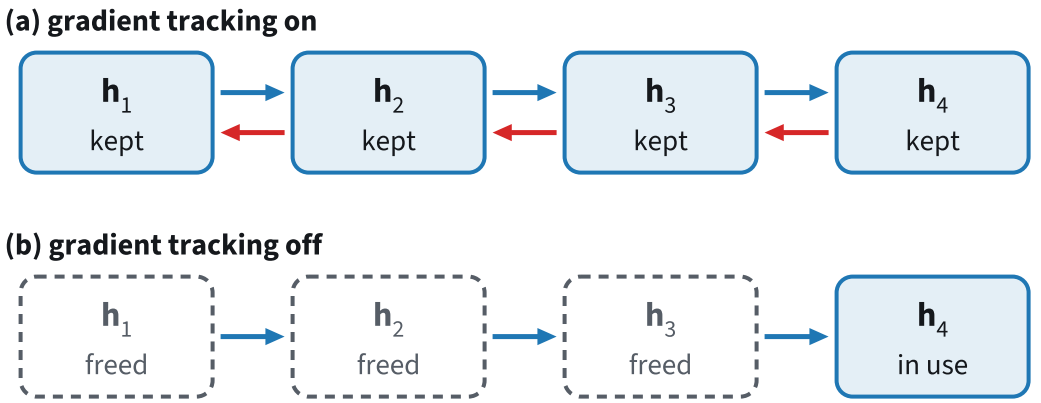


Figure 1.27: 出力 $\mathbf{h}_1, \ldots, \mathbf{h}_4$ をもつ 4 つの層を通る順伝播でのメモリ使用状況。(a) 勾配を追跡する場合、逆伝播（赤）で必要になるため、すべての出力が保持される。(b) 推論時のように勾配を追跡しない場合、次の層が使い終えた時点で出力を解放できる。

```
L = loss(w)
L.backward()
w.grad  # the new gradient was added to the old one
```

```
tensor([4., 8.])
```

PyTorch では `w.grad` が `(4, 8)` になる。これは PyTorch がデフォルトで新しい勾配を `w.grad` に**加算**するためであり、マイクロバッチや複数の損失項にわたって勾配を合計する際に便利である。標準的な訓練ループでは、次のステップの前に必ず `w.grad.zero_()` でバッファをクリアする。

1.5.1.4 ベクトル値関数

ある演算が m 個の出力からなるベクトル $\mathbf{u} \in \mathbb{R}^m$ を n 個の入力 $\mathbf{x} \in \mathbb{R}^n$ から生成し、その $m \times n$ のヤコビ行列（Jacobian）を $\mathbf{J}$ とする。このとき、重みベクトル $\mathbf{v} \in \mathbb{R}^m$ をシードとして逆伝播を実行すると、転置を用いた**ベクトル・ヤコビ行列積**（vector–Jacobian product）$\mathbf{J}^\top \mathbf{v} = \nabla_{\mathbf{x}}(\mathbf{v}^\top \mathbf{u})$ が、$\mathbf{J}$ を明示的に構築することなく計算される。$\mathbf{v} = \mathbf{1}$ を例ごとの二乗誤差 (r_1^2, r_2^2, r_3^2) に渡すと、その和が微分される。

非スカラーのテンソルに対して `backward()` を呼び出すには、`gradient=torch.ones(3)` によって $\mathbf{v}$ を明示的に渡す必要がある（`losses.sum().backward()` と等価である）。

```
w.grad.zero_()
losses = (X @ w - y) ** 2  # one loss per example
losses.backward(gradient=torch.ones(3))  # = losses.sum().backward()
w.grad
```

```
tensor([2., 4.])
```

1.5.2 グラフの制御

1.5.2.1 値の切り離し

テンソルの**切り離し**（detaching）とは、数値は同じだが計算グラフ上の祖先とのつながりをもたないビューまたはコピーを作る操作であり、誤差逆伝播法ではそのテンソルが定数として扱われる。$u = x^2$ と $z = ux$ を $\mathbf{x} = (0, 1, 2, 3)$ 上で考えよう（Figure 1.28）。u がグラフに接続されたままであれば、勾配は両方の経路を流れて $\partial z/\partial x = 3x^2$ となる。一方、u を切り離して c とすると、$z = cx$ の微分では c が定数として扱われ、$\partial z/\partial x = c = x^2$ が得られる（15.4 節の深層 Q 学習のように、ネットワークが自身のブートストラップターゲットを計算する場合には不可欠である）。

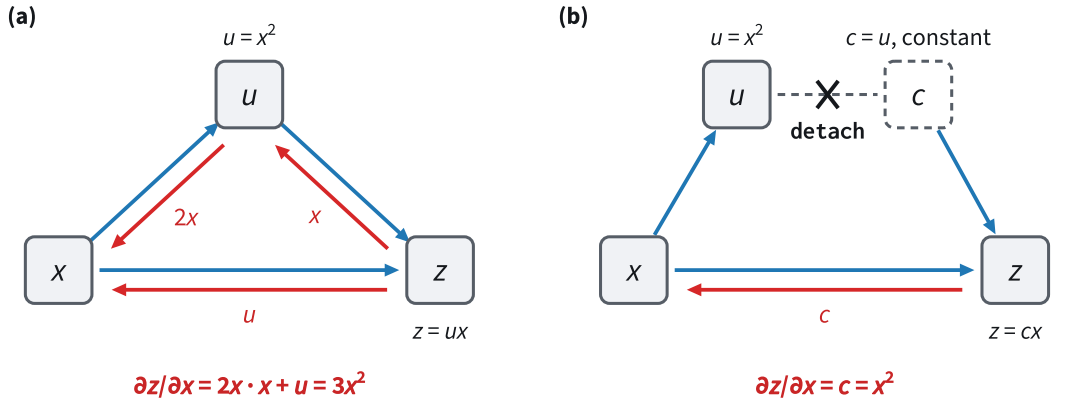


Figure 1.28: 積 $z = ux$（ただし $u = x^2$）における、(a) 接続したままの場合と (b) 切り離した場合。青い矢印は順伝播の演算を表し、赤い矢印は逆伝播とそこで乗じられる局所導関数を表す。u を切り離すと、同じ値をもつ定数 c が作られ、x とのつながりは断たれる。

`u.detach()` は、`u` と値を共有しつつ計算履歴を取り除いたテンソルを返す。

```
x = torch.arange(4.0, requires_grad=True)
u = x * x
c = u.detach()  # values of u, no history
z = c * x
z.sum().backward()
x.grad
```

```
tensor([0., 1., 4., 9.])
```

対照的に、`z = u * x` を `u` の切り離しなしに直接計算すると、誤差逆伝播法は $z = x^3$ の両方の因子を通って伝播し、$\partial z/\partial x = 3x^2 = (0, 3, 12, 27)$ を返す。

```
x.grad.zero_()
z = u * x
z.sum().backward()
x.grad
```

```
tensor([ 0.,  3., 12., 27.])
```

1.5.2.2 勾配追跡の無効化

検証、推論、あるいはパラメータ更新 $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$ の際には、それらの演算が損失の計算グラフに記録されないよう勾配追跡を無効化する。

`with torch.no_grad():` は追跡を無効化する（`requires_grad=True` をもつリーフテンソルをインプレースで変更する際に必要である）。

```
with torch.no_grad():  # nothing in this block is recorded
    w -= 0.1 * w.grad
    L = loss(w)
w, L
```

```
(tensor([ 0.8000, -0.4000], requires_grad=True), tensor(0.4000))
```

1.4 節と同様、$\eta = 0.1$ による 1 ステップで $\mathbf{w}$ は $(1, 0)$ から $(0.8, -0.4)$ へ移動し、損失は 1 から 0.4 へと下がる。

1.5.3 プログラムの微分

1.5.3.1 制御フロー

自動微分は与えられた入力に対して**実際に実行された**基本演算の系列を記録するため、データに依存する `while` ループや `if` と `else` の分岐を含む Python 関数もそのまま微分できる。

```
def f(a):
    b = a * 2
    while torch.linalg.vector_norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

任意の非ゼロのスカラー a に対して（$a = 0$ では `while` ループが終了しない）、`f(a)` は 2（および場合によっては 100）を掛ける定まった回数の乗算を実行するため、`f(a) = k * a` は局所的な傾き $k = f(a)/a$ をもつ区分線形（piecewise linear）関数となる（Figure 1.29）。具体的には、$a = 0.3$ のときは 12 回倍増されて $0.3 \times 2^{12} = 1228.8 \geq 1000$ に達し、正の分岐を通る（$k = 2^{12} = 4096$）。一方、$a = -0.75$ のときは 11 回倍増されて -1536 となり、`else` 分岐を通る（$k = 100 \times 2^{11} = 204{,}800$）。自動微分はこれら両方の傾きを正しく求める。

```
for value in (0.3, -0.75):
    a = torch.tensor(value, requires_grad=True)
    fa = f(a)
    fa.backward()
    print(f'a = {value:5.2f}   f(a) / a = {fa.item() / value:6.0f}   '
          f'gradient = {a.grad.item():6.0f}')
```

```
a =  0.30   f(a) / a =   4096   gradient =   4096
a = -0.75   f(a) / a = 204800   gradient = 204800
```

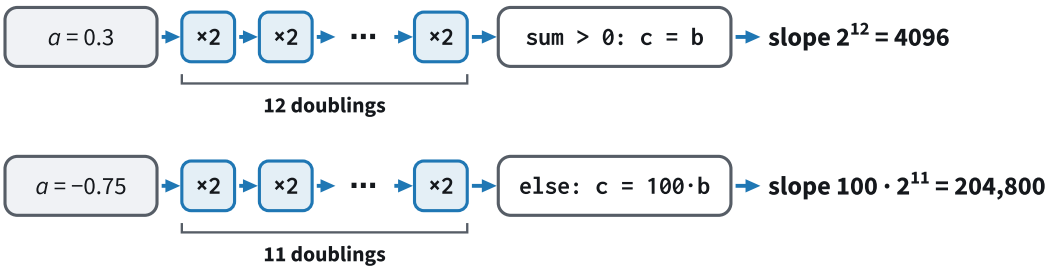


Figure 1.29: 2 つの入力に対して `f` が記録した計算グラフ。$a = 0.3$ では、`f` は a を 12 回倍増させて最初の分岐を実行する。$a = -0.75$ では、a を 11 回倍増させて 2 番目の分岐を実行する。いずれのグラフも a に関して線形であり、それぞれ固有の傾きをもつ。

なお、自動微分が微分するのは抽象的な数学的関数ではなく、**実行されたトレース**であることに注意しよう。ループ回数が切り替わる閾値では $f(a)$ は不連続になるが、自動微分は実行された側の分岐の傾きをそのまま返す。

1.5.4 フォワードモードとリバースモード

写像 $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ とその $m \times n$ ヤコビ行列 $\mathbf{J}$ に対して、連鎖律は順方向と逆方向のどちらの向きにもたどることができる（Figure 1.30、詳細は Wengert (1964); Linnainmaa (1970) および 34.5 節を参照）。

- **リバースモード（ベクトル・ヤコビ行列積、VJP）：**出力に $\mathbf{v} \in \mathbb{R}^m$ をシードとして与える逆方向スイープは $\mathbf{J}^\top \mathbf{v}$ を計算する。j^{th} の標準基底ベクトル $\mathbf{v} = \mathbf{e}_j$ をシードとすれば、第 j 行を $\mathbf{J}$ から取り出せる（ヤコビ行列全体を得るには m 回のスイープを要する）。
- **フォワードモード（ヤコビ行列・ベクトル積、JVP）：**入力に接ベクトル $\mathbf{d} \in \mathbb{R}^n$ をシードとして与える順方向スイープは、値とともに微分値を順方向に伝播させて $\mathbf{Jd}$ を計算する。$\mathbf{d} = \mathbf{e}_i$ をシードとすれば、第 i 列を $\mathbf{J}$ から取り出せる（ヤコビ行列全体を得るには n 回のスイープを要し、メモリオーバーヘッドは定数の $2\times$ で済む）。

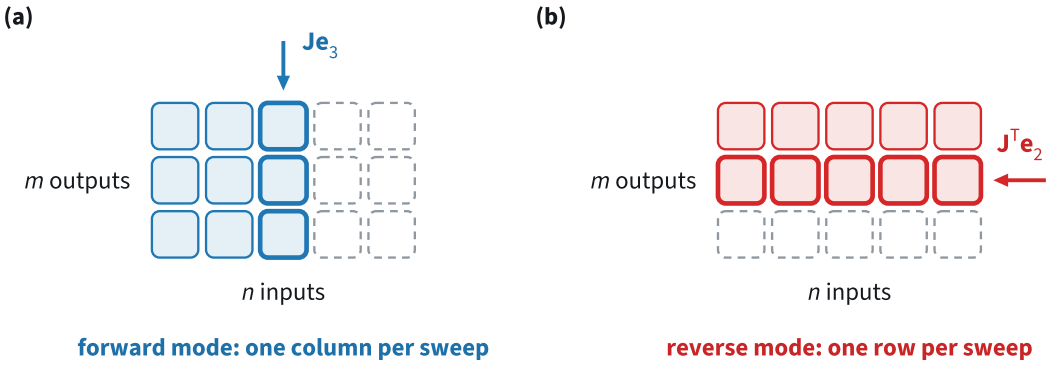


Figure 1.30: $m \times n$ ヤコビ行列 $\mathbf{J}$（n 入力 m 出力の関数のもの）を埋める 2 つの方法。(a) フォワードモード（計 n 回のスイープ）、(b) リバースモード（計 m 回のスイープ）。

PyTorch は `torch.func.vjp`、`torch.func.jvp`、`torch.func.jacrev`、`torch.func.jacfwd` を提供している。

スカラーの損失関数は出力が $m = 1$ 個、パラメータが n 個であるため、そのヤコビ行列は 1 行のみの $\nabla_{\mathbf{w}} L^{\top}$ となる。リバースモード（**誤差逆伝播法**）は勾配の全 n 成分を **1 回のスイープ**で計算できるのに対し、フォワードモードでは n 回のスイープが必要になる。深層ネットワークの訓練でリバースモードが普遍的に使われているのはこのためである。

1.5.5 まとめ

自動微分は、順伝播の基本演算をトレースして計算グラフを構築し、局所微分規則を逆向きに適用することで厳密な勾配を計算する。これにより、直線モデルでも手書きの微分コードなしに $\nabla_{\mathbf{w}} L = (2, 4)$ を再現できた。中間値を切り離すと計算履歴が断たれ、誤差逆伝播法ではその値が定数として扱われる。一方、勾配追跡を無効化すれば、パラメータ更新や推論時の評価が計算グラフに記録されず、順伝播の活性化値のキャッシュも回避できる。計算グラフは与えられた入力に対して実際に実行された演算を記録するため、自動微分は Python のループや条件分岐もシームレスに微分できる。最後に、フォワードモードの 1 回のスイープはヤコビ行列・ベクトル積 **Jd**（基底ベクトルのシード 1 つにつき 1 列）を計算するのに対し、リバースモードの 1 回のスイープはベクトル・ヤコビ行列積 $\mathbf{J}^{\top}\mathbf{v}$（基底ベクトルのシード 1 つにつき 1 行）を計算する。したがって、スカラーの損失（$m = 1$）に対しては 1 回の逆方向スイープだけで n 個すべてのパラメータの勾配が得られる。

1.5.6 演習問題

1. [code] **中間値の切り離し**。`u = x * x` および `c = u.detach()` とし、`z = c * c`、`z = u * c`、`z = u * u` を考える。各 `z` について、`u` と `c` を再計算し、`x.grad` をリセットしたうえで、`backward` を `z` の要素の和に対して呼び出せ。コードを実行する前に、得られる `x.grad` を含めて各呼び出しの結果を予測せよ。予測を確かめ、`z = u * c` と `z = u * u` に対する勾配の比を説明せよ。

2. [code] **手動でのパラメータ更新**。本節の勾配ステップを `w = w - 0.1 * w.grad` として `torch.no_grad()` の外側で実行し、`w.requires_grad`、`w.grad_fn`、`w.is_leaf`、`w.grad` を調べよ。この更新が計算グラフに何をもたらしたかを説明し、2 回目の順伝播と逆伝播を実行すると `w.grad` がどうなるかを予測して確かめよ。

3. [code] **スカラー入力からベクトル入力へ**。スカラー入力 `a`（1.5.3.1 節で定義した関数 `f` の入力）を長さ 5 のランダムなベクトルに置き換え、`fa = f(a)` の要素の和を `a` に関して微分せよ。

    1. `f(a)` を `a` を用いて表し、`f` の `a` におけるヤコビ行列、および `fa` の要素の和の勾配を求めよ。
    2. 自動微分でこの勾配を計算し、要素ごとに `fa / a` と一致するかどうかを確かめよ。
    3. 入力 `a` のうち `f` が不連続に跳ぶ点の集合を記述せよ。この集合のうち、スカラー入力の場合には対応するものがない部分はどれか。

4. [code] **プログラムと関数**。自動微分が微分するのはプログラムが実行する演算であり、プログラムが計算する数学的関数そのものではない。

    1. 関数 $|x|$ と $\sqrt{x^2}$ は等しい。フレームワークの絶対値と平方根を用いて、$x = 0$ におけるそれぞれの微分値を自動微分で計算せよ。得られた両方の結果を、関与する演算の局所導関数から説明せよ。どちらの関数も 0 で導関数をもたないが、訓練においてより大きな悪影響を及ぼすのはどちらの結果か。また、それはなぜか。
    2. 出力として x^2 をすべての入力 x に対して返すが、$x = 0$ における自動微分の結果が 1（0 ではなく）になるプログラムを書け。自動微分がその違いを検出できない理由を説明せよ。

5. **二階導関数のコスト。**●1.5.4 節の回数評価の議論によれば、n 個のパラメータをもつスカラー損失の勾配のコストは逆方向スイープ 1 回分である。同じ議論を用いて、ヘッセ行列全体（二階導関数からなる $n \times n$ 行列）のコストと、1 回のヘッセ行列・ベクトル積のコストを関数評価回数を単位として見積もれ。また、この 2 つのコストの違いを説明せよ。

1.6 確率と統計

機械学習の本質は、不確実性のもとでの推論にある。患者が心臓発作を起こす可能性を予測すること、センサの測定値が異常かどうかを判断すること、あるいは正解率（accuracy）があるミニバッチの 0.90（テスト画像 230/256 枚）から次のミニバッチで 0.87 に変化したとき、それが実質的な変化なのかランダムなサンプリングによる揺らぎなのかを見極めることなど、いずれもその例である。

確率論における最も明快な思考実験は、やはりコイン投げである。コインを 100 回投げて表が 56 回出たとしたら、このコインには偏りがあると疑うべきだろうか。公平なコインであっても、表がちょうど 50 回出ることはまれであるため、表が 6 回多いというずれが偶然よく生じる範囲を超えているかどうかを問わなければならない。**確率**は既知のモデル（公平なコイン）から順方向に推論して結果の生じる見込みを導くのに対し、**統計**は観測データ（100 回投げて表が 56 回）から逆方向に推論して背後にある過程の性質を推測する。

本節では、コイン投げをシミュレーションしてサンプリングによる揺らぎがどのように現れるかを観察し、標本空間、事象、確率変数、同時分布、独立性を導入したうえで、期待値と分散から**標準誤差**（standard error）$\sigma/\sqrt{n}$ を導出して先ほどのコインの疑問に決着をつける。続いて、2 段階の医療診断検査を題材に、条件付き確率とベイズの定理を展開する。

```
import random
from torch.distributions.multinomial import Multinomial
```

1.6.1 サンプリング

1.6.1.1 コイン投げのシミュレーション

公平なコインでは、表と裏の対称性から $P(\text{heads}) = 0.5$ となる。確率過程（random process）から実現値を引き出す操作を**サンプリング**と呼ぶ。以下では 100 回のコイン投げをシミュレーションする。まず区間 $[0, 1)$ からの一様乱数が 0.5 を超えるかどうかを判定する方法を用い、次にフレームワークの**多項**（multinomial）サンプラを $n = 100$、確率 $(0.5, 0.5)$ として用いる。

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(num_tosses)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails:  [45, 55]
```

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([48., 52.])
```

1.6.1.2 頻度と確率

観測されたカウントを n で割ると経験**頻度**（frequency）$\hat{p}$ が得られる。これは真の確率 $p = P(\text{heads}) = 0.5$ の**推定量**（estimator）である。100 回投げの実験を同時に 5 回繰り返すと、経験頻度を並べた 5×2 行列が得られ、その値は 0.5 の周りで目立って揺らいでいる。

```
Multinomial(100, fair_probs).sample((5,)) / 100
```

```
tensor([[0.5000, 0.5000],
        [0.4500, 0.5500],
        [0.5000, 0.5000],
        [0.4900, 0.5100],
        [0.5800, 0.4200]])
```

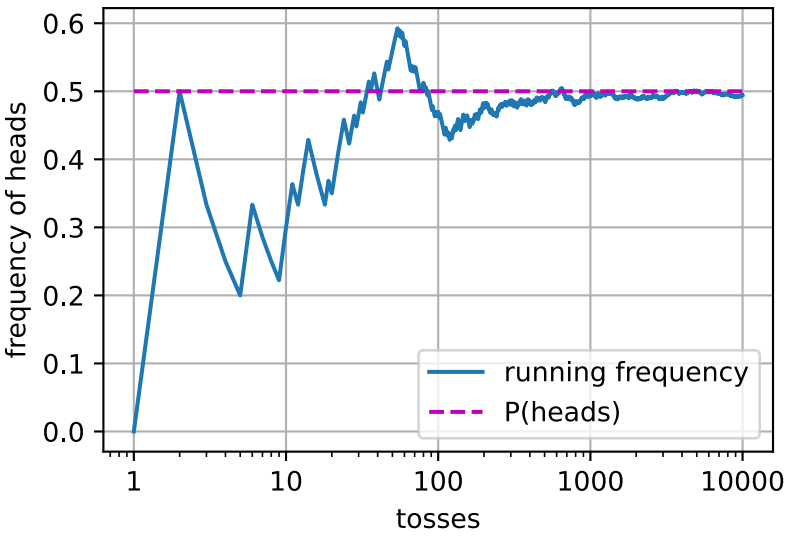
サンプルサイズを 100 倍の $n = 10{,}000$ 回に増やすと、典型的な標本誤差（sampling error）は $\sqrt{100} = 10$ 分の 1 に縮まり、表と裏の経験頻度は $(0.5, 0.5)$ にはるかに近づく。

```
Multinomial(10000, fair_probs).sample() / 10000
```

```
tensor([0.4959, 0.5041])
```

1 回目から 10,000 回目まで、1 回投げるごとにその時点までの逐次頻度を `cumsum` で追跡すると、**大数の法則**（law of large numbers、36.5 節）を見て取れる。この法則は、$\hat{p} \to p$ という収束が $n \to \infty$ の極限で成り立つことを保証する。

```
n = 10000
tosses = (torch.rand(n) < 0.5).to(torch.float32)  # 1 is heads, 0 is tails
estimates = tosses.cumsum(dim=0) / torch.arange(1, n + 1)
d2l.plot(torch.arange(1, n + 1), [estimates, torch.full((n,), 0.5)],
         'tosses', 'frequency of heads',
         legend=['running frequency', 'P(heads)'], xscale='log',
         figsize=(4.5, 3))
```



どれほどの速さで $\hat{p}$ が収束するのか、そして $56/100$ という結果が疑わしいのかどうかを知るために、ここで確率変数とそのばらつきを定式化しよう。

1.6.2 事象と確率変数

1.6.2.1 標本空間と事象

ランダムな試行において起こりうるすべての結果からなる集合を**標本空間**（sample space）$\mathcal{S}$ と呼ぶ（サイコロを 1 回振る場合、$\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ である）。**事象**（event）とは任意の部分集合 $\mathcal{A} \subseteq \mathcal{S}$ のことである（たとえば「偶数の目が出る」という $\mathcal{A} = \{2, 4, 6\}$ や、「5 以上の目が出る」という $\mathcal{B} = \{5, 6\}$ など）。**確率**関数 P は、Kolmogorov の 3 つの公理を満たすように、すべての事象 $\mathcal{A} \subseteq \mathcal{S}$ に実数を割り当てる (Kolmogorov 1933)。

- **非負性**：$P(\mathcal{A}) \geq 0$
- **正規化**：$P(\mathcal{S}) = 1$
- **可算加法性**：互いに排反な事象 $\mathcal{A}_1, \mathcal{A}_2, \ldots$（$\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$、ただし $i \neq j$）に対して $P(\bigcup_i \mathcal{A}_i) = \sum_i P(\mathcal{A}_i)$ が成り立つ。

これらの公理から、余事象の公式 $P(\mathcal{A}') = 1 - P(\mathcal{A})$ や $P(\emptyset) = 0$、そして次の包除原理（inclusion-exclusion formula、Figure 1.31）が導かれる。

$$P(\mathcal{A} \cup \mathcal{B}) = P(\mathcal{A}) + P(\mathcal{B}) - P(\mathcal{A} \cap \mathcal{B}). \tag{1.26}$$

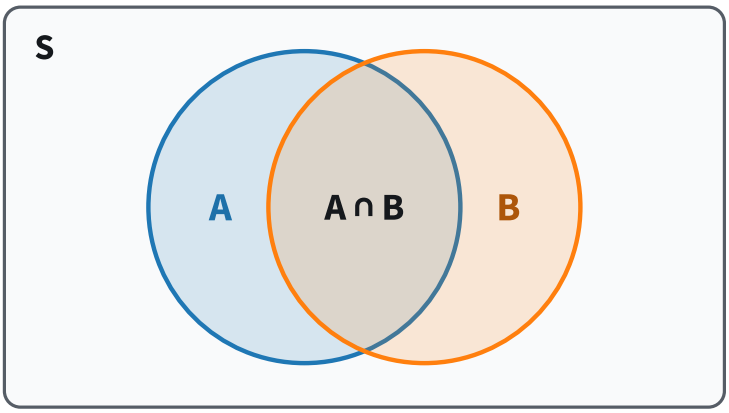


Figure 1.31: 2 つの事象 $\mathcal{A}$ と $\mathcal{B}$（標本空間 $\mathcal{S}$ 内）。和 $P(\mathcal{A}) + P(\mathcal{B})$ は共通部分 $\mathcal{A} \cap \mathcal{B}$ を 2 回数えている。

1.6.2.2 確率変数と分布

確率変数（random variable）X は、$\mathcal{S}$ 内の結果を数値に写す（たとえば 100 回のコイン投げで表が出る回数は、2^{100} 通りの系列を $\{0, \ldots, 100\}$ に写す）。**離散**（discrete）確率変数は、確率質量 $P(X = v)$ を各値 v に割り当てる（これをその**分布**（distribution）$P(X)$ と呼ぶ）。一方、**連続**（continuous）確率変数（人の正確な身長など）は**確率密度**（probability density）$p(x)$ によって記述され（Figure 1.32）、値が区間 $[a, b]$ に入る確率は密度曲線の下の面積 $P(a \leq X \leq b) = \int_a^b p(x)\,dx$ で与えられる（36.1 節）。

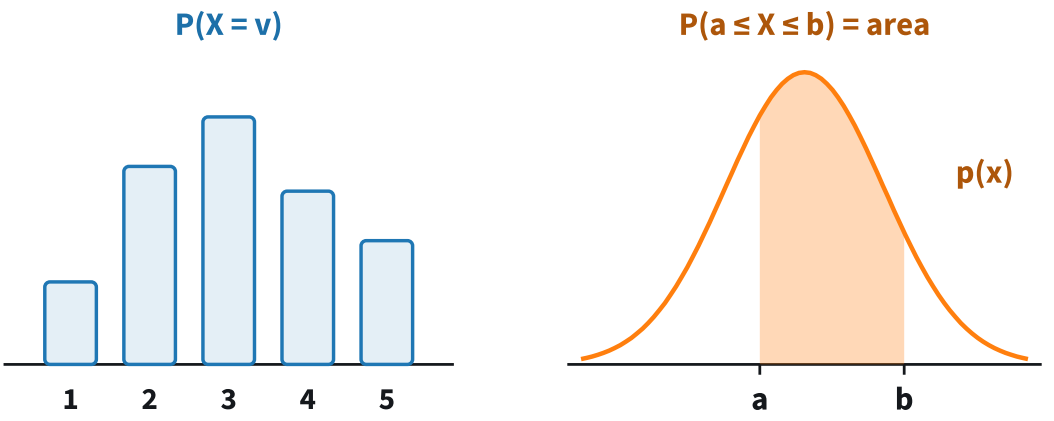


Figure 1.32: 離散確率変数は個々の値に確率を割り当てる（左）。連続確率変数は密度を持ち、ある区間の確率はその密度の下の面積となる（右）。

1.6.2.3 同時分布、周辺分布、独立な分布

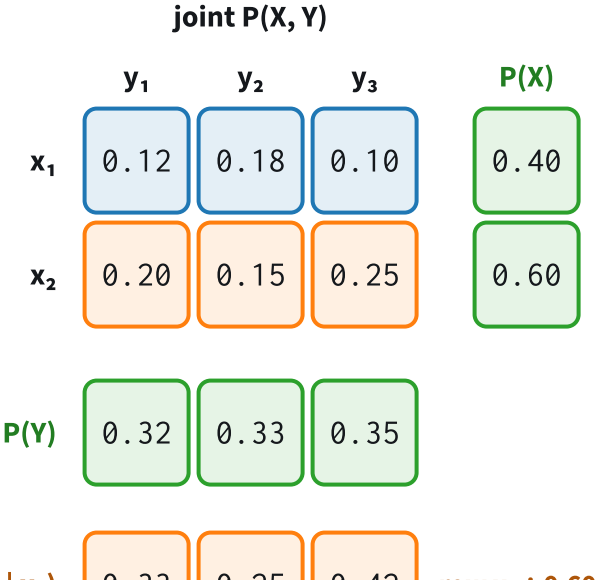
2 つの確率変数の**同時分布**（joint distribution）$P(X = x, Y = y)$ は、$X = x$ と $Y = y$ が同時に起こる確率を与える。たとえば、ある無作為な日における雨の有無（X）と傘の携帯（Y）について次の表を考える。

$P(X, Y)$	傘あり	傘なし
雨	0.25	0.05
雨なし	0.10	0.60

一方の変数について和をとって消去すると、もう一方の変数の**周辺分布**（marginal distribution）が得られる。この操作を**周辺化**（marginalization）と呼ぶ（Figure 1.33）。

$$P(X = x) = \sum_{y} P(X = x, Y = y), \tag{1.27}$$

したがって、$P(\text{rain}) = 0.25 + 0.05 = 0.30$ および $P(\text{umbrella}) = 0.25 + 0.10 = 0.35$ となる。



連続するコイン投げの結果は互いに独立であるが、雨と傘は独立ではない（$0.25 \neq 0.30 \cdot 0.35 = 0.105$）。

1.6.3 期待値と分散

1.6.3.1 期待値

離散確率変数 X の**期待値**（expectation）または**平均**は、確率で重み付けした平均として次のように定義される。

$$\mathbb{E}[X] = \sum_{x} x\, P(X = x), \tag{1.29}$$

また、任意の関数 f に対して $\mathbb{E}[f(X)] = \sum_x f(x)\, P(X = x)$ が成り立つ。たとえば、収益が $0\times$ になる確率が 0.5、$2\times$ になる確率が 0.4、$10\times$ になる確率が 0.1 の投資では、$\mathbb{E}[X] = 0.5(0) + 0.4(2) + 0.1(10) = 1.8$ となる。コイン投げの指示変数（indicator）$X = \mathbf{1}(\text{heads}) \in \{0, 1\}$ については、$\mathbb{E}[X] = 1 \cdot p + 0 \cdot (1 - p) = p$ である。

期待値は常に**線形**（linear）である。すなわち、$\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y]$ は X と Y が独立であるかどうかにかかわらず成り立つ。したがって、標本頻度 $\hat{p} = \frac{1}{n}\sum_{i=1}^{n} X_i$ は $\mathbb{E}[\hat{p}] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}[X_i] = p$ を満たし、$\hat{p}$ は p の**不偏**（unbiased）推定量となる。

1.6.3.2 分散と独立な変数の和

$\mu = \mathbb{E}[X]$ の周りのばらつきを測るため、**分散**（variance）は偏差の二乗の平均をとり、**標準偏差**（standard deviation）$\sigma = \sqrt{\mathrm{Var}(X)}$ は元の単位に戻す。

$$\mathrm{Var}(X) = \mathbb{E}\left[(X - \mu)^2\right] = \mathbb{E}[X^2] - \mu^2, \qquad \mathrm{Var}(aX + b) = a^2\,\mathrm{Var}(X). \tag{1.30}$$

先ほどの投資の例では、$\mathbb{E}[X^2] = 0.5(0) + 0.4(4) + 0.1(100) = 11.6$ であるから、$\mathrm{Var}(X) = 11.6 - 1.8^2 = 8.36$（$\sigma \approx 2.9$）となる。1 回のコイン投げの指示変数 $X \in \{0, 1\}$ では $X^2 = X$ であるため、$\mathrm{Var}(X) = p - p^2 = p(1-p)$ となり、最大値 $1/4$ を $p = 0.5$ のときにとる。1.2 節のように n 個の観測値から標本平均（sample mean）の周りの分散を推定する際には、$n - 1$ で割る**ベッセル補正**（Bessel's correction）によって、n で割る際に生じるわずかな下方向の統計的バイアス（bias）を補正する。

2 つの確率変数の和について、$\mathbb{E}[(X - \mu_X + Y - \mu_Y)^2]$ を展開すると次式が得られる。

$$\begin{aligned} \mathrm{Var}(X + Y) &= \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X, Y), \\ \mathrm{Cov}(X, Y) &= \mathbb{E}\left[(X - \mu_X)(Y - \mu_Y)\right], \end{aligned}$$

ここで、**共分散**（covariance）$\mathrm{Cov}(X, Y)$ と無次元の**相関**（correlation）$\rho = \mathrm{Cov}(X, Y)/(\sigma_X \sigma_Y) \in [-1, 1]$ は、線形な連動性を測る指標である（Figure 1.34）。より一般に、確率ベクトル（random vector）$\mathbf{x} \in \mathbb{R}^d$（平均 $\mu = \mathbb{E}[\mathbf{x}]$）に対して、すべての対の共分散 $\Sigma_{ij} = \mathrm{Cov}(x_i, x_j)$ を集めると対称な $d \times d$ の**共分散行列**（covariance matrix）$\Sigma = \mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^\top]$ が構成され、任意の線形結合 $\mathbf{v}^\top \mathbf{x}$ の分散は $\mathrm{Var}(\mathbf{v}^\top \mathbf{x}) = \mathbf{v}^\top \Sigma \mathbf{v}$ となる。

X と Y が**独立**であるとき、期待値は積に分解されるため $\mathrm{Cov}(X, Y) = \mathbb{E}[X - \mu_X]\mathbb{E}[Y - \mu_Y] = 0$ となり、**独立な変数の分散は足し合わされる**。

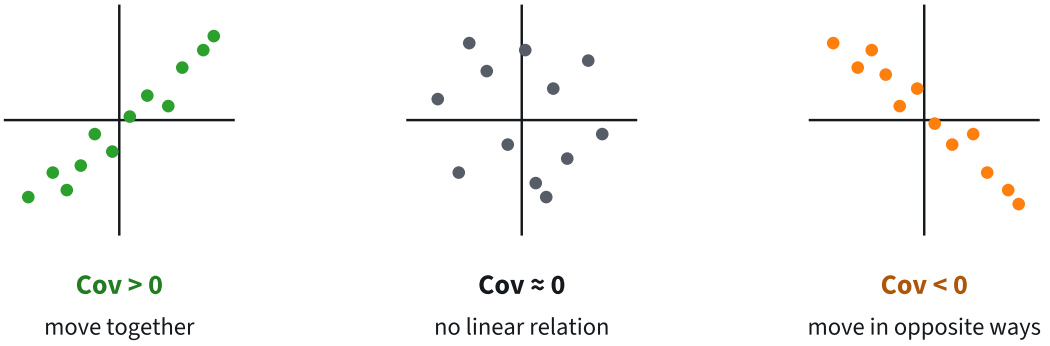


Figure 1.34: 正、ほぼゼロ、負の共分散を持つ 3 組の確率変数のサンプル。

$$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{1.31}$$

しかし、共分散がゼロであることは独立性を意味しない。たとえば X が $-1, 0, +1$ を等確率でとり $Y = X^2$ とすると、Y は X によって完全に決まるが、対称性により $\mathbb{E}[X] = 0$ かつ $\mathbb{E}[XY] = \mathbb{E}[X^3] = 0$ となるため、$\mathrm{Cov}(X, Y) = 0$ となる。

1.6.4 推定誤差

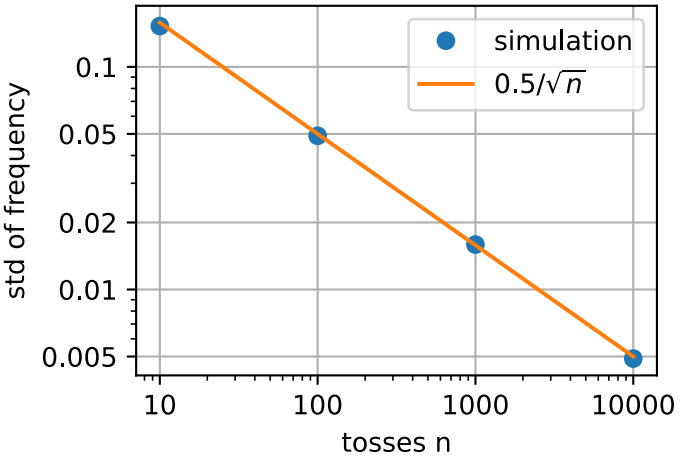
1.6.4.1 頻度の標準誤差

独立な n 回のコイン投げはそれぞれ分散 $p(1-p)$ を持つため、その和の分散は $np(1-p)$ となり、さらに n で割ると分散は $1/n^2$ 倍になる。

$$\mathrm{Var}(\hat{p}) = \frac{p(1-p)}{n}. \tag{1.32}$$

平方根をとると $\hat{p}$ の**標準誤差** $\sqrt{p(1-p)/n}$ が得られる。公平なコイン（$p = 0.5$）では $0.5/\sqrt{n}$ となり、その値は 0.05（$n = 100$ のとき）、0.005（$n = 10{,}000$ のとき）である。$1/\sqrt{n}$ の速さで減少するため、誤差を半分にするには $4\times$ のデータが必要になる。より一般に、独立な n 個のサンプルがそれぞれ分散 σ^2 を持つとき、その平均の標準誤差は $\sigma/\sqrt{n}$ となる。1000 回の反復を各 $n \in \{10, 100, 1000, 10000\}$ でシミュレーションすると、$0.5/\sqrt{n}$ が表す傾き $-1/2$ の直線が確認できる。

```
ns = [10, 100, 1000, 10000]
stds = [float((torch.rand(1000, n) < 0.5).to(torch.float32)
               .mean(dim=1).std()) for n in ns]
d2l.plot(ns, [stds, [0.5 / n ** 0.5 for n in ns]], 'tosses n',
         'std of frequency', legend=['simulation', r'$0.5/\sqrt{n}$'],
         xscale='log', yscale='log', fmts=('o', '-'))
```



1.6.4.2 偏りと不確実性の検定

これで最初のコインの疑問に答えることができる。100 回投げでは、$\hat{p} = 0.56$ は標準誤差の $0.06/0.05 = 1.2$ 倍だけ 0.5 から離れているにすぎないが、10,000 回投げでは同じ頻度 $\hat{p} = 0.56$（表が 5,600 回）でも標準誤差の $0.06/0.005 = 12$ 倍も離れている。公平なコインの実験を 1000 回シミュレーションすると、公平なコインが少なくともこれだけ大きく外れる頻度（**p 値**（p-value）の推定値）がわかる。

```
def fraction_as_far(heads, n, num_experiments=1000):
    """Fraction of fair-coin experiments at least as far from n/2 as heads."""
    counts = (torch.rand(num_experiments, n) < 0.5).sum(dim=1)
    far = (counts - n / 2).abs() >= abs(heads - n / 2)
    return far.to(torch.float32).mean()

fraction_as_far(56, 100), fraction_as_far(5600, 10000)
```

```
(tensor(0.2820), tensor(0.))
```

公平なコインを 100 回投げる試行の約 4 分の 1 では表の回数が 6 回以上ずれるため、$56/100$ は偏りの証拠にはならない。一方、公平なコインを 10,000 回投げる試行では、1000 回中 1 回も 5,600 の近くには達しない。こうした裾確率は 2 つの一般的な定理によって定量化される。**チェビシェフの不等式**（Chebyshev's inequality、36.1 節）は分散が有限である任意の分布に対して $P(|X - \mu| \geq k\sigma) \leq 1/k^2$ を保証し、**中心極限定理**（central limit theorem、36.3 節）は独立なサンプルの平均がガウス分布の釣鐘曲線に近づき、約 95% の確率で標準誤差の 2 倍以内に収まることを示す。同じ計算はモデルの評価にも当てはまる。真の正解率が 0.9 の分類器では、画像 256 枚の単一ミニバッチにおける標準誤差は $\sqrt{0.9 \cdot 0.1/256} \approx 0.019$（したがって $\pm 2\%$ の変動は通常のノイズである）であるのに対し、画像 10,000 枚のテストセット全体では 0.003 となる。

最後に、データを多く集めれば**認識的不確実性**（epistemic uncertainty、すなわちモデルパラメータ p に関する不確実性であり $1/\sqrt{n}$ で減衰する）は縮小するが、**偶然的不確実性**（aleatoric uncertainty、すなわち次のコイン投げそのものが持つ本質的な 50/50 のランダム性や、そもそも曖昧な画像）は決して減らないことに注意しよう (Der Kiureghian and Ditlevsen 2009)。

1.6.5 条件付き確率

1.6.5.1 条件付け、ベイズの定理、条件付き独立

対象を $X = x$（$P(X = x) > 0$）となる結果に限定して再正規化することで、$X = x$ が与えられたもとでの $Y = y$ の**条件付き確率**（conditional probability）が次のように定義される。

$$P(Y = y \mid X = x) = \frac{P(X = x, Y = y)}{P(X = x)} \quad \Longrightarrow \quad P(X, Y) = P(Y \mid X)P(X). \tag{1.33}$$

先ほどの天気の表において、雨という条件を付けると、雨の行 $(0.25, 0.05)$ を 0.30 で割り直すことになり、$P(\text{umbrella} \mid \text{rain}) = 0.25/0.30 \approx 0.83$ が得られる（条件なしでは 0.35）。同時確率（joint probability）を 2 通りの順序で $P(X, Y) = P(Y \mid X)P(X) = P(X \mid Y)P(Y)$ と書き、$P(Y) = \sum_x P(Y \mid X = x)P(X = x)$ で割ると、次の**ベイズの定理**（Bayes' theorem）が得られる (Bayes 1763)。

$$P(X \mid Y) = \frac{P(Y \mid X)P(X)}{P(Y)}. \tag{1.34}$$

仮説 H と観測データ D に対して、ベイズの定理は**事前分布**（prior）$P(H)$ を**尤度**（likelihood）$P(D \mid H)$ とエビデンス $P(D)$ によって更新し、**事後分布**（posterior）$P(H \mid D) \propto P(D \mid H)P(H)$ を導く。

2 つの変数 X と Y は、Z が与えられたもとで（$X \perp Y \mid Z$）、$P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$ を満たすとき**条件付き独立**（conditionally independent）であるという。条件付けは従属性を取り除くことも、新たに生み出すこともある（Figure 1.35）。たとえば**共通原因**（common cause）Z（子どもの年齢など）を観測すると靴のサイズと読解力の相関は消え、現在の状態 Y を**マルコフ連鎖**（Markov chain）$X \to Y \to Z$ において観測すると過去の X と未来の Z は独立になる。逆に、**共通結果**（common effect）Z で条件付けると（**説明し尽くし**、explaining away）、独立だった 2 つの原因が結びつく。たとえば入院患者が骨折しているとわかれば、肺がんで入院した確率は下がる。

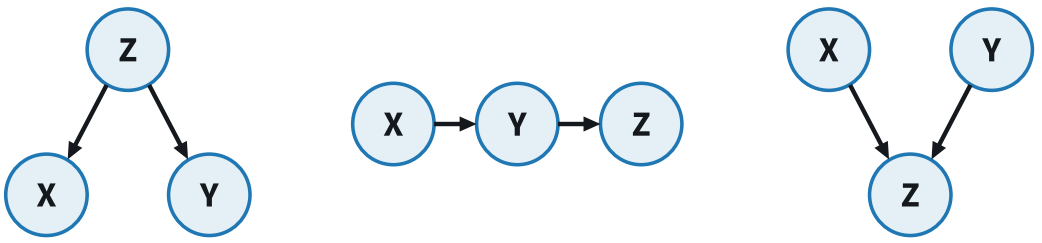


Figure 1.35: 条件付けは従属性を変化させる。共通原因 Z は X と Y を従属させるが、Z を観測するとその従属性は消える。連鎖構造では、X と Z は Y が与えられると独立になる。共通結果 Z は、Z が観測されると独立な原因どうしを従属させる。

1.6.5.2 例：診断検査

ある患者が HIV スクリーニング検査 $D_1 \in \{0,1\}$ を受け、真の HIV 感染状態が $H \in \{0,1\}$ であるとする。この検査は感度（sensitivity）が 100%、偽陽性率（false-positive rate）が 1% である。

$P(D_1 \mid H)$	$H=1$（HIV）	$H=0$（健康）
検査陽性（$D_1=1$）	1.00	0.01
検査陰性（$D_1=0$）	0.00	0.99

集団における有病率（prevalence）が $P(H=1)=0.0015$（$P(H=0)=0.9985$）であり、患者の検査結果が陽性（$D_1=1$）であったとすると、周辺化により $P(D_1=1)=0.01(0.9985)+1(0.0015)=0.011485$ となり、ベイズの定理から次式が得られる。

$$P(H=1 \mid D_1=1)=\frac{P(D_1=1 \mid H=1)P(H=1)}{P(D_1=1)}=\frac{0.0015}{0.011485}=0.1306. \quad (1.35)$$

10,000 人の集団（Figure 1.36）を考えると、感染している 15 人全員が陽性となる一方で、健康な人の 1%（9,985 人中 ≈ 100 件の誤報）も陽性となるため、陽性結果のうち本当に感染している割合は $15/115 \approx 13\%$ にとどまる。

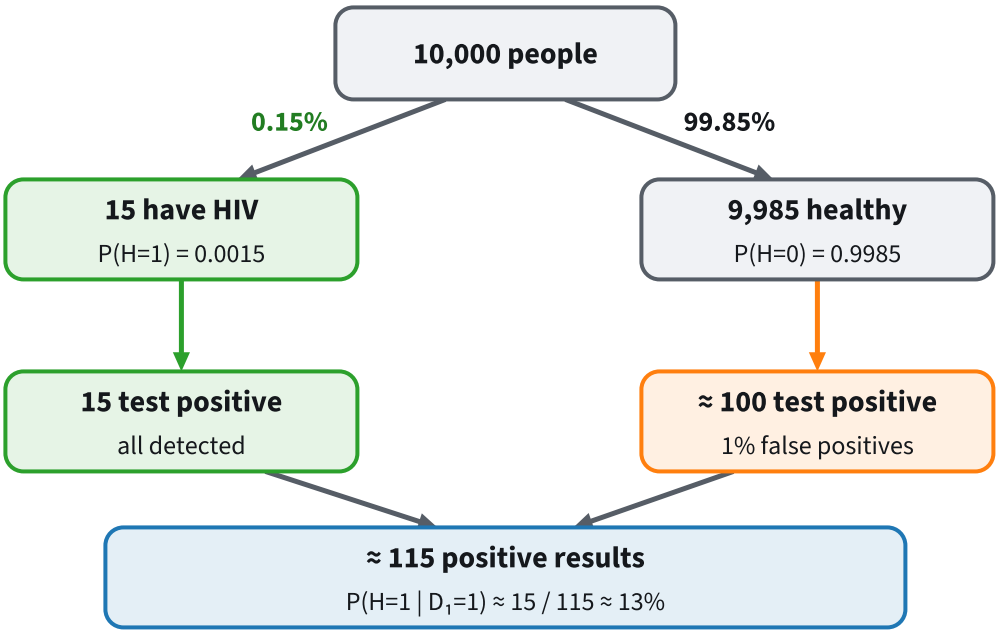


Figure 1.36: 10,000 人に HIV 検査を実施した例。感染者 15 人は全員陽性となるが、健康な 9,985 人のうち約 100 人も陽性となるため、約 115 件の陽性結果のうち正しいのは 15 件にすぎない。

ここで、追加検査 D_2（感度 $P(D_2=1 \mid H=1)=0.98$、偽陽性率 $P(D_2=1 \mid H=0)=0.03$）も陽性になったとする。D_1 と D_2 が H のもとで条件付き独立であると仮定すると、2 回の検査がともに偽陽性となる確率は $0.01 \cdot 0.03=0.0003$ に下がる一方、感度は $1 \cdot 0.98=0.98$ と

なる。周辺化により $P(D_1 = 1, D_2 = 1) = 0.0003(0.9985) + 0.98(0.0015) = 0.00176955$ が得られ、ベイズの定理によって感染の事後確率（posterior probability、Figure 1.37）は次の値まで上昇する。

$$P(H = 1 \mid D_1 = 1, D_2 = 1) = \frac{0.98 \cdot 0.0015}{0.00176955} = 0.8307. \tag{1.36}$$

仮想的な患者 200 万人をシミュレーションすると、経験的な割合は 0.82–0.83 前後となり、標準誤差 ± 0.006（$\sqrt{0.83 \cdot 0.17/3500}$）の 2 倍の範囲内で厳密値 0.8307 と一致する（両方陽性となる患者は $\approx 3{,}500$ 人である）。

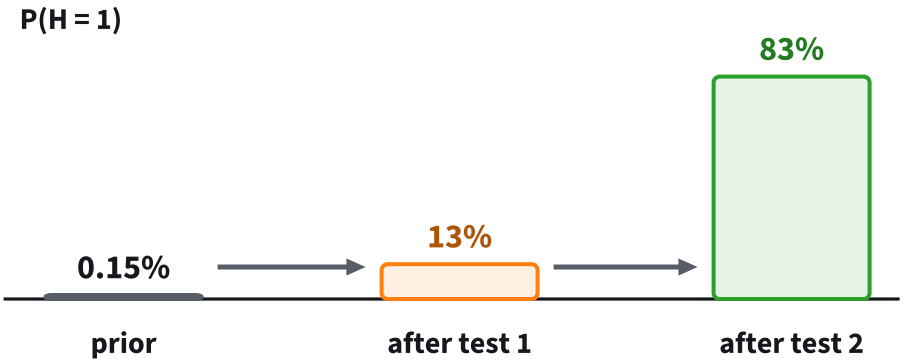


Figure 1.37: 患者が HIV に感染している確率。事前確率、1 回の陽性判定後、および条件付き独立な 2 回の陽性判定後。

```
n = 2000000
H = torch.rand(n) < 0.0015
D1 = torch.rand(n) < torch.where(H, 1.00, 0.01)
D2 = torch.rand(n) < torch.where(H, 0.98, 0.03)
H[D1 & D2].to(torch.float32).mean()  # Exact value: 0.8307
```

```
tensor(0.8220)
```

1.6.6 まとめ

確率はモデルから順方向に推論して結果の生じる見込みを導くのに対し、統計は観測データから逆方向に推論して背後にある過程の性質を推測する。Kolmogorov の 3 つの公理に基づく同時分布 $P(X, Y)$ は、変数の和をとって消去することで周辺化でき、変数が独立なときには $P(X)P(Y)$ と積に分解される。期待値 $\mathbb{E}[X]$ は常に線形であり、分散 $\mathrm{Var}(X) = \mathbb{E}[(X - \mu)^2]$ は独立な変数どうしで足し合わされるため、独立な n 個のサンプルの平均は標準誤差 $\sigma/\sqrt{n}$ を持つ。偏差を標準誤差の単位で測ることで、表が $56/100$ 回（標準誤差の 1.2 倍）出るのは日常的な偶然の範囲内である一方、$5{,}600/10{,}000$ 回（標準誤差の 12 倍）は決定的な証拠となる理由、そしてデータの追加収集によって**認識的**不確実性が $1/\sqrt{n}$ の速さで縮小する一方、**偶然的**不確実性は変わらない理由が明らかになる。最後に、ベイズの定理 $P(H \mid D) = P(D \mid H)P(H)/P(D)$ はエビデンスに照らして事前の信念を更新するものであり、まれな疾患（有病率 0.15%、偽陽性率 1%）に対する 1 回の陽性判定では事後確率が 13.1% にとどまり、条件付き独立な 2 回目の検査で陽性となって初めて 83.1% に達する理由を説明する。

1.6.7 演習問題

1. **ポートフォリオの収益とリスク**。 n 個の銘柄からなるポートフォリオを運用しており、各銘柄の収益 $\mathbf{s} \in \mathbb{R}^n$ は平均 $\mu = \mathbb{E}[\mathbf{s}]$、共分散行列 Σ を持つ確率変数であるとする。資金の割合 $\alpha_i \geq 0$ を銘柄 i に投資し、$\sum_i \alpha_i = 1$ とする。

   1. ポートフォリオ α の期待収益と収益の分散を計算せよ。
   2. すべての銘柄が分散 σ^2 を持ち、異なる銘柄のすべての対が相関 $\rho \geq 0$ を持つと仮定する。等重みポートフォリオ $\alpha_i = 1/n$ の分散と、その $n \to \infty$ における極限を計算せよ。また、$\rho = 0$ の場合を Equation 1.32 と比較せよ。
   3. （発展）分散をある上界以下に制約しつつ期待収益を最大化する最適化問題を定式化せよ。これは Markowitz のポートフォリオ問題である (Markowitz 1952)。この種の制約付き問題に対する手法は 35.6 節で展開する。

2. [code] **コイン投げ推定量に対するチェビシェフの不等式**。 Equation 1.32 は、表の頻度 $\hat{p}$（n 回投げた後）の分散を与える。

   1. 公平なコインで $n = 100$ のとき、チェビシェフの不等式を用いて $P(|\hat{p} - p| \geq 0.1)$ の上界を求めよ。
   2. 100 回投げの実験を 10,000 回反復するシミュレーションによって同じ確率を推定し、求めた上界と比較したうえで、その差が生じる理由を説明せよ。
   3. 公平なコインに対して $P(|\hat{p} - p| \geq 0.01) \leq 0.05$ を保証するためにチェビシェフの不等式が必要とする試行回数を求め、n のうち標準誤差 $\sqrt{p(1-p)/n}$ 自体が 0.01 に等しくなる値と比較せよ。

3. **2 つの分類器の比較**。●2 つの分類器を、画像 $n = 10{,}000$ 枚からなる同一のテストセット上で評価する。分類器 A は 9,050 枚の画像に正しいラベルを付け、分類器 B は 9,000 枚に正しいラベルを付けた。テスト画像は独立なサンプルとして扱え。

   1. それぞれの正解率の標準誤差を計算せよ。
   2. 2 つの正解率が独立であると仮定し、その差の標準誤差を計算せよ。0.005 という差はノイズと比べて大きいか。
   3. 2 つの正解率は同じ画像上で測定されているため、実際には独立ではない。300 枚の画像で両分類器の判定が食い違い、そのうち 175 枚では A のみが正解し、125 枚では B のみが正解したとする。画像 i について、A のみが正解なら $Z_i = 1$、B のみが正解なら $Z_i = -1$、それ以外なら $Z_i = 0$ とおくと、正解率の差は Z_i の平均となる。この平均の標準誤差を計算し、前問の答えと比較せよ。
   4. $\mathrm{Var}(X - Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) - 2\,\mathrm{Cov}(X, Y)$ を用いてこの違いを説明せよ。1 枚の画像に対する 2 つの分類器の正誤のあいだの共分散はどの符号を持ち、それはなぜか。

4. **連鎖構造の分解**。確率変数 X、Y、Z において、Figure 1.35 の連鎖構造のように、Z が X に対して Y を与えたもとで条件付き独立であるとする。

   1. 同時確率 $P(X, Y, Z)$ を、それぞれが高々 2 つの変数しか含まない因子の積として表せ。
   2. 系列 $X_1, \ldots, X_T$ において、各 X_t がそれ以前の変数に X_{t-1} を通じてのみ依存する場合へと一般化せよ。すべての X_t が k 通りの値をとるとき、同時分布を定めるのに必要な数値の個数は、そのような T 個の変数の任意の同時分布と比べていくつになるか。

5. **共通結果による条件付け**。独立な 2 枚の公平なコインを投げ、$X, Y \in \{0, 1\}$ をそれぞれで表が出たことを表す指示変数とする。また、$Z = \max(X, Y)$ を少なくとも 1 枚のコインで表が出たことを表す変数とする。$P(X = 1)$、$P(X = 1 \mid Z = 1)$、および

$P(X = 1 \mid Z = 1, Y = 1)$ を計算せよ。これらの値を用いて、X と Y が $Z = 1$ のもとで条件付き独立であるかどうかを判定し、2 番目の値から 3 番目の値への変化を Figure 1.35 と関連付けて説明せよ。

6. **相関のある 2 回目の検査**。●HIV 感染患者に対しては結果が条件付き独立であるが、健康な患者に対しては条件付き独立でない 2 つの新しい検査を用いて、1.6.5.2 節を再考せよ。いずれの検査も単独では偽陽性率が 10%、偽陰性率が 1% であるとする。

$$P(D_i = 1 \mid H = 0) = 0.1, \quad P(D_i = 0 \mid H = 1) = 0.01 \quad \text{for } i = 1, 2. \tag{1.37}$$

$H = 1$ に対しては、同時分布は $P(D_1, D_2 \mid H = 1) = P(D_1 \mid H = 1)\, P(D_2 \mid H = 1)$ と積に分解される。$H = 0$ に対しては、結果は $P(D_1 = D_2 = 1 \mid H = 0) = 0.02$ によって連動している。基準となる事前確率は先ほどと同じく $P(H = 1) = 0.0015$ である。

   1. D_1 と D_2 の（$H = 0$ のもとでの）同時確率表を求めよ。
   2. 1 回目の検査が陽性となり、2 回目の検査を実施する前の時点で、患者が HIV に感染している確率を導出せよ。
   3. 両方の検査が陽性となった後に、患者が HIV に感染している確率を導出せよ。
   4. $H = 0$ のもとでも条件付き独立である検査について前問の計算を繰り返し、2 つの事後確率が異なる理由を説明せよ。

7. **弁護人の統計**。ある容疑者には、殺害された妻を過去に虐待していた記録があった。弁護人は、統計的に妻を虐待する夫のうち実際に妻を殺害するに至るのは 1,000 人に 1 人にすぎないため、虐待歴は弱い証拠でしかなく陪審員は無罪とすべきだと主張している。「容疑者が妻を殺害した」を M、「妻が殺害された」を K、「容疑者に妻への虐待歴があった」を B と表す。

   1. 弁護人が挙げた 1,000 人に 1 人という数値と、有罪の事後確率をこれらの記号で表し、弁護人がどの確率とどの確率を取り違えているかを特定せよ。
   2. 弁護人の統計が対象とする期間において、虐待を受けている妻が夫以外の誰かに殺害される確率が 10^{-5} であると仮定する。有罪の事後確率を計算し、弁護人の数値と比較せよ。

   MIT 18.05、問題セット 7、問題 4 を改変。

Chapter 2

線形回帰

回帰（regression）は、床面積と築年数からの住宅販売価格の推定、バイタルサインからの患者の入院期間の予測、次の四半期の電力需要の予測のように、対象が離散的なカテゴリではなく意味のある数値であるあらゆる場面で、定量的な予測の主力となる。また、深層学習にとっても理想的な検証の場である。ニューラルネットワークの訓練を構成する要素、すなわちパラメータ化された順伝播、最尤法に基づく損失関数、ミニバッチ確率的勾配降下法、再利用可能な訓練ハーネス、汎化の診断、そして重みノルムによる正則化のほぼすべてが、**線形回帰**（linear regression）において最も単純で見通しのよい形で、すでに現れている。

まず 2.1 節では、アフィンモデルと二乗誤差損失を定式化し、直交射影による閉形式の解法とミニバッチ確率的勾配降下法（SGD）による反復的な解法の両方でこの問題を解いたうえで、二乗損失をガウス分布の下での最尤法と結びつける。実データを扱う前に、2.2 節では既知の正解パラメータ $(\mathbf{w}^*, b^*)$ とノイズレベル σ をもつ合成データのベンチマークを構築し、コードを検証するための厳密な基準を用意する。続く 3 つの節では、訓練パイプラインを段階的に構築していく。まず生のテンソルと自動微分（automatic differentiation）のみから実装し（2.3 節）、次に本書の以降の章を支える再利用可能な `Module`、`DataModule`、`Trainer` クラスへと整理し（2.4 節）、最後にフレームワークの高水準な層とオプティマイザを用いてわずか数行で表現する（2.5 節）。

訓練データへの当てはめは、問題の半分にすぎない。2.6 節では多項式曲線フィッティングを題材に、訓練誤差と汎化誤差（generalization error）を切り分け、期待テスト誤差を統計的バイアスの二乗、分散、ノイズへと分解したうえで、テストデータを覗き見ることなく検証データへの分割や K 分割交差検証（cross-validation）によってモデルの複雑さを選択する方法を示す。最後に 2.7 節では、二乗 ℓ_2 ペナルティを加えることで高次元における過学習（overfitting）を抑え、重み減衰（weight decay）が訓練入力からは十分に定まらない方向を選択的に縮小させる仕組みを明らかにする。

2.1　線形回帰

住宅の床面積と築年数から売却価格を見積もりたいとしよう。入力を離散的なカテゴリに割り当てる分類（classification）とは異なり、**回帰**（regression）は連続的な数値を予測する。定量的入力を数値出力に写す規則を求める試みには、2 世紀にわたる輝かしい歴史がある。1805 年、Adrien-Marie Legendre はノイズを含む天体観測から彗星の軌道を決定するために**最小二乗法**（method of least squares）を発表した (Legendre 1805)。その 4 年後、Carl Friedrich Gauss は、どのような測定誤差の分布のもとで最小二乗法による当てはめが最も確からしくなるかを問うことで最小二乗法を確率論で基礎づけ、ガウス分布に到達した (Gauss 1809)（もっとも Gauss 自身

は、新発見の準惑星ケレスが太陽のまぶしさの陰に消えた後、その位置を再特定するために 1801 年にはすでにこの手法を用いていた)。

今日、回帰は臨床リスクのスコアリングや需要予測から物体検出器のバウンディングボックスヘッドに至るまで、あらゆる応用を支えている。さらに都合のよいことに、予測が入力のアフィン結合となる**線形回帰**（linear regression）には、手計算で完全に解析できるほど単純な設定のなかに、深層学習のほぼすべての中核概念が詰まっている。計算グラフ（computational graph）として見れば、線形回帰は 1 個の人工ニューロン、すなわち出力ユニットを 1 つ持つ**全結合層**（fully connected layer）である。本節では、モデルと二乗損失（squared loss）を定式化し、正規方程式（normal equations）による解析的な解法と（深層学習の主力オプティマイザである）**ミニバッチ確率的勾配降下法**（minibatch stochastic gradient descent）による反復的な解法の両方でこれを解く。さらに、ベクトル化された配列演算と Python のループの速度を比較し、異なるノイズ分布のもとでの最尤推定（maximum likelihood estimation）へと損失関数（loss function）を結びつける。

```
import time
import numpy as np
import torch
```

2.1.1 モデルと損失

2.1.1.1 特徴量とラベル

住宅の面積（平方メートル単位）と築年数（年単位）から売却価格を推定するために、過去の売買記録からなる**訓練データセット**（training dataset）を集める。その記録に含まれる各住宅が 1 つの**例**（example。**データ点**、**サンプル**、**インスタンス**とも呼ばれる）である。予測したい量（ここでは価格）が**ラベル**（または**目的変数**、target）であり、予測の根拠となる測定可能な属性（面積と築年数）が**特徴量**（feature。または**共変量**、covariate）である。

データセットに n 個の例が含まれるとしよう。これらに $i \in \{1, \dots, n\}$ で添字を付け、$\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})^\top \in \mathbb{R}^2$ で i^{th} の住宅の特徴ベクトル（feature vector）を、$y^{(i)} \in \mathbb{R}$ でその観測された売却価格を表す。より一般に、各例が d 個の特徴量で記述されるとき $\mathbf{x}^{(i)} \in \mathbb{R}^d$ となり、データセット全体は対の集合 $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$ となる。

2.1.1.2 線形モデル

あらゆる教師あり学習（supervised learning）の手法は、特徴量を予測（prediction）へと写すパラメータ化された関数族、すなわち**モデルクラス**から出発する。線形の仮定では、目的変数を特徴量の重み付き和にオフセットを加えた形で近似できるとみなす。

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{2.1}$$

ここで、w_{area} と w_{age} は**重み**（weight）であり、b は**バイアス**（**オフセット**、あるいは**切片**、intercept とも呼ばれる）である。各重みはその特徴量の限界的な影響度（築年数を固定したとき、床面積が 1 平方メートル増えるごとに予測価格がどれだけ変化するか）を定め、一方バイアス b はすべての特徴量が 0 のときの予測値を決める。現実には面積が 0 の住宅は存在しなくても、b を残しておくことは不可欠である。これがないと当てはめる平面は必ず原点を通ることになり、モデルが表現できる関数が著しく制限されてしまうからである。厳密に言えば、Equation 2.1 は特徴量の**アフィン**関数（線形変換に定数の平行移動を加えたもの）であるが、機械学習ではこれを一様に線形モデルと呼ぶ。

特徴量が d 個あるとき、モデルの予測値 $\hat{y}$（ハット記号は観測ラベル y と推定値を区別するためのもの）は次の形をとる。

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b = \mathbf{w}^\top \mathbf{x} + b, \tag{2.2}$$

ここで、$\mathbf{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ は重みを 1 つの列ベクトルにまとめたものである。全 n 個の例に対してモデルを一度に評価する際には、特徴ベクトル $\mathbf{x}^{(i)}$ を行として積み重ねた**計画行列**（design matrix）$\mathbf{X} \in \mathbb{R}^{n \times d}$ と、予測値 $\hat{y}^{(i)}$ をまとめた列ベクトル $\hat{\mathbf{y}} \in \mathbb{R}^n$ を用いて次のように書く。

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{2.3}$$

ここで、スカラーのバイアス b はブロードキャスト（1.1.3.2 節）によって $\mathbf{X}\mathbf{w}$ のすべての成分に加算される。関係する各配列の形状を Figure 2.1 に示す。これにより、最良の線形予測器（linear predictor）を探す問題は、パラメータ $(\mathbf{w}, b) \in \mathbb{R}^d \times \mathbb{R}$ のうち、$\hat{\mathbf{y}}$ を観測ラベル $\mathbf{y} = (y^{(1)}, \ldots, y^{(n)})^\top$ にできる限り近づけるものを見つける問題へと帰着する。特徴量とラベルの真の関係が近似的にしか線形でなく、未測定の要因や測定ノイズが混入している場合でさえ、線形モデルは高速で解釈しやすく、驚くほど競争力のあるベースラインとなることが多い。

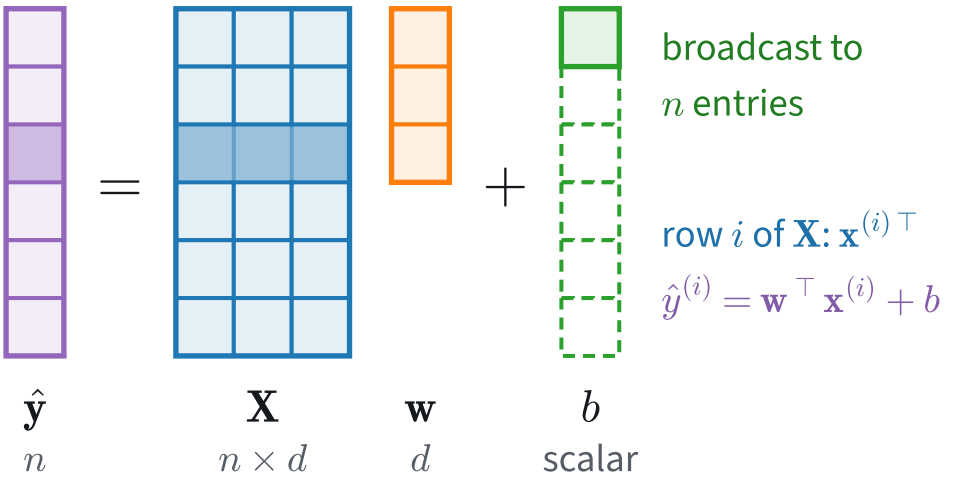


Figure 2.1: $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ における各配列の形状（$n = 6$ 個の例と $d = 3$ 個の特徴量の場合）。$\mathbf{X}$ の各行は 1 つの例を表し、$\hat{\mathbf{y}}$ の 1 つの要素（網掛け部分）を生成する。$\mathbf{w}$ は列ごとに 1 つの重みを持ち、スカラー b は全 n 個の要素にブロードキャストされる。

2.1.1.3 二乗損失

最良のパラメータを探索するには、それに先立って、任意の候補 $(\mathbf{w}, b)$ が観測ラベルからどれほど大きく外れているかを測る定量的な尺度が必要になる。**損失関数**は予測値 $\hat{y}$ と真の値 y のずれに対して非負のペナルティを割り当て、両者が一致するときに 0 となる。回帰における標準的な選択は**二乗誤差**（squared error）である。

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{2.4}$$

係数 $\frac{1}{2}$ は純粋に式を見やすくするための便宜的なものであり、二乗を微分した際に降りてくる係数 2 を打ち消すだけで、最小点の位置は変えない。差 $\hat{y}^{(i)} - y^{(i)}$ が二乗されるため、同じ大きさの過大予測と過小予測には対称なペナルティが科される一方、大きな外れには小さな外れよりもはるかに重いペナルティが科される。たとえば 3 だけ外れた予測は、1 だけ外れた予測の 9 倍の損失を生む。幾何学的には（Figure 2.2）、各残差（residual）$\hat{y}^{(i)} - y^{(i)}$ はデータ点と回帰直線のあいだの鉛直距離であり、$2\,l^{(i)}$ はその鉛直線分を 1 辺とする正方形の面積にあたる。

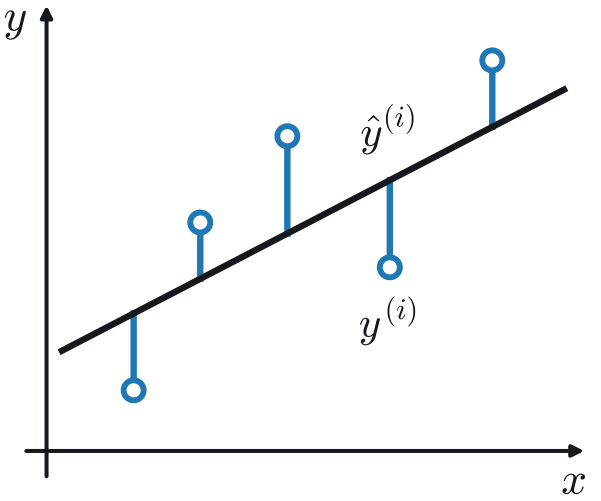


Figure 2.2: 鉛直方向の残差の二乗和を最小化することによるデータへの直線の当てはめ。

$(\mathbf{w}, b)$ を n 個の例からなる訓練セット（training set）全体で評価するため、例ごとの損失を平均して経験**損失**（または**コスト・目的関数**）を定義する。

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$
$$= \frac{1}{2n}\|\mathbf{X}\mathbf{w} + b - \mathbf{y}\|_2^2.$$

n で割ることで目的関数が 1 例あたりのスケールになるため、データセットのサイズが 2 倍になっても損失やその勾配（gradient）の数値的な大きさは 2 倍にならない。モデルを訓練するとは、次の最適化（optimization）問題を解くことを意味する。

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b). \tag{2.5}$$

誤差が二乗されるため、10 だけ外れた予測は 1 だけ外れた予測の 100 倍のペナルティを被り、たった 1 つの破損したラベルが当てはめ全体を支配してしまうことがある。その影響がどれほど深刻かを見るために、直線 $y = 2x$ 上に 20 個の点を置き、（データ入力ミスで起こりうるように）1 つのラベルだけを $y^{(6)} = 10{,}000$ に破損させたうえで、原点を通る直線 $\hat{y} = wx$ を 2 通りの方法で当てはめてみよう。1 つ目は二乗損失の閉形式（closed-form）の最小点（minimizer）$w = \sum_i x^{(i)} y^{(i)} / \sum_i (x^{(i)})^2$ による方法、2 つ目は傾きの候補を細かく並べたグリッド上で**平均絶対誤差**（mean absolute error）$\frac{1}{n}\sum_i |wx^{(i)} - y^{(i)}|$ を最小化する方法である。

```
x = np.arange(1.0, 21.0)
y = 2 * x
y[5] = 10000                            # corrupt a single label
w_sq = (x * y).sum() / (x * x).sum()    # squared-loss minimizer
w_grid = np.linspace(0, 30, 3001)       # slopes 0, 0.01, ..., 30
mae = np.abs(w_grid[:, None] * x - y).mean(axis=1)
w_mae = w_grid[mae.argmin()]            # MAE minimizer on the grid
print(f'true w: 2.00, squared loss: {w_sq:.2f}, MAE: {w_mae:.2f}')
```

```
true w: 2.00, squared loss: 22.88, MAE: 2.00
```

20 個のうちわずか 1 個の破損した点が二乗損失による傾きを 2.00 から 22.88 まで一気に引きずり込むのに対し、絶対誤差による傾きは 2.00 から微動だにしない。二乗損失のもとでは、ほぼ 10,000 に達する 1 つの誤差が残り 19 個の正常な点の引き戻す力の合計を圧倒してしまうが、絶対誤差のもとではその引っ張る力が線形にしか増えないため、19 個の正常な点が容易に多数決で勝つのである。どちらの振る舞いが望ましいかはラベルにどのようなノイズを想定するかに依存する。この問いには 2.1.3 節で第一原理から答える。

2.1.2 損失の最小化

$(\mathbf{w}^*, b^*)$ は、特徴量が d 個、例が n 個あるときにどうやって求めればよいだろうか。線形回帰は、厳密な閉形式の公式とスケーラブルな反復オプティマイザの両方を持つ、機械学習でもごくまれなモデルの 1 つである。この 2 つを比較すると、深層学習がなぜ圧倒的に反復的な勾配法を好むのかが明らかになる。

2.1.2.1 正規方程式

まず、スカラーのバイアス b は、計画行列に定数 1 の列を追加することで重みベクトルに吸収できる。各入力は $(\mathbf{x}^{(i)}, 1)^\top \in \mathbb{R}^{d+1}$ となり、パラメータベクトルは $(\mathbf{w}, b)^\top \in \mathbb{R}^{d+1}$ となる（あるいは同値な方法として、$\mathbf{X}$ と $\mathbf{y}$ の両方をそれぞれの標本平均で中心化してもよい。演習問題 1 を参照）。この拡大表記で $\mathbf{X}$ と $\mathbf{w}$ を書けば、損失は単純に次の形になる。

$$L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2. \tag{2.6}$$

$\mathbf{X}^\top\mathbf{X}$ が正則である限り（これは $\mathbf{X}$ の列が線形独立であるとき、かつそのときに限り成り立ち、$n \geq d$ が必要となる）、導関数を計算せず平方完成という純粋な代数操作だけで大域的最小値（global minimum）を求めることができる。$\hat{\mathbf{w}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$ と定義すると、$\mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^\top\mathbf{y}$ が成り立つ。ノルムの内部で $\mathbf{X}\hat{\mathbf{w}}$ を足して引き、二乗を展開すると次を得る。

$$\begin{aligned}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 &= \|(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) + \mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2 \\ &= \|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|_2^2 + \|\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2 + 2(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w}).\end{aligned} \tag{2.7}$$

末尾の交差項に注目しよう。

$$(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top\mathbf{X} = (\mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\hat{\mathbf{w}})^\top = \mathbf{0}^\top.$$

この項は**あらゆる** $\mathbf{w}$ に対して恒等的に 0 になる。残るのは 2 つの非負の二乗ノルムの和である。第 1 項 $\|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|_2^2$ は $\mathbf{w}$ にまったく依存せず、第 2 項 $\|\mathbf{X}(\hat{\mathbf{w}} - \mathbf{w})\|_2^2$ は $\mathbf{w} = \hat{\mathbf{w}}$ のとき、かつその

ときに限り 0 になる（$\mathbf{X}$ が列フルランクを持つため）。したがって、$\hat{\mathbf{w}}$ が唯一の大域的最小点である。微分法を用いて勾配 $\nabla_{\mathbf{w}} L(\mathbf{w}) = -\frac{1}{n}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$ を 0 とおいても同じ方程式が得られ、古典的な**正規方程式**が導かれる。

$$\mathbf{X}^\top \mathbf{X}\,\mathbf{w}^* = \mathbf{X}^\top \mathbf{y}, \qquad \text{or} \qquad \mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}. \tag{2.8}$$

$\mathbf{X}$ の列が線形従属である場合（たとえば 2 つの特徴量が冗長である場合や $d > n$ の場合）、$\mathbf{X}^\top\mathbf{X}$ は特異になる。このときも最小二乗法の最小点は依然として存在するが、無限に多くの重みベクトルが同じ最小損失を達成する（演習問題 6）。

コード上では、$(\mathbf{X}^\top\mathbf{X})^{-1}$ を明示的に計算すべき場合はまずない。専用の最小二乗ソルバ（`lstsq`）は QR 分解や特異値分解（singular value decomposition、SVD）を用いるため、同程度の計算コストではるかに数値的に安定である。1.4 節の 3 点の直線モデル（入力 $0, 1, 2$、ラベル $1, 1, 0$、拡大行 $(1, \text{input})$）で正規方程式を検証してみよう。このとき $\hat{\mathbf{w}} = (w_1, w_2)^\top = (7/6, -1/2)^\top$ は第 1 成分に切片、第 2 成分に傾きを持つ。あわせて、残差ベクトル $\mathbf{r} = \mathbf{X}\hat{\mathbf{w}} - \mathbf{y} = (1/6, -1/3, 1/6)^\top$ が `float32` の精度の範囲内で $\mathbf{X}^\top\mathbf{r} = \mathbf{0}$ を満たすことも確認しよう。

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows (1, input)
y = torch.tensor([[1.0], [1.0], [0.0]])
w_hat = torch.linalg.lstsq(X, y).solution
r = X @ w_hat - y                                       # errors
print('w_hat:', tuple(round(float(v), 4) for v in w_hat[:, 0]))
print('r:', tuple(round(float(v), 4) for v in r[:, 0]))
print(f'max |X^T r|: {float((X.T @ r).abs().max()):.0e}')
```

```
w_hat: (1.1667, -0.5)
r: (0.1667, -0.3333, 0.1667)
max |X^T r|: 2e-07
```

2.1.2.2 射影としての最小二乗法

なぜこれは**正規**（normal）方程式と呼ばれるのだろうか。交差項が消える Equation 2.7 の恒等式は、文字どおり $\mathbb{R}^n$ におけるピタゴラスの定理にほかならない（Figure 2.3）。$\mathbf{w}$ が $\mathbb{R}^d$ 全体を動くとき、とりうる予測ベクトル $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ は d 次元の部分空間（subspace）を $\mathbb{R}^n$ 内に描き出す。これが列空間（column space）、すなわち $\mathbf{X}$ が張る空間（span）である。通常、観測されたラベルベクトル $\mathbf{y} \in \mathbb{R}^n$ はこの部分空間上には乗らない。d 本の特徴量の列をどのように線形結合しても、ノイズを含む n 個の観測値すべてを再現することはできないからである。

$\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$ を最小化するとは、点 $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ として、$\mathbf{X}$ の列空間内でユークリッド距離（Euclidean distance）において $\mathbf{y}$ に最も近い点を見つけることを意味する。初等幾何学より、平坦な部分空間内で最も近い点は、その部分空間への $\mathbf{y}$ の**直交射影**（orthogonal projection）、すなわち差ベクトル $\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}$ が $\mathbf{X}$ のすべての列に垂直（**法線**方向、normal）となる唯一の点である。$\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}$ が $\mathbf{X}$ のすべての列と直交するという条件は $\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) = \mathbf{0}$ と書け、これこそが正規方程式にほかならない。

2.1.2.3 ミニバッチ確率的勾配降下法

線形回帰に 1 行で書ける解析解があるなら、なぜここで終わりにしないのだろうか。現代の深層学習全般において閉形式解を阻む障壁が 2 つある。

1. **計算量のスケーリング：**$\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{d\times d}$ の計算には $O(nd^2)$ 回の演算コストがかかり、$d \times d$ の連立方程式を解くには $O(d^3)$ のコストがかかる。データセットが数百万個の例を持ち、

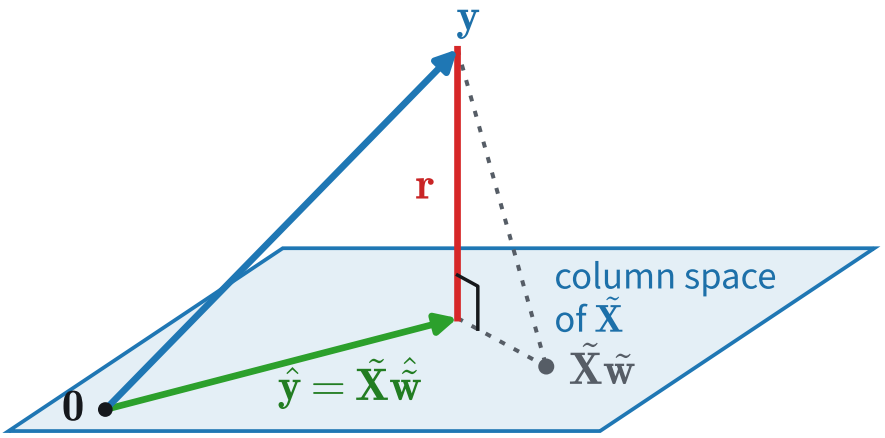


Figure 2.3: $\mathbb{R}^n$ における直交射影としての最小二乗法。$\mathbf{X}$ の列空間はあらゆる予測ベクトル $\mathbf{Xw}$ を含み、$\mathbf{y}$ に最も近い点が直交射影 $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ である。その残差 $\mathbf{y} - \hat{\mathbf{y}}$ は $\mathbf{X}$ のすべての列と直交する（法線をなす）。

モデルが数百万〜数十億個のパラメータを持つとき、$d \times d$ 行列を保存したり逆行列を求めたりすることは到底不可能である。

2. **非線形性：**二乗損失を交差エントロピー（cross-entropy、第 3 章）のような別の目的関数に置き換えたり、非線形な隠れ層（hidden layer、第 4 章）を積み重ねたりした途端に、閉形式解は一切存在しなくなる。

その代わりに、ほぼすべての深層学習モデルは**勾配降下法**（gradient descent）とその確率的変種を用いて反復的に訓練する。1.4 節で見たように、勾配 $\nabla L(\mathbf{w}, b)$ は損失が最も急激に増加する方向を指す。したがってその**逆**方向に進めば損失は減少するが、全勾配 $\nabla L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \nabla l^{(i)}(\mathbf{w}, b)$ を評価するには、パラメータをたった 1 ステップ更新するだけでも全 n 個の訓練例を走査しなければならない。$n = 10^7$ のとき、フルバッチ勾配降下法は気が遠くなるほど遅いうえに統計的にも無駄が多い。無作為に選んだ 100 個の例にわたる平均勾配で、すでに 1000 万個にわたる平均の実用的な推定値になるからである。その推定値がどれほど正確かは標本の大きさに依存するのであり、抽出元のデータセットの大きさには依存しない。

ミニバッチ確率的勾配降下法（ミニバッチ SGD）は、この統計的冗長性を活用する。各ステップ t において、小さな無作為**ミニバッチ** $\mathcal{B}_t \subset \{1, \dots, n\}$（例数 $|\mathcal{B}|$）を抽出し、$\mathcal{B}_t$ のみにわたって損失の勾配を平均し、正の**学習率**（learning rate、ステップサイズ）$\eta > 0$ を用いてパラメータを次のように更新する。

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w}, b)}\, l^{(i)}(\mathbf{w}, b). \tag{2.9}$$

$\mathcal{B}_t$ は訓練セットから一様にサンプリングされるため、ミニバッチ勾配は真のフルバッチ勾配の**不偏推定量**（unbiased estimator）となり、その標準誤差（standard error）は $1/\sqrt{|\mathcal{B}|}$ のオーダーで減少する。二乗損失 Equation 2.4 のもとでの線形回帰では、$l^{(i)} = \frac{1}{2}(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)})^2$ を明示的に微分することで次の更新則が得られる。

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_t}\mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right),\\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_t}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right).\end{aligned} \tag{2.10}$$

各項の働きを見てみよう。スカラー $\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}$ は、例 i に対する符号付き予測誤差である。モデルが過大に予測した場合（$\hat{y}^{(i)} > y^{(i)}$）、更新によって $\mathbf{x}^{(i)}$ の定数倍が $\mathbf{w}$ から引かれ、b も下方へ修正されるため、次の予測値は $y^{(i)}$ に近づく。概念上の区別も重要である。$\mathbf{w}$ と b は最適化によって学習される**モデルパラメータ**であるのに対し、ミニバッチサイズ $|\mathcal{B}|$ と学習率 η は訓練開始前に選択される**ハイパーパラメータ**である（これらは 2.6 節で議論するように、別途取り置いた検証データ上で調整する）。

2.1.2.4 勾配ノイズと学習率

ミニバッチ SGD の軌跡（trajectory）はフルバッチ勾配降下法とどのように異なるだろうか。経験損失 $L(\mathbf{w})$ の最小二乗解 $\hat{\mathbf{w}}$ のまわりにおける等高線を考えよう（Figure 2.4）。フルバッチ勾配降下法は毎ステップで L の全 n 個の例にわたる勾配を計算し、楕円状の等高線を滑らかに横切って最小点 $\hat{\mathbf{w}}$ へと進む（ただし η が大きすぎて行き過ぎて発散しないものとする）。

ミニバッチ SGD はこれとは異なる経路をたどる。大域的最小点 $\hat{\mathbf{w}}$ では全 n 個の例にわたる平均勾配こそ 0 になるものの、ノイズを含む個々の例には依然として 0 でない残差 $\hat{y}^{(i)} - y^{(i)} \neq 0$ がある。無作為なミニバッチ $\mathcal{B}_t$ において、例ごとの勾配が互いに打ち消し合って 0 になることはまずない。その結果、学習率 η を一定に保つと、ミニバッチ SGD は $\hat{\mathbf{w}}$ の小さな近傍へ急速に降下したのち、$\hat{\mathbf{w}}$ の周囲にある**ノイズ球**（noise ball）の内部で永続的に揺らぎ続ける。この球の半径の二乗は $\eta/|\mathcal{B}|$ に比例する。

η を半分にするか $|\mathcal{B}|$ を 2 倍にすればこの終端のノイズ球は縮小し、訓練の過程で $\eta \to 0$ と減衰させれば反復列は $\hat{\mathbf{w}}$ に収束する（第 9 章）。しかし実際には、訓練損失の底を小数点以下 10 桁まで突き詰めることが望ましい場合はめったにない。最終的な目標は**未知の**データに対する誤差を小さくすることであり、$\hat{\mathbf{w}}$ 自体が真の母集団パラメータから $1/\sqrt{n}$ のオーダーの統計的推定誤差だけずれている以上、その統計的不確実性の範囲内で生じるわずかな最適化の揺らぎは汎化（generalization）において何のコストももたらさないからである。

2.1.2.5 高速化のためのベクトル化

ミニバッチ更新 Equation 2.10 を実装する際、例 $i \in \mathcal{B}_t$ を Python で 1 つずつループ処理してはならない。代わりに計算を**ベクトル化**し、ミニバッチ全体の順伝播（forward pass）と勾配ステップを配列・行列演算として表現することで、SIMD ベクトル演算器や GPU コア上で並列実行される C++/CUDA ライブラリへと処理を委ねる。

その違いを実際に確かめるため、$n = 1{,}000$ 個の例に対するラベル予測を、重み $\mathbf{w} = (2, -3.4)^\top$、バイアス $b = 4.2$ のもとでベンチマークしてみよう。まず Python の `for` ループで 1 例ずつ計算し、次に全 1,000 個の例を単一の行列積 `X @ w + b` で一度に計算する。

```
n = 1000
X = torch.ones(n, 2)               # the values do not affect the timing
w = torch.tensor([[2.0], [-3.4]])
b = 4.2
# Wait until the device has finished (CUDA runs asynchronously)
```

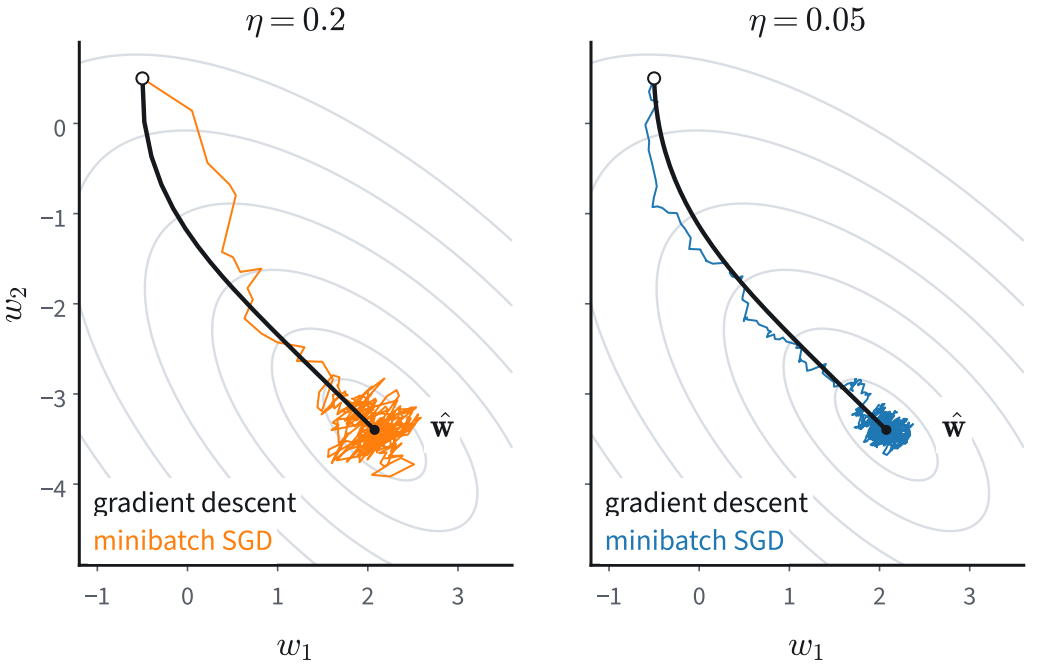


Figure 2.4: フルバッチ勾配降下法（黒）は L の勾配に沿って最小点 $\hat{\mathbf{w}}$ へ滑らかに進むが、1 ステップごとに全 n 個の例を 1 回走査する。ミニバッチ SGD（カラー）はノイズを含む勾配推定値を用いて 1 回の走査あたり $n/|\mathcal{B}|$ 回の低コストなステップを踏み、$\hat{\mathbf{w}}$ の周囲の小さなノイズ球へ急速に降下する。学習率を 4 分の 1 にすると（右）、この球は縮小する。

```
wait = lambda x: torch.cuda.synchronize() if torch.cuda.is_available() else None
```

計測用ヘルパ関数 seconds は、まずウォームアップ呼び出しを 1 回実行して（初回のみのセットアップや JIT コンパイルのコストを吸収するため）、計測終了前に wait を呼び出す。GPU やアクセラレータのフレームワークは演算を非同期にディスパッチするため、ハードウェアがまだ計算中であっても Python に制御を戻すことがあるからである。

```
def seconds(f):
    """Time one call of f, after a warm-up call."""
    wait(f())
    start = time.perf_counter()
    wait(f())
    return time.perf_counter() - start
```

```
loop = seconds(lambda: [X[i:i + 1] @ w + b for i in range(n)])
vectorized = seconds(lambda: X @ w + b)
print(f'one example at a time: {loop * 1e3:8.3f} ms')
print(f'all examples at once:  {vectorized * 1e3:8.3f} ms')
print(f'speedup: {loop / vectorized:.0f}x')
```

```
one example at a time:   11.358 ms
all examples at once:     0.024 ms
speedup: 482x
```

どちらの方法も同じ算術演算をしているにもかかわらず、全 1,000 個の予測を 1 回のベクトル化された呼び出しで計算する方が数百倍も高速である。ループでは 1 例ごとに Python インタプリタのオーバーヘッドとカーネル起動のレイテンシが発生するのに対し、行列積ではそのオーバーヘッドを 1 度支払うだけでバッチ全体をコンパイル済みの単一の線形代数ルーチンに引き渡せる。本書を通じて、ミニバッチ単位でベクトル化すること、すなわち n 個の内積 $\mathbf{w}^\top \mathbf{x}^{(i)}$ を 1 つの行列ベクトル積 $\mathbf{X}\mathbf{w}$ に置き換えることは、高速な深層学習コードを書くうえで最も重要な習慣である。

2.1.3 二乗損失とガウスノイズ

ここまでは、幾何学的な扱いやすさと導関数の簡潔さから二乗誤差損失 $\frac{1}{2}(\hat{y} - y)^2$ を動機づけてきた。ここで、Gauss が立てたより深い問いに進もう。すなわち、**データに関するどのような確率論的仮定のもとで、二乗誤差の最小化が統計的に最適な選択になるのだろうか。**

2.1.3.1 正規分布

平均 μ、分散 σ^2（標準偏差 σ）を持つ正規分布（normal distribution、ガウス分布とも呼ばれる）を振り返ろう。その確率密度（probability density）は次の式で与えられる。

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

この密度は μ を中心とする対称な釣鐘型の曲線を描き、その幅は標準偏差 σ によって決まる（確率の約 68% が平均から σ 以内に、約 95% が 2σ 以内に収まる）。偏差の二乗 $(x - \mu)^2$ が負の指数関数の内部に現れるため、極端な外れが生じる確率は無視できるほど小さい。たとえば平均から 5σ 離れた値の密度は、ピーク時の $e^{-12.5} \approx 4 \times 10^{-6}$ 倍にすぎない。正規分布に従う確率変

数（random variable）を $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ と表記する。多数の小さな独立した撹乱の和は正規分布へと収束するため（36.3.2.3 節）、性質のよい測定ノイズに対する自然な標準モデルとしてガウス分布が用いられる。

定式化すると、加法的ガウスノイズを伴う線形回帰では、観測される各ラベルが線形信号と独立な正規ノイズ項の和であると仮定する。

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.11}$$

このモデルのもとでは、条件付き平均 $\mathbb{E}[y \mid \mathbf{x}]$ は確かに $\mathbf{w}^\top \mathbf{x} + b$ となり（2.1.1.2 節と一致する）、ラベルは直線のまわりに同じノイズスケール σ で、どの入力 $\mathbf{x}$ においても対称にばらつく（Figure 2.5）。

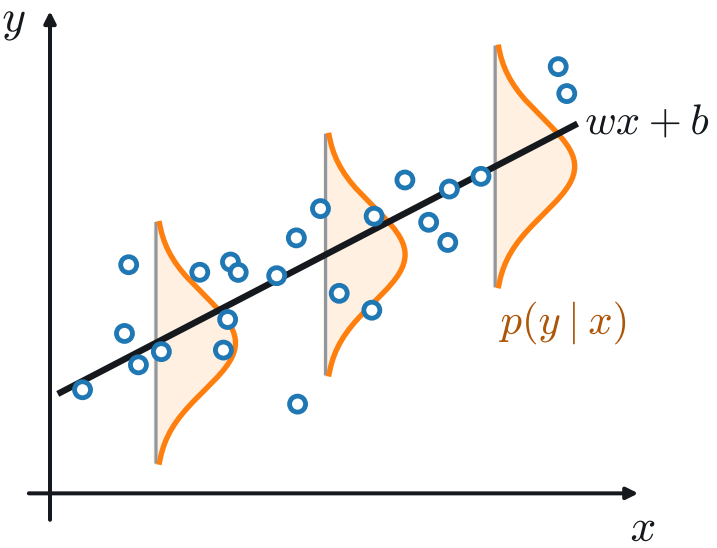


Figure 2.5: ガウスノイズモデル。どの入力 x に対しても、ラベル y は正規密度 $p(y \mid x)$（直線 $wx + b$ を中心とし、どこでも同じ幅を持つ）から抽出される。

2.1.3.2 最尤法

この加法的ガウスノイズモデルのもとでは、特徴量 $\mathbf{x}$ が与えられたときの観測ラベル y の条件付き分布（conditional distribution）は、平均 $\mathbf{w}^\top \mathbf{x} + b$、分散 σ^2 の正規分布となる。

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{2.12}$$

最尤法（maximum likelihood）の原理に従えば、$(\mathbf{w}, b)$ はデータセット全体 $\{y^{(1)}, \ldots, y^{(n)}\}$ が（$\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}\}$ を条件として）観測される条件付き確率密度を最大化するように選ぶべきである。各例が独立に抽出されると仮定すると、データセットの尤度（likelihood）は積 $\prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)})$ に分解される。正の積を最大化することは、その負の対数である**負の対数尤度**（negative log-likelihood）を最小化することと同値である。

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{2.13}$$

右辺の 2 つの項を詳しく見てみよう。第 1 項 $\frac{n}{2}\log(2\pi\sigma^2)$ は $\mathbf{w}$ にも b にもまったく依存しない。第 2 項は単なる $\frac{n}{\sigma^2}L(\mathbf{w}, b)$、すなわち平均二乗誤差（mean squared error）損失に正の定数 n/σ^2 を掛けたものである。したがって、二乗誤差損失を最小化すると、独立な等分散（homoscedastic）ガウスノイズのもとでの $(\mathbf{w}, b)$ の**最尤推定量**（maximum-likelihood estimator）が、σ^2 の値にかかわらず得られる。

2.1.3.3 ノイズモデルからの損失の選択

ノイズがガウス分布では**ない**とき、二乗損失は何をしているのだろうか。その答えに尤度は一切不要である。入力 $\mathbf{x}$ を固定し、どの単一の数値 c が期待二乗誤差 $\mathbb{E}[(y-c)^2 \mid \mathbf{x}]$ を最小化するかを問うてみよう。導関数 $2(c - \mathbb{E}[y \mid \mathbf{x}])$ を 0 とおくと、有限の分散を持つあらゆるノイズ分布に対して条件付き平均 $c = \mathbb{E}[y \mid \mathbf{x}]$ が得られる。一方、期待絶対誤差 $\mathbb{E}[|y-c| \mid \mathbf{x}]$ を最小化するのは条件付き**中央値**（median）である。これを非対称に傾けて、単位あたりの過小予測に q のコスト、単位あたりの過大予測に $1-q$ のコストがかかるようにすると、最小点は q 分位点（quantile）になる。平均よりも 90% 需要水準を必要とすることの多い予測実務家は、これを**ピンボール損失**（pinball loss）と呼ぶ。2.1.1.3 節の破損したラベルの例はまさにこの違いを示していた。平均はたった 1 つの巨大な値に引きずられるが、中央値はそれを無視する。したがって、損失を選ぶとは、実際のノイズがどのような分布であれ、y のどの統計量をモデルに予測させるかを選ぶことにほかならない。

さらに最尤法とのつながりから、ガウスノイズが現実の適切なモデルでは**ない**場合に損失関数を設計するための原理的な指針が得られる（Figure 2.6）。

出力の種類	ノイズモデル	損失（負の対数尤度）	予測される統計量
実数値、性質のよいノイズ	ガウス分布	二乗誤差（本節）	y の平均
実数値、外れ値または重い裾	ラプラス分布	絶対誤差（演習問題 2）	y の中央値
正値、乗法的な変動（価格）	$\log y$ 上のガウス分布	$\log y$ の二乗誤差（演習問題 3）	$\log y$ の平均
計数（販売数、到着数）	ポアソン分布	$\nu(\mathbf{x}) - y\log\nu(\mathbf{x})$（演習問題 4）	y の平均

- **裾の重い外れ値（ラプラスノイズ $\to \ell_1$ 損失）：**まれに生じる破損した測定値が極端な外れ値を生む場合、$(\hat{y}-y)^2$ は二次関数的に増大するため、たった 1 つの異常なデータ点が当てはめた直線を自分の方へ引き寄せてしまう。ラプラス分布 $p(\epsilon) = \frac{1}{2\tau}\exp(-|\epsilon|/\tau)$ を用いてより重い指数関数的な裾を持つノイズとしてモデル化すると、負の対数尤度は**絶対誤差**（absolute error、ℓ_1）損失 $|\hat{y}-y|$ となり（演習問題 2）、外れ値がどれほど遠くにあっても勾配の大きさは一定の ± 1 に保たれる。
- **滑らかな最適点と外れ値への頑健性の両立（フーバー損失）：**$|\hat{y}-y|$ は 0 に鋭い角を持ち、そこで導関数が -1 と $+1$ のあいだで不連続に跳ぶため、実務では**フーバー損失**（Huber loss）がよく用いられる。これは小さな残差（$|\hat{y}-y| \le \delta$）に対する二次関数と、大きな残差（$|\hat{y}-y| > \delta$）に対する線形の傾きを滑らかにつなぎ合わせた損失であり、外れ値への頑健性（robustness）と滑らかさを兼ね備える。

- **計数データ（ポアソンノイズ）**：目的変数 $y \in \{0, 1, 2, \ldots\}$ が離散的な計数（1 日あたりの来店客数など）である場合、ポアソン尤度からそれ固有の凸な負の対数尤度が導かれる（演習問題 4）。

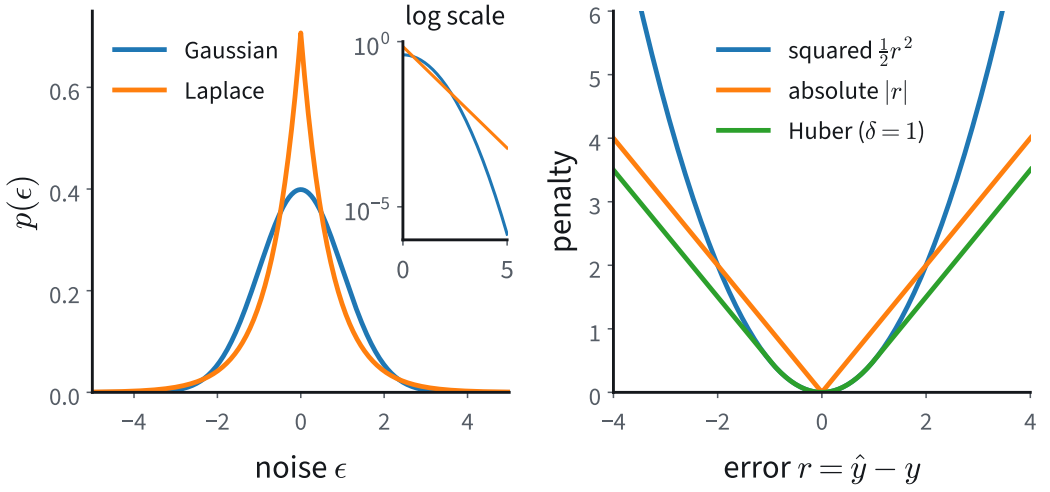


Figure 2.6: 3 つの回帰損失関数（右）と、そのうち最初の 2 つが最尤法のもとで対応するノイズ密度 $p(\epsilon) \propto \exp(-l(\epsilon))$（左）。二乗誤差（ガウス分布）と絶対誤差（ラプラス分布）のあいだにフーバー損失が位置する。対数スケールの挿入図はラプラス分布のより重い裾を示す。

2.1.4 ニューラルネットワークとしての線形回帰

線形回帰を後の章で扱う深層アーキテクチャへと結びつけるため、これを単層のニューラルネットワークとして描くことができる（Figure 2.7）。この図では、入力特徴量 $x_1, \ldots, x_d$ が**入力層**（input layer）のノードとして描かれ、それぞれ固有のシナプス重み $w_1, \ldots, w_d$ を介して**出力層**（output layer）の単一の計算ユニット o_1 へと結線されている。このユニットは重み付き入力の総和をとり、バイアス b を加えて $\hat{y} = o_1$ を出力する。

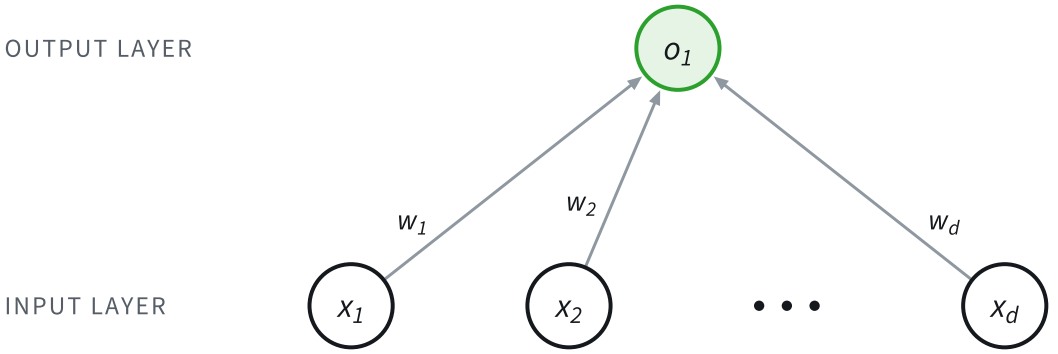


Figure 2.7: 線形回帰は単層のニューラルネットワーク（1 個の人工ニューロン）である。

慣例として、調整可能なパラメータを持つ層だけを数えるため、線形回帰は **1 層**のネットワークである。すべての入力特徴量が出力ノードに結合しているため、この層は**全結合層**（または**密結**

合層、dense layer）と呼ばれる。大脳皮質の個々のニューロンも同様に、樹状突起上の何千ものシナプスからの重み付き信号を統合している。この類推は McCulloch と Pitts のような初期の先駆者たちに着想を与えた (McCulloch and Pitts 1943)。もっとも、本書全体を通じた焦点は、そのような層を合成して深層ネットワークを構築する数学と工学にある。

2.1.5 まとめ

線形回帰は特徴量のアフィン結合 $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ によって連続値の目的変数を予測し、当てはまりの悪さを平均二乗誤差 $L(\mathbf{w}, b) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} + b - \mathbf{y}\|_2^2$ で評価する。幾何学的には、閉形式の正規方程式 $\mathbf{X}^\top\mathbf{X}\mathbf{w}^* = \mathbf{X}^\top\mathbf{y}$ は $\mathbf{y}$ を $\mathbf{X}$ の列空間へ直交射影するが、その $d \times d$ 連立方程式の構成と求解には $O(nd^2 + d^3)$ 回の演算コストがかかり、非線形ネットワークへは一般化できない。これに対しミニバッチ確率的勾配降下法は、小さな無作為バッチ $\mathcal{B}_t$ 上でのベクトル化された行列演算によってパラメータを更新し、1 ステップあたり $O(|\mathcal{B}|d)$ の計算量で最適点の周囲の小さなノイズ球へと急速に降下する。統計学的には、二乗誤差の最小化は加法的ガウスノイズのもとでの最尤推定にあたり、ガウスモデルを裾の重い分布や離散分布に置き換えれば、絶対誤差やポアソン負の対数尤度といった代替損失が導かれる。

2.1.6 演習問題

1. **切片と中心化**。すべての入力に定数の特徴量 1 を追加すると、アフィンモデル $\mathbf{w}^\top\mathbf{x} + b$ は $(\mathbf{x}, 1)$ に関する線形モデルになる。データの中心化は、これとは別の方法でバイアスを取り除く。訓練セット全体にわたる特徴量の平均とラベルの平均をそれぞれ $\bar{\mathbf{x}}$、$\bar{y}$ とする。
   1. 二乗損失のあらゆる最小点 $(\hat{\mathbf{w}}, \hat{b})$ が $\hat{b} = \bar{y} - \hat{\mathbf{w}}^\top\bar{\mathbf{x}}$ を満たすことを示し、バイアスを持つ最小二乗法の当てはめにおける誤差 $\hat{y}^{(i)} - y^{(i)}$ の総和が 0 になることを導け。
   2. $\hat{\mathbf{w}}$ が、中心化されたデータ $(\mathbf{x}^{(i)} - \bar{\mathbf{x}},\, y^{(i)} - \bar{y})$ に対するバイアスなしの最小二乗法の当てはめになっていることを示せ。
2. **ラプラスノイズ**。加法ノイズ ϵ がスケール 1 のラプラス分布（36.3 節を参照）、すなわち $p(\epsilon) = \frac{1}{2}\exp(-|\epsilon|)$ に従うと仮定する。
   1. このモデルのもとでのデータの負の対数尤度 $-\log p(\mathbf{y} \mid \mathbf{X})$ を書き下せ。
   2. 閉形式解が存在するかどうかを判定せよ。
   3. この損失のもとでの $(\mathbf{w}, b)$ に対するミニバッチ確率的勾配降下法の更新式を Equation 2.10 の形で書き下せ。ただし、微分値として 0（$|\cdot|$ が原点で導関数を持たない場合）と、それ以外での傾き $\operatorname{sign}(\cdot) = \pm 1$ を用いよ。このような代替物は**劣勾配**と呼ばれる。各例の寄与の大きさがその誤差にどう依存するかを、この損失と二乗損失とで比較せよ。
   4. [code] この更新則を 2.1.1.3 節の破損ラベルデータに対し、原点を通る直線について適用せよ。毎ステップでデータセット全体を用い、学習率を 0.002 として $w = 0$ から 2,000 ステップ実行せよ。絶対誤差の最小点は厳密に $w = 2$ である。反復列はどこに行き着くか。最小点に落ち着かない理由を説明し、改善策を提案せよ。
3. **ガウスノイズを超えて**。住宅価格や株価を回帰によって推定したいとする。
   1. 価格を直接 $y = \mathbf{w}^\top\mathbf{x} + b + \epsilon$（ただし $\epsilon \sim \mathcal{N}(0, \sigma^2)$）とモデル化し、$\sigma = 100{,}000$ ドルとする。予測価格が $50{,}000$ ドルの住宅に対して、このモデルが負の価格に割り当てる確率を計算せよ。単一の σ では、この住宅と予測価格が $5{,}000{,}000$ ドルの住宅の両方に適合しえない理由を説明せよ。
   2. 次に価格の対数に対して回帰し、$\log \text{price} = \mathbf{w}^\top\mathbf{x} + b + \epsilon$ とする。ここでは $\epsilon \sim \mathcal{N}(0, \sigma^2)$ が対数スケール上で作用する。価格の条件付き平均と条件付き標準偏差を計算し、両者の比が $\mathbf{x}$ に依存しないことを示せ。さらに、$\exp(\mathbf{w}^\top\mathbf{x} + b)$ が価格の不偏な予測になっているかどうかを判定せよ。これと同じモデル化の選択が、オプション価格評価のブラック・ショールズモデルの基礎にもなっている (Black and

Scholes 1973)。

4. **リンゴの個数を数える。**食料品店で売れるリンゴの**個数**を回帰によって推定したいとする。
   1. このような目的変数に対して加法的なガウスノイズモデルを用いる場合の問題点を挙げよ。
   2. 計数をモデル化するにはポアソン分布が用いられる。これは $p(k \mid \nu) = \nu^k e^{-\nu}/k!$ で与えられ、ここで $\nu > 0$ はレート、$k \in \{0, 1, 2, \dots\}$ は観測される計数である。$\mathbb{E}[k] = \nu$ を証明せよ。
   3. 計数 k の負の対数尤度を、特徴量に依存するレート $\nu(\mathbf{x})$ のもとで（パラメータに依存しない項を除いて）導出し、2.1.3.3 節の表と一致することを確かめよ。
   4. パラメータ化 $\nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ と $\log \nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ を比較せよ。それぞれについて、すべての入力とパラメータに対して $\nu(\mathbf{x}) > 0$ が成り立つかどうかを判定し、(c) の損失にどのような帰結をもたらすかを述べよ。また、2 番目のパラメータ化について、その損失を $(\mathbf{w}, b)$ の式として書き下せ。
5. [code] **ノイズ球の大きさ。**$n = 1{,}000$ 個の例を、$\mathcal{N}(0, 1)$ から抽出した 2 つの特徴量とラベル $y = \mathbf{x}^\top \mathbf{w}^* + \epsilon$（ただし $\mathbf{w}^* = (2, -3.4)$、$\epsilon \sim \mathcal{N}(0, 1)$）によって生成し、`1stsq` で最小二乗解 $\hat{\mathbf{w}}$ を計算せよ。
   1. バイアスなしのミニバッチ SGD を $\hat{\mathbf{w}}$ を初期値として、ミニバッチサイズ 10、$\eta = 0.1$ で 20,000 ステップ実行せよ。平均二乗距離 $\mathbb{E}\|\mathbf{w} - \hat{\mathbf{w}}\|^2$（反復列と $\hat{\mathbf{w}}$ のあいだの距離の二乗平均）を推定せよ。
   2. $\eta \in \{0.05, 0.025\}$、およびミニバッチサイズ 20 と 40 について同じ実験を繰り返し、平均二乗距離がおよそ $\eta / |\mathcal{B}|$ に比例することを確かめよ。
   3. この距離が、$\hat{\mathbf{w}}$ と $\mathbf{w}^*$ がどれだけ離れているかに依存しないのはなぜか。もしラベルにノイズがまったくなければ、何が変わるか。
6. **特異な計画行列。**●列フルランクであれば最小二乗法の最小点は一意に定まる。ここでは逆に $\mathbf{X}^\top \mathbf{X}$ が特異であるとする。
   1. $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ の最小点が依然として存在することを示し、すべての最小点からなる集合を特徴づけよ。
   2. 特異値分解を $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ と書く（ただし $\mathbf{U}$ と $\mathbf{V}$ は直交行列、Σ は非負の特異値を対角に持つ長方対角行列）。ムーア・ペンローズ擬似逆行列は次で定義される。

$$\mathbf{X}^\dagger = \mathbf{V}\Sigma^\dagger \mathbf{U}^\top, \tag{2.14}$$

      ここで $\Sigma^\dagger$ は Σ を転置し、その非ゼロ成分の逆数をとった行列である。$\mathbf{X}^\dagger \mathbf{y}$ が (a) の集合の中で最小ノルムを持つ最小点であることを示せ。
   3. $\mathbf{w} = \mathbf{0}$ から出発したとき、勾配降下法はどの最小点に収束するか。ミニバッチ確率的勾配降下法にした場合に答えが変わるかどうかも述べよ。

2.2 合成データ

実世界のデータセットは雑多であり、さらに厄介なことに「解答集」が付属していない。実際の住宅データで新しいアルゴリズムを訓練して重みベクトル $\hat{\mathbf{w}}$ を得たとしても、真のデータ生成パラメータ $\mathbf{w}^*$ や削減不能なノイズフロアを知る術はない。熟練した研究者は、実装を実データに投入する前に、まず正解（ground truth）が既知のモデルから生成した**合成データ**（synthetic data）でテストする。

合成データセットでは、特徴量の分布、真の重み $\mathbf{w}^*$ とバイアス b^*、標本サイズ n、およびノイズレベル σ を自由に制御できる。これにより、コードを直接検証する 2 つの手段が得られる。学習されたパラメータ $(\hat{\mathbf{w}}, \hat{b})$ が $(\mathbf{w}^*, b^*)$ に近づくかどうかをテストでき、また最終的な損失を理論上のノイズフロア $\frac{1}{2}\sigma^2$ と比較できる。本節では、続く 3 つの節を通して使用する合成データセットを構築し、その散布図を観察したうえで、基準となるベンチマークとして最小二乗法による当

てはめを計算する。

2.2.1 データの生成

手軽に可視化できるようにしつつ、複数の特徴量を扱う行列演算も実践するため、$n = 1{,}000$ 個の訓練例を $d = 2$ 個の特徴量で生成する。計画行列（design matrix）$\mathbf{X} \in \mathbb{R}^{1000\times 2}$ は各要素を独立な標準正規分布 $X_{ij} \sim \mathcal{N}(0,1)$ からサンプリングし、次の線形モデルからラベルを生成する。

$$\mathbf{y} = \mathbf{X}\mathbf{w}^* + b^* + \epsilon, \tag{2.15}$$

ここで、正解パラメータには次の値を用いる。

$$\mathbf{w}^* = (2, -3.4)^\top, \qquad b^* = 4.2, \tag{2.16}$$

さらに、加法的なガウスノイズ $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ を標準偏差 $\sigma = 0.01$ で加える。信号 $(\mathbf{X}\mathbf{w}^* + b^*)$ が数単位のスケールで変動するのに対し、ノイズ $\sigma = 0.01$ はその数百分の 1 の大きさしかない。したがって、正常に動作するオプティマイザであれば、$\mathbf{w}^*$ と b^* を数千分の 1 以内の誤差で復元できるはずである。

```
def synthetic_data(w, b, noise, n):
    """n examples with standard normal features and labels Xw + b + noise."""
    X = d2l.randn(n, len(w))
    eps = d2l.randn(n, 1) * noise
    return X, d2l.matmul(X, d2l.reshape(w, (-1, 1))) + b + eps
```

2,000 個の例を、$\mathbf{w}^* = (2, -3.4)$、$b^* = 4.2$、$\sigma = 0.01$ を用いて生成しよう。最初の 1,000 個は訓練用に確保し、残りの 1,000 個は検証用に残しておく。検証例は訓練例とまったく同じ分布から得られるが、モデルの当てはめ中には一切使用しない。そのため、検証例で損失を評価すれば、モデルが未知のデータをどれほど正確に予測できるかがわかる。この区別については 2.6 節で詳しく検討する。

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = synthetic_data(w_star, b_star, noise, 2000)
```

`X` の各行には 2 次元の特徴ベクトルが格納され、`y` の各行には対応するスカラーのラベルが格納されている。最初のデータ例を調べ、そのノイズ $\epsilon^{(1)} = y^{(1)} - (\mathbf{w}^{*\top}\mathbf{x}^{(1)} + b^*)$ を逆算してみることで、データが正しく生成されているかを手早く確認できる。

```
x, label = X[0], y[0, 0]
eps = label - (d2l.reduce_sum(w_star * x) + b_star)
print(f'features: ({float(x[0]):.4f}, {float(x[1]):.4f})')
print(f'label: {float(label):.4f}, noise: {float(eps):.4f}')
```

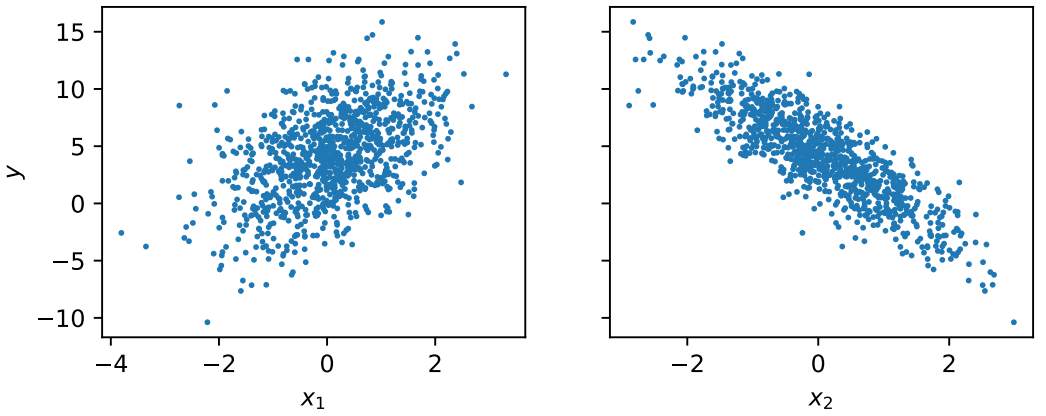
```
features: (1.2794, 1.2895)
label: 2.3786, noise: 0.0042
```

最初の例について手計算で確かめてみると、$2x_1^{(1)} - 3.4x_2^{(1)} + 4.2$ は $y^{(1)}$ と、微小な $\mathcal{N}(0, 0.01^2)$ の摂動の範囲内で一致する。

2.2.2 データの観察

モデルを当てはめる前に、ラベル y を各特徴量 x_1、x_2 に対して個別にプロットしてみよう。

```
d2l.set_figsize((7, 2.5))
_, axes = d2l.plt.subplots(1, 2, sharey=True)
for j, ax in enumerate(axes):
    ax.scatter(d2l.numpy(X[:1000, j]), d2l.numpy(y[:1000, 0]), s=2)
    ax.set_xlabel(f'$x_{j + 1}$')
    ax.set_ylabel('$y$' if j == 0 else '')
```



どちらの散布図にも、予想どおりの符号を持つ明確な線形トレンドが現れている。y は x_1 に伴って上昇し（傾き $+2$）、x_2 に伴ってより急激に下降する（傾き -3.4）。$\sigma = 0.01$ はこれほど小さいのに、なぜ各 2D 散布図はかみそりの刃のように細い直線ではなく、太い帯のように見えるのだろうか。それは、y を x_2 だけに対してプロットすると、各点における x_1 の変動が無視されるからである。プロットされていない項 $2x_1$ は標準偏差 2 を持つため、点の縦方向の位置が数単位の幅を持つ帯状に広がるのである。多変数の関係を周辺化した 2D 射影として観察すると、プロットされていない座標の変動が常に見かけ上の縦方向のばらつきとして折り畳まれる。

2.2.3 パラメータの復元

2.1.2.1 節で閉じた形の正規方程式を導出したので、1,000 個の例に対する最小二乗法による厳密解 $(\hat{\mathbf{w}}, \hat{b})$ を、`lstsq` を拡大計画行列 $[\mathbf{X}, \mathbf{1}]$ に適用して早速計算してみよう。これにより、2.3 節のミニバッチ SGD が一致を目指す基準（有限標本における理想的な当てはめ）が得られる。

```
n = 1000                                        # The training examples
X1 = torch.cat((X[:n], torch.ones(n, 1)), 1)    # A column of ones for b
w_hat = torch.linalg.lstsq(X1, y[:n]).solution[:, 0]
```

ここで、実際の推定誤差を、理論的に予測される標準誤差 $\sigma/\sqrt{n}$ と並べて出力する。

```
e_w = d2l.numpy(w_hat[:2] - w_star)
print(f'error in w: ({e_w[0]:.1e}, {e_w[1]:.1e})')
print(f'error in b: {float(w_hat[2] - b_star):.1e}')
print(f'standard error: {noise / n ** 0.5:.1e}')
```

```
error in w: (3.3e-04, 5.7e-04)
error in b: 1.1e-04
standard error: 3.2e-04
```

`lstsq` で経験損失を機械精度まで解いたとしても、$(\widehat{\mathbf{w}}, \widehat{b})$ は真の値 $(2, -3.4, 4.2)$ と小数第 4 位で異なる。このわずかなずれは数値誤差ではなく、**統計的推定誤差**（statistical estimation error）である。1,000 個のノイズの実現値 ϵ がたまたま $\mathbf{X}$ の各列とわずかに相関し、最小二乗平面を $\sigma/\sqrt{n} = 0.01/\sqrt{1000} \approx 3 \times 10^{-4}$ 程度の大きさだけ傾けるのである（演習問題 1）。これはソルバではなくデータの限界である。1 回の実行で偶然より近い値になることはあっても、ノイズを含むこれら 1,000 個の点から計算されるいかなる不偏推定量（unbiased estimator）も、これより小さい期待二乗誤差は達成できない（**クラメール・ラオの下界**、Cramér–Rao bound）。また、演習問題 2 で調べるように、真のデータ生成関数が非線形である場合、最小二乗法は特徴量の分布上でその関数に対する最良の線形近似を求める。

2.2.4 まとめ

既知の線形モデル $\mathbf{y} = \mathbf{X}\mathbf{w}^* + b^* + \epsilon$ から、$\mathbf{w}^* = (2, -3.4)^\top$、$b^* = 4.2$、$\sigma = 0.01$ を用いて合成データを生成すると、実データにはない 2 つの定量的検証手段が得られる。すなわち、$\frac{1}{2}\sigma^2 = 5 \times 10^{-5}$ という削減不能な損失の下限と、既知の正解パラメータである。y を単一の座標 x_j に対してプロットすると、プロットされていない特徴量の変動が縦方向の広がりに折り畳まれ、ほぼノイズのない平面が幅広い 2D の帯に変わる。最小二乗法の厳密ソルバであっても、$(\mathbf{w}^*, b^*)$ からオーダー $\sigma/\sqrt{n}$（$\approx 3 \times 10^{-4}$、$n = 1{,}000$ の場合）の統計的推定誤差だけずれるため、これが次節における反復的な最適化の精度目標となる。

2.2.5 演習問題

1. [code] **ノイズ下での復元。**●以下の各設定について、2.2.3 節の最小二乗法による当てはめを 1,000 個の独立なデータセット上で繰り返し、最初の重みの推定誤差 $\widehat{w}_1 - w_1^*$ の標準偏差を計測せよ。
   1. 1,000 個の訓練例を用いて `noise` を $\{0.001, 0.01, 0.1, 1\}$ の範囲で変化させ、次に訓練例の数 n を $\{10, 100, 1{,}000, 10{,}000\}$ の範囲で `noise=0.01` のもと変化させよ。両方の計測値を、理論予測 $\sigma/\sqrt{n}$ に対して両対数軸でプロットせよ。
   2. n のうち、計測値が $\sigma/\sqrt{n}$ と 10% 以内で一致する最小の値を求め、それより少ない例数ではこの予測の背後にあるどの近似が破綻するかを述べよ。
   3. $\tilde{\mathbf{X}} = [\mathbf{X}, \mathbf{1}]$、$\tilde{\mathbf{w}}^* = (\mathbf{w}^*, b^*)$、$\widehat{\tilde{\mathbf{w}}} = (\widehat{\mathbf{w}}, \widehat{b})$ とおく。正規方程式から $\widehat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^* = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^\top\epsilon$ を示し、そこから $\mathrm{Cov}(\widehat{\tilde{\mathbf{w}}}) = \sigma^2(\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}$ を導出せよ。次に、相関を持つ特徴量 $x_2 = \rho x_1 + \sqrt{1-\rho^2}\, z$（ただし x_1 と z は独立な標準正規分布に従う）を $\rho \in \{0, 0.9, 0.99\}$ に対して生成せよ。この式から $\widehat{w}_1$ の標準誤差を予測して数値的に確認し、重みが不正確であるにもかかわらず、同じ分布から得られた新しい例に対する予測 $\widehat{\tilde{\mathbf{w}}}^\top \tilde{\mathbf{x}}$（ただし $\tilde{\mathbf{x}} = (\mathbf{x}, 1)$）が正確であり続ける理由を説明せよ。
2. [code] **適合しないモデル。**●すべてのラベルに項 $c\,x_1^2$（$c = 0.5$）を加えてラベルが特徴量の線形関数に従わないようにし、本節と同様に最小二乗法で線形モデルを当てはめよ。
   1. $\mathbb{E}[x_1^2] = 1$ および $\mathbb{E}[x_1^3] = 0$ を用いて、$\widehat{w}_1$、$\widehat{w}_2$、$\widehat{b}$ のうちどれが真の値から離れるか、またどの程度離れるかを予測せよ。
   2. $n \in \{100, 1{,}000, 10{,}000\}$ 個の訓練例に対してその予測を確認せよ。n が大きくなるにつれて縮小する推定誤差はどれで、縮小しないものはどれか。また、縮小する誤差は $\sigma/\sqrt{n}$ と比べてどうか。
   3. 標準誤差の約 3 倍にあたる 10^{-3} を超えるパラメータ誤差を検出するテストは、本節のデータ上では正しい実装を合格と判定する。このデータ上では、正しい実装に欠陥

があると判定するだろうか。モデルがデータに適合していないとき、このようなテストの合否から何がわかるかを説明せよ。

3. **損失の下限とその変動。**次節では、1,000 個の検証例にわたる $\frac{1}{2}(\hat{y}-y)^2$ の平均である検証損失によって訓練の実行結果を評価する。
   1. 真のパラメータを持つモデルが新しい例に対して期待損失 $\sigma^2/2$ を持ち、他のいかなるパラメータもそれより小さい期待損失を持たないことを示せ。これは、正しい訓練の実行が最終的に落ち着く下限である。
   2. $\epsilon \sim \mathcal{N}(0,\sigma^2)$ に対して $\mathbb{E}[\epsilon^4] = 3\sigma^4$ が成り立つ。1 つの例の損失 $\frac{1}{2}\epsilon^2$ の標準偏差を計算し、そこから $n' = 1{,}000$ 個の例にわたる真のパラメータの検証損失の標準誤差を、下限に対する比として求めよ。
   3. [code] 多数の独立な検証セット上で真のパラメータを評価し、両方の数値を確かめよ。次に、小問 (a) の結果を用いて、期待損失を検証損失の標準誤差 1 つ分だけ増加させる最小のパラメータ誤差を求め、前の演習問題におけるパラメータ誤差の閾値 10^{-3} と比較せよ。

2.3 線形回帰のスクラッチ実装

線形回帰の理論（2.1 節）と、既知の正解パラメータを持つ合成データセット（2.2 節）が揃ったところで、いよいよこの両方をコードに落とし込んでいこう。本節では、基本的なテンソル演算と自動微分（automatic differentiation）のみに頼り、パラメータの初期化、線形の順伝播、二乗損失、自動微分、ミニバッチのイテレータ、SGD の更新ループといった訓練パイプラインのすべての構成要素をスクラッチで構築する。各ステップを手書きすることでブラックボックス化された仕組みが取り払われ、勾配がどのように流れ、学習率などのハイパーパラメータが収束をどのように左右するかが明確に見えてくる。全体の仕組みが明らかになった後、2.4 節ではこれらの構成要素を再利用可能なクラス（`Module`、`DataModule`、`Trainer`）へと整理し、さらに 2.5 節ではフレームワークの層、組み込みの損失関数、オプティマイザを用いて同じモデルをわずか数行で書き直す。

2.3.1 モデル、損失、勾配

まず、2,000 個の合成データ例（訓練用に 1,000 個、検証用に 1,000 個）を、真のパラメータ $\mathbf{w}^* = (2, -3.4)^\top$、$b^* = 4.2$、ノイズ $\sigma = 0.01$ で生成しよう。

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = d2l.synthetic_data(w_star, b_star, noise, 2000)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

2.3.1.1 パラメータ

このモデルは、重みベクトル $\mathbf{w} \in \mathbb{R}^2$ とスカラーのバイアス $b \in \mathbb{R}$ を持つ。重みは $\mathcal{N}(0, 0.01^2)$ からの小さな乱数で初期化し、バイアスは 0 に設定する（線形回帰ではすべて 0 から始めても同様に機能するが、これについては演習問題 1 を参照のこと。ただし、第 4 章で隠れ層を積み重ねるようになると、ランダムな重みで対称性を破ることが不可欠になる）。

両方のテンソルに `requires_grad=True` を設定し、これらに関わるすべての演算を PyTorch の autograd エンジンが記録するようにする。

```
def init_params(sigma=0.01):
    w = torch.normal(0, sigma, (2, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return w, b
```

2.3.1.2 順伝播

ミニバッチの計画行列（design matrix）$\mathbf{X} \in \mathbb{R}^{|\mathcal{B}| \times d}$、パラメータ $\mathbf{w} \in \mathbb{R}^{d \times 1}$、およびスカラーのバイアス b が与えられたとき、線形予測 $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ は 1 回の行列とベクトルの積にブロードキャストによる加算を足し合わせたものになる。

```
def linreg(X, w, b):
    """The linear regression model."""
    return d2l.matmul(X, w) + b
```

2.3.1.3 損失

次に、平均二乗誤差（mean squared error）$L(\mathbf{w}, b) = \frac{1}{2|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\hat{y}^{(i)} - y^{(i)})^2$ を実装する。`y` を列ベクトル (`-1, 1`) に変形して `y_hat` の形状に揃える。もし一方のテンソルの形状が (`n, 1`) で他方が (`n,`) だった場合、両者を減算すると暗黙のうちにブロードキャストされ、すべてのペアごとの差からなる (`n, n`) 行列になってしまう。このバグが厄介なのは、訓練自体はエラーにならずに進んでしまう点である。n^2 個すべての二乗差の平均は、すべての予測がラベルの平均に等しいときに最小となるため、SGD は $\mathbf{w}$ を $\mathbf{0}$ へ、b を $\bar{y}$ へと向かわせ、損失はラベルの分散の半分（このデータでは $\frac{1}{2}(2^2 + 3.4^2) \approx 7.8$）まで滑らかに減少する。したがって、損失曲線が下がっているだけでは、コードが正しいことの弱い証拠にしかならない。本節の後半では、損失が到達すべき値に達しているか、またパラメータが真の値と一致しているかを検証する。

```
def squared_loss(y_hat, y):
    """Half the squared error, averaged over the minibatch."""
    return d2l.reduce_mean((y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2)
```

2.3.1.4 勾配

自動微分に処理を任せる前に、32 個の例からなる 1 つのミニバッチを用いて、自動微分が 2.1.2.3 節で手計算により導出した次の式と同じ勾配を算出することを確認しておこう。

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{1}{|\mathcal{B}|} \mathbf{X}^\top (\hat{\mathbf{y}} - \mathbf{y}), \qquad \frac{\partial L}{\partial b} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\hat{y}^{(i)} - y^{(i)}). \tag{2.17}$$

最初の 32 個の訓練例 (`Xb, yb`) をスライスして取り出し、解析的な勾配と自動微分による勾配を比較してみよう。

```
Xb, yb = X_train[:32], y_train[:32]
w, b = init_params()
```

```
    squared_loss(linreg(Xb, w, b), yb).backward()
    with torch.no_grad():
        r = linreg(Xb, w, b) - yb               # One error per example
        by_hand = torch.cat([Xb.T @ r, r.sum(0, keepdim=True)]) / len(yb)
        by_autograd = torch.cat([w.grad, b.grad[None]])
    print('gradient:', d2l.numpy(by_autograd).reshape(-1).round(4))
    print(f'largest difference: {float(abs(by_hand - by_autograd).max()):.1g}')
```

```
gradient: [-4.9511  7.7252 -4.6794]
largest difference: 0
```

要素ごとの最大の差は 0（高々 $\sim 10^{-7}$、すなわち `float32` の丸め精度）であり、autograd が期待どおりのミニバッチ勾配を計算していることが確認できる。ここからは、順伝播と損失の微分をフレームワークに自動で任せることにする。

2.3.2 スクラッチからの訓練

2.3.2.1 更新

各ステップでは、$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta\,(\nabla_{\mathbf{w}} L, \nabla_b L)$ のようにパラメータをその勾配と逆方向に更新する。このとき、パラメータ更新そのものを計算グラフ（computational graph）に記録しないよう注意する。

2.3.2.2 ミニバッチ

データセット全体をサイズ $|\mathcal{B}|$ のランダムなミニバッチで一巡するため、ジェネレータ `minibatches` は各エポックの開始時にインデックス $0, \ldots, n-1$ の新しいランダムな順列を生成し、サイズ `batch_size` の連続するスライスを順に返す（演習問題 2）。

```
def minibatches(X, y, batch_size):
    """Minibatches of (X, y), in a new random order on every call."""
    indices = np.random.permutation(len(X))
    for i in range(0, len(indices), batch_size):
        batch = indices[i : i + batch_size]
        yield X[batch], y[batch]
```

1,000 個の訓練例と `batch_size=32` の場合、1 エポックで $\lceil 1000/32 \rceil = 32$ 個のミニバッチが得られる。内訳は、フルサイズのバッチが 31 個（各 32 例）と、端数のバッチが 1 個（8 例）である。このジェネレータを 1 回走査して中身を確認し、誤って**同じ**ジェネレータを 2 回目に反復処理するとどうなるか見てみよう。

```
loader = minibatches(X_train, y_train, 32)
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
```

```
32 minibatches, the last of size 8
second pass: 0 minibatches
```

返される各ペア `(Xb, yb)` の形状は `(32, 2)` と `(32, 1)` である。1 回目の走査（1 **エポック**）では期待どおり 32 個すべてのミニバッチが返されるが、2 回目の走査では 0 個になる。Python のジェ

ネレータは使い切りであり、一度消費されると何も警告を出さずに空になる。訓練ループで各エポックの開始時に毎回新しく `minibatches(...)` を呼び出すのはこのためである。

2.3.2.3 訓練ループ

ここで、完全な訓練ループを `train(lr, num_epochs, batch_size)` として組み立てよう。各エポックでは `minibatches(X, y, batch_size)` を反復処理し、ミニバッチの損失を計算して `w` と `b` に関して微分し、SGD のステップを適用する。

```
def train(lr, num_epochs=10, batch_size=32):
    w, b = init_params()
    val_losses = []
    for epoch in range(num_epochs):
        for Xb, yb in minibatches(X_train, y_train, batch_size):
            loss = squared_loss(linreg(Xb, w, b), yb)  # forward pass, loss
            loss.backward()                            # backward pass
            with torch.no_grad():                      # not recorded:
                for param in (w, b):
                    param -= lr * param.grad           # update
                    param.grad.zero_()                 # clear
        with torch.no_grad():
            val_losses.append(squared_loss(linreg(X_val, w, b), y_val).item())
    return w, b, val_losses
```

PyTorch は新しい勾配を保存済みの勾配に上書きせず加算するため（1.5 節）、ループの最後で各勾配をクリアする。

中核となる訓練ループはわずか 12 行ほどのコードにすぎず、この同じ 4 つのステップ（順伝播 $\rightarrow$ 損失 $\rightarrow$ 逆伝播 $\rightarrow$ パラメータ更新、Figure 2.8 に図示）が、本書の残りの章で扱うあらゆる深層ネットワークを訓練する。

2.3.3 実装の検証

2.3.3.1 損失の期待値

訓練が成功したとき、損失はどこまで小さくなるべきだろうか。真のパラメータ $(\mathbf{w}^*, b^*)$ においてさえ、残差はガウスノイズ $\hat{y}^{(i)} - y^{(i)} = -\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ に等しいため、損失の期待値は次の**削減不能なノイズフロア**（irreducible noise floor）となる。

$$\mathbb{E}\left[L(\mathbf{w}^*, b^*)\right] = \frac{1}{2}\,\mathbb{E}\left[\epsilon^2\right] = \frac{1}{2}\sigma^2 = \frac{1}{2}(0.01)^2 = 0.00005. \tag{2.18}$$

対照的に、初期化時点（$\mathbf{w} \approx \mathbf{0}, b = 0$）での残差は $-y = -(\mathbf{x}^\top \mathbf{w}^* + b^* + \epsilon)$ であり、その平均は $-b^* = -4.2$、分散は $\|\mathbf{w}^*\|_2^2 + \sigma^2 = 2^2 + (-3.4)^2 + 0.01^2 = 15.5601$ である。したがって、初期損失の期待値は次のようになる。

$$\mathbb{E}\left[L(\mathbf{0}, 0)\right] = \frac{1}{2}\left((b^*)^2 + \|\mathbf{w}^*\|_2^2 + \sigma^2\right) = \frac{1}{2}(17.64 + 15.5601) \approx 16.6. \tag{2.19}$$

したがって、訓練を通じて損失は約 16.6 からノイズフロアの 0.00005 まで、5 桁にわたって急落すると予想される。

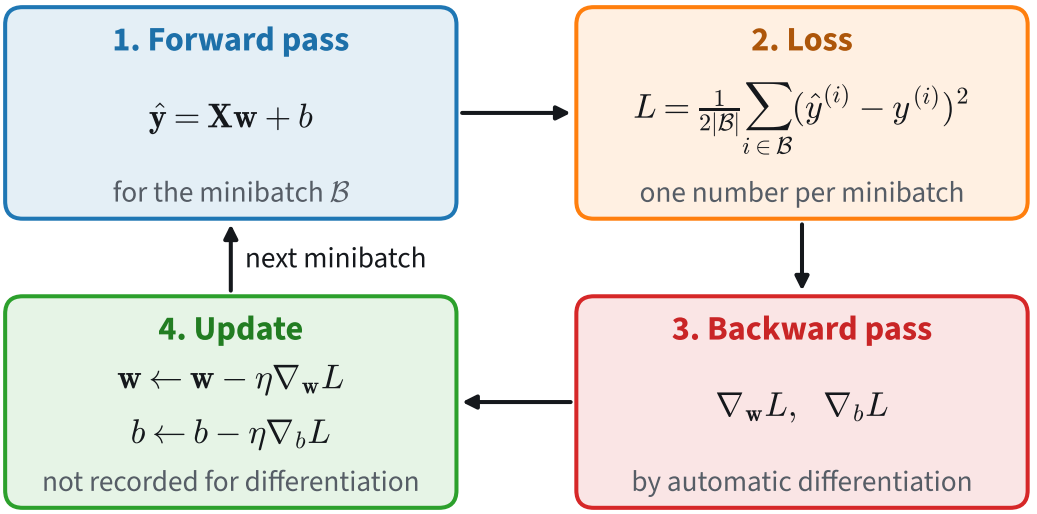


Figure 2.8: ミニバッチ SGD の 1 ステップ。順伝播でミニバッチに対する予測を計算し、損失でそれを 1 つの数値に集約し、逆伝播でその数値をパラメータに関して微分し、更新ステップでパラメータを勾配と逆方向に動かす。その後、次のミニバッチでこのサイクルを繰り返す。

同じ計算から、損失がどのくらい**速く**減少するかも予測できる。任意のパラメータにおいて、残差は $r = \mathbf{x}^\top(\mathbf{w} - \mathbf{w}^*) + (b - b^*) - \epsilon$ となる。2 つの特徴量は互いに独立に標準正規分布に従い、ノイズもその両方と独立であるため、すべての交差項の期待値は 0 となり、次式が成り立つ。

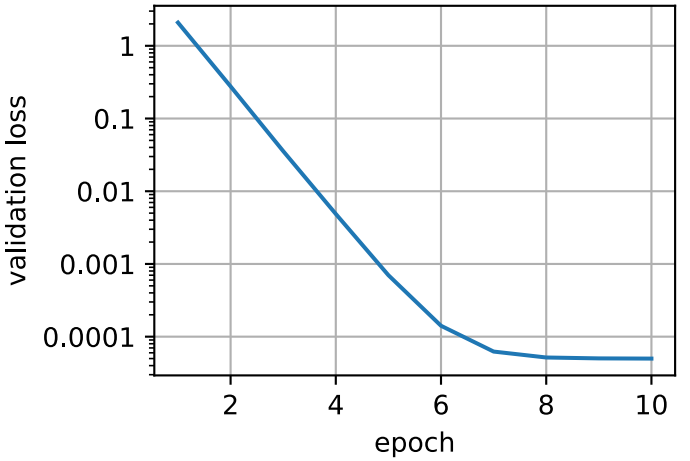
$$\mathbb{E}\left[L(\mathbf{w}, b)\right] = \frac{1}{2}\|\mathbf{w} - \mathbf{w}^*\|_2^2 + \frac{1}{2}(b - b^*)^2 + \frac{1}{2}\sigma^2. \tag{2.20}$$

この損失の期待値の $(\mathbf{w}, b)$ に関する勾配は、単に**パラメータ誤差**（parameter error）$(\mathbf{w} - \mathbf{w}^*, b - b^*)$ となる。そのため、サイズ η の勾配ステップによってパラメータ誤差は $1 - \eta$ 倍に縮小し、ノイズフロアを上回る**過剰損失**（excess loss）は $(1 - \eta)^2$ 倍に縮小する。ミニバッチ勾配が期待勾配と一致するのは平均的な意味においてのみであるため、これは厳密な恒等式ではなく近似的な軌跡を与える。$\eta = 0.03$ のとき、32 回のミニバッチ更新からなる 1 エポックで、過剰損失は約 $1/(1 - 0.03)^{64} \approx 7$ 分の 1 に減少するはずである。

2.3.3.2 訓練の実行

`train` を 10 エポックにわたり学習率 $\eta = 0.03$、ミニバッチサイズ $|\mathcal{B}| = 32$ で実行し、各エポック後の検証損失を対数スケールでプロットしてみよう。

```
w, b, val_losses = train(lr=0.03)
d2l.plot(list(range(1, 11)), val_losses, xlabel='epoch',
         ylabel='validation loss', yscale='log')
```



予想どおり、検証損失は直線に沿って急落し、エポックごとに約 7× の割合で減少した後、7〜8 エポック目あたりで予測された下限である 5×10^{-5} 付近に収まって横ばいになる。最初の 7 エポックにわたり、実測された過剰損失を閉形式による予測とエポックごとに比較してみよう。

```
floor = noise ** 2 / 2                              # sigma^2 / 2
excess = (float(d2l.reduce_sum(w_star ** 2)) + b_star ** 2) / 2  # 16.6
for epoch, loss in enumerate(val_losses[:7], start=1):
    excess *= (1 - 0.03) ** (2 * 32)                # 32 steps per epoch
    print(f'epoch {epoch}: measured {loss - floor:.1e}, '
          f'predicted {excess:.1e}')
```

```
epoch 1: measured 2.1e+00, predicted 2.4e+00
epoch 2: measured 2.8e-01, predicted 3.4e-01
epoch 3: measured 3.6e-02, predicted 4.8e-02
epoch 4: measured 4.8e-03, predicted 6.8e-03
epoch 5: measured 6.5e-04, predicted 9.7e-04
epoch 6: measured 9.1e-05, predicted 1.4e-04
epoch 7: measured 1.2e-05, predicted 2.0e-05
```

実測された過剰損失と予測された過剰損失は、序盤のエポックでは緊密に一致して推移し、その後少しずつ離れていくものの、7 エポック目でも高々 3〜4 倍程度の控えめな差にとどまる。なぜこのようなわずかなドリフトが生じるのだろうか。ここでの理想化した予測では、標本共分散行列（sample covariance matrix）$\mathbf{H} = \frac{1}{n}\sum_i \tilde{\mathbf{x}}^{(i)}(\tilde{\mathbf{x}}^{(i)})^\top$（演習問題 1 を参照）が厳密に単位行列に等しいと仮定していた。1,000 個の例からなる有限標本では、その固有値はおよそ 10% の範囲内で 1 の近くに収まっており、数エポックにわたってこの小さなずれが積み重なることで、実際の訓練は理想化した予測よりわずかに先行したり遅れたりするのである。減衰率とノイズフロアはいずれも理論と一致している。

2.3.3.3 復元されたパラメータ

このデータセットは合成データであるため、学習された $(\mathbf{w}, b)$ を真のパラメータ $\mathbf{w}^* = (2, -3.4)^\top$ および $b^* = 4.2$ と直接比較することもできる。

```
    w_error = d2l.numpy(d2l.reshape(w, w_star.shape) - w_star)
    b_error = d2l.numpy(b - b_star)
    print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
    print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: 1.7e-04, 5.1e-04
error in b: -4.6e-04
```

重みとバイアスはいずれも 1,000 分の 1 以内の誤差で復元されており、これは統計的推定の下限 $\sigma/\sqrt{n} \approx 3 \times 10^{-4}$（2.2 節で最小二乗法のソルバを用いて確認した値）と一致している。

2.3.3.4 学習率

学習率の選び方が適切でなかった場合はどうなるだろうか。まったく同じ 10 エポックのループを、3 桁にわたる 4 つの学習率 $\eta \in \{0.003, 0.03, 0.3, 3\}$ で実行し、それぞれの最終的な検証損失を表示してみよう（演習問題 1）。

```
    for lr in [0.003, 0.03, 0.3, 3]:
        val_loss = train(lr)[2][-1]
        print(f'lr = {lr:<6} validation loss: {val_loss:.1e}')
```

```
lr = 0.003  validation loss: 2.1e+00
lr = 0.03   validation loss: 5.0e-05
lr = 0.3    validation loss: 5.1e-05
lr = 3      validation loss: nan
```

結果はすべてのケースで理論と一致している。$\eta = 0.003$ では、誤差が 1 ステップあたり 0.997 倍にしかならないため、10 エポック後の損失は 2 から 3 付近にとどまる（$16.6 \times 0.997^{640} \approx 2.4$ のすぐ近くである）。$\eta = 0.3$ では、最初の 1 エポックのうちにモデルが一気にノイズフロアまで到達する。そして $\eta = 3$ では、更新のたびに誤差が符号を反転させながら約 2 倍に膨れ上がり、たちまちオーバーフローして nan になる。なお、安定性の上限である 2 という値は SGD の普遍的な定数ではなく、ここでの**データのスケーリング**に由来する性質であることに注意したい。これは、特徴量の分散が 1 であることから生じている。ある特徴量を $s > 1$ 倍にスケーリングすると、対応する重み方向の損失の曲率は s^2 倍になり、安定な最大学習率は約 $2/s^2$ まで縮小する。これが、訓練前に入力特徴量を標準化する実用上の大きな理由の 1 つである（1.2 節）。

続く 2.4 節では、今構築した訓練の仕組みを再利用可能なオブジェクト指向のクラス（`Module`、`DataModule`、`Trainer`）にまとめ、2.5 節では手書きのテンソル、損失、更新処理をフレームワークの既製コンポーネントに置き換える方法を示す。

2.3.4 まとめ

完全な訓練ループに必要な基本要素は、パラメータ化された順伝播 $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$、スカラーの損失 $\frac{1}{2|\mathcal{B}|}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$、リバースモード自動微分（reverse-mode automatic differentiation）、そして勾配の記録対象外で実行するミニバッチ SGD の更新という 4 つだけである。損失曲線が下がっているだけではコードにバグがないことの証明にはならないため（暗黙の形状の不一致があっても、損失はラベルの分散の半分に向かって下がりうる）、合成データを用いて実装を定量的に検証した。すなわち、検証損失は予測されたエポックあたり $(1-\eta)^{64}$ の割合でノイズフロア $\frac{1}{2}\sigma^2 = 5 \times 10^{-5}$ まで指数関数的に減衰し、学習されたパラメータは $(\mathbf{w}^*, b^*)$ と $O(\sigma/\sqrt{n})$ の範囲内で一致する。学習率を変えると、小さなステップサイズでは進行が遅く、適度なステップ

サイズでは素早くノイズフロアに到達し、$2/\mu_{\max}$ を超えるステップサイズでは二次関数の谷を飛び越えて発散する。

2.3.5 演習問題

1. [code] **極端な設定での初期化**。`init_params` は標準偏差 `sigma=0.01` で重みを生成する。重みを厳密に 0 で初期化した場合と、`sigma=1000` とした場合のそれぞれについて、訓練がノイズフロアに到達するかどうか予測せよ。Equation 2.20 を用いて、学習率 0.03 のもとで 2 番目のケースが何エポック多く必要になるかを見積もり、その見積もりを検証せよ。

2. [code] **二階導関数**。ミニバッチ損失の 3×3 ヘッセ行列を $\theta = (\mathbf{w}, b)$ に関して手計算で求め、それがどの量に依存するかを述べよ。次に、合成データの固定した 1 つのミニバッチに対して、初期化時点と訓練後の 2 回、`torch.func.hessian`（1.5 節）を用いた自動微分によってヘッセ行列を計算せよ。両方の結果を導出した式および Equation 2.20 の曲率と比較せよ。

3. [code] **安定性とミニバッチサイズ**。●損失の期待値からは、その勾配に沿ったステップは $\eta > 2$ で発散すると予測される。$\tilde{\mathbf{x}} = (\mathbf{x}, 1)$ を、バイアスのために 1 を付加した入力とする。

   1. フルバッチの勾配降下法において、各ステップがパラメータと最小二乗法による解との差に行列 $\mathbf{I} - \eta\mathbf{H}$ を掛けることを示せ。ここで $\mathbf{H}$ は、訓練セット全体での $\tilde{\mathbf{x}}\tilde{\mathbf{x}}^\top$ の平均である。$\mathbf{H}$ の固有値を計算し、安定な最大学習率を予測せよ。
   2. サイズ 1,000、32、1 のミニバッチで訓練し、各実行の更新回数を同じ値（たとえば 320 回）に揃えよ。各ミニバッチサイズについて、任意に選んだグリッド上で最終的な訓練損失が 10^{-3} 未満となる最大の学習率を求めよ。
   3. 単一の例に対する 1 ステップが、パラメータ誤差の $\tilde{\mathbf{x}}$ 方向の成分を $1 - \eta\|\tilde{\mathbf{x}}\|^2$ 倍にすることを示し、$\mathbb{E}\|\tilde{\mathbf{x}}\|^2 = 3$ を用いてサイズ 1 のミニバッチで見られた限界を説明せよ。

4. **なぜ再シャッフルするのか**。`minibatches` は毎エポック新しいランダムな順序を生成する。これによってミニバッチ勾配のどの性質が保たれるかを説明せよ。次に、ラベル順のソートなど、再シャッフルなしのミニバッチ SGD が異なる振る舞いを示す訓練セットの固定順序を 1 つ構成し、1 エポック内および複数エポックにわたる $(\mathbf{w}, b)$ の軌跡を記述せよ。

5. [code] **ロバストな損失**。●`squared_loss` を平均絶対誤差 `d2l.reduce_mean(d2l.abs(y_hat - d2l.reshape(y, (-1, 1))))` に置き換えよ。

   1. 合成データ上で訓練し、復元されたパラメータおよび損失曲線を二乗損失の場合と比較せよ。
   2. 2.1.1.3 節で用いた外れ値と同じく 1 つの訓練ラベルを $y^{(5)} = 10{,}000$ に設定し、両方の損失で訓練せよ。ミニバッチ更新ごとに b を記録して両方の実行について反復回数に対してプロットし、最終的な値を決める要因を含めて二乗損失の曲線のパターンを説明せよ。
   3. 外れ値を含むデータに対して、損失を次のフーバー損失のミニバッチ平均に置き換えよ。

$$l_\delta(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & \text{if } |y - y'| \le \delta \\ \delta\left(|y - y'| - \frac{\delta}{2}\right) & \text{otherwise,} \end{cases} \tag{2.21}$$

   この二次関数の部分は、本節の二乗損失と一致している。$\delta \in \{1, 10, 10^2, 10^3, 10^4\}$ に対して訓練し、学習されたバイアスと真のバイアスとの距離 $|\widehat{b} - b^*|$（ここで

$b^* = 4.2$ はデータ生成時のバイアス）を δ に対して両対数軸でプロットせよ。l_δ の y' に関する導関数から曲線の形状を説明し、その結果を Figure 2.6 のペナルティ曲線と関連づけよ。

2.4 実装のためのオブジェクト指向設計

2.3 節では、生のテンソルから線形回帰を実装し、1 つのモデルと 1 つのデータセットに直接結び付けられた独立な 12 行の `train` 関数で訓練した。今後、多層パーセプトロン（multilayer perceptron）、畳み込みネットワーク（convolutional network）、リカレントネットワーク、Transformer へと進むにあたり、データローダやエポックのループ、評価指標の平均計算、リアルタイムのプロットを毎節書き直していては、新しいアイデアが反復的な定型処理の下に埋もれてしまう。

モデリングに集中しつつ、訓練ハーネスのすべての行を見通しよく変更可能な状態に保つため、本書では `d2l` ライブラリを 3 つの明快なオブジェクト指向（object-oriented）の抽象化（Figure 2.9）を中心に構成する。これは PyTorch Lightning のようなフレームワークに着想を得たものである (Falcon and The PyTorch Lightning team 2019)。

1. **`Module`（モデル）**：訓練可能なパラメータを保持し、`forward` による順伝播（forward pass）を定義し、`loss` を計算し、`training_step` と `validation_step` を定め、`configure_optimizers` を通じてオプティマイザを返す。
2. **`DataModule`（データ）**：データセットのダウンロード、前処理、分割をカプセル化し、イテラブルな `train_dataloader()` と `val_dataloader()` を公開する。
3. **`Trainer`（訓練ループ）**：任意の `Module` と任意の `DataModule` を接続し、エポックごとのミニバッチ訓練パスと検証パスを実行し、バッチ間で評価指標を平均化して、リアルタイムの学習曲線（learning curve）を `ProgressBoard` 上にプロットする。

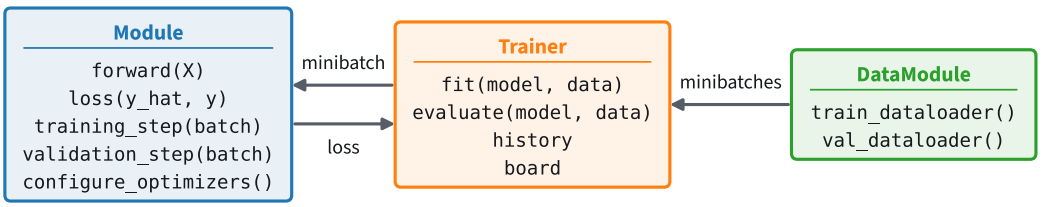


Figure 2.9: `d2l` 訓練ハーネスの中核をなす 3 つのクラス。`DataModule` は訓練用と検証用のミニバッチを供給し、`Module` はモデル・損失・オプティマイザを定義し、`Trainer` は訓練ループと検証ループを駆動する。

```
import collections
```

2.4.1 ユーティリティ

中核となる 3 つのクラスを構築する前に、ノートブック内の定義を読みやすく自己完結した形に保つ 2 つの小さな Python ユーティリティを導入する。

1. **`@add_to_class(Class)`**：教科書やノートブックにおいて、60 行のクラスを 1 つの巨大なコードブロックとして提示すると、各メソッドが登場するたびに説明するのが難しくなる。デコレータ `@add_to_class(Class)` を使えば、まずクラスの骨格だけを定義しておき、後のコードセルで新しいメソッドを段階的に追加できる。

2. **HyperParameters（save_hyperparameters()）**：self.a = a; self.b = b; ... のような記述をすべての __init__ メソッドの先頭に書くのは、純粋なボイラープレートである。HyperParameters を継承して self.save_hyperparameters() を呼び出すと、呼び出し元のフレームが検査され、すべての __init__ 引数（ignore に指定されたものを除く）が self の属性として自動的に束縛される。

```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
        return obj
    return wrapper
```

簡単なクラス A と、すでにインスタンス化されたオブジェクト a で試してみよう。

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

次に、別のセルで新しいメソッド do を追加する。Python はメソッドを呼び出し時にクラス上で解決するため、既存のインスタンス a からも直ちに do を利用できる。

```
@add_to_class(A)
def do(self):
    print('Instance attribute "b" is', self.b)

a.do()
```

```
Instance attribute "b" is 1
```

2 つ目の補助クラスは、コンストラクタのボイラープレートを取り除くものである。これがないと、コンストラクタは self.lr = lr や self.batch_size = batch_size のような反復的な代入で埋め尽くされてしまう。深層学習におけるコンストラクタ引数の大半はハイパーパラメータであるため、d2l.HyperParameters を継承して self.save_hyperparameters() を呼び出すことで、すべてのコンストラクタ引数（ignore に指定されたものを除く）を self の属性として自動的に束縛する。内部では Python の inspect モジュールを使って呼び出し元のスタックフレームを検査しており、その実装はオンライン付録「ユーティリティ関数とクラス」にある。

```
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('self.c exists:', hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
self.c exists: False
```

本節の残りで構築するすべてのクラスは HyperParameters を継承する。

2.4.2 モデル

Module クラスは、本書で実装するすべてのモデルの基底クラスである。このクラスは、各フレームワーク固有のニューラルネットワーク用コンテナと d2l.HyperParameters の両方を継承する。デフォルトの動作は次のとおりである。

- model(X) を呼び出すと model.forward(X) が実行される。
- training_step(batch) は、特徴量 batch[:-1] をモデルに通し、予測をラベル batch[-1] と比較して、ミニバッチの平均損失を返す（後の章のモデルでは、'loss' を含む評価指標の辞書を返す場合もある）。
- validation_step(batch) は、検証データのミニバッチに対して同じ損失を計算する（なお 3.3 節では、Classifier が validation_step をオーバーライドして 'loss' と 'acc' の両方を報告する）。
- 補助関数 as_metrics は、単一のスカラーまたは辞書の戻り値を評価指標の辞書へと正規化し、Trainer が任意の名前付き量を統一的に記録できるようにする。

PyTorch では、このコンテナは nn.Module である。

```
class Module(nn.Module, d2l.HyperParameters):
    """The base class of models."""
    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is not defined'
        return self.net(X)

    def training_step(self, batch):
        """The mean loss on one minibatch, or a dict of batch-mean metrics
        with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def validation_step(self, batch):
        """The mean loss on one minibatch of validation data, or a dict of
        batch-mean metrics with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def configure_optimizers(self):
        raise NotImplementedError
```

慣例として、すべてのミニバッチはタプルであり、末尾の要素がラベルテンソル、それより前の要素がモデルの入力である。training_step と validation_step はどちらも *batch[:-1] をアンパックして self(...) に渡し、batch[-1] に対する損失を計算して、その結果を単に**返す**だけである（それ自体では出力、プロット、ログ記録を一切行わない）。モデルがトレーナの存在をまったく意識しないようにしているのは意図的な設計上の選択であり、これによりどのようなカスタムループからでも隠れた副作用なしに Module を訓練・評価できる。

本章では、各ステップは単一のスカラー、すなわちミニバッチの平均損失を返す。3.3 節以降では、その代わりに評価指標名をミニバッチ平均に対応付ける辞書（常に 'loss' を含み、'acc' などの量も併せ持つ）を返す場合がある。小さな補助関数 as_metrics は、単なるスカラーの損失を {'loss': out} へと正規化し、トレーナが両者を統一的に扱えるようにする。

```
def as_metrics(out):
    """A step returns a loss or a dict of metrics; return a dict."""
    return out if isinstance(out, dict) else {'loss': out}
```

2.4.2.1 Module としての線形回帰

フレームワークの組み込み層を使う前に Module が実際にどう動くかを確認するため、スクラッチ実装した線形回帰のパラメータ $(\mathbf{w}, b)$、順伝播、二乗損失（squared loss）、およびスクラッチ実装の SGD 更新器を LinearRegressionScratch(d2l.Module) にまとめてみよう。

```
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = nn.Parameter(d2l.normal(0, sigma, (num_inputs, 1)))
        self.b = nn.Parameter(d2l.zeros(1))
```

次に、forward と loss を追加する。これらは 2.3 節の linreg および squared_loss と同一であり、モデルインスタンスから self.w と self.b を読み取るだけである。

```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return d2l.matmul(X, self.w) + self.b
```

```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2
    return d2l.reduce_mean(l)
```

続いて、スクラッチ実装したミニバッチ SGD の更新則を再利用可能な SGD(d2l.HyperParameters) クラスにまとめ、LinearRegressionScratch.configure_optimizers から返すようにする。

```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

最後に、configure_optimizers でモデルと SGD オプティマイザを結び付ける。

```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

2.4.3 データ

基底クラス DataModule は、データセットにおいて Module に対応するものである。具体的な各データセットのサブクラスは train_dataloader() と val_dataloader() を実装する（通常はどちらも get_dataloader(train=True/False) に処理を委譲する）。

```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

2.2 節で作成した合成回帰データの生成器を SyntheticRegressionData(d2l.DataModule) にまとめよう。このクラスはあらかじめ num_train + num_val 個の例（デフォルトでは訓練例 1,000 個、検証例 1,000 個）を生成しておくため、Trainer は毎エポック終了後に訓練損失（training loss）とホールドアウトされた検証損失（validation loss）の両方を追跡できる。

```
class SyntheticRegressionData(d2l.DataModule):
    """Synthetic data for linear regression."""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        self.X, self.y = d2l.synthetic_data(w, b, noise, num_train + num_val)
```

2.4.3.1 各エポック用のローダ

get_dataloader の内部では、エポックごとに 1 回反復でき、各エポックの開始時に訓練分割をシャッフルし直す一方で、検証分割は決定論的な順序で読み出すオブジェクトが必要になる。get_tensorloader メソッドは、分割されたテンソルからこのオブジェクトを構築する。

TensorDataset は先頭の例軸に沿ってテンソルを揃え、DataLoader はそれらをミニバッチへとスライスし、shuffle=True のときにはパスごとに自動でシャッフルし直す。

```
@d2l.add_to_class(d2l.DataModule)
def get_tensorloader(self, tensors, train, indices=slice(0, None)):
    tensors = tuple(a[indices] for a in tensors)
```

```
    dataset = torch.utils.data.TensorDataset(*tensors)
    return torch.utils.data.DataLoader(dataset, self.batch_size,
                                       shuffle=train)
```

これで SyntheticRegressionData.get_dataloader はわずか 2 行になり、訓練分割または検証分割のいずれかをスライスして get_tensorloader に委譲するだけである。

```
@d2l.add_to_class(SyntheticRegressionData)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader((self.X, self.y), train, i)
```

この合成データセットをインスタンス化し、先ほど単純なジェネレータがつまずいた 2 パスのテストをもう一度試してみよう。

```
data = SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
```

```
loader = data.train_dataloader()
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
for _ in range(2):  # Each new pass starts in a new random order
    X, y = next(iter(loader))
    print('a new pass starts with labels', d2l.numpy(y[:3, 0]).round(2))
```

```
32 minibatches, the last of size 8
second pass: 32 minibatches
a new pass starts with labels [-0.64 -5.75  3.55]
a new pass starts with labels [-3.35  0.07 -0.03]
```

今度はどちらのパスでも全 32 個のミニバッチ（末尾の 8 例の端数を含む）が得られ、パスごとに新しいランダムな順序で始まる。さらに都合がよいことに、データセットを反復処理することなく len(loader) を直接取得できる。

```
len(loader)
```

```
32
```

2.4.4 訓練

2.4.4.1 訓練ループと履歴

次に、Trainer を構築する。これは Module と DataModule を結び付けるエンジンである。trainer.fit(model, data) を呼び出すと、データローダを準備し、モデルとそのオプティマイザを設定して、max_epochs 回のパスを実行する。各エポックでは、fit_epoch が訓練ミニバッチを一通り処理し（パラメータを更新しつつ訓練の評価指標を平均化する）、続いて検証ミニバッチに対して validate を呼び出し、エポック平均を self.history に記録して ProgressBoard のリアルタイムプロットを更新する。

```
class Trainer(d2l.HyperParameters):
    """Fit a Module to a DataModule, keeping a per-epoch history."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        self.board = ProgressBoard(xlabel='epoch', ylabel='loss')

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()

    def prepare_model(self, model):
        self.model = model

    def prepare_optimizer(self, model):
        self.optim = model.configure_optimizers()
```

9.7 節で扱う学習率スケジュール（learning-rate schedule）に対応するため、`prepare_optimizer` は、任意選択でタプル `(optimizer, scheduler)` が `configure_optimizers` から返された場合にそれもアンパックする。

```
@d2l.add_to_class(Trainer)
def prepare_optimizer(self, model):
    """`configure_optimizers` may also return a learning-rate scheduler."""
    # PyTorch steps a schedule with a separate object (sec_scheduler)
    optim = model.configure_optimizers()
    self.optim, self.scheduler = (optim if isinstance(optim, tuple)
                                  else (optim, None))
```

`fit(model, data)` メソッドは、2.3 節の `train` 関数と同じ外側ループを実装する。3 つの準備用フックを実行した後、`max_epochs` エポックにわたってループを回す。各エポックでは、`fit_epoch()` が訓練ローダを一通り走査して訓練の評価指標の平均を返し、`validate()` が検証ローダ（存在する場合）を評価し、`record()` がその両方を記録・プロットする。

```
@d2l.add_to_class(Trainer)
def fit(self, model, data):
    self.prepare_data(data)
    self.prepare_model(model)
    self.prepare_optimizer(model)
    self.history = collections.defaultdict(list)
    self.board.clear()
    self.board.xlim = [0, self.max_epochs]
    for self.epoch in range(self.max_epochs):
        # The training mean averages over the epoch, so it sits mid-epoch
        self.record('train', self.fit_epoch(), x=self.epoch + 0.5)
        if self.val_dataloader is not None:
            self.record('val', self.validate(), x=self.epoch + 1)
        self.board.show(final=False)
    self.board.show()

@d2l.add_to_class(Trainer)
def record(self, split, metrics, x):
```

```
        for name in sorted(metrics, key=lambda name: (name != 'loss', name)):
            self.history[f'{split}_{name}'].append(metrics[name])
            self.board.draw(x, metrics[name], f'{split}_{name}')
```

なお、fit は現在のエポック番号を self.epoch として公開し、直接は何も返さない。記録されたすべての曲線は trainer.history に格納される。これは分割名と評価指標名（'train_loss'、'val_loss'、および後に登場する 'val_acc'）をキーとするリストの辞書である。したがって、trainer.history['val_loss'][-1] によって最終的な検証損失を直ちに取得できる。'loss' を先頭にソートしておくことで、モデルが異なってもプログレスボード上の曲線の色が常に一貫する。

これら 2 本の曲線は同じ位置には描画されない。その理由を Figure 2.10 に示す。エポック e の訓練損失は、そのエポック内の全ミニバッチステップにわたる平均であり、各ステップはそれぞれ異なるパラメータで計算されるため、fit はこれをエポックの中央である $e + \frac{1}{2}$ に描画する。一方、検証損失はそのエポック終了時点のパラメータを用いて 1 度だけ計算され、$e + 1$ に描画される。

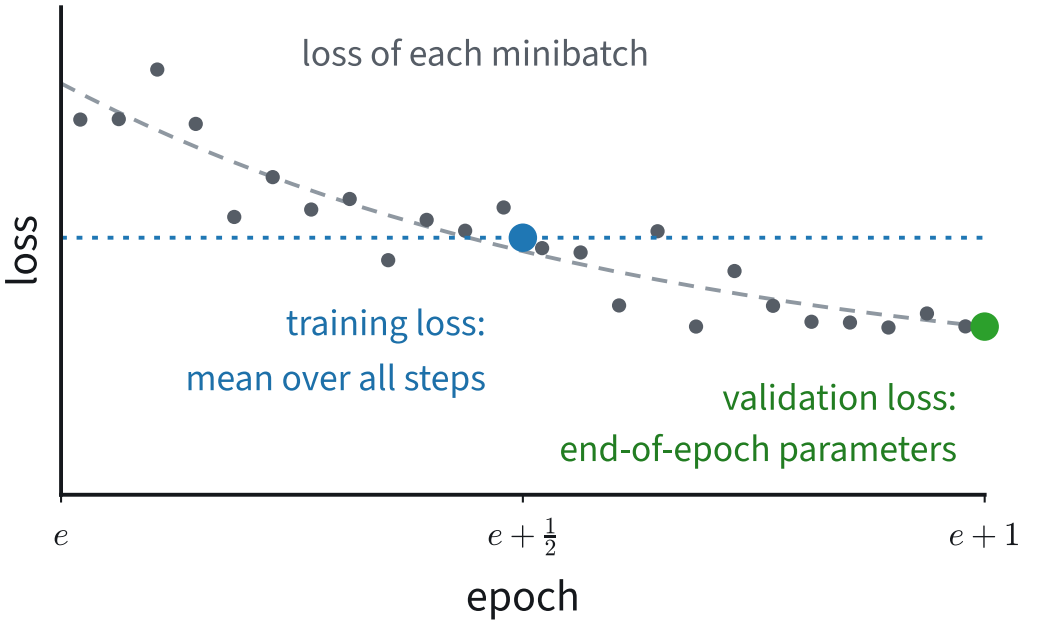


Figure 2.10: プログレスボード上の 1 エポック。パラメータが改善するにつれて個々のミニバッチの損失（灰色）は減少し、その平均である訓練損失が $e + \frac{1}{2}$ に描画される。検証損失はエポック終了時点のパラメータで計算され、$e + 1$ に描画される。

2.4.4.2 エポック平均

エポックの最後のミニバッチは batch_size より小さくなることが多いため（たとえば $1{,}000 = 31 \times 32 + 8$）、バッチごとの平均損失を重みなしで単純平均すると、最後のバッチに含まれる 8 個の例が、それ以前の各バッチに含まれる 32 個の例とまったく同じ発言力を持ってしまう。補助クラス Averager は各バッチの評価指標にそのバッチサイズ n で重み付けして累積するため、avg.mean() は常に分割全体にわたる正確な例単位の平均を返す。

```
class Averager:
    """Example-weighted means of batch metrics, summed on the device."""
    def __init__(self):
        self.sums, self.n = {}, 0

    def add(self, metrics, n):
        for name, value in metrics.items():
            self.sums[name] = self.sums.get(name, 0) + value * n
        self.n += n

    def mean(self):
        """One host synchronization per metric; the loss comes first."""
        names = sorted(self.sums, key=lambda name: (name != 'loss', name))
        return {name: float(self.sums[name]) / self.n for name in names}
```

2.4.4.3 訓練パスと検証パス

次に、validate と fit_epoch は、4 つのフレームワークそれぞれにおいてバッチごとの検証ループと訓練ループを実行する（第 8 章のリカレントネットワークに備え、任意の勾配クリッピング（gradient clipping）用の gradient_clip_val も用意してある）。

```
@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    return batch
```

このトレーナは本書の残りの章すべてであらゆるモデルを実行するため、単純な線形回帰だけに必要なものよりわずかに長くなっている。それでもアルゴリズムの中核は Figure 2.8 とまったく同じ 4 ステップの SGD サイクル（順伝播、損失の計算、逆伝播（backward pass）、パラメータ更新）であり、後の章に向けたフックがいくつか追加されているものの、ここではまだ使われない。

2 つの if ブロックは、勾配クリッピングと学習率スケジューラのためのフックである。.detach() を呼び出してから評価指標を Averager に渡すことで、累積する損失の和が各ミニバッチの計算グラフ（computational graph）をメモリ上に保持し続けるのを防ぐ。また、model.train() と model.eval() は、今後導入するドロップアウトやバッチ正規化などの層の挙動を切り替える。

```
@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    metrics = d2l.Averager()
    for batch in map(self.prepare_batch, self.train_dataloader):
        out = d2l.as_metrics(self.model.training_step(batch))
        self.optim.zero_grad()
        out['loss'].backward()
        if self.gradient_clip_val > 0:  # To be discussed later
            self.clip_gradients(self.gradient_clip_val, self.model)
        with torch.no_grad():
            self.optim.step()
        if self.scheduler is not None:
            self.scheduler.step()
        metrics.add({k: v.detach() for k, v in out.items()}, len(batch[-1]))
```

```
    return metrics.mean()

@d2l.add_to_class(d2l.Trainer)
def validate(self):
    self.model.eval()
    metrics = d2l.Averager()
    with torch.no_grad():
        for batch in map(self.prepare_batch, self.val_dataloader):
            metrics.add(d2l.as_metrics(self.model.validation_step(batch)),
                        len(batch[-1]))
    return metrics.mean()
```

2.4.4.4 プログレスボード

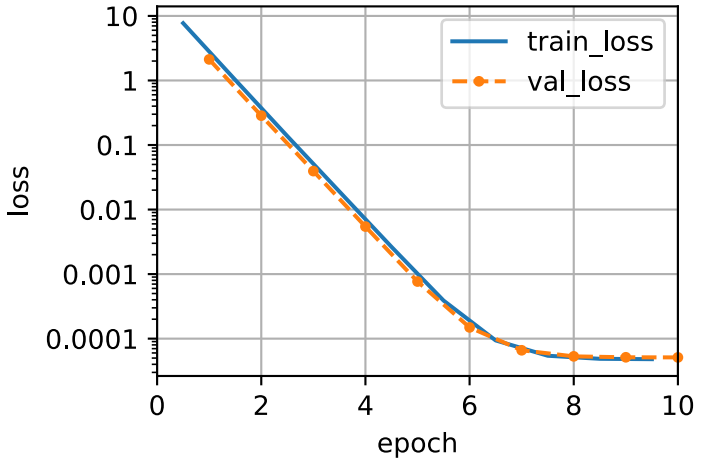
最後に、ProgressBoard はノートブックの実行中にエポックごとの学習曲線をリアルタイムで描画する（描画なしのスイープを実行する際には trainer.board.display = False によって非表示にできる）。

```
class ProgressBoard(d2l.HyperParameters):
    """Curves that grow during training; `show` redraws them."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None, ylim=None,
                 xscale='linear', yscale='linear', figsize=(3.5, 2.5),
                 display=True):
        self.save_hyperparameters()
        self.clear()

    def clear(self):
        """Forget all curves; the next `show` starts a new figure."""
        self.data, self.handle = {}, None

    def draw(self, x, y, label):
        """Append the point (x, y) to the curve called `label`."""
        xs, ys = self.data.setdefault(label, ([], []))
        xs.append(x)
        ys.append(y)

    def show(self, final=True):
        raise NotImplementedError
```

2.4.5 線形回帰の再訓練

新たに組み立てた Module、DataModule、Trainer の各クラスをエンドツーエンドでテストするため、LinearRegressionScratch を SyntheticRegressionData 上で 10 エポック訓練し、復元されたパラメータを確認してみよう。

```
model = LinearRegressionScratch(2, lr=0.03)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```



検証損失は 2.3 節とまったく同じ軌跡をたどり、エポックごとに約 $7\times$ の倍率で急減して 5×10^{-5} 付近のノイズフロアに達する。訓練曲線もその真上に重なっている（パラメータがわずか 3 個なのに対し例が 1,000 個あるため、モデルはラベルノイズをほとんど記憶しない）。最終的な検証損失とパラメータの復元誤差を数値で確認しよう。

```
w_error = d2l.numpy(d2l.reshape(model.w, data.w.shape) - data.w)
b_error = d2l.numpy(model.b - data.b)
print(f"validation loss: {trainer.history['val_loss'][-1]:.1e}")
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
validation loss: 5.1e-05
error in w: 4.3e-04, 3.0e-04
error in b: -6.7e-04
```

訓練損失と検証損失はいずれもノイズフロアである 0.00005 まで滑らかに減少し、復元されたパラメータも $(\mathbf{w}^*, b^*)$ と約 1,000 分の 1 以内で一致している。これにより、オブジェクト指向のハーネスが独立したスクラッチ実装のループを忠実に再現しつつ、以降の章をボイラープレートから解放することが確認できた。2.5 節では次のステップに進み、手書きのテンソル、二乗損失、SGD クラスを各フレームワークの組み込みの層、損失関数、オプティマイザへと置き換える。

2.4.6 まとめ

以降の章で反復的な訓練のボイラープレートではなくモデリングに集中できるよう、d2l ライブラリはモデル（Module）、データセット（DataModule）、最適化ループ（Trainer）を分離し、それらを @add_to_class、HyperParameters、Averager、ProgressBoard で支えている。DataModule は 1 エポックで枯渇することなくパスごとに訓練例をシャッフルし直す複数エポック対応のローダを返し、Averager は各ミニバッチの評価指標に例数で重み付けすることで、端数の最終バッチによってエポック単位の平均が歪むのを防ぐ。Module は明快な戻り値の契約（training_step、validation_step、configure_optimizers）を公開しているため、LinearRegressionScratch のように生のテンソルから構築したモデルであっても、フレームワークの層から構築したモデルであっても、すべて同じ trainer.fit(model, data) の呼び出しで訓練および検証を行える。

2.4.7 演習問題

1. **訓練曲線の読み取り**。トレーナは各エポックの平均訓練損失をエポックの中央に、検証損失をその終端に描画する。
    1. 訓練の初期では、両方の例集合が同じ分布から得られているにもかかわらず、訓練曲線が検証曲線より上に位置することが多い。2 つの平均の計算方法からその理由を説明せよ。
    2. 仮に `fit` が検証損失と同様に、各エポック終了時点のパラメータを使って訓練損失を計算したとする。1 エポックあたりどれだけの追加コストがかかるか。また、そのとき 2 本の曲線の差は何を測ることになるか。
2. [code] **早期停止**。固定されたエポック数だけ訓練すると、停止が早すぎたり遅すぎたりすることがある。`d2l.Trainer` のサブクラスを作成し、その `fit` が、検証損失が連続して `patience` エポック減少しなくなった時点で `self.history` のみに基づいて訓練を終了するようにせよ。また、モデルやデータセットのクラスを一切変更する必要がない理由を説明せよ。実装したトレーナを、大きな `max_epochs` を設定した合成回帰データ上でテストせよ。
3. [code] **プランクの法則**。●同じ `Module` と `Trainer` は、線形モデルだけでなく、損失が微分可能なあらゆるモデルに当てはめることができる。プランクの法則を用いて、複数の波長における分光放射輝度の測定値から黒体の温度を求めよ。分光放射輝度は次式で与えられる。

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda k T} - 1 \right)^{-1}, \tag{2.22}$$

    ここで、λ は波長、T は温度、c は光速、h はプランク定数、k はボルツマン定数である。既知の T における測定値を生成した上で、`d2l.Module` のサブクラスとして `forward` が $B(\lambda, T)$ を計算し T を唯一のパラメータとして持つものを作成し、`d2l.Trainer` で当てはめよ。当てはめる前に、B と T が 1 のオーダーになるような単位を選べ。
4. **端数の最終ミニバッチ**。各分割に 1,000 個の例があり `batch_size=32` のとき、各ローダはパスの最後に 8 例からなる端数のミニバッチを出力する。`DataLoader` に `drop_last=True` を指定すると、この端数ミニバッチは破棄される。訓練ローダは毎エポック再シャッフルするのに対し、検証ローダは固定された順序で読み出す。それぞれのローダについて、端数ミニバッチの破棄によって多数のエポックにわたりどの例が除外されるかを述べ、どちらのローダであれば破棄が許容されるかを判断せよ。
5. [code] **オンザフライ生成**。`DataModule` のサブクラスとして、$\mathbf{X}$ や $\mathbf{y}$ を保持しないものを作成せよ。その訓練ローダは、ミニバッチが要求されるたびにデータ生成過程から毎回新たにミニバッチを抽出するものとする。
    1. 新規ミニバッチのストリームには自然な終わりがないが、`Trainer` は訓練ローダに対する 1 回のパスが終了したときにのみエポックを終了する。1 エポックが生成するミニバッチ数を決め、すべてのパスがその数で停止するようにし、`len` がその数を返すようにせよ。
    2. 検証ローダもパスごとに毎回新たな例を抽出すべきかどうかを判断し、その選択を実装せよ。また、検証損失の用途に基づいてその判断の根拠を述べよ。

2.5 線形回帰の簡潔な実装

2.3 節では、パラメータの確保、アフィン順伝播、二乗損失、リバースモード微分、ミニバッチジェネレータ、SGD の更新ステップといった線形回帰のあらゆる構成要素を生のテンソルから構築し

た。続く 2.4 節では、それらの構成要素を再利用可能な基底クラス（d2l.Module、d2l.DataModule、d2l.Trainer）へと整理し、LinearRegressionScratch を trainer.fit(model, data) で訓練することで、ベンチマーク結果が再現されることを確認した。

生のパラメータテンソルや更新規則を手書きしてテストするのは、パラメータが 3 個の線形モデルであれば簡単であった。しかし、ResNet-50（7.4 節）や GPT（11.2 節）のような現代のアーキテクチャは、数十の層にわたって数百万〜数十億個のパラメータを管理し、状態を持つ適応的オプティマイザを用いて訓練する。そのため、PyTorch (Paszke et al. 2019)、JAX (Frostig et al. 2018)（Flax と Optax を併用）、TensorFlow (Abadi et al. 2016)、MXNet (Chen et al. 2015) といった深層学習フレームワークは、ハードウェア向けに最適化された層、数値的に検証済みの損失プリミティブ、標準的なオプティマイザを標準で提供している。本節では、こうした高水準の構成要素を d2l.Module と d2l.Trainer に組み込むことで、線形回帰をいかに簡潔に記述できるかを見ていく。

2.5.1 全結合層

この簡潔な実装では、サブクラス LinearRegression(d2l.Module) が出力ユニット数 1 の組み込みの全結合層（fully connected layer）を 1 つラップし、重みを $\mathcal{N}(0, 0.01^2)$ から、バイアスを 0 に初期化する。

PyTorch の nn.LazyLinear(1) は構築時には出力次元のみを必要とし、入力特徴量の数 d は最初の順伝播時に自動的に決まる。

```
class LinearRegression(d2l.Module):
    """Linear regression built from framework components."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

なお、LinearRegression では forward を定義する必要すらなく、Module.forward を 2.4 節から継承してそのまま self.net(X) に処理を委譲する。

2.5.2 損失関数

$(\hat{y} - y)^2/2$ を手書きする代わりに、loss はフレームワーク組み込みの平均二乗誤差（mean squared error、MSE）プリミティブに委譲する。標準的な平均二乗誤差はスクラッチ実装の損失にあった係数 $\frac{1}{2}$ を省くため、値はスクラッチ実装の 2 倍になり、$\sigma^2 = 10^{-4}$ で頭打ちになる（$\sigma^2/2$ ではない）。

```
@d2l.add_to_class(LinearRegression)
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

2.5.3 オプティマイザ

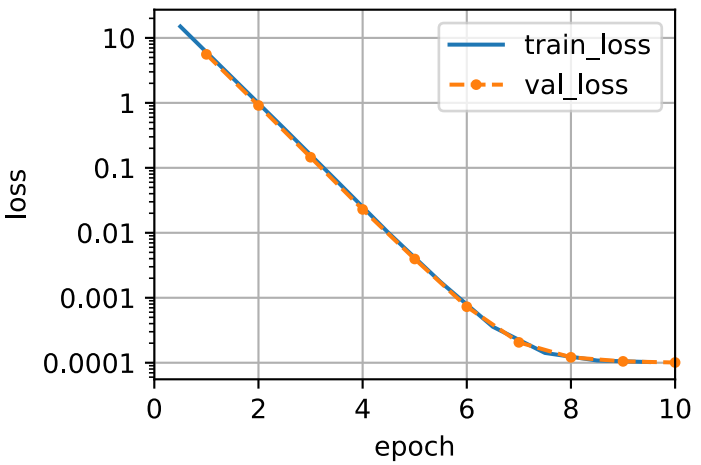
同様に、configure_optimizers はモジュールの登録済みパラメータと学習率 self.lr を接続した、フレームワーク組み込みのミニバッチ SGD オプティマイザを返す。

```
@d2l.add_to_class(LinearRegression)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

2.5.4 フレームワークの構成要素を用いた訓練

LinearRegression を定義すれば、訓練はデータモジュール、モデル（学習率は $\eta = 0.015$ とし、MSE の勾配が 2 倍になる分を相殺するためスクラッチ実装の $\eta = 0.03$ から半減させる）、および d2l.Trainer(max_epochs=10) をインスタンス化し、trainer.fit(model, data) を呼び出すだけで済む。

```
data = d2l.SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
model = LinearRegression(lr=0.015)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```




```
print(f"validation loss: {trainer.history['val_loss'][-1]:.2e}")
```

```
validation loss: 1.01e-04
```

最終的な検証損失は約 3% 以内の差で理論的なノイズフロア $\sigma^2 = 10^{-4}$ に収まり、スクラッチ実装の結果と一致する。ここで小さなヘルパーメソッド get_w_b を追加し、学習された重みテンソルとバイアスをフレームワークの層から取り出して $(\mathbf{w}^*, b^*)$ と比較してみよう。

```
@d2l.add_to_class(LinearRegression)
def get_w_b(self):
    return self.net.weight.detach(), self.net.bias.detach()
```

```
w, b = model.get_w_b()
w_error = d2l.numpy(d2l.reshape(w, data.w.shape) - data.w)
b_error = d2l.numpy(b - data.b)
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: -2.1e-04, 9.5e-04
error in b: -1.1e-03
```

今回も $\hat{\mathbf{w}}$ と $\hat{b}$ の誤差は 1,000 分の 1 程度に収まり、スクラッチ実装での実行結果と一致する。支配的な誤差要因（1,000 個の例にわたるラベルノイズと、10 エポック後に残る距離）は、誰が層を実装したかにかかわらず同一だからである。

ここまでの 3 つの節で確認してきた内容はすべて、正解（ground truth）のパラメータが既知であることに依拠していたが、これは現実のデータセットでは決して望めない贅沢である。続く 2.6 節では、ホールドアウトデータのみからモデルを評価する方法に取り組む。

2.5.5 まとめ

高水準のフレームワーク API を用いれば、線形回帰は `d2l.Module` のサブクラス内に 1 つの全結合層を置き、`loss` を組み込みの平均二乗誤差に、`configure_optimizers` を組み込みの SGD に委譲するだけで実装できる。標準的な平均二乗誤差はスクラッチ実装の目的関数にあった係数 $\frac{1}{2}$ を省くため、損失（$\sigma^2 = 10^{-4}$ で頭打ちになる）とその勾配がいずれも 2 倍になる。したがって、学習率を $\eta = 0.015$ へと半減させることで、スクラッチ実装と同じパラメータ更新が再現される。また、`LinearRegression` は `d2l.Module` の規約を満たしているため、データパイプラインや訓練ループを一切変更せずに `d2l.Trainer` で訓練できる。

2.5.6 演習問題

1. [code] **バイアスのない層**。バイアスを持たない層を用いて `LinearRegression` を構築し（どのフレームワークの全結合層にもバイアスを無効化する引数がある）、本節と同様に訓練せよ。
   1. 訓練の前に、2.3 節の期待二乗誤差の式を用いて（平均二乗誤差における係数 2 を考慮したうえで）、検証損失が頭打ちになる値を予測せよ。その後、予測を確かめよ。
   2. 重みが何に収束し、最終的に $\mathbf{w}^*$ からどれだけ離れた値になるかを予測せよ。特徴量の平均が 0 であるため、欠落したバイアスは標準偏差 $b^* = 4.2$ のラベルノイズのように重みに作用し、2.2.3 節の推定誤差とミニバッチ SGD のノイズ球（2.1 節）の両方に影響する。この予測を、訓練後の重みの誤差と比較せよ。
2. [code] **複数の出力**。`LinearRegression` の層の出力幅を 1 から 2 に変更せよ。`data` を構築した後、`data.y` に 2 番目のラベル列 $y_2 = \mathbf{x}^\top \mathbf{w}_2^* + b_2^* + \epsilon_2$（任意の生成パラメータと同じ大きさのノイズを用いて `data.X` から計算したもの）を追加せよ。
   1. 2 つの出力にわたって和をとった二乗損失が、出力ごとに 1 つずつの独立した 2 つの線形回帰に分解されることを示せ。そこから、一方の出力に対して学習されるパラメータが他方の出力のラベルに依存しないことを結論づけよ。
   2. フレームワークの損失関数は、例方向だけでなく、$|\mathcal{B}| \times 2$ の誤差行列（$\mathcal{B}$ はミニバッチ）のすべての要素にわたって平均をとる。単一出力モデルと比べて各出力のパラ

メータの勾配が何倍に縮小するか、またどの学習率を用いれば単一出力の訓練結果が再現されるかを述べよ。両方のモデルを同じ初期パラメータから同じミニバッチ上で訓練し、これを確認せよ。

3. [code] **特徴量のスケールとステップサイズ**。`data.X` の 1 列目を（`data` の構築後に）100 倍し、`data.y` はそのままにして、その特徴量の生成重みが 0.02 になるようにせよ。演習問題 1 で見たように、勾配降下法は η が $2/\mu_{\max}$ を超えると発散する（ここで $\mu_{\max}$ は損失のヘッセ行列 $\mathbf{H}$ の最大固有値である）。
   1. `LinearRegression.loss` に対する $\mathbf{H}$ を計算し、リスケーリングの前後でその固有値を推定せよ。
   2. リスケーリングの前後それぞれについて、訓練が発散しない最大の学習率を予測し、その前後の学習率で実際に訓練して両方の予測を確かめよ。
   3. 新たな上限よりわずかに小さい学習率において、2 番目の重みの誤差が 100 分の 1 に減少するまでに必要な更新回数を予測し、実測値と比較せよ。その後、リスケーリングされた特徴量を 1.2 節と同様に標準化し、`lr=0.015` が再び機能することを示せ。
4. [code] **最適化誤差と推定誤差**。●`LinearRegression` を `lr=0.015`、`max_epochs=10` として、`SyntheticRegressionData` 上で `num_train` を $5, 50, 500, 5{,}000$ のそれぞれに設定し、`batch_size=min(32, num_train)` で訓練せよ。各実行について、訓練後の重みと生成重みの距離 $\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2$（生成重み $\mathbf{w}^*$ は `data.w` に格納されている）を、同じ訓練データに対する最小二乗解（パラメータ復元の項である 2.2.3 節と同様に計算したもの）の同じ距離と比較せよ。
   1. 各実行におけるパラメータ更新の回数を数え、各 `num_train` における 2 つの誤差の比を説明せよ。
   2. `num_train=5` での実行が、その最小二乗法による解の誤差の 2 倍以内に収まるまでのエポック数を求めよ。

2.6 汎化

統計学の期末試験に向けて、過去問で演習している 2 人の大学生 Ellie と Irene を想像してみよう。Ellie は過去問のすべての問題文と正確な数値の答えを一言一句丸暗記し、過去問の採点では 100% をとる。一方、Irene は背後にある概念や公式の理解に努める。時折計算ミスをするため過去問での得点は 92% にとどまるが、まったく新しい問題が出題される本番の期末試験になると、Irene が 90% をとる一方で Ellie は手も足も出ない。

機械学習モデルも同じ分岐点に直面する。最終的な目標は、手元にある有限の**訓練セット**（training set）で単に高いスコアを出すことではなく、より広い母集団から抽出される未知の例に対して正確な予測を下すことにある。この性質を**汎化**（generalization）と呼ぶ。先ほどの合成データによる線形回帰では、3 個のパラメータを 1,000 個の例に当てはめたため丸暗記は不可能であり、訓練損失は検証損失とほぼ一致して推移した。しかし、モデルクラスが個々のデータ点の周りで自在に曲がれるほど表現力をもつようになると、訓練誤差を 0 に追い込むことで、新しいデータに対する予測がかえって**悪化**しうる。本節では、訓練誤差と汎化誤差の違いを定式化し、多項式曲線フィッティングを通じて**学習不足**（underfitting）と**過学習**（overfitting）を実演したうえで、**バイアス・分散分解**（bias-variance decomposition）を解き明かし、原理に基づいたモデル選択（model selection）のための**検証セット**（validation set）と K **分割交差検証**（K-fold cross-validation）を導入する。

2.6.1 訓練誤差と汎化誤差

教師あり学習（supervised learning）では、訓練例も、実運用時に遭遇する将来のテスト例も、背後にある同一の確率分布 $P(\mathbf{x}, y)$ から独立に抽出されると仮定する。これが**独立同分布**（independent and identically distributed、i.i.d.）の仮定である。

有限の訓練データセット $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$（$n$ 個の例を P から i.i.d. に抽出したもの）が与えられたとき、予測器 f の損失 l のもとでの**訓練誤差**（training error）、すなわち**経験リスク**（empirical risk）は、$\mathcal{D}$ 上の標本平均で定義される。

$$R_{\text{emp}}[f] = \frac{1}{n} \sum_{i=1}^{n} l\left(\mathbf{x}^{(i)}, y^{(i)}, f(\mathbf{x}^{(i)})\right). \tag{2.23}$$

これに対し、**汎化誤差**（generalization error）、別名**真のリスク**（true risk）または**母集団リスク**（population risk）は、新たな例 $(\mathbf{x}, y)$ を P から抽出したときの期待損失である。

$$R[f] = \mathbb{E}_{(\mathbf{x},y)\sim P}\Big[l(\mathbf{x}, y, f(\mathbf{x}))\Big] = \int l(\mathbf{x}, y, f(\mathbf{x}))\, dP(\mathbf{x}, y). \tag{2.24}$$

求めたいのは関数 f のうち真のリスク $R[f]$ を最小化するものだが、分布 P に直接アクセスすることは決してできない。観測できるのは有限のサンプル $\mathcal{D}$ だけであるため、訓練アルゴリズムはその代わりに $R_{\text{emp}}[f]$ を最小化する。これを**経験リスク最小化**（empirical risk minimization）と呼ぶ。

なぜ $R_{\text{emp}}[\hat{f}]$ は $R[\hat{f}]$ をこれほど大きく過小評価しうるのだろうか。もし事前に関数 f を**固定**して（$\mathcal{D}$ を見る前に）選んでいれば、大数の法則（law of large numbers）により、$R_{\text{emp}}[f]$ は $R[f]$ の不偏推定値となり、その変動は $1/\sqrt{n}$ のオーダーに収まる。しかし、訓練されたモデル $\hat{f}$ は $\mathcal{D}$ と独立に固定されては**いない**。モデルクラス全体 $\mathcal{F}$ の中から、まさに $\mathcal{D}$ 上の誤差を最小化するように選び出されたものである。$\mathcal{F}$ が十分に柔軟で、$\mathcal{D}$ にたまたま含まれるノイズの癖まで拾い上げられるようになると、訓練誤差 $R_{\text{emp}}[\hat{f}]$ には真のリスク $R[\hat{f}]$ に対して下向きの楽観的な統計的バイアスが生じる（演習問題 4）。$R[\hat{f}]$ を公正に測るには、$\hat{f}$ を、そのパラメータの当てはめに一切関与していない独立なホールドアウトの**テストセット**（test set）または**検証セット** $\mathcal{D}'$ 上で評価しなければならない。

2.6.2 学習不足と過学習

訓練誤差と検証誤差（validation error）を比較すると、正反対の 2 つの失敗モードを診断できる。

- **学習不足（高バイアス）：**モデルの自由度が低すぎてデータの背後にある真の傾向を捉えられず、訓練誤差と検証誤差の**両方**が高いまま近い値をとる。
- **過学習（高分散）：**データセットのサイズに対してモデルが柔軟すぎるため、信号だけでなく訓練例に含まれるランダムなノイズにまで適合してしまう。その結果、訓練誤差はごく小さくなる一方で検証誤差ははるかに高くなり、大きな**汎化ギャップ**（generalization gap）である $R[\hat{f}] - R_{\text{emp}}[\hat{f}]$ が開く。

2.6.2.1 多項式曲線フィッティング

線形回帰の枠組みから出ることなく、学習不足、バランスのとれた当てはめ、そして壊滅的な過学習が進行する様子を観察できる。スカラー入力 x が $[-1, 1]$ から一様に抽出され、真のラベルが滑らかな曲線にガウスノイズ（Gaussian noise）を加えた次の式で生成されるとする。

$$y = f^*(x) + \epsilon, \qquad f^*(x) = 5 + 1.2x - 3.4x^2 + 5.6x^3, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{2.25}$$

ここで $\sigma = 0.1$ とする。なお、二乗誤差 $(\hat{y} - y)^2$ における削減不能な母集団のノイズフロアは $\sigma^2 = 0.01$ である。真の関数 $f^*(x)$ でさえ 0.01 の期待二乗誤差を生じるのは、予測不可能なノイズ ϵ があるからである。

線形回帰を用いて曲線関数を当てはめるには、スカラー入力 x を単項式からなる**多項式特徴量**（polynomial features）のベクトル $(1, x, x^2, \ldots, x^k)^\top \in \mathbb{R}^{k+1}$ に写像する。k 次の多項式は次のように表される。

$$\hat{y} = \sum_{j=0}^{k} w_j x^j \tag{2.26}$$

この式は x については非線形であるが、**重み $\mathbf{w} = (w_0, \ldots, w_k)^\top$ については線形**である。したがって、任意の k 次多項式を n 個の訓練点に当てはめるには、Vandermonde 計画行列 $\mathbf{X} \in \mathbb{R}^{n \times (k+1)}$（成分は $X_{ij} = (x^{(i)})^j$）を構成して `lstsq` を呼び出せばよい（2.1.2.1 節と同様である）。それでは、$n = 20$ 個の訓練点と、テスト誤差（test error）を正確に推定するのに十分な 1,000 個のテスト点を生成し、$k = 1$（直線）、$k = 3$（3 次曲線）、$k = 19$（20 個のパラメータをもつ、20 個の点に対する 19 次多項式）の 3 つの次数を比較してみよう。

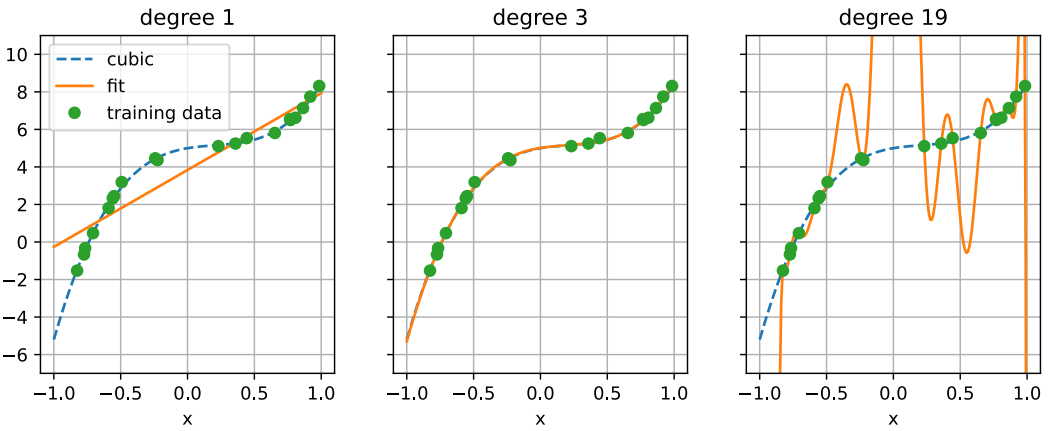
```
def target(x):
    """The cubic f* that generates the labels."""
    return 5 + 1.2 * x - 3.4 * x**2 + 5.6 * x**3

def make_data(rng, n, sigma):
    """Draw n inputs uniformly from [-1, 1] and label them by the noisy cubic."""
    x = rng.uniform(-1, 1, n)
    return x, target(x) + rng.normal(0, sigma, n)

sigma = 0.1                                 # standard deviation of the noise
rng = np.random.default_rng()               # a new random draw on every run
x_train, y_train = make_data(rng, 20, sigma)  # few, so high degrees overfit
x_test, y_test = make_data(rng, 1000, sigma)
```

以下のコードでは、`features(x, d)` が計画行列 $[1, x, \ldots, x^d]$ を構築し、`fit_degree` が最小二乗法（least squares）による厳密な重みを求め、`mse` が通常の平均二乗誤差（mean squared error）を計算する（係数 $\frac{1}{2}$ を省くことで、ノイズフロアがちょうど $\sigma^2 = 0.01$ になるようにしている）。

```
def features(x, d):
    """Design matrix whose columns are the powers x^0, x^1, ..., x^d."""
    return x.reshape(-1, 1) ** np.arange(d + 1)

def fit_degree(x, y, d):
    """Least-squares weights of the degree-d polynomial fitted to (x, y)."""
    return np.linalg.lstsq(features(x, d), y, rcond=None)[0]

def mse(w, x, y):
    """Mean squared error of the polynomial with weights w on (x, y)."""
    return ((features(x, len(w) - 1) @ w - y) ** 2).mean()
```

次数 1、3、19 の当てはめ結果を、ノイズを含む 20 個の訓練点および真の 3 次曲線とともにプロットしてみよう。

```
xs = np.linspace(-1, 1, 200)
_, axes = d2l.plt.subplots(1, 3, figsize=(10, 3.4), sharey=True)
for ax, d in zip(axes, (1, 3, 19)):
    w = fit_degree(x_train, y_train, d)
    d2l.plot([xs, xs, x_train], [target(xs), features(xs, d) @ w, y_train],
             xlabel='x', ylim=(-7, 11), fmts=('--', '-', 'o'),
             legend=['cubic', 'fit', 'training data'] if d == 1 else None,
             axes=ax)
    ax.set_title(f'degree {d}')
```



これら 3 つのパネルは、教師あり学習における 2 つの典型的な失敗モードのすべてを物語っている。

- **次数 $k = 1$（学習不足）：**直線は曲がることができず S 字形の 3 次曲線に追従できないため、訓練点とテスト点の双方を外してしまう。
- **次数 $k = 3$（適切な適合）：**3 次多項式は f^* とちょうど同じ柔軟性をもち、当てはめた曲線は真の曲線とほとんど見分けがつかない。
- **次数 $k = 19$（過学習）：**パラメータ数 $k + 1 = 20$ に対して相異なる訓練点が $n = 20$ 個あるため、この多項式はすべての訓練点を完全に補間できるだけの自由度をもつ（演習問題 2）。しかし、データ点の間では曲線が激しく振動し、テスト誤差は何桁も跳ね上がる。

多項式が過学習するとき、学習された重み w_j 自体には何が起きているのだろうか。3 つの次数のそれぞれについて、訓練誤差、テスト誤差、および重みの最大絶対値 $\max_j |\hat{w}_j|$ を出力してみよう。

```
for d in (1, 3, 19):
    w = fit_degree(x_train, y_train, d)
    print(f'degree {d:2d}: train {mse(w, x_train, y_train):.2g}, '
          f'test {mse(w, x_test, y_test):.2g}, '
          f'largest |weight| {np.abs(w).max():.2g}')
```

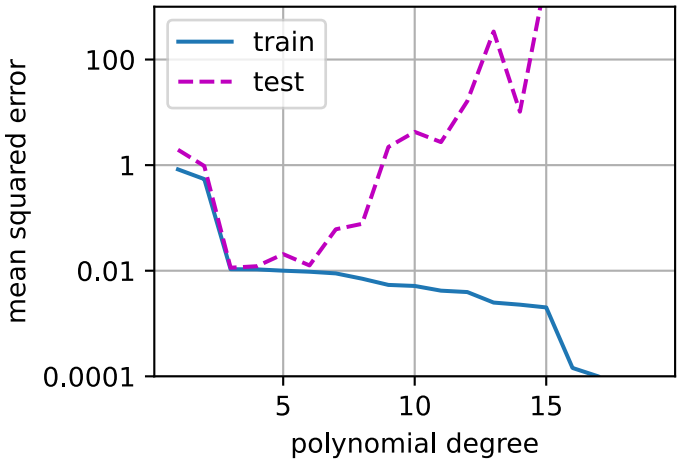
```
degree  1: train 0.83, test 2, largest |weight| 4.1
degree  3: train 0.011, test 0.011, largest |weight| 5.8
degree 19: train 2.4e-19, test 2.9e+06, largest |weight| 9.4e+06
```

乱数の抽出ごとに正確な数値はわずかに変わるものの、その対比は劇的である。次数 1 は両方の分割で大きな誤差を抱える（テスト MSE は 2 前後と、約 200× も $\sigma^2 = 0.01$ のノイズフロアを

上回る)。次数 3 は訓練誤差とテスト誤差の双方を $\sigma^2 = 0.01$ のノイズフロアまで引き下げ、最大の重みも真の最高次係数 5.6 の近くに収まる。次数 19 は訓練誤差をノイズフロアよりはるかに低く、実質的に 0 まで押し下げる一方で、テスト誤差は何桁も爆発する。その仕組みを知るには重みを見ればよい。次数 19 は巨大な正負の係数に頼っており、それらが 20 個の訓練入力では絶妙に打ち消し合う一方、それ以外のあらゆる場所では発散するのである。

次数を 1 から 19 まですべて動かすと、この推移の全体像が対数スケール上で浮かび上がる。

```
degrees = list(range(1, 20))
weights = [fit_degree(x_train, y_train, d) for d in degrees]
train_err = [mse(w, x_train, y_train) for w in weights]
test_err = [mse(w, x_test, y_test) for w in weights]
d2l.plot(degrees, [train_err, test_err], xlabel='polynomial degree',
         ylabel='mean squared error', legend=['train', 'test'],
         yscale='log', ylim=(1e-4, 1e3))
```



k が増えるにつれて、訓練誤差は単調に減少し(x^{k+1} を追加しても最小二乗法が利用できる関数の部分空間が広がるだけだからである)、σ^2 をはるかに下回る水準へ $k = 19$ までに急落する。対照的に、テスト誤差は典型的な **U 字型の曲線**を描く。学習不足が解消されるにつれて $k = 3$(またはその近傍の次数)で最小値まで急落したのち、余分な x のべき乗が 20 個の訓練点に含まれるランダムなノイズに適合していくため、何桁も上昇する。x^{17} と x^{19} は $[-1, 1]$ 上でほぼ同じ形をしているため、高次の当てはめでは一方の巨大な正の定数倍と他方の巨大な負の定数倍を打ち消し合わせることでノイズを含む 20 個の点を縫い通す。この不安定性は、2.7 節において重み減衰(weight decay)で直接解消する。

2.6.2.2 バイアスと分散

なぜモデルの複雑さ(model complexity)によって U 字型の汎化曲線が生じるのだろうか。モデル $\hat{f}_{\mathcal{D}}$(ランダムなデータセット $\mathcal{D}$ 上で訓練されたもの)のテスト点 x における期待二乗誤差を分解してみよう。ここで $y = f^*(x) + \epsilon$ であり、$\mathbb{E}[\epsilon] = 0$、$\mathrm{Var}(\epsilon) = \sigma^2$、かつ ϵ は $\mathcal{D}$ と独立であるとする。$\bar{f}(x) = \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(x)]$ を、サイズ n のあらゆる訓練セットにわたる平均予測器としよう。$f^*(x)$ と $\bar{f}(x)$ を $\mathbb{E}_{\mathcal{D},\epsilon}[(\hat{f}_{\mathcal{D}}(x) - y)^2]$ の中で足して引き、交差項が消えることを用いると、有名な**バイアス・分散分解**が得られる。

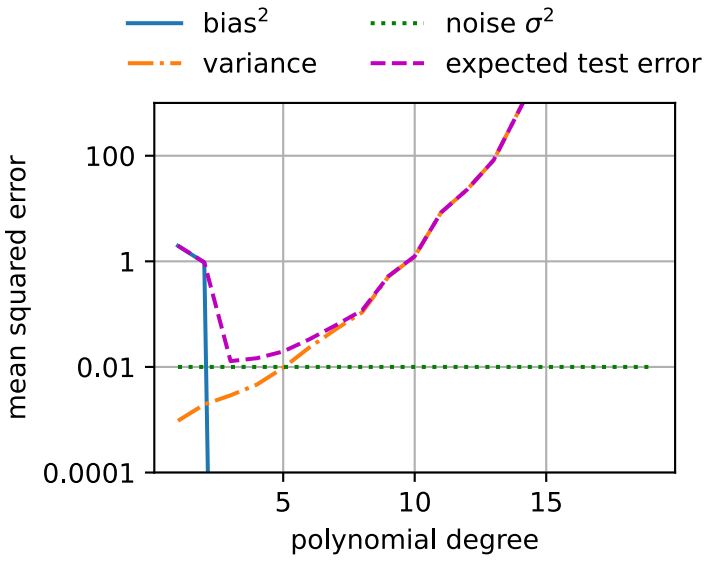
$$\mathbb{E}_{\mathcal{D},\epsilon}\Big[\big(\hat{f}_{\mathcal{D}}(x)-y\big)^2\Big] = \underbrace{\big(\bar{f}(x)-f^*(x)\big)^2}_{\text{bias}^2(x)} + \underbrace{\mathbb{E}_{\mathcal{D}}\Big[\big(\hat{f}_{\mathcal{D}}(x)-\bar{f}(x)\big)^2\Big]}_{\text{variance}(x)} + \underbrace{\sigma^2}_{\text{noise}}. \tag{2.27}$$

あらゆるモデルの期待テスト誤差は、次の 3 つの非負の項の和になる。

1. **バイアスの二乗** $(\bar{f}(x)-f^*(x))^2$：当てはめられたモデルの**平均**（無限に多くの訓練データセットにわたって平均したもの）が、真の関数 $f^*(x)$ からどれだけ離れているかを表す。自由度の低いモデル（3 次曲線に対する直線 $k=1$ など）は高いバイアスをもつ。
2. **分散** $\mathbb{E}_{\mathcal{D}}[(\hat{f}_{\mathcal{D}}(x)-\bar{f}(x))^2]$：ランダムな訓練サンプル $\mathcal{D}$ が変わるたびに、当てはめられたモデルがどれほど敏感に変動するかを表す。過剰に柔軟なモデル（$k=12$）は、20 個の点を新しく抽出するたびに大きく揺れ動き、巨大な分散を抱える。
3. **削減不能ノイズ** σ^2：ラベル y が $f^*(x)$ の周りにもつ固有の分散であり、いかなる予測器もこれを下回ることはできない。

この多項式の実験では、3 つの項すべてを直接計測できる。20 個の訓練入力を固定したまま、ラベルのノイズだけを再抽出しよう。最小二乗法の重みは $\mathbf{y}$ に線形に依存し、かつノイズの平均は 0 であるため、平均的な当てはめ $\bar{f}$ は単にノイズのないラベル $f^*(x^{(i)})$ への当てはめと一致し、ここからバイアスの二乗が厳密に求まる。また、訓練ノイズを独立に 200 回引き直して再当てはめすれば、各テスト点における分散を推定できる。

```
noise = rng.normal(0, sigma, (len(x_train), 200))   # 200 draws of the noise
bias2, var = [], []
for d in degrees:
    w_avg = fit_degree(x_train, target(x_train), d)              # average fit
    W = fit_degree(x_train, target(x_train)[:, None] + noise, d)  # fit per draw
    bias2.append(mse(w_avg, x_test, target(x_test)))
    var.append((features(x_test, d) @ W).var(axis=1).mean())
d2l.plot(degrees, [bias2, var, [sigma**2] * len(degrees),
                   np.add(bias2, var) + sigma**2],
         xlabel='polynomial degree', ylabel='mean squared error',
         fmts=('-', 'C1-.', 'g:', 'm--'), yscale='log', ylim=(1e-4, 1e3))
d2l.plt.legend(['bias$^2$', 'variance', r'noise $\sigma^2$',
                'expected test error'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



これらの曲線がテスト誤差の U 字型を見事に説明していることに注目しよう。バイアスの二乗は次数 1 と 2 では大きいが、次数 3（3 次曲線を含みうる最初のモデル族）で 0 に急落する。一方、分散は次数を増やすごとに上昇し、$\sigma^2(k+1)/n$（1 例あたりのパラメータ数でスケーリングされたノイズ分散。演習問題 4 を参照）付近から始まって、やがてノイズフロアより何桁も高く跳ね上がる。学習不足とは**バイアス**が支配的な誤差であり、過学習とは**分散**が支配的な誤差である。Figure 2.11 に示す理想化されたトレードオフは、古典統計学から機械学習にわたって繰り返し現れる。複雑さが増すにつれて訓練誤差は着実に減少する一方、テスト誤差は中間のスイートスポットで最小値に達する（ただし σ^2 を下回ることは決してない）。

2.6.2.3 モデルの複雑さ

Figure 2.11 が模式的に示すとおり、モデルの複雑さ、すなわちモデル容量（model capacity）を増やすと、バイアスは減少する一方で分散は増大する。最適な汎化性能は、バイアスの二乗と分散の和が最小となるスイートスポットにある。古典的なパラメトリックモデルでは、複雑さはパラメータ数 p と重みの大きさ $\|\mathbf{w}\|$ によって決まる。深層学習（4.5 節）では、正則化（regularization）、アーキテクチャ設計、およびデータセットの規模がこの曲線をどのように塗り替えるかを見ていく。

2.6.2.4 データセットのサイズ

モデルの複雑さを制限する以外に、分散を抑え込む最も直接的な方法は**訓練データを増やす**ことである。多項式の次数 k を固定したとき、当てはめに使う点を $n = 1{,}000$ 個に増やせば（$n = 20$ 個ではなく）、ランダムなノイズが平均化され、高次の係数でさえきれいに定まる（推定分散は $(k+1)\sigma^2/n$ に比例して減衰する）。次数 1、3、19 のモデルが、新たな訓練セットを $n = 20$ 個から $n = 1{,}000$ 個の例へと増やすにつれてどう応答するかを見てみよう。

```
for n in (20, 50, 100, 1000):
    x, y = make_data(rng, n, sigma)          # a fresh training set of size n
    errs = [mse(fit_degree(x, y, d), x_test, y_test) for d in (1, 3, 19)]
    print(f'n = {n:4d}: test error of degrees 1, 3, 19:',
```

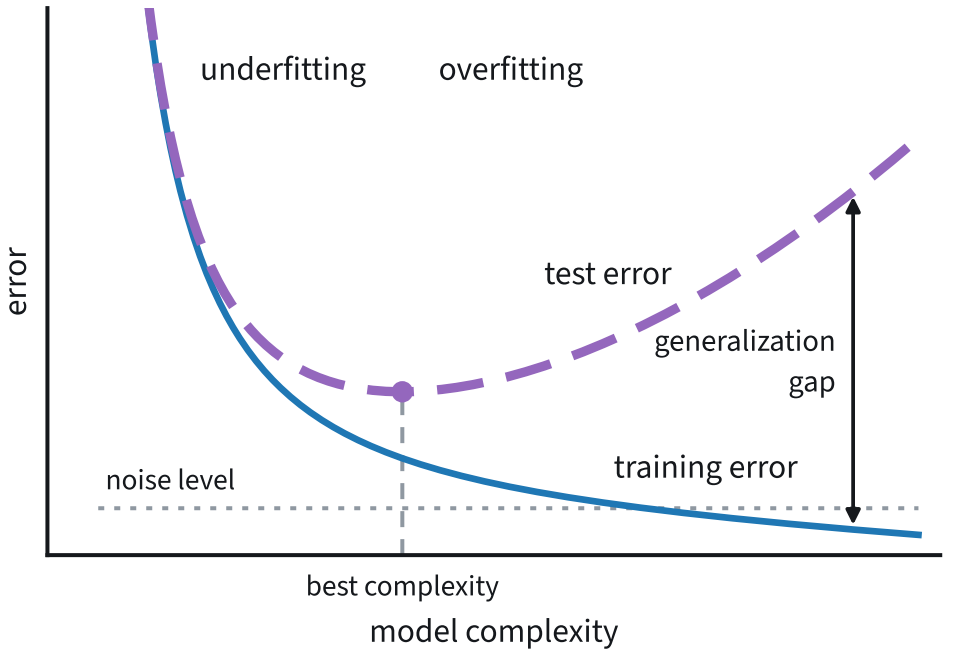


Figure 2.11: モデルの複雑さに対する訓練誤差とテスト誤差。最適な複雑さより左側ではモデルは学習不足となり両方の誤差が高い。右側では過学習し、訓練誤差はノイズレベルを下回って汎化ギャップが広がる。

```
            ', '.join(f'{e:.2g}' for e in errs))
```

```
n =   20: test error of degrees 1, 3, 19: 1.8, 0.013, 1.4e+04
n =   50: test error of degrees 1, 3, 19: 1.9, 0.011, 0.37
n =  100: test error of degrees 1, 3, 19: 1.9, 0.011, 0.014
n = 1000: test error of degrees 1, 3, 19: 1.9, 0.01, 0.011
```

この対比に注目してほしい。データセットを大きくすると、過学習していた 19 次多項式は劇的に救われる。テスト誤差は $n = 100$ までに何桁も急落し、$n = 1{,}000$ では真の 3 次曲線に並ぶ。一方、学習不足の直線はほとんど動かない（テスト誤差は 2 前後に張りついたままである）。データを増やせば**分散**は縮小するが、**バイアス**は決して解消できないからである。モデルが単独で「複雑すぎる」ということはなく、**それを制約するために利用できる信号やデータに対して**複雑すぎるにすぎない。このスケーリングの原理こそ、大規模なデジタルデータセットが当たり前になったことで深層学習が飛躍した大きな理由の 1 つである（ただし 4.5 節で見るように、深いネットワークはこの古典的な曲線に重要なひねりを加えることになる）。

2.6.3 モデル選択

2.6.3.1 検証セット

先ほどの合成データの実験では $f^*(x)$ が既知であり、1,000 個のテスト点を自在に生成できた。では現実の問題では、多項式の次数 k、学習率（learning rate）η、重み減衰の強さ λ といったハイパーパラメータをどのように選べばよいのだろうか。

訓練誤差の最小化によってハイパーパラメータを選ぶことは決してできない。訓練誤差は常に、最も複雑で丸暗記しやすいモデル（$k = 19$）を優遇してしまうからである。かといって、最終的な**テストセット**を繰り返し参照してハイパーパラメータを調整すべきでもない。そうするとテストセットのノイズが設計上の選択に漏れ出してしまう（第 3.5 章）。その代わりに、手元のデータを互いに素な 3 つの分割へと切り分ける。

1. **訓練セット**：ハイパーパラメータの各候補設定のもとで、モデルパラメータ $(\mathbf{w}, b)$ を当てはめるために用いる。
2. **検証セット**：ハイパーパラメータの候補設定どうしを比較し、最良のものを選ぶために用いる（**モデル選択**）。
3. **テストセット**：厳重に保管しておき、選択されたモデルの汎化誤差の不偏推定値を報告するために、最後の最後に **1 回だけ**評価する。

2.6.3.2 交差検証

データセットが小さい場合（たとえば $n = 20$ 件や $n = 100$ 件の臨床記録）、固定された 20% の検証セットを切り出すと、訓練に使える例も信頼できる検証に使える例も少なすぎることになる。K **分割交差検証**（Figure 2.12）は、すべての例に二役を担わせる手法である。まず、データセットをランダムに K 個の等サイズで互いに素な部分集合（**フォールド**）$\mathcal{D}_1, \ldots, \mathcal{D}_K$ へと分割する（通常は $K = 5$ または $K = 10$）。ラウンド $i \in \{1, \ldots, K\}$ では、$K - 1$ 個のフォールド $\mathcal{D} \setminus \mathcal{D}_i$ 上でモデルを最初から訓練し、ホールドアウトしたフォールド $\mathcal{D}_i$ 上で検証損失を評価する。この K 個のホールドアウト損失を平均したものが K 分割交差検証スコアとなり、最良のハイパーパラメータが選ばれたら、全 n 個の例を使ってモデルを再訓練する。なお、極端な $K = n$ の場合は **1 つ抜き交差検証**（leave-one-out cross-validation）と呼ばれ、各フォールドは単一のデータ点となる。線形最小二乗法では、驚くべき代数的恒等式により、**1 回**の当てはめから n 個すべての 1 つ抜き残差（residual）を計算でき、n 回の再訓練は不要になる（演習問題 6 を参照）。

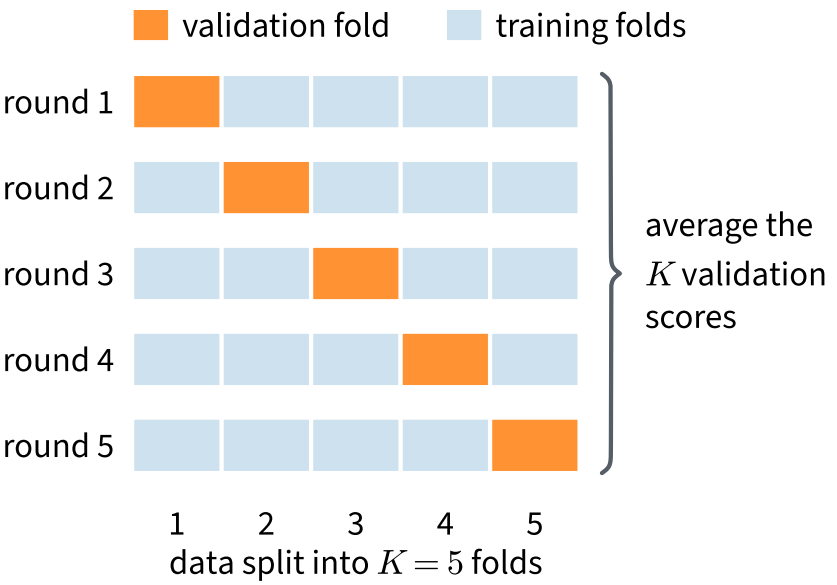


Figure 2.12: K 分割交差検証（図は $K = 5$ の場合）では、データセットを K 個のフォールドに分割し、各ラウンドで 1 つのフォールドを検証用（オレンジ色）として残し、残りの $K - 1$ 個のフォールドで訓練する。

実際のデータセットはソートされた状態で届くことが多く、また 1 フォールドあたり 4 点しかない場合は 1 回の分割だけではノイズが大きいため、フォールドは**ランダム**な並べ替えから作る必要がある。そこで以下の `kfold_splits` では、$K = 5$ 個のフォールドへの独立なランダム分割を 10 回生成し（**反復**交差検証）、`cv_error` では多項式の当てはめの検証誤差をそれらすべてのフォールドにわたって平均する。

```
def kfold_splits(rng, n, K=5, repeats=10):
    """Random partitions of the indices 0, ..., n-1 into K folds."""
    return [np.array_split(rng.permutation(n), K) for _ in range(repeats)]

def cv_error(x, y, d, splits):
    """Validation error of the degree-d fit, averaged over all folds."""
    return np.mean([mse(fit_degree(np.delete(x, f), np.delete(y, f), d),
                        x[f], y[f]) for folds in splits for f in folds])
```

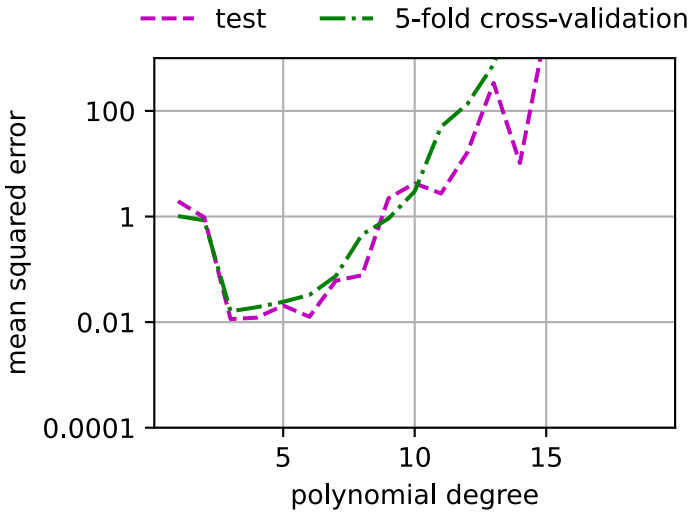
20 個の訓練例だけを使い、公平な比較のためにすべての次数を同じフォールド上で評価して多項式の次数を選択し、その選択が真のテストセット上の最適値（交差検証は一切参照しない）とどれほど一致するか確かめてみよう。

```
splits = kfold_splits(rng, len(x_train))  # the same folds for every degree
cv_err = [cv_error(x_train, y_train, d, splits) for d in degrees]
print('cross-validation picks degree:', degrees[np.argmin(cv_err)])
print('lowest test error at degree:  ', degrees[np.argmin(test_err)])
```

```
cross-validation picks degree: 3
lowest test error at degree:   3
```

全 19 個の次数にわたり、5 分割交差検証の曲線をテスト誤差の曲線と並べてプロットしてみよう。

```
d2l.plot(degrees, [test_err, cv_err], xlabel='polynomial degree',
         ylabel='mean squared error', fmts=('m--', 'g-.'), yscale='log',
         ylim=(1e-4, 1e3))
d2l.plt.legend(['test', '5-fold cross-validation'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



テストセットを一度も覗き見ることなく、反復 5 分割交差検証はテスト誤差の谷底を確実に捉え、通常は次数 3 を、時折テスト誤差がほとんど区別できない近傍の次数を選択する。最小値の付近では 2 つの曲線は小さな倍率の範囲内で追従し、交差検証のほうが 20 点で平均しているため（1,000 点ではなく）、当然ながら少しノイズが多い。高次では両方の曲線が何桁も跳ね上がる。交差検証には、テスト誤差よりさらに早く上昇し始める理由がある。各フォールドは 16 点だけで訓練するため（20 点ではなく）、次数 15 で既に 16 個の訓練点すべてを補間してしまい、$[-1, 1]$ の端付近にある検証点は当てはめ範囲の外に出て高次多項式が爆発する領域に入るからである。もっとも、20 点の単一の抽出では、どちらの曲線が先に上昇することもありうる。

K はどのように選ぶべきだろうか。各フォールドはデータセットの $(K-1)/K$ の割合でしか訓練しないため、当てはめられたモデルは全 n 個の例で訓練したモデルよりわずかに劣り、K 分割交差検証にはわずかに悲観的な統計的バイアスが生じる（演習問題 5 で厳密に導出する）。$K = n$（**1 つ抜き**交差検証）まで増やせば、このバイアスはほぼ解消される。しかし一般の学習器では、候補ごとに n 回の個別の当てはめが必要になる。さらに決定木のような不安定な学習器では、1 個の例を除いただけでモデル全体が変わりうるため、ほぼ同一の n 個の訓練セットを使うことで、より粗い分割よりも推定値のノイズがかえって大きくなる場合がある。1 回の当てはめが高コストな場合に $K = 5$ や $K = 10$ が標準の既定値とされるのはこのためである (Kohavi 1995)。一方、最小二乗法は高コストでも不安定でもない。1 回の当てはめから n 個すべての 1 つ抜き残差が得られ、データに対して当てはめ結果も滑らかに変化する。

この 3 次多項式について、バイアスとノイズを計測してみよう。500 個の新たな 20 点の訓練セットのそれぞれにおいて、全 20 点に 3 次多項式を当てはめてテスト誤差を記録するとともに、その 20 点だけを使った 3 通りの誤差推定値を記録する。すなわち、1 つ抜き法（$K = 20$、各フォールド 1 点）、単一のランダム分割による 5 分割交差検証、および 10 回の分割にわたって反復した 5 分割交差検証である。

```
loo = kfold_splits(rng, 20, K=20, repeats=1)  # K = n: one point per fold
est = {'test error': [], 'leave-one-out': [], '5-fold, 1 split': [],
       '5-fold, 10 splits': []}
for _ in range(500):                          # 500 fresh training sets
    x, y = make_data(rng, 20, sigma)
    est['test error'].append(mse(fit_degree(x, y, 3), x_test, y_test))
```

```
        est['leave-one-out'].append(cv_error(x, y, 3, loo))
        est['5-fold, 1 split'].append(
            cv_error(x, y, 3, kfold_splits(rng, 20, repeats=1)))
        est['5-fold, 10 splits'].append(cv_error(x, y, 3, kfold_splits(rng, 20)))
    for name, e in est.items():
        print(f'{name:18s} mean {np.mean(e):.4f}   std {np.std(e):.4f}')
```

```
test error         mean 0.0135   std 0.0045
leave-one-out      mean 0.0149   std 0.0252
5-fold, 1 split    mean 0.0200   std 0.0519
5-fold, 10 splits  mean 0.0175   std 0.0320
```

平均すると、1 つ抜き法はテスト誤差の近く、その約 10 分の 1 以内の差に収まる。5 分割の推定値はいずれも 1 つ抜き法の推定値を上回っており、19 点ではなく 16 点で訓練することによる悲観的なバイアスを反映している。3 つの交差検証推定値の中で、1 つ抜き法は 500 回の抽出にわたる標準偏差（standard deviation）も最小であり、1 つ抜き法のほうがノイズが大きいという通説はここでは成り立たない。5 分割を 10 回繰り返すとばらつきは狭まるものの、1 つ抜き法ほどには小さくならない。ばらつきの多くは、20 点をどう分割したかではなく、どの 20 点が抽出されたかという運に由来するためである。したがって最小二乗法においては、1 つ抜き法は計算が軽いだけでなく 3 つの中で最も精度が高い。一方、後の章で扱うニューラルネットワークのように 1 回の当てはめに数分から数時間かかる場合には、粗い分割が依然として現実的な選択肢となる。

ここで離散的な多項式の次数の選択がうまくいったのは、特徴量に $1, x, x^2, \dots, x^k$ という自然な階層があったからである。では、順序のない何百もの特徴量をもつデータセットが与えられ、列を 1 つずつ落とす自明な方法がない場合はどうすべきだろうか。2.7 節では、離散的な特徴選択を、重みのノルムに対する連続的なペナルティへと置き換える。

2.6.4 まとめ

教師あり学習は、有限の訓練サンプル上で経験リスク $R_{\text{emp}}[f]$ を最小化しつつ、未知のデータ上で低い母集団リスク $R[f] = \mathbb{E}[l(f(\mathbf{x}), y)]$ を達成することを目指す。最適化は訓練セットに適合するよう $\hat{f}$ を特別に選び出すため、モデルクラスがラベルのノイズを吸収できるほど柔軟である限り、$R_{\text{emp}}[\hat{f}]$ には下向きの楽観的な統計的バイアスが生じる。テスト誤差の二乗の期待値は、バイアスの二乗、推定分散、および削減不能ノイズ σ^2 へと分解される。自由度の低いモデルは高いバイアスを伴って学習不足に陥り、小さなサンプル上の過剰に柔軟なモデルは高い分散を伴って過学習し、訓練データを増やすとバイアスは減らないまま分散が $1/n$ に比例して縮小する。訓練誤差は常に丸暗記を優遇し、また最終的なテストセットへの繰り返しの問い合わせはテストセットを汚染するため、ハイパーパラメータはホールドアウトされた検証分割、または K 分割交差検証によって選ばなければならない。

2.6.5 演習問題

1. **IID が成り立たないとき。**観測どうしに従属性があるためにデータセットを IID として扱うべきでない例を 3 つ挙げよ。それぞれについて、訓練と検証へのランダム分割が汎化誤差を過大評価するか過小評価するかを述べ、その問題を回避する分割方法を説明せよ。
2. **厳密な多項式回帰。**n 個の訓練入力 $x^{(1)}, \dots, x^{(n)}$ が互いに異なるとき、次数 $n-1$ の多項式が n 個のラベルに厳密に適合することを示し、入力が互いに異なるという条件がどこで使われるかを説明せよ。2 つの入力が一致し、それらのラベルが異なるとき、最小二乗法は何を返すか。

3. [code] **次数 19 のためのより良い基底**。高次のべき乗 x^{17} と x^{19} は $[-1, 1]$ 上のどこでも差が 0.05 未満であるため、`features(x, 19)` の列ベクトルはほぼ平行になる。Chebyshev 多項式 $T_0, \dots, T_{19}$ は同じ関数空間を張るが、この区間上で -1 から 1 の間に収まる。

   1. まず予測を立てよ。Chebyshev 基底を用いて 20 個の訓練点に次数 19 を当てはめたとき、訓練誤差、テスト誤差、最大の重みのうちどれが、どの程度変化するか。
   2. 複数回実行してその予測を確かめよ。訓練入力とテスト入力の両方で `features(x, 19)` を `np.polynomial.chebyshev.chebvander(x, 19)` に置き換え、2 つの計画行列の条件数（`np.linalg.cond`）、訓練誤差とテスト誤差、および最大の重みを比較せよ。
   3. 厳密演算では 2 つの当てはめが同一の多項式になる理由と、float64 ではどの数値がなぜ異なるのかを説明せよ。より良い基底を使えば 20 点に対する次数 19 は救われるか。この結果から、複雑さの尺度としての重みの大きさについて何が言えるか。

4. **訓練誤差の楽観性**。●列フルランクの固定された $n \times p$ 計画行列 $\mathbf{X}$ と、ラベル $\mathbf{y} = \mathbf{X}\mathbf{w}^* + \epsilon$（ノイズ項は互いに独立で平均 0、分散 σ^2）に対する最小二乗法を考える。当てはめによる予測値は $\hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$（ただし $\mathbf{H} = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$）である。

   1. 期待訓練誤差が $\mathbb{E}[\|\mathbf{y} - \hat{\mathbf{y}}\|^2]/n = \sigma^2(n-p)/n$ となることを示せ。ヒント：$\mathbf{y} - \hat{\mathbf{y}} = (\mathbf{I} - \mathbf{H})\epsilon$ であり、$\mathbf{I} - \mathbf{H}$ のトレースは $n - p$ である。
   2. 同じ入力に対して新たなラベル $\mathbf{y}' = \mathbf{X}\mathbf{w}^* + \epsilon'$ を抽出する。これに対する $\hat{\mathbf{y}}$ の期待誤差が $\sigma^2(n+p)/n$ となり、したがって訓練誤差はそれを $2\sigma^2 p/n$ だけ過小評価することを示せ。また、$\sigma^2 p/n$ が n 個の入力にわたって平均した予測値の分散であることを示せ。
   3. 2.6.2.1 節の 3 次多項式の当てはめ（$n = 20$、$p = 4$、$\sigma = 0.1$）について両方の式を計算し、実験を複数回行ったときの次数 3 の訓練誤差およびテスト誤差と比較せよ。その違いを説明せよ。すなわち、1 回の実行における訓練誤差はその期待値の周りにどの程度ばらつくか、また訓練入力で成り立つ (b) の公式がテスト入力にはなぜ厳密に当てはまらないのか。

5. K **分割交差検証**。

   1. m 個の例で訓練されたモデルの期待汎化誤差が $R(m) = R_\infty + c/m$（ただし $c > 0$）であるとする。各フォールドのモデルは $(K-1)n/K$ 個の例で訓練される。K 分割交差検証の推定値がもつ統計的バイアスを、$R(n)$（全 n 個の例で訓練されたモデルの誤差）を基準として計算せよ。このバイアスを $R(n) - R_\infty$ に対する比として表し、$K = 2$、5、10、および n についてその値を求めよ。
   2. 交差検証でハイパーパラメータを選択したのち、最終モデルは全 n 個の例で再訓練される。最適な値が訓練セットのサイズに依存するハイパーパラメータがこの手順を難しくする理由を説明し、その具体例を 1 つ挙げよ。

6. **1 つ抜き法の近道**。●一般の学習器では、1 つ抜き交差検証に n 回のモデルの当てはめが必要となる。計画行列 $\mathbf{X}$（各行は $\mathbf{x}^{(i)\top}$）を用いた最小二乗法を考える。$\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ とおき、$\mathbf{X}$ から任意の 1 行を取り除いても正則性が保たれると仮定する。$\hat{y}^{(i)}$ を、$\mathbf{x}^{(i)}$ における全 n 個の例への当てはめの予測値とし、$\hat{y}^{(-i)}$ を、$\mathbf{x}^{(i)}$ における例 i を除いた当てはめの予測値とし、$h_i = \mathbf{x}^{(i)\top}\mathbf{A}^{-1}\mathbf{x}^{(i)}$ とおく。ここで、Sherman–Morrison の恒等式（33.2 節）は次のとおりである。

$$(\mathbf{A} - \mathbf{u}\mathbf{u}^\top)^{-1} = \mathbf{A}^{-1} + \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{u}^\top\mathbf{A}^{-1}}{1 - \mathbf{u}^\top\mathbf{A}^{-1}\mathbf{u}}$$

   この式で $\mathbf{u} = \mathbf{x}^{(i)}$ とおくことにより、次式が成り立つことを示せ。

$$y^{(i)} - \hat{y}^{(-i)} = \frac{y^{(i)} - \hat{y}^{(i)}}{1 - h_i}. \tag{2.28}$$

さらに、最小二乗法における 1 つ抜き交差検証のコストが、1 回の当てはめと比べてどうなるかを述べよ。

2.7 重み減衰

2.6 節では、多項式特徴量の離散的な個数 $k+1$ を変えるという、最も大雑把な方法でモデルの複雑さを制御した。しかし、離散的な特徴量選択はすぐに壁に突き当たる。数百から数千の特徴量を持つ高次元の問題では、特徴量の部分集合は 2^d 通りも存在する。さらに悪いことに、すでに見たとおり、多項式が過学習すると、ほぼ共線関係にある特徴量どうしを打ち消し合うために重み w_j が数百万以上にまで爆発してしまう。

この本質的な病理を 2 つの特徴量からなる最も単純な形で確認するため、隣り合わせに置かれた 2 本のノイズを含む温度計 x_1 と x_2 から室温を予測する状況を考えよう。それぞれの温度計は、真の室温にわずかで独立なノイズの揺らぎが加わった値を示すものとする。ここで、重みの和が 2 になる 2 つの線形モデルを考える。

- **モデル A** は $\mathbf{w} = (1, 1)^\top$ とおき、2 つのセンサの平均をとることでセンサのノイズ分散を半分に抑える。
- **モデル B** は $\mathbf{w} = (501, -499)^\top$ とおく。2 本の温度計が一致しているときは $501x_1 - 499x_2 = 2x_1$ となって同じ平均を与えるが、2 つのセンサ間にわずかなずれ $x_1 - x_2$ があるだけで、それが 500 倍に増幅されてしまう。

小さな訓練サンプルにおいて、たまたま $x_1 - x_2$ がラベルのノイズとわずかに相関していた場合、正則化なしの最小二乗法は訓練誤差をほんの少し削るために $(501, -499)^\top$ を選んでしまい、新しい測定値に対して大きく破綻する。どうすれば d 個の特徴量をすべてモデルに残したまま、定まりの悪い方向に沿って重みが爆発するのを防げるだろうか。

統計学では**リッジ回帰**（ridge regression）または **Tikhonov 正則化**（Tikhonov regularization）として知られ (Hoerl and Kennard 1970; Tikhonov and Arsenin 1977)、深層学習では**重み減衰**（weight decay）と呼ばれる古典的な解決策は、二乗ユークリッドノルム $\|\mathbf{w}\|_2^2$ に対する滑らかなペナルティを損失関数に直接加えることである。本節では、相補的な 4 つの視点（ペナルティ付き最適化、SGD におけるステップごとの乗法的減衰、ベイズ的なガウス事前分布、および $\mathbf{X}^\top\mathbf{X}$ の固有ベクトルに沿ったスペクトル縮小）から重み減衰を導出し、4 つのフレームワークすべてにおいてスクラッチ実装と簡潔な実装の両方を示す。

2.7.1 ノルムペナルティ

2.7.1.1 ペナルティ付き損失

特徴量の個数を制限する代わりに、重みベクトルの二乗 ℓ_2 ノルム $\|\mathbf{w}\|_2^2 = \sum_{j=1}^{d} w_j^2$ にペナルティを課すことで、線形モデルの実効的な容量を制御する。平均二乗誤差損失 $L(\mathbf{w}, b)$ に、非負の**正則化定数**（regularization constant）$\lambda \geq 0$ を掛けたこのペナルティを加えると、次の正則化目的関数が得られる。

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2. \tag{2.29}$$

ここでも係数 $\frac{1}{2}$ は、導関数に現れる 2 を $\|\mathbf{w}\|_2^2$ の微分時に打ち消すために付けてある。Equation 2.29 の 3 つの基本的な性質に注目しよう。

1. **λ によるトレードオフ：**$\lambda = 0$ とおけば正則化なしの最小二乗法に戻る。一方、λ を大きくすると大きな重みにより厳しいペナルティが課され、$\mathbf{w}$ は $\mathbf{0}$ に引き寄せられる（そして $\lambda \to \infty$ の極限では $\mathbf{w} \to \mathbf{0}$ となる）。
2. **バイアス b にペナルティを課さない理由：**バイアス b は、出力の平均レベルを目的変数 y の平均に合わせるために平行移動させるだけの役割を持つ。もし b を 0 に向かって縮小させると、たとえば y が摂氏ではなくケルビンで測られていた場合、入力の摂動に対するモデルの感度を少しも下げないまま、すべての予測を 0 の方向へ偏らせてしまう。
3. **特徴量のスケーリングの重要性：**$\sum_j w_j^2$ は座標ごとに生の二乗係数を足し合わせるため、ある 1 つの特徴量をメートルではなくミリメートルで表し直すと、その重み w_j は 1,000 で割られることになり、ペナルティへの寄与は 10^6 分の 1 に縮小する。実際には、等方的な $\|\mathbf{w}\|_2^2$ ペナルティを適用する前に、各特徴量を同程度のスケール（平均 0、単位分散）に標準化する。

なぜ二乗 ℓ_2 ノルムにペナルティを課すのだろうか。2 本の温度計の例では、$(1, 1)^\top$ のときは $\|\mathbf{w}\|_2^2 = 2$ であるのに対し、$(501, -499)^\top$ のときは $\|\mathbf{w}\|_2^2 = 500{,}002$ となる。そのため、ごく小さな $\lambda > 0$ であっても、2 つのセンサの差をとるより平均をとる方が強く選好される。

2.7.1.2 勾配ステップにおける重み減衰

ペナルティ $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ によって、ミニバッチ SGD の更新則はどう変わるだろうか。$\nabla_{\mathbf{w}} \left(\frac{\lambda}{2}\|\mathbf{w}\|_2^2\right) = \lambda \mathbf{w}$ であるから、ペナルティ付きの勾配ステップは次のようになる。

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left(\frac{\partial L}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) = (1 - \eta\lambda)\mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}. \tag{2.30}$$

右辺の第 1 項に注目しよう。**通常のデータに基づく勾配ステップ $-\eta\, \partial L/\partial \mathbf{w}$ を適用する前に、重みベクトルに $(1 - \eta\lambda) < 1$ を掛けて、更新のたびに 0 に向かってわずかに縮小（「減衰」）させている**。この乗法的な縮小因子 $(1 - \eta\lambda)$ こそ、深層学習の実践者が ℓ_2 正則化を**重み減衰**と呼ぶ理由である（通常の SGD では、損失に $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ を加えることと重みを $(1 - \eta\lambda)$ 倍に減衰させることは代数的に同一であるが、第 9 章で扱う Adam のような適応的オプティマイザでは、乗法的な重み減衰を分離して **AdamW** とすることが重要な違いを生む）。

2.7.1.3 ℓ_1 ペナルティとスパース性

二乗 ℓ_2 ノルム $\|\mathbf{w}\|_2^2 = \sum_j w_j^2$ を ℓ_1 ノルム $\|\mathbf{w}\|_1 = \sum_j |w_j|$ に置き換えたらどうなるだろうか。これは統計学で **Lasso** と呼ばれる手法である (Tibshirani 1996)。両者の制約集合の幾何学的形状を比較すると（Figure 2.13）、際立った違いが見えてくる。データ損失 $L(\mathbf{w})$ を制約 $\|\mathbf{w}\|_2^2 \le r^2$ のもとで最小化すると、滑らかな円板（$\mathbb{R}^d$ においては球体）の内部を探索することになり、楕円状の損失の等高線はほぼ常に、**すべての**座標 w_j が小さいながらも非ゼロである点で境界に接する。これに対し、$L(\mathbf{w})$ を制約 $\|\mathbf{w}\|_1 \le r$ のもとで最小化すると、頂点が座標軸上に直接乗った鋭い角を持つひし形（正軸体）の内部を探索することになる。楕円状の損失の等高線は、しばしばこうした鋭い角の 1 つに最初にぶつかるため、多くの座標 w_j が**厳密に 0** となり、自動的な特徴量選択、すなわち**スパース性**（sparsity）が実現される。

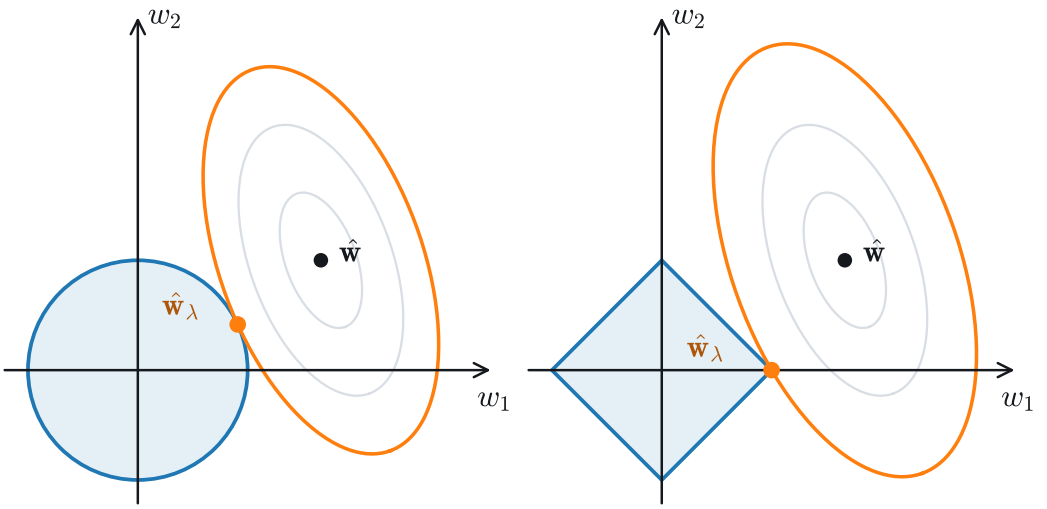


Figure 2.13: ℓ_2 リッジ回帰（左）と ℓ_1 Lasso 回帰（右）の制約の幾何学的形状。データ損失 $L(\mathbf{w})$ の楕円状の等高線は、正則化なしの最小点 $\hat{\mathbf{w}}$ から広がり、通常は軸から離れた位置で滑らかな ℓ_2 の円に接する（両方の重みが非ゼロに保たれる）のに対し、ℓ_1 のひし形では軸上にある鋭い角にぶつかることが多い（$w_2 = 0$ となる）。

2.7.1.4 重みに対するガウス事前分布

2.1.3 節においてガウス観測ノイズのもとでの最尤法から二乗データ損失が導かれたのと同様に、ベイズ統計学からは重み減衰に対しても同じくらい明快な確率的解釈が得られる。データを一切観測する前に、重みに対して平均 0 の等方的なガウス**事前分布**（prior distribution）を独立に仮定し、$w_j \sim \mathcal{N}(0, \tau^2)$、すなわち $p(\mathbf{w}) \propto \exp(-\|\mathbf{w}\|_2^2/(2\tau^2))$ とおく一方、ラベルは $y \mid \mathbf{x}, \mathbf{w}, b \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x} + b, \sigma^2)$ に従うものとする。

ベイズの定理により、**事後密度**（posterior density）$p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, b)\, p(\mathbf{w})$ を最大化することは、その負の対数を最小化することと同値である。これが**最大事後確率推定**（maximum a posteriori estimation、MAP 推定）であり、最小化する式は次のようになる。

$$\frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b \right)^2 + \frac{1}{2\tau^2} \|\mathbf{w}\|_2^2. \tag{2.31}$$

全体を n/σ^2 で割ると、重み減衰の目的関数 Equation 2.29 が $\lambda = \sigma^2/(n\tau^2)$ として得られる。この比がいかに理にかなっているかに注目しよう。重みが小さいという事前信念が強いほど（τ^2 が小さいほど）、あるいは観測ノイズが大きいほど（σ^2 が大きいほど）、より強い正則化 λ が求められる。一方で、サンプルサイズ $n \to \infty$ の極限では、豊富なデータ自体が十分に語るようになるため、ペナルティは $\lambda = \sigma^2/(n\tau^2) \to 0$ となる。

2.7.2 重み減衰のスクラッチ実装

訓練例の数とほぼ同じ数の重みを持つモデルを重み減衰がいかに救うかを確認するため、次元 $d = 200$ が訓練サンプルサイズ $n_{\text{train}} = 300$ に迫る高次元の合成回帰問題を構成しよう。具体的には、入力 $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{200})$ をサンプリングし、次の式からラベルを生成する。

$$y = 0.05 + \sum_{j=1}^{200} 0.1\, x_j + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, 1), \tag{2.32}$$

ここでは300個の訓練例（ミニバッチサイズ20）と1,000個の検証例を、SyntheticRegressionData（2.4節）を用いて用意する。真の重み $\mathbf{w}^* = (0.1, \ldots, 0.1)^\top$ の各座標のスケールは $\tau = 0.1$ であり、ノイズはかなり大きい。その標準偏差 $\sigma = 1$ は、ノイズのない信号の標準偏差 $\|\mathbf{w}^*\| = \sqrt{2} \approx 1.4$ に匹敵する。原理的には300個の例があれば200個すべての重みを特定できるものの、その線形結合の多くは弱くしか定まらないことがまもなくわかる。

```
tau, sigma, n = 0.1, 1.0, 300   # size of the weights, noise, examples
data = d2l.SyntheticRegressionData(w=d2l.ones(200) * tau, b=0.05,
                                   noise=sigma, num_train=n, num_val=1000,
                                   batch_size=20)
```

訓練を始める前に、検証損失に対する2つの基準値を設定しておこう。独立な標準正規入力 $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ とノイズ $\epsilon \sim \mathcal{N}(0, \sigma^2)$ に対し、$(\mathbf{w}, b^*)$ における期待損失は $\frac{1}{2}\|\mathbf{w} - \mathbf{w}^*\|_2^2 + \frac{1}{2}\sigma^2$ に等しい（2.3節）。したがって、200個の入力をすべて無視して常に b^* と予測する自明な定数モデルの期待損失は $\frac{1}{2}(\|\mathbf{w}^*\|^2 + \sigma^2) = \frac{1}{2}(2 + 1) = 1.5$ となる一方、オラクルのパラメータ $(\mathbf{w}^*, b^*)$ は $\sigma^2/2 = 0.5$ を達成する。学習されたモデルが200個の重みを持つ意味があるといえるには、検証損失が1.5を十分に下回らなければならない。

2.7.2.1 ペナルティ付きモデル

l2_penalty(w) を定義し、$\frac{1}{2}\|\mathbf{w}\|_2^2$ を返すようにする。

```
def l2_penalty(w):
    return d2l.reduce_sum(w**2) / 2
```

LinearRegressionScratch（2.4節で実装した、2.3節のスクラッチモデルのオブジェクト指向版）を継承してサブクラスを定義する。変更が必要なのは training_step だけであり、微分と最適化のためにペナルティ付き目的関数 Equation 2.29 を返すようにする。重要な点として、validation_step は引き続きペナルティなしの二乗損失を報告する。これにより、異なる λ の値どうしでも、検証損失を予測精度の尺度として直接比較できる。

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def training_step(self, batch):
        loss = super().training_step(batch)
        return loss + self.lambd * l2_penalty(self.w)
```

2.7.2.2 ペナルティあり・なしでの訓練

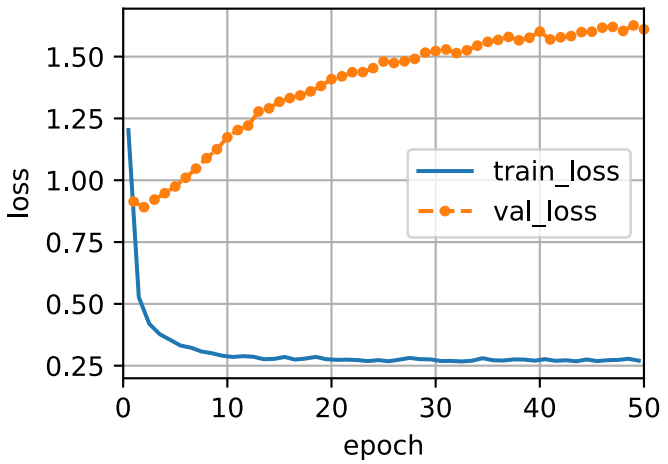
train_scratch(lambd, data) を作成し、WeightDecayScratch を50エポック、学習率 $\eta = 0.05$（750回のSGDステップ）で訓練しながら訓練曲線と検証曲線をプロットし、当てはめたモデル、エポックごとの検証損失、および学習された重みのノルム $\|\mathbf{w}\|_2$ を返すようにしよう（比較のた

め、真の重みベクトル $\mathbf{w}^* = (0.1, \ldots, 0.1)^\top \in \mathbb{R}^{200}$ のノルムは $\|\mathbf{w}^*\|_2 = \sqrt{200 \times 0.1^2} \approx 1.4$ である)。

```
def train_scratch(lambd, data):
    trainer = d2l.Trainer(max_epochs=50)
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.05)
    trainer.fit(model, data)
    norm = float(torch.linalg.vector_norm(model.w.detach()))
    return model, [float(v) for v in trainer.history['val_loss']], norm
```

まず `lambd = 0` とし、正則化を完全に無効にして訓練する。

```
model_0, val_losses, norm = train_scratch(0, data)
```



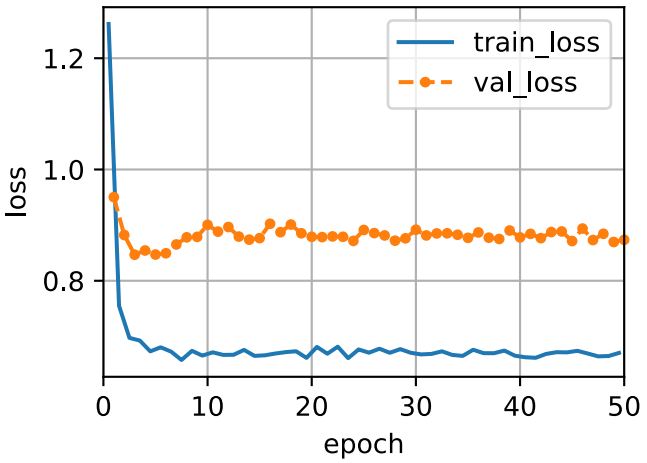

```
print(f'validation loss {val_losses[-1]:.3f} (lowest {min(val_losses):.3f}, '
      f'epoch {val_losses.index(min(val_losses)) + 1}), norm of w {norm:.2f}')
```

```
validation loss 1.610 (lowest 0.891, epoch 2), norm of w 2.03
```

訓練損失は約 0.2〜0.3 まで急落する。これは真のパラメータが達成する削減不能なノイズ下限 0.5 を大きく下回っており、モデルが 300 個の訓練ラベルに含まれるノイズを丸暗記していることを裏づけている。一方、検証損失はわずか 2〜3 エポック後に最小値 0.8〜1.0 に達したのち着実に上昇し、多くの場合 1.1〜1.9 の間で終了する。50 エポック後には、200 個の重みを持つこのモデルは序盤の改善の大半を失い、自明な定数予測 b^* の損失 1.5 と同程度（ときにはそれ以下）の性能にとどまる。まさに教科書どおりの過学習である。なぜ検証損失は、わずか 30〜45 回の勾配ステップというこれほど早い段階で底を打つのだろうか。その理由は、勾配降下法が重み空間のどの方向を先に学習するかを調べれば明らかになる。

次に、重み減衰を有効にする。λ を当て推量で決めるのではなく、ガウス事前分布の式を参照しよう。$\sigma = 1$、$n = 300$、真の重みのスケール $\tau = 0.1$ に対して、MAP 推定は $\lambda = \sigma^2/(n\tau^2) = \frac{1}{3}$ を与える。

```
lambd = sigma**2 / (n * tau**2)   # the value the prior suggests, 1/3
model_lambd, val_losses, norm = train_scratch(lambd, data)
```




```
print(f'validation loss {val_losses[-1]:.3f}, norm of w {norm:.2f}')
```

```
validation loss 0.874, norm of w 1.17
```

今度は訓練曲線がペナルティ付き目的関数を追跡し、0.5 を安定して上回り続ける。ペナルティによって、モデルが訓練ラベルのノイズを追いかけるのが抑えられているためである。検証損失は約 0.75〜1.05 に落ち着いて横ばいとなり、定数予測器の基準値 1.5 を大きく下回る。同時に重みのノルムは約 2 から約 1.2 へと縮小し、実際には真のノルム $\|\mathbf{w}^*\| \approx 1.4$ よりも**短く**なる。正則化された重みは真の長さより短く縮小されているにもかかわらず、正則化なしの重みより劇的によく汎化する。

ここで得られた効果を率直に評価しておこう。50 エポック最後まで訓練したペナルティなしモデルと比べれば、重み減衰は検証損失を 4 分の 1 から半分も削減している。しかし、同じペナルティなしモデルを最良のエポックで止めた場合と比べると、改善幅は小さく、どちらも約 0.8〜1.0 に達する。実用上の違いは安定性にある。重み減衰は目的関数そのものを変えるため、どれほど長く訓練しても解の良さが保たれるのに対し、ペナルティなしの実行は訓練損失だけでは見当もつかない絶妙なタイミングで止めなければならない。正則化と早期停止（early stopping）がいずれも**なぜ**このように振る舞うのかを理解するため、重みベクトルの内部に踏み込み、ペナルティが実際にどの成分を取り除いたのかを調べてみよう。

2.7.3 固有ベクトルに沿った縮小

$\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ を加えることで、単にすべての重みを一様に縮小するのではなく、なぜこれほど劇的に検証誤差が改善するのだろうか。特徴量の共分散行列の固有基底においてリッジ目的関数の厳密な最小点を調べることで、この問いに閉じた形で答えることができる。

2.7.3.1 リッジ推定量

特徴量とラベルが中心化されていて $b = 0$ であるとし、リッジ目的関数 Equation 2.29 を行列形式で書き下す。

$$L_\lambda(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2. \tag{2.33}$$

勾配を $\nabla_{\mathbf{w}} L_\lambda(\mathbf{w}) = \frac{1}{n}\mathbf{X}^\top(\mathbf{X}\mathbf{w}-\mathbf{y}) + \lambda\mathbf{w} = \mathbf{0}$ とおくと、次の正則化正規方程式が得られる。

$$\begin{aligned}\left(\frac{1}{n}\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\hat{\mathbf{w}}_\lambda &= \frac{1}{n}\mathbf{X}^\top\mathbf{y},\\ \hat{\mathbf{w}}_\lambda &= \left(\mathbf{X}^\top\mathbf{X} + n\lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\mathbf{y}.\end{aligned} \tag{2.34}$$

ここから直ちに導かれる代数的帰結に注目しよう。たとえ $\mathbf{X}^\top\mathbf{X}$ が特異であっても（訓練例が特徴量より少ないときは常にそうなる。たとえば $n = 20$、$d = 200$ のとき、$\mathbf{X}^\top\mathbf{X}$ は 180 個のゼロ固有値を持つ。演習問題 6 を参照）、$n\lambda\mathbf{I}$ を任意の $\lambda > 0$ に対して加えれば、すべての固有値が $n\lambda > 0$ だけ上方にシフトし、$\mathbf{X}^\top\mathbf{X} + n\lambda\mathbf{I}$ は狭義正定値となって必ず正則（可逆）になる。

2.7.3.2 縮小因子

Equation 2.34 が特徴空間の各方向に沿ってどのように作用するかを調べるため、$\mathbf{S} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$ を特徴量の $d \times d$ 標本二次モーメント（共分散）行列とする。$\mathbf{S}$ は対称かつ半正定値であるため（1.3 節）、正規直交固有値分解 $\mathbf{S} = \sum_{j=1}^{d} \mu_j \mathbf{v}_j \mathbf{v}_j^\top$ を持つ。ここで $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_d \geq 0$ は固有値、$\mathbf{v}_1, \ldots, \mathbf{v}_d$ は正規直交固有ベクトルである。このとき $\mu_j = \frac{1}{n}\|\mathbf{X}\mathbf{v}_j\|_2^2$ は、訓練入力の経験分散（広がり）を方向 $\mathbf{v}_j$ に沿って測った量になっている。

この固有基底では、$\mathbf{S}$ は $\mathbf{v}_j$ に μ_j を掛ける作用素として働き、$(\mathbf{S}+\lambda\mathbf{I})^{-1}$ は $\mathbf{v}_j$ を $\mu_j + \lambda$ で割る作用素として働く。$\mu_j > 0$ である限り、リッジ推定量の成分 $\mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda$ を正則化なしの最小二乗推定量の成分 $\mathbf{v}_j^\top\hat{\mathbf{w}}_0$ と比較すると、次の関係が得られる。

$$\begin{aligned}\mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda &= \frac{\mu_j}{\mu_j + \lambda}\left(\mathbf{v}_j^\top\hat{\mathbf{w}}_0\right),\\ \mathbf{v}_j^\top\hat{\mathbf{w}}_\lambda &= 0 \quad \text{whenever } \mu_j = 0.\end{aligned}$$

各固有方向 $\mathbf{v}_j$ には、それぞれ固有の**縮小因子**（shrinkage factor）が掛けられる（Figure 2.14）。

$$\frac{\mu_j}{\mu_j + \lambda} \in [0, 1). \tag{2.35}$$

ここにリッジ回帰の本質がある。**重み減衰とは一様な縮小ではなく、分散に適応した方向別のフィルタリングなのである。**

- データが豊かに変動する**よく定まる方向**（$\mu_j \gg \lambda$）では、傾きが観測値によってしっかりと特定されており、縮小因子 $\mu_j/(\mu_j+\lambda) \approx 1$ によって最小二乗推定量はほぼそのまま保たれる。
- データがほとんど変動しない**定まりの悪い方向**（$\mu_j \ll \lambda$）、たとえばほぼ同一な 2 本の温度計の差 $x_1 - x_2$ の方向では、正則化なしの最小二乗法はノイズをごく小さな μ_j で割るため爆発してしまう（係数の分散は $\sigma^2/(n\mu_j)$ となる）。これに対し、リッジ回帰は $\mu_j/(\mu_j+\lambda) \approx \mu_j/\lambda \ll 1$ を掛けることでノイズの多い成分を抑え込む。
- **観測されない零空間の方向**（$\mu_j = 0$）では、リッジ回帰は重み成分を恒等的に 0 にする。

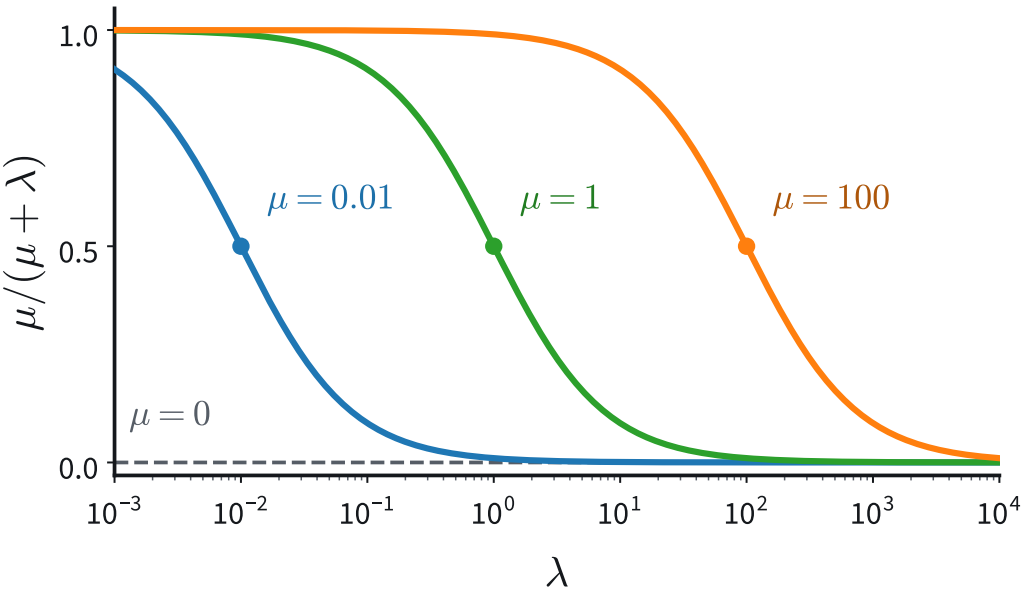


Figure 2.14: リッジの縮小因子 $\mu/(\mu+\lambda)$（固有値 μ を持つ固有ベクトルに沿った値）を正則化定数 λ（対数スケール）に対してプロットしたもの。データによってよく定まる方向は $\lambda \ll \mu$ の間ほぼそのまま保たれ、$\lambda \gg \mu$ になると 0 に向かって強く縮小される。また、この因子が $\frac{1}{2}$ となるのは $\lambda = \mu$ のときである（丸印）。$\mu = 0$ の方向はすべての $\lambda > 0$ に対して 0 に設定される（破線）。

2.7.3.3 データが定める方向

このスペクトル的な描像を $d = 200$ のデータセット `data` で検証しよう。まず、固有値 μ_j（$\mathbf{S} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$ の固有値）を 300 個の訓練入力（中心化なしだが、スペクトルはほとんど変わらない）に対して計算し、あわせて $\lambda = \frac{1}{3}$ における縮小因子も求める。1.3 節と同様に、`eigh` は固有値を昇順で返し、第 2 戻り値の各列として固有ベクトルを返す。さらに、λ を下回る固有値の個数も数えておく。これは、リッジ回帰が最小二乗座標の半分超を取り除く、定まりの悪い方向を表す。

```
X = data.X[:n]                                # the 300 training inputs
mu, V = torch.linalg.eigh(X.T @ X / n)        # increasing eigenvalues
factors = mu / (mu + lambd)                   # shrinkage at lambda = 1/3
k = int((mu < lambd).sum())                   # the poorly determined ones
print(f'mu from {float(mu[0]):.2f} to {float(mu[-1]):.1f}, '
      f'{k} of {len(mu)} below lambda')
print(f'factors from {float(factors[0]):.2f} to {float(factors[-1]):.2f}, '
      f'df = {float(factors.sum()):.0f}')
```

```
mu from 0.04 to 3.2, 54 of 200 below lambda
factors from 0.10 to 0.91, df = 127
```

ゼロになる固有値は 1 つもない。300 個の例があれば、重み空間の 200 次元すべてを張るのに十分だからである。しかし、データが各方向を制約する強さには大きな偏りがある。固有値は数百分の一から約 3.2 までほぼ 2 桁にわたって広がっており、その約 4 分の 1（50〜60 方向）が $\lambda = \frac{1}{3}$ を下回る。対応する縮小因子は約 0.1〜0.9 の範囲に分布する。これらの和は**有効パラメータ数**（effective number of parameters）$\mathrm{df}(\lambda) = \sum_j \mu_j/(\mu_j + \lambda)$ を定義し、ここでは $\mathrm{df}(\frac{1}{3}) \approx 127$ となる。つまり、元の 200 個の重みのうち、正則化されたモデルは実質的に約 127 個の有効パラメータを持つかのように振る舞う。

微小な固有値は最小二乗法にどれほどの悪影響を及ぼすのだろうか。固有ベクトル $\mathbf{v}_j$ に沿って、ラベルのノイズは $\frac{1}{n}(\mathbf{X}\mathbf{v}_j)^\top\epsilon$ を $\frac{1}{n}\mathbf{v}_j^\top\mathbf{X}^\top\mathbf{y}$ に混入させる。これは標準偏差 $\sigma\sqrt{\mu_j/n}$ を持つランダムな摂動である（$\|\mathbf{X}\mathbf{v}_j\|^2 = n\mu_j$ であるため）。最小二乗法はこれを μ_j で割るため、その座標 $\mathbf{v}_j^\top\hat{\mathbf{w}}_0$ は約 $\sigma/\sqrt{n\mu_j}$ だけずれる。このずれは約 0.06（$\mu_j = 1$ のとき）で済むが、実に 0.3（$\mu_j = 0.04$ のとき）にまで達する。真の座標の大きさは $\|\mathbf{w}^*\|/\sqrt{200} = 0.1$ 程度にすぎないため、最小固有値の方向に沿った最小二乗推定量はノイズに完全に埋もれてしまう。一方、検証入力は等方的な標準正規分布に従うため、$\|\mathbf{w} - \mathbf{w}^*\|_2^2 = \sum_j c_j^2$ の直交不変性により、誤差 c_j（$\mathbf{v}_j$ 方向の成分）は期待検証損失を $c_j^2/2$ だけ押し上げることになり、これは μ_j の値によらない。訓練済みの両モデルについて、パラメータ誤差 $\mathbf{w} - \mathbf{w}^*$ を固有ベクトルへ射影し（`V.T @ (w - w_star)`）、定まりの悪い k 個の方向に蓄積した二乗誤差の合計を残りの方向と比較して、これを直接確かめてみよう。

```
w_star = d2l.reshape(data.w, (-1, 1))
for name, model in (('lambda = 0', model_0), ('lambda = 1/3', model_lambd)):
    c = V.T @ (model.w.detach() - w_star)     # error along each eigenvector
    print(f'{name:>12}: squared error {float((c[:k] ** 2).sum()):.2f} in the '
          f'{k} poorly determined directions, '
          f'{float((c[k:] ** 2).sum()):.2f} in the rest')
```

```
  lambda = 0: squared error 1.49 in the 54 poorly determined directions, 0.52 in the rest
```

```
lambda = 1/3: squared error 0.30 in the 54 poorly determined directions, 0.40 in the rest
```

この内訳がすべてを物語っている。正則化なしでは、全体の約 4 分の 1 にすぎない固有ベクトルが、パラメータの総二乗誤差の半分超を占めている。残りのすべての方向を合わせても約 0.5 であるのに対し、この部分だけで通常 0.8〜2.2 に上る。重み減衰を有効にすると、この定まりの悪い方向の誤差が 2 分の 1 から 6 分の 1 に激減するとともに、残りの方向の誤差も緩やかに削られ、どちらのグループも約 0.3〜0.5 に収まる。よりよく定まる方向においてさえ、真の座標に対してラベルのノイズは無視できない大きさを持つため、0.5〜0.9 倍に縮小することで信号よりも多くのノイズが取り除かれるのである。簡単な検算として、パラメータの総二乗誤差の半分に $\sigma^2/2 = 0.5$ を足すと、両方の実行で観測された検証損失とほぼ一致する。要するに重み減衰は、約 50 個の悪条件な方向にわたって同時に、あの 2 本の温度計の差の方向 $x_1 - x_2$ に対して行ったのと同じ選択的な減衰を実行したのである。

このスペクトル的な視点は、先ほど保留にしていた謎も解き明かしてくれる。なぜペナルティなしの実行は、わずか 2〜3 エポックで最小の検証損失に達したのだろうか。ペナルティなしの損失では、各勾配ステップによって $\mathbf{v}_j$ 方向の現在の座標と最小二乗解との差が $1 - \eta\mu_j$ 倍に収縮する。15 ステップからなる 1 エポック（$\eta = 0.05$）を終えた時点で、大固有値の方向（$\mu_j = 3.2$）ではすでに差の $1 - 0.84^{15} \approx 93\%$ が縮まっているのに対し、小固有値の方向（$\mu_j = 0.04$）ではわずか $1 - 0.998^{15} \approx 3\%$ しか縮まっていない。2 エポック後には、勾配降下法はよく定まる方向ではほぼ収束している一方、ノイズの多い方向ではほとんど動いておらず、リッジ推定量をよく模倣した状態にある。この類似性は定量的に確かめられる。0 に近い重みから出発すると、t ステップ後には割合 $1 - (1 - \eta\mu_j)^t$ だけ $\mathbf{v}_j$ 方向の最小二乗座標が構築されるが、これはリッジ因子 $\mu_j/(\mu_j + \lambda)$ において $\lambda \approx 1/(\eta t)$ とおいたものに近い。実際、どちらもほぼ 1（$\mu_j \gg 1/(\eta t)$ のとき）となり、またほぼ $\eta t \mu_j$（$\mu_j \ll 1/(\eta t)$ のとき）となる。30〜45 ステップ（$\eta = 0.05$）という最良の停止点では、$1/(\eta t)$ は 0.45〜0.67 となり、事前分布が与えた $\lambda = \frac{1}{3}$ と同じオーダーになる。しかし訓練を続けると、定まりの悪い方向がラベルのノイズを着実に吸い上げ、検証損失を再び押し上げてしまう。したがって**早期停止**はそれ自体がれっきとした正則化の一形態であり、この実験では重み減衰に匹敵する精度を達成している。ただし、最適な停止点は学習率やミニバッチノイズによって変化し、実行のたびに検証データ上で監視し続けなければならないため、実運用はより難しい（演習問題 5）。

2.7.4 簡潔な実装

重み減衰はニューラルネットワークの訓練で至るところに使われるため、主要な深層学習フレームワークはいずれもオプティマイザ側で更新則 Equation 2.30 を適用できるようになっており、損失関数を変更する必要はない。便利なだけでなく、この統合は計算面でも無駄がない。オプティマイザはもともとステップごとにすべてのパラメータを 1 回ずつ走査するため、減衰因子を適用してもオーバーヘッドはほとんど生じないのである。また、各フレームワークはパラメータごとの更新則もサポートしており、バイアスには手をつけずに重みだけを減衰させることができる。

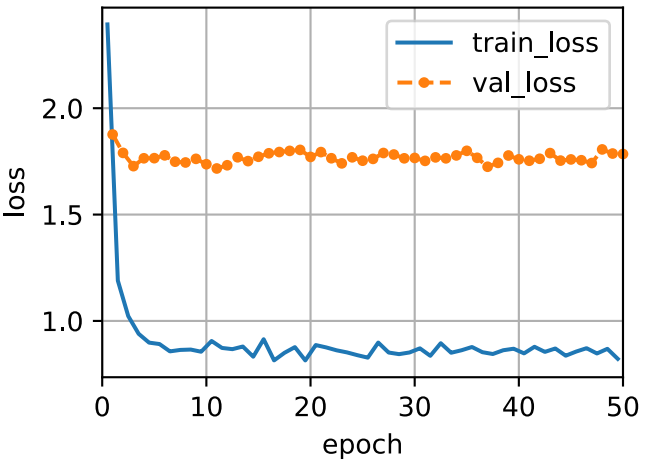
ここでは単に `d2l.LinearRegression` を継承してサブクラス `WeightDecay` を定義し、重み減衰のハイパーパラメータ `wd` を `configure_optimizers` 内のフレームワークの SGD オプティマイザに渡すだけでよい（ただしバイアスパラメータは減衰の対象から外し、重み $\mathbf{w}$ のみにペナルティが課されるようにして Equation 2.29 と一致させる）。

PyTorch のオプティマイザは `weight_decay` 引数を受け取る。パラメータを 2 つのパラメータグループに分けることで、`weight_decay` を `net.weight` のみに適用し、`net.bias` は正則化しないままにできる。2.5 節と同様に、最初の順伝播で遅延初期化（lazy）層のパラメータが実体化された後、重みを $\mathcal{N}(0, 0.01^2)$ から、バイアスを 0 に初期化する処理を `weight_init` の適用によって行う。

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()

    def configure_optimizers(self):
        return torch.optim.SGD([
            {'params': self.net.weight, 'weight_decay': self.wd},
            {'params': self.net.bias}], lr=self.lr)
```

`train_scratch(lambd, data)` と厳密に一致させるには、両方のハイパーパラメータをリスケールする必要がある。組み込みの二乗誤差損失には係数 $\frac{1}{2}$ が付いていないため、学習率は 0.025 へと半分にする（2.5 節と同様である）。しかし、減衰項 $\lambda\mathbf{w}$ は損失の勾配から生じるのではなく、オプティマイザの内部で直接加えられる。したがって、学習率を半分にした分、オプティマイザに渡す重み減衰は $2\lambda = \frac{2}{3}$ へと 2 倍にし、各ステップがまったく同じ収縮因子 $1 - 0.025 \cdot \frac{2}{3} = 1 - \frac{1}{60}$ を保つようにしなければならない。

```
trainer = d2l.Trainer(max_epochs=50)
model = WeightDecay(wd=2 * lambd, lr=0.025)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer.fit(model, data)
norm = float(torch.linalg.vector_norm(model.get_w_b()[0]))
```




```
print(f'validation loss {trainer.history["val_loss"][-1]:.3f}, '
      f'norm of w {norm:.2f}')
```

```
validation loss 1.785, norm of w 1.17
```

得られた重みのノルム（約 1.2）は $\lambda = \frac{1}{3}$ におけるスクラッチ実装の結果を再現しており、検証損失（スクラッチ実装の約 2 倍の値）もフレームワークの平均二乗誤差における係数 2 の違いを考慮すれば一致する。なお、減衰処理が損失関数ではなくオプティマイザの内部にあるため、プロットされる訓練曲線はペナルティ付き損失ではなく通常の平均二乗誤差を追跡している。

日常的な実践において重み減衰を有効にするには、このオプティマイザのハイパーパラメータを 1 つ渡すだけでよい。合成ベンチマーク以外では、ノイズレベル σ や事前分布のスケール τ が事前にわかっていることはまれなため、通常は複数の候補を実行して検証データ上で λ を調整する。なお、ここで述べたノルムに基づく議論がそのまま当てはまるのは、重みが関数そのものの傾きを表す線形モデルであることに留意しておこう。非線形関数に対して複雑さを厳密に測るのはより繊細な問題であり、再生核ヒルベルト空間（reproducing kernel Hilbert space、RKHS）のような枠組みがノルムペナルティを非線形関数へと拡張しているものの、古典的なカーネル法は大規模データセットに対するスケーラビリティに乏しい。それでもなお、ニューラルネットワークの各層の重みに重み減衰を適用することは、現代の深層学習において最も信頼性の高い正則化手法の 1 つであり続けている。

2.7.5 まとめ

訓練の目的関数に二乗 ℓ_2 ペナルティ $\frac{\lambda}{2}\|\mathbf{w}\|_2^2$ を加えること（ただしバイアス b にはペナルティを課さない）により、高次元モデルがほぼ共線な特徴量の結合方向に沿ってノイズを増幅するのを防ぐことができる。ミニバッチ SGD において、このペナルティは各勾配ステップの前に $\mathbf{w}$ に $(1-\eta\lambda)$ を掛けることと等価であり（これが**重み減衰**という名称の由来である）、等方的なガウス事前分布 $w_j \sim \mathcal{N}(0, \tau^2)$ のもとでの最大事後確率推定に $\lambda = \sigma^2/(n\tau^2)$ として対応する。ℓ_1 ペナルティの鋭い角が重みを厳密に 0 へと追い込むのに対し、滑らかな ℓ_2 ペナルティは $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ の固有基底における方向別フィルタとして働く。すなわち、各最小二乗成分に $\mu_j/(\mu_j+\lambda)$ を掛けることで、データによってよく定まる方向（$\mu_j \gg \lambda$）を保持し、ノイズの多い方向（$\mu_j \ll \lambda$）を強く減衰させ、観測されない零空間の方向（$\mu_j = 0$）を 0 にする。

2.7.6 演習問題

1. **ℓ_1 の更新則**。ペナルティ $\frac{\lambda}{2}\|\mathbf{w}\|^2$ を $\lambda\sum_j |w_j|$ に置き換えよ。
    1. $\mathbf{w}$ に対するミニバッチ SGD の更新則を導出せよ。ただし、劣勾配 $\operatorname{sign}(w_j)$（演習問題 2）を $|w_j|$ の導関数の代わりに用いよ。
    2. この更新則の 1 ステップにおいてペナルティ単独で w_j に生じる変化量を、更新則 Equation 2.30 における変化量と、$|w_j|$ が大きい場合と小さい場合のそれぞれについて比較せよ。
    3. (a) の更新則では、w_j が厳密に 0 になることはほとんどない。これを L に関する勾配ステップとそれに続くペナルティステップとに分割し、ペナルティステップが w_j の符号を反転させてしまう場合には常に w_j を 0 に設定するよう修正せよ。その結果を Lasso が行う特徴量選択と関連づけて説明せよ。この修正されたステップは**ソフト閾値処理**であり、ℓ_1 ノルムの近接ステップ（35.6.4.1 節）である。
2. **縮小とペナルティの比較**。重み減衰は、勾配ステップの前に重みを縮小する方法でも、$\frac{\lambda}{2}\|\mathbf{w}\|^2$ を損失 L に加える方法でも適用できる。
    1. 更新則 $\mathbf{w} \leftarrow (1-\eta\lambda)\mathbf{w} - \eta\nabla L$ が、ペナルティ付き損失 $L + \frac{\lambda}{2}\|\mathbf{w}\|^2$ に対する勾配ステップと同一であることを示せ。
    2. ステップの前に、勾配が座標ごとに固定された定数 $a_j > 0$ でリスケールされるとする。先に縮小してからステップを進める方法では下の第 1 式が得られ、ペナルティ付き勾配でステップを進める方法では第 2 式が得られる。

$$w_j \leftarrow (1-\eta\lambda)\, w_j - \eta a_j\, \partial_{w_j} L \qquad \text{and} \qquad w_j \leftarrow w_j - \eta a_j \left(\partial_{w_j} L + \lambda w_j\right). \tag{2.36}$$

どのような a_j に対して両者が一致するか、また第 2 式が w_j を何倍に縮小するか（$\partial_{w_j} L = 0$ となるステップにおいて）を求めよ。深層学習で用いられる多くのオプティマイザは、このように勾配をリスケールする。2 つの更新則のうち、すべての重みに対して λ を同一の減衰率に保つのはどちらか。

Simon Prince『Understanding Deep Learning』問題 9.5 を改変。

3. [code] **拡張最小二乗法としてのリッジ回帰。** 300 個の訓練入力とラベルを中心化してバイアス項を消去し、$\lambda = \frac{1}{3}$ に対するリッジ推定量を次の 2 通りの方法で計算せよ。

   1. 連立一次方程式 Equation 2.34 を、たとえば `np.linalg.solve` を用いて直接解け。
   2. `lstsq` を用いて、$\mathbf{X}$ の下に $\sqrt{n\lambda}\,\mathbf{I}$ を縦に連結した行列と、ラベル $\mathbf{y}$ の後に d 個の 0 を続けたベクトルに対して通常の最小二乗法を実行せよ。

   両者が丸め誤差の範囲で一致することを確かめ、その理由を説明せよ。さらに、得られた解のノルムを `train_scratch(lambd, data)` が返したノルムと比較し、両者の差を生みうる 2 つの手順の違いを挙げよ。

   Hastie、Tibshirani、Friedman『The Elements of Statistical Learning』演習 3.12 を改変。

4. [code] **λ のスイープ。** `WeightDecayScratch` を `data` 上で $\lambda \in \{0.01, 0.1, 0.3, 1, 3, 10, 30\}$ の各値について訓練せよ。

   1. 最終的な検証損失と $\mathbf{w}$ のノルムを対数軸の λ に対してプロットせよ。その際、`train_scratch(0, data)` の値を水平線として描き添え、検証損失が最小となる λ を報告せよ。
   2. 同じ各値に対する $\mathrm{df}(\lambda)$ を 2.7.3 節の固有値から計算し、検証損失を $\mathrm{df}(\lambda)$ に対してプロットせよ。その曲線の形状を Figure 2.11 と比較せよ。
   3. 事前分布が示唆する値 $\lambda = \frac{1}{3}$ は、スイープにおける最良値に近いか。ノイズ $\sigma = 0.5$ のデータと $\sigma = 2$ のデータをそれぞれ生成し、各場合について事前分布による λ を再計算して、スイープの最良値に追随するかどうかを確かめよ。

5. [code] **早期停止と重み減衰の比較。** 本節のペナルティなしの実行は、2〜3 エポック後に検証損失の最小値に達し、その値は $\lambda = \frac{1}{3}$ の実行が最終的に到達する値とほぼ同じくらい低い。

   1. 0 に近い重みから出発すると、ペナルティなし損失に対する勾配降下法は、$\mathbf{v}_j$ 方向の座標とその最小二乗値との差を 1 ステップごとに $1-\eta\mu_j$ 倍するため、t ステップ後にはその座標の割合 $1-(1-\eta\mu_j)^t$ を構築する。この因子を μ_j（2.7.3 節の固有値）に対してプロットせよ。ただし、t は 1 エポック・2 エポック・50 エポック（1 エポックあたり 15 ステップ、$\eta = 0.05$）とし、リッジ因子 $\mu_j/(\mu_j+\lambda)$（$\lambda = \frac{1}{3}$）も並べて描け。どの t がリッジに最も近いか。
   2. `WeightDecayScratch` を $\lambda = 0$ で 2 エポック訓練し、2.7.3 節と同様に固有ベクトルに沿って誤差を分解せよ。定まりの悪い k 個の方向と残りの方向における二乗誤差を、同項で出力された 50 エポック訓練時（$\lambda = 0$ および $\lambda = \frac{1}{3}$）の値と比較せよ。
   3. ペナルティなしの実行を $\eta = 0.01$ と $\eta = 0.1$ で繰り返し、`trainer.history` から最良のエポックを読み取れ。最良エポックがどのように移動するかを説明し、ペナルティには不要だが早期停止では実用上必要となるものを述べよ。

6. [code] **訓練例より多い重み。**●200 個の特徴量とわずか 20 個の訓練例を持ち、真の重みが 0.01、バイアスが 0.05、ノイズが $\sigma = 0.01$ であるデータをミニバッチサイズ 5 で生成せよ。
   1. $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ の固有値を 20 個の訓練入力に対して計算せよ。0 になる固有値は何個あるか。また、それらの固有ベクトルに沿った重みを勾配降下法が決して変化させないのはなぜか。
   2. $\lambda = 0$ と $\lambda = 3$ のそれぞれについて、30 エポック、学習率 0.01 で訓練せよ。その際、初期値を `WeightDecayScratch` のランダム初期化とした場合と、$\mathbf{w} = \mathbf{0}$（`sigma=0` を渡す）とした場合の 2 通りを試せ。実行する前に、定まらない方向における初期重みのノルムから 4 つの検証損失を予測し、その後に実行結果と比較せよ。
   3. 4 つの損失すべてを、常に b^* と予測するモデルの損失と比較せよ。ここで重み減衰は何を達成し、またどのような λ の値をもってしても達成できないことは何か。

Chapter 3

線形分類

第 2 章では、住宅の販売価格のような連続量を予測することが目標であった。しかし、多くのタスクでは代わりに離散的なカテゴリが求められる。たとえばオンラインカタログの写真に写っているのはシャツ、コート、サンダルのいずれかであり、何かの「5.5」では決してない。カテゴリに 1 から 10 までの番号を単純に振って線形回帰を適用しても、まったくうまくいかない。プルオーバーがズボンとドレスのちょうど中間にあるかのような恣意的な順序を押しつけてしまううえ、較正（calibration）された不確実性（「おそらくシャツだが、プルオーバーかもしれない」）をモデルが表現できないからである。

幸い、ミニバッチデータローダ、ベクトル化された順伝播、自動微分、ミニバッチ SGD といった回帰の訓練機構は、そのほぼすべてがそのまま**分類**（classification）に引き継がれる。変わるのは 3 つの要素だけである。順序のないカテゴリを**ワンホットベクトル**で表し、カテゴリごとに 1 つの線形スコア（**ロジット**）を計算して**ソフトマックス**関数により確率分布へと正規化し、二乗誤差を**交差エントロピー損失**（cross-entropy loss）に置き換える。この損失は、二乗誤差がガウスノイズから導かれたのと同様に、カテゴリラベルのモデルにおける最尤法から導かれる。本書で扱うニューラル分類器は、大規模言語モデルに至るまで、そのほぼすべてが最終層にこのソフトマックス層を持ち、この損失のもとで訓練される。

本章ではまず、ソフトマックス回帰（softmax regression）と交差エントロピー損失を定式化し、その残差勾配 $\hat{\mathbf{y}} - \mathbf{y}$ を導出して、Shannon の情報理論と結びつける（3.1 節）。次に、再利用可能な `FashionMNIST` 画像パイプラインを構築し（3.2 節）、ソフトマックス回帰をゼロからの方法と簡潔な方法の両方で実装する（3.3 節）。そこでは、混同行列（confusion matrix）と学習された 28×28 のテンプレートを用いて誤りを診断するとともに、融合された log-sum-exp 損失によって浮動小数点のオーバーフローとアンダーフローを解消する。

分類が回帰と大きく異なるのは、本章の後半を占める**評価**と**デプロイ**である。陽性例がまれな場合や誤りによってコストが異なる場合には、単純な正解率（accuracy）は誤解を招く。これが適合率（precision）、再現率（recall）、コスト考慮型の閾値、および較正の診断を導入する動機となる（3.4 節）。クラスが均衡していても、有限のテストセットには統計的なエラーバーが伴い、繰り返し再利用するとその保証が損なわれる。そこで Hoeffding 限界、適応的過学習（adaptive overfitting）、および Vapnik–Chervonenkis（VC）理論へと進む（第 3.5 章）。最後に 3.6 節では、デプロイ時のデータが訓練分布から乖離したときに何が起こるかを問い、重点重み付けによる共変量シフト（covariate shift）の補正、混同行列を用いたラベルシフトの推定、および暴走するフィードバックループへの防御策を展開する。

3.1 ソフトマックス回帰

回帰は、**どれくらいの量か**や**いくつあるか**といった問いに答えるための手法である。しかし、教師あり学習の多くの問題では、代わりに**どのカテゴリか**が問われる。たとえば、スパムフィルタはメールを 2 つのクラスに振り分け、臨床支援ツールは複数の診断候補から選択し、言語モデルは数万語の語彙から次のトークンを予測する。

あるオンライン小売業者が、商品の写真を T シャツ、ズボン、プルオーバー、ドレス、コート、サンダル、シャツ、スニーカー、バッグ、アンクルブーツという 10 個のアパレルカテゴリに分類したいとする。これらのカテゴリに 1 から 10 までの番号を振り、線形回帰をそのまま適用してはいけないのはなぜか。順序のないカテゴリに対する整数コードは単なる便宜上の名前にすぎず、数値的な量ではないからである。たとえば、5.5 という予測は写真をコートとサンダルのちょうど中間に位置づけてしまう。また二乗誤差では、T シャツ（1）をアンクルブーツ（10）と誤ると、ズボン（2）と誤る場合より 81 倍も重く罰せられる。さらに、最終的に**ハード割り当て**（hard assignment、単一のカテゴリを選ぶこと）が必要な場合であっても、後続の意思決定で確信度と誤りのコストを比較衡量できるよう、**ソフト割り当て**（soft assignment、すべての候補クラスに対する較正された確率）を出力するモデルがほぼ常に求められる。

ソフトマックス回帰（softmax regression）は**多項ロジスティック回帰**（multinomial logistic regression）とも呼ばれ、この両方の問題を明快に解決する。すなわち、クラスごとに 1 つのアフィンスコアを計算し、ソフトマックス演算子によってそれらのスコアを有効な確率分布へと写像する。2.1 節の最尤法（maximum likelihood）の手順に従い、観測されたカテゴリの負の対数尤度（negative log-likelihood）を最小化すると、**交差エントロピー損失**（cross-entropy loss）が得られる。

3.1.1 モデル

3.1.1.1 クラスラベル

まずは具体例として、2×2 のグレースケール画像分類（image classification）問題を考えよう。この問題は 4 つのスカラーのピクセル特徴量 x_1, x_2, x_3, x_4 を持ち、猫（1）、ニワトリ（2）、犬（3）の 3 つのカテゴリを扱う。一般に、クラス数が q のとき、各データ例のラベルは整数 $y \in \{1, \dots, q\}$ である。

順序のないカテゴリに誤った数値的順序を持ち込まずに表現するため、統計学では**ワンホットエンコーディング**を用いる。これはベクトル $\mathbf{y} \in \{0, 1\}^q$ であり、正解クラスの成分を 1（$y_y = 1$）、それ以外のすべての成分を 0 とする表現である。先ほどの 3 クラスの例では、猫は $(1, 0, 0)$、ニワトリは $(0, 1, 0)$、犬は $(0, 0, 1)$ となる。これにより、異なるどの 2 クラス間も等しいユークリッド距離 $\sqrt{2}$ に配置される。また $\mathbf{y}$ 自体が有効な確率分布（すべての確率質量を正解クラスに置く退化分布）になっており、モデルが予測する分布 $\hat{\mathbf{y}}$ と直接比較できる。（数式では 1 始まりのクラスインデックス y とワンホットベクトル $\mathbf{y}$ を用いるが、Python コードでは 0 始まりの整数インデックスをそのまま格納する。）

3.1.1.2 線形スコア

すべての q 個のクラスに同時にスコアを付けるため、ソフトマックス回帰は各クラス $j \in \{1, \dots, q\}$ に対して、入力特徴量の独立したアフィン関数を割り当てる。重み w_{kj} を特徴量 k からクラス j への結合重み、b_j をクラス j のバイアスと表すと、この 2×2 の画像分類器は正規化されていない 3 つのスコア（**ロジット**）を次の式で計算する。

$$
\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,\\
o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,\\
o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.
\end{aligned}
$$

ニューラルネットワークとして見ると（Figure 3.1）、ソフトマックス回帰は d 個の入力と q 個の出力ユニットを持つ単層の全結合（fully connected）ネットワークである（重みは $d \cdot q$ 個、バイアスは q 個）。

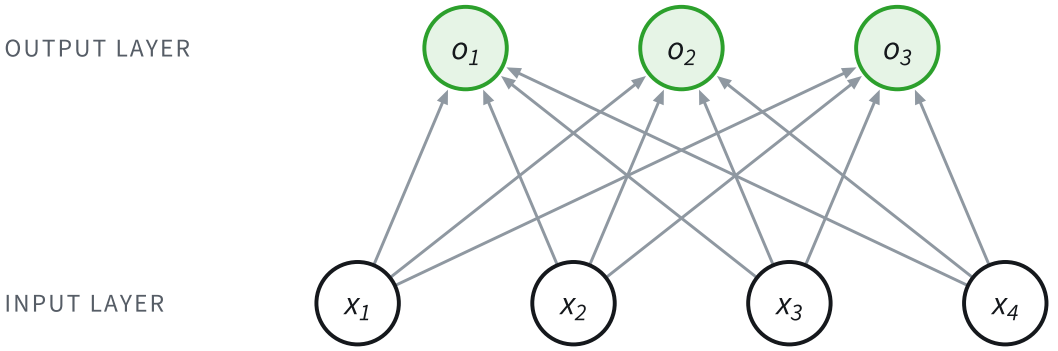


Figure 3.1: 単層の全結合ニューラルネットワークとして表したソフトマックス回帰。各入力 x_k は各出力ロジット o_j に重み w_{kj} を介して結合する。

重みを行列 $\mathbf{W} \in \mathbb{R}^{d \times q}$（特徴量ごとに 1 行、クラスごとに 1 列 $\mathbf{w}_j \in \mathbb{R}^d$）にまとめ、バイアスを $\mathbf{b} \in \mathbb{R}^q$ にまとめると、入力 $\mathbf{x} \in \mathbb{R}^d$ に対するロジットベクトルは次のように簡潔に書ける。

$$
\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}, \tag{3.1}
$$

ここで、$o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$ である。28×28 の Fashion-MNIST 衣類画像を 3.2 節で分類する際には、$d = 784$、$q = 10$ となり、7,840 個の重みと 10 個のバイアスが得られる。

3.1.1.3 ソフトマックス

線形層が、猫、ニワトリ、犬に対してロジット $\mathbf{o} = (1.0, 2.2, 0.3)$ を出力したとする。ニワトリのスコアが最も高いが、生のロジットは確率ではない。その和は 3.5 であって 1 ではなく、アフィンスコアが負になったり 1 を超えたりすることを防ぐ仕組みもない。任意の実数ベクトル $\mathbf{o} \in \mathbb{R}^q$ を、スコアのランキングを保ったまま有効な確率分布 $\hat{\mathbf{y}}$（q 個の正の数からなり、その和が 1 になるもの）に変換するため、**ソフトマックス**関数は各ロジットの指数をとり、その和で割る。

$$
\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)}. \tag{3.2}
$$

$\mathbf{o} = (1.0, 2.2, 0.3)$ の場合、各成分の指数は $(2.72, 9.03, 1.35)$、その和は 13.09 となり、$\hat{\mathbf{y}} = (0.21, 0.69, 0.10)$ が得られる。すなわち、ニワトリの予測確率は 69% である。exp は狭義単調増加関数であるため、ソフトマックスはロジットの大小関係を保つ。したがって、最も確率の高いクラスは常に最大のロジットを持つクラスと一致する。

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{3.3}$$

推論時に最も予測確率の高いカテゴリだけが必要な場合は、指数の計算をすべて省き、単に $\operatorname{argmax}_j o_j$ を求めればよい。

なぜ他の正の単調増加関数ではなく、指数関数を使うのだろうか。それは、指数関数が加法を乗法に変換することで、ソフトマックスに次の 3 つの基本的な性質をもたらすからである。

1. **シフト不変性（ロジット間の差のみが重要）：**任意のスカラー定数 c を $\mathbf{o}$ のすべての成分に加えても、Equation 3.2 の分子と分母の両方に $\exp(c)$ が掛かるため、確率は変化しない。すなわち、$\operatorname{softmax}(\mathbf{o} + c\mathbf{1}) = \operatorname{softmax}(\mathbf{o})$ が成り立つ。言い換えれば、任意の 2 つのロジットの差は予測確率の対数オッズ比（log-odds ratio）に等しい。

$$o_i - o_j = \log \frac{\hat{y}_i}{\hat{y}_j}. \tag{3.4}$$

   先ほどの例では $o_2 - o_1 = 1.2$ であるため、ニワトリは猫の $e^{1.2} \approx 3.3$ 倍の確率となる（$0.69/0.21$）。このシフト不変性（shift invariance）により、重要な数値計算上の安全策も得られる（3.3 節）。指数をとる前に $o_{\max} = \max_k o_k$ を引いておけば、計算結果を変えずに浮動小数点オーバーフローを防げるのである。
2. **$q = 2$ におけるロジスティック回帰への帰着：**2 クラスの場合、確率は単一の差 $o = o_1 - o_2$ のみに依存する。

$$\hat{y}_1 = \frac{\exp(o_1)}{\exp(o_1) + \exp(o_2)} = \frac{1}{1 + \exp(-o)} = \sigma(o), \tag{3.5}$$

   ここで、$\sigma(o)$ は**ロジスティックシグモイド**である。したがって、2 クラスのソフトマックス回帰は古典的な**ロジスティック回帰**（logistic regression）と同一であり、$o = \log(\hat{y}_1/\hat{y}_2)$ は対数オッズ（log-odds）を表す。これが**ロジット**という語の歴史的な由来である。
3. **最尤法における簡潔な勾配：**後述するように、ソフトマックスの指数関数と尤度の対数関数が対になることで、損失の勾配は残差（residual）$\hat{\mathbf{y}} - \mathbf{y}$ という簡潔な形に帰着する。

3.1.1.4 ミニバッチ

n 個のデータ例（各例は d 個の特徴量を持つ）からなるミニバッチに対してベクトル化するため、入力を $\mathbf{X} \in \mathbb{R}^{n \times d}$、重みを $\mathbf{W} \in \mathbb{R}^{d \times q}$、バイアスを $\mathbf{b} \in \mathbb{R}^{1 \times q}$ にまとめる。

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \operatorname{softmax}(\mathbf{O}), \end{aligned} \tag{3.6}$$

ここで、$\mathbf{b}$ は n 個の行（$\mathbf{O} \in \mathbb{R}^{n \times q}$ の各行）にわたってブロードキャストされ、softmax は**行ごと**に適用される（$\mathbf{O}$ の要素ごとに指数をとり、各行をその行の和で割る）。

3.1.1.5 決定領域

ソフトマックスはアフィンロジット $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$ の大小関係を保つため、ソフトマックス回帰は入力空間 $\mathbb{R}^d$ を平坦な面で囲まれた凸多面体へと分割する（Figure 3.2）。クラス j が選ばれるのは、$o_j \geq o_k$ がすべての $k \neq j$ に対して成り立つ領域であり、これは $q - 1$ 個の半空間 $(\mathbf{w}_j - \mathbf{w}_k)^\top \mathbf{x} + (b_j - b_k) \geq 0$ の共通部分にほかならない。隣接するクラス同士は、2 つのロジ

ットが等しくなる超平面に沿って接する。線形回帰と同様に、クラス間の境界は常に**平坦な**超平面であり、曲がった境界や非凸な決定領域（decision region）を切り出すには隠れ層が必要となる（第 4 章）。

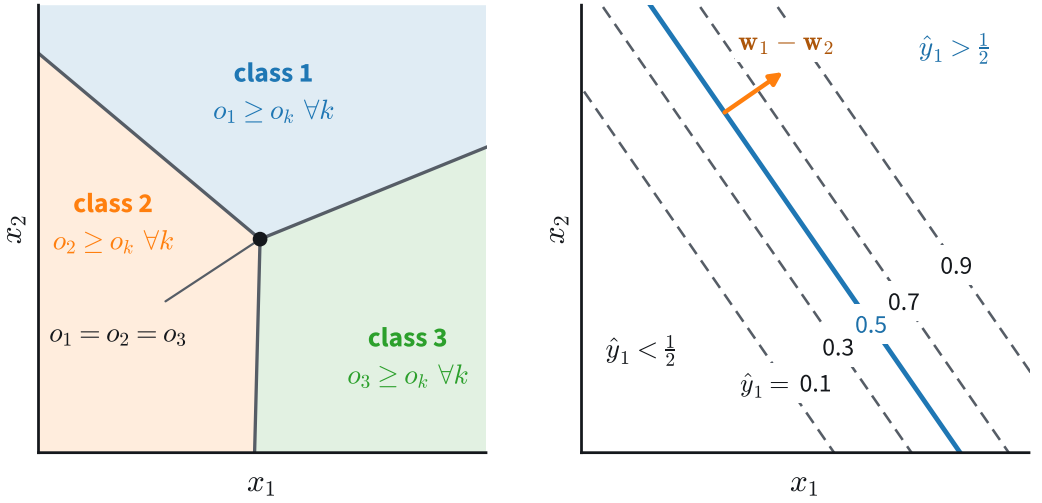


Figure 3.2: $\mathbb{R}^2$ におけるソフトマックス回帰の決定領域。左：3 クラスの場合、平面は 3 つの凸領域に分割され、ロジットの対が等しくなる直線に沿って接する。右：2 クラスの場合、等確率の等高線 $\hat{y}_1 = \sigma(o)$ は $\mathbf{w}_1 - \mathbf{w}_2$ に直交する平行線となり、決定境界は $\hat{y}_1 = 1/2$ にある。

ここで前提としている構造上の基本仮定に留意しておこう。それは、各データ例が**順序のない**集合から選ばれた**ただ 1 つの**クラスに属するという点である。同一の入力に複数のタグが同時に当てはまりうる場合（**マルチラベル分類**（multi-label classification））には、競合するソフトマックスではなくタグごとに独立した二値シグモイドを用いる (Tsoumakas and Katakis 2007)。また、クラスが自然な順序に従う場合（**順序回帰**（ordinal regression））には、その順序構造を活用する専用の閾値モデルを用いる (Moon et al. 2010)。

3.1.2 交差エントロピー損失

3.1.2.1 最尤法

予測された確率分布 $\hat{\mathbf{y}}$ と観測されたラベル y の食い違いは、どのように測ればよいだろうか。2.1 節の最尤法の原理に従い、$\hat{y}_j = P(y = j \mid \mathbf{x})$ をクラス j の条件付き確率（conditional probability）と解釈する。ワンホットベクトル $\mathbf{y} \in \{0, 1\}^q$ を用いると、観測されたクラス y に割り当てられる確率は次のように書ける。

$$P(y \mid \mathbf{x}) = \hat{y}_y = \prod_{j=1}^{q} \hat{y}_j^{y_j}.$$

n 個の独立な訓練例にわたる負の対数尤度は、データ例ごとの損失の和となる。

$$
\begin{aligned}
-\log \prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)}) &= \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \\
\text{where} \quad l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \hat{y}_j = -\log \hat{y}_y.
\end{aligned}
\tag{3.7}
$$

この目的関数が**交差エントロピー損失**である。その振る舞いを確認しておこう。

- $\hat{y}_y \in (0, 1)$ であるため、常に $l(\mathbf{y}, \hat{\mathbf{y}}) > 0$ である。また、$\hat{y}_y \to 1$ のとき 0 に近づき、$\hat{y}_y \to 0$ のとき $+\infty$ に発散する（確信を持った誤りに重いペナルティを科す）。
- ロジットが $(1.0, 2.2, 0.3)$、確率が $(0.21, 0.69, 0.10)$ であった先ほどの画像において、正解ラベルがニワトリ（$y = 2$）なら損失は $-\log 0.69 \approx 0.37$ nat であるが、もし実際は犬（$y = 3$）であれば、損失は 6 倍の $-\log 0.10 \approx 2.30$ nat へと跳ね上がる。

3.1.2.2 勾配

交差エントロピー損失をロジット $\mathbf{o}$ で微分してみよう。Equation 3.2 を Equation 3.7 に代入し、$\sum_{j=1}^{q} y_j = 1$ を用いると、損失は $\mathbf{o}$ によって直接次のように表される。

$$
\begin{aligned}
l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\
&= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\
&= \log \sum_{k=1}^{q} \exp(o_k) - o_y.
\end{aligned}
$$

第 1 項の $\log \sum_{k=1}^{q} \exp(o_k)$ は **log-sum-exp** 関数であり、$\max_k o_k$ の滑らかな凸上界となっていて、$\max_k o_k$ を超える量は高々 $\log q$ に抑えられる（演習問題 4）。これを o_j で微分すると、次式が得られる。

$$
\begin{aligned}
\frac{\partial l}{\partial o_j} &= \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \operatorname{softmax}(\mathbf{o})_j - y_j, \\
\text{or} \quad \nabla_{\mathbf{o}} l &= \hat{\mathbf{y}} - \mathbf{y}.
\end{aligned}
$$

二乗損失のもとでの線形回帰とまったく同様に、**自然パラメータ $\mathbf{o}$ に関する勾配は、単に予測から観測を引いた差 $\hat{\mathbf{y}} - \mathbf{y}$ となる**。これは指数分布族（exponential family）の一般化線形モデル（generalized linear model）に共通する普遍的な性質である（36.4.3.3 節）。先ほどのニワトリの画像（$\mathbf{y} = (0, 1, 0)$）では $\nabla_{\mathbf{o}} l = (0.21, -0.31, 0.10)$ となり、勾配降下法を 1 ステップ進めると、ニワトリのロジットが押し上げられる一方、猫と犬のロジットはそれぞれの競合クラスが誤って奪った確率質量に厳密に比例して押し下げられる。さらに $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ を介して連鎖律（chain rule）を適用すると、$\mathbf{W}$ に関するミニバッチ勾配は $\frac{1}{n}\mathbf{X}^\top(\hat{\mathbf{Y}} - \mathbf{Y})$ となり、損失は $(\mathbf{W}, \mathbf{b})$ に関して大域的に凹（convex）となる。実際、$\mathbf{o}$ に関するヘッセ行列（Hessian）は $\hat{\mathbf{y}}$ からのカテゴリカルな抽出の共分散行列（covariance matrix）に一致する（演習問題 2）。

	線形回帰	ソフトマックス回帰
目的変数	連続値 $y \in \mathbb{R}$	クラス $y \in \{1, \dots, q\}$、ワンホット $\mathbf{y}$
予測	$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$	$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$
確率モデル	ガウス分布 $\mathcal{N}(\hat{y}, \sigma^2)$	カテゴリカル分布 $\mathrm{Cat}(\hat{\mathbf{y}})$
損失（NLL）	$\frac{1}{2}(\hat{y} - y)^2$	$-\sum_{j=1}^{q} y_j \log \hat{y}_j = -\log \hat{y}_y$
勾配	$\hat{y} - y$（$\hat{y}$ に関して）	$\hat{\mathbf{y}} - \mathbf{y}$（$\mathbf{o}$ に関して）

3.1.2.3 エントロピーと交差エントロピー

交差エントロピー（cross-entropy）という名称は、Claude Shannon の情報理論（information theory）に由来する（37.1 節）。確率 p の事象が生起したとき、Shannon はその**サプライザル**（情報量、information content）を $-\log p$ と定義した（自然対数を用いる場合は **nat**、$\log_2$ を用いる場合は **bit** を単位とし、$1\ \text{nat} = 1/\log 2 \approx 1.44\ \text{bits}$ である）。真の分布 $\mathbf{p} = (p_1, \dots, p_q)$（$q$ 個のクラス上の分布）に対して、サプライザルの期待値がその**エントロピー**である。

$$H(\mathbf{p}) = -\sum_{j=1}^{q} p_j \log p_j, \tag{3.8}$$

この値は 0（1 つの事象が確実に起こる場合）から $\log q$（q 個すべての事象が等確率で起こる場合）までの範囲をとる。ここで、$\mathbf{p}$ から生じる事象を、モデル分布 $\hat{\mathbf{p}}$ に最適化された符号を使って符号化すると、その平均符号長（サプライザルの期待値）が**交差エントロピー**となる。

$$H(\mathbf{p}, \hat{\mathbf{p}}) = -\sum_{j=1}^{q} p_j \log \hat{p}_j. \tag{3.9}$$

ギブスの不等式（Gibbs' inequality）により $H(\mathbf{p}, \hat{\mathbf{p}}) \geq H(\mathbf{p})$ が成り立ち、等号が成立するのは $\hat{\mathbf{p}} = \mathbf{p}$ のとき、かつそのときに限る（両者の差はカルバック・ライブラー・ダイバージェンス $D_{\mathrm{KL}}(\mathbf{p} \,\|\, \hat{\mathbf{p}}) = H(\mathbf{p}, \hat{\mathbf{p}}) - H(\mathbf{p}) \geq 0$ である）。$H(\mathbf{p}, \hat{\mathbf{p}})$ を Equation 3.7 と比較すると、データ例ごとの損失 $l(\mathbf{y}, \hat{\mathbf{y}}) = H(\mathbf{y}, \hat{\mathbf{y}})$ は、文字通り経験ラベル分布 $\mathbf{y}$ と予測分布 $\hat{\mathbf{y}}$ の交差エントロピーにほかならない。さらに都合のよいことに、入力が本質的に曖昧な場合（たとえば、シャツである確率が 60%、T シャツである確率が 40% のぼやけた写真）でも、モデルが過剰に確信した推測ではなく真の較正された確率 $(0.6, 0.4)$ を出力するときに交差エントロピー損失の期待値が一意に最小化されることが、ギブスの不等式によって保証される。

3.1.3 正解率と確信度

3.1.3.1 誤り率

分類器の評価には**正解率**（accuracy）や**誤り率**（error rate、$= 1 - \text{accuracy}$）を用いるが、0–1 誤差を直接微分してモデルを訓練することはできない。データ例の**マージン**を、正解クラスのロジットから競合クラスの中で最大のロジットを引いた値として定義しよう。

$$m = o_y - \max_{j \neq y} o_j. \tag{3.10}$$

分類器が誤るか（$m < 0$）正しい予測をするか（$m > 0$）は m の不連続な階段関数で決まるが（Figure 3.3）、その導関数は存在するすべての点で 0 となるため、勾配降下法には学習のための信号がまったく届かない。交差エントロピーはこの平坦な階段関数を、滑らかで狭義単調減少する凸な代理損失（$\log(1 + e^{-m})$、ただし $q = 2$ の場合）に置き換える。この代理損失は（$\log 2$ で割って bit 単位で測れば）0–1 誤差の上界となり、すべてのデータ例に対して非ゼロの勾配を与える。

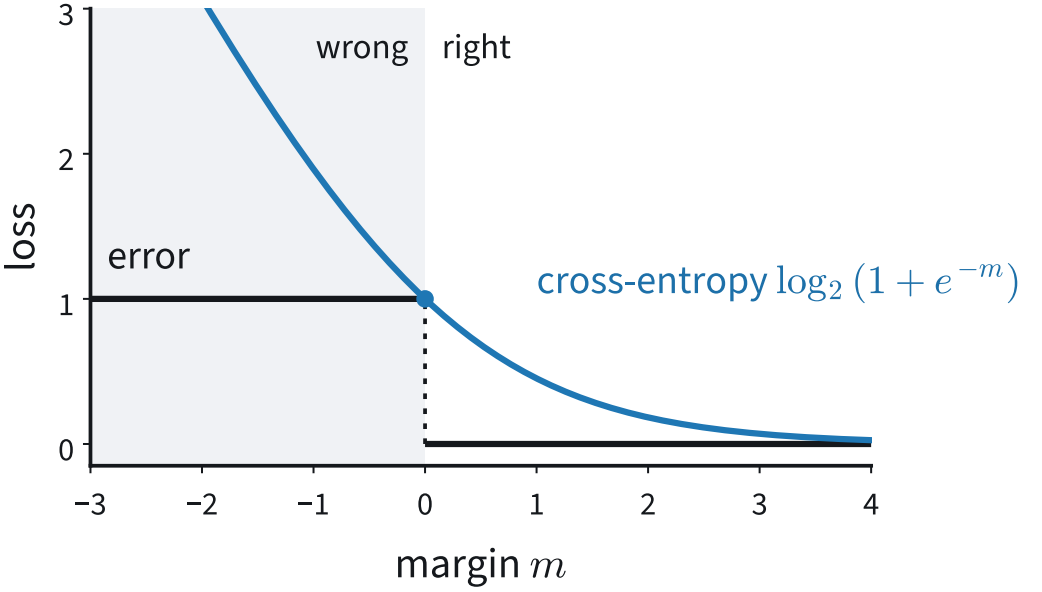


Figure 3.3: 0–1 分類誤差と 2 クラスの交差エントロピー損失（bit 単位、$\log_2(1 + e^{-m})$）をマージン $m = o_y - \max_{j \neq y} o_j$ の関数として描いた図。

3.1.3.2 較正と温度

Figure 3.3 の滑らかな曲線がもたらす 1 つの帰結に注目しよう。交差エントロピーは、$m > 0$ となって分類の判定がすでに正解した後も減少し続ける。もし訓練データセットが**線形分離可能**（linearly separable）であれば、すなわちある $(\mathbf{W}, \mathbf{b})$ がすべての訓練例を正しく分類できる（$m^{(i)} > 0$ がすべての i で成り立つ）ならば、$(\mathbf{W}, \mathbf{b})$ に任意のスケール因子 $c > 1$ を掛けるとすべてのマージンが c 倍になり、訓練損失はどこまでも 0 に近づいていく（演習問題 6）。その結果、決定境界（decision boundary）はまったく動かないまま、すべての予測確率が 0 と 1 に張り付いていく。

より一般に、ソフトマックスを適用する前にすべてのロジットを正の**温度**（temperature）$T > 0$ の逆数でスケーリングすると、次のようになる。

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}/T), \tag{3.11}$$

この操作は、分布をワンホットな argmax へと先鋭化させ（$T \to 0$）、逆に一様分布 $(1/q, \ldots, 1/q)$ へと平坦化させるが（$T \to \infty$、Figure 3.4）、ロジットのランキング（したがって分類器の正解率）は変えない。訓練されたモデルの予測確率が実際の経験的な正解率と一致しているか（**較正**、calibration）は実証的な問いであり、3.4 節でホールドアウトデータを用いて検証する。

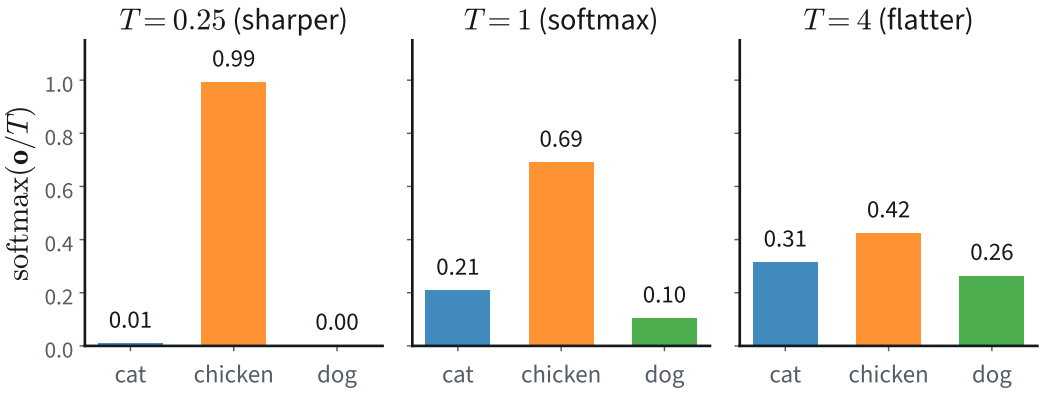


Figure 3.4: ロジット $\mathbf{o} = (1.0, 2.2, 0.3)$ に対する 3 つの温度でのソフトマックス確率。$T = 0.25$（鋭いピーク）、$T = 1$（標準のソフトマックス）、$T = 4$（ほぼ一様）。

3.1.4 まとめ

ソフトマックス回帰は、クラスごとに 1 つのアフィンロジット $\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$ を計算し、それを確率 $\hat{y}_j = \exp(o_j) / \sum_k \exp(o_k)$ へと正規化することで順序のない離散カテゴリを予測し、入力空間を超平面で囲まれた凸多面体の決定領域へと分割する。ソフトマックス演算子はロジットの対ごとの差 $o_i - o_j = \log(\hat{y}_i / \hat{y}_j)$ のみに依存するため、$\mathbf{o}$ の一様なシフトに対して不変であり、$q = 2$ のときには二値のロジスティック回帰に帰着する。ワンホットの目的変数 $\mathbf{y}$ に対するカテゴリカル尤度を最大化すると、凸な交差エントロピー損失 $l(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{y}_y = \log \sum_k \exp(o_k) - o_y$ が得られる。これは 0–1 分類誤差の滑らかな上界であり、ロジットに関する勾配は残差 $\hat{\mathbf{y}} - \mathbf{y}$ へと簡約される。ロジットを温度 $T > 0$ で割れば、クラスのランキングや正解率を変えることなく、決定論的な argmax から一様分布まで予測の確信度を調整できる。

3.1.5 演習問題

1. **ソフトマックスとシグモイド。** Equation 3.5 により、ロジスティック回帰は冗長なロジットを 1 つ取り除いた 2 クラスのソフトマックス回帰である。同僚から、当てはめ済みのロジスティックモデル $P(y = 1 \mid x) = \sigma(\hat{\beta}_0 + \hat{\beta}_1 x)$（ただし $\hat{\beta}_0 = 2$、$\hat{\beta}_1 = -1$）を渡されたとする。
    1. 2 クラスのソフトマックス係数 (w_1, b_1) と (w_2, b_2) のうち、すべての x に対してこのロジスティックモデルと予測が一致するものをすべて特徴づけ、さらに $o_2 \equiv 0$ を満たす組を特定せよ。
    2. 2 クラスのソフトマックスモデルの交差エントロピー損失に対する勾配降下法が、$w_1 + w_2$ および $b_1 + b_2$ を決して変化させないことを示せ。同僚の訓練データ上で勾配降下法を $w_1 = w_2 = b_1 = b_2 = 0$ から開始し、このロジスティックモデルを再現する係数に収束したとする。どの組に到達するかを求めよ。

    James、Witten、Hastie、Tibshirani 著『An Introduction to Statistical Learning』の演習問題 4.12 を改変。

2. **ソフトマックスのヘッセ行列。** 3.1.2.2 節と同様に、交差エントロピー損失をロジットの関数として $l(\mathbf{y}, \mathbf{o}) = g(\mathbf{o}) - \mathbf{y}^\top \mathbf{o}$（ただし $g(\mathbf{o}) = \log \sum_k \exp(o_k)$）と表す。
    1. ヘッセ行列 $\partial^2_{\mathbf{o}}\, l(\mathbf{y}, \mathbf{o})$ を計算し、それが $\mathbf{y}$ に依存しないことを示せ。

2. $\mathbf{e}_Y$ を、クラス Y（$\mathrm{softmax}(\mathbf{o})$ から抽出される）のワンホットベクトルとする。$\mathbf{e}_Y$ の共分散行列を計算し、前の小問で求めたヘッセ行列と一致することを示せ。
3. g が凸関数であり、交差エントロピー損失がロジットに関して凸性を持つことを結論づけよ。

3. **Bradley–Terry モデル**。Bradley–Terry モデル (Bradley and Terry 1952) は、ロジスティックモデルを用いて選好を表現する。リンゴとオレンジのどちらかを選ぶユーザーは、2 つのスコア o_{apple} と o_{orange} のソフトマックスで与えられる確率に従って各選択肢を選ぶ。ここに、どちらも選ばないというデフォルトの選択肢を固定スコア $o_{\text{none}} = 0$ で追加する。
   1. o_{apple} と o_{orange} に同じ定数を加えたとき、選択確率が変化するかどうかを、まずデフォルト選択肢がない場合、次にデフォルト選択肢がある場合について調べよ。それぞれのモデルにおいて、観測された選択からどのスコアが一意に定まるかを結論づけよ。
   2. デフォルト選択肢がある場合とない場合で、比 $P(\text{apple})/P(\text{orange})$ を比較せよ。実際のユーザーがこの比を一定に保たないような選択状況を説明せよ。

4. **log-sum-exp**。関数 $\mathrm{lse}(a, b) = \log(\exp(a) + \exp(b))$ は 3.1.2.2 節の log-sum-exp を 2 引数にしたものであるから、その勾配は (a, b) のソフトマックスとなる。
   1. 次の不等式が成り立ち、
   $$\max(a, b) < \mathrm{lse}(a, b) \le \max(a, b) + \log 2, \tag{3.12}$$
   右側の等号が $a = b$ のとき、かつそのときに限り成立することを証明し、さらに $\max(a, b)$ との差を任意に小さくできることを示せ。ヒント：一般性を失うことなく $b = 0$ かつ $a \ge 0$ と置いてよい。
   2. 任意の $\lambda > 0$ に対して
   $$\max(a, b) < \lambda^{-1}\mathrm{lse}(\lambda a, \lambda b) \le \max(a, b) + \lambda^{-1}\log 2, \tag{3.13}$$
   が成り立つことを証明し、$\lambda^{-1}\mathrm{lse}(\lambda a, \lambda b) \to \max(a, b)$（$\lambda \to \infty$ のとき）を結論づけよ。
   3. 同様にして滑らかな最小値関数を構成せよ。
   4. 定義と両側の上界・下界を q 個の数に拡張し、$\max_i a_i$ との差が高々 $\log q$ であることを示せ。

5. **温度**。$P = \mathrm{softmax}(\mathbf{o})$ とし、温度 T における分布を $\mathrm{softmax}(\mathbf{o}/T)$ とする。また、別の分布 Q を $Q(i) \propto P(i)^\alpha$（$\alpha > 0$）により定める。
   1. 温度を 2 倍にすることに対応する α の値は何か。また、温度を半分にすることに対応する値は何か。
   2. ちょうど s 個のロジット（全 q 個のうち）が最大値をとるとする。$\mathrm{softmax}(\mathbf{o}/T)$ の $T \to 0$ における極限を求めよ。
   3. $\bar{o} = \frac{1}{q}\sum_j o_j$ をロジットの平均とする。大きな T に対して次式が成り立つことを示せ。
   $$\mathrm{softmax}(\mathbf{o}/T)_i = \frac{1}{q} + \frac{o_i - \bar{o}}{qT} + O(T^{-2}). \tag{3.14}$$
   4. 温度スケーリングでは、分類器のロジットを固定したままホールドアウト集合上の交差エントロピーを最小化することで単一の T を当てはめる。この交差エントロピーが $\beta = 1/T$ の凸関数であり、したがってこの当てはめが 1 次元の凸最適化問題になることを示せ。ヒント：g の凸性（演習問題 2）を用いればよい。

6. **分離可能なデータ**。●あるパラメータ $(\mathbf{W}, \mathbf{b})$ がすべての訓練例を引き分けなしに正しく分類するとする。すなわち、どのデータ例においても正解クラスのロジットが他のすべてのロ

ジットを厳密に上回るとする。ここで、スケールされたパラメータ $(c\mathbf{W}, c\mathbf{b})$（$c > 0$）を考える。

1. すべての訓練例の交差エントロピー損失が、c の増加に伴って狭義単調減少し、0 に収束する（$c \to \infty$ のとき）ことを示せ。
2. 訓練損失が最小点を持たず、訓練損失が 0 に収束するような任意のパラメータ列は非有界なノルムを持つことを結論づけよ。
3. 2.7 節の重み減衰ペナルティを重みとバイアスの両方に適用した項 $\frac{\lambda}{2}(\|\mathbf{W}\|^2 + \|\mathbf{b}\|^2)$（$\lambda > 0$）を訓練損失に加える。このペナルティ付き損失が最小点を持つことを示せ。

7. **確率に対する二乗誤差**。交差エントロピーの代わりに、二乗損失 $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2$ を用いて予測確率をワンホットラベルに当てはめることも考えられる。
   1. ソフトマックスの偏導関数が $\partial \hat{y}_k / \partial o_j = \hat{y}_k(\mathbf{1}(j = k) - \hat{y}_j)$ となることを示し、これを用いてロジットに関する二乗損失の勾配を計算せよ。
   2. 2 クラスの場合について、両方の損失をマージン m の関数として表し、m に関する導関数を計算せよ。さらに $m \to -\infty$ のとき、すなわち確信を持った誤りに対して、二乗損失の導関数は 0 に収束する一方、交差エントロピーの導関数は -1 に収束することを示せ。
   3. 確信を持った誤った予測から訓練を開始するモデルにとって、この違いが何を意味するかを説明せよ。

8. [code] **Gumbel-max トリック**。●各ロジットに独立なノイズを加えて摂動させ、摂動後のスコアが最大となるクラス $y = \operatorname{argmax}_i (o_i + \gamma_i)$ を出力することでラベルが生成されるとする。ガウスノイズを用いるとプロビットモデルが得られるが、そのクラス確率は閉じた形を持たない。一方、ある別のノイズ分布を用いるとソフトマックス確率が厳密に再現される。$\gamma_1, \dots, \gamma_q$ を、累積分布関数が $P(\gamma \le z) = \exp(-\exp(-z))$ で与えられる標準ガンベル分布からの独立同一分布に従うサンプルとする。
   1. $P(o_k + \gamma_k \le z) = \exp(-\exp(o_k)\exp(-z))$ を示せ。
   2. $P\left(\operatorname{argmax}_i (o_i + \gamma_i) = k\right) = \operatorname{softmax}(\mathbf{o})_k$ を示せ。ヒント：$o_k + \gamma_k$ の値で条件付けし、置換 $u = \exp(-z)$ を用いればよい。
   3. ノイズを $\gamma_i \mapsto T\gamma_i$ とスケーリングすると、$\operatorname{softmax}(\mathbf{o}/T)$ からのサンプリングになることを結論づけよ。
   4. 結果を数値的に検証せよ。$\mathbf{o} = (1.0, 2.2, 0.3)$ および $T \in \{0.25, 1, 4\}$ に対して、10^5 個の $\operatorname{argmax}_i(o_i + T\gamma_i)$ のサンプルを生成し、ラベルの頻度がモンテカルロ誤差の範囲内で $\operatorname{softmax}(\mathbf{o}/T)$ と一致することを確認せよ。

3.2 画像分類データ

ソフトマックス回帰（softmax regression）を訓練してテストするには、弱いモデルと強いモデルを見分けられるほど十分に豊かでありながら、対話型ノートブック内で数秒のうちに訓練できるほどコンパクトな画像分類（image classification）ベンチマークが必要である。古典的な手書き数字の MNIST (LeCun, Léon Bottou, et al. 1998)（70,000 枚のグレースケール 28×28 画像、10 個の数字クラス）は 1990 年代には画期的なベンチマークであったが、今日ではごく単純なモデルでさえ 97% の正解率（accuracy）を超え、畳み込みニューラルネットワークは 99% を上回るため、アーキテクチャの改善効果を確認する余地がほとんど残っていない。その対極にある ImageNet (Deng et al. 2009)（1.2 百万枚のカラー写真、1,000 カテゴリ）では、実行ごとに数時間の GPU 計算を要する。

Fashion-MNIST (Xiao et al. 2017) はちょうど良いバランスを備えている。MNIST と同じ規模（訓練用 60,000 枚、テスト用 10,000 枚からなるサイズ 28×28 のグレースケール画像で、均等

な 10 クラス）を保ちつつ、数字を衣料品（T シャツ、ズボン、プルオーバー、ドレス、コート、サンダル、シャツ、スニーカー、バッグ、アンクルブーツ）に置き換えている。Fashion-MNIST では、線形ソフトマックス回帰の正解率は約 83% に達する一方、畳み込みネットワークは 90% を超える。本節では、以降のいくつかの章にわたって画像分類器にデータを供給する `FashionMNIST(d2l.DataModule)` クラスを構築する。

```
import time
import torchvision
from torchvision import transforms
```

```
d2l.use_svg_display()
```

3.2.1 データセットの読み込み

Fashion-MNIST には訓練画像 6,000 枚とテスト画像 1,000 枚が 10 個の衣類カテゴリそれぞれに含まれており（合計で 60,000 枚と 10,000 枚）、8 ビット符号なし整数として $[0, 255]$ の範囲で格納されている（0 が暗い背景を表す）。オブジェクト指向設計（2.4 節）に従い、`FashionMNIST` は `d2l.DataModule` を継承し、`batch_size` および任意の空間サイズ変更タプル `resize`（デフォルトは $(28, 28)$）を受け取る。

`torchvision.datasets.FashionMNIST` はデータセットをダウンロードし、各分割のすべての画像を 1 つの `uint8` テンソルに保持する。以下で定義するデータローダはこのテンソルを読み込み、ミニバッチ単位でまとめて変換する。`transforms.Compose` パイプラインは 1 枚の画像をリサイズし、そのピクセル輝度を float32 テンソルとして $[0, 1]$ の範囲にスケーリングする。この変換は、`data.train[i]` のようにデータセットへ直接インデックスアクセスするときや、`data.train` を独自のローダに渡すときに適用される。

```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

データセットをインスタンス化し、各分割のサイズを確認しよう。

```
data = FashionMNIST()
len(data.train), len(data.val)
```

```
(60000, 10000)
```

Fashion-MNIST には 2 つの分割しか用意されていないため、訓練中は 10,000 枚のテスト分割を検証セット（validation set）として使用する（テストセットで繰り返し評価することの統計的影響については第 3.5 章で分析する）。さらに、`text_labels` を追加し、整数のクラスインデックス $0, \ldots, 9$ を人間が読める衣類名に変換できるようにしよう。

```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    """Return text labels."""
    labels = ['T-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

3.2.2 ミニバッチの読み込み

深層学習フレームワークでは、サイズ n のミニバッチに含まれる、チャネル数 c（グレースケールなら $c = 1$、RGB カラーなら $c = 3$）、高さ h、幅 w の各画像は、フレームワークに応じて**チャネルファースト**のレイアウト (n, c, h, w) または**チャネルラスト**のレイアウト (n, h, w, c) のいずれかで格納される。

PyTorch は画像をチャネルファーストのレイアウト (n, c, h, w) で格納する。

次に、`get_dataloader(self, train)` を `FashionMNIST` に追加し、フレームワーク標準のデータローダをラップする。これにより、エポックごとに訓練分割を再シャッフルするとともに、最後の端数のミニバッチ（$60{,}000 = 937 \times 64 + 32$）も保持して、すべての画像がエポックごとに 1 回ずつ処理されるようにする。

ヘルパー `batchify` は、ローダが各ミニバッチを読み込む際に、それを float32 で $[0, 1]$ の範囲にスケーリングしてリサイズする。ミニバッチ全体に対する 1 回の演算は画像ごとに個別の演算を繰り返すよりもはるかに低コストであるため、このローダは画像単位の変換を経由しない。

```
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    # Serve the 8-bit image tensor directly, not one PIL image at a time
    X = data.data.unsqueeze(1)  # Add the channel axis
    return torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(X, data.targets), self.batch_size,
        shuffle=train, collate_fn=self.batchify)

@d2l.add_to_class(FashionMNIST)
def batchify(self, examples):
    """Stack examples into a minibatch, scale to [0, 1], resize if asked."""
    X, y = torch.utils.data.default_collate(examples)
    X = X.float() / 255
    if X.shape[2:] != tuple(self.resize):
        X = F.interpolate(X, size=self.resize, mode='bilinear', antialias=True)
    return X, y
```

1 つのミニバッチ `(X, y)` を `train_dataloader()` から取得し、単一画像のレイアウトならびにミニバッチ全体のテンソルの形状と dtype を確認しよう。

```
X, y = next(iter(data.train_dataloader()))
X[0].shape  # Channel-first: (channels, height, width)
```

```
torch.Size([1, 28, 28])
```

データ例を束ねてミニバッチにすると、画像の先頭にバッチ軸が追加される。一方、ラベルは画像ごとに 1 つの要素を持つ 1 次元ベクトルとして得られる。

```
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 28, 28]) torch.float32 torch.Size([64]) torch.int64
```

ミニバッチあたり $64 \times 28 \times 28 \times 4 \approx 200$ KB ($\sim$ 190 MB で 60,000 枚の float32 訓練画像すべてに相当) と小さいため、データセット全体が RAM に余裕をもって収まる。それでも、データローダ単体でベンチマークをとること（モデルを接続せずに 1 パス全体の時間を `train_dataloader()` について計測すること）は極めて重要な習慣である。どの訓練エポックも、そのデータパイプラインより速く実行することはできないからである。

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'0.42 sec'
```

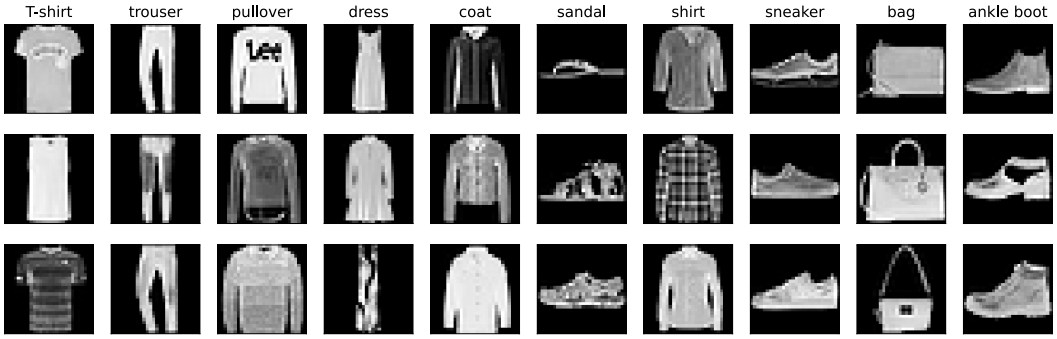
3.2.3 画像の確認

どのようなモデルを訓練する前にも、必ずラベル付き画像のサンプルを目視で確認すべきである。ラベルのずれ、輝度の反転、あるいは誤ったリシェイプによる画像の崩れは、実行時エラーを起こさないまま訓練を台無しにするからである。そこで、単一チャネルの軸を除去し、テキストラベルをタイトルに添えたグレースケール画像のグリッドを描画する `visualize` メソッドを `FashionMNIST` に追加する。

```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=None):
    """Show the images of a batch in grayscale, titled by their labels."""
    X, y = batch
    if labels is None:
        labels = self.text_labels(y)
    X = d2l.reshape(X, (-1, *self.resize))  # Drop the channel axis
    d2l.show_images(X, nrows, ncols, titles=labels, cmap='gray')
```

検証ローダが常に決定論的な順序で画像を返すことを利用して、各クラスの最初の 3 枚の検証画像をカテゴリごとに 1 列ずつ表示してみよう。

```
X, y = next(iter(data.val_dataloader()))
labels = d2l.numpy(y)
idx = [int(np.flatnonzero(labels == c)[r]) for r in range(3) for c in range(10)]
titles = data.text_labels(range(10)) + [''] * 20
data.visualize([d2l.stack([X[i] for i in idx]), None], nrows=3, ncols=10,
               labels=titles)
```



ざっと目を通すだけでも、線形分類器が直面する難しさが予見できる。ズボンとバッグは紛れもないシルエットを持ち、サンダル・スニーカー・アンクルブーツは明確な履物のクラスタを形成している。一方、4 つの上半身の衣類（T シャツ、プルオーバー、コート、シャツ）はほぼ同じ箱型の胴体輪郭を共有しており、28×28 の解像度ではかろうじて判別できる程度の襟、袖、ボタンのわずかな違いしか見られない。

3.2.4 まとめ

Fashion-MNIST は、訓練用 60,000 枚およびテスト用 10,000 枚のグレースケール 28×28 画像を均等な 10 個の衣類カテゴリにわたって提供しており、線形モデル（$\sim 83\%$）と畳み込みネットワーク（$> 90\%$）の性能差を見分けられるほど豊かでありながら、数秒で訓練できるほど小さい。本節で構築した `FashionMNIST(d2l.DataModule)` パイプラインは、生の `uint8` ピクセルを `float32` で $[0, 1]$ の範囲に再スケーリングし、任意の空間リサイズに対応するとともに、再シャッフルされたミニバッチと整数ラベルを各フレームワーク標準のチャネルレイアウトで供給する。ローダ単体のベンチマークはエポック時間の下界を与え、ラベル付き画像の一覧を目視確認することで、訓練開始前に表面化しない形状やラベルの不整合を発見できる。

3.2.5 演習問題

1. [code] **クラス平均**。訓練セット全体で各クラスの平均画像を計算し、10 個の平均画像を画像のグリッドとして表示せよ。
    1. クラス平均のすべてのペア間のユークリッド距離を計算し、その距離を 10×10 の表として示せ。どのペアが最も近く、他のすべてのクラスから最も遠いのはどのクラスか。
    2. この表から、線形分類器が最も混同しやすいクラスと、ほとんど混同しないクラスを予測せよ。この予測は控えておけ。3.3 節でその答えを計算する。
    3. 平均画像は近いにもかかわらず人間なら容易に見分けられるクラスのペアを挙げ、どのピクセルがその違いを担っているかを述べよ。
2. [code] **ローダのスループット**。訓練セットに対する 1 パスの時間は、ミニバッチのサイズと、ローダが供給する画像のサイズに依存する。
    1. 1 パス全体の時間を `train_dataloader` について計測し（`FashionMNIST` は `batch_size` を 1、16、64、256、1024 として構築する）、毎秒の画像枚数によるスループットを `batch_size` に対してプロットせよ。さらに、バッチごとのオーバーヘッドと画像ごとの処理量の観点から曲線の形状を説明せよ。
    2. `batchify` は、ローダが各ミニバッチを読み込む際にそれをリサイズする。1 パスの時間を `train_dataloader` について計測し（`FashionMNIST(resize=(s, s))` で $s = 28$、56、112、224 とする）、画像あたりのピクセル数に対してその時間をプロットせよ。時間のどの部分がピクセル数に伴って増加し、どの部分は増加しないか。

3. [code] **アップサンプリングと線形モデル**。●28×28 から 32×32 へのリサイズを、線形変換 $\mathbf{x}' = \mathbf{A}\mathbf{x}$（元の 784 ピクセルからリサイズ後の 1024 ピクセルへの変換であり、$\mathbf{A} \in \mathbb{R}^{1024\times784}$ の各列は線形独立）として扱え。
   1. $\mathbf{x}'$ 上のソフトマックス回帰が表現できる分類器の集合は、$\mathbf{x}$ 上のソフトマックス回帰が表現できる分類器の集合と完全に一致することを示せ。
   2. 3.3 節を終えた後、`SoftmaxRegressionScratch` を `FashionMNIST()` のデータと `FashionMNIST(resize=(32, 32))` のデータでそれぞれ訓練せよ（`num_inputs` は画像あたりのピクセル数に設定する）。検証正解率を比較し、前問の結果を踏まえて差異があればその理由を説明せよ。

3.3 ソフトマックス回帰の実装

第 2 章で線形回帰をスクラッチと簡潔な方法の両方で実装したのと同様に、本節では Fashion-MNIST 上でソフトマックス回帰（softmax regression）をこの 2 通りの方法で構築する。まずは生のテンソルから出発し、行ごとのソフトマックス、ベクトル化した交差エントロピー損失（cross-entropy loss）、`SoftmaxRegressionScratch` のパラメータ（$\mathbf{W} \in \mathbb{R}^{784\times10}, \mathbf{b} \in \mathbb{R}^{10}$）を、再利用可能な基底クラス `Classifier(d2l.Module)` とあわせて実装する。これを用いて検証正解率が約 83% に達するまで訓練し、10×10 の混同行列（confusion matrix）と 28×28 の重みテンプレートを通して線形モデルが何を学習したかを調べる。続く 3.3.5 節では、手動のテンソル管理と確率のクランプ処理を組み込みの線形層および生のロジットから直接計算するシフト付き **log-sum-exp** 交差エントロピー損失に置き換え、浮動小数点のオーバーフローとアンダーフローを一挙に解消する。

3.3.1 ソフトマックス

3.3.1.1 各行の正規化

n 個の例からなるミニバッチに対して、ロジットは $n \times q$ 行列 $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ をなす（3.1.1.4 節）。ソフトマックスは各行を独立に正規化する。

$$\text{softmax}(\mathbf{O})_{ij} = \frac{\exp(O_{ij})}{\sum_{k=1}^{q} \exp(O_{ik})}. \tag{3.15}$$

コードでは、`O` の指数をとり、列方向（`axis=1`）に `keepdims=True` で和をとることで行和の形状を `(n, 1)` に保ち、ブロードキャストによって除算する。

```
def softmax(O):
    O_exp = d2l.exp(O)
    return O_exp / d2l.reduce_sum(O_exp, 1, keepdims=True)  # Broadcast
```

2 つの行で `softmax` の動作を確認してみよう。猫・鶏・犬のロジット $(1.0, 2.2, 0.3)$（3.1 節）と、同じ 0 が 3 つ並ぶ行である。

```
O = d2l.tensor([[1.0, 2.2, 0.3], [0.0, 0.0, 0.0]])
Y_hat = softmax(O)
Y_hat, d2l.reduce_sum(Y_hat, 1)
```

```
(tensor([[0.2076, 0.6893, 0.1031],
         [0.3333, 0.3333, 0.3333]]),
 tensor([1., 1.]))
```

3.3.1.2 オーバーフロー

なぜこの 3 行の `softmax` の式を本番環境で決して使ってはならないのだろうか。32 ビット浮動小数点（`float32`）では表現可能な数値の上限が 3.4×10^{38} 付近にあるため、$\exp(o)$ は $o > 88.7$ になった途端に `inf` へオーバーフローするからである。`softmax` にロジット $(1000, 0, 0)$ を入力すると ∞/∞ が計算されて `NaN` になるが、真の確率分布は機械精度の範囲で $(1, 0, 0)$ である。

3.1 節で証明したように、ソフトマックスは行内のすべてのロジットを同じ定数だけシフトしても不変である。指数をとる前に行の最大値 $o_{\max} = \max_k o_k$ を引けば、すべての指数 $o_j - o_{\max}$ が ≤ 0（最大値が 0）となるため、オーバーフローは起こり得ない。どのフレームワークの組み込みソフトマックスも、内部でこのシフトを適用している。

```
O = d2l.tensor([[1000.0, 0.0, 0.0]])
O_max = O.amax(dim=1, keepdim=True)
softmax(O), softmax(O - O_max), torch.softmax(O, dim=1)
```

```
(tensor([[nan, 0., 0.]]), tensor([[1., 0., 0.]]), tensor([[1., 0., 0.]]))
```

以下の Fashion-MNIST のスクラッチ実行では、ピクセル値が $[0, 1]$ に収まり重みも小さいままであるため、訓練中に `softmax` がオーバーフローすることはない。しかし、シフトだけでは非常に小さな負のロジットに対して**アンダーフロー**が生じる危険が残る（float32 では $\exp(-200) = 0.0$ となり、$\log 0$ が $-\infty$ になる）。後述の 3.3.5 節では、対数空間でソフトマックスと対数を融合させることで、この両方の危険を恒久的に取り除く。

3.3.2 損失と正解率

3.3.2.1 ロジットの 2 つの用途

勾配降下法は滑らかな交差エントロピー損失を最適化するが、人間は分類器を**正解率**（accuracy）、すなわち最高スコアのクラスが正解ラベル $y^{(i)}$ と一致する例の割合によって評価する。

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\Big(\operatorname*{argmax}_j o_j^{(i)} = y^{(i)}\Big). \tag{3.16}$$

どちらの指標もまったく同じロジットベクトルから計算され（Figure 3.5）、検証時には両方を追跡することが不可欠である。モデルが一部の誤分類に対して過剰に確信を持つようになると、検証正解率が上昇し続けているにもかかわらず、検証交差エントロピー損失が増加することがあるからである。

3.3.2.2 `Classifier` クラス

この 2 指標による検証の振る舞いを、本書におけるすべての分類モデルの基底クラスとなる `Classifier(d2l.Module)` にまとめる。まずは各検証ミニバッチに対して `validation_step` が `'loss'` と `'acc'` の両方を返すように定義する。

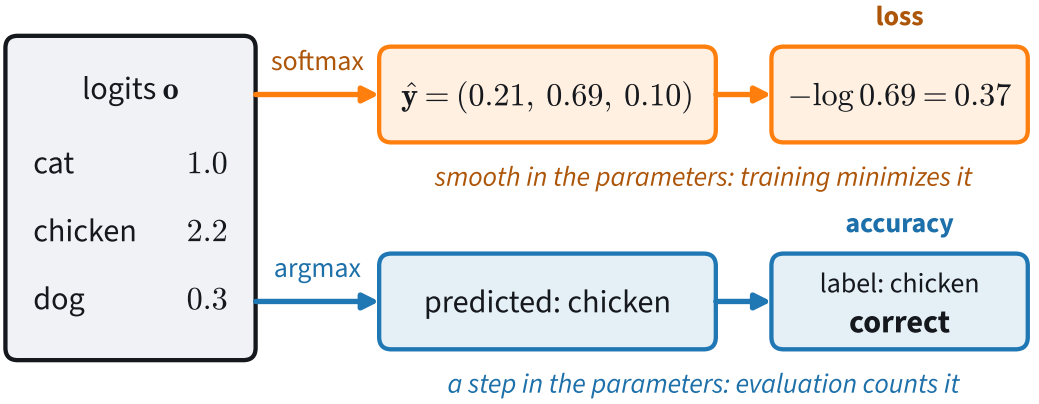


Figure 3.5: 訓練時（上）はロジットをソフトマックスに通して滑らかな交差エントロピー損失へ写像し、評価時（下）は $\mathrm{argmax}_j o_j$ をラベル y と比較して離散的な正解率を計算する。

```
class Classifier(d2l.Module):
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        return {'loss': self.loss(Y_hat, batch[-1]),
                'acc': self.accuracy(Y_hat, batch[-1])}
```

次に、`accuracy(self, Y_hat, Y, averaged=True)` を `Classifier` に追加する。このメソッドは `Y_hat` を 1 例あたり 1 行のクラススコアへと平坦化し、`argmax` を `axis=1` に沿って適用して予測クラスのインデックスを求め、`Y.dtype` にキャストしたうえで、平均値または例ごとの float32 等値マスク `preds == Y` を返す。

```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the fraction of correct predictions."""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, (-1,)), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

最後に、すべての分類器がそのまま使えるオプティマイザを継承できるよう、学習率 `self.lr` の通常のミニバッチ SGD をデフォルトの `configure_optimizers` メソッドとして `d2l.Module` に組み込む。

```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

3.3.3 モデルと損失

3.3.3.1 パラメータと順伝播

28×28 の各 Fashion-MNIST 画像を平坦化すると、入力特徴量は $d = 784$ 個、クラス数は $q = 10$ 個となるため、`SoftmaxRegressionScratch` では $\mathbf{W} \in \mathbb{R}^{784\times 10}$（$\mathcal{N}(0, 0.01^2)$ から初期化）と $\mathbf{b} \in \mathbb{R}^{10}$（0 で初期化）の計 7,850 個のパラメータを確保する。ここで簡単な妥当性確認をしておこう。ゼロに近い初期重みからはゼロに近いロジットが生じるため、未訓練のモデルは一様な予測（$\hat{y}_j \approx 0.1$）に近い状態から始まり、初期の交差エントロピー損失は $-\log(0.1) = \log 10 \approx 2.30$ 付近となる。

```
class SoftmaxRegressionScratch(Classifier):
    """Softmax regression implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = nn.Parameter(d2l.normal(0, std, (num_inputs, num_outputs)))
        self.b = nn.Parameter(d2l.zeros(num_outputs))
```

`forward(self, X)` では、ミニバッチ内の各画像を長さ `self.num_inputs`（784）の行ベクトルへと `d2l.reshape(X, (-1, self.num_inputs))` によって平坦化し、アフィン変換によるロジット $\mathbf{XW}+\mathbf{b}$ を `softmax` に通す。

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return softmax(d2l.matmul(X, self.W) + self.b)
```

3.3.3.2 交差エントロピー損失

3.1.2 節で見たように、$l(\mathbf{y}, \hat{\mathbf{y}}) = -\log \hat{y}_y$ であった。ワンホット行列を実体化して内積をとる代わりに、ベクトル化された 1 回のインデックス参照によって、正解クラスの確率 $\hat{y}^{(i)}_{y^{(i)}}$ をすべての行 i について抜き出す。

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

これにより、1 つ目の例から 0.1、2 つ目の例から 0.5 が抽出される。これを `cross_entropy` にまとめ、ミニバッチ全体で負の対数の平均をとる。対数をとる前に、防御策として下限を 10^{-12} にクランプしている点に注目してほしい。こうすることで、アンダーフローして 0 になった確率が $-\infty$ へと発散するのを防ぎ、1 例あたりの損失の上限を 27.6 に抑えられる。ただし、クランプが作動するとその箇所の勾配も 0 になってしまう。本節の後半（3.3.5 節）では、この場しのぎの処置を、ロジットから直接計算する融合型の損失に置き換える。

```
def cross_entropy(y_hat, y):
    # Tiny clip to keep log finite when softmax outputs underflow to 0.
```

```
    p = y_hat[list(range(len(y_hat))), y].clamp(min=1e-12)
    return -d2l.reduce_mean(d2l.log(p))

cross_entropy(y_hat, y)
```

```
tensor(1.4979)
```

結果は $-\log 0.1 \approx 2.30$ と $-\log 0.5 \approx 0.69$ の平均、すなわち約 1.50 となる。これをモデルの `loss` メソッドとして追加しよう。

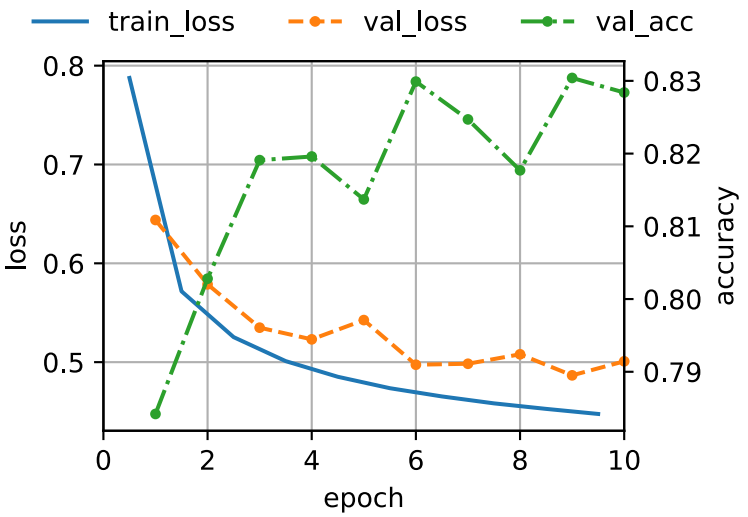
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

3.3.4 訓練と評価

3.3.4.1 訓練の実行

`SoftmaxRegressionScratch` を Fashion-MNIST 上で 10 エポック、ミニバッチサイズ 256、学習率 $\eta = 0.1$ で訓練してみよう。

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



訓練済みモデルを任意のタイミングでデータセット分割全体に対して評価できるよう、補助メソッド `evaluate(self, model, data, train=False)` を `d2l.Trainer` に追加する。このメソッドは `self.val_dataloader` に検証ローダまたは訓練ローダを紐づけて `self.validate()` を実行する。

```
@d2l.add_to_class(d2l.Trainer)
def evaluate(self, model, data, train=False):
    """Metrics of `model` on the validation data of `data` (on its training
    data if `train`), averaged over examples."""
    self.train_dataloader = None
    self.val_dataloader = (data.train_dataloader() if train
                           else data.val_dataloader())
    self.prepare_model(model)
    return self.validate()
```

全 10,000 枚の検証画像にわたって平均したこれらの指標は、`trainer.history['val_loss'][-1]` および `trainer.history['val_acc'][-1]` に記録された最終値と一致する。

```
trainer.evaluate(model, data)
```

```
{'loss': 0.500844775390625, 'acc': 0.8284}
```

この分類器は検証画像の約 83% を正しく分類できている。`train=True` を渡せば、まったく同じ重みを全 $60{,}000$ 枚の訓練画像に対して評価できる。

```
trainer.evaluate(model, data, train=True)
```

```
{'loss': 0.46587086588541665, 'acc': 0.8420666666666666}
```

10 エポック後、検証正解率は 83% 付近（乱数シードにより $\pm 1\%$ 変動）に落ち着き、訓練正解率は 84% 付近となる。この 1% というごく小さな汎化ギャップ（generalization gap）は、線形分類器の表現能力の限界を反映している。$7{,}850$ 個のパラメータでは $60{,}000$ 枚の画像を丸暗記することはできない（第 3.5 章）。

3.3.4.2 誤分類された画像

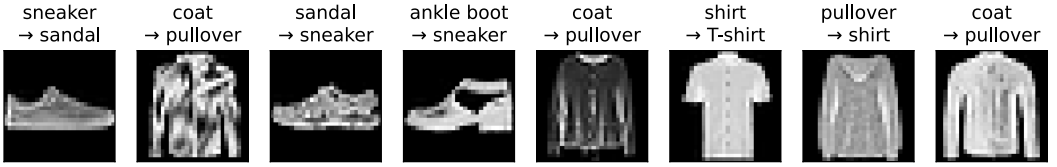
モデルが**どのように**誤るかを確認するため、最初の検証ミニバッチ（256 枚の画像）を取り出して誤分類の数を数えてみよう。

```
X, y = next(iter(data.val_dataloader()))
preds = d2l.astype(d2l.argmax(model(X), axis=1), y.dtype)
wrong = preds != y
print(f'{int(d2l.reduce_sum(d2l.astype(wrong, d2l.int32)))} of {len(y)} wrong')
```

```
35 of 256 wrong
```

誤分類された最初の 8 枚の画像を、正解ラベルをモデルの予測の上に添えて（`true` → `pred`）表示してみよう。

```
labels = [f'{true}\n→ {pred}' for true, pred in zip(
    data.text_labels(y[wrong]), data.text_labels(preds[wrong]))]
data.visualize([X[wrong], y[wrong]], labels=labels)
```



3.3.4.3 混同行列

全 10,000 枚の検証画像にわたる誤りのパターンを要約するため、10×10 の**混同行列** $\mathbf{N} \in \mathbb{R}^{q \times q}$ を集計する。ここでは、行が予測クラス $\hat{c}^{(m)} = \operatorname{argmax}_j o_j^{(m)}$ を、列が正解クラス $y^{(m)}$ を表す。

$$N_{ij} = \sum_{m=1}^{n} \mathbf{1}(\hat{c}^{(m)} = i)\, \mathbf{1}(y^{(m)} = j). \tag{3.17}$$

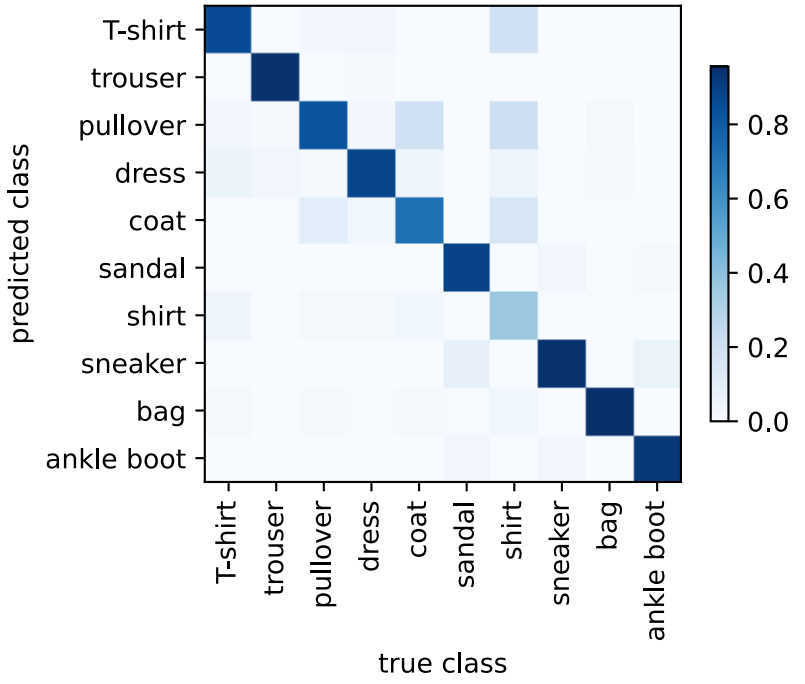
（正解クラスを列、予測クラスを行に配置しておくと、列正規化した条件付き確率の積が $\mu = \mathbf{Cp}$ とすっきり書け、3.6 節でラベルシフト（label shift）を扱う際に便利である。）ミニバッチの予測クラスと正解クラスを表すワンホット行列 $\mathbf{P}, \mathbf{Y} \in \{0,1\}^{n \times q}$ を用いれば、積 $\mathbf{P}^\top \mathbf{Y}$ によってミニバッチあたり 1 回の行列積ですべての 100 マス分のカウントを累積できる。

```
def one_hot(y, q):
    """Rows of the q x q identity matrix, one per class index in y."""
    return d2l.astype(d2l.reshape(y, (-1, 1)) == d2l.arange(q, dtype=y.dtype),
                      d2l.float32)

N = d2l.zeros((10, 10))
for X, y in data.val_dataloader():
    P = one_hot(d2l.argmax(model(X), axis=1), 10)
    N += d2l.matmul(d2l.transpose(P), one_hot(y, 10))
int(d2l.reduce_sum(N))
```

```
10000
```

これで全 10,000 枚の検証画像が集計された。生のカウント数はクラスの出現頻度に依存するため、$\mathbf{N}$ の各列をその列和で正規化して $C_{ij} = N_{ij} / \sum_{i'} N_{i'j}$ を求める。これは条件付き確率 $P(\hat{c} = i \mid y = j)$ の経験推定値である。各列は正解クラス j の例がどの予測クラスに振り分けられたかを示し、対角成分はクラスごとの認識率を与える。$\mathbf{C}$ を `d2l.show_heatmaps` でヒートマップとして描画してみよう。

```
C = N / d2l.reduce_sum(N, 0, keepdims=True)  # Each column sums to 1
names = data.text_labels(range(10))
d2l.show_heatmaps(d2l.reshape(C, (1, 1, 10, 10)), xlabel='true class',
                  ylabel='predicted class', figsize=(4, 4), cmap='Blues',
                  xticklabels=names, yticklabels=names)
```



最も頻繁に生じている混同を浮き彫りにするため、非対角成分の上位 5 件も出力してみよう。

```
mistakes = d2l.numpy(N) * (1 - np.eye(10))  # Zero the diagonal
for k in np.argsort(mistakes, axis=None)[::-1][:5]:
    i, j = divmod(int(k), 10)
    print(f'{names[j]} taken for {names[i]}: {int(mistakes[i, j])}')
```

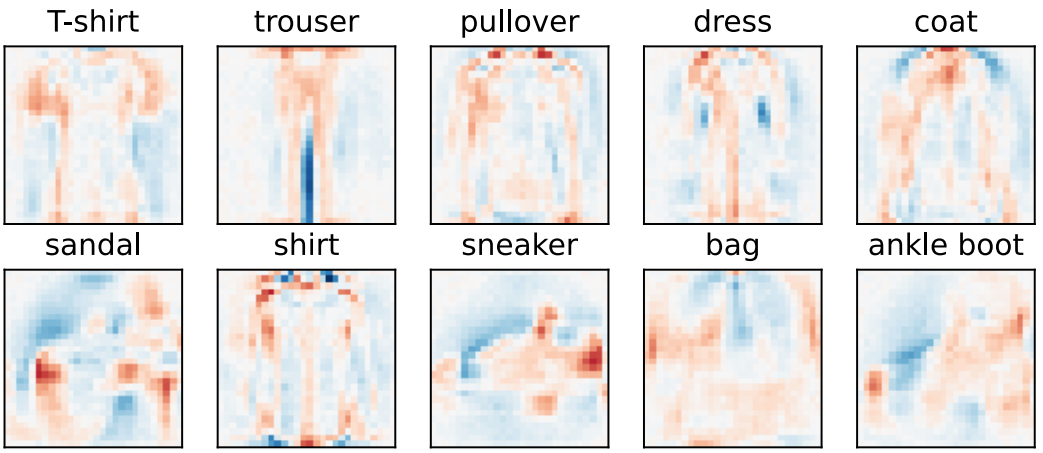
```
shirt taken for pullover: 207
coat taken for pullover: 192
shirt taken for T-shirt: 192
shirt taken for coat: 151
pullover taken for coat: 102
```

この混同行列は、3.2 節での目視確認から予想された結果を裏づけている。ズボン、バッグ、履物は $\sim 90\%$ 以上の割合で正しく認識される一方、非対角成分の上位 5 件の混同はすべて、見た目のよく似た上半身のトップス 4 種（T シャツ、プルオーバー、コート、シャツ）の間で起きている。なかでもシャツはシルエットが他の 3 種のちょうど中間に位置するため、半分程度以下しか正しく認識されない。

3.3.4.4 学習されたテンプレート

各ロジット $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$ は、平坦化された画像 $\mathbf{x}$ と $\mathbf{W}$ の第 j 列との内積であるため、784 要素の各列 $\mathbf{w}_j$ を 28×28 の形状に戻せば、分類器全体を 10 枚の視覚的な**テンプレート**（赤は正の重み、青は負の重み）として観察できる。

```
W = d2l.numpy(model.W)
templates = W.T.reshape(10, 28, 28)  # Column j of W as an image
v = np.abs(templates).max()
d2l.show_images(templates, 2, 5, titles=names, cmap='RdBu_r', vmin=-v, vmax=v);
```



各テンプレートには対応する衣類の淡いシルエットがはっきりと浮かび上がっている。ドレスに対して負の重みを与えるズボンの両脚の間の青い帯や、主に襟元と袖周りしか違わない 4 種のトップスのよく似た胴体形状に注目してほしい。線形層はクラスごとに固定された大域的なドット積を 1 回とることしかできないため、襟やボタンのような微細な局所的特徴ではシルエット全体の重なりを克服できない。$\sim 83\%$ の壁を超えるには、非線形な隠れ層（第 4 章）や空間的な畳み込み（第 6 章）が必要となる。

3.3.5 簡潔な実装

ここからは、深層学習フレームワークを用いることで、素朴な `softmax` やクランプ付きの `cross_entropy` が抱えていたオーバーフローとアンダーフローの危険を恒久的に解消しつつ、ソフトマックス回帰をいかに簡潔に記述できるかを見ていこう。

3.3.5.1 モデルの定義

`SoftmaxRegression(d2l.Classifier)` では、平坦化ステップと出力数 10 の組み込み全結合層（fully connected layer）1 つによってロジット $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ を計算する（スクラッチ版のベースラインと揃えるため、重みは $\mathcal{N}(0, 0.01^2)$、バイアスは 0 で初期化する）。ここで決定的に重要なのは、**`forward` がソフトマックスを適用せず、生のロジット $\mathbf{O}$ を直接返す**ことである。

```
class SoftmaxRegression(d2l.Classifier):
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def weight_init(self, module):
```

```
        # The Trainer calls this after the lazy layer has learned its shape
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)

    def forward(self, X):
        return self.net(X)
```

3.3.5.2 安定な交差エントロピー損失

なぜ `SoftmaxRegression.forward` は確率を返さず、ロジットの段階で止めるのだろうか。まず $\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o})$ を計算し、次に $-\log \hat{y}_y$ を `float32` で計算するときに生じる浮動小数点の 2 つの危険を思い出そう。

1. **オーバーフロー：**$\exp(o)$ は $o > 88.7$ で `inf` へオーバーフローする。すべてのロジットから $o_{\max} = \max_k o_k$ を引けば、シフト後のすべての指数 $o_k - o_{\max}$ が ≤ 0 となるため、オーバーフローは解消される。
2. **アンダーフロー：**$o_{\max}$ によるシフト後であっても、ロジットが 104 を超えて $o_{\max}$ を下回るクラス（たとえばコード上でラベル 1 にあたる $\mathbf{o} = (0, -200, 0)$ の第 2 クラス）では、float32 において $\exp(o_y - o_{\max})$ が厳密な `0.0` へとアンダーフローする。すると $\log(0.0)$ をとることで `-inf` が生じる（あるいはスクラッチ版の `1e-12` クランプを使った場合は損失が 27.6 で頭打ちになり、**勾配が 0 になって**、自信満々に犯した大間違いからモデルが学習できなくなる）。

中間の確率を一切実体化しないことで、オーバーフローとアンダーフローの両方を回避できる。**LogSumExp トリック**によって対数とシフト付きソフトマックスを代数的に結合すると、除算は減算へと変わる。

$$l(y, \mathbf{o}) = \log \sum_{k=1}^{q} \exp(o_k) - o_y = o_{\max} + \log \sum_{k=1}^{q} \exp(o_k - o_{\max}) - o_y. \tag{3.18}$$

シフト後のすべての指数 $o_k - o_{\max}$ は ≤ 0 であり、かつ少なくとも 1 つは 0 に等しいため、対数の中の和は $[1, q]$ に、その対数は $[0, \log q]$ に収まる。オーバーフローも $\log 0$ も起こり得ず、自動微分（automatic differentiation）によって有限の勾配 $\mathrm{softmax}(\mathbf{o}) - \mathbf{y}$ が得られる。

どのフレームワークも、この融合型 log-sum-exp 交差エントロピーを提供している。これを `d2l.Classifier.loss` に一度組み込んでおけば、本書の残りのすべての分類器がそのまま継承できる。

```
@d2l.add_to_class(d2l.Classifier)
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

`SoftmaxRegression.loss` を極端なロジット `O = [[1000.0, 0.0, 0.0], [1000.0, 0.0, 0.0]]` と正解ラベル `y = [0, 1]` でテストすると、融合型 LogSumExp 実装がオーバーフローもアンダーフローも起こすことなく、例ごとの正確な損失 `[0., 1000.]` を返すことが確認できる。

```
O = d2l.tensor([[1000.0, 0.0, 0.0], [1000.0, 0.0, 0.0]])
y = d2l.tensor([0, 1])
SoftmaxRegression(num_outputs=3, lr=0.1).loss(O, y, averaged=False)
```

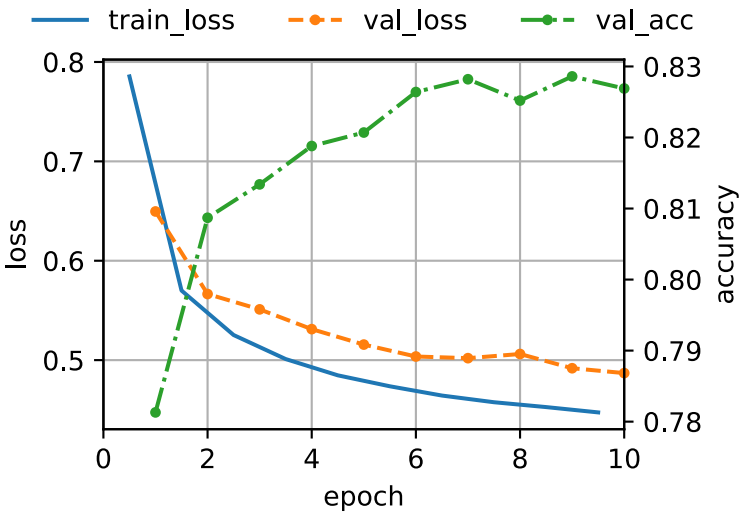
```
tensor([  -0., 1000.])
```

（典型的な落とし穴に注意しよう。`Classifier.loss` は生のロジットを受け取って内部で log-softmax を適用するため、誤って `forward` の末尾にソフトマックスを置くとソフトマックスが **2 回**適用され、訓練損失が正の下限値より下がらなくなってしまう。以下の演習問題 4 を参照のこと。）

3.3.5.3 訓練

`SoftmaxRegression` の `d2l.Trainer` による訓練は 4 行で記述でき、盤石な数値安定性（numerical stability）のもとで $\sim 83\%$ の検証正解率を再現する。

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



3.3.6 まとめ

ソフトマックス回帰をスクラッチで実装する過程で、再利用可能な基底クラス `Classifier` を導入した。このクラスは訓練時の滑らかな交差エントロピーの最適化と検証時の離散的な argmax 正解率を組み合わせ、Fashion-MNIST 上で 10 エポックのうちに約 83% の検証正解率に到達する。ワンホット行列の積 $\mathbf{P}^\top \mathbf{Y}$ を累積して 10×10 の混同行列を構築すると、学習された 28×28 の重みテンプレートが大きく重なり合う上半身の衣類 4 種に線形モデルの誤りが集中していることがわかる。`softmax` と `log` を別々に計算すると大きな正のロジットでオーバーフローし、大きな負のロジットで $\log 0$ へとアンダーフローするため、簡潔な実装では `forward` から生のロジットを

返し、シフト付き log-sum-exp 交差エントロピー $o_{\max} + \log \sum_k \exp(o_k - o_{\max}) - o_y$ を評価することで、あらゆる大きさのロジットに対して有限の値と損なわれない勾配を保証する。

3.3.7 演習問題

1. [code] **ワンホットラベルからの交差エントロピー**。本節の `cross_entropy` はインデックス参照によって正解クラスの確率を選択している。定義 $-\sum_j y_j \log \hat{y}_j$ に文字通り従い、`y` をワンホット行列に変換して行ごとの内積をとる別実装を作成せよ。
   1. これを `loss` として `SoftmaxRegressionScratch` に用い、訓練結果が変わらないことを確認せよ。
   2. 256 行、$q \in \{10, 10^3, 10^5\}$ 列のランダムな確率行列に対して両バージョンの実行時間を計測し、各バージョンが確保するメモリ量が q に応じてどのように増大するか述べよ。
   3. 内積による形式は適用できるが、インデックス参照による形式は適用できない設定を 1 つ挙げて説明せよ。
2. **大規模な語彙**。ソフトマックス回帰によって、次元 $d = 1024$ の特徴ベクトルから次の単語を予測するとする。語彙サイズは $V = 10^6$ 語であり、各ミニバッチは $n = 256$ 個の例を含む。`forward` と `cross_entropy` に含まれる演算のうち、コストが V に応じて増大するものを列挙せよ。また、行列積の積和演算回数、`W` が占める float32 メモリ量、および $n \times V$ の確率行列が占める float32 メモリ量を計算し、それぞれの数値を本節の Fashion-MNIST モデルと比較せよ。
3. [code] **テンプレートとクラス平均**。テンプレートとは、ミニバッチ SGD が観測したピクセルから作り上げたものである。
   1. `SoftmaxRegressionScratch` を `std=0` で再訓練し、すべての重みが 0 から始まり、すべてのクラスが同じロジットから始まるようにせよ。それでも訓練が問題なく進む理由を説明し、正解率とテンプレートを本文の結果と比較せよ。
   2. ゼロ重みから SGD を 1 ステップ進めた後、$\mathbf{W}$ の第 j 列が最初のミニバッチ上の $\sum_i \left(y_j^{(i)} - \frac{1}{10}\right) \mathbf{x}^{(i)}$、すなわちクラス j の画像の総和から全画像の総和の 10 分の 1 を引いたものに比例することを示せ。
   3. 訓練セット全体で各クラスの平均画像を計算し、目視およびピクセル間の相関によってそのクラスの訓練済みテンプレートと比較せよ。両者はどこで一致し、テンプレートはどこで平均から乖離しているか。また、単にクラス平均をコピーしただけのテンプレートでは、トップスの混同がさらに悪化する理由を説明せよ。
4. [code] **二重のソフトマックス**。本節の `loss` はロジットを入力として想定している。モデルの `forward` の末尾に誤ってソフトマックスを入れてしまったとしよう。このとき、損失関数はロジットの代わりに確率ベクトル $\mathbf{p}$ を受け取り、$-\log \mathrm{softmax}(\mathbf{p})_y$ を評価することになる。
   1. すべての確率ベクトル $\mathbf{p}$ とラベル y にわたってこの損失がとり得る最小値を、クラス数 q の関数として求めよ。また、$q = 10$ のときの値を計算せよ。
   2. 本節の前半で定義したスクラッチ版の `softmax`（`d2l` パッケージには含まれない）を用い、`SoftmaxRegression` を継承して `forward` がロジットにこの `softmax` を適用するようにせよ。これを `lr=0.1`、`d2l.FashionMNIST(batch_size=256)`、`d2l.Trainer(max_epochs=10)` で訓練せよ。その訓練損失と検証正解率を `SoftmaxRegression` と比較し、最終的な訓練損失を前問で求めた値と関連づけて説明せよ。
5. [code] **安定なソフトプラス**。ロジット $(x, 0)$ を持つ 2 クラスの場合、損失の第 1 項は $\log(1 + e^x)$、すなわち x の**ソフトプラス**となる。

   1. $\log(1+e^x)$ と $\max(x,0)$ を区間 $[-5,5]$ 上でプロットせよ。前者が後者を高々 $\log 2$ だけ上回る（最大差をとるのは $x=0$ のとき）ことを示し、log-sum-exp が滑らかな最大値と呼ばれる理由を 1 文で説明せよ。
   2. $\log(1+e^x) = \max(x,0) + \log(1+e^{-|x|})$ が成り立つことを示せ。
   3. 単精度において、素朴な式 `log(1 + exp(x))`、変形後の式、および `log1p` で `log(1 + ...)` を置き換えた変形後の式を $x \in \{-100, -20, -5, 0, 5, 20, 100\}$ に対して評価せよ。各値を倍精度の基準値と比較し、それぞれのバージョンがどこでオーバーフローし、どこで相対精度を完全に失うかを報告せよ。
   4. $x=-20$ における素朴な式の導関数を自動微分によって計算せよ。式の値自体は不正確であるにもかかわらず導関数が正確である理由と、これが訓練および高い確信度で分類された例の損失報告に対して何を意味するかを説明せよ。

6. [code] **生き残る勾配**。正解ラベル 1 に対するロジット $(0, -200, 0)$ では、クラス 1 のシフト付きソフトマックス確率が float32 で 0 にアンダーフローするため、クランプ付きのスクラッチ損失 `cross_entropy(softmax(O - O_max), y)` は下限 `1e-12` に張り付き、ロジットに関する勾配が 0 になる。同じロジットとラベルを融合型の `loss`（`SoftmaxRegression` のメソッド）に渡し、自動微分によってロジットに関する勾配を計算せよ。それを $\hat{\mathbf{y}} - \mathbf{y}$（3.1 節）と比較し、クランプ付きのスクラッチ損失ではクラス 1 のロジットが動かないのに対し、融合型の損失では動く理由を説明せよ。

7. **数値形式**。半精度形式は数値を 16 ビットで格納する。FP16 は指数部 5 ビット・仮数部 10 ビット、BFLOAT16 は 8 ビットと 7 ビット、FP32 は 8 ビットと 23 ビットを持つ。各形式について、ロジット o に対して $\exp(o)$ がオーバーフローしない最大の値を求めよ。次に、ロジットが 20 に達するネットワークは BFLOAT16 ではそのまま訓練できるが、FP16 ではソフトマックスを FP32 で計算しない限り訓練できない理由を説明せよ。最後に、Equation 3.18 のシフト付きの和を $q = 10^5$ クラス（各項は高々 1）について FP16 で累積する場合を考えよ。累積和が 2,048 を超えると何が起こるか、またそれと 10 ビットの仮数部はどのように関係しているか。

3.4 分類器の評価

Fashion-MNIST のソフトマックス回帰モデルは、検証画像の約 83% を正しく分類する。では、正解率（accuracy）が高いほど常に優れた分類器といえるだろうか。それが成り立つのは、すべてのクラスが同等に重要であり、どの種類の誤りもまったく同じコストを伴う場合に限られる。実運用では、次の 3 つの日常的な条件がこの対称性を崩す。

1. **クラス不均衡：**注目する陽性クラス（positive class）がまれな場合（たとえば、患者の 1% が疾患をもつ場合や、取引の 0.1% が不正である場合）、常に多数派クラスを予測する役に立たない分類器でも、陽性を 1 件も検出しないまま 99% や 99.9% の正解率を達成してしまう。
2. **非対称な誤りコスト：**命に関わる疾患を見逃すこと（偽陰性、false negative）は、追加検査を指示すること（偽陽性、false positive）よりもはるかにコストが大きい。
3. **確率に基づく意思決定：**後段の決定規則は予測確率 $\hat{\mathbf{y}}$ の数値そのものに基づいて動作するため、この値が信頼できるのは分類器が**較正**（calibration）されている場合に限られる。

本節では、これら 3 つすべてに対応する評価指標と診断手法として、**適合率**（precision）と**再現率**（recall）、**コスト考慮型の決定閾値**（cost-sensitive decision threshold）、**信頼性図**（reliability diagram）と**期待較正誤差**（expected calibration error、ECE）、そして事後的な**温度スケーリング**（temperature scaling）を順に導入する。

3.4.1 正解率の限界

3.4.1.1 クラス不均衡

100,000 人を対象に、有病率（prevalence）が 1% の疾患（病気 1,000 人、健康 99,000 人）のスクリーニング検査を実施するとしよう。すべての検査結果を無視して常に**健康**と予測する自明な規則は、病気の患者を 1 人も検出できないにもかかわらず 99% の正解率を達成する。これを、病気の患者の 90% を検出しつつ健康な人の 5% を誤って陽性と判定する実際のスクリーニング検査と比較してみよう。

	病気（陽性）	健康（陰性）
病気と予測	900 (TP)	4,950 (FP)
健康と予測	100 (FN)	94,050 (TN)

この 2×2 の二値の混同行列（confusion matrix）において、4 つのセルはそれぞれ**真陽性**（true positive、TP $= 900$）、**偽陽性**（FP $= 4{,}950$）、**偽陰性**（FN $= 100$）、**真陰性**（true negative、TN $= 94{,}050$）である。対角成分の和をとると正解率は $94{,}950/100{,}000 = 95\%$ となるため、単純な正解率で比較すると、役に立たない「常に健康」規則（99%）が人命を救う検査（95%）よりも上位に来てしまう。

陽性クラスに特化して性能を評価するため、2 つの比率を組にして用いる。すなわち、**感度**（sensitivity）や**真陽性率**（true-positive rate）とも呼ばれる**再現率**（実際の陽性のうち検出できた割合）と、**陽性的中率**（positive predictive value）とも呼ばれる**適合率**（陽性判定のうち本当に陽性である割合）である。

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{3.19}$$

1 列目を縦に読むと recall $= 900/1{,}000 = 0.90$ となる（「常に健康」では 0）。1 行目を横に読むと precision $= 900/5{,}850 \approx 0.15$ となる。陽性と判定された人のうち実際に病気なのはわずか 15% にすぎないが、これは大多数を占める健康な人々からの 5% の誤報（4,950 人）が 900 人の真陽性を上回るためである（1.6.5.2 節で見たベイズの定理の効果）。2 段階の臨床スクリーニングにおいては、再現率 90%、適合率 15% という特性は理想的である。安価な一次検査で病気の患者のほぼ全員を拾い上げ、陽性判定となった 5,850 人だけを確定診断のための精密検査へ回せばよいからである。

この検査固有の再現率（0.90）と誤報率（0.05）を固定したまま、疾患の有病率を $\{0.1\%, 1\%, 10\%, 50\%\}$ と変化させてみよう。

```
recall, false_alarms = 0.9, 0.05  # Properties of the test
for prevalence in [0.001, 0.01, 0.1, 0.5]:
    tp, fp = recall * prevalence, false_alarms * (1 - prevalence)
    accuracy = tp + (1 - false_alarms) * (1 - prevalence)
    print(f'{prevalence:5.1%} sick: accuracy {accuracy:.3f}, '
          f'recall {recall:.2f}, precision {tp / (tp + fp):.3f}')
```

```
 0.1% sick: accuracy 0.950, recall 0.90, precision 0.018
 1.0% sick: accuracy 0.950, recall 0.90, precision 0.154
10.0% sick: accuracy 0.945, recall 0.90, precision 0.667
50.0% sick: accuracy 0.925, recall 0.90, precision 0.947
```

何が変化し、何が不変であるかに注目しよう。再現率と誤報率は**分類器のみ**の性質であるのに対し、適合率は**集団の有病率**に大きく依存する。

3.4.1.2 非対称なコスト

病気の患者を見逃すコストを $\kappa_{\text{FN}} = 100$、誤報のコストを $\kappa_{\text{FP}} = 1$ とする（正しい判定のコストは 0 とする）。これらのペナルティで重み付けすると、「常に健康」規則では 1 人あたりの期待コストが $100 \times 1{,}000/100{,}000 = 1.00$ かかるのに対し、先ほどのスクリーニング検査では 1 人あたり $(100 \times 100 + 1 \times 4{,}950)/100{,}000 \approx 0.15$ となり、ほぼ 7 分の 1 に抑えられる。

より一般に、分類器がある個人に対して確率 $p = P(\text{sick} \mid \mathbf{x})$ を推定するとき、**健康**と予測する場合の期待コストは $p\,\kappa_{\text{FN}}$、**病気**と予測する場合の期待コストは $(1-p)\,\kappa_{\text{FP}}$ である。したがって、次の条件を満たすときに**病気**と予測すれば期待コストが最小になる。

$$p\,\kappa_{\text{FN}} > (1-p)\,\kappa_{\text{FP}}, \quad \text{that is,} \quad p > \frac{\kappa_{\text{FP}}}{\kappa_{\text{FP}} + \kappa_{\text{FN}}}. \tag{3.20}$$

コストが等しい場合（$\kappa_{\text{FN}} = \kappa_{\text{FP}}$）には、おなじみの閾値 $1/2$（argmax 分類）に戻る。しかし、$\kappa_{\text{FN}} = 100$、$\kappa_{\text{FP}} = 1$ の場合、最適な閾値は $1/101 \approx 0.01$ まで下がり、予測リスクがわずか 2% であっても患者を陽性と判定すべきことになる。閾値を下げると適合率を犠牲にして再現率が向上する（コストが明示されていない場合、実務ではこのトレードオフを適合率と再現率の調和平均である **F1 スコア** $= 2\,\text{precision}\cdot\text{recall}/(\text{precision}+\text{recall})$ で要約するか、ROC 曲線や適合率・再現率曲線によってすべての閾値を走査することが多い。演習問題 4 および 5 を参照）。

3.4.2 較正

3.4.2.1 信頼性図

Equation 3.20 のようなコスト考慮型の決定規則は p の数値に直接作用するため、分類器が**較正**されていることを前提としている。すなわち、分類器が確信度（confidence）$\hat{p} \approx 0.80$ と予測したすべての例のうち、実際に約 80% が正解でなければならない。

q クラス問題において最大クラスの較正を確認するには、各検証予測の**確信度** $\hat{p}^{(i)} = \max_j \hat{y}_j^{(i)}$ と正誤の指示変数 $c^{(i)} = \mathbf{1}(\operatorname{argmax}_j \hat{y}_j^{(i)} = y^{(i)})$ を記録し、データ例を確信度ビン $[0, 0.1), [0.1, 0.2), \ldots, [0.9, 1.0]$ に分割して、各ビンの平均確信度 $\bar{p}_b$ と経験正解率（empirical accuracy）a_b を比較する。a_b を $\bar{p}_b$ に対してプロットしたものが**信頼性図**であり（完全に較正されたモデルは対角線 $a_b = \bar{p}_b$ 上に並ぶ）、両者の差をサンプル数で重み付け平均したものが**期待較正誤差**（ECE）として定義される。

$$\text{ECE} = \sum_b \frac{n_b}{n} \left| a_b - \bar{p}_b \right|. \tag{3.21}$$

簡潔な実装の `SoftmaxRegression` モデルを Fashion-MNIST で訓練し、検証セットのロジット `O` とラベル `y` を収集して、ビンごとの信頼性統計量、ECE、および信頼性図を計算してみよう。

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
```

```
O, y = [], []
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
print(f'validation accuracy {(O.argmax(axis=1) == y).mean():.3f}')
```

```
validation accuracy 0.823
```

各予測の確信度を計算するには、数値的にシフトしたソフトマックス（3.3 節）を NumPy でロジットに適用し、行ごとの最大確率をとる。

```
def softmax(O):
    """Row-wise softmax of a NumPy array of logits."""
    E = np.exp(O - O.max(axis=1, keepdims=True))
    return E / E.sum(axis=1, keepdims=True)

conf = softmax(O).max(axis=1)  # Confidence of each prediction
correct = (O.argmax(axis=1) == y).astype(float)
```

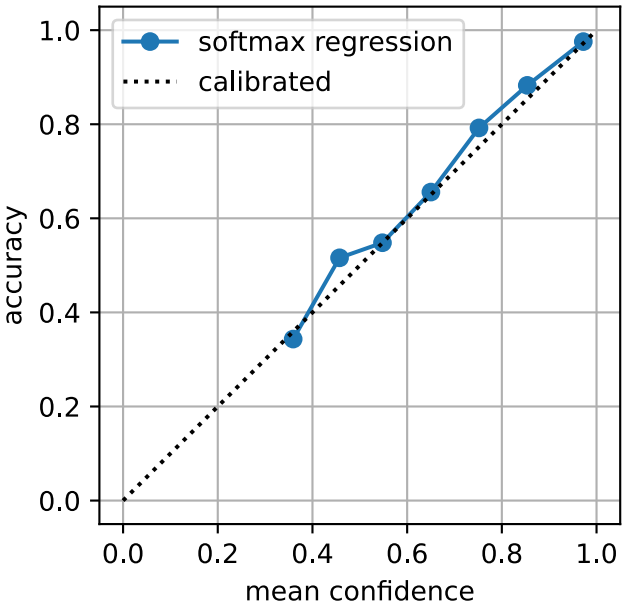
次に、`reliability(conf, correct, num_bins=10)` は `np.bincount` を用いて検証予測を等幅の 10 個の確信度ビンにグループ化し、ビンごとのサンプル数 `n`、平均確信度 `mean_conf`、経験正解率 `acc`、およびサンプル数で重み付けした `ece` を返す。

```
def reliability(conf, correct, num_bins=10):
    """Count, mean confidence, and accuracy per confidence bin; the ECE."""
    bins = np.minimum((conf * num_bins).astype(int), num_bins - 1)
    n = np.bincount(bins, minlength=num_bins)
    mean_conf = np.bincount(bins, conf, num_bins) / np.maximum(n, 1)
    acc = np.bincount(bins, correct, num_bins) / np.maximum(n, 1)
    return n, mean_conf, acc, np.sum(n * np.abs(acc - mean_conf)) / len(conf)

n, mean_conf, acc, ece = reliability(conf, correct)
for b in np.flatnonzero(n):
    print(f'confidence {b / 10:.1f} to {(b + 1) / 10:.1f}: {n[b]:4d} '
          f'predictions, mean {mean_conf[b]:.3f}, accuracy {acc[b]:.3f}')
print(f'ECE: {ece:.3f}')
```

```
confidence 0.1 to 0.2:    2 predictions, mean 0.178, accuracy 0.000
confidence 0.2 to 0.3:   39 predictions, mean 0.266, accuracy 0.231
confidence 0.3 to 0.4:  268 predictions, mean 0.359, accuracy 0.343
confidence 0.4 to 0.5:  655 predictions, mean 0.457, accuracy 0.516
confidence 0.5 to 0.6:  956 predictions, mean 0.548, accuracy 0.548
confidence 0.6 to 0.7:  953 predictions, mean 0.650, accuracy 0.656
confidence 0.7 to 0.8: 1025 predictions, mean 0.752, accuracy 0.792
confidence 0.8 to 0.9: 1372 predictions, mean 0.854, accuracy 0.883
confidence 0.9 to 1.0: 4730 predictions, mean 0.972, accuracy 0.976
ECE: 0.015
```

少数サンプルのばらつきで曲線が歪まないよう、少なくとも 100 個の予測を含む十分なサンプル数のビンに絞り込み（`used = n >= 100`）、`acc[used]` を `mean_conf[used]` に対して理想的な対角線 $y = x$ とともにプロットする。

```
used = n >= 100  # Fewer predictions give too noisy an accuracy
d2l.plot([mean_conf[used], [0, 1]], [acc[used], [0, 1]], 'mean confidence',
         'accuracy', legend=['softmax regression', 'calibrated'],
         fmts=('o-', 'k:'), figsize=(3.5, 3.5))
```



十分なサンプル数をもつすべてのビン（$n_b \geq 100$）にわたり、経験正解率は平均確信度と 8 パーセントポイント以内で追随しており、対角線よりわずかに**上方**に位置している（平均確信度 $\sim 81\%$ に対して正解率 $\sim 83\%$ であり、ECE は 0.03 未満）。深刻な過信（overconfidence）に陥りやすい過剰パラメータ化された深層ネットワークとは異なり、Fashion-MNIST 上の浅いソフトマックス回帰はわずかに過小確信（underconfidence）の傾向にあるものの、すでに良好に較正されている。

3.4.2.2 温度スケーリング

信頼性図によって較正のずれが明らかになった場合、**温度スケーリング** (Guo et al. 2017) は 1 パラメータによる事後的な補正手段となる。すなわち、訓練済みの重みを固定したまま、ロジットをスカラー温度 $T > 0$（3.1 節）で割り、ホールドアウト検証データ上で交差エントロピーを最小化するように T を当てはめる。$T > 0$ で割ってもロジットの順序は保たれるため、温度スケーリングによってクラス予測が 1 つでも変わったり正解率が変化したりすることは決してなく、確信度のスケールのみが調整される（$T > 1$ は過信を抑え、$T < 1$ は過小確信を補正して分布を鋭くする）。Fashion-MNIST の検証セットの前半を使って T を当てはめ、後半で ECE を確認してみよう。

```
def nll(O, y, T):
    """Mean cross-entropy of softmax(O / T), computed from the logits."""
    Z = O / T
    m = Z.max(axis=1)
    lse = m + np.log(np.exp(Z - m[:, None]).sum(axis=1))
```

```
    return np.mean(lse - Z[np.arange(len(y)), y])

half = len(y) // 2
Ts = np.linspace(0.5, 2, 151)
T = Ts[np.argmin([nll(O[:half], y[:half], t) for t in Ts])]
ece = lambda Z: reliability(softmax(Z).max(axis=1), correct[half:])[3]
print(f'T = {T:.2f}; ECE on the second half: {ece(O[half:]):.3f} before, '
      f'{ece(O[half:] / T):.3f} after scaling')
```

```
T = 0.93; ECE on the second half: 0.015 before, 0.012 after scaling
```

わずかに過小確信の傾向にあった信頼性曲線と整合的に、当てはめた温度は 1 をわずかに下回り（$T \approx 0.9$）、もともと非常に小さかった ECE は実質的に変化しない。

3.4.3 まとめ

単純な正解率が適切な評価指標となるのは、クラスが均衡しており、すべての誤りのコストが等しい場合に限られる。陽性例がまれな場合には真陰性が全体の正解率を支配するため、陽性検出を**再現率**（$\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$）、分類器固有の性質）と**適合率**（$\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$、ベイズの定理により集団の有病率に応じて変化する）に切り分けて評価する。偽陰性と偽陽性が異なるコスト κ_{FN} と κ_{FP} を伴う場合、$p > \kappa_{\mathrm{FP}}/(\kappa_{\mathrm{FP}}+\kappa_{\mathrm{FN}})$ のときに陽性と予測することで期待コストが最小化されるが、この規則は予測確率が較正されていることを前提としている。信頼性図と期待較正誤差はその一致度を検証するものであり、ホールドアウトデータ上で単一の事後的な温度 $T > 0$ を当てはめることで、正解率を変えずに確信度を再スケーリングできる。

3.4.4 演習問題

1. **最適な意思決定と棄却**。多クラス問題において、κ_{ij} をクラス i と予測し真のクラスが j であったときのコストとし、$p(j \mid x)$ をクラス j の確率（入力 x が与えられた場合）とする。
   1. クラス i と予測する場合の期待コストを各コストおよび $p(j \mid x)$ を用いて表し、最適な i を選ぶ規則を述べよ。また、2 クラスの場合に Equation 3.20 に帰着することを確認せよ。
   2. 正しい推測のコストを 0、誤った推測のコストを λ_s とし、分類器が推測する代わりに固定コスト $\lambda_r < \lambda_s$ で**棄権**できるようにする。最大クラスの確率がどの閾値を超えたときに推測が棄権よりも有利になるかを導出せよ。
   3. 医療診断では、誤診に比べて専門医への引き継ぎは低コストであり、たとえば $\lambda_r/\lambda_s = 0.05$ であるとする。このときの閾値を計算し、確信度 0.9 の予測に対して分類器がどう振る舞うべきかを述べよ。さらに、推測や棄権のいずれよりも、分布 $p(y \mid x)$ 全体を提示する方が医師にとって有益となる場合を説明せよ。

   Kevin Murphy 著『Probabilistic Machine Learning: An Introduction』演習問題 5.1 を改変。

2. **クラス重み付き損失**。先に 3.4.1 節で見たスクリーニングの例では、病気の患者の見逃しは誤報よりもはるかに大きなコストを伴う。各クラス j に重み $w_j > 0$ を与え、モデル f を次のクラス重み付き平均損失で訓練する。

$$\frac{1}{n}\sum_{i=1}^{n} w_{y^{(i)}}\, l\big(f(\mathbf{x}^{(i)}), y^{(i)}\big), \tag{3.22}$$

ここで、l は交差エントロピー損失、$y^{(i)}$ はデータ例 i のクラスであり、クラス j の各データ例は w_j 回分として数えられる。

1. 訓練例がクラスの事前分布 $p(y)$ とクラス条件付き密度 $p(\mathbf{x} \mid y)$ に従って抽出されるとする。クラス重み付き損失の期待値は、同じクラス条件付き密度をもちクラスの事前分布が $w_y\, p(y)$ に比例する分布のもとでの重みなし損失の期待値に比例することを示せ。また、損失ではなく訓練データをどのように変更すれば同じ効果が得られるか答えよ。
2. 入力 $\mathbf{x}$（クラス確率は $p(j \mid \mathbf{x})$）を固定し、モデルが任意の分布 $\hat{\mathbf{y}}$ を出力できると仮定する。重み付き損失の期待値

$$-\sum_j w_j\, p(j \mid \mathbf{x}) \log \hat{y}_j \tag{3.23}$$

が $\hat{y}_j \propto w_j\, p(j \mid \mathbf{x})$ によって最小化されることを示せ。
3. 健康と病気の 2 クラスについて、$p(\text{sick} \mid \mathbf{x})$ の閾値であって、それを超えると $\arg\max$ をこの $\hat{\mathbf{y}}$ にとった結果が病気となるものを求め、その閾値が Equation 3.20 の閾値と等しくなるように重みを選べ。さらに、$w_{\text{sick}} > w_{\text{healthy}}$ と選ぶとなぜ再現率が上がり、偽陽性の数にどのような影響が生じるかを説明せよ。

3. **較正と有用性**。ある 10 クラス問題では、どのクラスも同じ数のデータ例をもつとする。

1. ある分類器は入力を無視してすべてのクラスに確率 0.1 を出力し、同点の場合はランダムにクラスを選ぶ。この分類器が較正されていることを示し、その正解率を計算せよ。
2. 別の分類器はデータ例の 95% で正解し、常に確信度 1 を出力する。この分類器の信頼性図を描き、期待較正誤差を計算せよ。
3. 較正と正解率がそれぞれ異なる性質を測っている理由、および意思決定に閾値 Equation 3.20 を用いる際にどちらも他方の代わりにはならない理由を説明せよ。

4. [code] **ROC 曲線**。陽性クラスに対してスコア $s \in (0,1)$ を出力する二値分類器は、任意の $\tau \in [0,1]$（$\frac{1}{2}$ に限らない）を閾値にでき、$s > \tau$ のときに陽性と予測する。**受信者動作特性**（ROC）曲線は、真陽性率（再現率）を偽陽性率 $\text{FP}/(\text{FP}+\text{TN})$ に対して、τ を 1 から 0 まで動かしながらプロットしたものである。1000 個の陽性例と 1000 個の陰性例をスコア $s = \sigma(z)$ でシミュレーションせよ。ただし、$\sigma(z) = 1/(1+\exp(-z))$ はシグモイドであり、陽性例では $z \sim \mathcal{N}(\mu, 1)$、陰性例では $z \sim \mathcal{N}(0,1)$ とする。

1. $\mu \in \{0, 1, 2\}$ に対して ROC 曲線を計算してプロットし、端点 $\tau = 1$ と $\tau = 0$ がプロット上のどの点を与えるかを述べよ。
2. スコアがラベルと独立である分類器は、期待値において対角線を描くことを示せ。
3. $\mu = 2$ に対して、適合率を再現率に対して τ を動かしながらプロットした**適合率・再現率曲線**を、陽性例と陰性例が同数の場合と、陽性例が全体の 1% を占める場合（陰性例 1000 個を保ち陽性例 10 個を抽出するか、重み付けし直す）のそれぞれについて描け。陽性率が変わっても ROC 曲線が変化しない一方で適合率・再現率曲線が低下する理由、および両者のうちどちらが本節のスクリーニング検査をより適切に表しているかを説明せよ。

5. [code] **ランキング確率としての AUC**。●この問題では、問題 4 の ROC 曲線とシミュレーションしたスコアを用いる。

1. どの 2 つのスコアも同点にならないとき、ROC 曲線の下側の面積（AUC）は、ランダムに選んだ陽性例がランダムに選んだ陰性例よりも高いスコアを得る確率に等しく、したがって閾値に依存しないランキング品質の要約指標となることを示せ。

2. シミュレーションしたスコアについてこの確率を閉じた式で求め、各曲線の下側の台形近似による面積と比較せよ。
3. 訓練済みのソフトマックス回帰に対し、2 つのクラス a と b の検証画像上で、a を陽性クラス、スコアを 2 つのロジットの差のシグモイド $s = \sigma(o_a - o_b)$ として ROC 曲線と AUC を適用せよ。対象はスニーカー対サンダル、およびシャツ対 T シャツの 2 組とする。2 つの AUC の大小関係を、同じペアに対する混同行列 **C**（3.3 節）の成分と比較し、これら 2 つの診断指標が必ずしも一致しない理由を説明せよ。

6. [code] **温度スケーリング**。本文では、検証画像の前半を使って T を当てはめた。
   1. スケーリング前後の後半データに対する信頼性図を描き、正解率が変化しないことを示せ。
   2. 代わりに訓練画像上で T を当てはめ、本文の T と比較せよ。モデルの訓練に使っていないデータで当てはめるべき理由を説明せよ。
   3. スケーリング前のモデルの期待較正誤差を 5、10、20 個のビンでそれぞれ計算せよ。その値はビン分割にどの程度依存するか。

3.5 分類における汎化

Fashion-MNIST の分類器は、約 83% の正解率を 10,000 枚の検証画像に対して達成した。将来の衣類画像に対する性能保証として、この数値をどこまで信頼できるだろうか。そこには 2 つの根本的な障害が立ちはだかっている。

1. **有限標本の不確実性とテストセットの再利用：**未使用のテストセットであっても、$n = 10{,}000$ 枚の画像は有限個の無作為標本（random sample）にすぎない。さらに悪いことに、実際の開発ではモデルの比較やハイパーパラメータの調整を進めるなかで、同じ評価分割を繰り返し参照してしまう。
2. **暗記と学習：**すべての訓練画像が互いに異なる高次元空間では、ルックアップテーブル型の暗記器（memorizer）は 100% の訓練正解率を達成しながら、未知の画像に対してはランダムに推測するだけである。暗記器の訓練誤差がゼロであっても何も保証されないのに、**線形**分類器の低い訓練誤差が低い母集団誤差を保証するのはなぜだろうか。

2.6 節では回帰における過学習と検証分割を定性的に導入したが、本節では分類における汎化（generalization）を定量的に扱う。具体的には、サイズ n のテストセットに対する厳密な信頼区間（confidence interval）と漸近的な信頼区間を導出し、テストセットの再利用のもとで選択バイアス（selection bias）と適応的過学習（adaptive overfitting）がいかにスコアを水増しするかを定量化したうえで、統計学習理論から **Vapnik–Chervonenkis（VC）次元**と**一様収束**（uniform convergence）を導入する。

3.5.1 テストセット

3.5.1.1 経験誤差と母集団誤差

単一の分類器 f は、独立なテストセット $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ を母集団分布 $P(X, Y)$ から独立同分布（i.i.d.）に抽出する**前に**固定しておく。このとき、$\mathcal{D}$ 上での**経験誤差**（empirical error）と真の**母集団誤差**（population error）は次式で定義される。

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\left(f(\mathbf{x}^{(i)}) \neq y^{(i)}\right), \qquad \epsilon(f) = P\left(f(X) \neq Y\right). \tag{3.24}$$

f は事前に固定されているため、各誤り指示変数 $\mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)})$ は独立な Bernoulli$(\epsilon(f))$ のコイン投げとなり、その平均は $\epsilon(f)$、分散は $\epsilon(f)(1-\epsilon(f)) \leq 1/4$ である。したがって、テスト誤差 $\epsilon_{\mathcal{D}}(f)$ は不偏推定量（unbiased estimator）であり（$\mathbb{E}[\epsilon_{\mathcal{D}}(f)] = \epsilon(f)$）、その標準誤差（standard error）は次式となる。

$$\mathrm{SE}(\epsilon_{\mathcal{D}}(f)) = \sqrt{\frac{\epsilon(f)(1-\epsilon(f))}{n}} \leq \frac{0.5}{\sqrt{n}}. \tag{3.25}$$

Fashion-MNIST の分類器（$\epsilon(f) \approx 0.17$）を $n = 10{,}000$ 枚のテスト画像で評価した場合、標準誤差は $\sqrt{0.17 \times 0.83/10{,}000} \approx 0.0038$（0.38 パーセントポイント）となる。ここで $1/\sqrt{n}$ のスケーリング則（scaling law）に注目しよう。測定の不確実性を半分にするには、テスト例が **4 倍**必要になる。

3.5.1.2 信頼区間と Hoeffding の不等式

標準誤差を 95% 信頼区間 $\epsilon_{\mathcal{D}}(f) \pm r$ に変換するには、2 つの手法がある。

1. **漸近正規区間（中心極限定理、central limit theorem）：**n が大きいとき、$\epsilon_{\mathcal{D}}(f)$ は近似的に正規分布に従い、標準誤差の 1.96 倍以内の幅で $\epsilon(f)$ の周りに確率 ≈ 0.95 で収まる。

$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \leq 1.96\sqrt{\frac{\epsilon(f)(1-\epsilon(f))}{n}}\right) \approx 0.95. \tag{3.26}$$

   $\epsilon \approx 0.17$、$n = 10{,}000$ のとき、95% 正規信頼半径は $1.96 \times 0.0038 \approx 0.007$（$\pm 0.7$ パーセントポイント）である。最悪の場合（$\epsilon = 1/2$）でも、95% 信頼半径 ≤ 0.01 を保証するには $n \geq (1.96 \times 0.5/0.01)^2 = 9{,}604$ 個の例があればよい。これは MNIST、Fashion-MNIST、CIFAR-10 のテストセットが 10,000 例で構成されている理由を理解するうえで有益な見方であり、いずれも正解率を約 1 ポイントの精度で特定できる。

2. **有限標本 Hoeffding 区間：**$\epsilon(f)$ が 0 に非常に近い場合、正規近似では被覆率が著しく低下し得るため（小さなテストセットでは誤分類がゼロとなってプラグイン正規半径が 0 に潰れることがある）、**Hoeffding の不等式** (Hoeffding 1963) は各誤りが $[0,1]$ の範囲にあるという事実のみを用いて、あらゆる n とあらゆる分布に対して成り立つ厳密な上界を与える（36.9 節）。

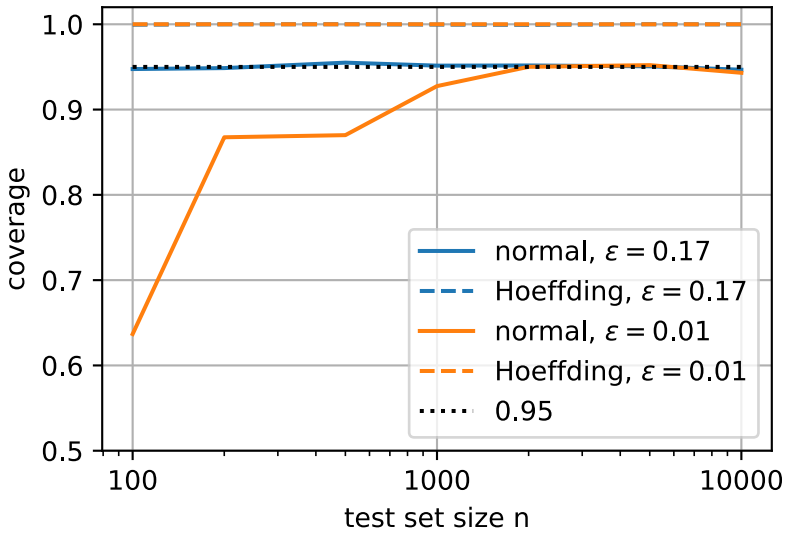
$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \geq t\right) \leq 2\exp\left(-2nt^2\right). \tag{3.27}$$

   右辺を $\delta = 0.05$ とおくと、95% Hoeffding 信頼半径 $r = \sqrt{\log(2/\delta)/(2n)}$ が得られる（これには $n \geq \log(40)/(2 \times 0.01^2) \approx 18{,}445$ 個の例が $r \leq 0.01$ の達成に必要となり、正規近似による必要数の約 2 倍にあたる。どちらも最悪ケースの分散 $1/4$ を仮定しているが、Hoeffding の指数型裾評価はすべての n で成り立つぶん、正規分布の厳密な分位点（quantile）より緩いためである）。

10,000 個の独立なテストセットをサイズ $n \in [100, 10{,}000]$ にわたって 2 つの分類器（$\epsilon = 0.17$ と $\epsilon = 0.01$）についてシミュレーションし、両方の 95% 信頼区間の経験被覆率（empirical coverage）をプロットしてみよう。

```
rng = np.random.default_rng()
num_sets = 10000
ns = np.array([100, 200, 500, 1000, 2000, 5000, 10000])
curves, legend = [], []
for eps in [0.17, 0.01]:
```

```
    # The number of errors on a test set of size n is Binomial(n, eps)
    errs = [rng.binomial(n, eps, num_sets) / n for n in ns]
    normal = [1.96 * np.sqrt(e * (1 - e) / n) for e, n in zip(errs, ns)]
    hoeffding = [np.sqrt(np.log(2 / 0.05) / (2 * n)) for n in ns]
    for name, radii in [('normal', normal), ('Hoeffding', hoeffding)]:
        curves.append([np.mean(np.abs(e - eps) <= r)  # coverage
                       for e, r in zip(errs, radii)])
        legend.append(rf'{name}, $\epsilon = {eps}$')
d2l.plot(ns, curves + [[0.95] * len(ns)], 'test set size n', 'coverage',
         legend=legend + ['0.95'], xscale='log', ylim=(0.5, 1.02),
         fmts=('C0-', 'C0--', 'C1-', 'C1--', 'k:'), figsize=(4.5, 3))
```



$\epsilon = 0.17$ では、プラグイン正規区間はほぼ 95% の被覆率をすべての n で維持している。一方、$\epsilon = 0.01$ で n が小さいときには、誤差ゼロのテストセットが生じるたびに正規区間の被覆率は $\sim 63\%$ まで落ち込むのに対し、Hoeffding の保守的な区間はあらゆる領域で 95% の被覆率を上回っている。

3.5.2 テストセットの再利用

3.5.2.1 多数の分類器からの選択

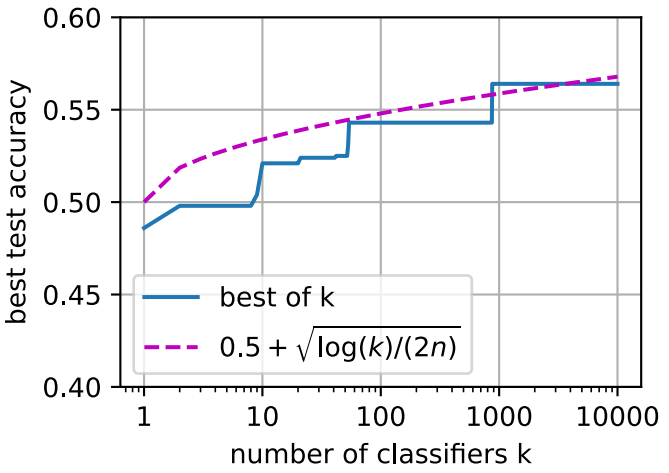
k 個の固定された候補分類器 $f_1, \ldots, f_k$ を**同じ**テストセット $\mathcal{D}$ 上で評価し、最も成績の良かったものを報告すると何が起きるだろうか。たとえ k 個の候補すべてが $\mathcal{D}$ の抽出前に固定されていたとしても、最高スコアを選ぶことは、$\mathcal{D}$ 上でたまたま最も運が良かった候補をつまみ食い（チェリーピッキング）することになる。**ユニオンバウンド**（k 個の悪い事象のうち少なくとも 1 つが起こる確率は、個々の確率の和で上から抑えられるという性質）を Equation 3.27 に適用すると、次式が得られる。

$$P\left(\max_{j=1,\ldots,k} |\epsilon_{\mathcal{D}}(f_j) - \epsilon(f_j)| \geq t\right) \leq 2k \exp\left(-2nt^2\right). \tag{3.28}$$

右辺を δ とおくと、信頼度 $1-\delta$ の精度を k 個すべてのモデルに対して同時に保証するには、信

頼半径が $\sqrt{\log(2k/\delta)/(2n)}$ まで広がることがわかる。この穏やかな $\sqrt{\log k}$ の増大に注目しよう。$k = 1{,}000$ 個の固定モデルを比較しても半径は対数因子ぶんしか大きくならないが、それでも $n = 1{,}000$ 個のコイン投げで決めたテストラベルに対し、純粋にランダムなコイン投げで推測する $k = 10{,}000$ 個の予測器（いずれも真の正解率は 0.50）を評価すると、最も運の良かった予測器のテスト正解率は $0.5 + \sqrt{\log(k)/(2n)}$ に沿って着実に上昇し、~ 0.56 に達する。

```
n, k = 1000, 10000
labels = rng.random(n) < 0.5            # one test set with coin-flip labels
guesses = rng.random((k, n)) < 0.5      # k classifiers that guess at random
accuracy = (guesses == labels).mean(axis=1)
ks = np.arange(1, k + 1)
d2l.plot(ks, [np.maximum.accumulate(accuracy),
              0.5 + np.sqrt(np.log(ks) / (2 * n))],
         'number of classifiers k', 'best test accuracy',
         legend=['best of k', r'$0.5 + \sqrt{\log(k) / (2n)}$'], xscale='log',
         ylim=(0.4, 0.6))
```



3.5.2.2 適応的過学習

過去のテストスコアに基づいて新たな候補を**適応的に**選ぶようになると、選択バイアスははるかに危険なものとなる (Dwork et al. 2015)。テスト結果のフィードバックがモデル設計を導くようになると、分類器はもはや $\mathcal{D}$ と独立ではなくなり、ユニオンバウンド Equation 3.28 は完全に破綻する。Blum and Hardt (2015) に倣い、先ほどのランダムなコイン投げ予測器のうち最初の 1,000 個だけを取り出し、テストセット上でたまたま 0.50 を超えるスコアを出したものだけを選んで、各テスト例に対して多数決をとるとどうなるか見てみよう。

```
kept = guesses[:1000][accuracy[:1000] > 0.5]  # guessers that scored above 0.5
vote = kept.mean(axis=0) > 0.5                # their majority vote
print(f'majority vote: {(vote == labels).mean():.3f}, '
      f'best single guesser: {accuracy[:1000].max():.3f}')
```

```
majority vote: 0.709, best single guesser: 0.564
```

個々のテストラベルを一度も直接見ることなく、わずか 1,000 回のスカラーの正解率クエリだけ

で、純粋なコイン投げ予測器の多数決はテスト正解率で $\sim 70\%$ に跳ね上がる一方、未知のデータに対する母集団正解率は依然として 50% のままである（実際、$O(n)$ 回のクエリがあればすべてのテストラベルを復元できる。演習問題 3 を参照）。だからこそ、ハイパーパラメータ最適化とアーキテクチャ探索は検証分割上で行い（演習問題 4）、テストセットは一度限りの最終評価のために残しておかなければならない。

10 年にわたる再利用は、標準的なベンチマークにどれほどの悪影響を与えてきたのだろうか。Recht et al. (2019) が CIFAR-10 と ImageNet の新しいテストセットを収集したところ、正解率は 3 から 15 ポイント低下したが、モデル間のランキングはほとんど変わらなかった。上述のような最悪の事態は起こり得るものの、通常のモデル開発におけるテストセットへの過学習は、理論が許容する最悪ケースよりはるかに小さかったのである。正解率の低下そのものは、データ分布（data distribution）のわずかな変化によってより適切に説明される。これについては 3.6 節で扱う。

3.5.3 統計学習理論

3.5.3.1 一様収束

分類器 $f_{\mathcal{S}}$ を仮説クラス（hypothesis class）$\mathcal{F}$ の中から、訓練誤差 $\epsilon_{\mathcal{S}}(f)$（訓練セット $\mathcal{S}$、サイズ n 上）の最小化によって選んだとき、それが母集団へ汎化することをどうすれば**事前に**（a priori）保証できるだろうか。学習器による $f_{\mathcal{S}} \in \mathcal{F}$ の選択は $\mathcal{S}$ を調べたあとに行われるため、訓練誤差が母集団誤差の近くにとどまるという性質が**すべての** $f \in \mathcal{F}$ **に対して同時に**成り立つ必要がある（**一様収束**）。$\mathcal{F}$ が有限（$|\mathcal{F}| < \infty$）であれば、ユニオンバウンド Equation 3.28 を全 $k = |\mathcal{F}|$ 個の仮説に対して $\mathcal{S}$ 上で適用することで、少なくとも確率 $1 - \delta$ で次式が成り立つ。

$$|\epsilon_{\mathcal{S}}(f) - \epsilon(f)| \le \sqrt{\frac{\log(2|\mathcal{F}|/\delta)}{2n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.29}$$

3.5.3.2 VC 次元と VC 上界

線形半空間 $f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^\top \mathbf{x} + b)$（$\mathbb{R}^d$ 上）のような連続パラメータ族では $|\mathcal{F}| = \infty$ となるため、Equation 3.29 は自明（vacuous）となり意味をなさない。しかし Vapnik and Chervonenkis (1971) は、任意の有限な n 個の点の集合上では、まったく同じ ± 1 の二値ラベル付けを生成する 2 つの分類器は $\mathcal{S}$ 上で区別できないことに着目した。$\mathcal{F}$ が m 個の点の集合を**粉砕する**（shatter）とは、$\mathcal{F}$ 内の関数によって全 2^m 通りの可能な二値ラベル付けをその m 点上で実現できることをいう。そして **Vapnik–Chervonenkis（VC）次元** $v = \mathrm{VC}(\mathcal{F})$ とは、$\mathcal{F}$ が粉砕できる最大の点集合のサイズである。

- **平面（$\mathbb{R}^2$）上の直線の VC 次元は 3（Figure 3.6）：**同一直線上にない任意の 3 点は直線によって全 $2^3 = 8$ 通りに粉砕できるのに対し、平面 $\mathbb{R}^2$ 上のいかなる 4 点も決して粉砕できない（凸四角形の向かい合う頂点同士を分離するには対角線を交差させる必要があり、また三角形の 3 頂点を含む半平面はその内部の任意の点も自動的に含んでしまうためである）。
- **$\mathbb{R}^d$ 内の線形半空間の VC 次元は $d + 1$（演習問題 5）：**線形半空間は原点と d 個の標準基底ベクトル（basis vector）を粉砕でき、Radon の定理により $d + 2$ 個の点（$\mathbb{R}^d$ 内）は決して粉砕できない。
- **制約のないルックアップテーブル型の暗記器は $v = \infty$：**互いに異なる点のあらゆる有限集合を粉砕できる。

Sauer の補題により、$n \ge v$ となると、VC 次元 v のクラスが n 個の点上に誘導できる異なるラベル付けの総数は多項式オーダーでしか増大せず、高々 $(en/v)^v$ で抑えられる（2^n にはならない）。これらのラベル付けに対して対称化したユニオンバウンドを適用すると（36.10 節）、**VC 上**

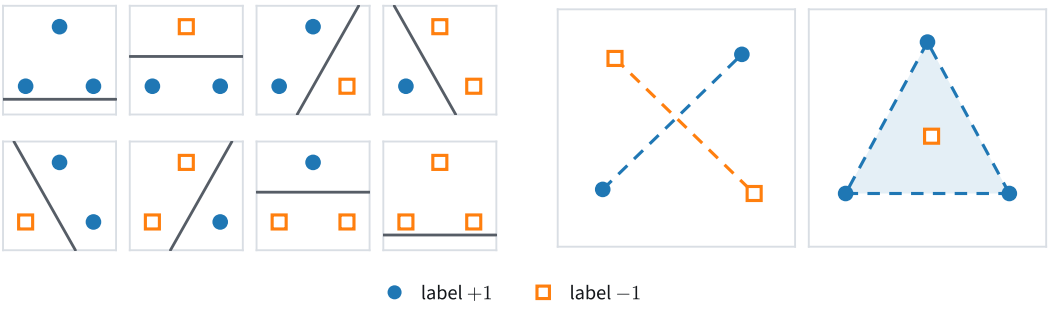


Figure 3.6: $\mathbb{R}^2$ 上の直線は同一直線上にない 3 点を粉砕できるが（左、全 $2^3 = 8$ 通りのラベル付け）、いかなる 4 点も粉砕できない（中央および右）。

界（VC bound）が得られる。すなわち、ある絶対定数 c に対し、確率 $1-\delta$ 以上（訓練セット $\mathcal{S}$ の抽出に関して）で次式が成り立つ。

$$\epsilon(f) \le \epsilon_{\mathcal{S}}(f) + c\sqrt{\frac{v + \log(1/\delta)}{n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.30}$$

この定理は冒頭の概念的な問いに答えてくれる。$\mathbb{R}^d$ 内の線形分類器は有限の VC 次元 $v = d + 1$ を持つため、$n \gg d$ のときに訓練誤差を小さくすれば、**いかなる**分布 P に対しても低い母集団誤差が保証されるのに対し、VC 次元が無限大の暗記器にはそのような上界が一切存在しない。同時に、最悪ケースの一様収束上界は敵対的な分布に対しても成り立たなければならないため、Equation 3.30 は自然画像に対しては数値的に緩くなり（1 対他の単一の半空間でも $\sqrt{785/60{,}000} \approx 0.11$ にさらに定数 c が掛かり、全 10 クラスを合わせた容量はさらに大きくなるのに対し、Fashion-MNIST での実測の訓練・テスト間ギャップは 0.01 である）、ランダムなラベルさえ暗記できる過剰パラメータ化された深層ニューラルネットワークに対しては自明（$v \ge n$）になってしまう (Zhang et al. 2021)。この謎については 4.5 節で改めて立ち返ることとし、当面は鋭い経験的測定のためにホールドアウトされたテストセットを頼りにする。

3.5.4 まとめ

n 個の例からなる未使用のテストセット上では、固定された分類器の経験誤差 $\epsilon_{\mathcal{D}}(f)$ の標準誤差は $\sqrt{\epsilon(1-\epsilon)/n} \le 0.5/\sqrt{n}$ となるため、正解率を 1 パーセントポイント以内の精度に信頼度 95% で特定するには、正規近似のもとで約 10,000 個、Hoeffding の有限標本上界のもとでは約 18,500 個のテスト例が必要になる。同じテストセットを再利用して k 個の固定モデルから選択すると、この不確実性は $\sqrt{\log(2k/\delta)/(2n)}$ に従って緩やかに広がる一方、過去のテストスコアに導かれる適応的な設計ははるかに少ないクエリ数でテストセットに過学習し得る。訓練誤差のみから汎化を上から抑えるため、統計学習理論では仮説クラスの容量を VC 次元 v（そのクラスが粉砕できる最大の点集合のサイズであり、線形半空間では $v = d + 1$、ただし入力は $\mathbb{R}^d$）によって測り、一様な $O(\sqrt{v/n})$ の上界を導く。これにより、線形モデルが大規模なデータセットを暗記できない理由が説明される一方で、過剰パラメータ化された深層ネットワークに対してはこの上界は自明なままにとどまる。

3.5.5 演習問題

1. **2 つのモデルと 1 つのテストセット**。ある論文で、モデル A は 91.2%、モデル B は 91.6% の正解率を 10,000 枚の画像のテストセット上で達成したと報告されている。
   1. 各正解率の標準誤差と、正規近似による 95% 信頼半径を計算せよ。もし 0.4 ポイントの差が、2 つのモデルをこのサイズの独立な 2 つのテストセットで評価して得られたものだったとしたら、意味のある差といえるか。
   2. 実際には、両方のモデルが同じ 10,000 枚の画像上で評価されていた。この場合、問題の本質が 2 つのモデルの予測が食い違う画像にある理由を説明し、もし論文がすべてのテスト画像についてどちらのモデルが正解したかを報告していたなら、どのように判定するか述べよ。
   3. 0.4 ポイントの差（独立に評価された 2 つのモデル間の差）が信頼半径の外に出るには、どれほど大きなテストセットが必要か。
2. **テストセット上での誤差ゼロ**。ある分類器が n 個の例からなるテストセット上で誤りを一切出さなかったとする。
   1. 正規近似において $\epsilon_{\mathcal{D}}(f)$ を $\epsilon(f)$ の代わりに代入したときに報告される区間と、Hoeffding 区間を示せ。
   2. 母集団誤差 ϵ を持つ分類器が n 個の独立なテスト例で誤りを出さない確率が $(1-\epsilon)^n \le e^{-n\epsilon}$ であることを示せ。さらに、$\epsilon \ge 3/n$ ならば誤りのないテストセットが生じる確率は 0.05 未満になることを導け（「3 の法則」）。
   3. $3/n$ と Hoeffding 信頼半径を $n = 100$ および $n = 10{,}000$ の場合について比較し、その違いを説明せよ。
3. **テストセットの漏洩**。第三者がテストセット $\mathcal{D}$（二値ラベル付きの n 個の例）を保持しており、入力のみを公開しているとする。モデルクラスに制限はなく任意のモデル f を提出でき、そのテスト誤差 $\epsilon_{\mathcal{D}}(f)$ のみを受け取れるとする。
   1. すべてのテストラベルを復元し、したがって母集団誤差がどうあれテスト誤差 0 を達成するモデルを得る戦略を与え、その戦略が使用する提出回数を求めよ。
   2. 各応答 $\epsilon_{\mathcal{D}}(f)$ は $n+1$ 通りの値のいずれかをとる。これを用いて、いかなる戦略であっても全 2^n 通りのラベル付けを識別するために必要な提出回数の下界を求めよ。
4. [code] **Fashion-MNIST の検証分割**。本章では 10,000 枚のテスト画像上でモデルを選択した。代わりに 10,000 枚を 60,000 枚の訓練画像から検証分割として取り置き、残りの 50,000 枚でソフトマックス回帰を訓練せよ。
   1. 学習率を $\{0.01, 0.03, 0.1, 0.3, 1\}$ の中から検証分割上の正解率によって選択せよ。
   2. 選択したモデルをテスト画像上で一度だけ評価し、テスト正解率を検証正解率と比較せよ。その差は 95% 信頼区間の半径（10,000 例に対するもの）以内に収まっているか。
   3. 選択後に一度だけ測定したテスト正解率は選択されたモデルに対する不偏推定値であるのに対し、最良の検証正解率はそうではない理由を説明せよ。
5. **線形分類器の VC 次元**。●線形分類器 $f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^\top \mathbf{x} + b)$（$\mathbb{R}^d$ 上）を考える。
   1. これらが $d+1$ 個の点 $\mathbf{0}, \mathbf{e}_1, \dots, \mathbf{e}_d$ を粉砕することを示せ。任意のラベル $y_0, y_1, \dots, y_d \in \{\pm 1\}$ が与えられたとき、それらを生成する b と $\mathbf{w}$ を構成せよ。ヒント：$f(\mathbf{0})$ は b のみに依存するため、まず b を選ぶとよい。
   2. $\mathbf{x}_1, \dots, \mathbf{x}_{d+2} \in \mathbb{R}^d$ とする。$d+2$ 個のベクトル $[\mathbf{x}_i, 1] \in \mathbb{R}^{d+1}$（各 $\mathbf{x}_i$ の末尾に 1 を付加したもの）が線形従属であり、したがって $\sum_i a_i [\mathbf{x}_i, 1] = \mathbf{0}$ がある係数 a_i（すべてがゼロとは限らない）に対して成り立つことを示せ。添字集合 I を $a_i > 0$ となる添字全体、J を $a_i < 0$ となる添字全体とする。どちらも空集合ではなく、いかなる線形分類器も I に属する点を $+1$、J に属する点を -1 とラベル付けできないことを示せ。ヒント：$\sum_i a_i (\mathbf{w}^\top \mathbf{x}_i + b)$ を 2 通りの方法で計算するとよい。
   3. 以上から VC 次元が $d+1$ であることを結論づけ、さらに $\{\mathbf{x}_i : i \in I\}$ の凸包と $\{\mathbf{x}_j : j \in J\}$ の凸包が交わることを示せ。

6. **多項式の VC 次元**。分類器 $f(x) = \text{sign}(p(x))$（ただし p は $x \in \mathbb{R}$ に関する高々 5 次の多項式）からなるクラスの VC 次元はいくつか。また、$\mathbf{x} \in \mathbb{R}^d$ の場合はいくつか。ヒント：このような分類器は $\mathbf{x}$ の単項式上の線形分類器である。さらに、Fashion-MNIST 上の 5 次多項式分類器の訓練誤差に対して VC 上界が何を予測するか、そしてなぜ誰もそのような分類器を訓練しないのかを述べよ。

3.6 環境と分布シフト

第 3.5 章におけるすべての保証は、ある暗黙の仮定の上に成り立っていた。それは、運用時のデータが訓練セットやテストセットとまったく同じ分布から抽出されるという仮定である。現実の世界では、この仮定は絶えず破られる。スキャナやカメラはアップグレードされ、季節性インフルエンザの流行は疾患の有病率を一晩で変え、地域の俗語は変化していく。さらに厄介なことに、モデルを運用すること**それ自体**が集団の行動を変化させる場合さえある。たとえば、ある銀行が過去の記録から、オックスフォードシューズを履いた融資申込者はスニーカーの申込者よりも返済率が高いことを発見し、オックスフォードシューズを承認してスニーカーを却下する分類器を導入したとしよう。数週間もすれば、申込者は全員オックスフォードシューズを履いて現れるようになるが、根本的な債務不履行率は少しも変わらない。これは **Goodhart の法則**（統計的な相関指標が意思決定の目標になると、もはや有効な相関指標ではなくなること）の典型例である。

本節では、**ソース分布**（source distribution）である $p_S(\mathbf{x}, y)$（ラベル付き訓練データが得られる）と、**ターゲット分布**（target distribution）である $p_T(\mathbf{x}, y)$（ラベルなしの運用時入力 $\mathbf{x}$ のみが観測される）の間の分布シフト（distribution shift）を検出し、補正する方法を学ぶ。同時分布（joint distribution）のどの因子が不変に保たれるかによって受動的なシフトの代表的な 3 つの類型を整理し、**重要度重み付き共変量シフト補正**（importance-weighted covariate shift correction）と**混同行列によるラベルシフト補正**（confusion-matrix label shift correction）を導出して実装したうえで、最後に概念シフト（concept shift）とフィードバックループを扱う。

3.6.1 分布シフトの種類

$p_S(\mathbf{x}, y)$ と $p_T(\mathbf{x}, y)$ を結びつける**何らかの**構造的仮定がなければ、ラベルなしのターゲットドメインへ適応することは数学的に不可能である。もし $p_T(\mathbf{x}) = p_S(\mathbf{x})$ が同一のまま、猫と犬のラベルが一晩のうちに密かに入れ替わったとしても（$p_T(y = \text{cat} \mid \mathbf{x}) = p_S(y = \text{dog} \mid \mathbf{x})$）、ラベルなしのターゲット画像しか見ないアルゴリズムがその入れ替わりを検出することは決してできない。

あらゆる同時分布は、次の 2 通りに分解できる。

$$p(\mathbf{x}, y) = p(\mathbf{x})\, p(y \mid \mathbf{x}) = p(y)\, p(\mathbf{x} \mid y). \tag{3.31}$$

1 つの条件付き因子を不変に保ちながら周辺分布がドリフトする場合を考えることで、分布シフトの古典的な 3 つの形態が定義される（Figure 3.7）。

シフトの形態	変化するもの	不変に保たれるもの	自然な因果構造
共変量シフト	$p(\mathbf{x})$	$p(y \mid \mathbf{x})$	特徴量 $\mathbf{x}$ がラベル y の原因
ラベルシフト	$p(y)$	$p(\mathbf{x} \mid y)$	ラベル y が特徴量 $\mathbf{x}$ の原因
概念シフト	$p(y \mid \mathbf{x})$	多くの場合 $p(\mathbf{x})$	ラベルの定義が時間・空間とともにドリフト

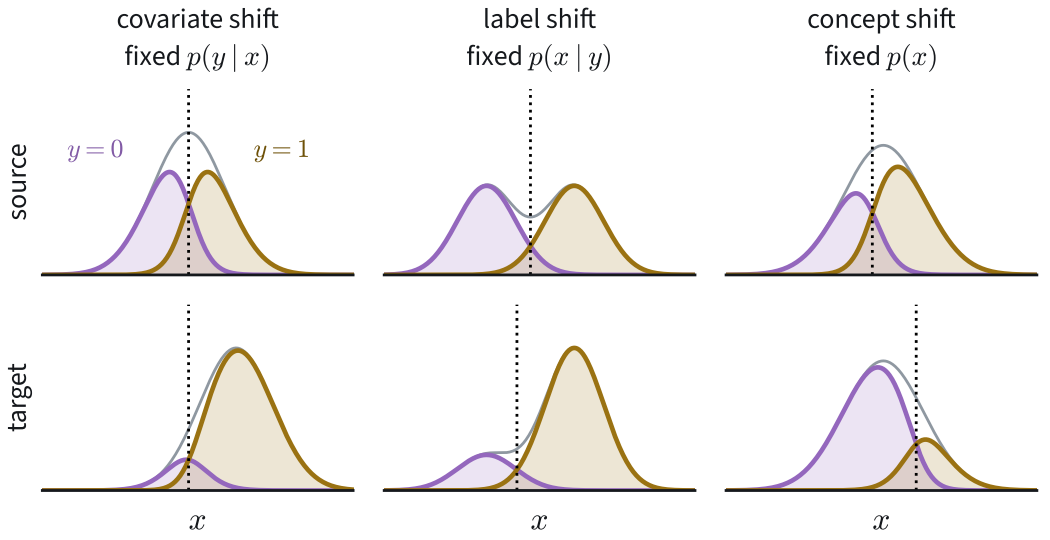


Figure 3.7: 1 次元入力 x と 2 クラスに対する分布シフトの 3 つの形態（上段：ソース p_S、下段：ターゲット p_T、点線：最適決定境界）。共変量シフトは $p(x)$ を移動させるが $p(y \mid x)$ と真の境界は固定されたままである。ラベルシフトはクラス事前分布 $p(y)$ を変化させる一方で $p(x \mid y)$ を固定したままにする（ベイズ境界は希少なクラスの側へ移動する）。概念シフトは $p(y \mid x)$ を直接変化させる。

3.6.1.1 共変量シフト、ラベルシフト、概念シフト

1. **共変量シフト（$p(y \mid \mathbf{x})$ が不変）：** $\mathbf{x}$ が y の原因となるときに自然に生じる。たとえば、猫と犬の分類器を実写写真（Figure 3.8）で訓練し、イラスト（Figure 3.9）に対して運用する場合を考えよう。イラストの猫も依然として猫であるが（$p(y \mid \mathbf{x})$ は不変）、周辺入力分布 $p(\mathbf{x})$ はピクセル空間の新たな領域へとシフトしている。真のベイズ境界は動いていなくても、ソースの実写写真に当てはめた誤特定（misspecified）されたモデルや表現力の限られたモデルは、$p_S(\mathbf{x})$ の密度が高い領域に正解率を集中させるため、$p_T(\mathbf{x})$ が分布する領域では大きく破綻し得る。
2. **ラベルシフト（$p(\mathbf{x} \mid y)$ が不変）：** y が $\mathbf{x}$ の原因となるときに自然に生じる。たとえば疾患は症状を引き起こすため、インフルエンザの流行期にはインフルエンザの有病率 $p(y)$ が急増する一方、インフルエンザ患者の症状分布 $p(\mathbf{x} \mid y)$ は変化しない。ベイズの定理により $p(y \mid \mathbf{x}) \propto p(\mathbf{x} \mid y)\,p(y)$ が成り立つため、クラスの事前分布（prior）がシフトすると最適な決定境界（decision boundary）も移動する。ラベルシフトの補正で推定が必要なのは q 個のクラス事前分布比だけであり、$\mathbf{x}$ 上の高次元の密度比（density ratio）を推定する必要はない。
3. **概念シフト（$p(y \mid \mathbf{x})$ が変化）：** 米国の郡ごとの炭酸飲料の地域的な呼び名（Figure 3.10）や、時代とともに改定される診断基準、季節ごとの購買意図など、地域や時間によってラベルそのものの意味が変化する。

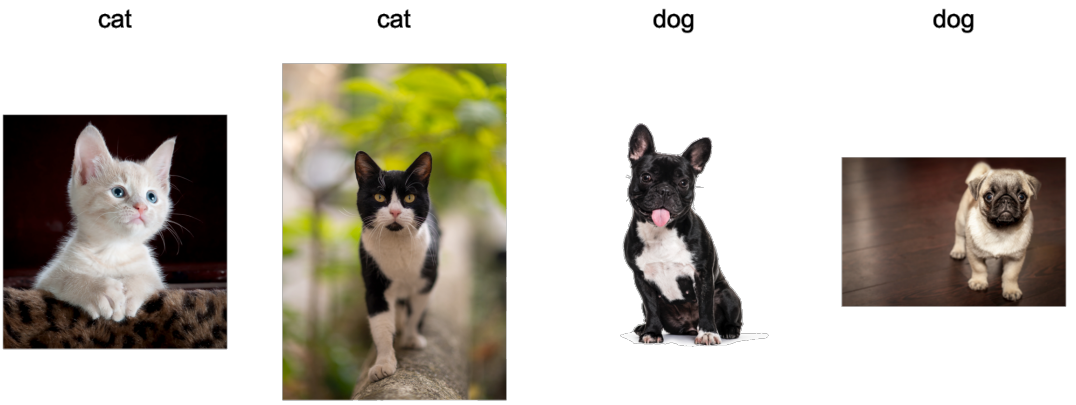


Figure 3.8: 猫と犬の分類におけるソース訓練画像：実写写真（画像提供：Lafeez Hossain / 500px / Getty Images、ilkermetinkursova / iStock / Getty Images Plus、GlobalP / iStock / Getty Images Plus、Musthafa Aboobakuru / 500px / Getty Images）。

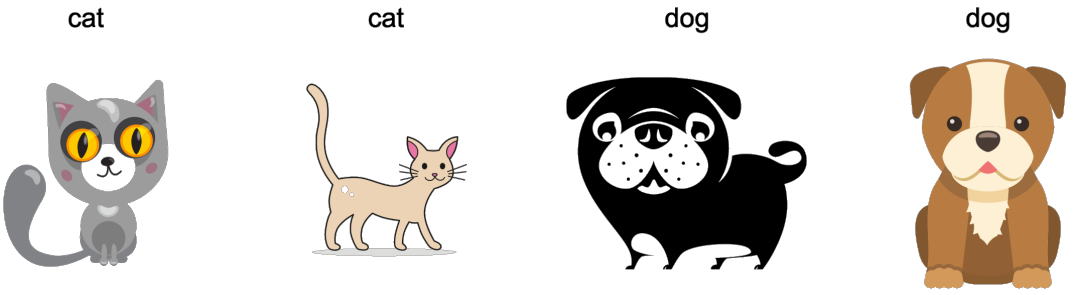


Figure 3.9: 猫と犬の分類におけるターゲット運用画像：イラスト（画像提供：SIBAS_minich / iStock / Getty Images Plus、Ghrzuzudu / iStock / Getty Images Plus、id-work / DigitalVision Vectors / Getty Images、Yime / iStock / Getty Images Plus）。

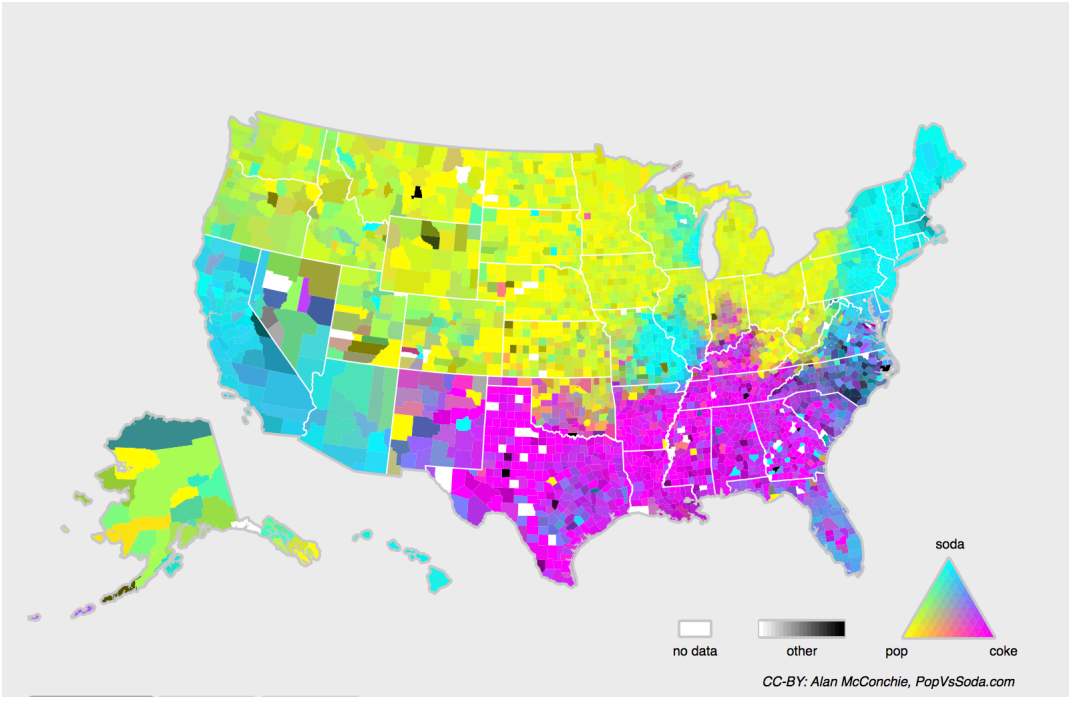


Figure 3.10: 地理的な概念シフト：米国の郡ごとに加糖炭酸飲料を指す最も一般的な呼び名（「pop」「soda」「coke」）（CC-BY: Alan McConchie, PopVsSoda.com）。

3.6.2 共変量シフトの補正

3.6.2.1 重要度重み

運用時に最小化したいのは、**ターゲットリスク** $R_T[f] = \mathbb{E}_{(\mathbf{x},y)\sim p_T}[l(f(\mathbf{x}), y)]$ である。共変量シフト（$p_T(y \mid \mathbf{x}) = p_S(y \mid \mathbf{x}) = p(y \mid \mathbf{x})$）のもとでは、ソースのサポートがターゲットを覆っている限り（すなわち $p_S(\mathbf{x}) > 0$ が $p_T(\mathbf{x}) > 0$ なるすべての点で成り立つとき）、$p_S(\mathbf{x})$ を掛けて割ることにより、ターゲットリスクを**ソース**分布に関する期待値へと書き換えられる (Shimodaira 2000)。

$$
\begin{aligned}
R_T[f] &= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_T(\mathbf{x})\, d\mathbf{x} \\
&= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_S(\mathbf{x})\, \frac{p_T(\mathbf{x})}{p_S(\mathbf{x})}\, d\mathbf{x} \\
&= \mathbb{E}_{(\mathbf{x},y)\sim p_S}\left[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)\right],
\end{aligned}
\tag{3.32}
$$

ここで、$\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ は入力 $\mathbf{x}$ の**重要度重み**（importance weight）である。Equation 3.32 を n 個のラベル付きソース例で近似すると、次の**重み付き経験リスク最小化**（weighted empirical risk minimization）が得られる。

$$\begin{aligned} &\underset{f}{\text{minimize}}\ \frac{1}{n}\sum_{i=1}^{n}\beta_i\, l(f(\mathbf{x}^{(i)}), y^{(i)}), \\ &\text{where } \beta_i = \frac{p_T(\mathbf{x}^{(i)})}{p_S(\mathbf{x}^{(i)})}. \end{aligned} \tag{3.33}$$

サポートの重なり（support overlap、すなわち $p_S(\mathbf{x}) > 0$ が $p_T(\mathbf{x}) > 0$ なるすべての点で成り立つこと）が譲れない条件である理由に注目しよう。もし医療系スタートアップが、疾患のある血液サンプルを高齢患者から集め、健常な対照サンプルを 20 歳の大学生のみから集めたとすると、どれほど再重み付けしても $p_S(\mathbf{x}) = 0$ である領域に高齢の健常対照データを作り出すことは決してできない。たとえサポートが重なっていても、右裾で密度比が極端に大きくなると（Figure 3.11）、ごく少数の点が和を支配して**有効標本サイズ**（effective sample size）$n_{\text{eff}} = (\sum_i \beta_i)^2 / \sum_i \beta_i^2$ が激減してしまう。重みを上限で打ち切る $\min(\beta_i, c)$ のクリッピングを用いれば、わずかな統計的バイアスと引き換えに n_{eff} を大幅に増やすことができる。

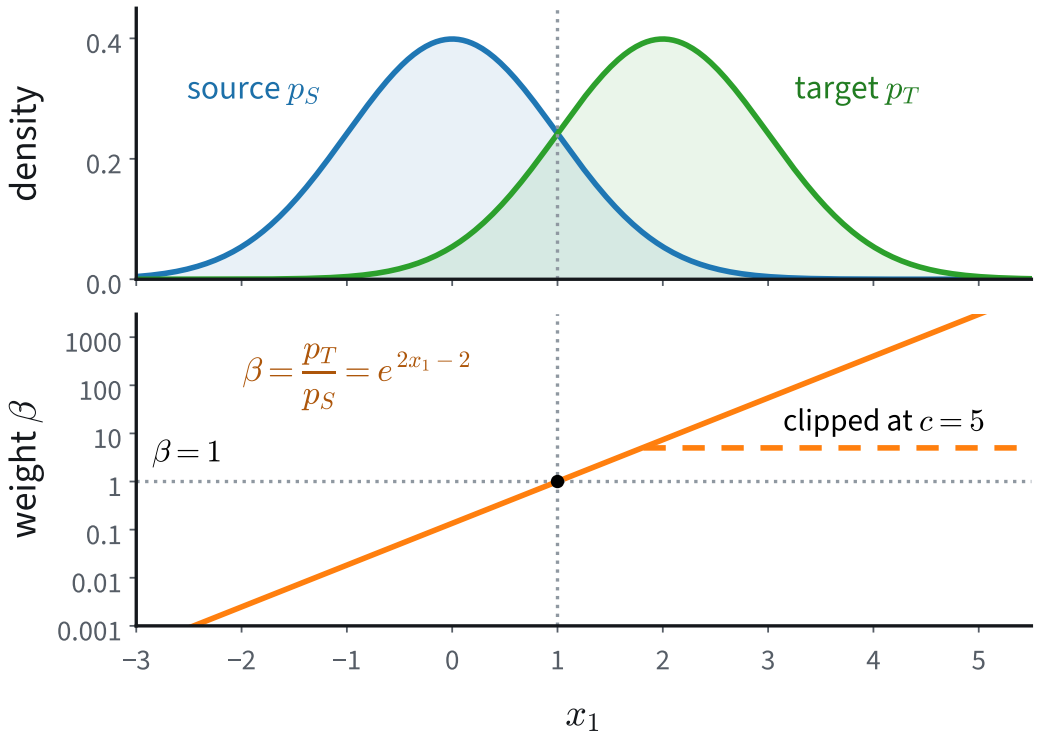


Figure 3.11: ソース密度 $p_S = \mathcal{N}(0, 1)$ とシフトしたターゲット密度 $p_T = \mathcal{N}(2, 1)$（上段）、および結果として得られる重要度重み $\beta(x_1) = p_T(x_1)/p_S(x_1) = e^{2x_1 - 2}$ をクリッピング閾値 $c = 5$ とともに対数スケールで示したもの（下段）。

3.6.2.2 ドメイン分類器による重みの推定

$\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ を推定するのに、高次元の密度 p_S と p_T を別々に推定せずに済ませるにはどうすればよいだろうか。n 個のソース入力（ドメインラベル $z = 0$ を付与）と m 個のラベルなしターゲット入力（ドメインラベル $z = 1$ を付与）を統合し、二値ロジスティック回帰の**ド**

メイン分類器（domain classifier）$P(z=1 \mid \mathbf{x}) = \sigma(h(\mathbf{x}))$（ロジットは $h(\mathbf{x})$）を訓練すればよい。ベイズの定理により、次式が成り立つ。

$$\frac{P(z=1 \mid \mathbf{x})}{P(z=0 \mid \mathbf{x})} = \frac{m}{n}\,\frac{p_T(\mathbf{x})}{p_S(\mathbf{x})} = \exp(h(\mathbf{x})),$$
$$\text{so} \quad \beta(\mathbf{x}) = \frac{n}{m}\exp(h(\mathbf{x})).$$

定数因子 n/m（目的関数を定数倍するだけで最小点を動かさない）を除けば、各ソース例を単に $\beta_i = \exp(h(\mathbf{x}^{(i)}))$（あるいは $\min(\exp(h(\mathbf{x}^{(i)})), c)$）で重み付けすればよい。

ドメイン分類器には 2 つ目の用途がある。補正すべきシフトが存在するかどうかを判定できることである。もし $p_T(\mathbf{x}) = p_S(\mathbf{x})$ であれば、ソース入力とターゲット入力を区別する手がかりは何もなく、訓練に使っていない統合データに対するドメイン分類器の性能は偶然レベルにとどまる（正解率は $1/2$、ただし $n = m$ のとき）。したがって、偶然レベルを明確に上回る正解率は、入力がシフトしたことの証拠となる。この検査は**分類器二標本検定**（classifier two-sample test）として知られている (Lopez-Paz and Oquab 2017)。ラベルシフトにはさらに手軽な検査法がある。運用中の分類器をラベルなしのターゲット入力に適用し、各クラスを予測する頻度をソース検証データ上での頻度と比較すればよい。シフトはまず気づかなければ補正できないため、運用システムではこれら両方の数値を監視すべきである。

3.6.2.3 2 次元の例

$p_S = \mathcal{N}(\mathbf{0}, \mathbf{I}_2)$、$p_T = \mathcal{N}((2,0)^\top, \mathbf{I}_2)$ であり、真のラベルが $y = \mathbf{1}(x_2 > \frac{1}{2}x_1^2 - 1)$ で与えられる 2 次元問題において、ドメイン分類器による再重み付けが線形分類器をどのように救うかを見てみよう。真の放物線状の境界は上向きに湾曲しているため、$(0,0)$ 付近で最良の線形境界はほぼ水平となり、$(2,0)$ 付近では惨憺たる結果に終わる。そこで $n = 1{,}000$ 個のソース点と $1{,}000$ 個のターゲット点を生成し、ロジスティックドメイン分類器 $h(\mathbf{x})$ を当てはめて $\beta_i = \exp(h(\mathbf{x}^{(i)}))$ を推定したうえで、重みなし、重要度重み付き、およびクリッピング付き（$c = 5$）のロジスティック回帰を比較する。

```
rng = np.random.default_rng()
n = 1000
X_src = rng.normal(0, 1, (n, 2))            # source p_S: centered at (0, 0)
X_tgt = rng.normal(0, 1, (n, 2)) + [2, 0]   # target p_T: centered at (2, 0)
label = lambda X: (X[:, 1] > 0.5 * X[:, 0]**2 - 1).astype(float)
y_src, y_tgt = label(X_src), label(X_tgt)   # one labeling rule for both
```

ここで用いる学習の基本要素は、勾配降下法で訓練する重み付きロジスティック回帰ただ 1 つである。3.1.2.2 節より、ロジットに関する交差エントロピーの勾配は残差（residual）$\sigma(\mathbf{w}^\top\mathbf{x}+b)-y$ となるため、データ例 i を β_i で重み付けすることは、単にその残差に β_i を掛けることに帰着する。

```
def fit_logreg(X, y, weights=None, lr=0.1, steps=2000):
    """Logistic regression by gradient descent on the weighted mean loss."""
    w, b = np.zeros(X.shape[1]), 0.0
    v = np.ones(len(y)) if weights is None else weights / weights.mean()
    for _ in range(steps):
        r = v * (1 / (1 + np.exp(-(X @ w + b))) - y)   # weighted error
        w -= lr * X.T @ r / len(y)
```

```
        b -= lr * r.mean()
    return w, b
```

まず、X_src（ドメインラベル $z = 0$）と X_tgt（$z = 1$）を統合してドメイン分類器 (w_h, b_h) を fit_logreg により訓練し、学習されたロジット $h(\mathbf{x})$ を解析的なガウス対数密度比 $2x_1 - 2$ と比較したうえで、得られたソースの重要度重み beta とその有効標本サイズ ess を確認する。最後の数行では、新たなソース入力とターゲット入力をサンプリングし、ドメイン分類器がどの程度の割合で両者を識別できるかを測定する。

```
z = np.concatenate([np.zeros(n), np.ones(n)])      # domain label: 1 = target
w_h, b_h = fit_logreg(np.concatenate([X_src, X_tgt]), z)
beta = np.exp(X_src @ w_h + b_h)                   # weights of source examples
ess = lambda v: v.sum()**2 / (v**2).sum()          # effective sample size
print(f'learned h(x) = {w_h[0]:.2f} x1 {w_h[1]:+.2f} x2 {b_h:+.2f}'
      f'   (true log-ratio: 2 x1 - 2)')
print(f'largest weight: {beta.max():.0f}; effective sample size: '
      f'{ess(beta):.0f} of {n}, {ess(np.minimum(beta, 5)):.0f} if clipped at 5')
X_out = np.concatenate([rng.normal(0, 1, (n, 2)),          # fresh source and
                        rng.normal(0, 1, (n, 2)) + [2, 0]])  # target inputs
print(f'domain classifier accuracy on held-out inputs: '
      f'{((X_out @ w_h + b_h > 0) == (z == 1)).mean():.2f} (chance: 0.50)')
```

```
learned h(x) = 1.98 x1 +0.09 x2 -1.98   (true log-ratio: 2 x1 - 2)
largest weight: 193; effective sample size: 23 of 1000, 221 if clipped at 5
domain classifier accuracy on held-out inputs: 0.84 (chance: 0.50)
```

新たな入力に対して、ドメイン分類器は約 0.84 の正解率でソースとターゲットを識別しており、偶然レベルの 0.50 を大きく上回るため、このシフトの検出は容易である。この設定では、これ以上高い性能を達成できる分類器は存在しない。2 つのガウス分布は x_1 の平均のみが異なり、最良の判定規則は $x_1 > 1$ であって、その正答確率は $\Phi(1) \approx 0.84$（ただし Φ は標準正規分布関数）となるからである。

次に、fit_logreg をラベル付きソースデータ (X_src, y_src) に対して 3 通りの重み付け方式（重みなし 'unweighted'、生の重要度重み 'weighted'、クリッピング重み 'clipped'（上限 5））で訓練し、各分類器の X_tgt および X_src 上での正解率を評価する。

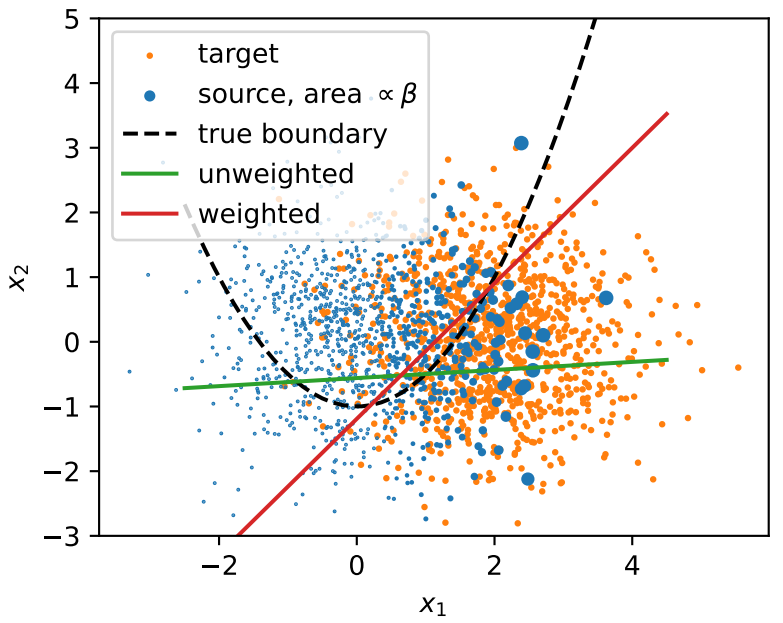
```
acc = lambda wb, X, y: ((X @ wb[0] + wb[1] > 0) == (y == 1)).mean()
models = {name: fit_logreg(X_src, y_src, v) for name, v in [
    ('unweighted', None), ('weighted', beta), ('clipped', np.minimum(beta, 5))]}
for name, wb in models.items():
    print(f'{name:10s}  target accuracy {acc(wb, X_tgt, y_tgt):.3f}'
          f'   source accuracy {acc(wb, X_src, y_src):.3f}')
print(f'always predicting the most common target class: '
      f'{max(y_tgt.mean(), 1 - y_tgt.mean()):.3f}')
```

```
unweighted  target accuracy 0.603   source accuracy 0.853
weighted    target accuracy 0.945   source accuracy 0.854
clipped     target accuracy 0.944   source accuracy 0.854
always predicting the most common target class: 0.711
```

再重み付けがこれほど劇的に効く理由を可視化するため、ターゲットとソースの点群（各ソース点のマーカー面積を beta に比例させる）を、真の放物線 $x_2 = \frac{1}{2}x_1^2 - 1$ および 'unweighted' と

'weighted' の線形決定境界とともにプロットする。

```
x1 = np.linspace(-2.5, 4.5, 50)
d2l.set_figsize((4.5, 3.5))
d2l.plt.scatter(*X_tgt.T, s=2, c='C1', label='target')
d2l.plt.scatter(*X_src.T, s=np.clip(beta, 0.2, 20), c='C0',
                label=r'source, area $\propto \beta$')
d2l.plt.plot(x1, 0.5 * x1**2 - 1, 'k--', label='true boundary')
for name, fmt in [('unweighted', 'C2'), ('weighted', 'C3')]:
    (w1, w2), b = models[name]
    d2l.plt.plot(x1, -(w1 * x1 + b) / w2, fmt, label=name)
d2l.plt.xlabel('$x_1$'), d2l.plt.ylabel('$x_2$'), d2l.plt.ylim(-3, 5)
d2l.plt.legend();
```



結果を見てみよう。シフトしたターゲット点群に対して、重みなしの訓練ではスコアが ～ 50%～60% にとどまる（多数派クラスを常に予測する場合の ～ 70% より低い）のに対し、重要度による再重み付けは線形決定境界を放物線の右側の枝に沿うように傾け、ターゲットラベル**ゼロ**でターゲット正解率を ～ 95% まで引き上げている。

この結果から、共変量シフトのもとでは常に再重み付けすべきだと結論づけるのは誤りである。ここで再重み付けが有効だったのは、直線が放物線に沿えないからである。すなわち、モデルが誤特定されており、どこで高い精度を出すかを選ばざるを得ないためである。ソースデータが存在するすべての領域で $p(y \mid \mathbf{x})$ に適合できるほど柔軟なモデルは何も得るところがない。その場合、重み付きの目的関数と重みなしの目的関数は同じ最小点を共有するからである (Shimodaira 2000)。深層ネットワークはこの状況に近い。ひとたび低い訓練損失（training loss）まで訓練されると、重要度重みの効果は薄れていき (Byrd and Lipton 2019)、実世界のシフトを集めた WILDS ベンチマークでも、分布シフト向けに設計された手法が通常の経験リスク最小化を上回ることはまれである (Koh et al. 2021)。実務において共変量シフトへの対処は、主により幅広いデータで訓練することと、少数のラベル付きターゲット例でファインチューニングすることによってなされる。

3.6.3 ラベルシフトの補正

ラベルシフト（$p_T(\mathbf{x} \mid y) = p_S(\mathbf{x} \mid y)$ が保たれる一方でクラス比率 $p_T(y) \neq p_S(y)$ がシフトする状況）のもとでは、まったく同じ重要度重み付けの恒等式により、クラス y の各ソース例をクラス事前分布の比 $\beta(y) = p_T(y)/p_S(y)$ で重み付けすることになる。$p_S(y)$ は訓練ラベルから既知であるため、唯一の課題は**ラベルなしの**ターゲットデータからターゲットのクラス比率 $\mathbf{p}_T \in \mathbb{R}^q$ を推定することである。

3.6.3.1 混同行列によるターゲット比率の推定

$\mathbf{C} \in \mathbb{R}^{q \times q}$ を、3.3 節で扱ったソース分類器の列正規化された混同行列とし、その各成分 $C_{ij} = P_S(\hat{c} = i \mid y = j)$ をラベル付きソース検証データから推定する。$\hat{c}$ は $\mathbf{x}$ のみの関数であり、かつラベルシフトのもとでは $p(\mathbf{x} \mid y)$ が不変であるため、ターゲットドメイン上でも $P_T(\hat{c} = i \mid y = j) = C_{ij}$ が成り立つ。全確率の法則により、ラベルなしターゲット入力における予測クラス頻度のベクトル $\mu \in \mathbb{R}^q$ は次式を満たす (Saerens et al. 2002; Lipton et al. 2018)。

$$\mu_i = \sum_{j=1}^{q} P(\hat{c} = i \mid y = j)\, p_T(y = j) = \sum_{j=1}^{q} C_{ij}\, p_T(y = j), \quad \text{or} \quad \mathbf{C}\,\mathbf{p}_T = \mu. \tag{3.34}$$

$\mathbf{C}$ が正則である限り（分類器が各クラスの半分超を正しく分類していれば、すなわち $C_{jj} > 1/2$ であれば保証される。演習問題 3 を参照）、$q \times q$ の連立一次方程式 $\hat{\mathbf{p}}_T = \mathbf{C}^{-1}\mu$ を解くことで、ターゲットラベルを 1 つも使わずに未知のターゲットクラス比率を復元できる。さらに、ソース分類器の出力確率がソースの事後分布（posterior distribution）$p_S(y \mid \mathbf{x})$ をよく近似しているならば（これは 3.4 節における最上位クラスの較正（calibration）よりも強い条件である）、各クラスのロジットに対数事前分布のシフト $\log(\hat{p}_T(y)/p_S(y))$ を加えるだけで、ベイズ則により再訓練なしで直ちに予測をターゲットドメインへ適応させられる。

Fashion-MNIST を用いて、混同行列の逆行列計算とベイズロジット調整を試してみよう。まず、クラス比率の均等な訓練分割で `d2l.SoftmaxRegression` を訓練し、検証データに対するロジット `O`、ラベル `y`、および予測クラス `pred = O.argmax(axis=1)` を収集する。

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
O, y = [], []
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
pred = O.argmax(axis=1)
```

次に、10,000 個の検証例をランダムに分割し、5,000 枚のソース検証セット `src`（列正規化された 10×10 の混同行列 `C` の推定に使用）とターゲットプール `pool` に分ける。このプールから、衣類全体の 50% をシャツが占める偏ったターゲットセット `tgt`（シャツ 400 枚、他の各クラスは 45 枚ずつ）を抽出し、`p_hat = np.linalg.solve(C, mu)` をラベルなしターゲットの予測頻度 `mu` から解く。

```
perm = rng.permutation(len(y))
src, pool = perm[:5000], perm[5000:]  # Source validation set; target pool
C = np.zeros((10, 10))
np.add.at(C, (pred[src], y[src]), 1)  # The confusion matrix of the source
C /= C.sum(axis=0, keepdims=True)     # Each column sums to 1
counts = np.full(10, 45)
counts[6] = 400                       # Half of the target images are shirts
p_T = counts / counts.sum()
tgt = np.concatenate([rng.choice(pool[y[pool] == j], counts[j], replace=False)
                      for j in range(10)])
mu = np.bincount(pred[tgt], minlength=10) / len(tgt)  # Needs no labels
p_hat = np.linalg.solve(C, mu)
for j, name in enumerate(data.text_labels(range(10))):
    print(f'{name:10s} true {p_T[j]:.2f}, predicted {mu[j]:.3f}, '
          f'estimated {p_hat[j]:.3f}')
```

```
T-shirt    true 0.06, predicted 0.133, estimated 0.067
trouser    true 0.06, predicted 0.055, estimated 0.055
pullover   true 0.06, predicted 0.133, estimated 0.091
dress      true 0.06, predicted 0.070, estimated 0.051
coat       true 0.06, predicted 0.082, estimated 0.050
sandal     true 0.06, predicted 0.052, estimated 0.054
shirt      true 0.50, predicted 0.294, estimated 0.468
sneaker    true 0.06, predicted 0.058, estimated 0.055
bag        true 0.06, predicted 0.066, estimated 0.054
ankle boot true 0.06, predicted 0.057, estimated 0.057
```

最後に、対数事前分布のシフト `log_w = np.log(np.maximum(p_hat, 1e-6) / 0.1)` をターゲットのロジット `O[tgt]` に加算することで分類器の予測を推定されたターゲット分布へ適応させ、調整前後のターゲット正解率を比較する。

```
log_w = np.log(np.maximum(p_hat, 1e-6) / 0.1)  # log p_T(y) - log p_S(y)
accuracy = lambda O: (O.argmax(axis=1) == y[tgt]).mean()
print(f'target accuracy: {accuracy(O[tgt]):.3f} as trained, '
      f'{accuracy(O[tgt] + log_w):.3f} adapted to the estimated proportions')
```

```
target accuracy: 0.706 as trained, 0.816 adapted to the estimated proportions
```

元の分類器がターゲット画像のうち「シャツ」と予測するのは $\sim 20\%$〜40% にすぎないにもかかわらず（残りは似た形状のトップスに紛れ込む）、$\mathbf{C}^{-1}\mu$ はその混同を解きほぐして真の値 0.50 に近いシャツ比率を復元し（通常は小数第 2 位の数単位以内、まれに 1 割近くずれる程度）、ロジットに $\log(\hat{\mathbf{p}}_T/0.1)$ を加えることで、ターゲットラベルも勾配ステップも一切要さずにターゲット正解率を約 5〜15 パーセントポイント引き上げる。

3.6.4 概念シフトとフィードバック

概念シフトによって $p(y \mid \mathbf{x})$ そのものが変化すると、過去のラベルは陳腐化し、古いデータにどのような重要度重み付けを施しても新しいラベル付け規則を学習させることはできない。新たにラベル付けされた少量のデータ例による継続的な監視と、定期的なファインチューニングや再訓練が不可欠である。

モデルの予測が**将来の訓練データを作り変える行動**を引き起こす場合（**遂行的予測**（performative prediction）および**フィードバックループ**）には、さらなる注意が必要である。予測的警察活動

（Figure 3.12）において、犯罪予測値の高い地区へ重点的にパトロールを派遣すると、その地区でより多くの事案が記録されるようになる。こうしてバイアスのかかった逮捕記録を次の訓練ラウンドへ戻すと、予測が自己強化されて暴走ループに陥る。このようなループを断ち切ったり監査したりするには、ランダム化された探索（exploration）や独立したデータ収集が必要となり、受動的な教師あり学習から**強化学習**（reinforcement learning）（第 14 章）の領域へと踏み込むことになる。

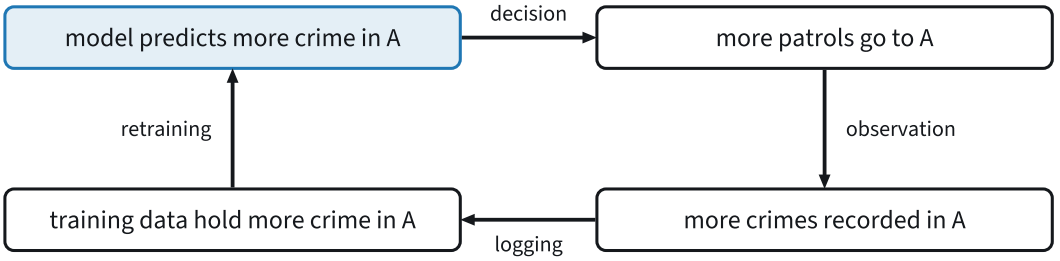


Figure 3.12: 予測的警察活動における暴走フィードバックループ：モデルの予測がパトロール配置を左右し、パトロールの有無が事案の観測場所を決定し、記録された事案が次の訓練データセットとなる。

3.6.5 まとめ

ラベルなしの運用環境への適応は、ソース $p_S(\mathbf{x}, y)$ とターゲット $p_T(\mathbf{x}, y)$ を結ぶ構造的な不変性なしには不可能であり、その不変性は共変量シフト（$p(y \mid \mathbf{x})$ が固定され $p(\mathbf{x})$ がドリフトする）、ラベルシフト（$p(\mathbf{x} \mid y)$ が固定され $p(y)$ がドリフトする）、あるいは概念シフト（$p(y \mid \mathbf{x})$ がドリフトする）として整理される。サポートが重なる共変量シフトのもとでは、ソース入力とラベルなしターゲット入力を識別するロジスティックドメイン分類器で推定した $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x}) \propto \exp(h(\mathbf{x}))$ によってソースの損失を重み付けすることで、誤特定されたモデルをターゲットリスクに整合させられる。ラベルシフトのもとでは、ソースの混同行列の逆行列を用いた $\hat{\mathbf{p}}_T = \mathbf{C}^{-1}\mu$ によりラベルなしターゲット予測からターゲットのクラス比率を復元でき、較正されたロジットに $\log(\hat{\mathbf{p}}_T/\mathbf{p}_S)$ を加えることで直接適応できる。概念シフトや意思決定が駆動するフィードバックループによって環境そのものが変化する場合には、受動的な再重み付けだけでは不十分であり、運用システムには新たなターゲットラベルと独立した探索が必要となる。

3.6.6 演習問題

1. **サポートの欠如。** $p_S(\mathbf{x}) = 0$ が入力のある領域 $\mathcal{A}$（ターゲットのもとで確率 $\pi > 0$ を持つ）上で成り立ち、それ以外のすべての点では $p_S(\mathbf{x}) > 0$ であるとする。$\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ を $\mathcal{A}$ の外側で定義する。
   1. 重み付きソースリスク $\mathbb{E}_{(\mathbf{x},y)\sim p_S}[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)]$ が、ターゲットリスクから $\mathcal{A}$ の寄与を取り除いたものに等しいことを示せ。また、欠落した項を π と条件付き期待値の積として表せ。
   2. ソース例に対する重みをどのように選んでも、この欠落した項を復元できない理由を説明せよ。さらに、$\mathcal{A}$ の外側では一致し、同じ重み付きソースリスクを持ちながらターゲットリスクが異なる 2 つのモデルを構成せよ。
2. [code] **再重み付けが失敗するとき。** 3.6.2 節のコード（`fit_logreg`、`X_src`、`X_tgt`、`label`）ではターゲットの平均を $(2, 0)$ に置いている。ターゲットの平均 $(s, 0)$ を

$s \in \{0, 0.5, 1, 2, 3, 4, 6\}$ にわたって変化させよ。ただし、$n = 1000$ とラベル付け規則は固定したままとする。

1. 各 s について、ホールドアウトされた統合データに対するドメイン分類器 h の正解率を報告し、s の増大に伴うその上昇をシフトの検出しやすさの観点から解釈せよ。
2. 各 s について、重みなし、重み付き、およびクリッピング付き（$c = 5$）の各モデルと、最も頻度の高いターゲットクラスを常に予測する規則のターゲット正解率を記録せよ。また、大きな s においてはターゲット正解率だけでは再重み付けが機能したかどうかを判断できない理由を説明せよ。

3. **ラベルシフトの連立方程式。** $\mathbf{C}$ を Equation 3.34 の列正規化された混同行列とする。

1. $\mathbf{C}$ は、条件 $C_{jj} > \frac{1}{2}$ がすべてのクラス j で成り立つ限り正則であることを示せ。ヒント：$\mathbf{v}^\top \mathbf{C} = \mathbf{0}$ がある $\mathbf{v} \neq \mathbf{0}$ に対して成り立つと仮定し、列 j として $|v_j|$ が最大となるものに注目して、その列の和が 1 であることを用いるとよい。
2. $\mathbf{C}$ が特異となる分類器の例を 1 つ挙げ、その場合でも μ から $\mathbf{p}_T$ について何がわかるかを述べよ。
3. 2 クラスの場合、誤差 δ が陽性予測の割合 μ_1 に生じると、推定される陽性割合が $\delta / (C_{11} - C_{10})$ だけ変化することを示せ。標準誤差 $\sqrt{\mu_1(1-\mu_1)/m}$（m 個のターゲット入力から推定した割合の標準誤差）を用いて、スクリーニングの例において罹患者割合の推定値の標準誤差を 0.01 にするために必要なターゲット入力数を求めよ。同じ集団に対して、より精度の低い検査（$C_{11} = 0.6$、$C_{10} = 0.4$）を用いた場合についても同様に求めよ。

4. [code] **学習された重みと真の重み。**●演習問題 2 におけるターゲット平均 $(s, 0)$ のスイープを再利用せよ。

1. 真の対数密度比 $\log(p_T(\mathbf{x})/p_S(\mathbf{x}))$ を s の関数として導出し、学習された h と各 s において比較せよ。また、大きな s においてドメイン分類器が高い正解率を達成しながら、h が真の対数密度比から大きく乖離し得る理由を説明せよ。
2. ソース分布のもとで真の重みが $\mathbb{E}[\beta(\mathbf{x})] = 1$ および $\mathbb{E}[\beta(\mathbf{x})^2] = e^{s^2}$ を満たし、したがって大きな n に対して有効標本サイズが約 ne^{-s^2} となることを示せ。この値を $n = 1000$、$s = 1, 2, 3$ について計算せよ。
3. 各 s について、実際のソースサンプル上での真の重みと学習された重みの有効標本サイズを計算せよ。$s = 2$ においてサンプルから計算した値が通常 $1000\, e^{-4} \approx 18$ よりはるかに大きくなる理由と、サンプルのみから再重み付け推定の信頼性を判断するうえでそれが何を意味するかを説明せよ。$c = 5$ でのクリッピングはこのデータ不足を解消するか。

5. **ラベルなしデータからのシフト仮定の識別。**●二値のソースドメインにおいて $p_S(y = 1) = p_S(y = 0) = \frac{1}{2}$ であり、スカラー入力が $x \mid y = 1 \sim \mathcal{N}(1, 1)$ および $x \mid y = 0 \sim \mathcal{N}(-1, 1)$ に従うとする。このとき $p_S(y = 1 \mid x) = 1/(1 + \exp(-2x))$ となる。ラベルなしターゲット入力が次の混合分布に従うとする。

$$p_T(x) = 0.8\, \mathcal{N}(x; 1, 1) + 0.2\, \mathcal{N}(x; -1, 1).$$

1. ラベルシフトを仮定せよ。$p_T(y = 1)$ と、ベイズ最適なターゲット分類器の x に関する閾値を求めよ。
2. 代わりに同じ $p_T(x)$ のもとで共変量シフトを仮定せよ。$p_T(y = 1)$ を数値的に計算し、ベイズ最適な閾値を求めよ。
3. 以上の結果を用いて、ラベルなしターゲットサンプルのいかなる統計量によっても、本節の 2 つの補正手法のどちらを適用すべきか決定できない理由を説明せよ。

Chapter 4

多層パーセプトロン

本章では、最初の**深層**ニューラルネットワークを導入する。第 3 章では、ソフトマックス回帰が Fashion-MNIST の衣類画像を構成する 784 ピクセルそれぞれに固定の重みを割り当ててその根拠を合計していたため、周囲に何があるかにかかわらず、すべてのピクセルがクラスに対して同じ票を投じていた。しかし、明るいピクセルは袖、ハンドバッグのストラップ、襟のいずれにも属しうるため、そのどれであるかは近傍のピクセルを見なければわからない。4 つの二値点においてさえ、単一の線形層では 2 ビットの排他的論理和（exclusive-OR、XOR）を計算できない。これは Marvin Minsky と Seymour Papert が 1969 年に、Frank Rosenblatt の単層**パーセプトロン**について指摘した限界である。

多層パーセプトロン（multilayer perceptron、MLP）は、それぞれアフィン変換と非線形な活性化関数を組み合わせた全結合の**隠れ層**（hidden layer）を出力層の前に 1 つ以上積み重ね、すべての層を**誤差逆伝播法**（backpropagation）で同時に訓練することによって、この線形の壁を乗り越える (Rumelhart et al. 1986)。まず、なぜ線形モデルが XOR のような特徴量の相互作用を扱えないのか、隠れ層が深さに応じて入力空間をどのように指数関数的に多くの線形領域へと折り畳むのか、そして代表的な活性化関数（ReLU、GELU、シグモイド、tanh）が微分においてどのように振る舞うのかを示す（4.1 節）。続いて、256 ユニットの ReLU MLP を Fashion-MNIST 上でゼロから、また簡潔に実装すると（4.2 節）、ソフトマックス回帰の分類誤差が約 5 分の 1 削減され（正解率 $\sim 87\%$ 対 $\sim 84\%$）、GPU でのエポック時間はほぼ同じまま、ぼやけた衣類全体のテンプレートが局所的なコントラスト検出器に置き換わる。

隠れ層を追加すると、深層学習における 3 つの根本的な問いも生じる。これらが本章の残りの節の主題である。第 1 に、4.3 節では MLP を通じたリバースモード微分の 3 つの行列微分則を導出し、自動微分（automatic differentiation）と機械精度まで一致することを検証したうえで、訓練時の活性化メモリと FLOP の所要量を明らかにする。第 2 に、4.4 節では K 個の層にわたる信号と勾配の分散を追跡し、10 層の MLP における勾配消失と勾配爆発を防ぐ **Xavier** 初期化と **He** 初期化を導出するとともに、隠れユニットの対称性を破るためになぜランダムな重みが必要なのかを示す。第 3 に、4.5 節と 4.6 節では、大幅に過剰パラメータ化されたネットワークがなぜ汎化（generalization）するのかを検討し、ランダムラベルの記憶、二重降下（double descent）、勾配降下法の暗黙的バイアス（implicit bias）、早期停止、**ドロップアウト**による乗法的ノイズ正則化を扱う。そして最後に、4.7 節でこれら一連の手法を Kaggle の Ames Housing 回帰ベンチマークに適用する。

4.1 多層パーセプトロン

第 3 章では、ソフトマックス回帰は入力ピクセルのアフィン重み付き和 $o_j = \sum_{i=1}^{d} x_i w_{ij} + b_j$ を取ることで、衣類の各カテゴリにスコアを付けた。この線形形式には、2 つの厳しい制約が組み込まれている。

1. **単調性：**特徴量 x_i を増やすと、o_j は常に同じ方向へ動く（$w_{ij} > 0$ なら増加し、$w_{ij} < 0$ なら減少する）。しかし、現実の多くの関係は非単調である。たとえば人間の健康リスクは、体温が 37°C から**どちらの**方向に外れても上昇する。
2. **特徴量の相互作用の欠如：**x_i を 1 単位増やすと、他のすべての座標の値にかかわらず、o_j は w_{ij} だけ変化する。しかし画像においては、位置 $(13, 17)$ にある明るいピクセルが意味するものは、周囲のピクセルが襟、袖、ハンドバッグのストラップのどれを形作っているかによって全く異なる。

Figure 4.1 にこの体温の例をプロットした。最良の線形スコアは温度とともに一貫して上昇するため、低体温症の患者を誰よりも安全だと評価してしまう。

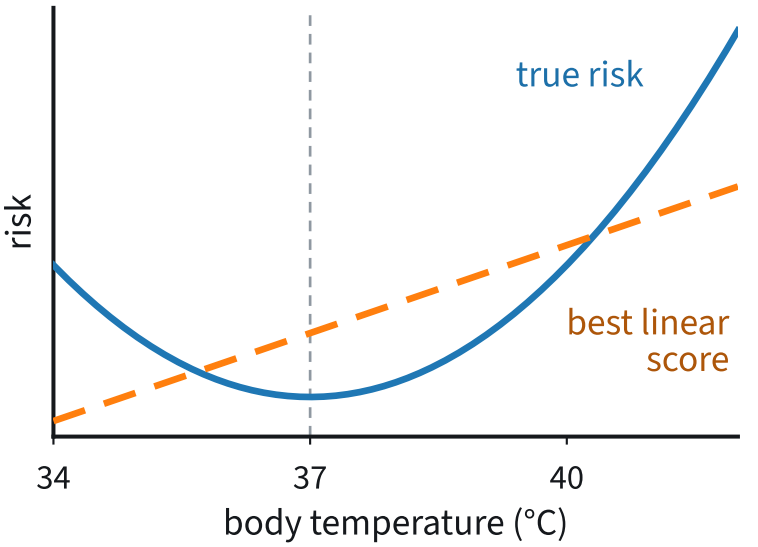


Figure 4.1: 体温に対する健康リスク。リスクは 37°C 付近で最も低く、その両側で上昇するが、線形スコアは単調に上昇または下降することしかできない。破線はプロット範囲における最小二乗法による線形当てはめである。

多層パーセプトロン（multilayer perceptron、MLP）は、生の入力を学習された中間表現へと変換する 1 つ以上の**隠れ層**（hidden layer）を挿入することで、これら両方の限界を克服する。各隠れ層ではアフィン写像に続けて非線形な**活性化関数**（activation function）を適用し、その後に線形の出力層が最終的な予測を下す。本節では、いかなる線形モデルも 4 点の XOR 問題を解けないことを証明し、2 個の ReLU 隠れユニットで XOR を明示的に解き、浅いネットワークと深いネットワークがどのような関数を表現できるかを調べ、現代の深層学習で広く用いられている活性化関数を比較する。

4.1.1 線形モデルの限界：XOR 問題

特徴量の相互作用を最も鮮明に示す例は、わずか 4 つのデータ点しか使わない。単位正方形 $\{0,1\}^2$ の各頂点に**排他的論理和**（exclusive-OR、XOR）でラベルを付けたものであり、$y=1$ となるのは**ちょうど 1 つ**の座標が 1 に等しいとき、かつそのときに限る。

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

この相互作用に注目しよう。x_1 を 0 から 1 に反転させると、$x_2=0$ のときには y が増加するが、$x_2=1$ のときには y が減少する。

命題：任意の $w_1, w_2, b \in \mathbb{R}$ に対し、線形決定規則「$w_1 x_1 + w_2 x_2 + b > 0$ ならばクラス 1 と予測する」は、XOR の 4 点のうち少なくとも 1 点を誤分類する。

証明：背理法のため、ある w_1, w_2, b が 4 つの頂点すべてを正しく分類すると仮定する。クラス 1 の 2 点 $(0,1)$ と $(1,0)$ からは $w_2 + b > 0$ および $w_1 + b > 0$ が必要となり、辺々を加えると $w_1 + w_2 + 2b > 0$ を得る。一方、クラス 0 の 2 点 $(0,0)$ と $(1,1)$ からは $b \le 0$ および $w_1 + w_2 + b \le 0$ が必要となり、辺々を加えると $w_1 + w_2 + 2b \le 0$ を得るが、これは直ちに矛盾する。$\blacksquare$

幾何学的には、単位正方形の 2 本の対角線はどちらも中点 $(1/2, 1/2)$ を共有している（Figure 4.2）。この中点において任意のアフィン関数はどちらの対角線の両端点の値の平均に等しくなるため、いかなる直線も 2 本の対角線を分離することはできない。

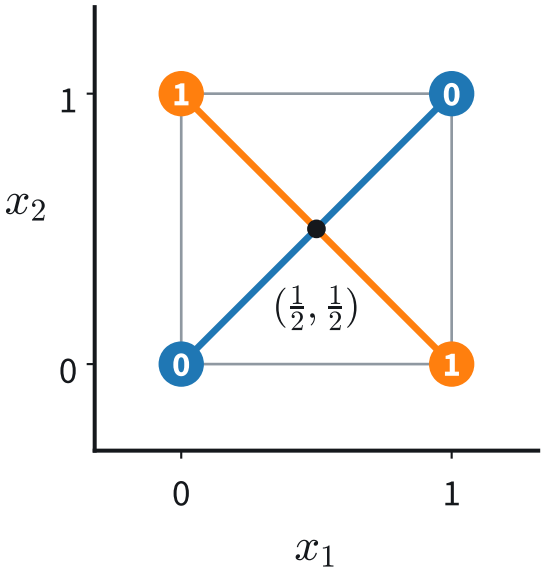


Figure 4.2: クラスでラベル付けされた XOR の 4 点。各対角線は同一クラスの 2 点を結び、2 本の対角線は $(1/2, 1/2)$ で交差するため、いかなる直線もクラス間を分離できない。

4.1.2 隠れ層

4.1.2.1 モデル

線形性から脱却するため、MLP は入力特徴量と出力層の間に 1 つ以上の**隠れ層**を挿入する（Figure 4.3）。ミニバッチの計画行列 $\mathbf{X} \in \mathbb{R}^{n \times d}$（データ例数 n、入力特徴量数 d）に対し、幅 h の単一の隠れ層は重み $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ とバイアス $\mathbf{b}^{(1)} \in \mathbb{R}^{h}$ を持ち、出力層は重み $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ とバイアス $\mathbf{b}^{(2)} \in \mathbb{R}^{q}$ を持つ。2 層（隠れ層 1 層）の MLP は次式を計算する。

$$\begin{aligned} \mathbf{H} &= \phi\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \end{aligned} \tag{4.1}$$

ここで、バイアスは各行にブロードキャストされ、非線形な**活性化関数** ϕ が**活性化前入力**（pre-activation）$\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$ に要素ごとに作用して、隠れ層の**活性化値**（activation）$\mathbf{H} \in \mathbb{R}^{n \times h}$ を生成する。パラメータ数が $(d+1)h + (h+1)q$ であることに注目しよう。Fashion-MNIST（$d = 784, q = 10$）では、$h = 256$ ユニットの隠れ層を持つモデルのパラメータ数は $785 \times 256 + 257 \times 10 = 203{,}530$ となり、ソフトマックス回帰の 7,850 の 26 倍に上る。

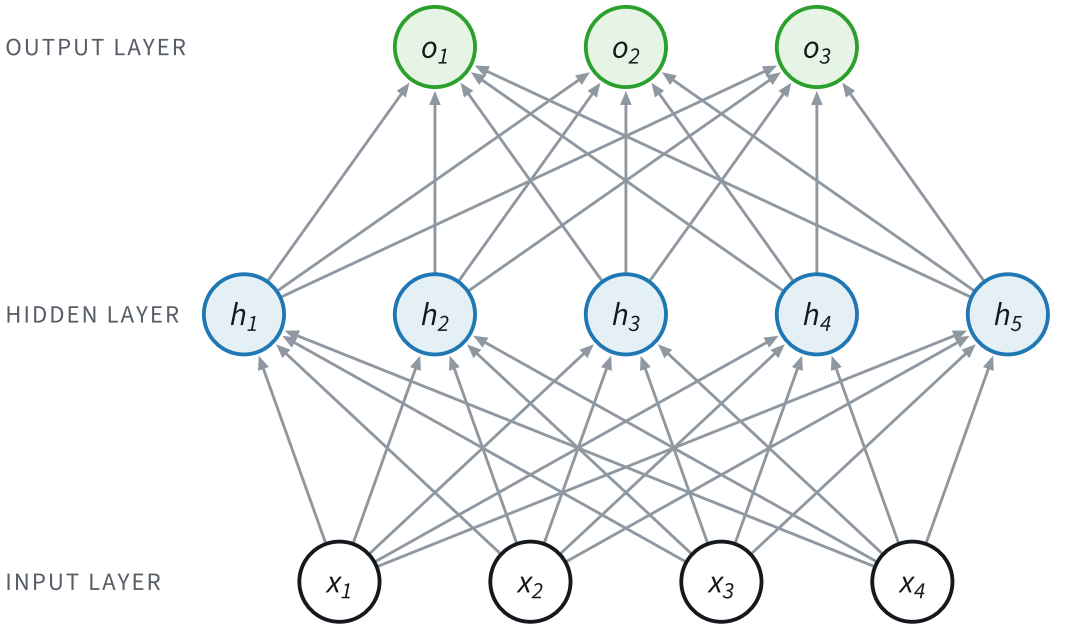


Figure 4.3: 4 個の入力、5 個の隠れユニットからなる 1 つの隠れ層、および 3 個の出力を持つ MLP。パラメータを持つ両方の層は全結合層である。

4.1.2.2 非線形性の役割

なぜ非線形な活性化関数 ϕ が不可欠なのだろうか。仮に ϕ を省略し（$\phi(z) = z$ とおき）、$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$ としたとしよう。この $\mathbf{H}$ を出力の式に代入すると、2 つの層は単一のアフィン変換（affine transformation）へと縮退してしまう。

$$\mathbf{O} = \left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\underbrace{\mathbf{W}^{(1)}\mathbf{W}^{(2)}}_{\mathbf{W}} + \underbrace{\mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}}_{\mathbf{b}}.$$

非線形性（nonlinearity）がなければ、全結合層（fully connected layer）を何層積み重ねても、表現できる関数の族は単一の線形層から決して広がらない（しかも、隠れ層の幅が $\min(d, q)$ より狭い場合、実際には $\mathbf{W}$ が低ランクに制限される。演習問題 1 を参照）。そこで、次式の**整流線形ユニット**（rectified linear unit、ReLU）のような要素ごとの非線形性を挿入する。

$$\mathrm{ReLU}(z) = \max(z, 0), \tag{4.2}$$

これにより、この代数的な縮退を防ぐことができる。

4.1.2.3 2 個の ReLU ユニットによる XOR の解法

ReLU 活性化関数を用いれば、わずか $h = 2$ ユニットの隠れ層で XOR を鮮やかに解くことができる (Goodfellow et al. 2016)。パラメータを次のように設定しよう。

$$\mathbf{W}^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}^{(1)} = (0, -1), \quad \mathbf{w}^{(2)} = (1, -2), \quad b^{(2)} = 0. \tag{4.3}$$

両方の隠れユニットは座標の和 $s = x_1 + x_2$ に依存する。ユニット 1 は $h_1 = \mathrm{ReLU}(s) = s$ を計算する一方、ユニット 2 は $h_2 = \mathrm{ReLU}(s - 1)$ を計算し、$s = 2$ のとき（点 $(1, 1)$）に**のみ**オンになるため、$o = h_1 - 2h_2$ が 2 を引いて出力を再び 0 へと折り返す。

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
W1, b1 = np.array([[1, 1], [1, 1]]), np.array([0, -1])
w2, b2 = np.array([1, -2]), 0
H = np.maximum(X @ W1 + b1, 0)  # hidden layer with ReLU
o = H @ w2 + b2  # output layer
for x, h, out in zip(X, H, o):
    print(f'x = {x}, h = {h}, o = {out}')
```

```
x = [0 0], h = [0 0], o = 0
x = [0 1], h = [1 0], o = 1
x = [1 0], h = [1 0], o = 1
x = [1 1], h = [2 1], o = 0
```

Figure 4.4 の幾何学的な構造を見てみよう。隠れ層はクラス 1 の**両方の**頂点 $(0, 1)$ と $(1, 0)$ を全く同じ隠れ空間上の点 $(1, 0)$ に写像するため、隠れ空間（右）における直線 $h_1 - 2h_2 = 1/2$ によって、入力空間（左）における半空間ではない対角帯 $1/2 < x_1 + x_2 < 3/2$ を切り出すことができる。

4.1.2.4 より深いネットワーク

より一般に、K 層の MLP は $K - 1$ 個の隠れ層を $\mathbf{H}^{(0)} = \mathbf{X}$ の上に次のように積み重ねる。

$$\begin{aligned} \mathbf{H}^{(k)} &= \phi\left(\mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}\right) \quad (k = 1, \ldots, K-1), \\ \mathbf{O} &= \mathbf{H}^{(K-1)}\mathbf{W}^{(K)} + \mathbf{b}^{(K)}, \end{aligned}$$

こうして、生の入力からクラスのロジットに至る表現の合成的な階層構造を築き上げる。

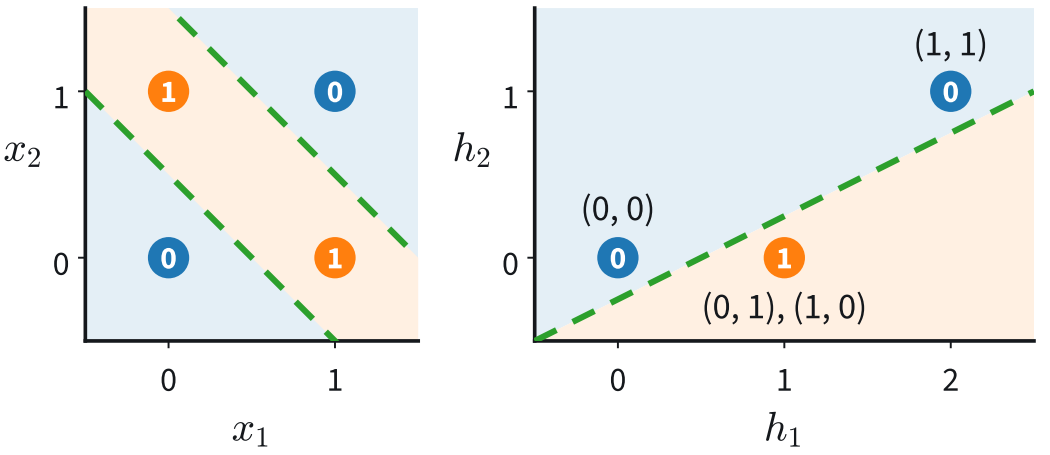


Figure 4.4: 2 個の ReLU 隠れユニットによる XOR の解法。左：入力空間 (x_1, x_2) では、ネットワークは対角帯 $1/2 < x_1 + x_2 < 3/2$ の内側でクラス 1 と予測する。右：隠れ空間 (h_1, h_2) では、クラス 1 の 2 点が $(1, 0)$ に縮退し、1 本の直線 $h_1 - 2h_2 = 1/2$ がクラス間を分離する。

4.1.3 表現力

4.1.3.1 万能近似

単一の隠れ層（$K = 2$）で計算できる関数のクラスはどれほど豊かなのだろうか。実直線上（$d = 1$）では、各 ReLU 隠れユニット $a_j \,\mathrm{ReLU}(x - t_j)$ は、節点（knot）t_j の左側で平坦であり右側で傾きを a_j だけ変化させる**ヒンジ**として寄与する（Figure 4.5）。h 個のヒンジの和に定数を加えれば、h 個の内部節点を持つ任意の連続な区分線形折れ線を作ることができる。有界閉区間 $[a, b]$ 上の任意の連続関数は、十分に細かい折れ線によって任意の $\epsilon > 0$ 以内で一様近似できるため、十分に幅の広い単一の ReLU 隠れ層は、任意の 1 次元連続関数を任意の精度で近似できる。

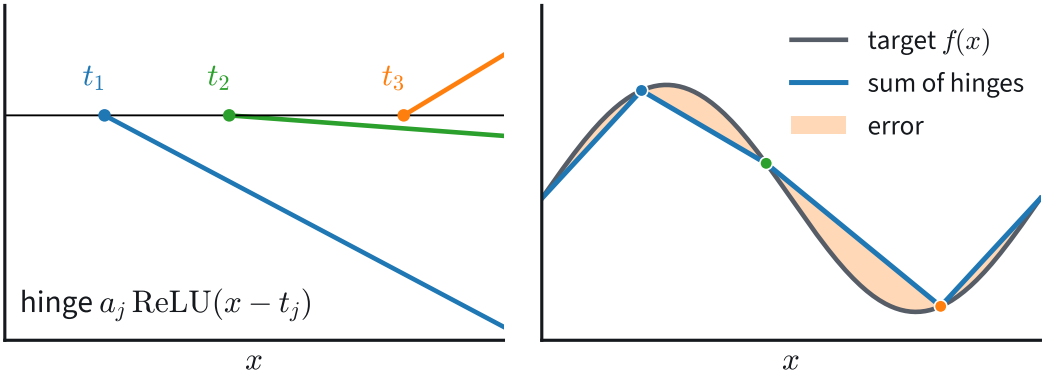


Figure 4.5: 区間上の滑らかな曲線を、傾きの変化ごとに 1 つの隠れユニットを割り当てた ReLU ヒンジ $a_j \,\mathrm{ReLU}(x - t_j)$ の和で近似する。

万能近似定理（universal approximation theorem）は、この事実を任意の入力次元 d へと一般化する。すなわち、隠れ層 1 層の MLP は、$\mathbb{R}^d$ の任意のコンパクト部分集合上の**任意の**連続関数を、**任意の**連続な非多項式の活性化関数 ϕ を用いて任意の一様精度で近似できる (Cybenko 1989;

Leshno et al. 1993)（リッジ関数と Stone–Weierstrass の定理を用いた 2 次元での証明については、演習問題 5 を参照）。

4.1.3.2 深さと空間の折りたたみ

隠れ層 1 つですでに何でも近似できるなら、なぜ**深い**ネットワークを構築するのだろうか。それは、万能近似が単一の層に必要な**ユニット数**（d に対して指数関数的に増大し得る）について何も語っておらず、また勾配降下法（gradient descent）が有限のデータから学習・汎化（generalization）できるかどうかも保証しないからである。

深さは、空間を折りたたむ写像を**合成**することで指数関数的なパラメータ効率を達成する (Telgarsky 2016)。2 個の ReLU ユニットによって計算される $[0,1]$ 上の 1 次元テント関数を考えよう。

$$g(z) = 2\,\mathrm{ReLU}(z) - 4\,\mathrm{ReLU}(z - 1/2), \tag{4.4}$$

この関数は $[0, 1/2]$ 上で 0 から 1 まで上昇し、$[1/2, 1]$ 上で再び 0 まで下降する（Figure 4.6）。g をそれ自身と合成すると、層ごとに $[0,1]$ が半分に折りたたまれる。$g \circ g$ は 4 個の線形区間を持ち、$g \circ g \circ g$ は 8 個を持ち、m 回の合成 $g^{\circ m}$ は m 個の隠れ層（各層 2 個の ReLU ユニット、合計 $2m$ ユニット）で計算され、2^m 個の線形区間を持つのこぎり波を生成する。この同じ 2^m 区間ののこぎり波を**単一の**隠れ層で表現するには、少なくとも $2^m - 1$ 個の隠れユニットが必要になる（$m = 10$ のとき、1 層では 1,023 ユニット必要なのに対し、10 層にわたる構成なら 20 ユニットで済む）。

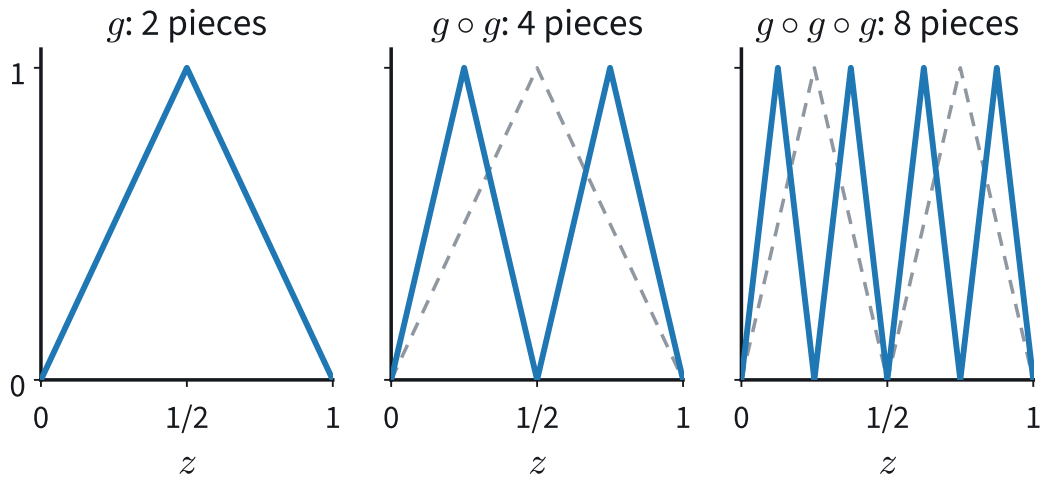


Figure 4.6: 2 個の ReLU からなるテント関数 g をそれ自身と m 回合成すると、層ごとに線形区間の数が倍増し、2^m 個の区間がわずか $2m$ 個の隠れユニットで生成される。

4.1.4 活性化関数

万能近似には非多項式であればほぼどのような ϕ でも十分であるが、勾配降下法による最適化は導関数 $\phi'(z)$ に決定的に依存する。隠れユニット j において、活性化前入力を $z_j = \sum_i x_i w_{ij}^{(1)} + b_j^{(1)}$、

活性化値を $h_j = \phi(z_j)$ とすると、連鎖律（chain rule）により入力側の各重みの勾配は局所的な傾き $\phi'(z_j)$ 倍にスケーリングされる。

$$\frac{\partial l}{\partial w_{ij}^{(1)}} = \frac{\partial l}{\partial h_j}\,\phi'(z_j)\,x_i. \tag{4.5}$$

$\phi'(z_j) \approx 0$ となる場所ではどこでも、勾配フロー（gradient flow）が停滞する。Figure 4.7 に 4 つの主要な活性化関数とその導関数を比較する。

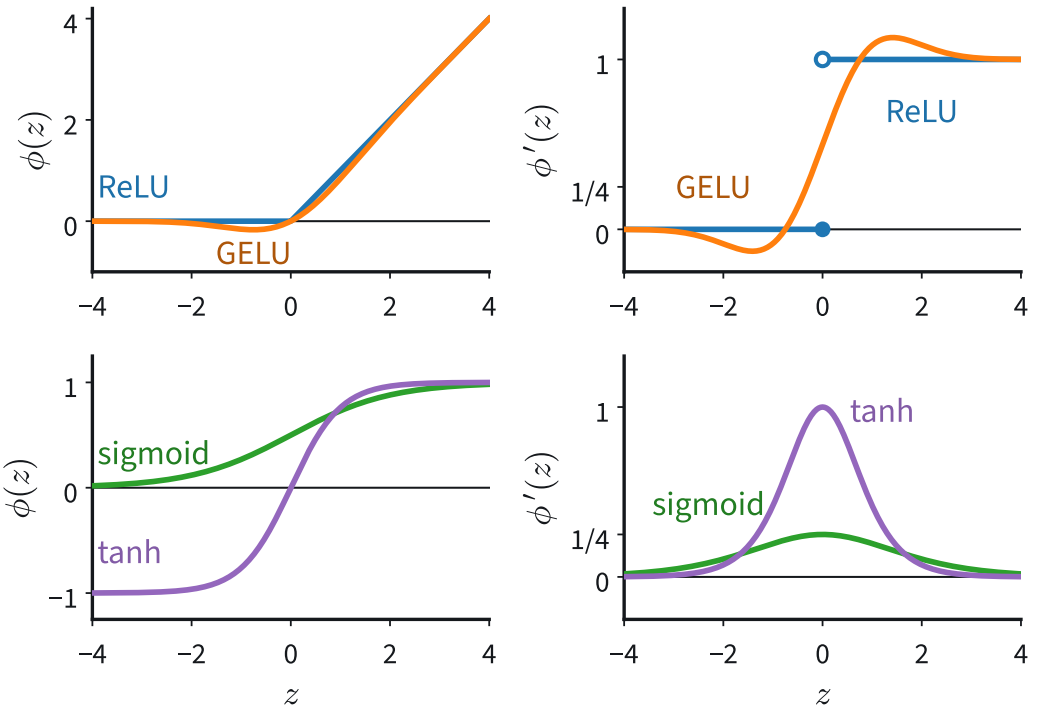


Figure 4.7: 4 つの活性化関数 $\phi(z)$（左）とその導関数 $\phi'(z)$（右）。ReLU と GELU（上）、シグモイドと tanh（下）。

- **ReLU と Leaky ReLU**：$\mathrm{ReLU}(z) = \max(z, 0)$ の導関数は $z > 0$ で 1、$z < 0$ で 0（慣例として $z = 0$ では 0）である。正の半直線全体にわたって傾きが恒等的に 1 であるため、活性な ReLU ユニットでは z が大きくなっても勾配が決して減衰しない (Nair and Hinton 2010; Glorot et al. 2011)。しかし、ユニットの重みが変化してすべての訓練入力に対して $z < 0$ になると、その勾配は恒久的に消失する（**死んだ ReLU**、dead ReLU）。そこで **Leaky ReLU** すなわち $\max(z, \alpha z)$（$\alpha \approx 0.01$）は、$z < 0$ に対して小さな正の傾きを保つことで、死んだユニットが回復できるようにする。
- **GELU と SiLU（Swish）**：このうち**ガウス誤差線形ユニット**（Gaussian error linear unit）$\mathrm{GELU}(z) = z\,\Phi(z)$（ただし Φ は標準正規分布の累積分布関数）とその類縁である $\mathrm{SiLU}(z) = z\,\sigma(z)$ は、原点における ReLU の折れ角を滑らかにし（$z = 0$ を傾き $1/2$ で通り、$z \approx -0.75$ 付近に緩やかな負のくぼみを持つ）、現代の Transformer においてデフォルトの活性化関数として用いられている (Hendrycks and Gimpel 2016)。

- **シグモイドと Tanh**：ロジスティックシグモイド $\sigma(z) = 1/(1+e^{-z}) \in (0,1)$ の導関数は $\sigma'(z) = \sigma(z)(1-\sigma(z)) \le 1/4$ である一方、ゼロ中心の双曲線正接（hyperbolic tangent）$\tanh(z) = (1-e^{-2z})/(1+e^{-2z}) = 2\sigma(2z) - 1 \in (-1,1)$（演習問題 3）の導関数は $1 - \tanh^2(z) \le 1$ である。どちらも、両裾で平坦な水平漸近線へと飽和して $\phi'(z) \to 0$ となり（$|z|$ が ~ 3 を超えて大きくなるとき）、$\sigma'(z) \le 1/4$ を K 層にわたって掛け合わせると勾配は少なくとも 4^{-K} 倍に縮小する（**勾配消失**、vanishing gradients。4.4 節で解析する）。現在、シグモイドと tanh は主にリカレントアーキテクチャにおける有界な**ゲート**や状態の圧縮関数として用いられている（第 12 章）。

4.1.5 まとめ

線形モデルはすべての座標に関して単調であり、XOR のような特徴量の相互作用を捉えることができない。多層パーセプトロンは、線形の出力層の前に隠れ層 $\mathbf{H}^{(k)} = \phi(\mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)})$ を挿入することでこれら両方の限界を克服する。非線形な活性化関数 ϕ がなければ、アフィン変換を何層積み重ねても単一の線形変換へと縮退してしまう。任意の連続な非多項式の活性化関数を持つ十分に幅広い単一の隠れ層は、コンパクト集合上の任意の連続関数を近似できる一方、m 個の隠れ層を合成すれば空間が繰り返し折りたたまれ、2^m 個の線形領域を $O(m)$ 個のユニットで作り出せる。活性化関数のなかでも、ReLU（$\max(z,0)$）や GELU（$z\Phi(z)$）のような整流型活性化関数は正の入力に対して 1 または 1 に近い傾きを維持するため、シグモイドや tanh ユニットを深く積み重ねた際に勾配を減衰させる裾の飽和を回避できる。

4.1.6 演習問題

1. **線形ネットワーク：**活性化関数がなければ、層を積み重ねてもアフィン写像が合成されて別のアフィン写像になるだけである。入力次元 d、出力次元 q、隠れ層の幅 $h_1, \dots, h_{K-1}$ を持つこのようなネットワークを考える。このネットワークが表現する関数が、アフィン写像 $\mathbf{XW} + \mathbf{b}$ のうち重み行列 $\mathbf{W}$ のランクが高々 $\min(h_1, \dots, h_{K-1})$ であるもの全体とちょうど一致することを示せ。この結果を用いて、線形モデルに隠れ層を追加するとその関数クラスが真に縮小する例を挙げよ。
2. **手計算による XOR：**@fig-mdl-mlp-xor は、クラス 1 の 2 つの入力を同じ隠れ空間上の点に写像することで XOR を計算する 2 ユニットの ReLU ネットワークを示している。
   1. 同じく XOR を計算するが、クラス 1 の 2 つの入力を**異なる**隠れ空間上の点に写像する 2 ユニットの ReLU ネットワークの重みを求め、4 つの入力で検証せよ。
   2. 単一の ReLU 隠れユニットと線形出力を持ついかなるネットワークも、その出力にどのようなしきい値を適用しても XOR を正しく分類できないことを示せ。
3. **シグモイドと tanh：**これら 2 つの圧縮関数は密接に関連している。
   1. $\tanh(z) = 2\sigma(2z) - 1$ を示せ。
   2. 隠れ層の活性化関数に tanh を持つ任意の MLP に対し、同じアーキテクチャで隠れ層の活性化関数にシグモイドを持ち同じ関数を計算する MLP が存在すること、またその逆も成り立つことを示せ。重みとバイアスがどのように変化するかを述べよ。
4. **深さと線形区間：**●実直線上で h 個の隠れユニットを持つ隠れ層 1 層の ReLU ネットワークは高々 $h+1$ 個の線形区間しか持たない一方、Figure 4.6 の 2 ユニットのテント関数を合成するたびに区間数は倍増する。本問では、1 層あたり 3 ユニットを持つネットワークについて深さによる利得を厳密に求める。
   1. 3 個の隠れユニット $\mathrm{ReLU}(w_j x + b_j)$ をとり、入力の重みを $w_1 = w_2 = 1$ および $w_3 = -1$、折れ点 $-b_j/w_j$ を $1/6$、$2/6$、$4/6$ とする。ネットワークの出力 f が値 $0, 1, 0, 1, 0$ を $x = 0, 1/6, 2/6, 4/6, 1$ で順にとるように、出力の重みと出力バイアスを選べ。
   2. $f \circ f$ が $[0,1]$ 上で持つ線形区間の数、および m 回の合成 $f \circ \cdots \circ f$ の線形区間の数

を数えよ。f との合成がグラフをどのように折りたたむかという観点から、合成のたびに区間数が加算ではなく乗算される理由を説明せよ。

3. この m 回の合成は、各 3 ユニットからなる m 個の隠れ層を持つネットワークによって計算される。単一の隠れ層を持つネットワークで同じ関数を表現するには、何個の隠れユニットが必要か。2 つのネットワークのパラメータ数を比較せよ。

Simon Prince『Understanding Deep Learning』問題 4.8 より改変。

5. **単一の隠れ層で十分 :●**実直線上では、単一の隠れ層は折れ点ごとに 1 つのヒンジを配置した折れ線によって連続関数を近似する。平面上には配置すべき折れ点が存在しないため、証明には別のアイデア、すなわち 1 つの方向に沿ってのみ変化する関数が必要になる。
   1. ψ を $\mathbb{R}$ 上の連続関数とし、$\mathbf{w} \in \mathbb{R}^2$ とする。単一の隠れ層が、**リッジ関数** $\psi(\mathbf{w}^\top \mathbf{x})$（$\mathbf{w}$ に直交するすべての直線上で定数となる関数）を平面の任意のコンパクト部分集合上で一様に近似できることを示せ。
   2. Stone–Weierstrass の定理は次のように述べる。コンパクト集合上の連続関数の集合が、和、積、スカラー倍に関して閉じており、定数関数を含み、任意の異なる 2 点に対してその 2 点で異なる値をとる関数を含むとする。このとき、そのコンパクト集合上のすべての連続関数は、この集合に属する関数の一様極限となる。$\cos(\omega^\top \mathbf{x})$ と $\sin(\omega^\top \mathbf{x})$（$\omega \in \mathbb{R}^2$）の有限線形結合全体がこのような集合をなすことを示せ。
   3. 以上より、単一の隠れ層が平面のコンパクト部分集合上の任意の連続関数を近似できることを結論づけよ。

Michael Nielsen『Neural Networks and Deep Learning』第 4 章より改変。

4.2 実装

4.1 節では、XOR を解くために 2 ユニットの ReLU ネットワークの重みを手作業で設計した。今度は、ミニバッチ SGD によって Fashion-MNIST 画像から隠れ特徴を直接学習させてみよう。隠れ層（hidden layer）として $h = 256$ 個の ReLU ユニットからなる単一の層を、$d = 784$ 個の入力ピクセルと $q = 10$ 個のクラスロジットの間に挿入すると、パラメータ数（parameter count）と画像あたりの FLOPs は 26 倍になる（パラメータ数は 203,530 個に対し、ソフトマックス回帰（softmax regression）では 7,850 個）。

本節では、この 2 層 MLP をスクラッチ（`MLPScratch`）と、簡潔な方法（`MLP`、フレームワークの `Sequential` コンテナを使用）の両方で実装する。さらに、訓練前の手軽な 3 つの妥当性確認によって未訓練のネットワークを検証し、その学習曲線（learning curve）と GPU エポック時間をソフトマックス回帰と直接比較したうえで、隠れユニット（hidden unit）が $\mathbf{W}^{(1)}$ 内に学習する重みパターンを観察する。

```
import time
```

4.2.1 スクラッチ実装

4.2.1.1 パラメータと順伝播

各 28×28 画像を $d = 784$ 個の特徴量（feature）に平坦化すると、$h = 256$ 個の隠れユニットと $q = 10$ 個の出力を持つ隠れ層 1 層の MLP（Figure 4.8）は、4 つのパラメータテンソルを確保する。

$$\mathbf{W}^{(1)} \in \mathbb{R}^{784 \times 256}, \quad \mathbf{b}^{(1)} \in \mathbb{R}^{256}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{256 \times 10}, \quad \mathbf{b}^{(2)} \in \mathbb{R}^{10}. \tag{4.6}$$

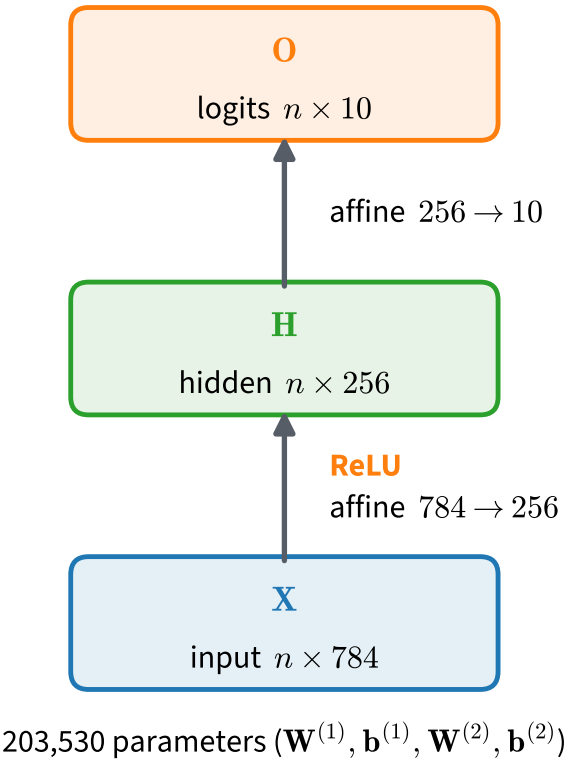


Figure 4.8: Fashion-MNIST 用 MLP のアーキテクチャ。n 枚の画像からなるミニバッチを $n \times 784$ に平坦化し、アフィン層と ReLU を通して $n \times 256$ の隠れ表現 $\mathbf{H}$ に写像したのち、$n \times 10$ のクラスロジット $\mathbf{O}$ へ射影する。

ソフトマックス回帰との直接比較で条件を揃えるため、両方の重み行列を $\mathcal{N}(0, 0.01^2)$ から、両方のバイアスを 0 に初期化する（4.4 節では、隠れユニット間の対称性を破るためにランダム初期化（random initialization）が不可欠である理由と、He 初期化（He initialization）が `std=0.01` をどのように改善するかを見る）。

```
class MLPScratch(d2l.Classifier):
    """An MLP with one hidden layer, implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(d2l.normal(0, std, (num_inputs, num_hiddens)))
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens))
        self.W2 = nn.Parameter(d2l.normal(0, std, (num_hiddens, num_outputs)))
        self.b2 = nn.Parameter(d2l.zeros(num_outputs))
```

次に、要素ごとの活性化関数 `relu(X)` をスクラッチで実装する。`X` にブールマスク `X > 0` を掛けることで、正の要素はそのまま通過させ、負の要素を 0 にする。

```
def relu(X):
    return X * (X > 0)
```

続いて隠れ層の処理を `hidden(self, X)` として切り出す。このメソッドは各画像ミニバッチを `(-1, self.num_inputs)` に平坦化して $\mathbf{H} = \mathrm{relu}(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ を計算するため、`forward(self, X)` は生のクラスロジット $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ を返すだけでよい（この出力は `Classifier.loss` にそのまま渡される）。

```
@d2l.add_to_class(MLPScratch)
def hidden(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return relu(d2l.matmul(X, self.W1) + self.b1)

@d2l.add_to_class(MLPScratch)
def forward(self, X):
    return d2l.matmul(self.hidden(X), self.W2) + self.b2  # The logits
```

4.2.1.2 訓練前の確認

時間のかかる訓練を開始する前に、8 枚の画像からなる小さなスライスを**未訓練**のネットワークに通すことで、3 つの不変条件を数ミリ秒で検証できる。

1. **形状：** `H` の形状は `(8, 256)`、`O` の形状は `(8, 10)` である。
2. **初期損失：** ゼロに近い重みからはゼロに近いロジット（$\hat{y}_j \approx 1/10$）が生じるため、初期の交差エントロピー損失（cross-entropy loss）は $\log 10 \approx 2.303$ のすぐ近くに収まる。
3. **活性な ReLU の割合：** バイアスが 0 で重みが対称なため、隠れ層の活性化前の値（pre-activation）の約 50% が正になる。

```
data = d2l.FashionMNIST(batch_size=256)
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
X, y = next(iter(data.train_dataloader()))
```

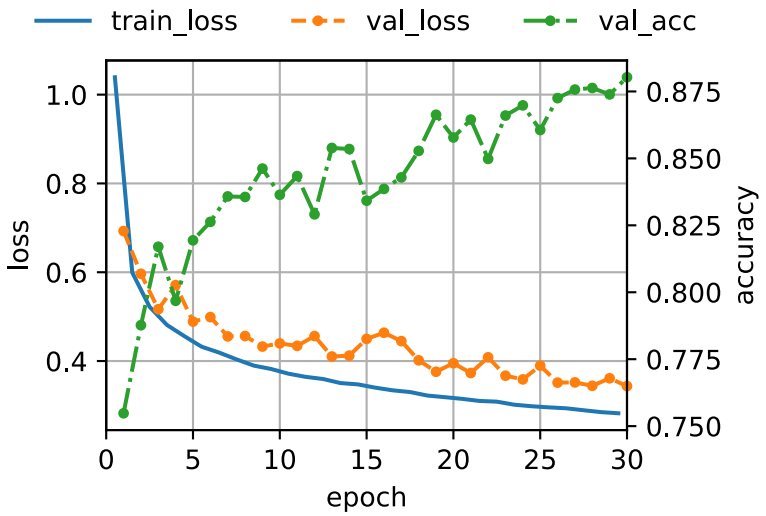
```
X, y = X[:8], y[:8]  # Eight images suffice for the checks
H, O = model.hidden(X), model(X)
active = d2l.reduce_mean(d2l.astype(H > 0, d2l.float32))
print('H:', H.shape, 'O:', O.shape)
print(f'loss {d2l.numpy(model.loss(O, y)):.3f}, '
      f'fraction of active units {d2l.numpy(active):.2f}')
```

```
H: torch.Size([8, 256]) O: torch.Size([8, 10])
loss 2.300, fraction of active units 0.52
```

4.2.1.3 訓練

それでは、Fashion-MNIST 上で MLPScratch を 30 エポックにわたり、ミニバッチサイズを 256、学習率（learning rate）を $\eta = 0.1$ として訓練しよう（num_gpus=1 を渡し、GPU が利用可能なときには Trainer がそれを使うようにする）。

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
```



4.2.2 簡潔な実装

各フレームワークの Sequential コンテナと組み込みの線形層および ReLU 層を使えば、パラメータ配列を手動で定義することなく、まったく同じアーキテクチャを数行で記述できる（重みを $\mathcal{N}(0, 0.01^2)$ から、バイアスを 0 に初期化して MLPScratch と揃える）。

```
class MLP(d2l.Classifier):
    """An MLP with one hidden layer, built from framework layers."""
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
```

```
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        # The Trainer calls this once the lazy layers know their shapes
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

4.2.3 ソフトマックス回帰との比較

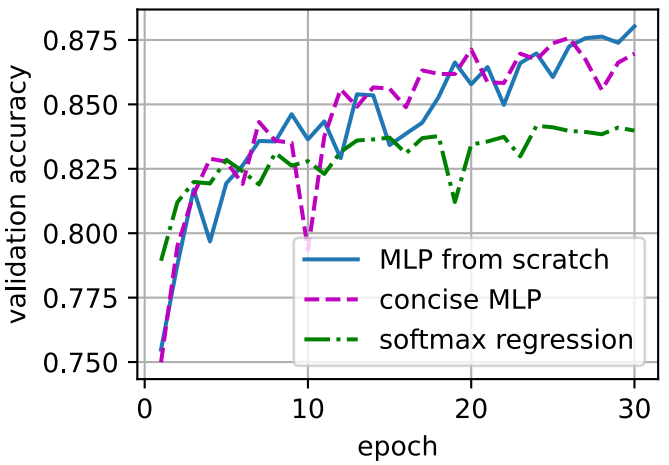
4.2.3.1 正解率

簡潔な実装の MLP と d2l.SoftmaxRegression の両方を同一の設定のもとで 30 エポック訓練し、各モデルのエポックあたりの平均実時間（wall-clock time）を seconds に、検証データでの正解率（accuracy）の推移を curves に記録しよう。

```
curves, seconds = {'MLP from scratch': trainer.history['val_acc']}, {}
trainer.board.display = False
models = {'concise MLP': MLP(num_outputs=10, num_hiddens=256, lr=0.1),
          'softmax regression': d2l.SoftmaxRegression(num_outputs=10, lr=0.1)}
for name, net in models.items():
    start = time.perf_counter()
    trainer.fit(net, data)
    seconds[name] = (time.perf_counter() - start) / trainer.max_epochs
    curves[name] = trainer.history['val_acc']
```

curves に記録した 3 つの検証正解率の履歴をエポック 1 から 30 にわたってプロットし、スクラッチ実装と簡潔な実装の両 MLP を線形のソフトマックスベースラインと直接比較する。

```
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
```



単一エポックごとの検証値のばらつきをならすため、最後の 5 エポック（acc[-5:]）における各

モデルの平均検証正解率を、MLPScratch の訓練セット（training set）での正解率とあわせて出力する。

```
print('validation accuracy averaged over the last 5 epochs')
for name, acc in curves.items():  # A single epoch can land a few points low
    print(f'  {name}: {sum(acc[-5:]) / 5:.3f}')
train_acc = trainer.evaluate(model, data, train=True)['acc']
print(f'training accuracy of the MLP from scratch: {train_acc:.3f}')
```

```
validation accuracy averaged over the last 5 epochs
  MLP from scratch: 0.876
  concise MLP: 0.867
  softmax regression: 0.840
training accuracy of the MLP from scratch: 0.903
```

この学習曲線には、示唆に富む 2 つのパターンが見て取れる。

1. **std=0.01 による遅いウォームアップ：**エポック 1 では、実はどちらの MLP もソフトマックス回帰に遅れをとる。$\mathbf{W}^{(1)}$ と $\mathbf{W}^{(2)}$ の両方が 0.01 前後の要素から始まるため、隠れ層の活性化値（activation）$\mathbf{H}$ および $\mathbf{W}^{(1)}$ へ逆伝播される勾配が 0.01 倍に抑制され、その状態は $\mathbf{W}^{(2)}$ が成長するまで続く。このボトルネックは 4.4 節で He 初期化によって解消する。
2. **分類誤りの 5 分の 1 の削減：**エポック 26〜30 までに、ソフトマックス回帰は検証正解率 84% 前後（誤り率 16%）で頭打ちになるのに対し、両 MLP は検証正解率 ～ 87%（誤り率 ～ 13%、訓練正解率 87〜90%）に達し、ソフトマックス回帰の誤分類の約 20% を解消する。

残っている誤分類はどのようなものだろうか。3.2 節で見たように、上半身のトップス 4 種（T シャツ、プルオーバー、コート、シャツ）は互いに似ているため、当然の候補となる。いま訓練した 2 つのモデルそれぞれについて、検証データでの誤分類数と、そのうちトップス同士を取り違えた件数を数えてみよう。

```
tops = [0, 2, 4, 6]  # T-shirt/top, pullover, coat, shirt
for name, net in models.items():
    errors = among_tops = 0
    for batch in data.val_dataloader():
        X, y = trainer.prepare_batch(batch)
        pred, y = d2l.numpy(net(X)).argmax(axis=1), d2l.numpy(y)
        errors += (pred != y).sum()
        among_tops += ((pred != y) & np.isin(y, tops) & np.isin(pred, tops)).sum()
    print(f'{name}: {errors} errors, {among_tops} of them between two tops')
```

```
concise MLP: 1303 errors, 772 of them between two tops
softmax regression: 1602 errors, 959 of them between two tops
```

どちらのモデルでも、全誤分類の半数以上が 2 種のトップスの混同である。MLP はソフトマックス回帰よりもこうした誤りを減らしているものの、トップス類の識別は依然として Fashion-MNIST の難所である。

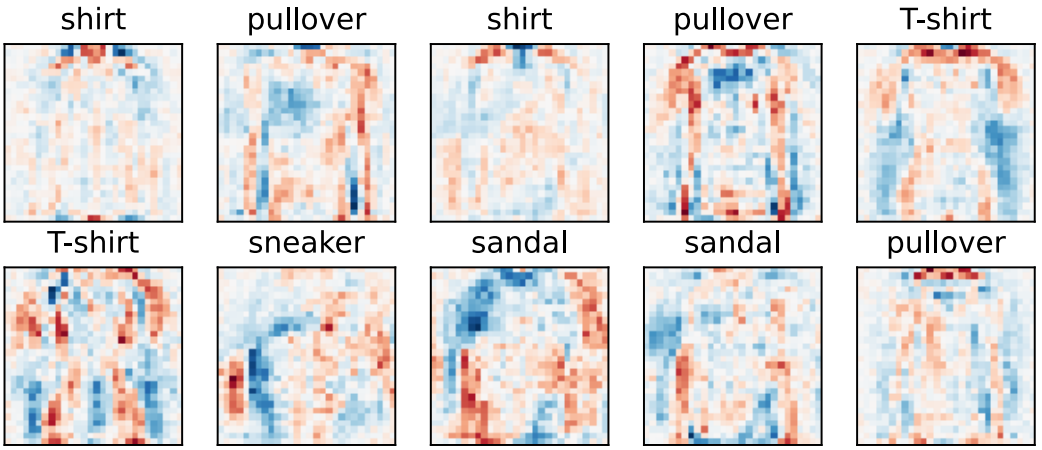
4.2.3.2 学習された特徴

3.3 節では、ソフトマックス回帰の重み行列 $\mathbf{W}$ の 10 個の列が、それぞれ衣類全体のテンプレートとして機能せざるを得なかった。これに対し、MLP の 256 個の隠れユニットが何を学習したかを調べるため、訓練済みの重み行列 $\mathbf{W}^{(1)}$ と $\mathbf{W}^{(2)}$ を NumPy 配列として取り出す。

```
W1, W2 = d2l.numpy(model.W1), d2l.numpy(model.W2)
```

重みのユークリッドノルムが最も大きい 10 個の隠れユニットを 28×28 の画像として表示し、各ユニットには出力側の正の重みが最大となる衣類クラスのラベルを添える。

```
names = data.text_labels(range(10))
units = np.argsort(np.linalg.norm(W1, axis=0))[::-1][:10]  # Largest weights
images = W1[:, units].T.reshape(10, 28, 28)
images /= np.abs(images).max(axis=(1, 2), keepdims=True)  # Own color scale
d2l.show_images(images, 2, 5, titles=[names[k] for k in W2[units].argmax(1)],
                cmap='RdBu_r', vmin=-1, vmax=1);
```



これらの画像は、衣類全体のテンプレートには見えない。ノルムの大きい各隠れユニットは、画像の一部の領域に正の重み（赤）を、隣接する領域に負の重み（青）を与えており、たとえば肩の輪郭や胴体の両脇に沿った「ある領域の明るいピクセルとその隣の暗いピクセル」という**コントラスト**に反応する。もっとも、パターンの大半は依然として衣類の広い範囲にまたがっているため、小さなエッジ検出器とみなすべきではない。出力層は、このようなコントラストを 256 個組み合わせることで 10 個のスコアを作る。

4.2.3.3 パラメータ、計算量、時間

モデル	パラメータ数	画像あたりの積和演算数	検証正解率（エポック 26〜30）
ソフトマックス回帰	7,850	7,840	$\sim 84\%$
MLP（$h = 256$ 個の ReLU ユニット）	203,530	203,264	$\sim 87\%$

この MLP は画像あたりの積和演算数がソフトマックス回帰の 26 倍にのぼるにもかかわらず、エポックあたりの実時間がどうなるか比較してみよう。

```
for name, s in seconds.items():
    print(f'{name}: {s:.1f} s per epoch')
```

```
concise MLP: 1.3 s per epoch
softmax regression: 1.3 s per epoch
```

現代のハードウェア（特に GPU）では、どちらのモデルもエポックあたりの実時間はほぼ同じである。幅 $h = 256$ では、データの読み込みや Python またはカーネル起動のオーバーヘッドが実行時間の大部分を占めるため、密行列演算が 26 倍に増えても実時間上のコストはほとんど増えない（演習問題 3 では、どの幅から行列演算がデータ読み込みのオーバーヘッドを上回るかを調べる）。

4.2.4 まとめ

訓練を開始する前に、小さなバッチに対して順伝播（forward pass）を 1 回実行することで、中間テンソルと出力テンソルの形状を検証し、初期の交差エントロピー損失が $\log q \approx 2.30$ 付近にあることを確かめ、さらに ReLU 隠れユニットの約半数が活性化することを確認できる。Fashion-MNIST では、256 ユニットの ReLU 隠れ層を 1 層挿入することで検証正解率が $\sim 84\%$ から $\sim 87\%$ へと向上し、エポックあたりの GPU 実時間をほぼ変えずに誤り率を約 5 分の 1 削減できる。このとき隠れユニットは、ぼやけた衣類全体のテンプレートを作るのではなく、隣接する領域に逆符号の重みを与えるコントラストパターンを学習し、それらが出力層で組み合わされる。

4.2.5 演習問題

1. [code] **活性化関数なしの隠れ層**。`relu` を `MLPScratch` 内で恒等写像に置き換え、得られたモデルを本節と同様に訓練せよ。
   1. 30 エポック後の検証正解率を、MLP およびソフトマックス回帰の検証正解率と比較せよ。
   2. 訓練後に $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ と $\mathbf{b} = \mathbf{W}^{(2)\top}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}$ を計算し、ミニバッチ上で $\mathbf{X}\mathbf{W} + \mathbf{b}$ がモデルのロジットを再現することを確かめよ。
   3. 演習問題 1 で示したとおり、このようなネットワークが計算するのは、重み行列のランクが高々隠れ層の幅であるようなアフィン写像にちょうど一致する。ここでこの制約が実質的に何の制限にもならない理由を説明し、さらに隠れ層の幅がいくつ未満になると Fashion-MNIST 上でモデルが表現できる範囲が縮小するかを述べよ。
2. [code] **単一の隠れユニット**。`num_hiddens=1` を `MLPScratch` に設定し、隠れ層が重み $\mathbf{w}^{(1)} \in \mathbb{R}^{784\times 1}$ とバイアス $b^{(1)} \in \mathbb{R}$ を持つ単一の ReLU ユニットからなるようにせよ。このとき各入力 $\mathbf{x} \in \mathbb{R}^{1\times 784}$ はスカラー $t = \mathrm{ReLU}(\mathbf{x}\mathbf{w}^{(1)} + b^{(1)}) \geq 0$ に縮約され、ロジットは $\mathbf{o} = t\,\mathbf{w}^{(2)} + \mathbf{b}^{(2)}$（ただし $\mathbf{w}^{(2)} \in \mathbb{R}^{1\times 10}$、$\mathbf{b}^{(2)} \in \mathbb{R}^{10}$）となる。
   1. 各クラスが予測されるのは t の高々 1 つの区間上であること、および t 軸に沿ったそれらの区間の順序が $\mathbf{w}^{(2)}$ の各成分によって定まることを示せ。さらに、モデルが各クラスに割り当てる入力空間の領域を記述し、それが Fashion-MNIST の 10 クラスを分離するうえで何を意味するかを述べよ。
   2. ある更新の後に、すべての訓練例に対して $\mathbf{x}\mathbf{w}^{(1)} + b^{(1)} < 0$ になったとする。それ以降、0 でない勾配を受け取るパラメータはどれか、またネットワークは何を予測するか。本節の 256 個の隠れユニットのうち 1 つに同じ現象が起きてもほとんど悪影響がないのはなぜか。
   3. モデルを 5 回訓練し、各実行について検証正解率と、検証例のうち $t > 0$ となる割合（`model.hidden` から得られる）を報告せよ。この割合が 0 になった実行があれば、その結果を (b) の解答と比較せよ。
3. [code] **幅**。本節と同様に `MLP` を学習率 0.1 で 30 エポック訓練せよ。ただし隠れ層の幅を $h \in \{16, 64, 256, 1024, 4096\}$ と変化させ、最終的な検証正解率とエポックあたりの時間を記録せよ。
   1. 両方を対数軸上の h に対してプロットせよ。どの幅を超えると、正解率の改善幅がエ

ポック間の変動を超えなくなるか。
2. 積和演算数は h に比例して増加する。モデルを一切使わずミニバッチの読み込みだけを行う訓練データ 1 周分の時間を計測し、それを用いて時間曲線のうち h に伴って増加しない部分を説明せよ。使用しているハードウェアでは、どの幅から演算時間が支配的になるか。
3. 幅 64 と 1024 について、学習率を $\{0.03, 0.1, 0.3\}$ として訓練を繰り返せ。最良の学習率は幅に依存するか。

4.3 誤差逆伝播法

4.2 節では、導関数を 1 つも手書きすることなく Fashion-MNIST の MLP がもつ 203,530 個すべてのパラメータを訓練した。自動微分（automatic differentiation）があらゆる勾配を計算してくれたからである。1.5 節ではその一般原理を見た。すなわち、リバースモード自動微分は順伝播（forward pass）を計算グラフ（computational graph）として記録し、スカラー損失から逆向きに連鎖律（chain rule）をたどるのである。

本節では、多層パーセプトロン（multilayer perceptron、MLP）についてそのブラックボックスの中身を開いてみよう。MLP 全体に必要なのは、行列積、ブロードキャストされたバイアスの加算、要素ごとの活性化という **3 つの局所的な行列微分の規則**だけである。これらを順にたどることで、工学上重要な 3 つの問いに答えることができる。

1. どのようにして 1 回の逆伝播（backward pass）は、1 回の順伝播の高々約 $2\times$ の演算コストで数十万個もの偏微分（partial derivative）を計算するのか。
2. なぜ訓練は推論（inference）よりもはるかに多くの GPU メモリを消費し、そのメモリ量はミニバッチサイズ n と深さ K に対して線形に増加するのか。
3. 死んだ ReLU ユニット（dead ReLU unit）や勾配消失・勾配爆発（vanishing/exploding gradients）は、逆伝播の漸化式から機械的にどのように生じるのか。

4.3.1 計算グラフ

4.3.1.1 小さなグラフ

リバースモード微分を最も単純な形で復習するため、3 つのスカラーからなる式 $e = (a + b)\,c$ を $a = 2, b = 1, c = -3$ において考えよう（Figure 4.9）。**順伝播**（左から右）では $u = a + b = 3$ と $e = u\,c = -9$ を計算する。**逆伝播**（右から左）では $\partial e/\partial e = 1$ をシードとして与え、ノードごとに 1 つの局所導関数を適用する。

- 乗算ノード $e = u\,c$ では、$\partial e/\partial u = c = -3$ および $\partial e/\partial c = u = 3$ となる。
- 加算ノード $u = a + b$ では、$\partial u/\partial a = \partial u/\partial b = 1$ であるため、$\partial e/\partial a = \partial e/\partial b = -3$ となる。

ここにすでに表れている 2 つの普遍的な規則に注目しよう。(i) 加算ノードは入力された勾配をそのまま各被加数へコピーするのに対し、乗算ノードは入力された勾配に**他方**の因子を掛ける（そのため順伝播で u と c をキャッシュしておく必要がある）。(ii) 逆伝播はグラフを**逆トポロジカル順序**（reverse topological order）で走査する。

4.3.1.2 MLP のグラフ

次に、隠れ層（hidden layer）を 1 つもつ MLP（Figure 4.10）をミニバッチ $\mathbf{X} \in \mathbb{R}^{n \times d}$ 上で考えよう。パラメータを $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$、$\mathbf{b}^{(1)} \in \mathbb{R}^{h}$、$\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$、$\mathbf{b}^{(2)} \in \mathbb{R}^{q}$ とし、ℓ_2 重み減衰（weight decay）ペナルティを $s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2\right)$ とすると、次のようになる。

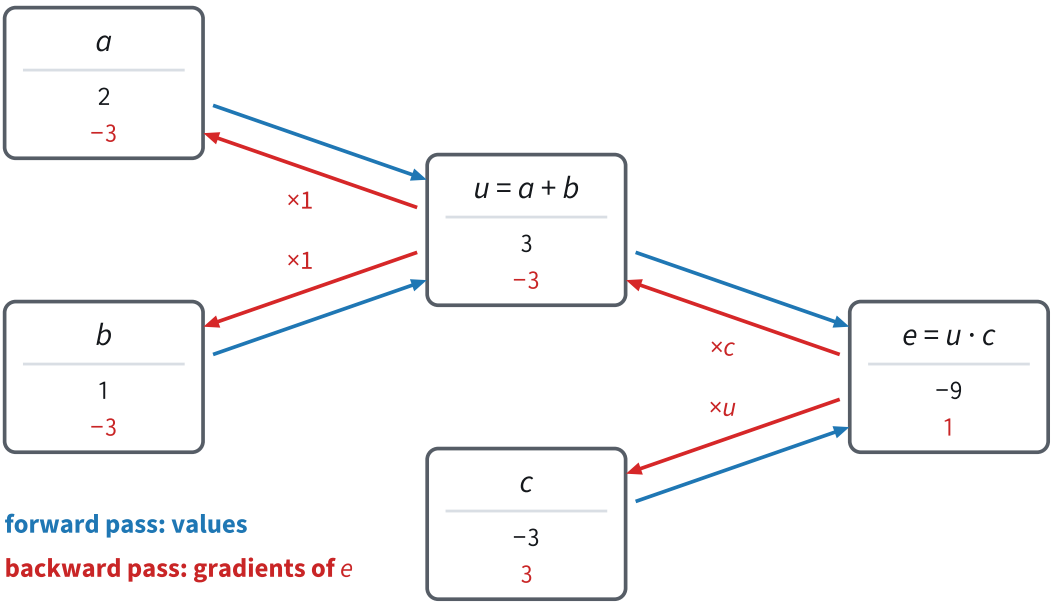


Figure 4.9: $e = (a + b)\,c$ の $a = 2, b = 1, c = -3$ における計算グラフ。順伝播の値（黒）とリバースモードの導関数 $\partial e/\partial(\cdot)$（赤）を示す。

$$
\begin{aligned}
\mathbf{Z} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\
\mathbf{H} &= \phi(\mathbf{Z}), \\
\mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \\
J &= L(\mathbf{O}, \mathbf{Y}) + s,
\end{aligned}
\tag{4.7}
$$

ここで、L はミニバッチ上の平均交差エントロピー損失（cross-entropy loss）であり、$\|\mathbf{W}\|_{\mathrm{F}}^2 = \sum_{r,j} w_{rj}^2$ は二乗フロベニウスノルムである。

4.3.2 MLP における誤差逆伝播法

4.3.2.1 行列に対する局所規則

任意の行列 $\mathbf{A}$ に対し、$\partial J/\partial \mathbf{A}$ は偏微分の行列 $(\partial J/\partial a_{ij})$ を表し、$\mathbf{A}$ と**まったく同じ形状**をもつものとする。3 つの局所的なベクトル・ヤコビ行列規則（34.4.3.2 節）によって、MLP のすべての層を扱うことができる。

1. **行列積（$\mathbf{A} = \mathbf{X}\mathbf{W}$、ただし $\mathbf{X} \in \mathbb{R}^{n\times d}, \mathbf{W} \in \mathbb{R}^{d\times h}$）：**上流の勾配 $\mathbf{G} = \partial J/\partial \mathbf{A} \in \mathbb{R}^{n\times h}$ が与えられたとき、$a_{ij} = \sum_r x_{ir} w_{rj}$ であるから、連鎖律より $\partial J/\partial w_{rj} = \sum_i x_{ir} g_{ij}$ および $\partial J/\partial x_{ir} = \sum_j g_{ij} w_{rj}$ が得られる。行列形式で書けば次のようになる。

$$
\frac{\partial J}{\partial \mathbf{W}} = \mathbf{X}^{\top}\mathbf{G}, \qquad \frac{\partial J}{\partial \mathbf{X}} = \mathbf{G}\,\mathbf{W}^{\top}.
\tag{4.8}
$$

   形状がぴったりかみ合うことに注目しよう。$\mathbf{X}^{\top}\mathbf{G}$ は $(d\times n)(n\times h) = d\times h$（$\mathbf{W}$ と一致）となり、$\mathbf{G}\mathbf{W}^{\top}$ は $(n\times h)(h\times d) = n\times d$（$\mathbf{X}$ と一致）となる。スカラーの乗算とまった

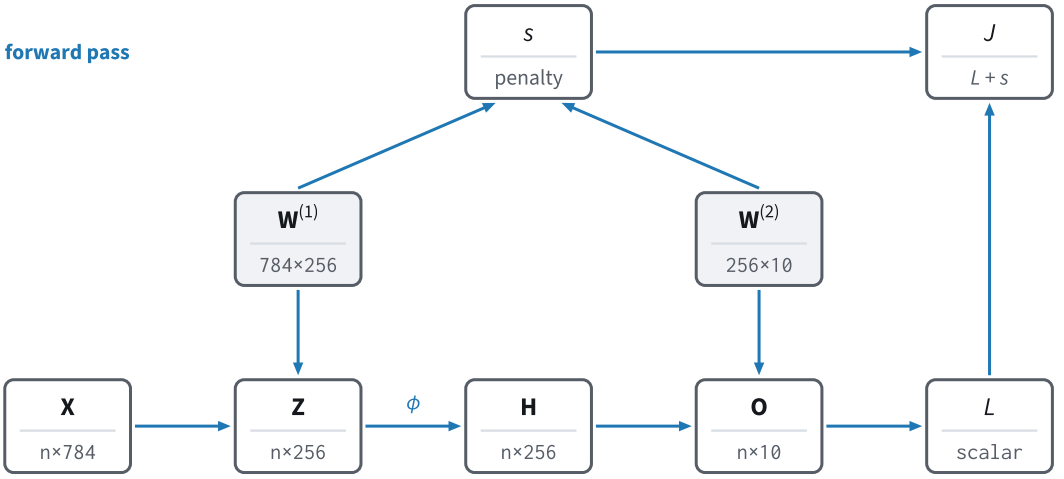


Figure 4.10: 重み減衰を伴う 1 隠れ層 MLP（$d = 784, h = 256, q = 10$）における順伝播の計算グラフ。パラメータノードは網掛けされており、各重み行列はそのアフィン層と正則化ペナルティ s の両方に入力される。

く同様に、$\partial J/\partial \mathbf{W}$ の計算にはキャッシュされた入力 $\mathbf{X}$ が必要であり、一方で $\partial J/\partial \mathbf{X}$ の計算は $\mathbf{W}^\top$ を通じて誤差信号を逆方向へ伝播させる。

2. **ブロードキャストされたバイアスの加算（$\mathbf{Z} = \mathbf{A} + \mathbf{b}$、ただし $\mathbf{b} \in \mathbb{R}^h$）：** b_j はすべての行 i にわたって列 j に加算されるため、$\partial J/\partial \mathbf{A} = \mathbf{G}$ は変化せずそのまま通過し、一方で $\partial J/\partial \mathbf{b} = \mathbf{G}^\top \mathbf{1}_n$ は $\mathbf{G}$ を n 個のデータ例（行）にわたって合計する。
3. **要素ごとの活性化（$\mathbf{H} = \phi(\mathbf{Z})$）：** $h_{ij} = \phi(z_{ij})$ は成分 (i, j) のみに依存するため、上流の勾配は局所的な活性化の傾きによって成分ごとにゲートされる（アダマール積 $\odot$）。

$$\frac{\partial J}{\partial \mathbf{Z}} = \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}).$$

4.3.2.2 逆伝播

$\partial J/\partial \mathbf{O} = \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y})$（3.1.2.2 節参照。ここで $\hat{\mathbf{Y}} = \mathrm{softmax}(\mathbf{O})$ であり、$\mathbf{Y} \in \{0, 1\}^{n \times q}$ はワンホットラベル行列である）および $\partial s/\partial \mathbf{W}^{(k)} = \lambda \mathbf{W}^{(k)}$ から出発し、3 つの規則を出力から入力へ向かって逆トポロジカル順序で適用すると、MLP の完全な逆伝播が得られる。

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{O}} &= \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y}), \\
\frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \mathbf{H}^\top \frac{\partial J}{\partial \mathbf{O}} + \lambda \mathbf{W}^{(2)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(2)}} = \left(\frac{\partial J}{\partial \mathbf{O}}\right)^\top \mathbf{1}, \\
\frac{\partial J}{\partial \mathbf{H}} &= \frac{\partial J}{\partial \mathbf{O}} \mathbf{W}^{(2)\top}, \\
\frac{\partial J}{\partial \mathbf{Z}} &= \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}), \\
\frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \mathbf{X}^\top \frac{\partial J}{\partial \mathbf{Z}} + \lambda \mathbf{W}^{(1)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(1)}} = \left(\frac{\partial J}{\partial \mathbf{Z}}\right)^\top \mathbf{1}.
\end{aligned} \tag{4.9}$$

一般の K 層 MLP（$\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}, \mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$）では、各層 $k = K-1, \ldots, 1$ がまったく同じ 2 ステップのモジュールを実行する。

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{Z}^{(k)}} &= \left(\frac{\partial J}{\partial \mathbf{Z}^{(k+1)}}\mathbf{W}^{(k+1)\top}\right) \odot \phi'(\mathbf{Z}^{(k)}),\\ \frac{\partial J}{\partial \mathbf{W}^{(k)}} &= \mathbf{H}^{(k-1)\top}\frac{\partial J}{\partial \mathbf{Z}^{(k)}} + \lambda \mathbf{W}^{(k)}.\end{aligned}$$

4.3.2.3 計算例

Equation 4.9 を、ごく小さなネットワーク（$n = 1, d = h = 2, q = 1$、ReLU 活性化、バイアスは 0、$\lambda = 0$、二乗損失（squared loss）$L = \frac{1}{2}(o-y)^2$）上で手計算により追ってみよう。

$$\mathbf{X} = \begin{bmatrix}1 & 2\end{bmatrix}, \quad \mathbf{W}^{(1)} = \begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix}2\\ -1\end{bmatrix}, \quad y = 0. \tag{4.10}$$

順伝播（Figure 4.11）では、$\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} = \begin{bmatrix}-1 & 2\end{bmatrix}$、$\mathbf{H} = \mathrm{ReLU}(\mathbf{Z}) = \begin{bmatrix}0 & 2\end{bmatrix}$（ユニット 1 は非活性）、$o = \mathbf{H}\mathbf{W}^{(2)} = -2$、$L = \frac{1}{2}(-2)^2 = 2$ となる。逆伝播では、$\partial L/\partial o = o - y = -2$ であり、各勾配は次のようになる。

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(2)}} &= \mathbf{H}^\top \frac{\partial L}{\partial o} = \begin{bmatrix}0\\ 2\end{bmatrix}(-2) = \begin{bmatrix}0\\ -4\end{bmatrix},\\ \frac{\partial L}{\partial \mathbf{H}} &= \frac{\partial L}{\partial o}\mathbf{W}^{(2)\top} = \begin{bmatrix}-4 & 2\end{bmatrix},\\ \frac{\partial L}{\partial \mathbf{Z}} &= \begin{bmatrix}-4 & 2\end{bmatrix} \odot \begin{bmatrix}0 & 1\end{bmatrix} = \begin{bmatrix}0 & 2\end{bmatrix},\\ \frac{\partial L}{\partial \mathbf{W}^{(1)}} &= \mathbf{X}^\top \frac{\partial L}{\partial \mathbf{Z}} = \begin{bmatrix}1\\ 2\end{bmatrix}\begin{bmatrix}0 & 2\end{bmatrix} = \begin{bmatrix}0 & 2\\ 0 & 4\end{bmatrix}.\end{aligned}$$

$\partial L/\partial \mathbf{W}^{(1)}$ の 1 列目がすべて 0 になっていることに注目しよう。$z_1 = -1 < 0$ であるため、$\phi'(z_1) = 0$ によって上流の勾配 $\partial L/\partial h_1 = -4$ が遮断され、ユニット 1 へ入力されるどの重みにも届かないからである。

4.3.2.4 ミニバッチ上での逆伝播の検証

Equation 4.9 を `mlp_backward` として実装し、64 ビット浮動小数点（floating point）を用いてランダムな 256×784 のミニバッチ（$h = 256, q = 10, \lambda = 10^{-3}$）上で各フレームワークの自動微分と比較して検証しよう。

```
def mlp_backward(X, Y, W1, b1, W2, b2, lambd):
    """Gradients of J = cross-entropy + weight decay, by backprop."""
    Z = X @ W1 + b1  # forward pass, keeping X, Z, H, and Y_hat
    H = torch.relu(Z)
    Y_hat = torch.softmax(H @ W2 + b2, dim=1)
    G_O = (Y_hat - Y) / X.shape[0]  # dJ/dO
    dW2, db2 = H.T @ G_O + lambd * W2, G_O.sum(dim=0)
    G_Z = (G_O @ W2.T) * (Z > 0)  # dJ/dH, then dJ/dZ
    dW1, db1 = X.T @ G_Z + lambd * W1, G_Z.sum(dim=0)
```

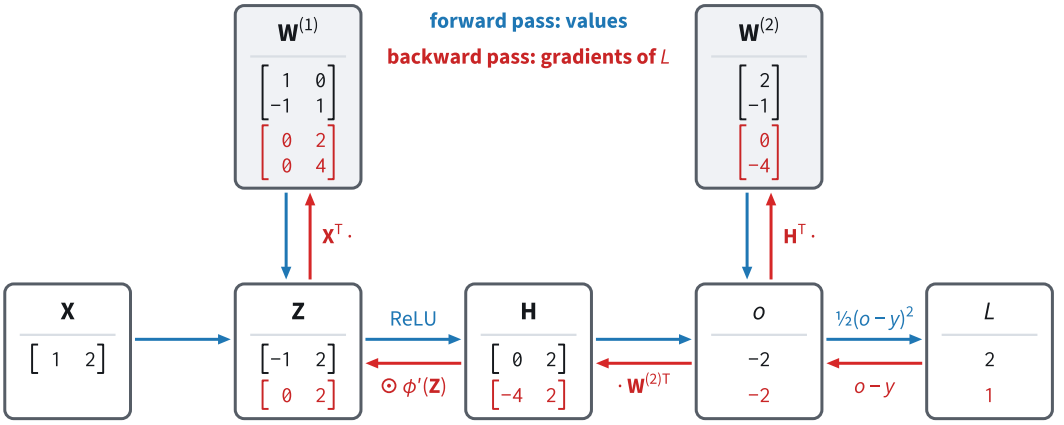


Figure 4.11: 計算例（Equation 4.10）における順伝播の値（黒）と逆伝播の勾配（赤）。$z_1 = -1 < 0$ であるため、ReLU ゲートによって $\partial L/\partial z_1$ および $\partial L/\partial \mathbf{W}^{(1)}$ の 1 列目が 0 になる。

```
    return dW1, db1, dW2, db2
```

256 個のデータ例からなるランダムなミニバッチを、4.2 節とまったく同じ次元および初期化（initialization）に $\lambda = 10^{-3}$ を加えた設定で生成し、`mlp_backward` を各フレームワーク組み込みの自動微分と比較しよう。勾配チェックの定石に従い、64 ビット浮動小数点（丸め誤差の下限が約 10^{-16} と小さいため、わずかな代数的バグも確実に検出できる）で実行し、パラメータテンソルごとに最大の相対差を出力する。

```
n, d, h, q, lambd = 256, 784, 256, 10, 1e-3
f64 = torch.float64  # 64-bit floating point throughout the check
X, y = torch.randn(n, d, dtype=f64), torch.randint(0, q, (n,))
params = [torch.randn(d, h, dtype=f64) * 0.01, torch.zeros(h, dtype=f64),
          torch.randn(h, q, dtype=f64) * 0.01, torch.zeros(q, dtype=f64)]
for p in params:
    p.requires_grad_(True)
W1, b1, W2, b2 = params
J = (F.cross_entropy(torch.relu(X @ W1 + b1) @ W2 + b2, y)
     + lambd / 2 * ((W1 ** 2).sum() + (W2 ** 2).sum()))
J.backward()  # automatic differentiation
with torch.no_grad():
    grads = mlp_backward(X, F.one_hot(y, q).to(f64), *params, lambd)
for name, p, g in zip(['W1', 'b1', 'W2', 'b2'], params, grads):
    rel = (p.grad - g).abs().max() / p.grad.abs().max()
    print(f'{name}: relative difference {float(rel):.1e}')
```

```
W1: relative difference 4.4e-16
b1: relative difference 2.9e-16
W2: relative difference 3.8e-16
b2: relative difference 2.1e-16
```

すべてのパラメータ勾配は自動微分の結果と相対誤差 10^{-15} 以内で一致しており、これは倍精度（double precision）のマシンイプシロン 2.2×10^{-16} の数倍程度にすぎない。

4.3.3 メモリと計算量

4.3.3.1 保存される活性化値とパラメータメモリ

Equation 4.9 を見ると、逆伝播において $\partial J/\partial \mathbf{W}^{(2)}$ と $\partial J/\partial \mathbf{W}^{(1)}$ を計算するには、順伝播で $\mathbf{X}$、$\mathbf{H}$（ReLU の場合はマスク $\mathbf{Z} > 0$ の情報も兼ねる）、および $\hat{\mathbf{Y}}$ を保持しておく必要がある。Fashion-MNIST の MLP（$d = 784, h = 256, q = 10$）においてミニバッチサイズを $n = 256$ とすると、キャッシュされたこれらのテンソルが保持する要素数は次のようになる。

$$256 \times 784 + 256 \times 256 + 256 \times 10 = 268{,}800 \text{ floats} \approx 1.1 \text{ MB}, \tag{4.11}$$

これはすでに 203,530 個のパラメータ（≈ 0.8 MB）を上回っている。K 層のネットワークでは、活性化メモリ（activation memory）はバッチサイズ n と深さ K の**両方**に対して線形に増大するのに対し、推論では次の層を計算した直後に各層の活性化値を破棄できる。

4.3.3.2 演算コスト

順伝播では、各データ例あたり $dh + hq = 784 \times 256 + 256 \times 10 = 203{,}264$ 回の積和演算（multiply-add）を実行する。逆伝播では、内部層の各行列積 $\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}$ から同じサイズの逆方向の行列積が **2 つ**（重みに対する $\mathbf{H}^{(k-1)\top}\mathbf{G}$ と下層の活性化値に対する $\mathbf{G}\mathbf{W}^{(k)\top}$）生じる一方、最初の層で必要なのは $\mathbf{X}^\top \mathbf{G}_Z$ のみである（$\partial J/\partial \mathbf{X}$ は不要なため）。したがって、この 2 層 MLP の逆伝播は $dh + 2hq = 205{,}824$ 回の積和演算で済み、順伝播とほぼ同じである。計算量の大部分を最初の層が占めており、そこでは逆方向の積が 1 回しか必要ないからである。深層ネットワークではほぼすべての層が内部層となるため、逆伝播には順伝播の $\sim 2\times$ のコストがかかり、訓練 1 ステップ全体では順伝播による推論のおよそ $3\times$ となる。いずれにせよ、203,530 個すべての導関数が追加の順伝播 1〜2 回分のコストで求まるのに対し、片側有限差分（finite difference）では**たった 1 つ**のスカラーパラメータにつき 1 回分の追加の順伝播を費やすことになる。

4.3.4 まとめ

多層パーセプトロンにおける誤差逆伝播法（backpropagation）は、逆トポロジカル順序で適用される 3 つの局所的な行列規則に帰着する。すなわち、$\partial J/\partial \mathbf{W} = \mathbf{X}^\top \mathbf{G}$ と $\partial J/\partial \mathbf{X} = \mathbf{G}\mathbf{W}^\top$（行列積 $\mathbf{A} = \mathbf{X}\mathbf{W}$ に対して）、行方向の和 $\mathbf{G}^\top \mathbf{1}$（ブロードキャストされたバイアスに対して）、および要素ごとのゲート処理 $\mathbf{G} \odot \phi'(\mathbf{Z})$（活性化に対して）であり、パラメータがデータ損失と重み減衰の両方に入力される箇所では流入する勾配を合算する。$\mathbf{X}^\top \mathbf{G}$ は各層への順伝播の入力に依存するため、訓練時には順方向スイープ全体にわたって中間の活性化値をキャッシュする必要があり、バッチサイズと深さに対して線形に増大する $O(nKh)$ のメモリを要する。その見返りとして、1 回の逆方向スイープだけで、1 回の順伝播の高々約 2 倍の FLOPs ですべてのパラメータ勾配を計算できる。

4.3.5 演習問題

1. **層のヤコビ行列**。スカラー J の $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ に関する勾配は、$\mathbf{W}^{(2)}$ と同じ形状をもつ。一方、行列 $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} \in \mathbb{R}^{n \times q}$ の $\mathbf{W}^{(2)}$ に関する導関数はより大きな対象であり、4 つの添字のあらゆる組み合わせに対して 1 つずつ成分 $\partial o_{ij}/\partial w_{rt}$ をもつ。
    1. $\partial \mathbf{O}/\partial \mathbf{W}^{(2)}$ の成分数はいくつか。また、そのうち非ゼロになり得るものはいくつあるか。
    2. 連鎖律 $\partial J/\partial w_{rt} = \sum_{i,j} (\partial J/\partial o_{ij})(\partial o_{ij}/\partial w_{rt})$ を計算し、$\mathbf{H}^\top \partial J/\partial \mathbf{O}$ を導出せよ。

3. $\partial \mathbf{O}/\partial \mathbf{W}^{(2)}$ は、4.2 節の MLP で $n = 256$ としたとき、32 ビット浮動小数点でどれだけのメモリを消費するか。また、誤差逆伝播法が決してこれを構成しない理由を説明せよ。

2. **より深いネットワークのメモリ**。d 個の入力、q 個の出力、および K 個の層（そのうち $K-1$ 個の隠れ層は幅 h の ReLU 層）をもち、n 個のデータ例からなるミニバッチで訓練される MLP を考える。

   1. 本文中の K 層に対するステップを用いて、逆伝播が読み取る値を列挙し、その個数を求めよ。また、パラメータの個数を求めよ。
   2. 予測時には、次の層が使い終わり次第、各値を破棄できる。予測が任意の時点で保持する値の最大個数を求めよ。
   3. $d = 784$、$h = 256$、$q = 10$ のとき、保存される値の個数がパラメータ数を上回る最小のミニバッチサイズを、$K = 2$ および $K = 11$ のそれぞれについて求めよ。

3. [code] **勾配チェック**。自動微分が登場する以前、手書きの逆伝播は有限差分と比較して検証されていた。

   1. 1 つの成分 w を $\mathbf{W}^{(1)}$ から選び、`mlp_backward` による勾配を、中心差分 $\big(J(w+\epsilon) - J(w-\epsilon)\big)/(2\epsilon)$ ($\epsilon = 10^{-1}, 10^{-2}, \ldots, 10^{-7}$) と 32 ビット浮動小数点および 64 ビット浮動小数点のそれぞれで比較せよ。相対誤差を ϵ に対して両対数軸でプロットせよ。
   2. 曲線の形状を説明せよ。ϵ を小さくするとき減少するのはどの誤差で、増加するのはどの誤差か。また、それぞれの精度において両者が釣り合うのはどこか。
   3. 活性化前の値が 0 から ϵ 以内にあるとき、勾配が正しくてもこのチェックが失敗することがある。その理由と回避方法を説明せよ。

4. **フォワードモード**。誤差逆伝播法は出力に $\partial J/\partial J = 1$ をシードとして与え、グラフを逆方向にスイープする。これに対し、フォワードモード微分は選んだ 1 つのスカラーパラメータに $\partial w/\partial w = 1$（対象パラメータ w に対する微分）をシードとして与え、順方向にスイープする。各ノードは、入力の導関数から自身の値の w に関する導関数を計算する。

   1. フォワードモードを用いて、$\partial L/\partial \mathbf{W}^{(1)}$ を Equation 4.10 のネットワークについて再計算せよ（$\mathbf{W}^{(1)}$ の成分ごとに 1 回のスイープを行うこと）。
   2. このネットワークの 6 個すべての重み勾配を得るには、各モードで何回ずつスイープが必要か。リバースモードでは 1 回しか計算しない積のうち、フォワードモードが毎回のスイープで繰り返し計算するのはどの積か。

   Simon Prince『Understanding Deep Learning』問題 7.12 および 7.13 を改変。

5. [extended] **自動微分エンジン**。●小型の自動微分エンジンを構築し、それを用いて Equation 4.10 のネットワークの勾配を再計算せよ。

   1. 各計算結果について、その入力と生成元の演算（計算グラフ）を記録するスカラーの `Value` クラスを作成し、`+`、`*`、`relu` をサポートせよ（**ヒント：** `__add__` と `__mul__` を実装し、それぞれが親ノードへの参照と勾配を 1 ステップ伝播させる小さな関数を保持する新しい `Value` を返すようにする）。
   2. `backward()` を実装せよ。グラフをトポロジカルソートし、出力の勾配に 1 をシードとして与え、逆順にノードをたどって各ノードから親ノードへ勾配を渡すようにせよ。1 つの値が複数回使われる場合に勾配が**累積**される（`+=` であり、`=` ではない）ようにすること。
   3. 作成したエンジンを用いて Figure 4.9 の計算グラフと Equation 4.10 の計算例を再計算し（行列積はスカラー演算に展開せよ）、得られた勾配を本節で導出した勾配と比較せよ。

4. スカラー入力 x から $v_i = f_i(x)$（$i = 1,2,3$）、中間値 $m = v_1 + v_2 + v_3$、値 $p_j = g_j(m)$（$j = 1,2,3$）、および損失 $L = p_1 + p_2 + p_3$ が順に生成される連鎖を考える。$\alpha_i = f_i'(x)$、$\beta_j = g_j'(m)$ とおく。経路上の和としての連鎖律は $3 \times 3 = 9$ 本の経路（x から L への経路）を列挙し、各経路が積 $\alpha_i \beta_j$ を寄与する。これに対し、作成したエンジンは $(\alpha_1 + \alpha_2 + \alpha_3)(\beta_1 + \beta_2 + \beta_3)$ を計算すること（m で勾配を 1 回だけ累積することによる）を示し、この共有によって深層ネットワークでも誤差逆伝播法が現実的な計算量に収まる理由を説明せよ。

Andrej Karpathy の micrograd（`micrograd/engine.py`）を改変。

4.4 数値安定性と初期化

4.2 節では、隠れ層が 1 層の MLP の重みを $\mathcal{N}(0, 0.01^2)$ で初期化すると最初の 1 エポックは進みが鈍かったものの、その後ネットワークは持ち直して 87% の正解率に到達した。では、幅 256 の隠れ層を **10** 層積み重ね、同じ $\mathcal{N}(0, 0.01^2)$ で初期化するとどうなるだろうか。10 エポック経過しても損失は $\log 10 \approx 2.30$ からまったく動かず、正解率は当て推量と同じ 10% にとどまる。逆に、重みを $\mathcal{N}(0, 1)$ で初期化すると、活性化値（activation）と勾配が天文学的に大きくなり、たった 1 回の SGD 更新で重みが大きく飛びすぎて、次の順伝播で `inf` や `NaN` へとオーバーフローしてしまう。

4.3 節で見たように、K 層のネットワークでは、順伝播の活性化値も逆伝播の勾配も、重みと傾きからなる K 個の行列の積を通過する。各層が信号の分散を保ち、その乗数を 1 付近に維持しない限り、信号は指数関数的にゼロへ**消失**するか、無限大へ**爆発**するかのいずれかになる。本節では、層ごとの分散の伝播を定量化して **Xavier** 初期化と **He** 初期化を導出し、Fashion-MNIST 上の 10 層 MLP で検証したうえで、隠れユニット間の置換対称性（permutation symmetry）を破るために初期重みが**ランダム**でなければならない理由を証明する。

4.4.1 勾配消失と勾配爆発

4.4.1.1 層にわたる積

K 層の MLP において、隠れ層が $\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}$ および $\mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$（幅 h）で与えられるとき、4.3 節の逆伝播の漸化式は、活性化値の勾配を次の式によって 1 層下へ伝播させる。

$$\frac{\partial L}{\partial \mathbf{H}^{(k-1)}} = \Big(\frac{\partial L}{\partial \mathbf{H}^{(k)}} \odot \phi'\big(\mathbf{Z}^{(k)}\big) \Big)\, \mathbf{W}^{(k)\top}. \tag{4.12}$$

Equation 4.12 を最上位の隠れ層 $K-1$ から第 k 層まで展開すると、第 k 層における勾配は、行列と傾きからなる $K-1-k$ 個の因子の積として表される。

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{H}^{(k)}} = \frac{\partial L}{\partial \mathbf{H}^{(K-1)}} \underbrace{\odot\, \phi'\big(\mathbf{Z}^{(K-1)}\big)\, \mathbf{W}^{(K-1)\top}}_{\text{layer } K-1} \\ \cdots \underbrace{\odot\, \phi'\big(\mathbf{Z}^{(k+1)}\big)\, \mathbf{W}^{(k+1)\top}}_{\text{layer } k+1}. \end{aligned} \tag{4.13}$$

1 つのデータ例に対する上流の勾配行を $\mathbf{g}'$、下流の勾配行を $\mathbf{g}$ とし、幅 h の層の重み w_{ij} が平均 0、分散 τ^2 の独立同分布から抽出されるとする。初期化時点において重み、傾き、上流の勾配を近

似的に独立とみなせば、各成分 $g_i = \sum_{j=1}^{h} w_{ij}\,\phi'(z_j)\,g'_j$ の二乗の期待値は次の式で与えられる。

$$\begin{aligned}\mathbb{E}[g_i^2] &= \sum_{j=1}^{h} \mathbb{E}[w_{ij}^2]\,\mathbb{E}[\phi'(z_j)^2]\,\mathbb{E}[(g'_j)^2] \\ &= \underbrace{h\,\tau^2\,\mathbb{E}[\phi'(z)^2]}_{\text{per-layer factor}}\;\mathbb{E}[(g'_j)^2].\end{aligned} \tag{4.14}$$

t 個の層を通過すると、勾配の二乗平均（mean square）は $\left(h\,\tau^2\,\mathbb{E}[\phi'(z)^2]\right)^t$ 倍になる。活性化前の値（pre-activation）の分布が対称で活性化関数が ReLU の場合、$\phi'(z) \in \{0, 1\}$ は半分の確率で 1 になるため、$\mathbb{E}[\phi'(z)^2] = 1/2$ となり、層ごとの乗数は $\frac{1}{2}h\tau^2$ となる。幅 $h = 256$ では次のようになる。

- $\tau = 0.01$ の場合、乗数は $256 \times 10^{-4}/2 = 0.0128$ となり、9 段の伝播で勾配の二乗平均は $0.0128^9 \approx 10^{-17}$ 倍にまで縮小し、**勾配消失**（vanishing gradients）が生じる。
- $\tau = 1$ の場合、乗数は $256 \times 1/2 = 128$ となり、9 段の伝播で勾配の二乗平均は $128^9 \approx 10^{19}$ 倍にまで膨張し、**勾配爆発**（exploding gradients）が生じる。
- シグモイド活性化関数（$\sigma'(z) \le 1/4$ より $\mathbb{E}[\sigma'(z)^2] \le 1/16$）の場合、たとえ $h\tau^2 = 1$ であっても、各層を通るたびに勾配の二乗平均は少なくとも 16 分の 1 に縮小する (Bengio et al. 1994)。

`moments` を用いて、初期化時点における 10 層・幅 256 のネットワーク上でこれら 3 つの予測を直接確認してみよう。

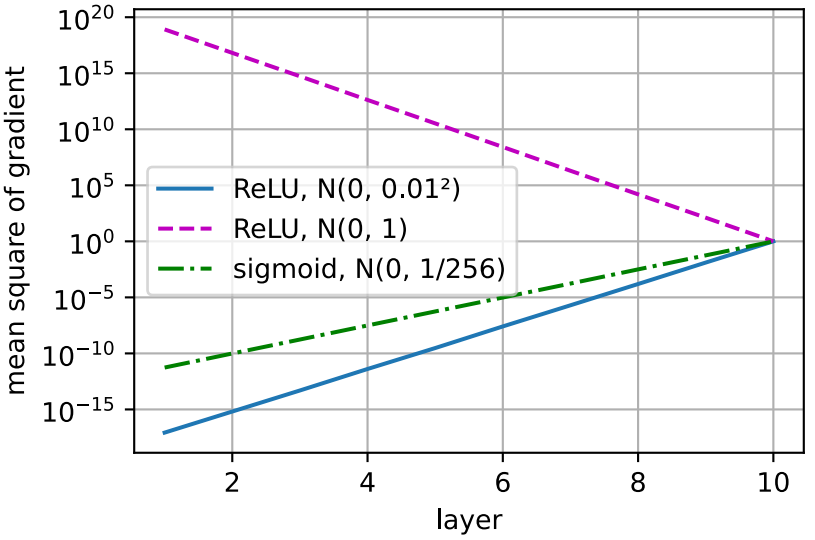
```
def moments(phi, dphi, scale, depth=10, width=256, n=256):
    """Mean squares of the activations and of their gradients, per layer."""
    Ws = [scale * torch.randn(width, width) for _ in range(depth)]
    H, Zs, acts = torch.randn(n, width), [], []
    for W in Ws:  # forward pass, keeping the pre-activations
        Zs.append(H @ W)
        H = phi(Zs[-1])
        acts.append(float((H ** 2).mean()))
    G, grads = torch.randn(n, width), []  # dL/dH at the top layer
    for W, Z in zip(reversed(Ws), reversed(Zs)):  # backward pass
        grads.insert(0, float((G ** 2).mean()))
        G = (G * dphi(Z)) @ W.T  # dL/dH one layer down
    return acts, grads

relu = (torch.relu, lambda Z: (Z > 0).float())
sigmoid = (torch.sigmoid, lambda Z: torch.sigmoid(Z) * (1 - torch.sigmoid(Z)))
```

3 つの設定を比較しよう。すなわち、$\tau = 0.01$ の ReLU（4.2 節と同じ設定）、$\tau = 1$ の ReLU、そして $\tau^2 = 1/256$ のシグモイド（$h\tau^2 = 1$ として活性化関数のみによる減衰を切り出した設定）である。

```
runs = {'ReLU, N(0, 0.01²)': moments(*relu, 0.01),
        'ReLU, N(0, 1)': moments(*relu, 1.0),
        'sigmoid, N(0, 1/256)': moments(*sigmoid, 1 / 16)}
for name, (acts, grads) in runs.items():
    print(f'{name}: gradient mean square at layer 1: {grads[0]:.1e}')
```

```
d2l.plot(list(range(1, 11)), [grads for acts, grads in runs.values()],
         'layer', 'mean square of gradient', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
ReLU, N(0, 0.01²): gradient mean square at layer 1: 8.5e-18
ReLU, N(0, 1): gradient mean square at layer 1: 8.2e+18
sigmoid, N(0, 1/256): gradient mean square at layer 1: 5.5e-12
```



縦軸を対数スケールにとると、3 本の曲線はいずれも直線となり、その傾きは解析的に導いた乗数 Equation 4.14 と一致する（第 1 層でそれぞれ 10^{-17}、10^{19}、5×10^{-12}）。

4.4.2 パラメータの初期化

4.4.2.1 Xavier 初期化

初期重みの分散 τ^2 をどのように選べば、すべての層の乗数を 1 にできるだろうか。線形層 $o_j = \sum_{i=1}^{n_{\text{in}}} x_i w_{ij}$ を考えよう。この層は n_{in} 個の入力（層の**ファンイン**）と n_{out} 個の出力（**ファンアウト**）を持ち、重み w_{ij} は平均 0、分散 τ^2 の独立同分布に従い、入力 x_i は互いに無相関で平均 0、分散 γ^2 を持つとする。順伝播では $\mathbb{E}[o_j] = 0$ であり、分散は次のようになる。

$$\text{Var}[o_j] = \mathbb{E}[o_j^2] = \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[x_i^2]\,\mathbb{E}[w_{ij}^2] = n_{\text{in}}\,\tau^2\gamma^2.$$

層をまたいで順伝播の分散を保つには $n_{\text{in}}\tau^2 = 1$ が必要である。また、傾きを 1 とした Equation 4.14 より、逆伝播の勾配の分散を保つには $n_{\text{out}}\tau^2 = 1$ が必要となる。$n_{\text{in}} \neq n_{\text{out}}$ のとき、**Xavier 初期化**（または **Glorot 初期化** (Glorot and Bengio 2010)）はこれら 2 つのファン数を次のように平均する。

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\,\tau^2 = 1 \quad \Longleftrightarrow \quad \tau^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}, \tag{4.15}$$

すなわち、重みを正規分布 $\mathcal{N}\left(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$、またはそれと分散を揃えた一様区間 $U\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right)$ から抽出する。Xavier 初期化は、線形層や tanh のようなゼロ中心で傾き 1 の活性化関数に対する標準的な選択肢である。

4.4.2.2 He 初期化

ReLU 層では、平均 0 で対称な活性化前の値 z の負の領域がゼロに切り捨てられるため、その二乗平均は半分になる（$\mathbb{E}[\text{ReLU}(z)^2] = \frac{1}{2}\mathbb{E}[z^2]$）。重みが独立で平均 0 であれば、$\mathbb{E}[o_j^2] = n_{\text{in}}\tau^2\,\mathbb{E}[x_i^2]$ は入力 $x_i = \text{ReLU}(z_i)$ が正の平均を持つ場合にも成り立つため、活性化値の二乗平均 $v^{(k)} = \mathbb{E}[(h_i^{(k)})^2]$ は ReLU 層を通過すると次のように変化する。

$$v^{(k)} = \frac{1}{2}\, n_{\text{in}}\, \tau^2\, v^{(k-1)}. \tag{4.16}$$

したがって、深さにわたって $v^{(k)} = v^{(k-1)}$ を一定に保つには、分散を 2 倍にする必要がある。

$$\tau^2 = \frac{2}{n_{\text{in}}}. \tag{4.17}$$

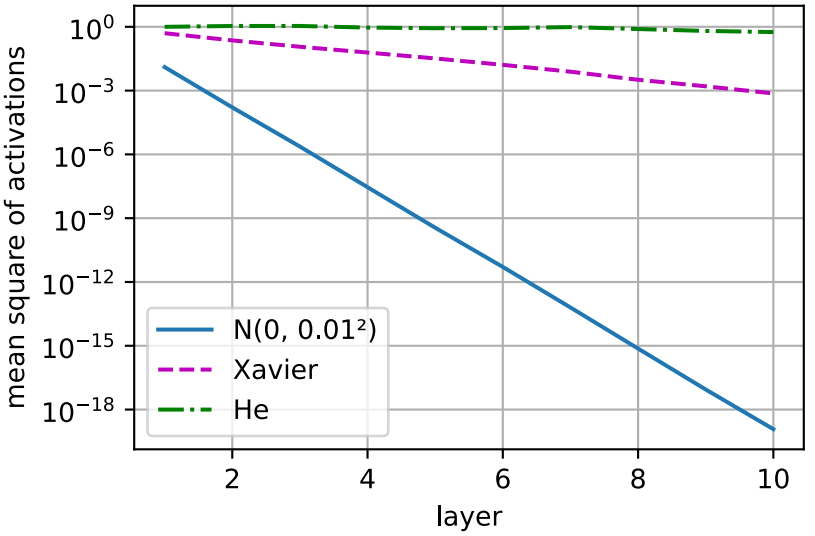
これが **He 初期化**（または **Kaiming 初期化** (He et al. 2015)）であり、ReLU ネットワークにおける標準の手法である（なお、フレームワークの標準的な層では、He 初期化を明示的に指定しない限り、デフォルトが $1/(3n_{\text{in}})$ や $1/n_{\text{in}}$、あるいは Xavier 初期化になっていることが多い）。

4.4.2.3 深さにわたる活性化値、勾配、訓練

まず、$\mathcal{N}(0, 0.01^2)$、Xavier（$\tau^2 = 1/256$）、He（$\tau^2 = 2/256$）を、10 層の合成 ReLU 検証コード `moments(*relu, ...)` で比較してみよう。

```
runs = {'N(0, 0.01²)': moments(*relu, 0.01),
        'Xavier': moments(*relu, (1 / 256) ** 0.5),
        'He': moments(*relu, (2 / 256) ** 0.5)}
for name, (acts, grads) in runs.items():
    print(f'{name}: activations at layer 10: {acts[-1]:.1e}, '
          f'gradient at layer 1: {grads[0]:.1e}')
d2l.plot(list(range(1, 11)), [acts for acts, grads in runs.values()],
         'layer', 'mean square of activations', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
N(0, 0.01²): activations at layer 10: 1.2e-19, gradient at layer 1: 1.2e-17
Xavier: activations at layer 10: 7.4e-04, gradient at layer 1: 1.9e-03
He: activations at layer 10: 5.7e-01, gradient at layer 1: 8.3e-01
```



$\mathcal{N}(0, 0.01^2)$ では第 10 層の活性化値が $\sim 10^{-19}$ まで潰れてしまい、Xavier でも各 ReLU 層で活性化値の二乗平均が半減する（第 10 層までに $\sim 2^{-10} \approx 10^{-3}$）のに対し、He 初期化では活性化値と勾配の両方が $O(1)$ 付近に保たれている点に注目しよう。この分散の伝播が実際の訓練をどのように左右するかを確かめるため、隠れ層 10 層の ReLU 分類器 `DeepMLP` を定義し、層ごとの標準偏差の規則 `std(n_in, n_out)` を引数として渡せるようにする。

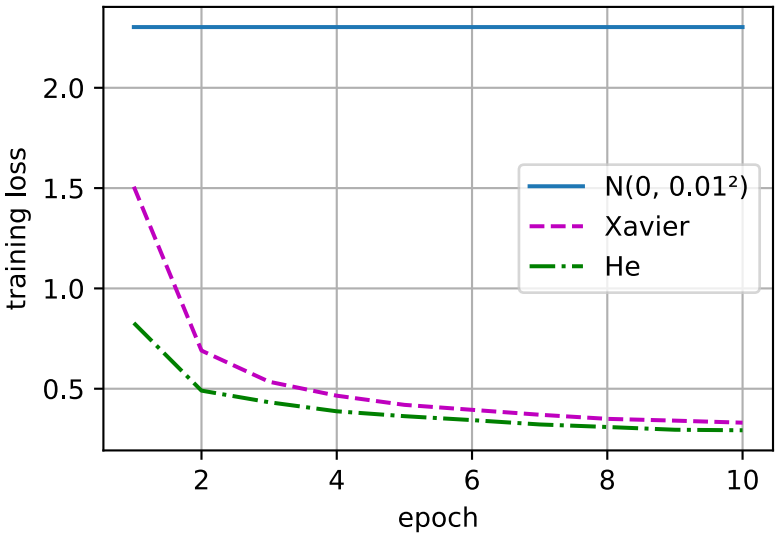
```
class DeepMLP(d2l.Classifier):
    """A deep MLP whose weights have the standard deviation std(n_in, n_out)."""
    def __init__(self, std, lr, depth=10, width=256):
        super().__init__()
        self.save_hyperparameters(ignore=['std'])
        sizes = [784] + [width] * depth + [10]
        layers = [nn.Flatten()]
        for n_in, n_out in zip(sizes[:-1], sizes[1:]):
            layer = nn.Linear(n_in, n_out)
            nn.init.normal_(layer.weight, std=std(n_in, n_out))
            nn.init.zeros_(layer.bias)
            layers += [layer, nn.ReLU()]
        self.net = nn.Sequential(*layers[:-1])  # No ReLU after the logits
```

3 つの実行はすべてまったく同じアーキテクチャとハイパーパラメータを共有しており、異なるのは `std(n_in, n_out)` のみである。これらの訓練曲線と最終的な検証正解率を比較しよう。

```
inits = {'N(0, 0.01²)': lambda n_in, n_out: 0.01,
         'Xavier': lambda n_in, n_out: (2 / (n_in + n_out)) ** 0.5,
         'He': lambda n_in, n_out: (2 / n_in) ** 0.5}
data = d2l.FashionMNIST(batch_size=256)
losses = {}
for name, std in inits.items():
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False
    trainer.fit(DeepMLP(std, lr=0.05), data)
```

```
    losses[name] = trainer.history['train_loss']
    print(f'{name}: validation accuracy {trainer.history["val_acc"][-1]:.2f}')
d2l.plot(list(range(1, 11)), list(losses.values()), 'epoch', 'training loss',
         legend=list(losses), figsize=(4.5, 3))
```

```
N(0, 0.01²): validation accuracy 0.10
Xavier: validation accuracy 0.84
He: validation accuracy 0.87
```



10 エポックにわたる訓練曲線が、先ほどの分散解析とどれほどよく一致しているか見てみよう。

- $\mathcal{N}(0, 0.01^2)$ では、第 10 層の活性化値（$\sim 10^{-19}$）と第 1 層の勾配（$\sim 10^{-17}$）が完全に消失し、訓練損失は $\log 10 \approx 2.30$ に張り付いたまま、検証正解率も 10% にとどまる。
- Xavier では、ReLU 層ごとに信号が 2 分の 1 へと減衰して全体で $2^{-10} \approx 10^{-3}$ 倍に縮小するため、最初の 1 エポックは立ち上がりが遅いものの、その後訓練が回復して 80% を超える正解率に達する。
- He 初期化では、全 10 層にわたって活性化値と勾配の両方がオーダー 1 付近に保たれ、最初のエポックから訓練損失が滑らかに低下する。

もっとも、初期化だけで深さの問題がすべて解決するわけではない。ここまでの分散の議論はステップ 0 におけるネットワークを記述しているにすぎず、ひとたび SGD が重みを動かせば、Equation 4.14 の層ごとの乗数を 1 付近に保ち続ける保証は何もない。そのため、数十層から数百層に及ぶネットワークでは、訓練全体を通して機能する仕組みに頼ることになる。正規化層（normalization layer、7.3 節）は毎ステップ活性化値のスケールを調整し、残差接続（residual connection、7.4 節）は信号と勾配に各層を迂回する経路を与え、勾配クリッピング（gradient clipping、8.4 節）は時折生じる勾配爆発に上限を設ける。適切な初期化は訓練を軌道に乗せるためのものであり、その後の信号のスケールを維持するのはこれらの仕組みである。

4.4.3 隠れユニット間の対称性

なぜ $\mathbf{W}^{(1)}$ のすべての成分を決定論的な定数 $c = \sqrt{2/n_{\text{in}}}$ に設定するのではなく、初期重みを**ランダム**にしなければならないのだろうか。理由は、同一層内の隠れユニットが置換対称だからである。もし $\mathbf{W}^{(1)}$ のすべての列が同一の値から始まり（かつ $\mathbf{W}^{(2)}$ のすべての行も同一の値から始

まり)、そのまま訓練を進めると、どの隠れユニットもまったく同じ順伝播の活性化値を計算し、同じ逆伝播の勾配を受け取り、その後もずっと同じ SGD 更新ステップをたどり続けるため、256 ユニットの層が実質的に 1 個のニューロンへと縮退してしまう。16 ユニットの層を用いて、He のスケールに合わせた定数の重みでは勾配の各列が同一になる（ユニット間の差が `0.00`）のに対し、He のスケールに合わせたランダムな重みでは各ユニットが異なる勾配方向を持つことを確認してみよう。

```
def hidden_grad(W1, W2, X, y):
    """Gradient of the squared loss with respect to the hidden weights."""
    W1 = W1.clone().requires_grad_(True)
    L = ((torch.relu(X @ W1) @ W2 - y) ** 2).mean() / 2
    L.backward()
    return W1.grad

d, h = 20, 16
X, y = torch.randn(64, d), torch.randn(64, 1)
c, c_out = (2 / d) ** 0.5, (1 / h) ** 0.5
inits = {'constant': (torch.full((d, h), c), torch.full((h, 1), c_out)),
         'random': (c * torch.randn(d, h), c_out * torch.randn(h, 1))}
for name, (W1, W2) in inits.items():
    G = hidden_grad(W1, W2, X, y)
    print(f'{name}: largest difference between two hidden units '
          f'{float((G - G[:, :1]).abs().max()):.2f}')
```

```
constant: largest difference between two hidden units 0.00
random: largest difference between two hidden units 0.16
```

4.4.4 まとめ

K 層の MLP（幅 h、初期重みの分散 τ^2）では、各層が活性化値と勾配の二乗平均をおよそ $h\tau^2\,\mathbb{E}[\phi'(z)^2]$ 倍するため、この乗数が 1 付近に保たれない限り、信号は深さに対して指数関数的に消失または爆発する。Xavier 初期化は $\tau^2 = 2/(n_{\text{in}} + n_{\text{out}})$ と設定することで線形層や tanh 層における順伝播と逆伝播の分散のバランスをとり、一方 He 初期化は分散を 2 倍の $\tau^2 = 2/n_{\text{in}}$ とすることで ReLU が負の半直線をゼロに切り捨てる影響を相殺し、深い ReLU 積層でも最初のエポックから安定して訓練できるようにする。また、同一層内の隠れユニットは置換対称であるため、重みは定数にするのではなく独立かつランダムに抽出しなければならない。そうしなければ、すべてのユニットが訓練を通じて同一の勾配と更新を受け続けることになる。さらにステップ 0 以降についても、より深いアーキテクチャでは分散保持の初期化に正規化層や残差接続を組み合わせることで、重みが変化しても信号のスケールを適切に維持する。

4.4.5 演習問題

1. **スペクトルノルムによる上界**。1 つのデータ例について列ベクトルとして表記すると、Equation 4.12 のステップは、勾配 $\mathbf{g}'$（第 k 層より上流）に行列 $\mathbf{M}^{(k)} = \mathbf{W}^{(k)}\,\mathrm{diag}\big(\phi'(\mathbf{z}^{(k)})\big)$ を掛ける操作となる。ここで $\mathrm{diag}(\cdot)$ はベクトルの各成分を対角成分に並べ、他の成分をゼロとした行列を表す。したがって、Equation 4.13 の勾配は、このような $K-1-k$ 個の行列の積と最上位層の勾配との積になる。スペクトルノルム（最大特異値）は劣乗法的であり、$\|\mathbf{AB}\|_2 \le \|\mathbf{A}\|_2\|\mathbf{B}\|_2$ を満たす。
   1. 定義 $\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq 0} \|\mathbf{Ax}\|/\|\mathbf{x}\|$ から劣乗法性を証明せよ。
   2. これを用いて、勾配のノルムを各因子のノルム $\|\mathbf{M}^{(k)}\|_2$ によって上から評価せよ。深さに対してこの上界が幾何級数的に増大しないためには、これらのノルムがどのよう

な条件を満たせばよいか。
3. この評価はあくまで上界にすぎない。前問の条件によって勾配爆発は排除できるが勾配消失は排除できない理由を説明し、さらにその条件を等号で満たすにもかかわらず、積のスペクトルノルムが真に 1 より小さくなる 2 つの 2×2 行列の例を挙げよ。

2. **その他の対称性**。MLP が表す関数を不変に保つパラメータの変化は、隠れユニットの置換だけではない。隠れ層が 1 層の MLP を考え、隠れユニット j の入力重みを $\mathbf{w}_j$、バイアスを b_j、出力重みを $\mathbf{v}_j$ とする。
   1. $\mathrm{ReLU}(\alpha z) = \alpha\,\mathrm{ReLU}(z)$ が $\alpha > 0$ に対して成り立つことを証明せよ。この恒等式を用いて、$(\mathbf{w}_j, b_j, \mathbf{v}_j)$ を $(\alpha\mathbf{w}_j, \alpha b_j, \mathbf{v}_j/\alpha)$ に置き換えても ReLU ネットワークの関数が変化しないことを示せ。
   2. 単一の隠れユニットのパラメータに対する変換であって、tanh ネットワークの関数を不変に保つものを求めよ。
   3. 同一の隠れユニットからなる状態は置換の不動点であり、その不動点から開始した勾配降下法はそこにとどまり続ける。前問の変換によって不変に保たれるユニットのパラメータを求めよ。さらに、tanh ネットワークのすべての重みとバイアスをゼロで初期化すると、勾配降下法はどのステップにおいても出力バイアスしか変化させないことを示せ。

   Simon Prince『Understanding Deep Learning』問題 3.5、4.3、7.15 より改変。

3. **ファンインと平均ファン数の比較**。@He.Zhang.Ren.ea.2015 に従い、K 層の線形ネットワークを考える。第 k 層は n_{k-1} 個の入力を n_k 個の出力へ写し、その重みは分散 τ_k^2 で抽出されるとする。4.4.2.1 節の計算より、第 k 層は順伝播信号の分散を $n_{k-1}\tau_k^2$ 倍し、逆伝播信号の分散を $n_k\tau_k^2$ 倍する。
   1. すべての層で $\tau_k^2 = 1/n_{k-1}$ と設定する。このとき任意の深さにおいて、順伝播の分散が保たれ、逆伝播の分散は全体で n_K/n_0 倍変化することを示せ。
   2. 各層の幅が $100, 400, 100, 400, \ldots$ と交互に並ぶものとし、すべての層で Xavier 初期化を用いるとする。$100 \to 400 \to 100$ という 1 組の層が順伝播の分散を何倍にするか、またその組を 50 回繰り返した後に何倍になるかを計算せよ。
   3. あらゆる幅の系列に対して、順伝播と逆伝播の両方の分散を初期値から深さに依存しない定数倍の範囲内に保つのは、2 つの規則のうちどちらか。

4. **ReLU のもとでの分散**。●4.4.2.2 節の導出では、分散ではなく二乗平均 $\mathbb{E}[\mathrm{ReLU}(z)^2]$ を追跡している。これは、次の層の活性化前の値 $o = \sum_i x_i w_i$ において、入力を $x_i = \mathrm{ReLU}(z_i)$ とし重みが独立で平均 0 であるとき、$\mathrm{Var}[o] = n_{\text{in}}\tau^2\mathbb{E}[x_i^2]$ が x_i の平均によらず成り立つからである。
   1. $z \sim \mathcal{N}(0, \nu^2)$ に対して $\mathbb{E}[\mathrm{ReLU}(z)]$ と $\mathrm{Var}[\mathrm{ReLU}(z)]$ を計算し、分散が $\left(\frac{1}{2} - \frac{1}{2\pi}\right)\nu^2$ となり、$\frac{1}{2}\nu^2$ にはならないことを示せ。
   2. 一見もっともらしい近道として、Xavier の計算で $\mathrm{Var}[x_i]$ を $\mathbb{E}[x_i^2]$ の代わりに代入する方法が考えられる。この近道からは $\tau^2 = 2/\big(n_{\text{in}}(1 - 1/\pi)\big)$ が導かれ、$2/n_{\text{in}}$ とはならないことを示せ。さらに、この方法で初期化した各 ReLU 層が実際には二乗平均を何倍にするか、また 50 層全体で何倍になるかを計算せよ。

5. [code] **Tanh ネットワーク**。●4.4.2.3 節の計測を、ReLU の代わりに tanh を用い、Xavier 初期化かつ 50 層の設定で繰り返せ。このとき、1 つの層は入力の二乗平均 m を $m' = \mathbb{E}[\tanh(z)^2]$ へ写す（ここで z は近似的に $\mathcal{N}(0, m)$ に従う）。
   1. $\tanh(z)^2 \approx z^2 - \frac{2}{3}z^4$（小さい z での近似）を用いて、$m' \approx m - 2m^2$ となることを示せ。多くの層を経たとき、二乗平均は一定に保たれるか、幾何級数的に減衰するか、それともより緩やかに減衰するか。また、第 k 層における二乗平均が約 $1/(2k)$（大きい k に対して）となることを示せ。
   2. 実測した二乗平均を層に対して対数軸上でプロットし、この予測を確かめよ。勾配には何が起きるか。

4.5 深層学習における汎化

2.6 節では、20 個のパラメータを持つ 19 次の多項式がノイズを含む 20 個のデータ点を補間（interpolation）し、テストデータ上で発散した。この古典的な直感に従えば、4.2 節の Fashion-MNIST 用 MLP は 203,530 個のパラメータを 60,000 枚の訓練画像に対して持つため（画像 1 枚あたり 3.4 個のパラメータ）、とりわけすべての訓練画像を補間するまで訓練した場合には激しく過学習するはずである。ところが実際には、未知の画像に対して $\sim 87\%$ の正解率に達した。さらに以下で見るように、同じアーキテクチャにわずか 1,000 枚の訓練画像しか与えず、訓練正解率が 100% になるまで訓練しても、なお未知の画像の約 80% を正しく分類する。

極度に過剰パラメータ化（overparameterization）されたネットワークがなぜ汎化（generalization）するのかを理解することは、現代の機械学習における中心的な未解決問題の 1 つである。本節では、統制された実験と解析可能なモデルに対する厳密な結果を組み合わせ、このパズルを解く 4 つの重要な手がかりを明らかにする。

1. **ランダムラベルの記憶：**標準的な MLP は、完全にランダムなラベルに対しても容易に訓練正解率 100% に達する (Zhang et al. 2021)。この事実は、分布に依存しない古典的な容量上界（VC 次元など）では実際のラベルに対する汎化を説明できないことを証明している。
2. **二重降下：**モデル幅を補間閾値（interpolation threshold）$h = n$ を超えて変化させると、特徴行列がほぼ特異になる $h = n$ でテスト誤差が急増し、その後、余分なパラメータによってよりノルムの小さい補間解が可能になるにつれて**第 2 の降下**が現れる (Belkin et al. 2019)。
3. **勾配降下法の暗黙的バイアス：**損失がゼロになる無数の解の中から、勾配降下法（gradient descent）は二乗損失のもとでは最小ノルム補間解を、ロジスティック損失のもとでは最大マージン分離器を自動的に選択する (Soudry et al. 2018)。
4. **クリーンなパターンの優先学習と早期停止：**ノイズを含むデータセットでは、勾配降下法は破損したラベルを記憶するよりずっと前に、クリーンな構造的パターンに適合する (Arpit et al. 2017)。そのため、**早期停止**（early stopping）が強力な正則化手法となる。

4.5.1 記憶と容量

4.5.1.1 ランダムラベル

Zhang et al. (2021) に倣って生の記憶容量（memorization capacity）を検証するため、Fashion-MNIST の訓練画像を 2,000 枚（訓練用に 1,000 枚、検証用に 1,000 枚）とテスト画像を 2,000 枚抽出し、補助クラス `ArrayData` と `Tracker`、および He 初期化を施した 256 ユニットの `MLP` を定義して、真のラベル（`fraction=0`）と完全にランダムなラベル（`fraction=1`）のそれぞれで 200 エポック訓練する。

```
data = d2l.FashionMNIST(batch_size=2000)
X, y = next(iter(data.train_dataloader()))          # 2,000 random images
X_test, y_test = next(iter(data.val_dataloader()))  # the first 2,000
X, y, X_test, y_test = (d2l.numpy(a) for a in (X, y, X_test, y_test))
X, X_test = X.reshape(2000, -1), X_test.reshape(2000, -1)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

以降の実験を簡潔に進めるため、2 つの軽量な補助クラスを用意する。`ArrayData` は NumPy 配列をシャッフルされたミニバッチにまとめ、`Tracker` は `Trainer` を継承して、名前付きデータセットの任意の辞書にわたってエポックごとの正解率を記録する。

```
class ArrayData(d2l.DataModule):
    """Minibatches of the arrays X and y; training batches are shuffled."""
    def __init__(self, X, y, batch_size=64):
        super().__init__()
        self.save_hyperparameters()

    def get_dataloader(self, train):
        return self.get_tensorloader(
            (d2l.tensor(self.X), d2l.tensor(self.y)), train)

class Tracker(d2l.Trainer):
    """A Trainer that records the accuracy on named datasets every epoch."""
    def __init__(self, max_epochs, datasets):
        super().__init__(max_epochs, num_gpus=1)
        self.datasets = datasets
        self.accuracy = {name: [] for name in datasets}
        self.board.display = False  # We plot the accuracies ourselves

    def validate(self):
        for name, dataset in self.datasets.items():
            self.val_dataloader = dataset.val_dataloader()
            self.accuracy[name].append(super().validate()['acc'])
        return {}
```

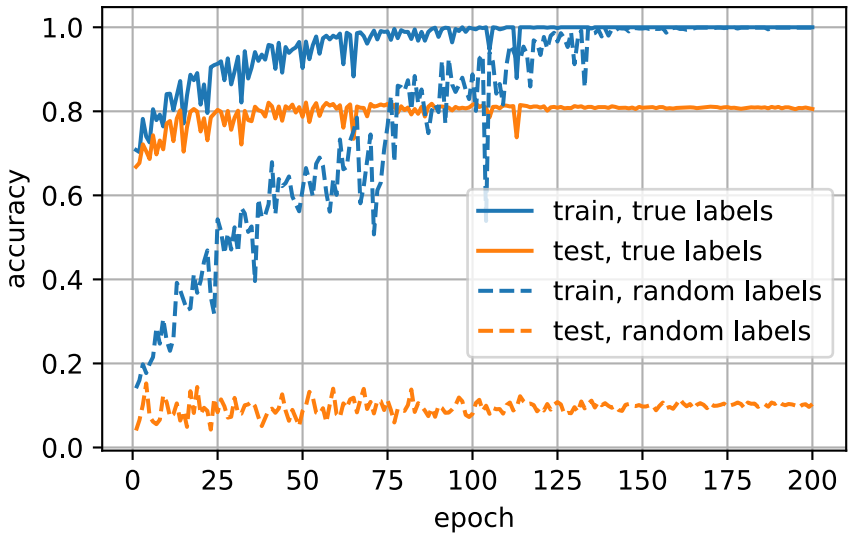
モデルは 4.2 節と同じ 256 ユニットの ReLU MLP であるが、He 初期化（4.4.2.2 節）を `std=0.01` の代わりに用いることで、第 1 エポックから直ちに訓練が進むように改良している（バイアスはゼロとし、入力は事前に平坦化している）。

```
class MLP(d2l.Classifier):
    def __init__(self, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(10))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

次に、`corrupt(y, fraction, rng)` を定義して各訓練ラベルを $\{0, \ldots, 9\}$ から一様ランダムに選んだクラスへ確率 `fraction` で置き換え、`MLP(num_hiddens=256, lr=0.1)` を 200 エポックにわたって `fraction = 0`（真のラベル）と `fraction = 1`（純粋なランダムラベル）で訓練し、訓練正解率とテスト正解率の推移をプロットする。

```
def corrupt(y, fraction, rng):
    """Replace each label, with probability fraction, by a random class."""
    redraw = rng.random(len(y)) < fraction
    y = y.copy()
    y[redraw] = rng.integers(0, 10, redraw.sum())
    return y
```

```
rng = np.random.default_rng()
curves = []
for fraction in (0, 1):
    labels = corrupt(y_train, fraction, rng)
    trainer = Tracker(max_epochs=200, datasets={
        'train': ArrayData(X_train, labels, 1000),
        'test': ArrayData(X_test, y_test, 1000)})
    trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
    curves += [trainer.accuracy['train'], trainer.accuracy['test']]
d2l.plot(list(range(1, 201)), curves, 'epoch', 'accuracy',
         fmts=('C0-', 'C1-', 'C0--', 'C1--'), figsize=(5, 3))
d2l.plt.legend(['train, true labels', 'test, true labels',
                'train, random labels', 'test, random labels'],
               loc='center right', bbox_to_anchor=(1, 0.4));
```



この 2 回の実行結果は、次の事実を証明している。

- **真のラベル**では、訓練正解率が 60〜90 エポック以内に 100% に達する一方、テスト正解率は 80% 超で横ばいになる（わずか 1,000 枚の訓練画像での結果である）。
- **完全にランダムなラベル**でも、まったく同じアーキテクチャとオプティマイザで依然として訓練正解率 100% に達する（やや長く ～ 120〜150 エポックを要する）一方、テスト正解率は当て推量の 10% にとどまる。

4.5.1.2 容量上界の限界と帰納バイアス

この MLP は訓練誤差ゼロ（$\epsilon_S(f) = 0$）を、真の母集団誤差が $\epsilon(f) = 0.90$ であるランダムラベル上で達成するため、$\epsilon(f) \leq \epsilon_S(f) + B(\mathcal{F}, n, \delta)$ という形をとり仮説クラス（hypothesis class）$\mathcal{F}$ のみに依存する分布に依存しない上界は、**いかなるものでも** $B(\mathcal{F}, n, \delta) \geq 0.90$ とならざるを得ず、実際のラベルに対しても空虚（vacuous）なものになる（演習問題 1）。

ノーフリーランチ定理（no free lunch theorem）(Wolpert 1996) により、あらゆる分布にわたって汎化できる学習器は存在しない。汎化するかどうかは、学習器の**帰納バイアス**（inductive

bias)、すなわち無数にある補間関数のうちどれを選好するかが実データの構造と一致しているかにかかっている。これは、1 最近傍分類器が訓練データの 100% を補間しながらも、近傍の入力同士が同じラベルを共有していさえすれば良好に汎化するのと同様である (Cover and Hart 1967)。

4.5.2 二重降下

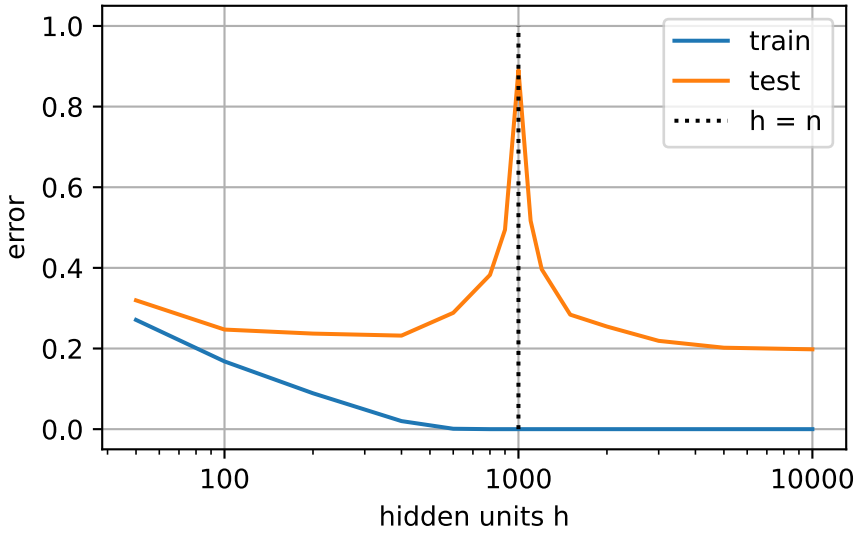
4.5.2.1 補間閾値と行列の条件

モデル幅 h をサンプルサイズ n をちょうど超えるまで増やすと何が起きるだろうか。解析しやすい**ランダム特徴量**（random-features）モデル (Rahimi and Recht 2007) で検証してみよう。He 初期化した第 1 層 $\mathbf{W}^{(1)} \in \mathbb{R}^{784\times 10000}$ を固定し、中心化した $n = 1{,}000$ 枚の訓練画像から 10,000 個のランダム ReLU 特徴量 `H` と `H_test` を事前計算する。さらに、ラベルをワンホット行列 `Y` に符号化し、`error(H, W2, y)` を定義する。

```
W1 = rng.normal(0, np.sqrt(2 / 784), (784, 10000))  # He-initialized, frozen
mean = X_train.mean(axis=0)                         # The mean training image
H = np.maximum((X_train - mean) @ W1, 0)
H_test = np.maximum((X_test - mean) @ W1, 0)
Y = np.eye(10)[y_train]                             # One-hot labels

def error(H, W2, y):
    """Fraction of examples whose largest output is not their label."""
    return (np.argmax(H @ W2, axis=1) != y).mean()
```

ここで隠れ層の幅 `h` を 50 から 10,000 までの 15 通りの値にわたって変化させ、出力重み $\mathbf{W}^{(2)} \in \mathbb{R}^{h\times 10}$ を先頭 `h` 個の特徴量の列 `H[:, :h]` に対して最小ノルム最小二乗法（`np.linalg.lstsq`）で当てはめつつ、最小特異値（singular value）`s_min[h]` とフロベニウスノルム `norm[h]` を記録する。

```
widths = [50, 100, 200, 400, 600, 800, 900, 1000, 1100, 1200, 1500, 2000,
          3000, 5000, 10000]
train_err, test_err, s_min, norm = [], [], {}, {}
for h in widths:
    W2, _, _, s = np.linalg.lstsq(H[:, :h], Y, rcond=None)
    train_err.append(error(H[:, :h], W2, y_train))
    test_err.append(error(H_test[:, :h], W2, y_test))
    s_min[h], norm[h] = s.min(), np.linalg.norm(W2)
d2l.plot([widths, widths, [1000, 1000]], [train_err, test_err, [0, 1]],
         'hidden units h', 'error', legend=['train', 'test', 'h = n'],
         xscale='log', fmts=('C0-', 'C1-', 'k:'), figsize=(5, 3))
```



これがよく知られた**二重降下**（double descent）曲線である (Belkin et al. 2019)。

1. **古典的な U 字曲線**（$h < n$）：テスト誤差は $\sim 30\%$（$h = 50$ のとき）から $\sim 23\%$（$h = 200$ 付近）まで下がるが、やがて $\sim 90\%$ へと、ちょうど**補間閾値** $h = n = 1{,}000$ において急上昇する。
2. **第 2 の降下**（$h > n$）：h が $n = 1{,}000$ を超えて 10,000 まで増えるにつれ、どのモデルも訓練セットを補間するにもかかわらず、テスト誤差は再び $\sim 20\%$ まで急落し、過少パラメータ化（underparameterized）された最良のモデルを上回る。

なぜテスト誤差は $h = n$ でこれほど激しく急増し、$h \gg n$ では回復するのだろうか。最小特異値 $s_{\min}$（$\mathbf{H}_{:,:h}$ に対する値）とフロベニウスノルム $\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$ を確認してみよう。

```
for h in (200, 900, 1000, 1100, 10000):
    print(f'h = {h:5d}: smallest singular value {s_min[h]:.1e}, '
          f'norm of W2 {norm[h]:6.1f}')
```

```
h =   200: smallest singular value 1.7e+00, norm of W2    3.7
h =   900: smallest singular value 1.2e-01, norm of W2   20.4
h =  1000: smallest singular value 2.0e-04, norm of W2 1577.9
h =  1100: smallest singular value 2.3e-02, norm of W2   20.4
h = 10000: smallest singular value 2.1e-01, norm of W2    2.2
```

$h = n = 1{,}000$ のとき、$\mathbf{H}$ は 1000×1000 の正方行列であり、最小特異値 $s_{\min}$ がきわめて小さいため、一意な厳密逆行列による解 $\mathbf{H}^{-1}\mathbf{Y}$ はラベルノイズを $s_{\min}$ で割ることになり、$\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$ を数百以上にまで爆発させる。一方、$h \gg n$ では余分な列によって $(h - n)$ 次元の部分空間（subspace）をなす厳密な補間解が生じるため、**最小ノルム**解は悪条件（ill-conditioned）の方向を避けて $\|\mathbf{W}^{(2)}\|_{\mathrm{F}}$ を ~ 2 にまで縮小できる。実際、$h = 1{,}000$ を固定したまま訓練セットを $n = 500$ から $n = 1{,}000$ へと倍増させると、正則化なしの当てはめはちょうど $h = n$ の崖の上に乗り、テスト誤差は $\sim 30\%$ から 85–90% へと**上昇**してしまう。

```
for n in (500, 1000):
    W2 = np.linalg.lstsq(H[:n, :1000], Y[:n], rcond=None)[0]
```

```
    test_error = error(H_test[:, :1000], W2, y_test)
    print(f'{n} training images: test error {test_error:.2f}')
```

```
500 training images: test error 0.28
1000 training images: test error 0.89
```

4.5.3 勾配降下法の暗黙的バイアス

上の第 2 の降下は、損失がゼロになる無数の解の中から `lstsq` が**最小ノルム**補間解を選ぶことに依拠していた。驚くべきことに、勾配降下法にはまさにこれと同じ選好が組み込まれている。これは、明示的なペナルティ項がなくても働く**暗黙的バイアス**（implicit bias）、別名**暗黙的正則化**（implicit regularization）である。

4.5.3.1 二乗損失のもとでの最小ノルム補間

過剰パラメータ化された連立一次方程式 $\mathbf{H}\mathbf{w} = \mathbf{y}$（$h > n$、行フルランク）において、一意な最小ノルム補間解は $\mathbf{w}^\star = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}$ である。

命題。 $L(\mathbf{w}) = \frac{1}{2}\|\mathbf{H}\mathbf{w} - \mathbf{y}\|_2^2$ に対する勾配降下法は、初期値を $\mathbf{w}_0 = \mathbf{0}$ とすると（収束するステップサイズのもとで）最小ノルム補間解 $\mathbf{w}^\star$ に収束する。

証明。 すべての勾配 $\nabla L(\mathbf{w}) = \mathbf{H}^\top(\mathbf{H}\mathbf{w} - \mathbf{y})$ は $\mathbf{H}$ の行空間に属する。初期値を $\mathbf{w}_0 = \mathbf{0}$ とすると、各反復点 $\mathbf{w}_t$、ひいては極限 $\mathbf{w}_\infty = \mathbf{H}^\top\mathbf{a}$ も $\mathbf{H}$ の行空間にとどまる（Figure 4.12）。$\mathbf{H}\mathbf{w}_\infty = \mathbf{H}\mathbf{H}^\top\mathbf{a} = \mathbf{y}$ であるから、$\mathbf{a} = (\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}$ となり、$\mathbf{w}_\infty = \mathbf{w}^\star$ が成り立つ。他の任意の補間解 $\mathbf{w}^\star + \mathbf{v}$ は $\mathbf{H}\mathbf{v} = \mathbf{0}$ を満たし（したがって $\mathbf{v} \perp \mathbf{w}^\star$）、二乗ノルムは $\|\mathbf{w}^\star\|_2^2 + \|\mathbf{v}\|_2^2$ となって真に大きくなる。■

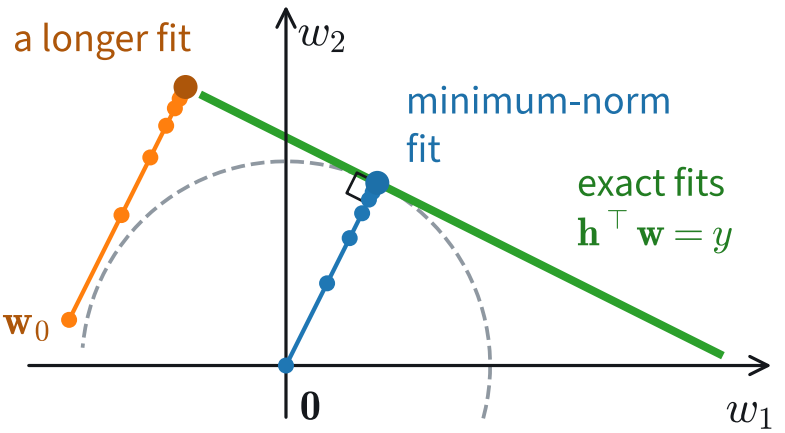


Figure 4.12: 劣決定な一次方程式 $\mathbf{h}^\top\mathbf{w} = y$ に対する勾配降下法は解超平面に対して厳密に垂直に動くため、$\mathbf{0}$（青）から出発すると最小ノルム解 $\mathbf{w}^\star$ に到達する一方、$\mathbf{w}_0 \neq \mathbf{0}$（オレンジ）から出発すると $\mathbf{w}_0$ の零空間成分がそのまま残る。

この命題を、式が 20 本、未知数が 100 個の系で数値的に検証し、$\mathbf{w}_0 = \mathbf{0}$ とランダム初期化を比較してみよう。

```
H_small = rng.normal(size=(20, 100))
y_small = rng.normal(size=20)
w_star = np.linalg.lstsq(H_small, y_small, rcond=None)[0]
lr = 1 / np.linalg.norm(H_small, 2) ** 2  # Below 2 / largest curvature
for name, w in (('zero', np.zeros(100)), ('random', rng.normal(size=100))):
    for _ in range(2000):
        w = w - lr * H_small.T @ (H_small @ w - y_small)
    residual = np.linalg.norm(H_small @ w - y_small)
    print(f'start {name:6s}: residual {residual:.0e}, norm '
          f'{np.linalg.norm(w):5.2f}, distance to w* '
          f'{np.linalg.norm(w - w_star):.1f}')
```

```
start zero  : residual 8e-17, norm  0.79, distance to w* 0.0
start random: residual 3e-15, norm  9.39, distance to w* 9.4
```

4.5.3.2 ロジスティック損失のもとでの最大マージン分類

線形分離可能な二値分類データ（$y_i \in \{-1, +1\}$）に対し、ロジスティック（交差エントロピー）損失 $\sum_i \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i))$ のもとでは $\|\mathbf{w}_t\|_2 \to \infty$（$t \to \infty$ のとき）となる。ここで Soudry et al. (2018) は、その**方向** $\mathbf{w}_t / \|\mathbf{w}_t\|_2$ が（$1/\log t$ の速さで）ハードマージン SVM の**最大マージン分離器**へ収束することを証明した（Figure 4.13）。これは、最小のマージンを持つサポートベクトルが $\log(1 + e^{-m})$ の指数関数的に減衰する裾を支配するためである（多層 ReLU ネットワークにおいて層のリスケーリングがパラメータノルムによるマージンをどのように複雑にするかについては、演習問題 5 を参照）。

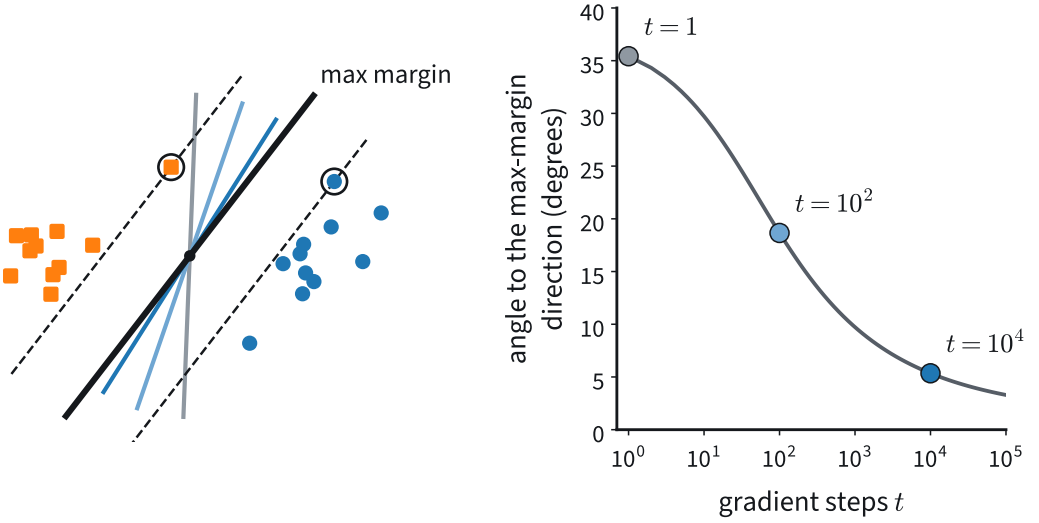
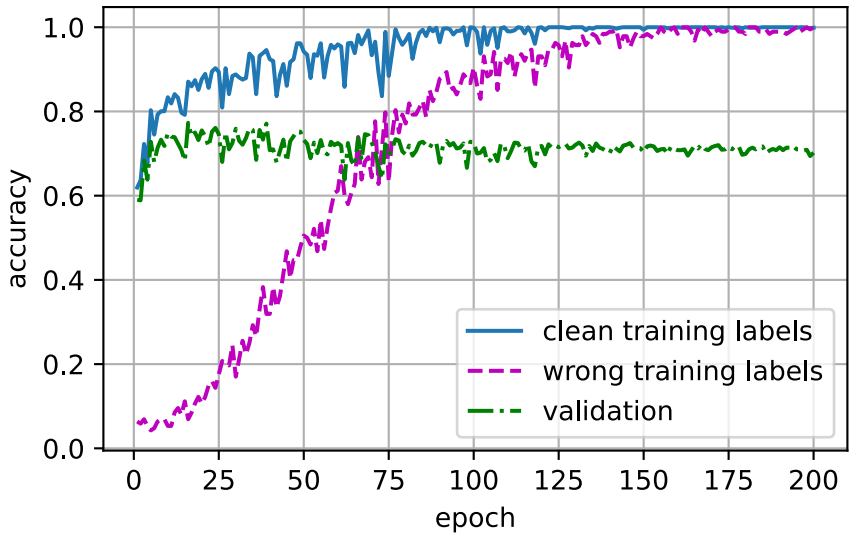


Figure 4.13: 線形分離可能なロジスティック回帰に対する勾配降下法は、$\mathbf{w}_t / \|\mathbf{w}_t\|_2$ を最大マージン超平面へ向けて対数的な速度で回転させ（左）、その回転は訓練正解率が 100% に達した後も長く続く（右）。

4.5.4 早期停止

破損したラベルを記憶するのに十分な容量を MLP が備えている場合でも、勾配降下法は**まずクリーンな構造的パターンを学習**し、**ラベルノイズはずっと後になって記憶**する (Arpit et al. 2017)。ここではラベルの 20% (画像 1,000 枚の訓練セット中) を破損させ、クリーンな訓練例、誤ラベル付きの訓練例 (破損したラベルを基準に評価)、および 1,000 枚のクリーンな検証画像それぞれに対する正解率を 200 エポックにわたって個別に追跡してみよう。

```
labels = corrupt(y_train, 0.2, rng)
wrong = labels != y_train
trainer = Tracker(max_epochs=200, datasets={
    'clean': ArrayData(X_train[~wrong], labels[~wrong], 1000),
    'mislabeled': ArrayData(X_train[wrong], labels[wrong], 1000),
    'validation': ArrayData(X_val, y_val, 1000),
    'test': ArrayData(X_test, y_test, 1000)})
trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
acc = trainer.accuracy
d2l.plot(list(range(1, 201)), [acc['clean'], acc['mislabeled'],
                               acc['validation']], 'epoch', 'accuracy',
         legend=['clean training labels', 'wrong training labels',
                 'validation'], figsize=(5, 3))
```



エポック 20 までに、MLP はクリーンな訓練画像の 80% 超を正しく分類する一方で、破損したラベルの記憶は 20% 未満にとどまる。検証正解率はこの初期の区間でピークに達し、その後のエポック終盤におけるノイズの記憶によって押し下げられる。**早期停止**とは、指定した**ペイシェンス**の期間にわたって検証性能が改善しなくなった時点で訓練を打ち切り (演習問題 2)、検証性能が最良であったチェックポイントを復元する手法である。これにより計算量を節約しつつテスト正解率を数パーセントポイント回復できる (なお線形最小二乗法においては、ℓ_2 重み減衰 (weight decay) と類似した働きをする。演習問題 4 を参照)。

```
best = int(np.argmax(acc['validation']))
print(f'best validation accuracy after epoch {best + 1}: '
      f'test accuracy {acc["test"][best]:.3f}')
print(f'after all 200 epochs: test accuracy {acc["test"][-1]:.3f}')
```

```
best validation accuracy after epoch 16: test accuracy 0.775
after all 200 epochs: test accuracy 0.716
```

早期停止の有名な例外として**グロッキング** (Power et al. 2022) がある（Figure 4.14、演習問題6）。剰余加算のような小規模なアルゴリズム的タスクで重み減衰を伴ってネットワークを訓練すると、訓練データの対応表は急速に記憶される一方、検証正解率は当て推量付近のまま $100\times$ 長く停滞した後に突然 100% へと跳ね上がる。

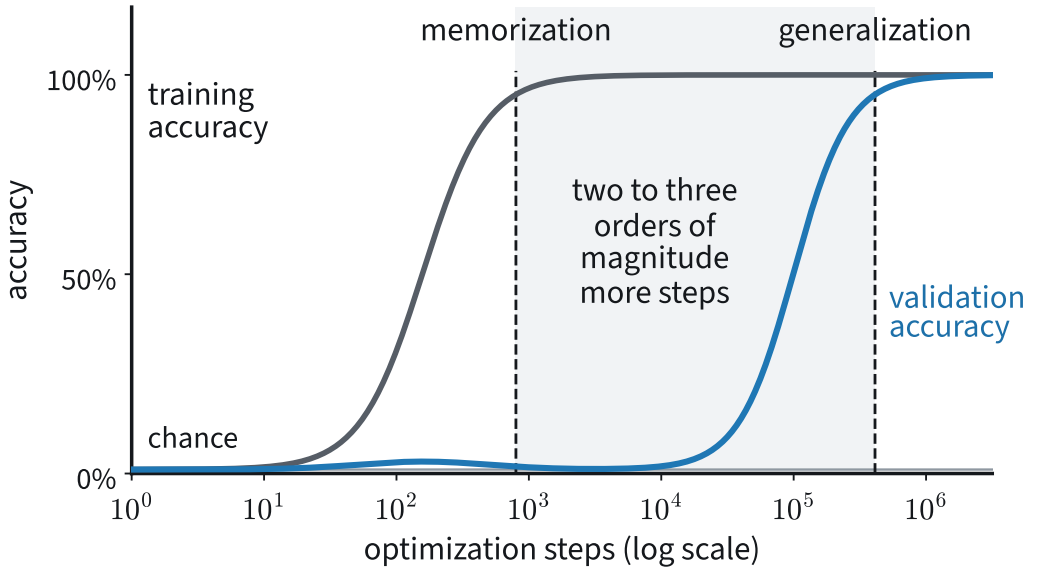


Figure 4.14: グロッキング（Power et al. (2022)）：モジュラ算術において、訓練正解率は $\sim 10^3$ ステップ以内に飽和する一方、検証正解率は $\sim 10^5$ ステップを経た後に初めて汎化へと跳ね上がる。

4.5.5 まとめ

過剰パラメータ化されたニューラルネットワークは、完全にランダムなラベルを記憶できる一方で構造を持つデータには良好に汎化するため、分布に依存しない容量上界は空虚なものとなり、汎化を説明する鍵は帰納バイアスに委ねられる。ランダム特徴量最小二乗法では、テスト誤差は補間閾値 $h = n$ で急増する。そこでは極端に小さい最小特異値 $s_{\min}(\mathbf{H})$ が $\|\mathbf{W}^{(2)}\|$ を膨張させるためであるが、$h \gg n$ になると最小ノルム補間が冗長な特徴量にわたって重みを分散させることで、テスト誤差は再び降下する。驚くべきことに、ゼロ初期化からの勾配降下法は、二乗損失のもとではこの最小ノルム補間解を自動的に見つけ出し、ロジスティック損失のもとでは方向において最大マージン分離器へと収束する。さらに勾配降下法は破損したラベルを記憶するより先に整合的なパターンへ適合するため、検証セットを用いた早期停止が単純かつ有効な正則化手法として機能する。

4.5.6 演習問題

1. **複雑さ尺度の限界**。あらゆるデータ分布に対して成り立つ汎化上界を考える。すなわち、少なくとも確率 $1-\delta$ で（n 個の訓練例の抽出に関して）、任意の分類器 f（クラス $\mathcal{F}$ の要素）が次式を満たすとする。

$$\epsilon(f) \leq \epsilon_{\mathcal{S}}(f) + B(\mathcal{F}, n, \delta). \tag{4.18}$$

   ここで、$\epsilon(f)$ は母集団誤差、$\epsilon_{\mathcal{S}}(f)$ は訓練誤差であり、項 B は $\mathcal{F}$、n、δ のみに依存する（VC 上界における項 $c\sqrt{(v+\log(1/\delta))/n}$ と同様である）。本文では、このような上界に対して 1,000 枚の画像を用いたランダムラベルの議論を展開した。

   1. Fashion-MNIST の全 60,000 枚の訓練画像に対してラベルを一様ランダムに引き直したときにも、訓練によって $\mathcal{F}$ の中から訓練誤差ゼロの要素が見つかるとする（これは Zhang et al. (2021) が画像分類器について観察した通りである）。このとき $B(\mathcal{F}, 60{,}000, \delta)$ に対してどのような下界が導かれるか。また、その場合この上界は、真のラベルで訓練されたネットワークに対して何を保証するか。
   2. 近年の多くの汎化上界では、右辺の第 2 項が訓練の出力する分類器に依存することを許している（たとえば重みのノルムや、訓練例に対するマージンを通じて依存する）。この場合、上の議論のどの段階が成り立たなくなるか。また、真のラベルに対して意味のある主張を与えるためには、そのような量はランダムラベルでの実行時にどのような振る舞いをしなければならないか。

2. [code] **ペイシェンスに基づく停止**。`FashionMNIST` は Fashion-MNIST のテストセットを検証データとして使用しているため、代わりに 10,000 枚の訓練画像を検証セットとして取り分けておけ。残りの画像上で、256 ユニットの隠れ層を 1 つ持つ MLP を 4.2 節と同様に、過学習するまで十分に大きな固定エポック数だけ訓練し、各エポック終了時の検証損失とテスト正解率を記録せよ。許容幅 θ とペイシェンス k を持つペイシェンス規則は、検証損失がこれまでの最良値から θ を超えて改善しない状態が k エポック連続した時点で訓練を停止し、そこまでのうち検証損失が最小であったエポックのパラメータを採用する。

   1. θ を固定し、記録した曲線に対して $k \in \{1, 5, 20\}$ のそれぞれでこの規則を適用せよ。各 k について、停止エポック、採用されたエポック、およびその両方におけるテスト正解率を報告せよ。
   2. 採用されたエポックにおけるテスト正解率は k にどのように依存するか。その依存性を、エポックごとの検証損失の揺らぎと関連づけて説明せよ。

   Michael Nielsen『Neural Networks and Deep Learning』第 3 章より改作。

3. [code] **閾値における重み減衰**。二重降下の実験における最小ノルム解による当てはめを、リッジ解 $\mathbf{W}^{(2)} = (\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top\mathbf{Y}$（2.7 節と同じく n 個の例にわたって二乗誤差の平均をとったもの）に置き換えよ。

   1. $\lambda \in \{10^{-5}, 10^{-3}, 10^{-1}\}$ に対して、幅に対するテスト誤差をプロットせよ。$h = n$ におけるピークはどうなるか。
   2. リッジ解が $\lambda \to 0$ のとき（$h > n$ において）最小ノルム解に近づくことを示せ。ヒント：$(\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top + n\lambda\mathbf{I})^{-1}$ と変形する。
   3. 各幅に対して、1,000 枚の検証画像を用いて λ を選択せよ。調整後のモデルの曲線は依然として 2 重降下するか。

4. **早期停止と重み減衰**。●重み減衰が重みの大きさを制限するのに対し、早期停止は訓練エポック数を制限する。線形最小二乗法ではこの対応関係を厳密に記述できる。$\mathbf{X}^\top\mathbf{X}/n$ が固有

値 $\mu_i > 0$ を持つとし、$\mathbf{w}^*$ を $L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ の最小点とし、L に対する勾配降下法を初期値 $\mathbf{w}_0 = \mathbf{0}$、学習率 $\eta < 1/\max_i \mu_i$ で実行する。

1. t ステップ後における $\mathbf{w}_t$ の第 i 固有ベクトル方向の座標が、係数 $1 - (1 - \eta\mu_i)^t$ を $\mathbf{w}^*$ の同じ座標に掛けたものに等しいことを示せ。
2. $L(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$ の最小点は、同じ座標を $\mu_i/(\mu_i + \lambda)$ 倍に縮小する（2.7.3 節）。λ を η と t の関数としてどのように選べば、これら 2 つの係数が $\eta\mu_i t \ll 1$ の方向と $\eta\mu_i t \gg 1$ の方向の両方で近似的に一致するか。

Ian Goodfellow、Yoshua Bengio、Aaron Courville『Deep Learning』7.8 節より改作。

5. **ロジスティック損失の暗黙的バイアス。**●線形分離可能なデータに対して、ロジスティック損失上の勾配降下法は方向において最大マージン分離器へ収束する (Soudry et al. 2018)。ラベルを $y_i \in \{-1, +1\}$ とする。
   1. 線形予測器 $f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$ について、分離可能なデータ上では損失 $\sum_i \log\left(1 + \exp(-y_i\mathbf{w}^\top\mathbf{x}_i)\right)$ が任意の $\mathbf{w}$ に対して正である一方、その下限はゼロであることを示せ。ここから、勾配降下法が損失をゼロへ近づけるとき $\|\mathbf{w}_t\|$ は際限なく増大し、したがって方向 $\mathbf{w}_t/\|\mathbf{w}_t\|$ のみが収束し得ることを結論づけよ。
   2. 2 層の ReLU ネットワーク $f(\mathbf{x}) = \sum_j v_j \max(0, \mathbf{w}_j^\top\mathbf{x})$ について、$(\mathbf{w}_j, v_j)$ を $(c_j\mathbf{w}_j, v_j/c_j)$（ただし $c_j > 0$）に置き換えても f が変化しないことを示せ。このリスケーリング全体にわたる $\sum_j(\|\mathbf{w}_j\|^2 + v_j^2)$ の最小値を求めよ。また、ネットワークのパラメータノルムで測ったマージンが、線形予測器に対して $\|\mathbf{w}\|$ で測ったマージンとは異なり、関数 f のみで決まる性質ではない理由を説明せよ。
6. [extended] **グロッキング。**●●Power et al. (2022) の設定を再現せよ。すなわち、小規模なネットワーク（元論文では 2 層の Transformer を用いているが、ワンホット表現の組を入力とする幅広の MLP でもよい）を用い、$c = (a + b) \bmod 97$ を組 (a, b) から予測するよう、全ペアからランダムに選んだ 50% を訓練データとし、重み減衰を加えて 10^5 ステップ以上訓練せよ。ステップ数の対数に対して訓練正解率と検証正解率をプロットし、Figure 4.14 と比較せよ。汎化が起こるまでの遅れは、訓練データの割合や重み減衰の強さによってどのように変化するか。

4.6 ドロップアウト

過剰パラメータ化されたネットワークを希少なデータやノイズの多いデータで何エポックも訓練すると、やがて訓練データ特有の癖や破損したラベルまで暗記してしまう（4.5 節）。重み減衰（weight decay、2.7 節）が大きな重みにペナルティを課し、早期停止（early stopping、4.5.4 節）が終盤の暗記が始まる前に訓練を打ち切るのに対し、**ドロップアウト**は訓練中に乗法的なノイズを隠れ層へ直接注入することでネットワークを正則化する (Hinton et al. 2012; Srivastava et al. 2014)。すなわち、各順伝播において各隠れユニット（hidden unit）をそれぞれ独立に確率 p でゼロにすることで、特定のパートナーユニットに頼る**共適応**（co-adaptation）を起こさず、どのユニットにも単独で確実に機能する特徴を学習させる。テスト時にはドロップアウトを無効にし、ネットワーク全体を決定論的に動作させる。

本節では、逆ドロップアウト（inverted dropout）におけるマスク処理と再スケーリングを定式化し、Fashion-MNIST 上で隠れ層を 2 つ持つ MLP にゼロからの実装と簡潔な実装の両方を施す。さらに、ラベルを大幅に破損させたラベルノイズ実験を通じて、ドロップアウトがいかに暗記を遅らせ、検証正解率の低下をどれだけ食い止めるかを実証する。

4.6.1 隠れ層におけるノイズ

4.6.1.1 マスク処理と再スケーリング（逆ドロップアウト）

訓練中に入力へランダムなノイズを注入すると、摂動に強く滑らかな関数が学習されやすくなる(Bishop 1995)。実際、二乗誤差（squared error）のもとでの線形モデル $\mathbf{w}^\top \mathbf{x} + b$ において、入力に平均 0・分散 s^2 の独立なノイズを加えると、期待損失には $s^2 \|\mathbf{w}\|_2^2$ が加算される。これは ℓ_2 重み減衰にほかならない。ドロップアウトは、ランダムな二値マスクによってこのノイズ注入を隠れ層へと拡張する。

隠れ層の活性化値（activation）h_j と**ドロップアウト確率** $p \in [0, 1)$ に対し、各訓練ステップで独立なベルヌーイマスク $\xi_j \in \{0, 1\}$（$P(\xi_j = 0) = p$、$P(\xi_j = 1) = 1 - p$）をサンプリングし、h_j を次の式で置き換える。

$$h_j' = \frac{\xi_j h_j}{1 - p} = \begin{cases} 0 & \text{with probability } p, \\ \dfrac{h_j}{1 - p} & \text{with probability } 1 - p. \end{cases} \tag{4.19}$$

訓練中に生き残った活性化値を $1 - p$ で割ること（**逆ドロップアウト**）により、期待値は不偏に保たれる。

$$\mathbb{E}[h_j'] = p \cdot 0 + (1 - p) \frac{h_j}{1 - p} = h_j, \qquad \mathrm{Var}[h_j'] = \frac{p}{1 - p} h_j^2.$$

訓練中に $\mathbb{E}[h_j'] = h_j$ が保たれるため、**推論時には重みの再スケーリングを一切せず、単にドロップアウトをオフにする（$h_j' = h_j$）だけでよい**。（なお、入力ドロップアウトを伴う線形回帰（linear regression）モデルでは、分散の式 $\mathrm{Var}[x_j'] = \frac{p}{1-p} x_j^2$ により、二乗損失（squared loss）に適応的なリッジペナルティ $\mathbb{E}[(\mathbf{w}^\top \mathbf{x}' + b - y)^2] = (\mathbf{w}^\top \mathbf{x} + b - y)^2 + \frac{p}{1-p} \sum_j x_j^2 w_j^2$ が加わる。一方、深い非線形ネットワークでは $\mathbb{E}[\mathrm{ReLU}(z)] \neq \mathrm{ReLU}(\mathbb{E}[z])$ であるため、テスト時の決定論的なネットワークは、間引かれた 2^h 個のサブネットワークからなるアンサンブルに対する実用的な近似となっている。演習問題 3 および 5 を参照。）

4.6.2 実装

4.6.2.1 ドロップアウト層

ゼロから実装する `dropout_layer(X, dropout)` は、区間 $[0, 1)$ の一様分布からテンソルをサンプリングし、値が $\geq$ `dropout` である要素を残して `1.0 - dropout` で割る（ただし `dropout == 1` のときはゼロのテンソルを返す）。

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return torch.zeros_like(X)
    mask = torch.rand_like(X) >= dropout
    return mask.to(X.dtype) * X / (1.0 - dropout)
```

`dropout_layer` を 2×8 のテンソルに対し、ドロップアウト確率 0、0.5、1 でテストしてみよう。また、100 万個の 1 からなるベクトルに $p = 0.2$ を適用し、標本平均と標本分散が理論値である 1 および $p/(1-p) = 0.25$ と一致することを確かめる。

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (2, 8))
for p in (0, 0.5, 1):
    print(f'p = {p}:', dropout_layer(X, p))
Y = dropout_layer(d2l.ones(10**6), 0.2)
print(f'mean {float(d2l.reduce_mean(Y)):.3f}, '
      f'variance {float(d2l.reduce_mean((Y - 1) ** 2)):.3f}')
```

```
p = 0: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
p = 0.5: tensor([[ 0.,  2.,  4.,  6.,  8., 10., 12., 14.],
        [ 0., 18.,  0., 22., 24., 26.,  0., 30.]])
p = 1: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
mean 1.000, variance 0.250
```

4.6.2.2 モデルのゼロからの定義と訓練

各隠れ層の出力に対し、ReLU 活性化の直後にドロップアウトを適用する（Figure 4.15）。ただし、有効にするのは訓練中**のみ**である。以下では、He 初期化（He initialization）を施した隠れ層 2 層の分類器 `DropoutMLPScratch`（$784 \to 256 \to 256 \to 10$）を、層ごとに独立したドロップアウト確率 `dropout_1` と `dropout_2` を持たせて定義する。

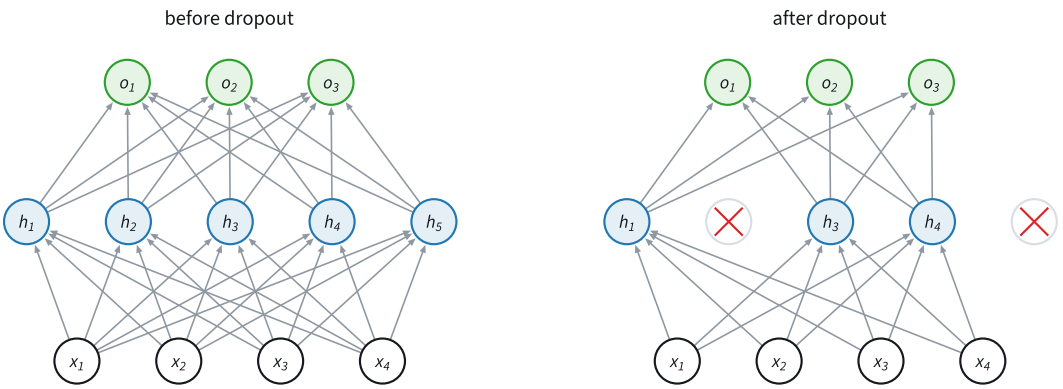


Figure 4.15: ドロップアウトは各訓練ステップにおいて、隠れユニットのランダムな部分集合（図では h_2 と h_5）およびそれらに接続する結合を無効化する。

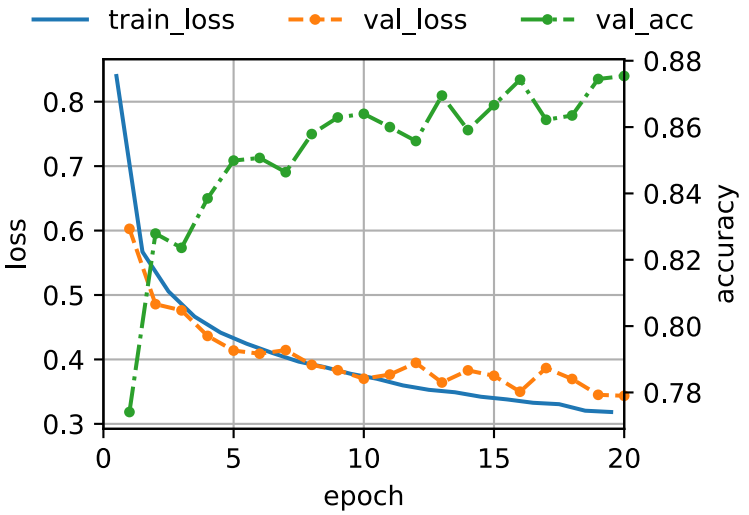
各層は `self.training` を判定する。トレーナの `model.eval()` はこれを `False` に設定する。

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr, num_inputs=784):
        super().__init__()
        self.save_hyperparameters()
        he = lambda n_in, n_out: nn.Parameter(  # Variance 2 / n_in
            d2l.normal(0, (2 / n_in) ** 0.5, (n_in, n_out)))
        self.W1 = he(num_inputs, num_hiddens_1)
        self.W2 = he(num_hiddens_1, num_hiddens_2)
        self.W3 = he(num_hiddens_2, num_outputs)
```

```
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens_1))
        self.b2 = nn.Parameter(d2l.zeros(num_hiddens_2))
        self.b3 = nn.Parameter(d2l.zeros(num_outputs))

    def forward(self, X):
        X = X.reshape(-1, self.num_inputs)
        H1 = torch.relu(X @ self.W1 + self.b1)
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = torch.relu(H1 @ self.W2 + self.b2)
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return H2 @ self.W3 + self.b3
```

ここで DropoutMLPScratch を、第 1 隠れ層の後に dropout_1 = 0.2、第 2 隠れ層の後に dropout_2 = 0.5、学習率（learning rate）に lr = 0.1 を指定してインスタンス化し（後で再利用するためハイパーパラメータは hparams にまとめておく）、Fashion-MNIST 上で 20 エポック訓練する。

```
hparams = {'num_outputs': 10, 'num_hiddens_1': 256, 'num_hiddens_2': 256,
           'dropout_1': 0.2, 'dropout_2': 0.5, 'lr': 0.1}
model = DropoutMLPScratch(**hparams)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
trainer.fit(model, data)
```



最初の数エポックでは、訓練損失（training loss）が検証損失（validation loss）よりも上に位置していることがわかる。これは、訓練損失がノイズを伴うドロップアウトマスクを有効にした状態で評価されるのに対し、検証損失は決定論的なネットワーク全体を使って計算されるためである。

4.6.2.3 簡潔な実装

フレームワークのドロップアウト層 `Dropout` を `Sequential` の内部で用いれば、まったく同じ隠れ層 2 層のアーキテクチャを数行で記述できる。

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.LazyLinear(num_hiddens_1), nn.ReLU(),
            nn.Dropout(dropout_1), nn.LazyLinear(num_hiddens_2), nn.ReLU(),
            nn.Dropout(dropout_2), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

4.6.3 ラベルノイズ下におけるドロップアウトの効果

60,000 枚の画像すべてを含む Fashion-MNIST データセット全体で 20 エポック訓練する程度では、正則化なしの MLP でもほとんど過学習（overfitting）しないため、ドロップアウトが真価を発揮する余地は小さい。深刻な暗記にドロップアウトがどう対抗するかを観察するため、4.5.4 節で用いた 1,000 枚の画像からなるラベルノイズ付きベンチマークに立ち返り、`d2l.corrupt` によって訓練ラベルの 40% をランダムに破損させる（誤ラベルは全体の $\sim 36\%$ となり、ブールマスク `wrong = labels != y` で参照する）。

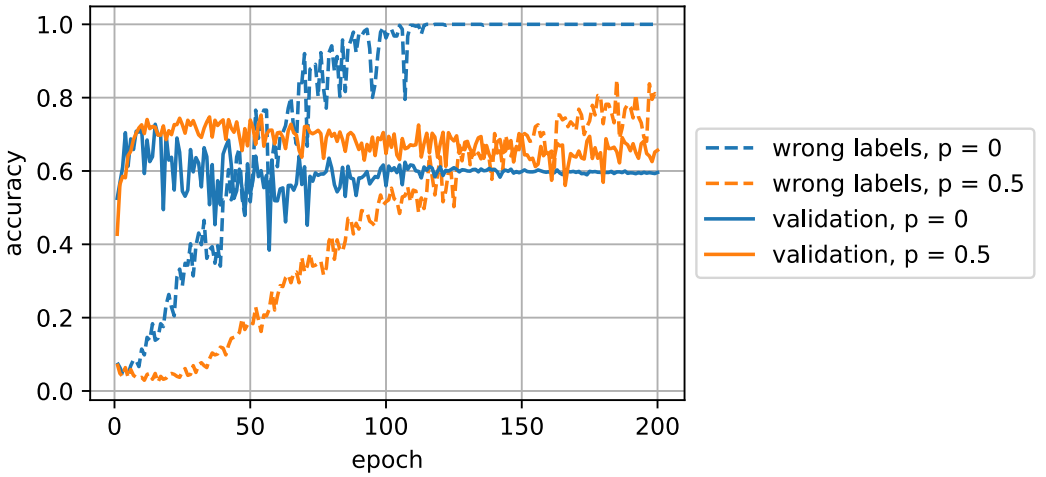
```
images = d2l.FashionMNIST(batch_size=2000)
X, y = (d2l.numpy(a) for a in next(iter(images.train_dataloader())))
X_val, y_val = (d2l.numpy(a) for a in next(iter(images.val_dataloader())))
X, y, X_val = X[:1000].reshape(1000, -1), y[:1000], X_val.reshape(2000, -1)
labels = d2l.corrupt(y, 0.4, np.random.default_rng())
wrong = labels != y  # The training labels that are now wrong
```

ここで `DropoutMLP` を 200 エポックにわたり `(X, labels)` 上で訓練し、両方の隠れ層のドロップアウト確率を `p = 0`（ドロップアウトなし）とした場合と `p = 0.5` とした場合を比較する。その際、`d2l.Tracker` を用いて、破損した訓練サブセット `X[wrong]` に対する正解率（accuracy）と、ホールドアウトされた `X_val` に対する検証正解率を記録する。

```
acc = {}
for p in (0, 0.5):
    trainer = d2l.Tracker(max_epochs=200, datasets={
        'wrong labels': d2l.ArrayData(X[wrong], labels[wrong], 1000),
        'validation': d2l.ArrayData(X_val, y_val, 1000)})
    trainer.fit(DropoutMLP(**{**hparams, 'dropout_1': p, 'dropout_2': p}),
                d2l.ArrayData(X, labels))
    acc[p] = trainer.accuracy
    print(f'dropout {p}: wrong labels reproduced '
```

```
            f'{acc[p]["wrong labels"][-1]:.2f}, validation accuracy over the '
            f'last 10 epochs {np.mean(acc[p]["validation"][-10:]):.3f}')
d2l.plot(list(range(1, 201)), [acc[0]['wrong labels'], acc[0.5]['wrong labels'],
                               acc[0]['validation'], acc[0.5]['validation']],
         'epoch', 'accuracy', fmts=('C0--', 'C1--', 'C0-', 'C1-'),
         legend=['wrong labels, p = 0', 'wrong labels, p = 0.5',
                 'validation, p = 0', 'validation, p = 0.5'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
dropout 0: wrong labels reproduced 1.00, validation accuracy over the last 10 epochs 0.595
dropout 0.5: wrong labels reproduced 0.80, validation accuracy over the last 10 epochs 0.654
```



2 つの軌跡を見てみよう。

- **ドロップアウトなし**（$p = 0$）：100 エポック目までに MLP は破損した訓練ラベルを 100% 暗記してしまい、検証正解率は序盤のピークから ~ 10 パーセントポイント低下する。
- **ドロップアウトあり**（$p = 0.5$）：誤ラベルの暗記が減速する。200 エポック目でも MLP が再現する誤ラベルはすべてではなく約 $70\text{〜}85\%$ にとどまり、最後の 10 エポックにおける検証正解率は正則化なしのネットワークより **3〜6 パーセントポイント高い**。もっとも、誤ラベルの正解率曲線はなお上昇を続けている。すなわち、ドロップアウトは悪影響を取り除くのではなく遅らせる手法であり、だからこそ早期停止と組み合わせて使われるのである。

実務上、ドロップアウトが最も威力を発揮するのは、パラメータ数（parameter count）がデータセットのサイズを圧倒的に上回り、かつ何エポックにもわたって訓練する場合である（7.1 節の AlexNet のような 2010 年代初頭の画像分類器がその典型である）。一方、データを 1 周のみ処理する基盤モデルの事前学習（pretraining）や、バッチ正規化（batch normalization）層の直前（7.3 節、訓練時にドロップアウトが分散を $1/(1-p)$ 倍に膨張させることで訓練時とテスト時の間に分散のずれが生じる (Li et al. 2019)）では、通常ドロップアウトを弱めるか省略する。

4.6.4 まとめ

逆ドロップアウトは、訓練中に各隠れ層の活性化値をそれぞれ独立に確率 p でゼロにし、生き残った活性化値を $1/(1-p)$ 倍に再スケーリングする。この再スケーリングは $\mathbb{E}[h'_j] = h_j$ を保ちつつ分散 $\frac{p}{1-p}h_j^2$ を注入するため、テスト時には重みの再スケーリングを一切せずにマスクなしの

ネットワーク全体を決定論的に動作させられる。パラメータ数がサンプル数を大きく上回る小規模なデータセットやノイズの多いデータセットでは、ドロップアウトによって隠れユニット間の脆い共適応が抑えられ、破損したラベルの暗記が遅くなり、ホールドアウト正解率が数パーセントポイント向上する。ただし、ドロップアウトは最終的な暗記を防ぐのではなく遅らせるものであるため、早期停止や重み減衰と併用するときに最も効果を発揮する。

4.6.5 演習問題

1. [code] **ドロップアウト確率**。本節のモデルは、第 1 隠れ層の後に確率 0.2 で、第 2 隠れ層の後に確率 0.5 でユニットをドロップする。
    1. 2 つの確率を入れ替えた場合、両方を 0.5 に設定した場合、両方を 0.8 に設定した場合のそれぞれについてモデルを訓練せよ。各実行における最良の検証正解率を報告せよ。より高いドロップアウト確率に耐えられるのはどちらの層か。
    2. `Trainer` がプロットする訓練損失は、マスクをオンにした状態で計算されている。訓練後にマスクをオフにした状態で訓練画像に対する損失を評価し、ドロップアウトありで訓練したネットワークとドロップアウトなしで訓練したネットワークとで、訓練損失と検証損失のギャップを比較せよ。
2. **ドロップアウトと He 初期化**。He 初期化（4.4.2.2 節）では、活性化前の値の二乗平均が層から層へと一定に保たれるように重みの分散を選ぶ。
    1. 訓練中、確率 p のドロップアウトは次の層の活性化前の値の二乗平均を何倍に変化させるか。本節のモデルにおける 2 つのドロップアウト層のそれぞれについて、その倍率を求めよ。
    2. $K-1$ 個の隠れ層を持ち、各隠れ層の後に確率 p のドロップアウトが続くネットワークにおいて、この倍率は K に伴ってどのように増大するか。テスト時にはドロップアウトなしでネットワークが動作することを踏まえると、初期化でこの倍率を補正すべきか。バッチ正規化層の直前にドロップアウトを置いたときに生じる訓練時とテスト時の不一致と関連づけて答えよ。
3. [code] **モンテカルロドロップアウト**。テスト時にドロップアウトをオフにする代わりに、訓練済みの `DropoutMLPScratch` でドロップアウトをオンのままにし、各検証画像に対して $T = 20$ 回の順伝播を実行してソフトマックス出力を平均せよ (Gal and Ghahramani 2016)。その正解率と較正（最大のソフトマックス確率の平均と正解率との一致度）を、ネットワーク全体による 1 回の順伝播の結果と比較せよ。T を 1 から 20 まで増やすにつれて、正解率はどのように変化するか。
4. [code] **ドロップアウトと重み減衰**。重み減衰とドロップアウトはいずれも、異なる仕組みによって本節のモデルを正則化する。各実行を、最良の検証正解率、およびマスクをオフにした状態で評価した訓練損失と検証損失のギャップによって測定せよ。
    1. 両方のドロップアウト確率を 0 に設定したうえで、2.7 節と同様に強度 $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}\}$ の重み減衰を加えよ。これらの実行結果を、本節のドロップアウトありおよびドロップアウトなしの実行結果と比較せよ。
    2. ドロップアウト確率を 0.2 と 0.5 に保ったまま、同じ強度の重み減衰を加えよ。2 つの正則化手法の効果は加法的に積み重なるか、収穫逓減を示すか、それとも互いに打ち消し合うか。
5. **テスト時のドロップアウト**。●`dropout_1` を 0 に設定して第 2 隠れ層のみがマスクされるようにし、ξ がそのマスク全体を動くとする。$\mathbf{o}(\xi)$ と $\hat{\mathbf{y}}(\xi)$ を、マスク ξ（確率 $P(\xi)$ で生じる）のもとでのロジットおよびソフトマックス出力とする。ネットワーク全体のロジットが

$\sum_\xi P(\xi)\,\mathbf{o}(\xi)$ に等しいことを示せ。さらに、そのソフトマックス出力が次の正規化された重み付き幾何平均に等しいことを示せ。

$$\tilde{y}_j = \frac{\prod_\xi \hat{y}_j(\xi)^{P(\xi)}}{\sum_k \prod_\xi \hat{y}_k(\xi)^{P(\xi)}}. \tag{4.20}$$

また、dropout_1 が非ゼロになると、いずれの等式も一般には成り立たない理由を説明せよ。

4.7 Kaggle での住宅価格予測

Fashion-MNIST のような整った画像ベンチマークは、$[0, 1]$ の均一なピクセル格子として提供され、欠損値も含まれない。実世界の表形式データ（tabular data）がこれほど整然としていることは稀である。本章の集大成となる本節では、Kaggle の House Prices: Advanced Regression Techniques コンペティションに取り組む。このデータセットは De Cock (2011) が米国アイオワ州エイムズの不動産売買記録（2006〜2010 年）から編纂したもので、ラベル付きの訓練用住宅 1,460 軒とラベルなしのテスト用住宅 1,459 軒からなる。各住宅はスケールが 5 桁にわたる異種混合の数値属性およびカテゴリ属性 79 個で記述されており、19 列に欠損値がある。

このエンドツーエンドの表形式パイプラインを一通り構築することで、第 1 章から第 4 章までで学んだツールが一つに結びつく。具体的には、データリークのない pandas による前処理、ターゲットの対数変換、共有フォールドを用いた K 分割交差検証（cross-validation）、線形回帰のベースライン、そして重み減衰（weight decay）で正則化した He 初期化 MLP である。

```
import pandas as pd
```

4.7.1 データセットと log-RMSE 指標

d2l.download を使って訓練用およびテスト用の CSV ファイルをダウンロードしてキャッシュし、pandas の DataFrame に読み込む（1.2 節）。

```
def read_house_prices(root='../data'):
    """Download and read the training and test tables of the competition."""
    train = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_train.csv', root,
        sha1_hash='585e9cc93e70b39160e7921475f9bcd7d31219ce'))
    test = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_test.csv', root,
        sha1_hash='fa19780a7b011d9b009e8bff8e99922a8ee2eb90'))
    return train, test
```

read_house_prices() を呼び出すと 2 つのテーブルが raw_train と raw_test に読み込まれ、その形状から訓練用住宅が 1,460 軒で 81 列（SalePrice を含む）、テスト用住宅が 1,459 軒で 80 列であることを確認できる。

```
raw_train, raw_test = read_house_prices()
print(raw_train.shape, raw_test.shape)
```

```
(1460, 81) (1459, 80)
```

raw_train の先頭と末尾の数列について最初の 4 行を確認すると、整数コード（Id、MSSubClass）、文字列のカテゴリ（MSZoning、SaleType、SaleCondition）、連続値の計測値（LotFrontage）、そしてドル建てのターゲットである SalePrice が混在している様子がわかる。

```
print(raw_train.iloc[:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

予測に用いる 79 個の属性（情報を持たない行インデックス Id を除いたもの）のうち、36 個は数値として、43 個は文字列として格納されており、欠損値は設備そのものが存在しないことを表している場合が多い（たとえばプールがない住宅での PoolQC = NaN など）。

住宅価格は $34,900 から $755,000 まで幅があるため、$50,000 の誤差は手頃な住宅では致命的である一方、豪邸ではわずかな誤差にすぎない。そのため Kaggle では、価格の**対数**の二乗平均平方根誤差（root-mean-squared error、RMSE）を用いて、予測の**相対的な**精度によってスコアを算出する。

$$\text{log-RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\log y_i - \log \hat{y}_i\right)^2}. \tag{4.21}$$

$\delta = |\log \hat{y} - \log y|$ が小さいとき、$\hat{y}/y = e^{\pm\delta} \approx 1 \pm \delta$ となるため、log-RMSE が 0.135 であることは典型的な相対誤差が約 13.5% であることに相当する。二乗誤差損失のもとで $\log y$ を予測するようにモデルを訓練すれば、コンペティションの評価指標の二乗を直接最適化できる。

4.7.2 データリークのない前処理

元の DataFrame を float32 の特徴量行列に変換するには、以下の preprocess(train, *others) に実装されている 4 つの慎重なステップが必要である。

1. **数値のカテゴリコードを文字列に変換する：**MSSubClass は数値（$20, 60, 90, \ldots$）を使って住居タイプ（1 階建て、2 階建て、二世帯住宅）を表している。これを str に変換することで、モデルがコード 90 をコード 20 の定数倍として扱うのを防ぐ。
2. **数値特徴量を訓練セットの統計量のみで標準化する：**平均 μ と標準偏差 s を train のみから計算し、訓練テーブルと検証・テストテーブルの両方に $(x - \mu)/s$ を適用することで、SGD と重み減衰のためにすべての数値特徴量を単位スケールに揃えつつ、検証行やテスト行からの**データリーク**を防ぐ。
3. **欠損している数値を 0（訓練平均）で補完する：**標準化後の NaN 要素を 0 で埋めることで、欠損している数値の計測値を訓練セットの平均で置き換える（演習問題 1）。
4. **カテゴリ特徴量をワンホットエンコーディングする（pd.get_dummies(..., dummy_na=True)）：**各カテゴリ列を二値インジケータ（NaN に対する明示的なインジケータを含む）に展開し、reindex(..., fill_value=0) によって検証・テスト列を訓練列の語彙と厳密に一致させる。

```
def preprocess(train, *others):
    """Fit the feature preprocessing on train; apply it to all tables."""
```

```
    frames = [df.drop(columns=['Id', 'SalePrice'], errors='ignore')
                .astype({'MSSubClass': str})  # A code, not a quantity
              for df in (train, *others)]
    numeric = frames[0].select_dtypes(include='number').columns
    mean, std = frames[0][numeric].mean(), frames[0][numeric].std()
    for df in frames:
        df[numeric] = ((df[numeric] - mean) / std).fillna(0)
    frames = [pd.get_dummies(df, dummy_na=True) for df in frames]
    return [df.reindex(columns=frames[0].columns, fill_value=0)
              .to_numpy(dtype=np.float32) for df in frames]
```

コンペティションのテーブル全体に対して `preprocess(raw_train, raw_test)` を実行すると、元の 79 個の属性が展開され、列の揃った `float32` の計画行列（design matrix）である `X_train` と `X_test`（各住宅につき 345 個の特徴量）が得られる。

```
X_train, X_test = preprocess(raw_train, raw_test)
X_train.shape, X_test.shape
```

```
((1460, 345), (1459, 345))
```

最後に、`KaggleHouse(d2l.DataModule)` は `preprocess` と対数ターゲットの中心化を組み合わせ、訓練分割における対数価格の平均 `self.offset`（≈ 12.02）を $\log(\text{SalePrice})$ から差し引くことで、ネットワークがステップ 1 から適切なベースラインで出発できるようにする（演習問題 2）。

```
class KaggleHouse(d2l.DataModule):
    """Features and centered log prices, fitted on the training rows."""
    def __init__(self, raw_train, raw_val, batch_size=64):
        super().__init__()
        self.save_hyperparameters(ignore=['raw_train', 'raw_val'])
        self.X_train, self.X_val = preprocess(raw_train, raw_val)
        self.num_inputs = self.X_train.shape[1]
        log_price = lambda df: np.log(
            df['SalePrice'].to_numpy(np.float32)).reshape(-1, 1)
        self.offset = log_price(raw_train).mean()  # Mean log price
        self.y_train = log_price(raw_train) - self.offset
        self.y_val = (log_price(raw_val) - self.offset
                      if 'SalePrice' in raw_val else None)

    def get_dataloader(self, train):
        X, y = ((self.X_train, self.y_train) if train
                else (self.X_val, self.y_val))
        if y is None:  # No prices to validate against
            return None
        return self.get_tensorloader((d2l.tensor(X), d2l.tensor(y)), train)
```

4.7.3 K 分割交差検証によるモデル選択

4.7.3.1 共有 5 分割と線形ベースライン

ラベル付き住宅が 1,460 軒しかないため、単一の 20% 検証分割（292 軒）では、フォールド間の標準誤差が log-RMSE で 0.01 の数倍に達する。そこでヘルパー関数 `k_fold_data(raw, k)` を用

いて raw を K 個のランダムなフォールドに分割し、検証フォールドが訓練特徴量の統計量に決してリークしないよう、ラウンドごとに個別の KaggleHouse データモジュールを構築する。

```
def k_fold_data(raw, k):
    """Split the table raw into k random folds; one data module per round."""
    folds = np.array_split(np.random.default_rng().permutation(len(raw)), k)
    return [KaggleHouse(raw.drop(index=raw.index[f]), raw.iloc[f])
            for f in folds]
```

事前に構築した folds のリストとモデル生成関数 make_model を受け取ると、k_fold は各フォールドで新しいモデルを max_epochs エポック訓練する。さらに、1 エポックごとのばらつきを抑えるために最後の 5 エポック分の二乗損失を平均してから平方根をとり、全フォールドにわたる訓練および検証 log-RMSE の平均を表示する。

```
def k_fold(folds, make_model, max_epochs=100):
    """Train a fresh model in each round; return the RMSEs of every fold."""
    rmse = []
    for fold in folds:
        trainer = d2l.Trainer(max_epochs=max_epochs)
        trainer.board.display = False
        trainer.fit(make_model(fold.num_inputs), fold)
        rmse.append([np.mean(trainer.history[f'{split}_loss'][-5:]) ** 0.5
                     for split in ('train', 'val')])  # Last five epochs
    train, val = np.array(rmse).T
    print(f'training RMSE {train.mean():.3f}, validation RMSE '
          f'{val.mean():.3f} (folds {val.min():.3f} to {val.max():.3f})')
    return train, val
```

ここで、共有の 5 分割 folds を 1 回だけ生成しておく。これにより、以降のすべての候補モデルがまったく同じ 5 組の訓練・検証分割で評価される。

```
folds = k_fold_data(raw_train, 5)
```

まず、make_linear を定義して d2l.LinearRegression のベースライン（100 エポック、バッチサイズ 64、学習率 $\eta = 0.02$）を用意しよう。

```
make_linear = lambda num_inputs: d2l.LinearRegression(lr=0.02)
```

k_fold(folds, make_linear) を実行すると、正則化なしの線形ベースラインが 5 つのフォールドすべてで訓練され、訓練および検証 log-RMSE のベクトルが linear に記録される。

```
linear = k_fold(folds, make_linear)
```

```
training RMSE 0.114, validation RMSE 0.145 (folds 0.129 to 0.182)
```

線形ベースラインは、5 分割の検証 log-RMSE で ~ 0.15（典型的な価格誤差で $\sim 15\%$）を達成し、対する訓練 log-RMSE は ~ 0.11 である。

4.7.3.2 重み減衰を伴う MLP

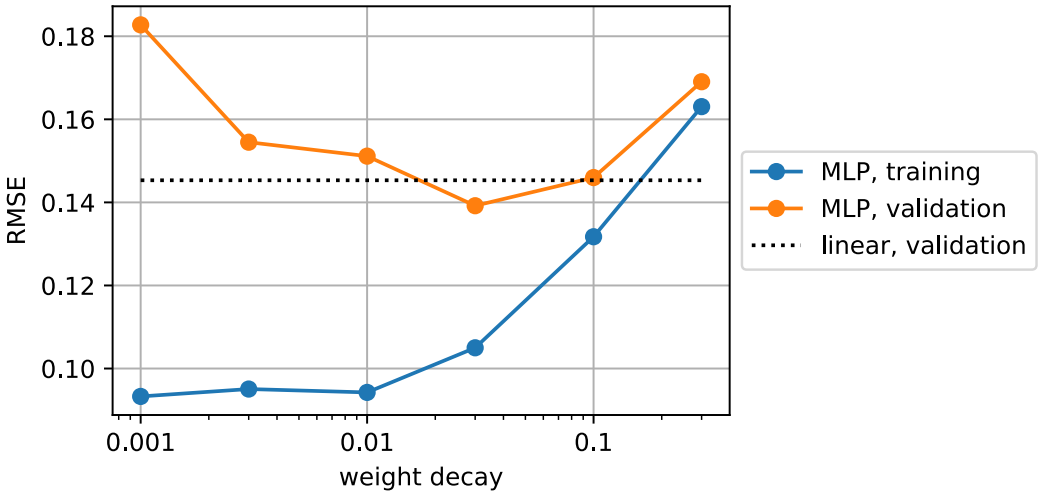
次に、HouseMLP を構築しよう。これは 64 個の ReLU ユニットを持つ隠れ層 1 層の MLP（パラメータ数は ～ 22,000 個、すなわち 1 フォールド内の訓練用住宅 1 軒あたり ～ 19 個のパラメータ）であり、He 初期化された重みと、重み行列のみ（バイアスを除く）に適用される ℓ_2 重み減衰 wd を備えている。

```
class HouseMLP(d2l.Module):
    """An MLP with one hidden layer, squared loss, and weight decay."""
    loss = d2l.LinearRegression.loss

    def __init__(self, num_hiddens, wd, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(1))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)

    def configure_optimizers(self):
        weights = [p for p in self.parameters() if p.dim() > 1]
        biases = [p for p in self.parameters() if p.dim() == 1]
        return torch.optim.SGD([{'params': weights, 'weight_decay': self.wd},
                                {'params': biases}], lr=self.lr)

make_mlp = lambda num_inputs, wd: HouseMLP(num_hiddens=64, wd=wd, lr=0.02)
```

ここで、wd を wds = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3] にわたって共有の 5 分割 folds 上で変化させ、訓練 RMSE と検証 RMSE の平均を線形ベースラインと比較してプロットする。

```
wds = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]
mlp = {}
for wd in wds:
    print(f'weight decay {wd}: ', end='')
    mlp[wd] = k_fold(folds, lambda num_inputs: make_mlp(num_inputs, wd))
d2l.plot(wds, [[mlp[wd][0].mean() for wd in wds],
               [mlp[wd][1].mean() for wd in wds], [linear[1].mean()] * 6],
         'weight decay', 'RMSE', xscale='log', fmts=('C0-o', 'C1-o', 'k:'),
         legend=['MLP, training', 'MLP, validation', 'linear, validation'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
weight decay 0.001: training RMSE 0.093, validation RMSE 0.183 (folds 0.146 to 0.277)
weight decay 0.003: training RMSE 0.095, validation RMSE 0.154 (folds 0.137 to 0.174)
weight decay 0.01: training RMSE 0.094, validation RMSE 0.151 (folds 0.138 to 0.168)
weight decay 0.03: training RMSE 0.105, validation RMSE 0.139 (folds 0.130 to 0.153)
weight decay 0.1: training RMSE 0.132, validation RMSE 0.146 (folds 0.139 to 0.159)
weight decay 0.3: training RMSE 0.163, validation RMSE 0.169 (folds 0.153 to 0.180)
```



5 分割の検証 RMSE の平均が最小となる重み減衰 `best` を選択し、フォールドのサンプリングノイズが相殺されるフォールドごとの対応のある差 `mlp[best][1] - linear[1]` も確認できる。

```
best = min(wds, key=lambda wd: mlp[wd][1].mean())
print(f'weight decay {best}: MLP minus linear, per fold',
      np.round(mlp[best][1] - linear[1], 3))
```

```
weight decay 0.03: MLP minus linear, per fold [-0.009  0.    -0.029  0.012 -0.006]
```

交差検証の曲線は教科書通りの U 字型を描く。弱い重み減衰（`wd = 0.001`）では訓練フォールドに過学習し（訓練 RMSE が ~ 0.09、検証 RMSE が ~ 0.15〜0.18 で線形回帰より悪い）、中程度の重み減衰（`wd = 0.03` または `0.1`）では検証 RMSE が ~ 0.13〜0.14 というスイートスポットに入り（大半のフォールドで線形ベースラインを上回る）、強すぎる減衰（`wd = 0.3`）では学習不足に陥る。

4.7.4 予測の提出

5 分割交差検証で選ばれた最良の重み減衰 `best` を用いて、ラベル付き住宅全 $1{,}460$ 軒で `HouseMLP` を再訓練し（あるいは 5 つのフォールドモデルをアンサンブルしてもよい。演習問題 5 を参照）、訓練セットの対数価格の平均 `data.offset` を足し戻し、指数変換でドル単位の予測値に戻して `submission.csv` に保存する。

```
data = KaggleHouse(raw_train, raw_test)
model = make_mlp(data.num_inputs, wd=best)
trainer = d2l.Trainer(max_epochs=100)
trainer.board.display = False
trainer.fit(model, data)
log_price = d2l.numpy(model(d2l.tensor(data.X_val))).reshape(-1) + data.offset
submission = pd.DataFrame({'Id': raw_test['Id'],
                           'SalePrice': np.exp(log_price)})
submission.to_csv('submission.csv', index=False)
submission.head()
```

```
     Id      SalePrice
0  1461  119006.625000
1  1462  154548.484375
2  1463  177151.687500
3  1464  201103.609375
4  1465  186113.750000
```

4.7.5 まとめ

実世界の表形式回帰では、データリークのない規律ある前処理が鍵を握る。具体的には、任意の行識別子の除去、数値カテゴリコードの文字列への変換、訓練分割の平均と標準偏差による数値列の標準化とそれに続く欠損値のゼロ補完、明示的な欠損性インジケータを伴うカテゴリ変数のワンホットエンコーディング、そして対数変換したターゲットの訓練平均による中心化である。すべての候補モデルを同一の $K = 5$ 交差検証フォールド（各ラウンドで 4 つの訓練フォールドのみを用いて前処理器を再適合させる）で評価することで、モデル同士を直接比較する際にフォールドのサンプリングノイズが相殺される。Ames Housing データセットでは、ℓ_2 重み減衰を調整することで、22,000 パラメータの ReLU MLP は過学習の状態（log-RMSE で ~ 0.15〜0.18）から線形ベースライン（~ 0.15）を超え、log-RMSE で ~ 0.13〜0.14 に到達する。もっとも、異種混合の表形式特徴量においては、勾配ブースティング決定木が依然として強力な競合手法である（演習問題 3）。

4.7.6 演習問題

1. [code] **平均値補完**。`preprocess` は欠損しているすべての数値を、その列の訓練セットの平均で置き換える。また、`pd.get_dummies` を `dummy_na=True` で呼び出すことで、すべてのカテゴリ特徴量に対して欠損値インジケータ列も追加される。

    1. このデータセットの中から、販売価格に対して欠損性がランダムではない数値特徴量を 1 つ特定せよ。
    2. その特徴量に対して平均値補完のみを適用するとどのような統計的バイアスが生じるか、また該当する住宅の予測がどの方向にずれるかを説明せよ。続いて、`preprocess` がすでに作成しているインジケータ列によって、線形モデルがこのバイアスを除去できるかどうかを判定せよ。
    3. 平均値補完を中央値補完に置き換え、それとは別に補完した各特徴量に対して二値の欠損性インジケータ列を追加せよ。これら 3 つの方式の交差検証誤差を比較せよ。

    Kaggle Learn の Intermediate Machine Learning の演習問題 2 を改変。

2. [code] **特徴量とターゲットのスケーリング**。`preprocess` から標準化を取り除け。`fillna(0)` が列の平均を補完するのは標準化された列に限られるため、代わりに欠損している数値を訓練セットの列平均で明示的に埋めよ。

    1. 本節と同じ学習率 0.02 で線形モデルの交差検証を再実行し、訓練損失の挙動を述べよ。
    2. 標準化していない線形モデルが発散せずに訓練できる最大の学習率を 1 桁（10 倍）の精度で求めよ。100 エポック後の交差検証誤差を標準化ありのパイプラインと比較し、両方の観測結果を説明せよ。
    3. 標準化を元に戻したうえで、対数価格から平均を引かずに訓練せよ。100 エポック後の交差検証誤差を中心化したターゲットの場合と比較し、その違いを説明せよ。
    4. 本節のパイプラインを用い、毎エポック後の検証フォールドにおける線形モデルの検証 RMSE と残差の平均を記録せよ。RMSE が跳ね上がるエポックでは何が動いているか。どの住宅もちょうど 44 個のワンホット特徴量が 1 に等しい（バイアスを含めれば

45 個の入力が 1 に等しい）。SGD の 1 ステップがいずれか単一の重みの変化量よりも平均予測を大きく変化させる理由を説明し、学習率を半分にするとこの跳ね上がりが消えるかどうかを確認せよ。

3. [code] **勾配ブースティング木**。このような表形式データでは、決定木のアンサンブルがニューラルネットワークよりも高い精度で予測できることが多い (Grinsztajn et al. 2022; Shwartz-Ziv and Armon 2022)。勾配ブースティング木モデル（たとえば scikit-learn の `GradientBoostingRegressor`、あるいはインストールされていれば XGBoost や LightGBM）を、`folds` の各フォールドの訓練行における中心化対数価格に適合させ、各フォールドでの検証 RMSE を計算せよ。その平均を本節のモデルと比較せよ。データのどのような性質が決定木に有利に働くか。

4. [code] **ターゲットリーク**。予測対象の事象が起きた後、あるいはその事象が起きた結果として初めて記録される特徴量は、実運用モデルには役立たないまま検証スコアだけを不当に押し上げる。

    1. このデータセットの元の特徴量の中から、売却後に初めて記入されうると考えられるものを少なくとも 1 つ特定し、交差検証ではなぜこの問題を検出できないのかを説明せよ。
    2. この仕組みを実験的に確認せよ。対数販売価格に小さなガウスノイズを加えた合成特徴量を追加して交差検証を再実行し、検証 RMSE がどう変化するかを報告せよ。

    Kaggle Learn の Intermediate Machine Learning の演習問題 7 を改変。

5. [code] **フォールドアンサンブルと再訓練の比較**。 4.7.4 節では、選択した MLP を全住宅で再訓練している。別の方法として、交差検証中に訓練した 5 つのモデルにそれぞれのラウンドの前処理を用いてテスト住宅を予測させ、その対数価格の予測を平均することもできる。

    1. このフォールドアンサンブルを実装し、その予測と再訓練モデルの予測の両方を提出せよ。2 つのリーダーボードのスコアを互いに比較するとともに、交差検証 RMSE とも比較せよ。
    2. アンサンブルの各モデルは全住宅の 5 分の 4 でしか訓練されていない。それにもかかわらず、全住宅で再訓練したモデルよりアンサンブルのほうが精度よく予測しうるのはなぜか。
    3. 再訓練モデルの訓練誤差が交差検証による推定値の代わりにならない理由を説明せよ。

6. [code] **ターゲットエンコーディング**。●**ターゲットエンコーディング**は、カテゴリ列の各カテゴリをそのカテゴリに属する訓練行の対数販売価格の平均で置き換える手法であり、`preprocess` が作成するカテゴリごとの列ではなく単一の列で特徴量を表現できる。

    1. 10 個を超えるカテゴリを持つすべてのカテゴリ列に対して、ワンホットエンコーディングの代わりに各フォールドの訓練行からカテゴリ平均を計算してターゲットエンコーディングを適用せよ。訓練行に出現しないカテゴリには全体の平均を与え、エンコード後の列を数値特徴量と同様に標準化せよ。その交差検証誤差をワンホットエンコーディングの場合と比較せよ。
    2. 上記のエンコーディングでは、各訓練行の値を計算する平均にその行自身のラベルが含まれている。代わりに訓練行をフォールド外でエンコードせよ。すなわち、各フォールドの訓練行をさらに 5 分割し、残りの 4 分割から計算したカテゴリ平均で各分割をエンコードせよ。交差検証誤差をサンプル内エンコーディングの場合と比較し、カテゴリのサイズ（行数）が 2 つのエンコーディングの差にどう影響するかを説明せよ。

7. **幾何平均**。●K 個のモデルからなるアンサンブルが住宅の対数価格 $z_1, \dots, z_K$ を予測するとき、指数変換の前にそれらを平均でき、これは価格空間での幾何平均に相当する。

1. $\exp\left(\frac{1}{K}\sum_{k=1}^{K} z_k\right)$ と価格空間での平均 $\frac{1}{K}\sum_{k=1}^{K} e^{z_k}$ を比較せよ。どちらがもう一方を決して上回らないかを示し、両者が一致する条件を述べよ。

2. 住宅の真の対数価格を $\log y$ とする。対数空間での平均が次式を満たすことを示せ。

$$\left(\log y - \frac{1}{K}\sum_{k=1}^{K} z_k\right)^2 \leq \frac{1}{K}\sum_{k=1}^{K} \left(\log y - z_k\right)^2 . \tag{4.22}$$

   また、価格空間での平均の対数も、すべての予測 $z_1, \dots, z_K$ に対して同じ上界を満たすかどうかを判定せよ。

Chapter 5

計算

ここまでは、モデルの構築、パラメータの管理、初期化、デバイス上での実行をフレームワークの裏側の処理に任せてきた。しかし、教科書レベルのプロトタイプから本番規模のシステムへと進むには、そのブラックボックスを開ける必要がある。80 億パラメータの言語モデルを 32 ビット浮動小数点の Adam で訓練する場合、活性化値を 1 つ保持する前の段階で、パラメータ、勾配、オプティマイザ状態（optimizer state）だけで 128 GB のメモリを要するうえ、数時間おきにハードウェア障害で訓練が中断される数千基の GPU 上で実行することになる。この規模では、フレームワークがモデルをどのように表現し、初期化し、計算に用い、永続化するかを理解することが不可欠である。

本章のあらゆる内容は、1 つの中心的なデータ構造の上に築かれている。すなわち、ニューラルネットワークとは**モジュールの木**（tree of modules）であり、その葉がモデルの**状態**（state、訓練可能なパラメータと固定バッファの両方）を保持し、それぞれが根からのパスによって指定されるという構造である。本章の前半では、この構造的な抽象化を展開し、フレームワークの各操作がそれをどのように走査するかを示す。まず、モジュールの階層構造、コンテナへの登録、残差スキップ接続、形状の遅延決定（lazy shape inference）、設定駆動のビルダから始める（5.1 節）。続いて状態木を開き、訓練可能なパラメータと訓練対象外のバッファを切り分け、fp32 の Adam 訓練におけるパラメータあたり 16 バイトというメモリ下限を導出し、ストレージを確保せずにパラメータ数を数え上げ、重みの共有と凍結を扱う（5.2 節）。次に、層ごとの分散規則から木全体および残差構造の初期化へと進み、スケーリングしない残差分岐が活性化値の分散を $(1+g^2)^N$ 倍に増幅させる理由と、$1/\sqrt{N}$ またはゼロによる出力射影（output projection）のスケーリングが深い積層モデルを安定させる仕組みを示す（5.3 節）。そのうえで、状態をもたない層、パラメータをもつ層（`RMSNorm`）、固定バッファをもつ層（`RandomFeatures`）、およびストレートスルー推定器（straight-through estimator）によるカスタム逆伝播規則を用いてフレームワークを拡張する（5.4 節）。

後半では、モデルの構造から数値表現、永続化、ハードウェア実行、および診断へと目を向ける。まず fp32、fp16、bf16 を比較し、16 ビットの重みでは微小な更新が丸めによって消えてしまう理由を説明したうえで、混合精度訓練（mixed-precision training）がいかに fp32 のマスター重みと 16 ビット演算、および fp16 の損失スケーリング（loss scaling）を組み合わせるかを示す（5.5 節）。次に、安全でない `pickle` アーカイブとコードを含まない `safetensors` ファイルを対比し、ビット単位で完全に再開できるアトミックな 5 要素の訓練チェックポイントを構築するとともに、事前学習済みバックボーンを読み込む際の欠落キーや予期しないキーを検査する（5.6 節）。最後に、テンソルとモデルを GPU 上に配置し、非同期コマンドキューの実行時間を正確に計測し、$O(\sqrt{K})$ の活性化チェックポイント（activation checkpointing）とあわせて GPU メモリの内訳を詳しく調べ（5.7 節）、擬似乱数シードの制御および GPU の決定論的実行を非侵襲的な順

伝播・逆伝播フックと組み合わせる（5.8 節）。

参考資料と関連文献

以下の資料は本章の各節に対応しており、それぞれ関連する節を併記している。特記のない限り、すべてオンラインで無料公開されている。

書籍

- Deep Learning with PyTorch (Stevens, Antiga & Viehmann)：PyTorch チームによる無料の PDF 版（アーカイブ版）。第 1 部でテンソル、ストレージ、自動微分、`nn.Module` を 5.1 節および 5.2 節と同じ水準で解説している。
- Machine Learning Systems (Vijay Janapa Reddi)：オンラインで無料公開。フレームワーク、データパイプライン、訓練インフラストラクチャ、効率的なデプロイメントなど、本章を取り巻くシステム面の視点を扱っている（5.7 節、5.8 節）。

講義コースと講義動画

- CMU 10-414/714: Deep Learning Systems (Chen & Kolter)：無料の講義と課題。自動微分、モジュール、初期化、GPU サポートを備えた小型フレームワーク「needle」を受講生自ら構築する（5.1 節、5.3 節、5.7 節）。
- Neural Networks: Zero to Hero (Andrej Karpathy)：無料の動画シリーズ。標準の Python のみを用いて、自動微分、モジュール、訓練ループをゼロから構築する（5.1 節、5.4 節）。
- fast.ai Part 2: Deep Learning Foundations (Howard et al.)：無料の講義コース。モジュール、初期化、混合精度を含め、テンソル演算から積み上げて訓練フレームワークを再構築する（5.1 節、5.3 節、5.5 節）。

チュートリアル・解説記事・インタラクティブ教材

- PyTorch internals (Edward Yang)：PyTorch 内部のテンソル、ストライド、ディスパッチ、自動微分を順を追って解説する記事であり、5.1 節より 1 段低い抽象度を扱っている。
- JAX: The Sharp Bits：純粋関数、明示的な PRNG キー、JIT の制約など、JAX の背後にある関数型設計を実践上の注意点としてまとめた解説である（5.2 節、5.8 節）。
- Train With Mixed Precision (NVIDIA)：fp16/bf16 演算と損失スケーリングに関するベンダー公式ガイドであり、5.5 節はその要点を凝縮したものである。
- What Every Computer Scientist Should Know About Floating-Point Arithmetic (David Goldberg)：あらゆる丸めやオーバーフローの問題（5.5 節）の根底にある標準的な参考文献である。
- Making Deep Learning Go Brrrr From First Principles (Horace He)：演算能力、メモリ帯域幅、フレームワークのオーバーヘッドを第一原理から解き明かす解説記事である（5.7 節）。
- safetensors (Hugging Face)：読み込み時にコードを実行せずに重みを共有するためのファイル形式と、それが pickle ベースのチェックポイントを置き換えた理由を解説するドキュメントである（5.6 節）。
- Reproducibility (PyTorch notes)：決定論的実行のためのフラグ、その計算性能上のコスト、および限界についての解説であり、5.8 節を補完する。

基礎となった論文

- Automatic Differentiation in Machine Learning: a Survey (Baydin, Pearlmutter, Radul & Siskind, 2018, **JMLR**)：フォワードモードおよびリバースモードの自動微分に関する決定版のサーベイ論文であり、5.4 節のあらゆるカスタム勾配の背後にあるアルゴリズムを解説している。

- Mixed Precision Training (Micikevicius et al., 2018)：fp16・マスター重み・損失スケーリングを組み合わせたレシピ（5.5 節）の原典となった論文である。
- PyTorch: An Imperative Style, High-Performance Deep Learning Library (Paszke et al., 2019)：即時実行、自動微分、キャッシングアロケータなど、5.1 節および 5.7 節の命令型スタイルを支える設計思想を述べた論文である。

5.1 モジュールとモデル構築

4.2 節の多層パーセプトロン（multilayer perceptron、MLP）が持つパラメータ配列はわずか 4 個であり、手作業でも管理できるほど小さかった。しかし、実世界のアーキテクチャははるかに複雑である。コンピュータビジョン向けの現代的な残差ネットワーク（residual network）は何十層、何百層もの層を積み重ねており、Llama 3 8B のような言語モデル（language model）では (Grattafiori et al. 2024)、32 個の同一の Transformer ブロックにわたる 80 億個のパラメータの重みが 291 個もの個別の配列に分散している。これらの配列は 1 つ残らずオプティマイザに渡り、チェックポイントに書き出され、GPU 上へ転送されなければならない。さらに厄介なことに、配列が 1 つ漏れても構文エラーにはならず、訓練も保存もされないまま静かに放置されてしまう。

深層学習フレームワークは、この煩雑な管理の問題を**モジュール**という単一の再帰的な抽象化によって解決する。モジュールは、個々の層、層のグループ、あるいはモデル全体を表すことができる。モジュールはパラメータと子モジュールを所有し、入力を出力へ写像する順伝播の計算を定義する。モジュールは他のモジュールの内部に入れ子にできるため、あらゆるアーキテクチャはモジュールの木構造をなす（Figure 5.1）。各子モジュールが親モジュールに正しく**登録**（register）されてさえいれば、どのフレームワークもこの木を走査することで全パラメータを自動的に見つけ出す。

本節は実践的な内容であり、すでにカスタムモジュールを実装した経験がある読者は基礎的な部分を流し読みしてもよい。ただし、登録の仕組みとそれが静かに失敗する挙動だけは読み飛ばさないようにしよう。これは非常に踏みやすいバグとして知られているからである。また本節では、残差接続（residual connection）などのカスタム順伝播計算の書き方、遅延初期化（lazy initialization）によって実行時に形状を自動的に決める仕組み、そしてアーキテクチャ上のハイパーパラメータをコンパクトな設定（configuration）オブジェクトへ分離する方法も示す。後続の節では、同じ木の走査を再利用して、パラメータ数の集計（5.2 節）、重みの初期化（5.3 節）、チェックポイントの保存（5.6 節）、および GPU 上へのモデル配置（5.7 節）を扱う。

```
from dataclasses import dataclass
```

5.1.1 モジュールの抽象化

5.1.1.1 逐次コンテナとレジストリ

2.4 節以降に記述してきたモデルはすべてモジュールであった。すなわち、`d2l.Module` は各フレームワークにおける基底モジュールクラスのサブクラスである。層のパイプラインを最も手早く組み立てる方法は、子モジュールを順番につなぎ合わせる**逐次コンテナ**（sequential container）を使うことである。以下では、4.2 節と同様に 256 個の ReLU ユニットを持つ 1 つの隠れ層と 10 個の出力を備えた MLP を構築し、それぞれ 20 個の特徴量を持つ 2 個の例からなるミニバッチを入力する。

PyTorch では基底クラスが `nn.Module` であり、`nn.Sequential` は引数として渡された層を順番につなぐ。

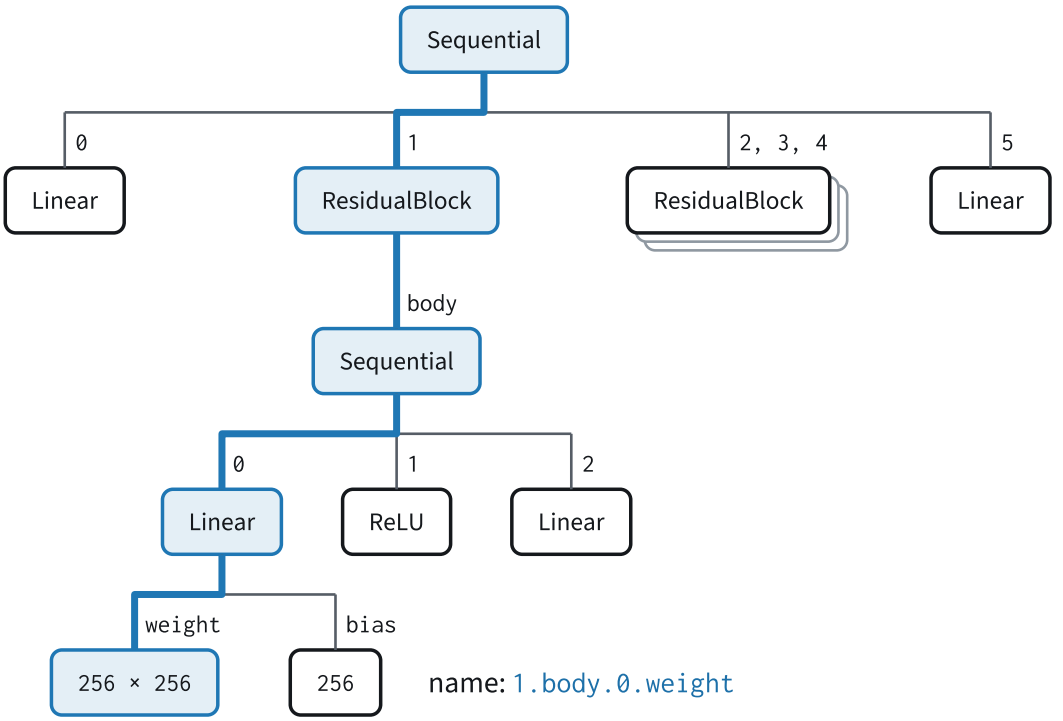


Figure 5.1: 本節の最後で構築する残差 MLP のモジュール木。各辺には親が子を登録した際の名前が付いており、根からパラメータに至る名前のパスがそのパラメータの名前（ここでは `1.body.0.weight`）になる。ブロック 2〜4 はブロック 1 と同じ構造を繰り返す。

```
net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

内部では、コンテナは名前を子モジュールに対応付ける順序付きの**レジストリ**に子モジュールを格納しており、順伝播ではそのテーブルを挿入順に反復処理する。

`named_children` は `(name, module)` の組（`'0'`、`'1'`、`'2'`）を生成する。

```
dict(net.named_children())
```

```
{'0': Linear(in_features=20, out_features=256, bias=True),
 '1': ReLU(),
 '2': Linear(in_features=256, out_features=10, bias=True)}
```

モデルに対するあらゆる木の走査は、このレジストリを起点とする。オプティマイザ用のパラメータ収集、ディスクへの重みのシリアライズ、GPU へのテンソル転送はいずれも、登録された子モジュールを再帰的にたどってすべての葉パラメータに到達する。これらの操作にとって、未登録の層は存在しないも同然である。

5.1.1.2 モジュールクラスのサブクラス

逐次コンテナは一直線の連鎖を簡潔に扱えるが、多くのアーキテクチャは分岐や合流を含んだり、構成要素を再利用したりする。単純な直列構造を超えるモデルでは、フレームワークの基底モジュールを継承してサブクラスを作り、子モジュールを生成するコンストラクタと、それらの間をデータがどう流れるかを規定する順伝播メソッドの 2 つを実装する。

PyTorch では、順伝播メソッドは `forward` である。

先ほどと同じ MLP を改めて以下に示す。

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.out = nn.Linear(256, 10)

    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

`MLP` をインスタンス化して `net(X)` を呼び出すと、定義したカスタム順伝播メソッドへと処理が渡り、2×20 の入力バッチ `X` が形状 `(2, 10)` の出力テンソルへと写像される。

```
net = MLP()
net(X).shape
```

```
torch.Size([2, 10])
```

裏側では 2 つの仕組みが重要な役割を果たしている。第 1 に、`self.hidden = ...` という代入は単なる Python の属性保存ではない。基底モジュールが `__setattr__` を介して代入を捕捉し、その値がモジュールであることを検知して、属性名（`hidden` と `out`）のもとで登録する。この自動登録だけで、両方の層のパラメータがオプティマイザへ渡るようになる。第 2 に、逆伝播メソッドを自分で書く必要は一切ない。1.5 節で扱った自動微分（automatic differentiation）が、順伝播で実行された演算から直接勾配を導出するためである。

モジュールの呼び出しは常に `net(X)` とする。

`forward` を直接呼び出すと、`__call__` が実行するフックがスキップされてしまう（5.8 節）。

5.1.2 モジュールコンテナ

5.1.2.1 コンテナの実装

逐次コンテナの役割は、追加される各子モジュールを登録することと、それらに順番に入力を通すことの 2 つだけである。最小限のバージョンを自作してみることで、登録が正確にどのように機能し、どこで問題が生じうるのかが明確になる。

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

`add_module` は各子モジュールを文字列インデックスで登録し、`children()` はレジストリを挿入順に反復処理する。

自作した `MySequential` は、組み込みのコンテナとまったく同じように動作する。

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X).shape
```

```
torch.Size([2, 10])
```

5.1.2.2 通常のリストに入れた層

モジュール登録に不慣れな人がつい書いてしまうように、通常の Python の `list` に層を格納するとどうなるだろうか。順伝播ではそのリストをループ処理できるものの、通常の `list` はモジュールではない。それでもフレームワークはリスト内部の層を見つけ出せるのだろうか。

```
class PlainListMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = [nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10)]
```

```
    def forward(self, X):
        for layer in self.layers:
            X = layer(X)
        return X

net = PlainListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 0)
```

順伝播は成功するにもかかわらず、net.parameters() は空になる。__setattr__ が通常の Python の list をモジュールコンテナではなく通常の属性として扱うからである。torch.optim.SGD が拒否する完全に空のモデルよりも厄介なのは、外側の層が属性で内側の層が通常のリストに入っている、**一部だけ**登録されたモデルである。SGD を 1 ステップ実行し、どの重みが更新されるか確認してみよう。

```
class PartlyRegisteredMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.first = nn.Linear(20, 256)
        self.blocks = [nn.Linear(256, 256), nn.Linear(256, 256)]  # Plain list
        self.last = nn.Linear(256, 10)

    def forward(self, X):
        X = F.relu(self.first(X))
        for block in self.blocks:
            X = F.relu(block(X))
        return self.last(X)
```

訓練中にどの層が実際に更新されるかを確かめるため、PartlyRegisteredMLP の 4 つの重み行列すべてを before に記録し、net(X).sum() に対して 1 回の逆伝播と 1 ステップの SGD を実行したうえで、各層の重みを before と比較する。

```
net = PartlyRegisteredMLP()
layers = [net.first, *net.blocks, net.last]
before = [layer.weight.clone() for layer in layers]
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).sum().backward()
optimizer.step()
[not torch.equal(b, layer.weight) for b, layer in zip(before, layers)]
```

```
[True, False, False, True]
```

警告もエラーも一切出ないまま、外側の重みだけが更新された。誤差逆伝播法により勾配は self.blocks を通って self.first まで伝わったが、オプティマイザは中間の層をまったく認識していなかったのである。リストを nn.ModuleList（辞書の場合は nn.ModuleDict）でラップすれば、すべての子モジュールが正しく登録される。

```
class ModuleListMLP(PlainListMLP):
    def __init__(self):
```

```
        super().__init__()
        self.layers = nn.ModuleList(self.layers)

net = ModuleListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 7946)
```

これで 7,946 個すべてのパラメータが登録された。一直線の連鎖には `nn.Sequential` を使い、`nn.ModuleList` や `nn.ModuleDict` は `forward` メソッド内で子モジュールを手動でループ処理したりインデックス参照したりする場合に使う。

5.1.3 順伝播の計算

モジュールの順伝播メソッドは通常の Python コードであるため、任意のテンソル演算、ループ、条件分岐、スキップ経路を呼び出すことができる。この柔軟性を活かして残差接続を実装し、続いてフレームワークが訓練ステップをコンパイルする際にどのような制約が生じるかを見ていこう。

5.1.3.1 残差接続

単純な直列構造ではないアーキテクチャのうち、広く使われている最も基本的なものが、ブロックの入力をその出力へ直接加算する**残差接続**である (He et al. 2016b)。ここでは間に ReLU を挟んだ 2 つの全結合層（fully connected layer）からなるブロックの**本体**を f と書くと、このブロックは次の式を計算する。

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x}). \tag{5.1}$$

$f(\mathbf{x}) = \mathbf{0}$ であれば常にこのブロックは恒等写像になるため、深い積層を通して信号がそのまま伝わり、各ブロックは表現をゼロから作り直すのではなく補正分 $f(\mathbf{x})$ だけを学習すればよい。7.4 節の ResNet アーキテクチャはこのパターンに基づいている。残差接続のために特別な組み込み層を用意する必要はどのフレームワークにもなく、カスタムモジュールでは `X + self.body(X)` というわずか 1 行で書ける。

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)
```

Figure 5.2 にその配線を示す。$\mathbf{x}$ と $f(\mathbf{x})$ を加算するには両者の形状が一致している必要があるため、残差ブロックは単一の隠れ層の幅 `num_hiddens` を受け取る。

```
block = ResidualBlock(24)
block(torch.randn(2, 24)).shape
```

```
torch.Size([2, 24])
```

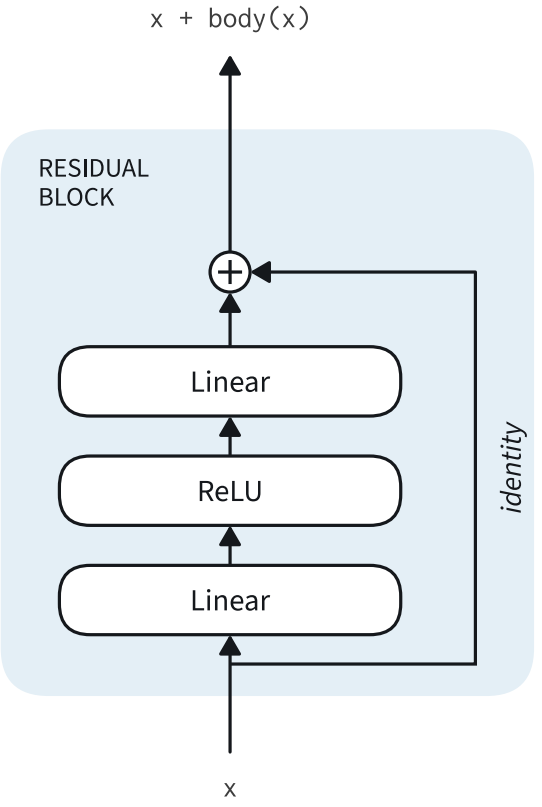


Figure 5.2: 残差ブロック。入力は本体（Linear、ReLU、Linear）と恒等経路に分岐し、加算によって再び合流する。

順伝播メソッドは、入力でも訓練可能なパラメータでもない値に依存することもある。スカラー定数は通常の Python 属性として保持できる一方、データから追跡されるテンソル（入力の移動統計量など）は、オプティマイザに更新されることなく、モデルとともにチェックポイントに保存されデバイス間を移動しなければならない。こうした訓練対象外の**バッファ**を登録する方法は 5.2.1.2 節で扱う。

5.1.3.2 即時実行とコンパイル実行

ここまでの順伝播は通常の Python と同様に、演算を 1 つずつ実行してきた。しかし、Python のオーバーヘッドをなくして演算を融合するために、フレームワークは計算を一度トレースし、以降の呼び出しではコンパイル済みのグラフを再生できる。この高速化にはトレードオフがあり、コンパイル実行では順伝播内部でのデータ依存の Python 制御フローが制限される。

Gluon の `Block` と PyTorch の `nn.Module` はどちらもデフォルトで**即時**（eager）に実行され、Python の文を 1 つずつ処理するため、データ依存の制御フローも NumPy とまったく同じように動作する（計算グラフのコンパイルについては第 13 章で扱う）。

5.1.4 遅延初期化

5.1.4.1 データからの形状決定

多くの層ライブラリでは、構築時に出力次元だけを指定しておき、最初のミニバッチから入力幅を自動的に決めることができる。

最初の順伝播の前には、`nn.LazyLinear(256)` は形状を持たない `UninitializedParameter` を保持している。

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
net[0].weight
```

```
<UninitializedParameter>
```

最初のミニバッチ `X` をネットワークに通すと、層から層へと形状が連鎖的に決まり、すべての重みテンソルが実体化される。

```
net(X)
net[0].weight.shape
```

```
torch.Size([256, 20])
```

5.1.4.2 ドライラン後の初期化

遅延層を使えば、深いモデルですぐにミスのもとになる層から層への入力次元の手計算から解放される。その代償として、データがネットワークを流れるまでパラメータが存在しないため、最初の順伝播より前にはパラメータ数を調べたり、Xavier 初期化（4.4.2.1 節）のような形状に依存するイニシャライザを実行したりできない。したがって遅延モデルでは、イニシャライザをあらかじめ記録しておくか、すべての形状を実体化させる**ドライラン**（出力を捨てるダミーのミニバッチに対する 1 回の順伝播）の後に適用する必要がある。

`d2l.Module.apply_init` は勾配を追跡せずに 1 回の順伝播を実行して遅延形状を実体化し、続いて `self.apply` によりモジュール木全体へイニシャライザのコールバックを適用する。

```
@d2l.add_to_class(d2l.Module)
def apply_init(self, inputs, init=None):
    """Run a dry pass to fix lazy shapes, then apply `init`."""
    with torch.no_grad():
        self(*inputs)
    if init is not None:
        self.apply(init)
```

nn.Module.apply(fn) は木に含まれるすべてのモジュールに対して、親より先に子の順で fn を呼び出す。これもまた木の走査によって実現される処理の 1 つである。Xavier の一様イニシャライザを用いた apply_init の例を以下に示す。

```
class TinyMLP(d2l.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(),
                                 nn.LazyLinear(10))

def init_xavier(module):
    if isinstance(module, nn.Linear):
        nn.init.xavier_uniform_(module.weight)

model = TinyMLP()
model.apply_init([X], init_xavier)
model.net[0].weight.shape
```

```
torch.Size([256, 20])
```

遅延層と明示的な次元指定はどのように使い分けるべきだろうか。未知のデータセットで素早くプロトタイピングを行う際には、遅延層が便利である。一方、パラメータ数を数えたり、プログラムから初期化したり、ダミーデータなしで設定ファイルからインスタンス化したりしたい本番用のモデルでは、入力幅と出力幅を明示的に宣言する方が見通しがよく、本章の残りの部分でもこの流儀を採用する。

5.1.5 設定からのモデル構築

5.1.5.1 設定とビルダ

保存された重みテンソルだけから、訓練済みモデルを復元できるだろうか。完全には不可能である。訓練済みモデルは、数値配列とそれを解釈するコードの両方で構成されているからである。先ほどまでに書いた幅や深さはすべてコンストラクタ呼び出しに直接ハードコードされていたが、重みファイルはそうした選択を間接的にしか反映しない（テンソルの形状から層の幅はわかっても、ReLU か tanh かといったパラメータを持たない選択は配列のどこにも残らない）。

したがって、実験ごとに変わるアーキテクチャ上のハイパーパラメータは、評価指標とともに記録したりチェックポイントと並べて保存したりできる専用の**設定**オブジェクト（**config**）にまとめるべきであり、トポロジーそのもの（各ブロックが何を計算し、ブロック同士がどう接続されるか）はコード側に置く。以下では、Python の dataclass でアーキテクチャの 4 つのフィールドを保持し、ファクトリ関数 build によって任意の設定を完全なモジュール木へと変換する。

```
@dataclass
class MLPConfig:
    num_inputs: int = 784
    num_hiddens: int = 256
    num_blocks: int = 4
    num_outputs: int = 10

def build(cfg: MLPConfig) -> nn.Module:
    blocks = [ResidualBlock(cfg.num_hiddens) for _ in range(cfg.num_blocks)]
    return nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                         *blocks, nn.Linear(cfg.num_hiddens, cfg.num_outputs))
```

1 つの設定が、num_inputs から num_hiddens への入力射影（projection）、num_blocks 個の残差ブロック、および出力ヘッドからなるアーキテクチャのインスタンスを完全に規定する。なお、blocks は一時的な Python リストとして作られているが、逐次コンテナへアンパックして渡すことで各ブロックが即座に登録されるため、ここではまったく安全である。すべての層が入力幅と出力幅の両方を指定しているため、ドライランなしでただちにパラメータを調べたり数えたりできる。

```
net = build(MLPConfig())
[(name, tuple(p.shape)) for name, p in net.named_parameters()][:4]
```

```
[('0.weight', (256, 784)),
 ('0.bias', (256,)),
 ('1.body.0.weight', (256, 256)),
 ('1.body.0.bias', (256,))]
```

構築されたモデルは、784 次元ベクトル（平坦化した Fashion-MNIST 画像など）のミニバッチを 10 個の出力ロジットへ写像する。

```
net(torch.rand(2, 784)).shape
```

```
torch.Size([2, 10])
```

5.1.5.2 パラメータ数の集計

設定があれば、メモリを確保する前にモデルの正確なパラメータ数を紙の上で計算することもできる。入力射影は $784 \cdot 256 + 256 = 200{,}960$ 個のパラメータを持ち、各残差ブロックは 2 つの全結合層で合計 $2 \cdot (256 \cdot 256 + 256) = 131{,}584$ 個のパラメータを含み、出力層が $256 \cdot 10 + 10 = 2{,}570$ 個を加えるため、次のようになる。

$$200{,}960 + 4 \cdot 131{,}584 + 2{,}570 = 729{,}866 \tag{5.2}$$

全体ではこれだけのパラメータ数になり、その大半を残差ブロックが占めている。モデルをスケールアップするには、モデルのコードを書き換えるのではなく設定の 2 つのフィールドを変更するだけでよく、パラメータ木を走査すればこの計算が正しいことを検証できる。

```
for cfg in (MLPConfig(), MLPConfig(num_hiddens=512, num_blocks=8)):
    n = sum(p.numel() for p in build(cfg).parameters())
    print(f'num_hiddens={cfg.num_hiddens}, num_blocks={cfg.num_blocks}: '
```

```
            f'{n:,} parameters')
```

```
num_hiddens=256, num_blocks=4: 729,866 parameters
num_hiddens=512, num_blocks=8: 4,609,546 parameters
```

隠れ層の幅を 2 倍にすると各ブロック内のパラメータ数は 4 倍になり、ブロック数を 2 倍にするとその合計がさらに 2 倍になるため、大きい方の設定は 6 倍以上のパラメータを含むことになる。実世界の基盤モデル（foundation model）もまったく同じパターンに従っており、冒頭で挙げた 8B の言語モデルは、コンパクトな設定（たとえば 32 ブロック、隠れ層の幅 4,096）とモジュール木を実体化するビルダコードの組によって規定される。

なお、設定が保持するのはハイパーパラメータであり、実装コードではない点に注意が必要である。チェックポイントからモデルを復元するには、重み、設定、そして対応する `build` 関数およびブロック定義の 3 つすべてが揃っていなければならない。5.6 節では重みと設定を一緒に保存し、コードはバージョン管理に任せる。

ここまではパラメータの場所を特定して数を集計しただけで、その中身はまだ調べていない。5.2 節では、モデルが所有するテンソルの中身を開き、訓練がどれだけのメモリを消費するかを計算し、重みを共有したり凍結したりする方法を見ていく。

5.1.6 まとめ

モデル構築の根幹をなすのは 3 つの考え方である。第 1 に、すべてのニューラルネットワークはモジュールの木であり、各パラメータは根からのパスによって一意に指定される。第 2 に、オプティマイザ、チェックポイント、デバイス転送はいずれも登録された子モジュールを走査して動作するため、未登録のコンテナの内部に層を隠してしまうと、訓練や保存の対象から静かに除外される。第 3 に、可変な次元を軽量な設定へ切り出して `build` 関数と組み合わせることで、アーキテクチャの記録、スケールアップ、再現が容易になる。これらの基礎の上に、カスタムの順伝播メソッドは Python ならではの柔軟性をもたらし（ただしコンパイル実行下ではトレース上の制約を受ける）、遅延層は形状の生成をドライランまで遅らせる引き換えとして素早い実験に利便性を提供する。

5.1.7 演習問題

1. [code] **未登録のモジュール**。`PlainListMLP` 内の通常のリストは、その層を `net.parameters()`、`net.to(device)`、および `net.state_dict()` の対象外にしてしまう。さらに次の 3 つの操作がモデルに何をもたらすかを予測し、実際に検証せよ。
    1. `net.apply(init_xavier)` を呼び出し、`net.layers[0]` の重みが変化するか確認せよ。
    2. `nn.Dropout(0.5)` をリスト内の `nn.ReLU()` の後に追加し、`net.eval()` を呼び出して、同じ入力をモデルに 2 回通せ。
    3. メソッド `load_state_dict` は `state_dict` の逆操作であり、保存されたテンソルの辞書をモデルへコピーする。`net.load_state_dict(ModuleListMLP().state_dict())` を呼び出せ。
    4. これらの失敗のうち、静かに進行するものはどれで、エラーを送出するものはどれか。同じ登録漏れという原因から 3 つすべてを説明せよ。
2. **残差ブロックの幅の拡大**。`ResidualBlock` は入力幅と出力幅が一致していることを要求する。あるブロックが幅 d をより大きな幅 d' へ写像するものとし、その本体が幅 d' の層で終わり、スキップ経路も幅 d' を出力しなければならないとする。このスキップ経路を構築する方法を、パラメータを持つ場合と持たない場合でそれぞれ 1 つずつ挙げよ。各方法に

ついて、追加されるパラメータ数を求め、どの出力座標が引き続き恒等経路を受け取るかを答えよ。

3. [code] **ゼロからのモジュール実装。**●フレームワークの基底クラスを一切使わず、通常の Python と NumPy のみを用いて、配列をラップする `Parameter` クラスと、最小限の `Module` クラス（`__setattr__` によって代入されたすべての `Parameter` または `Module` を登録するもの）を実装せよ。`Module` には、登録された子モジュールを再帰的にたどる `parameters()` メソッドを持たせよ。さらに、重みとバイアスを保持する `Linear` を `Module` のサブクラスとして実装し、その 2 つのインスタンスから 2 層の MLP を構築して、`parameters()` が 4 つすべての配列を返すことを確認せよ。続いて、その 2 つの層を通常のリストに格納して `parameters()` が何を返すかを確認し、リスト版でも 4 つすべての配列を見つけられるように `Module` を拡張せよ。

   CMU 10-714、Homework 2、問題 2 を改変。

4. [code] **遅延入力幅。** `net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))` を構築し、20 個の特徴量を持つバッチを通した後、30 個の特徴量を持つバッチを通せ。最初の順伝播が各層に何をしたかという観点から結果を説明せよ。さらに、最初の層以降のすべての層を共有しつつ、どちらの幅の入力も受け取れるモデルを述べよ。

5.2 パラメータ、状態、メモリ

前節では、フレームワークがすべてのパラメータを自動的に見つけられるよう、モデルをモジュールの木構造として構成した。これまでは、モデルが他に何を保持しているかや、訓練にどれだけのメモリが必要かを問うことなく、こうしたテンソルの管理をフレームワークの裏側の処理に任せておけばよかった。小規模なうちは、そうした細部が問題になることはめったにない。しかし大規模になると、あらゆる設計判断がそれに左右される。5.1 節で取り上げた Llama 3 のような 80 億パラメータのモデルを考えてみよう。その重みを 32 ビット浮動小数点数で格納するだけでも 32 GB を要するが、これを**訓練**するには、データを 1 例も処理する前の段階でその 4 倍（128 GB）ものメモリが必要になる。すべてのパラメータに対応する勾配が必要であるうえ、Adam オプティマイザ（9.5 節）が勾配と二乗勾配の移動平均のために同じサイズのテンソルをさらに 2 つ保持するからである。これだけで、すでに 80 GB GPU のメモリを超えてしまう。

このようなメモリ予算を正確に計算するには、モデルが所有するすべてのテンソルとそれぞれの役割を把握する必要がある。日常的なワークフローも、まったく同じ一覧に依存している。チェックポイントには順伝播が依存するすべての永続テンソルを記録しなければならず、GPU への転送ではそれらをまとめて移動する必要があり、ファインチューニングではオプティマイザが更新してよいテンソルを明確に指定しなければならない。

フレームワークは、これらすべての操作を**登録**（registration、5.1 節）によって統一している。各層が構造化された名前の下で自身のテンソルをモジュール木（module tree）に登録することで、オプティマイザ、チェックポイント保存処理、デバイス転送ルーチンがその単一の木を走査できるようになる。本節では、この状態木（state tree）を 3 つのパートに分けて見ていく。第 1 に、訓練可能な**パラメータ**と訓練対象外の**バッファ**を区別する。第 2 に、訓練時のメモリをパラメータあたりのバイト数で見積もり、ストレージを確保せずにパラメータ数を数える方法を示す。第 3 に、言語モデリングや転移学習（transfer learning）の随所で用いられる 2 つの代表的なパラメータのパターンとして、重み結合（weight tying）とパラメータ凍結（parameter freezing）を取り上げる。

```
import torch
from torch import nn
```

5.2.1 状態木

5.2.1.1 パスによるアクセス

本節を通して用いる例は、コンパクトな残差 MLP である。これは、20 個の特徴量を 64 個の隠れユニット（hidden unit）に射影する入力層、幅 64 の残差ブロック（residual block）2 つ、そして 10 ユニットの出力層からなる。このノートブック単体で動作するよう、5.1 節の `ResidualBlock` をここで再定義しておく。各ブロックは、2 層の**本体**（body）の出力を入力に直接加算する。

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)

net = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                    ResidualBlock(64), nn.Linear(64, 10))
X = torch.randn(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

パラメータはモジュール木の葉に位置する。任意の葉に至るドット区切りのパス（Figure 5.1）は Python でのインデックス参照に対応しており、デバッグ時に単一の層を調べるのに便利である。一方、オプティマイザやチェックポイントの書き出し処理は**すべて**の葉を一度に必要とするため、各フレームワークにはすべてのパラメータをそのパスとともに取得する木全体の走査手段が用意されている。

PyTorch では `named_parameters()` がこれにあたる。各パラメータはモジュールシステムに自身を訓練可能として登録するテンソルである `nn.Parameter` であり、その勾配 `.grad` は誤差逆伝播法によって値が格納されるまで `None` のままである。

この残差 MLP には合計 12 個のパラメータテンソルが含まれている。以下では、その総数とともに、最初の残差ブロックに属する 4 つのテンソルを確認する。

```
named = list(net.named_parameters())
len(named), [(name, p.shape) for name, p in named[2:6]]
```

```
(12,
 [('1.body.0.weight', torch.Size([64, 64])),
  ('1.body.0.bias', torch.Size([64])),
  ('1.body.2.weight', torch.Size([64, 64])),
  ('1.body.2.bias', torch.Size([64]))])
```

`1.body.0.weight` のような名前は、子モジュール `1`（`net` の最初の残差ブロック）から `body` とその最初の層 `0` を経て `weight` に至るパス（`net[1].body[0].weight`）をたどっている。Gluon では `grad_req == 'write'` が訓練可能なパラメータを表し、PyTorch ではモデルにバッファがない限り `state_dict()` のキーは `named_parameters()` と一致する。

```
list(net.state_dict()) == [name for name, _ in net.named_parameters()]
```

```
True
```

フレームワークのあらゆるユーティリティは、これらのパスの下で同じ状態木を走査する。上記の等価性が成り立つのは、この残差 MLP 内のすべてのテンソルがたまたま訓練可能だからであり、すべての層でそうとは限らない。

5.2.1.2 パラメータとバッファ

訓練セットから一度だけ計算した特徴量ごとの平均と標準偏差を用いて、入力を標準化するモデルを考える。これらの統計量はチェックポイント内に保存しなければならず（保存しなければ推論時の予測が誤ったものになる）、重みとともに GPU へ移動する必要もある。しかし、これらは学習される重みではなくデータの統計量であるため、オプティマイザが勾配降下法（gradient descent）によって更新しては**決して**ならない。モジュールの走査では通常の Python 属性は無視されるため（5.1 節）、通常の属性として保持するだけではうまくいかない。バッチ正規化（batch normalization、7.3 節、(Ioffe and Szegedy 2015)）も、まったく同じ種類の状態に依存している。すなわち、訓練時の順伝播で更新され、推論時に適用される移動平均と移動分散である。

このような永続的だが訓練対象外のテンソルは一般に**バッファ**と呼ばれ、どのフレームワークでも登録が可能になっている。登録しておけば、チェックポイント保存やデバイス間の移動には含まれる一方、オプティマイザからはスキップされる。ここから、層内のあらゆるテンソルに対して明快な 3 通りの使い分けが導かれる。オプティマイザで更新すべきものは**パラメータ**として登録し、勾配は不要だがチェックポイントやデバイス転送で必要なものは**バッファ**として登録し、1 回の順伝播・逆伝播の間だけ存在するもの（4.3 節で扱う誤差逆伝播法（backpropagation）のためにキャッシュされる中間活性化値（activation）など）は通常のローカル変数のままにする。層の幅のようなアーキテクチャ上の定数は、通常の Python 属性のままでよい。後に参照できるよう、Table 5.1 にこれらの規則をまとめる。

Table 5.1: 層内の 4 種類の値と、オプティマイザ・チェックポイント・デバイス移動におけるそれぞれの扱い。

種類	例	オプティマイザによる更新	チェックポイントへの保存	モデルと一緒の移動
パラメータ	重み、バイアス	あり	あり	あり
バッファ	入力統計量、BatchNorm の移動平均	なし	あり	あり
通常の属性	層の幅	なし	なし	なし
ローカル変数	中間活性化値	なし	なし	なし

具体例として、事前計算された特徴量ごとの平均と標準偏差で入力を正規化してから線形層を適用する `Standardizer` モジュールを構築する。

`self.register_buffer(name, tensor)` は訓練対象外のバッファを登録する。

```
class Standardizer(nn.Module):
    def __init__(self, mean, std):
        super().__init__()
        self.register_buffer('mean', mean)
```

```
            self.register_buffer('std', std)
            self.out = nn.Linear(4, 2)

        def forward(self, X):
            return self.out((X - self.mean) / self.std)

    sample = torch.randn(100, 4) * torch.arange(1., 5.)
    standardizer = Standardizer(sample.mean(0), sample.std(0, correction=0))
    list(standardizer.state_dict())
```

```
['mean', 'std', 'out.weight', 'out.bias']
```

訓練可能な状態と訓練対象外の状態を調べると、チェックポイントからは 4 つのテンソルすべてが見える一方、オプティマイザからは線形層の重みとバイアスしか見えないことが確認できる。

```
    [name for name, _ in standardizer.named_parameters()]
```

```
['out.weight', 'out.bias']
```

登録が制御するのはチェックポイント保存と最適化だけではない。dtype のキャスト（5.5 節）や GPU への転送（5.7 節）といったモデル全体に対する操作もまったく同じ状態木を走査し、パラメータとバッファを同様に移動する一方で、未登録の通常の属性はそのまま残す。

5.2.2 訓練時のメモリ

5.2.2.1 パラメータあたりのバイト数

この残差 MLP を訓練するには、どれだけのメモリが必要だろうか。モデルとオプティマイザの状態だけに注目すると、メモリ量はパラメータごとの 3 つの要素、すなわちパラメータそのもの、その勾配、そしてパラメータごとのオプティマイザ状態に比例する。通常のミニバッチ SGD は状態を保持しないため、32 ビット浮動小数点数（fp32、1 数値あたり 4 バイト）ではパラメータあたり 8 バイトを消費する。Adam はこれに加えて、勾配と二乗勾配それぞれの指数移動平均（exponential moving average）として、パラメータごとに 2 つの fp32 テンソルを保持する。大規模な訓練のほぼすべてが Adam またはその派生手法を用いるため、この 16 バイトという値が実用上の基準となる。

訓練時の状態	精度	パラメータあたりのバイト数
重み	fp32	4
勾配	fp32	4
Adam の勾配の平均	fp32	4
Adam の二乗勾配の平均	fp32	4
合計		16

このパラメータあたり 16 バイトという値は、厳密な下界にすぎない。ここには、誤差逆伝播法のために順伝播中に保存され、バッチサイズや系列長（sequence length）に応じて増大する中間**活性化値**や、一時的なカーネル作業領域、アロケータの断片化は含まれていない。これらはすべて 5.7 節で GPU 上で直接計測する。この残差 MLP の場合、基準値は次のようになる。

```
n = sum(p.numel() for p in net.parameters())
weights, grads, adam_state = 4 * n, 4 * n, 8 * n   # bytes, all fp32
print(f'{n} parameters: {(weights + grads + adam_state) / 2**20:.2f} MiB '
      'for weights + gradients + Adam state')
```

```
18634 parameters: 0.28 MiB for weights + gradients + Adam state
```

このパラメータ数は手計算でも確かめられる。入力層が $20 \times 64 + 64 = 1344$ 個、各残差ブロックが $2 \times (64 \times 64 + 64) = 8320$ 個、出力層が $64 \times 10 + 10 = 650$ 個で、合計 18,634 個のパラメータとなる。Adam のオプティマイザ状態は表の上だけの理論値ではなく、テンソルごとに実際に確保されるメモリであり、モデルの完全なコピーを 2 つ分追加する。

```
adam = torch.optim.Adam(net.parameters())
net(X).sum().backward()
adam.step()
moments = sum(t.numel() for state in adam.state.values()
              for t in state.values() if t.ndim > 0)
moments == 2 * n
```

```
True
```

Adam は最初の `step()` 呼び出し時に、各パラメータの 2 つのモーメントを確保する。

この小さな MLP では、その基準値は 0.28 MiB とごくわずかであり、これまでの章でメモリをほとんど気にせずに済んだのはそのためである。しかし、まったく同じ計算を 80 億パラメータのモデルに拡大すると、バッチサイズにかかわらずたちまち 128 GB に達する。16 ビット**混合精度**（mixed-precision、5.5 節）訓練を使えば、この静的な下限を半分に減らせると期待するかもしれない。ところが意外なことに、そうはならない。混合精度では微小な更新が丸めによって消えてしまわないよう、重みのマスターコピーと Adam のモーメントを fp32 で保持しており (Micikevicius et al. 2018)、順伝播で用いる 16 ビット値はそれらからキャストして作られるからである。

5.2.2.2 メモリ確保なしのカウント

ここまでのパラメータ数カウントはすべて、すでにメモリ上に確保されたテンソルを調べていた。8B モデルではこれでは順序が逆転してしまう。32 GB の重みが RAM に収まるかを確認するために、まず 32 GB を確保しなければならないようでは本末転倒である。幸い、パラメータ数を数えるのに必要なのは形状（shape）と dtype だけであり、どちらもコンストラクタの引数のみから決まる。例として、4,096 次元の隠れ状態（hidden state）を語彙の 128,256 個の**トークン**に対するロジットへ写す Llama 3 8B の出力射影（output projection）を考えよう。その重み行列は $4096 \times 128{,}256 = 525{,}336{,}576$ 個の要素を持つ（fp32 で 2.1 GB）。以下では、ストレージを確保せずにこの層をインスタンス化し、パラメータ数を数える。

`with torch.device('meta'):` の内部でモジュールを構築すると、ストレージを確保せずに形状と dtype だけを持つ**メタ**テンソルが生成される。

```
with torch.device('meta'):
    head = nn.Linear(4096, 128256, bias=False)   # shapes, no data
n_head = sum(p.numel() for p in head.parameters())
print(head.weight.device, f'{n_head:,} parameters, '
      f'{4 * n_head / 1e9:.1f} GB in fp32')
```

```
meta 525,336,576 parameters, 2.1 GB in fp32
```

チェックポイントを読み込む前にランダム初期化で時間を浪費することなくメタデバイス上のモデルを実体化するには、`to_empty(device=...)` を呼び出してから `load_state_dict` を実行する。

```
with torch.device('meta'):
    clone = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                          ResidualBlock(64), nn.Linear(64, 10))
clone = clone.to_empty(device='cpu')   # memory, no values yet
clone.load_state_dict(net.state_dict())
torch.equal(clone(X), net(X))
```

```
True
```

8B モデルのすべての層にわたって同じ抽象的な検査を実行すれば、テンソル用のメモリを 1 バイトも確保することなく、正確な 80 億個のパラメータ数と訓練時の下限である 128 GB が得られる。

5.2.3 パラメータの共有と凍結

5.2.3.1 パラメータの結合

単一のパラメータテンソルが、ネットワーク内で複数の役割を担うことがある。典型的な例は言語モデル（8.3 節）であり、そこでは語彙 V からトークンを読み取って次のトークンを予測する。その入力**埋め込み**（embedding）は $|V| \times d$ のルックアップテーブルであり、v^{th} の行は d 次元で表されたトークン v の表現になっている。一方の出力ヘッドは、d 次元の隠れ状態を $|V|$ 個のトークンロジットへと射影するものであり、まったく同じ幾何構造と意味を持つもう 1 つの $|V| \times d$ テーブルである。両方の層を 1 つの共有重み行列に**結合**すると、$|V| \times d$ 個のパラメータが削減されるうえ、汎化性能（generalization）も向上することが多い (Press and Wolf 2017; Inan et al. 2017)。入力として互換的に振る舞うトークンは出力としても当然似たスコアを得るべきであり、重み結合によって単一の統一されたトークン空間の幾何構造が保たれるからである。メモリの節約効果も劇的になり得る。GPT-2 small (Radford et al. 2019) では、結合された 50257×768 のテーブルがモデル全体の 1 億 2400 万パラメータのうち約 3860 万個、すなわちネットワーク全体のほぼ 3 分の 1 を占めている。

Figure 5.3 にこの 2 つの呼び出し箇所を示す。語彙サイズ 100 トークン、隠れ層の幅 $d = 16$ の小規模な言語モデル `ToyLM` でその仕組みを確かめてみよう。

`self.head.weight = self.emb.weight` と代入することで、両方の層が同じ `nn.Parameter` テンソルを参照するようになる。

```
class ToyLM(nn.Module):
    def __init__(self, vocab_size, d, tied=True):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, d)
        self.body = nn.Linear(d, d)
        self.head = nn.Linear(d, vocab_size, bias=False)
        if tied:
            self.head.weight = self.emb.weight   # one tensor, two roles

    def forward(self, tokens):
        return self.head(torch.relu(self.body(self.emb(tokens))))
```

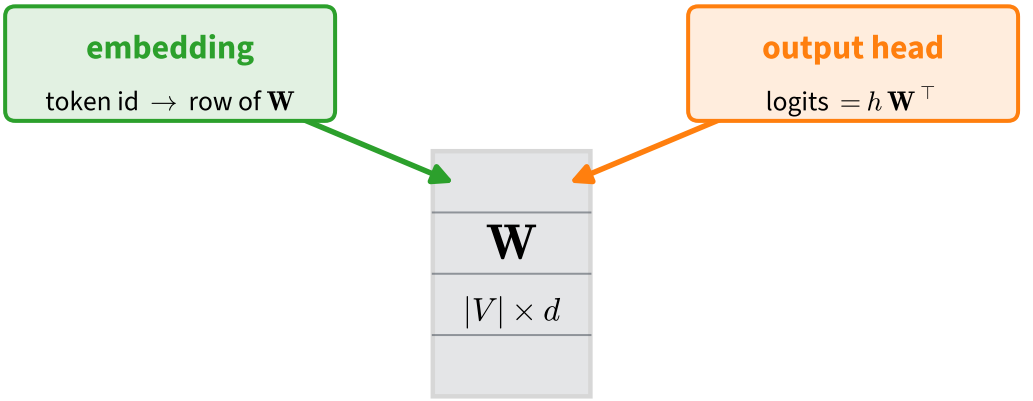


Figure 5.3: 1 つの重み行列 W と 2 つの呼び出し箇所。埋め込みはトークン ID で行を参照し、出力ヘッドはその転置を掛けてロジットを生成する。結合とは両者が同じテンソルを指すことを意味し、両方の使用箇所からの勾配は 1 か所に合算される。

```
lm = ToyLM(vocab_size=100, d=16)
lm.head.weight is lm.emb.weight
```

```
True
```

共有された行列は 1 回数えられるのだろうか、それとも 2 回数えられるのだろうか。重みを結合したモデルと結合しないモデルを比較してみよう。

```
untied = ToyLM(vocab_size=100, d=16, tied=False)
(sum(p.numel() for p in lm.parameters()),
 sum(p.numel() for p in untied.parameters()))
```

```
(1872, 3472)
```

重みを結合したモデルのパラメータ数は $1600 + 272 = 1872$ 個（埋め込みテーブル 1 つと本体の層）であるのに対し、結合しないモデルでは 2 つ目の 100×16 テーブルが加わり、合計 3472 個となる。

`collect_params()` と `state_dict()` はどちらも、共有ストレージを指す両方のキー（`emb.weight` と `head.weight`）を保持する（以下に出力する PyTorch の `named_parameters()` は、共有テンソルを 1 回だけ列挙する）。

```
sd = lm.state_dict()
(sd['emb.weight'].data_ptr() == sd['head.weight'].data_ptr(),
 [name for name, _ in lm.named_parameters()])
```

```
(True, ['emb.weight', 'body.weight', 'body.bias'])
```

1 つのテンソルが 2 回使われるとき、誤差逆伝播法では何が起きるだろうか。多変数の連鎖律（chain rule）により、各呼び出し箇所がそれぞれの勾配項をもたらし、その両方が共有テンソルの勾配に合算される。重みを結合しないモデルの 2 つのテーブルを $\mathbf{W}_{\text{emb}}$ および $\mathbf{W}_{\text{head}}$ と書き、結合行列 $\mathbf{W}$ と同じ値に初期化すると、2 つのモデルはまったく同じ関数を計算し、次の関係を満たす。

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}_{\text{emb}}} + \frac{\partial L}{\partial \mathbf{W}_{\text{head}}}. \tag{5.3}$$

結合された重みを結合しないモデルへコピーし、同じバッチに対する勾配を比較して、この恒等式を数値的に確かめよう。

```
untied.load_state_dict(lm.state_dict())   # same values, separate tensors
tokens = torch.randint(0, 100, (2, 8))
lm(tokens).sum().backward()
untied(tokens).sum().backward()
torch.allclose(lm.emb.weight.grad,
               untied.emb.weight.grad + untied.head.weight.grad)
```

```
True
```

5.2.3.2 パラメータの凍結

事前学習済みのネットワークを新しいタスクでファインチューニングする際（32.2 節（英語版））には、事前学習済みのバックボーンを固定したまま新しい出力ヘッドだけを訓練する戦略がよくとられる。これにより、必要なデータ量とオプティマイザのメモリの双方を大幅に削減できる。勾配による更新が行われないようにパラメータを固定することを**凍結**と呼ぶ。以下では、新しく用意した残差 MLP の出力ヘッド以外のすべての層を凍結し、18,634 個のパラメータのうち 650 個だけを訓練可能にする。

バックボーンのパラメータに対して `p.requires_grad = False` を設定すると、それらは勾配累積（gradient accumulation）の対象から除外される。

```
finetune = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                         ResidualBlock(64), nn.Linear(64, 10))
for p in finetune[:-1].parameters():
    p.requires_grad = False

(sum(p.numel() for p in finetune.parameters() if p.requires_grad),
 sum(p.numel() for p in finetune.parameters()))
```

```
(650, 18634)
```

この状態で Adam のステップを 1 回実行すると、出力ヘッドだけが更新され、それより前の層はすべて変化しないはずである。ステップの前後ですべてのパラメータを確認してみよう。

```
head_opt = torch.optim.Adam(
    (p for p in finetune.parameters() if p.requires_grad), lr=0.1)
before = [p.clone() for p in finetune.parameters()]
finetune(X).sum().backward()
```

```
    head_opt.step()
    [not torch.equal(b, p) for b, p in zip(before, finetune.parameters())]
```

```
[False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 True,
 True]
```

出力ヘッドの重みとバイアスだけが変化し、オプティマイザは訓練可能なその 650 個のパラメータに対してのみ Adam のモーメントを確保した。パラメータの凍結には見落としやすい落とし穴が 2 つあり、どちらもエラーを送出しない。第 1 に、パラメータを凍結しても、すでに確保されたオプティマイザ状態は解放され**ない**。すべてのパラメータを対象に構築した Adam オプティマイザですでにステップを実行していた場合、後から凍結した層についてもパラメータあたり 8 バイトの移動モーメントが保持され続ける。18,634 パラメータであれば無害だが、80 億パラメータ規模では大きな負担となる（演習問題で扱う）。

第 2 の落とし穴はさらに見落としやすい。凍結は勾配を遮断するが、層の状態の中には勾配**なしに**書き換わるものがある。特にバッチ正規化（7.3 節）は、訓練モードで動作している限り、オプティマイザを一切経由せず**順伝播**の途中で移動平均と移動分散を更新する。そのため、BatchNorm 層の訓練可能な重みを凍結したときに移動統計量も固定されるかどうかは、フレームワークによって異なる。以下では、オプティマイザのステップを一切行わずに、訓練モードのまま凍結した BatchNorm 層へ平均をずらしたバッチを通し、移動平均がドリフトするかどうかを確認する。

```
    bn = nn.BatchNorm1d(4)
    for p in bn.parameters():
        p.requires_grad = False
    before = bn.running_mean.clone()
    bn(torch.randn(8, 4) + 3)          # training mode: the statistics update
    torch.allclose(bn.running_mean, before)
```

```
False
```

移動平均が変化した。パラメータの勾配の凍結（`requires_grad=False`）と、層の評価モードへの切り替え（`bn.eval()`）は別物である。ファインチューニング中に BatchNorm 層を完全に固定するには、両方を設定する必要がある。

本節で見てきたすべてのパラメータ（結合されたもの、凍結されたもの、平均をとるもの）は、フレームワークが与えた初期値から出発していた。次節では、そうした初期値がどのように選ばれるのか、またモジュール木全体にわたって初期化（initialization）をどのように制御するのかを調べる。

5.2.4 まとめ

モデルの永続状態は名前付きテンソルの木をなしており、フレームワークの各操作からどのテンソルが見えるかは登録によって決まる。オプティマイザは**パラメータ**を更新し、チェックポイントやデバイス転送には**バッファ**（移動統計量のような勾配を持たない永続テンソル）も含まれる一方、未登録の属性は黙ってスキップされる。fp32 と Adam による訓練では、活性化値を考慮する前の段階でパラメータあたり 16 バイト（重み、勾配、2 つの Adam モーメントであり、8B モデルでは 128 GB）の基準メモリが必要となるが、抽象的な形状検査を用いればメモリを確保せずにその予算を計算できる。重み結合は複数の役割にわたって 1 つのテンソルを共有させることで、カウントを 1 回に抑えつつ全呼び出し箇所からの勾配の和を受け取る。また、凍結は勾配による更新を止めるが、既存のオプティマイザ状態を解放するわけではなく、順伝播における BatchNorm の移動統計量の更新も止めるとは限らない。

5.2.5 演習問題

1. [code] **移動統計量**。この問題で必要なのは、7.3 節のバッチ正規化層に関する次の更新式だけである。訓練モードでは、この層はミニバッチ平均 $\hat{\mu}_{\mathcal{B}}$ と分散を用いて正規化し、各訓練ステップで移動平均 $\bar{\mu}$ を次式により更新する。

$$\bar{\mu} \leftarrow \rho\,\bar{\mu} + (1-\rho)\,\hat{\mu}_{\mathcal{B}}. \tag{5.4}$$

   1. 新しい層は $\bar{\mu} = \mathbf{0}$ から始まる。すべての特徴量でミニバッチ平均が 3 となるバッチを訓練モードで処理したとき、$\bar{\mu}$ が k バッチ後にどの値になるか予測せよ。オプティマイザのステップを一切行わずに、4 個の特徴量に対する新しい層で $k = 1, 5, 20$ の場合の予測を確かめ、この層がデフォルトで用いる ρ の値を読み取れ。
   2. $\bar{\mu}$ を訓練可能なパラメータとして登録し、上記の更新を取り除いたとする。訓練損失は $\bar{\mu}$ にどのような勾配を送るか。また、そのとき訓練損失と評価時の正解率は、変更を加えていないモデルと比べてどうなるか。
   3. $\bar{\mu}$ のみに依存する損失であって、各ミニバッチに対する学習率 η の勾配ステップが上記の取り除いた更新を再現するものを求め、一致させる η を ρ を用いて答えよ。

2. [code] **メモリ使用量**。すべてのパラメータを訓練する fp32 の Adam 訓練において、`net` の各トップレベルブロックについて状態要素数とバイト数を報告する補助関数を書け。

   1. どのブロックが支配的か。出力層の幅がいくつになると、1 つの残差ブロックを上回るか。
   2. 2 つの残差ブロックの間に、64 個の隠れ特徴量に対するバッチ正規化層を挿入せよ。この層の移動統計量のコストを正しく見積もるように補助関数を拡張し、各ブロックの 1 要素あたりのバイト数を報告せよ。全バイト数に占める割合が全要素数に占める割合より大きくなるのはどのブロックで、小さくなるのはどのブロックか。両方の理由を説明せよ。

3. [code] **結合と凍結**。本節の重み結合された `ToyLM` を考える。

   1. `lm.emb.weight` を凍結し、`lm.head` はそのままにせよ。訓練可能なパラメータはいくつ残るか。また、ここから結合と凍結の相互作用について何がわかるか。
   2. `lm` の値から出発し、埋め込みは固定したままヘッドだけが適応するモデルを構築し、オプティマイザの 1 ステップ後に両方の性質が成り立つことを確かめよ。全体を訓練可能にした `lm` と比べて、格納するパラメータ数と訓練するパラメータ数はそれぞれいくつか。

4. **ファインチューニングのメモリ**。ある言語モデルが 80 億個のパラメータを持つとする。出力層は埋め込みとは結合されていない $4096 \times 128{,}256$ の行列であり、状態はすべて fp32、

オプティマイザは Adam である。

1. すべてのパラメータを訓練するとき、重み、勾配、および Adam の移動平均は何バイトを占めるか。その状態だけで 80 GB GPU 何枚分を埋めるか。
2. 出力層以外をすべて凍結し、出力層のみを対象にオプティマイザを構築せよ。このとき永続状態は何バイトを占め、1 枚の 80 GB GPU に収まるか。
3. 代わりに、全パラメータを対象に構築したオプティマイザで凍結前に 1 ステップを実行し、その後もそのまま使い続けたとする。確保されたまま残るのは何バイトで、その状態はなお収まるか。

5.3 初期化

これまで訓練してきたすべてのニューラルネットワークは、通常フレームワークのデフォルトのイニシャライザによって抽出されたランダムな重みから出発していた。しかし 4.4 節で見たように、この出発点はきわめてシビアになり得る。各層が活性化値をわずかに増幅または減衰させるだけでも、層を深く積み重ねるとその倍率が指数関数的に積み重なり、訓練が始まる前に立ち往生してしまうからである。

同節では、単一の層に対する解決策も導出した。すなわち、標準偏差 $g/\sqrt{n_{\text{in}}}$ で重みを抽出する方法である。ここで、n_{in} は層の入力幅であり、**ゲイン** g は直後に続く活性化関数を補正する係数で、ReLU の前では $g = \sqrt{2}$（He 初期化、4.4.2.2 節）、線形出力では $g = 1$ とする。この規則のもとでは、各層は入力の二乗平均を保つ。

この規則をすべての層に適用すれば、初期化の問題は完全に解決すると思うかもしれない。ところが驚くべきことに、そうはならない。残差ネットワーク（residual network）では、個々の層がすべて He 初期化に従っていても、深さとともに活性化値が爆発してしまうのである。各残差ブロックは、その本体 f（5.1 節）の出力を入力に加算する。本体が入力の二乗平均を保ち、かつ入力と無相関な出力を生成するなら、両者の和の二乗平均は **2 倍**になる。32 個の残差ブロックを経ると、勾配ステップを 1 回も進める前に、活性化値の標準偏差は $2^{16} \approx 65{,}000$ 倍に膨れ上がる。

したがって、適切な初期化は個々の層の種類だけでなく**アーキテクチャの配線**にも依存するため、モジュールツリー全体にカスタム規則を適用するための見通しのよい仕組みが必要になる。本節の前半では、モジュールツリー全体へのイニシャライザの適用、切断正規分布による極端な抽出値の抑制、カスタムイニシャライザの作成、重みのインプレース編集といった具体的な手法を扱う。後半ではじっくりと、残差ブロックをまたぐ分散の増大を解析し、各ブロックの出力射影（output projection）を縮小またはゼロ化することでこれを解決する。

5.3.1 イニシャライザの適用

4.4 節で見たように、線形層のデフォルトのイニシャライザは He 初期化（4.4.2.2 節）を適用しない。モデル全体に He 初期化（あるいは後ほど導出する深さ依存の残差スケーリング）を適用するには、モジュールツリーを走査する（5.1 節）。以下では、幅 256 の 10 層 ReLU スタック（バイアスなし）に標準正規分布から抽出した 1,024 個の入力を与え、デフォルトの初期化と He 初期化を比較する。

PyTorch のデフォルトは $U(-1/\sqrt{n_{\text{in}}}, 1/\sqrt{n_{\text{in}}})$（He の分散の 6 分の 1）から抽出する。`net.apply(init_he)` はすべてのサブモジュールに対して `init_he` を呼び出し、インプレースの `nn.init.kaiming_normal_`（末尾にアンダースコアが付く）を用いることで既存の `nn.Parameter` への参照をそのまま保つ。

```
net = nn.Sequential(*[module for _ in range(10) for module in
                      (nn.Linear(256, 256, bias=False), nn.ReLU())])
X = torch.randn(1024, 256)
print(f'default: mean square of the output {net(X).pow(2).mean():.2g}')

def init_he(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')

net.apply(init_he)
print(f'He:      mean square of the output {net(X).pow(2).mean():.2g}')
```

```
default: mean square of the output 1.9e-08
He:      mean square of the output 1.1
```

デフォルトのイニシャライザのもとでは、すべての層が分散に 1 未満の係数を掛けるため、出力の二乗平均は何桁も潰れてしまう。He 初期化に切り替えると、出力の二乗平均は 1（入力の二乗平均）から約 2 倍以内の範囲へと一気に戻る。残りのばらつきは、有限幅に起因するランダムな揺らぎが 10 層にわたって積み重なったものにすぎない。

このパターンがインターフェースのすべてである。後の章では畳み込みネットワーク用の `init_cnn` のように補助関数としてまとめるが、いずれも `init_he` のような関数を `apply` に渡しているだけである。

5.3.1.1 切断正規分布

標準ガウス分布は分散こそ適切になるものの、その裾は無限遠まで広がっている。小さな層ではめったに問題にならないが、大規模モデルが持つ 10^8 個の重み全体で見ると、標準偏差の 5 倍を超える要素が数十個はほぼ確実に生じる。極端な初期の外れ値を避けるため、BERT や Vision Transformer などの現代的な言語モデルや視覚モデルは、σ の定数倍の外側にあるサンプルを棄却して引き直す**切断正規分布**（truncated normal）から抽出する (Devlin et al. 2018; Dosovitskiy et al. 2021)。裾を切り落とすと分散もわずかに小さくなるが、フレームワークの実装によっては再スケーリングでこれを補正するものもあれば、そのままにするものもある。

`nn.init.trunc_normal_` は**絶対値での**切断位置 `a` と `b`（デフォルトは ± 2）を受け取るため、$\pm 2\sigma$ で切断するには `std=0.02` の場合 `a=-0.04, b=0.04` を渡す必要がある。

```
w = torch.empty(1000, 1000)
nn.init.normal_(w, std=0.02)
print(f'normal:    std {w.std():.4f}, max weight {w.abs().max():.4f}')
nn.init.trunc_normal_(w, std=0.02, a=-0.04, b=0.04)
print(f'truncated: std {w.std():.4f}, max weight {w.abs().max():.4f}')
```

```
normal:    std 0.0200, max weight 0.0998
truncated: std 0.0176, max weight 0.0400
```

$\sigma = 0.02$ の通常の正規分布から 100 万個を抽出すると、極端な重みは $5\sigma = 0.1$ に迫るが、切断正規分布では $|w| \leq 0.04$ が保証される。裾を切り落とすことで、経験標準偏差は 0.02 から約 0.0176 に低下する。BERT のように固定した σ を用いるレシピではこのわずかな減少をそのまま受け入れる一方、分散保持（variance-preserving）型の方式では切断後に再スケーリングして目標の分散に正確に合わせる。

5.3.2 カスタムイニシャライザと直接編集

組み込みのイニシャライザが論文やアーキテクチャの要件に合わない場合は、独自のイニシャライザを作成するか、パラメータテンソルを直接編集できる。いずれも各フレームワークのイニシャライザのインターフェースに基づいている。

PyTorch では、イニシャライザはパラメータをインプレースで上書きする関数である。

5.3.2.1 任意のゲインを持つ規則

それでは、冒頭で述べたゲイン付きの正規分布規則を実装しよう。すなわち、重みを標準偏差 $g/\sqrt{n_{\text{in}}}$（g は任意のゲイン）で抽出する規則である。He 初期化は $g=\sqrt{2}$ に対応し、以下の残差ネットワークの実験では各ブロックの出力射影に対して $g=1$、$g=1/\sqrt{N}$（ブロック数 N の場合）、および $g=0$ も試す。再利用可能なイニシャライザを 1 つ書いておけば、4 つのケースすべてをすっきりと扱える。

この関数は `torch.nn.init` の慣例に従い、渡されたテンソルを上書きし、関数名の末尾にアンダースコアを付ける。PyTorch の重みの形状は $(n_{\text{out}}, n_{\text{in}})$ である。

```
def scaled_normal_(weight, gain):
    nn.init.normal_(weight, std=gain / math.sqrt(weight.shape[1]))
```

5.3.2.2 直接編集

初期化の直後や重みを読み込んだ後に、パラメータの値を直接変更したい場合がある。ReLU は正斉次（positively homogeneous）であるため、先ほどの 10 層スタックで最終層の重みを半分にすると、すべての出力の活性化値も半分になり、したがって出力の二乗平均は 4 分の 1 に減少するはずである。

リーフである `nn.Parameter` テンソルへのインプレース編集は、自動微分に拒否されないよう `with torch.no_grad():` で囲む。

```
before = net(X).pow(2).mean()
with torch.no_grad():
    net[-2].weight *= 0.5
print(f'{before:.2g} before, {net(X).pow(2).mean():.2g} after')
```

```
1.1 before, 0.28 after
```

予想どおり、出力の二乗平均は元の値のちょうど 4 分の 1 に下がる。

5.3.3 残差ネットワークの初期化

5.3.3.1 ブロックをまたぐ分散の増大

先ほど、個々の層がすべて He の規則に従っていても、深い残差スタックは初期化時点で爆発すると述べた。これを示すため、残差ブロックが $\mathbf{x}_{k+1}=\mathbf{x}_k+f_k(\mathbf{x}_k)$ を計算することに着目しよう。ここでは 5.1 節の 2 層の本体を取り上げ、多くの現代的な残差ブロックや Transformer と同様に中間で幅を 4 倍に拡大する構成を考える。すなわち、幅 d から $4d$ への線形層、ReLU、そして $4d$ から d へ戻す線形の**出力射影**である。バイアスはすべて 0 で初期化する。最初の層は直後に ReLU が続くため He 初期化（$g=\sqrt{2}$）を用い、出力射影は標準偏差 $g/\sqrt{4d}$ を用いる。ここ

でゲイン g は自由パラメータとして残しておく（$g = 1$ が 4.4 節で導出した線形層の標準規則である）。

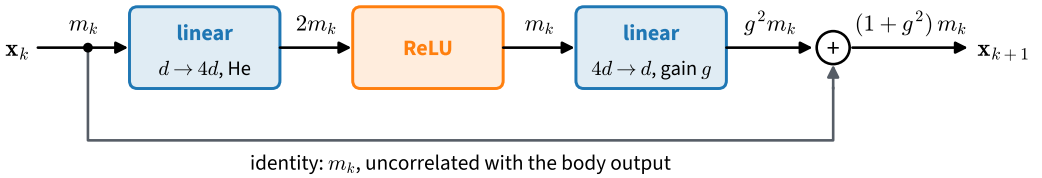


Figure 5.4: 1 つの残差ブロックの各時点における信号の二乗平均。He 層が m_k を 2 倍にし、ReLU が再び半分にし、出力射影が g^2 倍にする。恒等経路は本体の外側で m_k をそのまま伝え、加算点で 2 つの二乗平均が足し合わされる。

m_k を $\mathbf{x}_k$ の各要素の二乗平均としよう。Figure 5.4 は、ブロック内でのその変化を追ったものである。最初の線形層を通過すると m_k は 2 倍になり、ReLU がそれを半分にして m_k に戻す（4.4.2.2 節）。続いて出力射影が二乗平均を $4d \cdot g^2/(4d) = g^2$ 倍にするため、本体の出力 $f_k(\mathbf{x}_k)$ の二乗平均は $g^2 m_k$ となる。ここが重要なポイントである。出力射影の重みは平均 0 で $\mathbf{x}_k$ と独立であるため、本体の出力 $f_k(\mathbf{x}_k)$ はスキップ信号 $\mathbf{x}_k$ と無相関になる。したがって、両者の二乗平均は次のように足し合わされる。

$$m_{k+1} = m_k + g^2 m_k = (1 + g^2)\, m_k. \tag{5.5}$$

どのブロックも、活性化値の二乗平均を $1 + g^2 > 1$ 倍（$g \neq 0$ である限り）に膨張させる。$g = 1$ では各ブロックが分散を 2 倍にするため、N ブロックを経ると活性化値の標準偏差は $2^{N/2}$ 倍に増大する。すなわち、2 ブロック後には 2×、8 ブロック後には 16×、32 ブロック後には 65,536× となる。個々の層はすべて単一層の分散規則を守っていたにもかかわらず、ネットワーク全体が爆発してしまった。その局所的な規則は、ブロックごとにスキップ接続が入力を足し戻すことを考慮していないからである。

5.3.3.2 出力射影の縮小

N ブロック全体では、ブロックあたりの倍率が積み重なって $(1 + g^2)^N$ となる。したがって、任意の深さで活性化値のスケールを適切に保つには、g を N の増加に伴って縮小する必要がある。$g^2 = 1/N$ と選び、すべての本体の出力射影を $1/\sqrt{N}$ 倍にスケーリングすると、この増幅は次のようにきれいに抑えられる。

$$m_N = \left(1 + \frac{1}{N}\right)^N m_0 \leq e\, m_0, \tag{5.6}$$

これは $(1 + 1/N)^N$ が e に向かって単調増加するためである。これにより、活性化値の標準偏差の増大は深さによらず高々 $\sqrt{e} \approx 1.65$ 倍に抑えられる（2 ブロック後で 1.5、8 ブロック後で 1.60、32 ブロック後で 1.64）。同じ $1/\sqrt{N}$ スケーリングは実用上も広く使われている。たとえば GPT-2 は、すべての残差**ブランチ**の出力射影における初期標準偏差を $1/\sqrt{N}$ 倍（N はネットワーク内の残差ブランチの総数）に縮小する (Radford et al. 2019)。GPT-2 の各ブロックは入力を正規化するため、増大は乗法的ではなく加法的になるが（演習問題を参照）、同じスケーリングによって和が有界に保たれる。Fixup は、正規化層を一切持たない残差ネットワークに対して、この種の深さ依存スケーリングを導出している (Hongyi Zhang et al. 2019)。

5.3.3.3 ゼロ出力射影

この考えを極限まで推し進め、$g = 0$ としたらどうなるだろうか。一見すると、ゼロ初期化は危険に思える。**すべての**重みをゼロにすると、全隠れユニットが永久に同一の対称性に閉じ込められてしまうからである（4.4 節）。しかし、各残差ブロックの本体の出力射影**のみ**をゼロにする場合はまったく異なる。ステップ 0 ではすべての本体が 0 を出力するため、各ブロックは厳密な恒等写像として振る舞い、ネットワーク全体が安定した状態から出発して、訓練中に深さが徐々に働き始める。本体の最初の層はランダムな重みを保持しているため、対称性によって学習が凍りつくことは決してない。最初の逆伝播において、出力射影は（その ReLU 入力が非ゼロであるため）非ゼロの勾配を受け取って即座に 0 から離れ、2 ステップ目以降は最初の層にも勾配が自由に流れるようになる。Figure 5.5 に、この 3 つの段階と全重みゼロ化の罠との対比を示す。

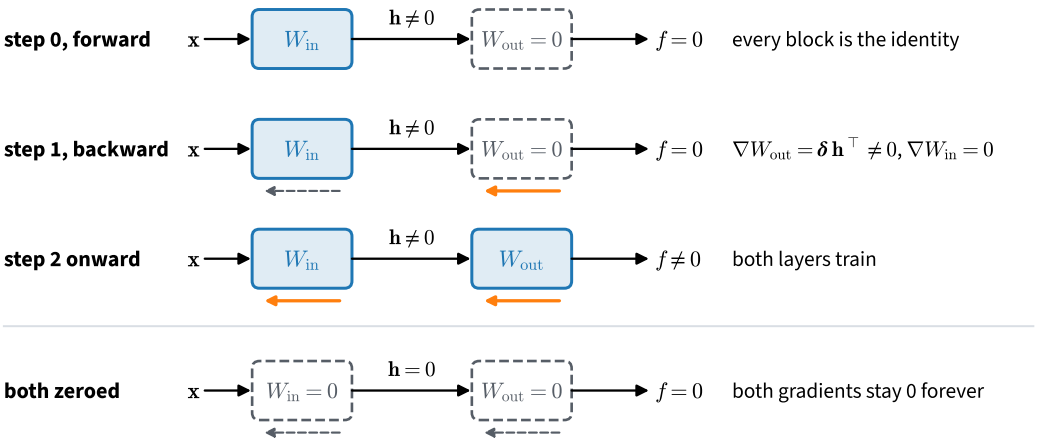


Figure 5.5: 残差ブロックの本体の出力射影 W_{out} のみをゼロ化した場合。最初の順伝播では本体の出力が $f = 0$ となるためブロックは恒等写像となるが、隠れ層の活性化値 $\mathbf{h}$ は非ゼロである。したがって最初の逆伝播では、W_{out} は非ゼロの勾配 $\delta\mathbf{h}^{\top}$（δ は本体の出力に到達する勾配）を受け取る一方、W_{in} は勾配を受け取らない。次のステップからは両方の層が訓練される。下段：両方の層をゼロ化すると $\mathbf{h} = 0$ となり、どちらの層も永久に勾配を受け取らない。

ゼロ初期化したブランチには明確な系譜がある。Goyal et al. (2017) は、大きなミニバッチで ResNet を訓練するために各残差ブロックの最終スケール係数を 0 に初期化した。Fixup はゼロ出力射影を深さ依存のスケーリングと組み合わせている。その後の研究では、ゼロ行列の代わりに 0 から始まるブランチあたり 1 つの学習可能なスカラーを用いている (De and Smith 2020; Bachlechner et al. 2021)。

5.3.3.4 深さにわたる分散の計測

それでは、3 つの予測すべてを実験で検証しよう。`WideResidualBlock` は、先ほど作成したゲイン付き正規イニシャライザを用いて $d \to 4d \to d$ の残差ブロックを実装したものであり、拡大層にはゲイン $\sqrt{2}$ を、出力射影には設定可能な `out_gain` を用いる。

```
class WideResidualBlock(nn.Module):
    def __init__(self, d, out_gain):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(d, 4 * d), nn.ReLU(),
```

```
                                nn.Linear(4 * d, d))
        for layer, gain in ((self.body[0], math.sqrt(2)),
                            (self.body[2], out_gain)):
            scaled_normal_(layer.weight, gain)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)
```

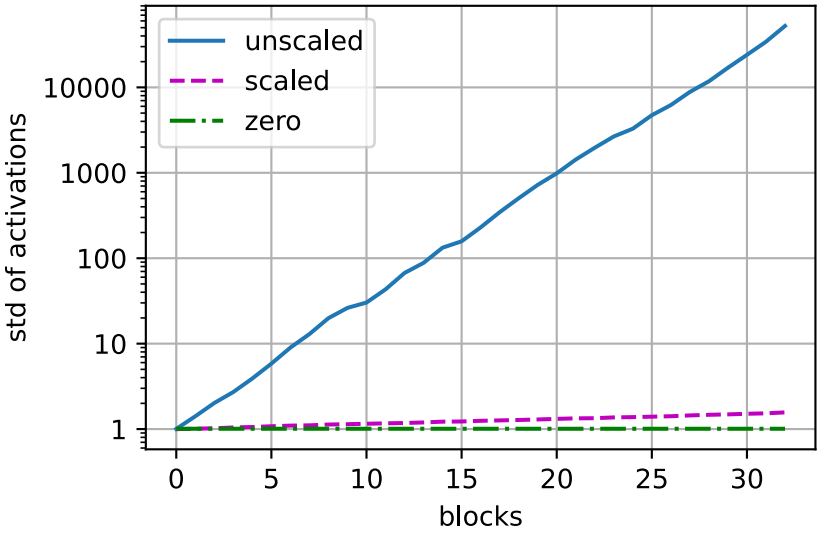
次に、std_across_depth は幅 64 のブロックを num_blocks 個積み重ねたスタックを構築し、256 個の $\mathcal{N}(0,1)$ 入力を与えて、入力時点および各ブロック通過後の活性化値の標準偏差を記録する。

```
def std_across_depth(num_blocks, out_gain):
    net = nn.Sequential(*[WideResidualBlock(64, out_gain)
                          for _ in range(num_blocks)])
    X = torch.randn(256, 64)
    stds = [X.std().item()]
    with torch.no_grad():
        for block in net:
            X = block(X)
            stds.append(X.std().item())
    return stds
```

スケーリングなし（$g = 1$）、スケーリングあり（$g = 1/\sqrt{N}$）、ゼロ（$g = 0$）の 3 つのゲインスケジュールすべてを比較し、$N = 2, 8, 32$ に対する最終的な標準偏差を表示するとともに、$N = 32$ における深さ全体のプロファイルをプロットする。

```
gains = {'unscaled': lambda n: 1.0, 'scaled': lambda n: n ** -0.5,
         'zero': lambda n: 0.0}
profiles = {n: [std_across_depth(n, gain(n)) for gain in gains.values()]
            for n in (2, 8, 32)}
print(f'{"N":>3}' + ''.join(f'{name:>10}' for name in gains))
for n, stds in profiles.items():
    print(f'{n:>3}' + ''.join(f'{s[-1]:>10.3g}' for s in stds))
d2l.plot(list(range(33)), profiles[32], 'blocks', 'std of activations',
         legend=list(gains), yscale='log', figsize=(4.5, 3))
```

```
  N  unscaled    scaled      zero
  2      2.04      1.49     0.996
  8      13.8      1.55     0.988
 32  5.24e+04      1.57      1.01
```



実測された曲線はこれまでの解析を裏づけている。スケーリングなしのスタックはブロックごとに約 $\sqrt{2}$ 倍で増大し（対数スケール上で直線となり）、32 ブロック後には約 20,000〜200,000 に達する（ランダムな正の係数を掛け合わせると右に裾を引く分布になり、その中央値が平均の平方根を下回るため、個々の試行の多くは理論値 $\sqrt{\mathbb{E}[m_{32}]} = 65{,}536$ よりわずかに下に位置する）。初期の活性化値がすでに数万に達している状態では、スケーリングなしのスタックの訓練を試みる意味はほとんどない。対照的に、$1/\sqrt{N}$ でスケーリングしたスタックはどの深さでも 1.4〜1.9 の間に余裕をもって収まっており、1.5〜1.65 という予測と一致する。一方、出力射影をゼロ化した場合は 32 ブロックすべてにわたって標準偏差が 1 のまま一定に保たれる。

このような簡単な順伝播による診断を実行すれば、訓練に GPU 時間を浪費する前に、破綻した初期化を即座に見つけ出せる。ただし、いざ訓練が始まった後に $1/\sqrt{N}$ スケーリングとゼロ初期化のどちらがより速く最適化されるかは、この診断からはわからないことに注意しよう。どちらもステップ 0 で信号のスケールを保っており、違いは残差ブランチが最初から機能しているか、最初の数回の更新で立ち上がるかという点だけである。

最後に、初期化が制御するのはステップ 0 だけであることを忘れてはならない。勾配ステップによって重みが更新されるにつれ、初期化だけでは活性化値が適切なスケールに保たれる保証はなくなる。そこで出番となるのが正規化層（7.3 節）である。

5.3.4 まとめ

フレームワークのデフォルトのイニシャライザは He 初期化を用いないため、正規化層を持たない深いネットワークでは（コンストラクタ引数、ツリー全体へのコールバック、あるいは直接のインプレース編集による）明示的な初期化が必要であり、さらに大規模モデルでは切断正規分布によってまれに生じる極端な外れ値を防ぐ。とりわけ重要な点として、層ごとの分散規則（4.4 節）だけでは残差アーキテクチャには不十分である。各ブロックの本体を入力に足し戻すことで活性化値の二乗平均が $1 + g^2$ 倍になるため、$g = 1$ ではブロックごとに $\sqrt{2}$ 倍ずつ爆発してしまう。各ブロックの出力射影を $1/\sqrt{N}$ 倍にスケーリングすれば活性化値の増大は $\sqrt{e}$ 倍以下（任意の深さ N に対して）に抑えられ、一方で出力射影をゼロ化すればネットワークを対称性に閉じ込めることなくすべてのブロックを恒等写像として開始できる。

5.3.5 演習問題

1. **切断正規分布**。$\pm 2\sigma$ で切断された正規分布について、切断によって棄却される抽出値の割合はいくらか、また標準偏差はどれだけ縮小するか。いずれの量も初等的な閉じた形を持たない。両者を標準正規分布の密度関数と分布関数を用いて表し、数表の値を使って数値的に評価せよ。さらに、標準偏差を 5.3.1.1 節の実演における出力結果と照合し、通常の正規分布からの抽出値のうち $\pm 2\sigma$ を超える要素を数えることで棄却割合を検証せよ。

2. [code] **正規化された入力**。多くの残差ネットワークは各本体の入力を正規化する。すなわち、本体は入力として $\mathbf{x}_k / r_k$（r_k は各データ例における $\mathbf{x}_k$ の要素の二乗平均平方根）を $\mathbf{x}_k$ の代わりに受け取る。

    1. このとき本体の出力の二乗平均が g^2（$\mathbf{x}_k$ のスケールによらず）になること、および $m_0 = 1$ のとき N ブロック後の二乗平均が $1 + Ng^2$ になることを示せ。深さに伴う増大は、本節の正規化なしブロックと比べてどう異なるか。
    2. スタック末端での二乗平均を 2（すべての N に対して）に保つゲインはいくらか。これを GPT-2 のスケーリングと比較せよ。
    3. `WideResidualBlock` にこの正規化を追加し、$g = 1$ と $g = 1/\sqrt{N}$ について $N = 32$ における深さにわたる標準偏差を計測し、自身の予測および 5.3.3.4 節の実測プロットと比較せよ。

3. [code] **Kaiming 初期化**。4.4.2.2 節の Kaiming（He）イニシャライザ（一様分布版と正規分布版）を、明示的な `nonlinearity` 引数（ゲインを `'relu'` と `'linear'` の間で切り替える）付きで実装せよ。

    1. 入力数 512、出力数 256 の層の重みに対して両者を適用し、平均、標準偏差、値域を確認せよ。すなわち、正規イニシャライザについては本節のゲイン付き正規イニシャライザと比較し、一様イニシャライザについては理論上の境界および標準偏差と比較せよ。値域に違いがあればその理由を説明せよ。
    2. 実装した正規イニシャライザを 5.3.3.4 節のスタックに適用し、すべてのブロックの両方の層に `'relu'` のゲインを設定せよ。各ブロックが二乗平均を何倍にするかを予測し、5.3.3.4 節のプロットに実測曲線を追加して検証せよ。

    CMU 10-714 の Homework 2 および Homework 4 に基づく。

4. [code] **深さにわたる勾配**。●ブロックを通過する逆伝播では、勾配に $\mathbf{x} \mapsto \mathbf{x} + f(\mathbf{x})$ の転置ヤコビ行列が掛けられる。

    1. 本節の議論を逆伝播に対して繰り返せ。逆伝播では各層の入力ではなく出力について和をとるにもかかわらず、本体が勾配の二乗平均を g^2 倍にし、その結果として各ブロックがやはり二乗平均を $1 + g^2$ 倍にすることを示せ。
    2. $N = 32$ における 3 種類のゲインの選択についてこの予測を検証せよ。すなわち、スタックの出力の総和の、その入力に関する勾配を計算し、その二乗平均を出力時点での勾配の二乗平均と比較せよ。

5. [code] **データ駆動の較正**。●5.3.3.4 節のスタックをスケーリングなしのゲインで初期化したうえで、単位分散の入力からなる 1 つのミニバッチを用いてブロックごとに較正せよ。各ブロックについて順番に、そのブロックまでミニバッチを順伝播させ、本体の出力 $f_k(\mathbf{x}_k)$ の標準偏差を計測し、その標準偏差が $1/\sqrt{N}$ になるようブロックの出力射影を再スケーリングする。これは Mishkin and Matas (2016) による層逐次の単位分散初期化を残差スタックに適合させたものである。

    1. 1 回の順伝播から全ブロックを一度に較正するのではなく、ブロックを順番に較正しなければならない理由を説明せよ。
    2. 較正の対象をブロック出力 $\mathbf{x}_k + f_k(\mathbf{x}_k)$ ではなく本体の出力とする理由を説明せよ。

3. 較正後のスタック末端における標準偏差を予測せよ。較正後の深さ対標準偏差の曲線をスケーリングありのゲインの曲線と並べてプロットし、両者の違いを説明せよ。

5.4 カスタム層とカスタム関数

深層学習フレームワークの組み込みライブラリには、線形射影、活性化関数、ドロップアウトなど、これまで誰もが定番として必要としてきた標準的な層が揃っている。しかし研究はリリースサイクルよりも速く進むため、新たに提案された層が公式ライブラリに加わるまでには何年もかかることがある。各活性化値ベクトルの二乗平均平方根（root mean square、RMS）の大きさを単位長に再スケーリングする正規化層（normalization layer）である **RMSNorm** は、まさにその好例である。2019 年に発表されてから (Zhang and Sennrich 2019)、標準の層ライブラリに採用されるまでに数年を要した。それまでの間、RMSNorm を使いたいコードベースはすべて、自前で実装するかサードパーティ製の実装をインポートするしかなかった。自作するカスタム層が一流の組み込み層と同じように振る舞うには、フレームワークにシームレスに統合されなければならない。すなわち、パラメータがオプティマイザに渡り、状態がチェックポイントへシリアライズされ GPU 上へ移動でき、さらに組み込み層と並べてシーケンシャルコンテナやカスタムコンテナに組み込めなければならない。

ときには、順方向の演算そのものではなく、その導関数が障壁になることもある。80 億パラメータのモデル（5.1 節）を限られたメモリに収めるため、実務では重みを 4 ビット整数へ量子化（quantization）し、ストレージ容量をわずか 4 GB（16 ビット浮動小数点の 4 分の 1）に削減することが多い。訓練済みのモデルを量子化するだけであれば勾配は不要である。しかし、順伝播で量子化された重みを用いて**訓練**するには勾配が必要であり、そこでは丸め処理が立ちはだかる。丸め関数の導関数はほとんど至るところで 0 であるため、順伝播の内部で量子化された重みはすべて受け取る勾配が 0 になり、学習が止まってしまうからである。

本節では、この両方の種類の拡張を構築する。フレームワークのあらゆる操作（最適化、チェックポイント保存、デバイス間の移動）はモジュール木に登録された状態をたどるため（5.1 節）、基底モジュールをサブクラス化してテンソルを登録するだけで、カスタム層はこれらの仕組みに自動的に組み込まれる。以下では 4 つの具体例を順に進めていく。状態をもたない層、学習されるパラメータをもつ `RMSNorm`、訓練されない固定バッファを保持するランダム特徴量の層、そして丸め処理に勾配を通すカスタム逆伝播としての**ストレートスルー推定器**（straight-through estimator）である。

5.4.1 パラメータをもたない層

まずは最も単純なカスタム層、すなわち状態を一切保持しない層から始めよう。`CenteredLayer` は入力テンソルから全体の平均を引く層である。これを構築するには、フレームワークの基底モジュールをサブクラス化し、順伝播だけを実装すればよい。

```
class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

ベクトル `[1, 2, 3, 4, 5]` の平均は 3 であるため、`CenteredLayer` はこれを -2 から 2 までの範囲にシフトする。

```
layer = CenteredLayer()
layer(torch.tensor([1.0, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

CenteredLayer は組み込み層とまったく同じモジュールインターフェースを共有しているため、シーケンシャルコンテナ内で線形層の直後にそのまま配置して出力を中心化できる。結果の平均は 0 になるはずだが、浮動小数点演算において 10^{-8} 程度以下のごく小さな残差が見られるのは通常の丸め誤差であり、バグではない。

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(2.5611e-09, grad_fn=<MeanBackward0>)
```

5.4.2 パラメータをもつ層

4.4 節では、層の重みに応じて活性化値のスケールが深さにわたってドリフトしうることを見た。**正規化層**は、各活性化値ベクトルを制御された大きさに再スケーリングし、代わりにそれを調整するための学習されるスケールパラメータをネットワークに与えることで、この感度を解消する。この層の族については 7.3 節以降で詳しく調べるが、ここではその最も簡潔な一例である **RMSNorm** を実装する。

5.4.2.1 RMSNorm

入力ベクトル $\mathbf{x} \in \mathbb{R}^d$ の二乗平均平方根は $\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$ で与えられる。RMSNorm は $\mathbf{x}$ をこの RMS で割り、学習される**ゲイン**ベクトル $\mathbf{g} \in \mathbb{R}^d$ を要素ごとに掛けてスケーリングする。

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \odot \mathbf{g}.$$

小さな定数 ϵ（たとえば 10^{-6}）はゼロ除算を防ぐためのものである。$\mathbf{g}$ を $\mathbf{1}$ に初期化すると、入力のスケールにかかわらず、出力の二乗平均平方根は常に 1 に近くなる。たとえば $\mathbf{x} = (3, 4)$ の場合、二乗平均は $(9+16)/2 = 12.5$、RMS は 3.536 となって出力は $(0.849, 1.131)$ になり、入力を $(300, 400)$ に再スケーリングしてもまったく同じ出力が得られる。幾何学的には（Figure 5.6）、RMSNorm は $\mathbf{x}$ の向きを保ったまま半径 $\sqrt{d}$ の球面上に射影し、その後 $\mathbf{g}$ が各座標軸を引き伸ばす。

RMSNorm は**層正規化**（layer normalization）を簡素化したものである (Ba et al. 2016)。層正規化では再スケーリングに加えて、各例の平均を引き（行ごとの CenteredLayer に相当）、再スケーリング後に学習されるバイアスを加える。Zhang と Sennrich は、平均の中心化を省いても性能が損なわれることはめったになく計算量を削減できることを示した。そのため、Llama 3 8B (Grattafiori et al. 2024)（5.1 節）をはじめとする多くの現代的な言語モデルがモデル全体で RMSNorm を採用している。

$\mathbf{g}$ は訓練する必要があるため、層はこれをパラメータとして登録する（5.2 節）。実装の残りの部分は定義式をそのまま反映している。

PyTorch では、nn.Parameter でラップしたテンソルを属性に代入することで登録される。

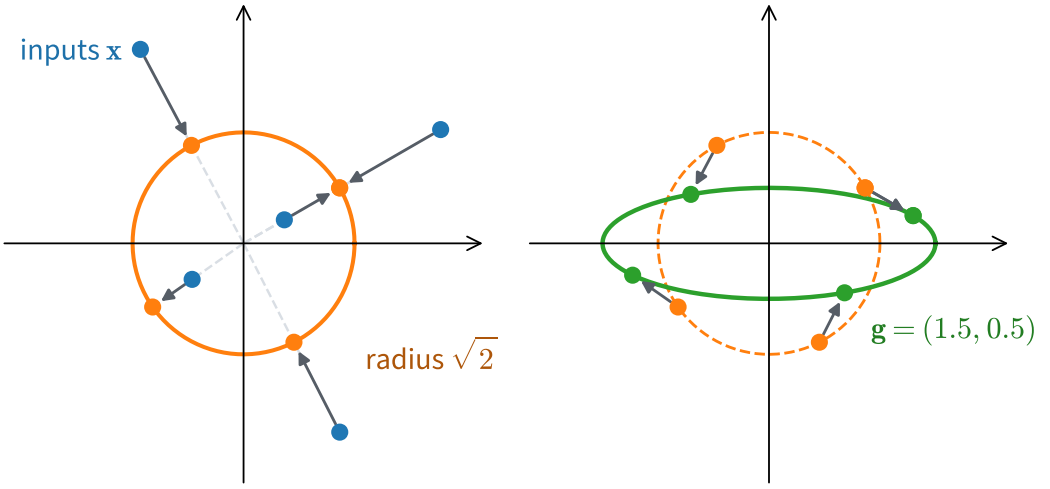


Figure 5.6: 2 次元における RMSNorm。左：ゲインが 1 のとき、すべての入力（青）は原点からの半直線に沿って半径 $\sqrt{2}$ の円周（オレンジ）上へ移り、同じ半直線上にある長さ 0.6 と 2.9 の 2 つの入力は同じ点に着地する。右：ゲイン $\mathbf{g} = (1.5, 0.5)$ は、この円を半軸 $1.5\sqrt{2}$ と $0.5\sqrt{2}$ の楕円へと引き伸ばす。

```
class RMSNorm(nn.Module):
    def __init__(self, d, eps=1e-6):
        super().__init__()
        self.gain = nn.Parameter(torch.ones(d))
        self.eps = eps

    def forward(self, X):
        rms = X.pow(2).mean(-1, keepdim=True).add(self.eps).sqrt()
        return self.gain * X / rms
```

RMSNorm を $(3, 4)$ と、それを 100 倍に再スケーリングしたベクトル $(300, 400)$ でテストしてみよう。どちらの行も $(0.849, 1.131)$ に写されるはずである。

```
norm = RMSNorm(2)
norm(torch.tensor([[3.0, 4.0], [300.0, 400.0]]))
```

```
tensor([[0.8485, 1.1314],
        [0.8485, 1.1314]], grad_fn=<DivBackward0>)
```

5.4.2.2 登録

単独のゲインテンソルを受け取る通常の関数を呼び出すのではなく、なぜ RMSNorm をモジュールのサブクラスに包むのだろうか。それは、モジュールへの登録によって、オプティマイザによるパラメータの自動検出、コンテナ内での合成可能性、チェックポイントのシリアライズ、デバイス間の移動という組み込み層が備える 4 つの機能すべてがカスタム層に自動的に付与されるからである（5.1 節、5.2 節）。このことをエンドツーエンドで確かめるため、2 つの線形層の間に RMSNorm を挟み、勾配ステップを 1 回進めてゲインを 1 から変化させたうえで、状態全体を新し

く初期化したクローンにコピーしてみよう。すると両方のモデルは完全に同一の出力を生成するはずである（ディスク上のチェックポイントへの拡張は 5.6 節で扱う）。

```
net = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
X = torch.randn(4, 20)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).pow(2).sum().backward()
optimizer.step()
clone = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
clone.load_state_dict(net.state_dict())
net[1].gain.data, torch.equal(net(X), clone(X))
```

```
(tensor([0.9681, 0.9702, 0.9415, 0.7670, 1.0491, 0.9845, 0.9079, 0.9954]),
 True)
```

実際、ゲインは 1 から更新され、クローンは訓練後のモデルとビット単位まで一致する。しかも `RMSNorm` の内部には、`self.gain` を登録する以外の定型コードは一切不要である。

5.4.2.3 フレームワーク実装との比較

統合テストに通ったからといって証明されるのは、自作の層がフレームワークに正しく組み込まれていることであり、数式処理にバグがないことではない。ライブラリにネイティブの同等実装が用意されている場合は常に、自明でない重みを使ってカスタム実装をそれと突き合わせることで、数値的な妥当性を即座に確認できる。

自明でないゲイン（`linspace(0.5, 1.5, 8)`）を自作の `RMSNorm` とネイティブの `nn.RMSNorm` の両方に設定し、出力が一致することを確かめる。

```
ours, native = RMSNorm(8), nn.RMSNorm(8, eps=1e-6)
with torch.no_grad():
    ours.gain.copy_(torch.linspace(0.5, 1.5, 8))
    native.weight.copy_(ours.gain)
X = torch.randn(16, 8)
torch.allclose(ours(X), native(X))
```

```
True
```

組み込みライブラリの層が仕様に合致する場合は常に、保守性と融合カーネルによる計算速度の観点から本番環境では組み込み層を優先すべきである。カスタム層を書くのは、ライブラリにない新しい演算や改良が必要なときに限るとよい。

5.4.3 固定状態をもつ層

層によっては、勾配降下法（gradient descent）で学習されるのでも呼び出しのたびにゼロから再計算されるのでもなく、構築時に一度だけ生成されてモデルの存続期間を通じて固定されたままになる配列に依存するものがある。古典的な例が**ランダム特徴量**（random features）層であり (Rahimi and Recht 2007)、ミニバッチ $\mathbf{X}$ を固定された標準正規分布の行列 $\mathbf{W} \in \mathbb{R}^{d \times D}$ で射影し、ランダムな位相 $\mathbf{b} \in \mathbb{R}^{D}$（$[0, 2\pi]$ から一様にサンプリングしたもの）だけシフトして、要素ごとにコサインを適用する。

$$\mathbf{Z} = \sqrt{2/D}\,\cos(\mathbf{X}\mathbf{W} + \mathbf{b}). \tag{5.7}$$

$\mathbf{Z}$ の各行は D 個の固定された非線形特徴量を与える。驚くべきことに、$\mathbf{W}$ や $\mathbf{b}$ を一切更新しなくても、その上に訓練可能な線形層を 1 つ載せるだけで多様な非線形関数を近似できる。

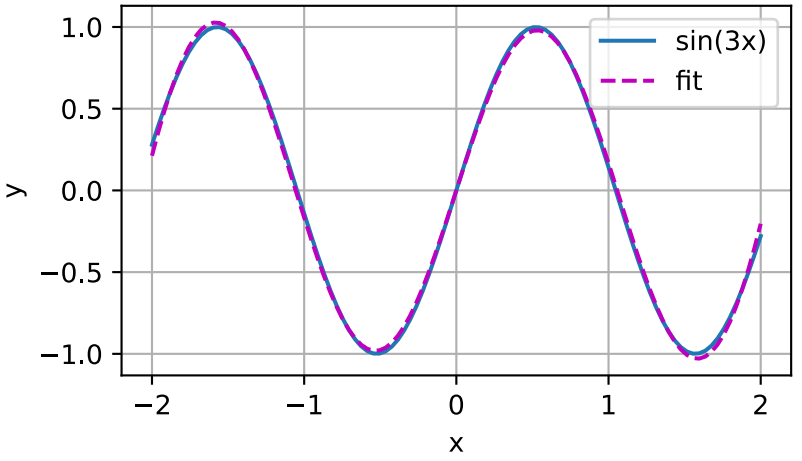
`Standardizer` の移動統計量（5.2.1.2 節）とまったく同様に、$\mathbf{W}$ と $\mathbf{b}$ は最適化の間は固定されたままでありながら、チェックポイントへの保存やデバイス間の移動ではモデルと行動をともにしなければならない。したがって、これらをバッファとして登録する。

```
class RandomFeatures(nn.Module):
    def __init__(self, d, num_features):
        super().__init__()
        self.register_buffer('W', torch.randn(d, num_features))
        self.register_buffer('b', 2 * torch.pi * torch.rand(num_features))

    def forward(self, X):
        return (2 / self.b.numel()) ** 0.5 * torch.cos(X @ self.W + self.b)
```

ここで実際に `RandomFeatures` を動かし、$y = \sin(3x)$ を $[-2, 2]$ 内の等間隔な 100 点にわたって近似してみよう。500 個のランダム特徴量を単一の線形層に入力し、Adam で訓練する。$\mathbf{W}$ と $\mathbf{b}$ はパラメータではなくバッファとして登録されているため、モデル全体をオプティマイザに渡しても線形ヘッドだけが更新される。

```
x = torch.linspace(-2, 2, 100)
X, y = x.reshape(-1, 1), torch.sin(3 * x)

def mse(net, X, y):
    return ((net(X)[:, 0] - y) ** 2).mean()

net = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
optimizer = torch.optim.Adam(net.parameters(), lr=0.05)
for _ in range(500):
    optimizer.zero_grad()
    mse(net, X, y).backward()
    optimizer.step()
print(f'mean squared error {mse(net, X, y):.2g}')
d2l.plot(x, [y, net(X)[:, 0].detach()], 'x', 'y',
         legend=['sin(3x)', 'fit'], figsize=(4.5, 2.5))
```

```
mean squared error 0.00054
```



学習された線形結合は $\sin(3x)$ をよく捉えており、平均二乗誤差（mean squared error、MSE）は 0.01 未満まで下がる（定数 0 を出力するベースラインでは 0.5 である）。

ここで、チェックポイントに訓練済みの線形ヘッドだけを保存した場合に何が起きるか見てみよう。新しくインスタンス化されたモデルはまったく別の $\mathbf{W}$ と $\mathbf{b}$ を引き直すため、線形層には訓練時に一度も見たことのない特徴量が入力されてしまう。以下では、線形ヘッドだけを復元した場合と、バッファを含む状態全体を復元した場合とを比較する。

```
fresh = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
fresh[1].load_state_dict(net[1].state_dict())
print(f'linear layer only: mean squared error {mse(fresh, X, y):.2g}')
fresh.load_state_dict(net.state_dict())
print(f'full state:        mean squared error {mse(fresh, X, y):.2g}')
```

```
linear layer only: mean squared error 0.96
full state:        mean squared error 0.00054
```

線形層の重みだけを読み込むと MSE は 2 桁以上も跳ね上がり（定数 0 を予測するのと変わらない）、一方でバッファを含む状態全体を復元すれば訓練済みの近似が正確に再現される。

これとまったく同じバッファの仕組みは、バッチ正規化（batch normalization）の移動平均や移動分散（7.3 節）をはじめ、後の章でも訓練対象外のテーブルや移動統計量を保持するために使われる。

5.4.4 カスタム勾配をもつ関数

ここまでは順伝播だけを書き、逆伝播の導出は自動微分（automatic differentiation）に任せてきた。自動微分はあらゆる演算を忠実に微分するが、ときにはまさにその忠実さが邪魔になることがある。冒頭の 4 ビット量子化の例に戻ろう。順伝播で重みを整数に丸める演算は区分的に定数であるため、真の導関数はほとんど至るところで 0 になり、丸め処理の手前にあるすべての重みの勾配が 0 になってしまう。

```
w = torch.tensor([0.9, 1.4, 2.6], requires_grad=True)
w.round().sum().backward()
w.grad
```

```
tensor([0., 0., 0.])
```

数学的には勾配が 0 であることは厳密に正しいが、実用上はその場で学習が完全に止まってしまう。**ストレートスルー推定器**（STE）(Bengio, Léonard, et al. 2013) は、順方向と逆方向の計算を切り離すことでこの問題を回避する。順伝播では従来どおり整数への丸めを行う一方、逆伝播では丸めが恒等関数であったかのように見なし、上流からの勾配をそのまま通過させる（導関数を 1 とする）。これは意図的に真の導関数とは**異なる**ものを用いているが、階段関数 $\mathrm{round}(x)$ と恒等関数とのずれは高々 $1/2$ に収まるため（Figure 5.7）、実際にはよく機能する。自動微分は順方向の関数とは異なる代理導関数（surrogate derivative）を勝手に作り出すことはできないため、逆伝播の規則を明示的に与える。

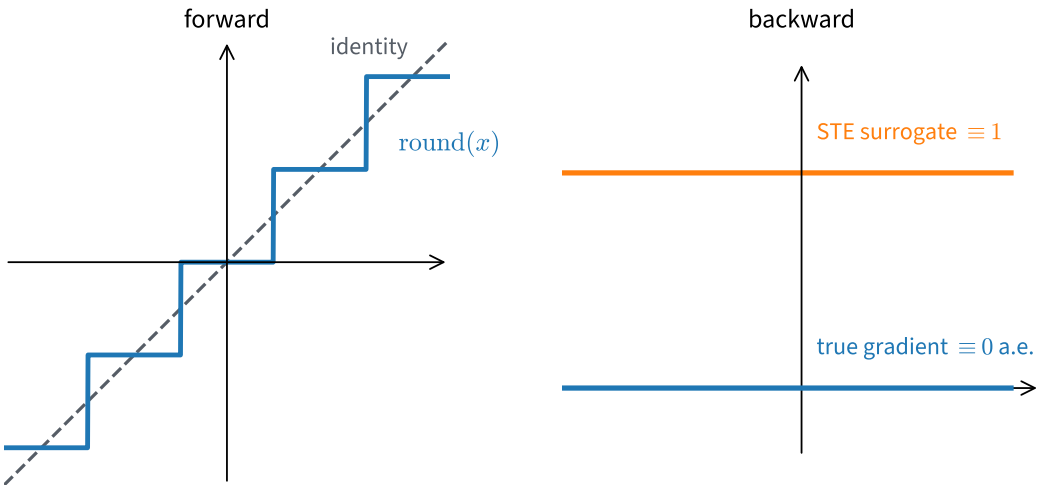


Figure 5.7: ストレートスルー推定器。順伝播：階段関数 round(x) は恒等関数（破線）の近くにとどまる。逆伝播：丸めの真の導関数はほとんど至るところで 0 であるが、この推定器は代わりに恒等関数の導関数である 1 を用いる。

PyTorch では、順方向と逆方向の 2 つを静的メソッドとしてもつ `torch.autograd.Function` のサブクラスを用いる。

```
class RoundSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, X):
        return X.round()

    @staticmethod
    def backward(ctx, grad_output):
        # Treat the forward pass as the identity: pass the gradient on
        return grad_output
```

呼び出しには必ず `RoundSTE.apply(X)` を使う（`forward` を直接呼ぶと自動微分への登録がバイパスされてしまう）。`backward` で順伝播のテンソルが必要な場合は、`ctx.save_for_backward` で保存する。

丸めた重みに $(1, 2, 3)$ を掛けて和をとることで、このカスタム逆伝播規則を確かめてみよう。ストレートスルー推定器は、$(1, 2, 3)$ をそのまま $\mathbf{w}$ に関する勾配として伝えるはずである。

```
w.grad = None
y = (RoundSTE.apply(w) * torch.tensor([1.0, 2.0, 3.0])).sum()
y.backward()
w.grad
```

```
tensor([1., 2., 3.])
```

順伝播は引き続き整数に丸められた重みで計算される一方、勾配はあたかも丸めが恒等関数であるかのようにそのまま $\mathbf{w}$ へと逆伝播する。勾配降下法は、この代理を通じて実際に学習できるのだろうか。以下では、順伝播中に重みが整数へ丸められる 8 個の重みをもつ線形モデルを、$[-7, 7]$（4 ビット符号付き整数の範囲 $[-8, 7]$ の対称な部分）から一様に引いた重みで生成した合成データ（synthetic data）に適合させる。$\mathbf{w} = \mathbf{0}$ から出発し、通常の丸めとストレートスルー推定器を、初期損失および基準点となる真の重みを丸めたときの損失（各重みを個別に丸めたものが最良の整数近似とは限らない）と並べて比較する。

```
X = torch.randn(256, 8)
w_true = torch.empty(8).uniform_(-7, 7)
y = X @ w_true

def fit(rnd, X, y, num_steps=100, lr=0.05):
    w = torch.zeros(8, requires_grad=True)
    optimizer = torch.optim.SGD([w], lr=lr)
    for _ in range(num_steps):
        loss = ((X @ rnd(w) - y) ** 2).mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return loss.item()

print(f'start {(y ** 2).mean():.3g}, '
      f'rounding {fit(torch.round, X, y):.3g}, '
      f'straight-through {fit(RoundSTE.apply, X, y):.3g}, '
      f'rounded target {((X @ w_true.round() - y) ** 2).mean():.3g}')
```

```
start 82.9, rounding 82.9, straight-through 1.04, rounded target 0.557
```

通常の丸めではすべての勾配が 0 になるため、損失は初期値から一歩も動かない。一方、ストレートスルー推定器を用いると、損失は数十〜数百から 1 または 2 程度まで急減し、正解の重みを丸めたときの損失の数倍以内に収まる。意図的に近似された代理勾配（surrogate gradient）によって、量子化モデルの訓練が確かに成功するのである。

恒等関数は数ある代理の 1 つにすぎない。完全な 4 ビット量子化器は入力を $[-8, 7]$ にクランプする処理も含むため、実用的な実装では表現可能な範囲の内側でのみ勾配を通し、飽和した外側では勾配を 0 にすることが多い（演習問題を参照）。適切な代理を選ぶことは、近似対象の演算に合わせたモデリング上の設計判断である。

代理が恒等関数**である**場合には、カスタムの自動微分関数を書く必要を完全になくせる洗練された 1 行のテクニックがあり、本番コードで最もよく目にするのもこの書き方である。順伝播では恒等関数として振る舞い逆伝播では導関数が 0 になる**勾配停止**（stop-gradient）演算子を用いて $X + \mathrm{stop}(\mathrm{round}(X) - X)$ と書くと、順伝播では $\mathrm{round}(X)$ と評価される一方、逆伝播では先頭の X のみを通じて微分される。クリップ付きの量子化器のように恒等関数でない代理には依然としてカスタム逆伝播の仕組み全体が必要だが、単純なストレートスルー丸めであれば、両方の定義は値も勾配も完全に一致する。

PyTorch における勾配停止演算は `detach()` メソッドであり、同じ値をもち自動微分から定数として扱われるテンソルを返す。

```
def round_ste2(X):
    return X + (X.round() - X).detach()

def grad_of(f):
    w.grad = None
    (f(w) * torch.tensor([1.0, 2.0, 3.0])).sum().backward()
    return w.grad

(torch.allclose(RoundSTE.apply(w), round_ste2(w)),
 torch.allclose(grad_of(RoundSTE.apply), grad_of(round_ste2)))
```

```
(True, True)
```

手書きした逆伝播規則には誤りが紛れ込む可能性があり、順伝播の結果だけではそれに気づけない。標準的なテスト手法は**勾配チェック**（gradient check）である。各入力を微小量だけ摂動させて出力の変化量を測り、この有限差分（finite difference）による傾きと逆伝播規則が返す勾配とを比較する。以下では、滑らかな組み込み関数と先ほどのストレートスルー規則をチェックしてみよう。

```
def check(f):
    x = torch.tensor([0.9, 1.4, 2.6], dtype=torch.float64, requires_grad=True)
    return torch.autograd.gradcheck(f, (x,), raise_exception=False)

check(torch.tanh), check(RoundSTE.apply)
```

```
(True, False)
```

組み込み関数はテストに合格し、ストレートスルー推定器は不合格となる。この不合格は設計どおりの結果である。順伝播は階段関数であり、有限差分で測れる場所では傾きが 0 であるのに対し、ここで定義した逆伝播規則は傾き 1 を返すからである。つまり勾配チェックが教えてくれるのは、ある規則がその順伝播の真の導関数になっているかどうかであり、その規則が有用かどうかではない。演習問題のクランプのように、厳密な導関数であることを意図したすべての規則に対して勾配チェックを実行するとよい。

なお、カスタム勾配は既存のテンソルプリミティブから構築された式の導関数を上書きするものである。カスタム GPU カーネルをゼロから書くことはシステム分野の別領域であり本書の範囲外だが、フレームワークがすでに提供しているプリミティブを使って計算速度を最大化する方法については第 13 章で扱う。

`RMSNorm` のような層については、重要な疑問が 1 つ残っている。その浮動小数点演算は本当に安全なのだろうか。活性化値を二乗して和をとる計算は、16 ビット形式では容易にオーバーフローを起こしうる。5.5 節では、数値形式と混合精度（mixed precision）訓練を直接取り上げて詳しく調べる。

5.4.5 まとめ

カスタム層とは、フレームワークの基底モジュールをサブクラス化して順伝播を実装したものにほかならない。テンソルを登録しておけば、最適化、コンテナへの組み込み、チェックポイント保存、デバイス配置が自動的に処理される。学習される重みは（`RMSNorm` のゲインのように）パラメ

ータとして登録し、訓練されない固定テンソルは（RandomFeatures のランダム射影のように）バッファとして登録することで、オプティマイザに公開することなくチェックポイントに保持できる。丸め処理のように演算の真の導関数から有益な情報が得られない場合には、ストレートスルー推定器（あるいはその勾配停止によるショートカット）のようなカスタム逆伝播規則を用いることで訓練を進められる。仕様に合致する組み込みライブラリの層がある場合は常に、保守性と融合カーネルの計算速度の観点から組み込み層を優先すべきである。

5.4.6 演習問題

1. [code] **RMSNorm のスケール不変性**。●$\epsilon = 0$ とする。
   1. $\mathrm{RMSNorm}(c\mathbf{x}) = \mathrm{RMSNorm}(\mathbf{x})$ がすべての $c > 0$ に対して成り立つことを示せ。
   2. 両辺を c に関して $c = 1$ で微分し、RMSNorm のヤコビ行列 $\mathbf{J}$（点 $\mathbf{x}$ におけるもの）が $\mathbf{J}\mathbf{x} = \mathbf{0}$ を満たすことを導け。
   3. 任意の損失の RMSNorm の入力 $\mathbf{x}$ に関する勾配が $\mathbf{x}$ と直交することを導き、ランダムな $\mathbf{x}$ とランダムな線形損失に対して自動微分でこれを確かめよ。微小な勾配ステップは $\mathbf{x}$ の長さにどのような影響を与えるか。

2. [code] **RMSNorm のバイアス**。オプションの学習されるバイアス $\mathbf{b}$ を RMSNorm に追加し（ゲインの後に適用するシフトであり、層正規化がもっていた要素の一部を復活させるもの）、これを含むモデルの状態辞書に期待どおりの項目が増えることを確かめよ。次に、この層を重み $\mathbf{W}$ とバイアス $\mathbf{c}$ をもつ線形層の直前に配置し、出力を変えることなく $\mathbf{b}$ を $\mathbf{c}$ に吸収できることを示せ。吸収後のバイアスはどうなるか。

3. [code] **ランダム特徴量によるカーネルの近似**。$\mathbf{w}$ の各成分は独立な標準正規分布に従い、b は $[0, 2\pi]$ 上の一様分布に従うとする。
   1. $\cos\alpha\cos\beta = \frac{1}{2}(\cos(\alpha-\beta)+\cos(\alpha+\beta))$ および $\mathbb{E}[\cos(\mathbf{w}^\top\mathbf{u})] = \exp(-\|\mathbf{u}\|^2/2)$ を用いて、RandomFeatures の特徴量 $\mathbf{z}(\mathbf{x})$ が $\mathbb{E}[\mathbf{z}(\mathbf{x})^\top\mathbf{z}(\mathbf{x}')] = \exp(-\|\mathbf{x}-\mathbf{x}'\|^2/2)$ を任意の特徴量数 D に対して満たすことを示せ。
   2. 距離が 1 である 2 つの入力に対して、$D = 10$、100、1,000、10,000 としてこれを数値的に確かめよ（各 D について複数の層にわたって誤差を平均せよ）。D が大きくなるにつれて誤差はどのように減少するか。
   3. 本節の $\sin(3x)$ の近似を、特徴量数 $D = 5$ と $D = 50$ の場合、およびランダム特徴量の代わりに x のみを入力とする線形モデルの場合で繰り返せ。それぞれどの程度の誤差に達するか。また、それはなぜか。

   (Rahimi and Recht 2007) に基づく。

4. [code] **カスタム勾配をもつクランプ**。autograd.Function を実装し、順伝播を X.clamp(lo, hi) とし、逆伝播では入力がクランプ範囲の厳密に内側にあった箇所でのみ勾配を通すようにせよ。範囲の内側、外側、およびちょうど境界上にある入力について、その勾配をネイティブの torch.clamp の勾配と比較せよ。ネイティブの演算は境界上でどの規約を採用しているか。

   PyTorch チュートリアル Learning PyTorch with Examples および JAX ノートブック Custom derivative rules に基づく。

5.5 数値形式

これまでのモデルでは、ニューラルネットワークに本当にそれだけのビット数が必要かを問うことなく、すべてのテンソルに 32 ビット浮動小数点（floating point、fp32）を使ってきた。ビッ

ト幅を半分にすれば、メモリとスループットの両面ですぐに恩恵が得られる。たとえば、本章で一貫して扱う 80 億パラメータのモデルは、重みに 32 ビット精度では 32 GB を要するが 16 ビットなら 16 GB で済み、最新の GPU は 16 ビットの行列積を 32 ビットの少なくとも 2 倍高速に実行する。では、16 ビットで訓練しても同じモデルが得られるだろうか。

単純にはうまくいかない。16 ビット形式は目に見えて粗く、実行時エラーを起こさない 2 つのサイレントな失敗モードを招く。fp16 では 300^2 の時点で `inf` にオーバーフローし、訓練中に `inf` と NaN の値があふれ返る。bf16 では、0.001 を重み 1 に足しても丸められてそのまま 1 に戻ってしまい、勾配が健全に見えていても学習が静かに停滞する。

本節では、これら両方の落とし穴を避けつつ 16 ビットで確実に訓練する方法を示す。すべてのテンソルは **dtype**（数値形式）を持ち、どの形式も**ダイナミックレンジ**（表現可能な数値がどこまで大きく、または小さくなれるか）と**精度**（precision、隣り合う数値がどれだけ細かく並ぶか）のトレードオフの上にある。**混合精度訓練**（mixed-precision training）では、微小な勾配による更新が忠実に累積するよう重みの fp32 マスターコピーを保持する一方で、計算負荷の高い行列積の実行と活性化値（activation）の保存を 16 ビットで行い、さらに fp16 を使う際には小さな勾配が 0 にアンダーフローしないよう**損失スケーリング**（loss scaling）を組み合わせる。5.2 節ではパラメータあたりのバイト数を数えたのに対し、ここでは数値 1 個あたりのバイト数を制御する。2 番目の項（dtype の昇格とモデルのキャスト）は実用上のリファレンスであり、ざっと目を通すだけでよい。すべての例は CPU でも GPU でも動作するが、16 ビットによる実際の高速化には GPU の行列演算ユニットが必要であり、その計測は 5.7 節で行う。

```
import random
```

5.5.1 浮動小数点形式

5.5.1.1 範囲と精度

300 を両方の 16 ビット形式で 2 乗してみるだけで、低精度が引き起こす 2 つの異なる問題が浮かび上がる（300 という値はニューラルネットワークの内部では決して珍しいものではなく、正規化前のロジットや分散計算の中間和は容易にこの大きさに達する）。

```
x = torch.tensor(300.0)
x.to(torch.float16)**2, x.to(torch.bfloat16)**2
```

```
(tensor(inf, dtype=torch.float16), tensor(90112., dtype=torch.bfloat16))
```

fp16（**半精度**、half precision）では、300^2 はいきなり `inf` にオーバーフローする。最大の有限値がわずか 65504 しかないためである。bf16（**brain floating point**）では結果は有限にとどまるものの（90000 ではなく 90112）、すでに有効数字 3 桁目で丸め誤差（rounding error）が生じている。どちらの形式も 16 ビットに収まるが、そのビットの割り当て方が異なるにすぎない。任意の正規化浮動小数点数は、符号ビット s、指数 e、および t ビットの仮数部（小数部 $f \in [0, 1)$ を符号化する）を格納し、次の値を表す。

$$x = (-1)^s \cdot 2^e \cdot (1 + f). \tag{5.8}$$

指数部のビット数はダイナミックレンジ（その形式で表現できる最大および最小の絶対値 $|x|$）を決定し、仮数部のビット数は相対精度を決定する。具体的には、1 から 2 の間では隣り合う数値の間隔は 2^{-t} であり、2^e から 2^{e+1} の間では 2^{e-t} となって、スケールによらず一様な相対間隔が保たれる。ここで扱う 3 つの形式のビット配分は次のとおりである。

形式	符号	指数部	仮数部
fp32	1	8	23
bf16	1	8	7
fp16	1	5	10

トレードオフは一目瞭然である。bf16 は fp32 の 8 ビットの指数部をすべて維持しつつ仮数部を 7 ビットに切り詰めることで、解像度は粗くなる代わりに fp32 と同じダイナミックレンジを実現している。fp16 はその逆で、仮数部を 10 ビット残す代わりに指数部を 5 ビットに抑えており、局所的な刻み幅は細かくなるものの表現範囲ははるかに狭くなる。先ほどの 2 つの出力は手計算でも確かめられる。指数部が 5 ビットのとき、e は $[-14, 15]$ の範囲をとり、fp16 の上限は $2^{15}(2 - 2^{-10}) = 65504$ となる。仮数部が 7 ビットのとき、2^{16} と 2^{17} の間にある bf16 の値の間隔は $2^{9} = 512$ であるため、90000 は $175 \cdot 512 = 89600$ と $176 \cdot 512 = 90112$ の間に位置し、より近い値である 90112 に丸められる。

Figure 5.8 にこれらのビット配分を並べて示す（IEEE 浮動小数点演算のより詳しい解説については 35.8 節を参照）。

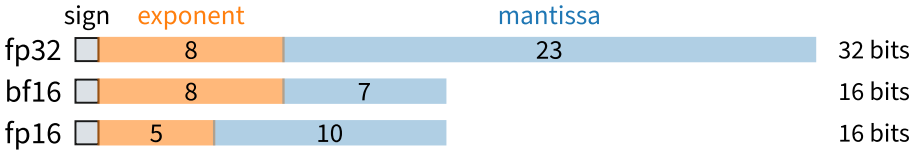


Figure 5.8: 表の各形式のビット配置を符号ビットの位置で揃えたもの。棒が長いほどビット数が多いことを表す。bf16 は fp32 の 8 ビット指数部を維持し、fp16 と bf16 は同じ 16 ビットを正反対のやり方で分割している。

各形式の正確な数値限界は `finfo` で照会でき、ここでは `max`（最大の有限値）、`tiny`（正の最小の**正規化数**、normal number）、`eps`（1 の直上における間隔 2^{-t}）の 3 つのフィールドに注目する。指数が下限に達する `tiny` 未満の領域では、等間隔に並ぶ**非正規化数**（subnormal number）を通じて 0 まで段階的に精度が低下する。fp16 ではこれらは 2^{-24} の倍数となるため、3×10^{-5} 付近の値はまだ 9 ビットの有効ビットを保つのに対し、10^{-7} 付近の値はわずか 1 ビットしか保持できない。

```
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    fi = torch.finfo(dtype)
    print(f'{str(dtype):15} max {fi.max:10.3e}  tiny {fi.tiny:9.3e}'
          f'  eps {fi.eps:8.3e}')
```

```
torch.float32   max  3.403e+38  tiny 1.175e-38  eps 1.192e-07
torch.bfloat16  max  3.390e+38  tiny 1.175e-38  eps 7.812e-03
torch.float16   max  6.550e+04  tiny 6.104e-05  eps 9.766e-04
```

bf16 は fp32 と同じ 8 ビットの指数部を共有するため、`tiny` は fp32 と一致し、`max` もほぼ一致する一方、`eps` は 0.0078 であり 10 進で 2〜3 桁の精度となる。fp16 は 10 進で 3〜4 桁を分解できるものの、65504 を超えるとオーバーフローし、6.1×10^{-5} 未満では非正規化数に落ち込み、$2^{-24} \approx 6.0 \times 10^{-8}$ 未満の正の値は一切表現できない。深層学習にとって、どちらの不足がより深刻だろうか。活性化値や勾配は日常的に何桁にもわたって変化する。表現範囲を超えると数値が `inf` や 0 に変わって破滅的な結果を招くのに対し、精度がやや粗い程度であれば穏やかな丸

めノイズが加わるだけで済む。最新のハードウェアで bf16 が標準の 16 ビット訓練形式となり、fp16 が主に旧世代の GPU や推論（inference）向けに残されているのはそのためである。

5.5.1.2 小さな更新の丸め

bf16 の粗い間隔が**実際に**悪影響を及ぼすのは、パラメータ更新（parameter update）の場面である。勾配降下法（gradient descent）の 1 ステップは重みを ηg だけ動かすが、その大きさは重みの絶対値の 10^{-3} 以下であることが多い。最近傍丸め（round-to-nearest）のもとでは、局所的な格子間隔の半分より小さい更新ステップは完全に消えてしまう。以下では、各形式で 10^{-3} と 10^{-4} を重み 1 に加算してみる。

```
w = torch.tensor(1.0)
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    print(f'{str(dtype):14}: {float(w.to(dtype) + 1e-3):.7f}, '
          f'{float(w.to(dtype) + 1e-4):.7f}')
```

```
torch.float32 : 1.0010000, 1.0001000
torch.bfloat16: 1.0000000, 1.0000000
torch.float16 : 1.0009766, 1.0000000
```

fp32 では、どちらの増分も問題なく反映される。1 付近での間隔が $2^{-7} \approx 0.0078$ である bf16 では、どちらも丸められて 1 に戻り、重みは完全に凍りついたままとなる。fp16（間隔 $2^{-10} \approx 0.00098$）では、10^{-3} のステップは（格子 1 目盛り分に丸められて）生き残るが、10^{-4} は消えてしまう。Figure 5.9 は両方の格子を同じ縮尺で描いたものであり、更新が反映されるのは重みを格子間隔の半分より大きく動かしたときに限られる。したがって、重みを直接 16 ビットで保持すると、たとえすべての勾配が正確であっても最適化が停滞し得る。訓練においてマスター重み（master weights）を fp32 で保持するのはこのためである。

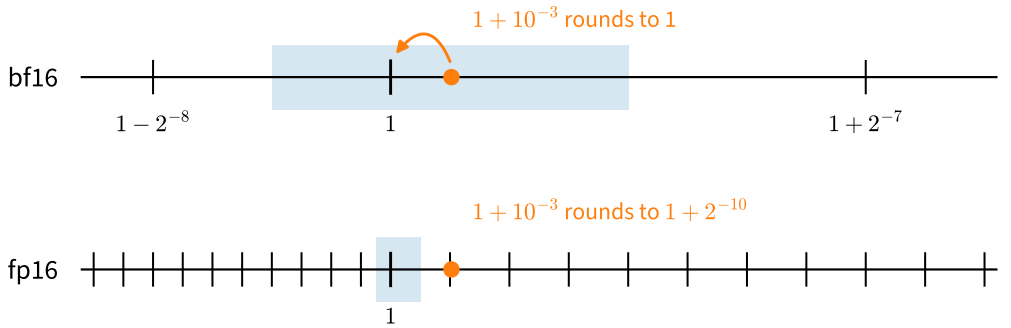


Figure 5.9: 重み 1 付近における bf16 の格子（上）と fp16 の格子（下）を 1 つの縮尺で描いた図。網掛けの区間は丸められて 1 に戻る値の範囲を表し、左右それぞれに格子間隔の半分まで広がっている（1 より下側の間隔は 1 より上側の間隔の半分である）。bf16 では更新後の $1 + 10^{-3}$ がこの区間の内側に収まるため失われる。fp16 では区間の外側に出るため、次の格子点 $1 + 2^{-10}$ に丸められる。

5.5.2 dtype の昇格とモデルのキャスト

5.5.2.1 昇格

型が一致しないオペランドを式の中で組み合わせると、結果の dtype はどうなるだろうか。fp16 テンソルに、fp32 テンソル、Python の float、整数テンソルをそれぞれ加算して確かめてみよう。

```
x16 = torch.ones(3, dtype=torch.float16)
x32 = torch.ones(3, dtype=torch.float32)
(x16 + x32).dtype, (x16 + 1.0).dtype, (x16 + torch.arange(3)).dtype
```

```
(torch.float32, torch.float16, torch.float16)
```

2 つの浮動小数点テンソルを組み合わせるとより広い形式へ昇格する（また fp16 + bf16 は fp32 へ昇格する）。そのため、紛れ込んだたった 1 つの fp32 テンソルによって、後続の 16 ビット計算全体が暗黙のうちに fp32 へ昇格してしまう。Python の float リテラルや整数テンソルは、浮動小数点テンソルを昇格させない。

5.5.2.2 モデルのキャスト

パラメータは単なるテンソルであるため、モデル全体を 16 ビットにキャストできる。本節を通して、4.2 節で構築した Fashion-MNIST 用の MLP を使用する。これは 784 個の入力、256 ユニットの ReLU 隠れ層（hidden layer）、10 個の出力を持ち、重みを $\mathcal{N}(0, 0.01^2)$、バイアスを 0 で初期化したものである（パラメータ数は 203,530 個で、fp32 では 814,120 バイトを占める）。

```
def mlp():
    net = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    for layer in net[1], net[3]:
        nn.init.normal_(layer.weight, std=0.01)
        nn.init.zeros_(layer.bias)
    return net

def param_bytes(net):
    return sum(p.numel() * p.element_size() for p in net.parameters())

net = mlp()
param_bytes(net)
```

```
814120
```

`net.to(dtype)`（または `net.bfloat16()`）はすべてのパラメータとバッファをキャストする。各層は、重みと dtype が一致しない入力を拒否する。

```
net = net.bfloat16()
X = torch.randn(8, 1, 28, 28)
try:
    net(X)
except RuntimeError as e:
    print(e)
print(net(X.bfloat16()).dtype, param_bytes(net))
```

```
mat1 and mat2 must have the same dtype, but got Float and BFloat16
torch.bfloat16 407060
```

キャストによってパラメータのメモリ使用量は即座に半減するが、16 ビットのモデルが fp32 の入力を自動的にキャストするか、dtype の不一致として拒否するかはフレームワークによって異なる。

推論においては、訓練済みモデルを 16 ビットにキャストすれば（タスクの正解率が維持されることを確認した上で）重みのメモリ使用量をただちに半減できる。しかし**訓練**においては、純粋な 16 ビットの重みは先ほど見た小さな更新の丸めという落とし穴に直面する。以下ではこれを実験で確かめる。

5.5.3 混合精度訓練

5.5.3.1 マスター重みと 16 ビット計算

一見するとジレンマに直面しているように思える。信頼できる重み更新には 32 ビットでの保存が求められる一方で、ハードウェアによる高速化は 16 ビット演算にあるからである。**混合精度訓練** (Micikevicius et al. 2018) は、パラメータの保存と計算を切り離すことでこの対立を解消する。fp32 の**マスター重み**を保持して小さな更新がすべて正確に累積するようにしつつ、計算負荷の高い順伝播と逆伝播の行列積および活性化値にはそれを 16 ビットにキャストして使い、損失のような精度に敏感なリダクションは fp32 のまま保つのである。Figure 5.10 に 1 回の訓練ステップの概要を示す。

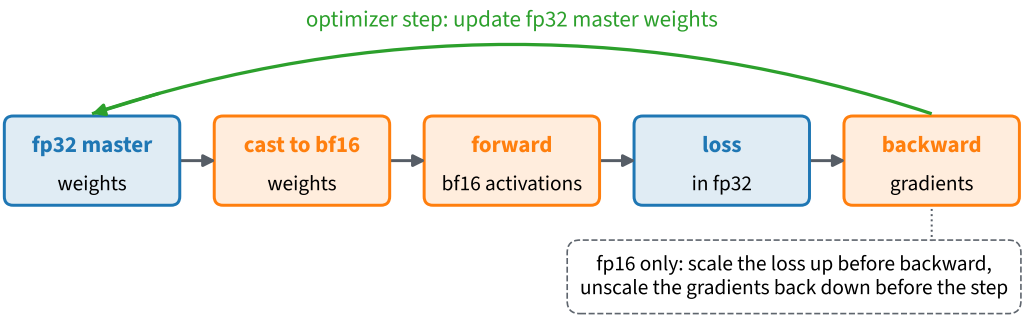


Figure 5.10: 混合精度訓練の 1 ステップ。順伝播のために fp32 のマスター重みが bf16 にキャストされ、bf16 の活性化値が生成される。損失は fp32 で計算され、逆伝播によって勾配が求められ、オプティマイザがそれを fp32 のマスター重みに適用する。fp16 を用いる場合は、逆伝播の前に損失がスケールアップされ、更新の前に勾配が元のスケールに戻される。

`torch.autocast` はパラメータを fp32 のまま残し、対象の演算（行列積や畳み込み）を自動的に 16 ビットへキャストする一方、リダクションと損失は fp32 に保つ。

```
net = mlp()
with torch.autocast('cpu', dtype=torch.bfloat16):
    Y = net(X)
Y.dtype, net[1].weight.dtype
```

```
(torch.bfloat16, torch.float32)
```

5.5.3.2 fp32 と 16 ビットでの訓練

16 ビット演算は最適化にどの程度影響するのか、また fp32 のマスターコピーを省くと何が起きるのだろうか。以下では、256 枚の画像からなる Fashion-MNIST のミニバッチ上で、通常の SGD を用いて MLP を 3 通りで訓練する。すべて fp32、純粋な bf16 重み（fp32 のマスターコピーなし）、および混合精度の 3 つである。3 回の実行はいずれも同一のランダムな初期重みから開始し、まったく同じミニバッチを同じ順序で処理する。

学習率（learning rate）に注目してほしい。ここでは $\eta = 0.01$（4.2 節で用いた値の 10 分の 1）に設定し、1000 ステップ（約 4 エポック）実行する。なぜ η がこれほど大きく影響するかは、先ほどの丸めの解析から説明できる。更新量 ηg は局所的な bf16 の格子間隔の半分を下回ると消えてしまうため、学習率が小さいほど多くの更新が失われるのである。小さな学習率は決して人為的な設定ではない。実際のどの訓練でも、学習率が減衰すればいずれこの領域に到達する。訓練を始める直前に、これら 2 つの学習率のそれぞれでどれだけの更新が失われるかを計測しよう。

```
data = d2l.FashionMNIST(batch_size=256)
batches = list(torch.utils.data.DataLoader(data.train, batch_size=256,
                                           drop_last=True))  # One epoch
val_batch = next(iter(torch.utils.data.DataLoader(data.val, batch_size=1024)))
```

訓練用のミニバッチと 1,024 枚の画像からなる検証用スライスをあらかじめメモリに読み込んだ上で、`train` は 1,000 ステップの SGD を $\eta = 0.01$ で実行し、ステップごとの損失と最終的な検証正解率（accuracy）を返す。

```
def train(net, mixed=False, num_steps=1000):
    dtype = next(net.parameters()).dtype  # The inputs follow the weights
    optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
    losses = []
    for step in range(num_steps):
        X, y = batches[step % len(batches)]
        with torch.autocast('cpu', dtype=torch.bfloat16, enabled=mixed):
            loss = F.cross_entropy(net(X.to(dtype)).float(), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.detach())
    Xv, yv = val_batch
    with torch.no_grad():
        accuracy = (net(Xv.to(dtype)).argmax(1) == yv).float().mean().item()
    return torch.stack(losses).tolist(), accuracy
```

`train` の中にある 1 つの実装上の工夫は、精度とは無関係にシステム面で良い習慣である。ループ中はステップごとの損失テンソルをデバイス上に置いたままにし、最後にまとめて Python の float へ変換することで、毎ステップの CPU と GPU 間の同期（synchronization）による待ち時間を回避している（5.7 節）。

訓練の前に、最初のミニバッチを使って丸めの議論を確かめておこう。初期化（initialization）時点における第 1 層の非ゼロの更新量 ηg のうち、bf16 で保存した場合に $\eta = 0.01$ と $\eta = 0.1$ でそれぞれどれだけの割合が丸めで消えてしまい、失われた更新が更新量全体の絶対値和のうちどれだけの割合を占めるだろうか。

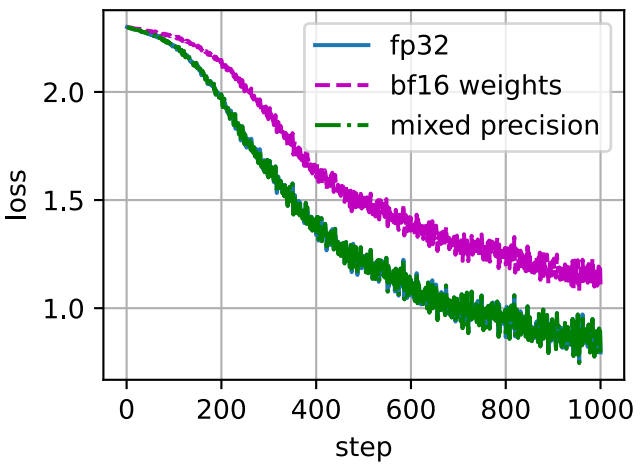
```
net = mlp()
X, y = batches[0]
F.cross_entropy(net(X), y).backward()
w, g = net[1].weight.detach(), net[1].weight.grad
w16, moved = w.bfloat16(), g != 0  # Count only the nonzero updates
for lr in (0.01, 0.1):
    lost = ((w16.float() - lr * g).bfloat16() == w16) & moved
    share = lost.sum() / moved.sum()
    mass = g[lost].abs().sum() / g.abs().sum()
    print(f'lr {lr}: {share:.0%} of updates rounded away, '
          f'{mass:.0%} of the update mass')
```

```
lr 0.01: 92% of updates rounded away, 80% of the update mass
lr 0.1: 52% of updates rounded away, 16% of the update mass
```

$\eta = 0.01$ では更新の約 10 個中 9 個が丸めで消え、それらは更新量全体の大きさ（出力中の「mass」）の約 5 分の 4 を占めている。つまり、勾配ステップの大部分が重みにまったく届かない。$\eta = 0.1$ でも依然として約半分の更新が失われるが、それらは小さな成分であり、大きさの合計に占める割合は約 15% にすぎない。そのためステップの大半は反映され、学習率が減衰するまで問題は隠れたままになる。それでは、$\eta = 0.01$ ですべての 3 通りを訓練してみよう。

```
def seeded_mlp(seed):
    torch.manual_seed(seed)  # The same seed gives the same initial weights
    return mlp()

seed = random.randrange(2**31)  # One random draw, shared by all runs
loss32, acc32 = train(seeded_mlp(seed))
loss_bf16, acc_bf16 = train(seeded_mlp(seed).bfloat16())
loss_mixed, acc_mixed = train(seeded_mlp(seed), mixed=True)
print(f'validation accuracy: fp32 {acc32:.3f}, bf16 weights {acc_bf16:.3f}, '
      f'mixed {acc_mixed:.3f}')
d2l.plot(list(range(1, 1001)), [loss32, loss_bf16, loss_mixed], 'step',
         'loss', legend=['fp32', 'bf16 weights', 'mixed precision'])
```

```
validation accuracy: fp32 0.684, bf16 weights 0.639, mixed 0.685
```



fp32 と混合精度の損失曲線はほぼ完全に重なっており、検証正解率も約 1 パーセントポイント以内で一致する。対照的に、純粋な bf16 重みは最初の 100 ステップのうちに遅れをとり始める。1000 ステップ後の損失は 0.85 に対して約 1.15 にとどまり、検証正解率も 68% に対して約 64% と後れをとっている。スカラーでの実験が予測したとおり、重みを直接 bf16 で保持すると第 1 層の更新の大半が丸めで消えてしまうのに対し、混合精度では fp32 のマスター重みへ正確に適用される。3 回の実行はいずれも収束（convergence）までは訓練していないが、重要なのは同一のデータとステップ数での対等な比較である。

混合精度で何が得られ、何が得られないかについて、2 つの重要な注意点がある。第 1 に、CPU 上の小規模な MLP では実時間の高速化は見られない。16 ビットによる高速化は、大規模な行列積を処理する専用の GPU テンソルコアによってもたらされるものであり、これについては 5.7.2.1 節でベンチマークを行う。第 2 に、混合精度は 5.2 節で見たパラメータあたりの静的なメモリ下限を縮小**しない**。マスター重みとオプティマイザのモーメントが fp32 のまま残るからである。冒頭で述べた 16 GB というメモリ使用量は、**推論**用にキャストした 8B モデルに対する値である。**訓練**中に混合精度が削減するのは、活性化メモリ（activation memory、16 ビットで保存される活性化値が半減する）と行列積の計算時間である。

5.5.3.3 fp16 のための損失スケーリング

bf16 であれば、マスター重みだけで手法として完結する。しかし fp16 では、指数部の範囲が狭いために下限側で 2 つ目の落とし穴が生じる。小さな勾配が**アンダーフロー**するのである。絶対値が $2^{-14} \approx 6.1 \times 10^{-5}$ を下回ると非正規化数に落ちて有効ビットが失われ、最小の非正規化数の半分（$\approx 3 \times 10^{-8}$）を下回る値はすべて 0 に丸められる。10^{-8} の勾配は fp16 では消えてしまうが、これに 2^{16} を掛ければ安全な正規化数の範囲へと引き上げられる。

```
g = torch.tensor(1e-8)
g.half(), (g * 2**16).half()
```

```
(tensor(0., dtype=torch.float16), tensor(0.0007, dtype=torch.float16))
```

実際には、微小な勾配はどれほど頻繁に現れるのだろうか。以下では、最初のミニバッチに対する MLP の第 1 層の fp32 勾配を計算し、非ゼロ要素のうち fp16 の非正規化数または 0 の領域に落ちる割合を、スケーリングなしの場合と 2^{16} 倍にスケーリングした場合の両方で計測する。

```
net = mlp()
X, y = batches[0]
F.cross_entropy(net(X), y).backward()
g = net[1].weight.grad.flatten()
g = g[g != 0]
for scale in (1, 2**16):
    print(f'scale {scale:5}: '
          f'{((g * scale).abs() < 2**-14).float().mean():.1%} subnormal, '
          f'{((g * scale).half() == 0).float().mean():.2%} zero')
```

```
scale     1: 32.5% subnormal, 0.21% zero
scale 65536: 0.0% subnormal, 0.00% zero
```

第 1 層の非ゼロ勾配のおよそ 3 分の 1 が fp16 の非正規化数の範囲に入り、約 1000 個に 2 個は完全に 0 までアンダーフローしている。この小規模な MLP ではごく微小な要素しか消えないため影響は軽微であり、実際、fp16 が完全に破綻し損失スケーリングによって救われるようなトイモデルを作るのは難しい。深刻なアンダーフローが直撃するのは、損失を多数のトークンやピクセルにわたって平均し、勾配が何十もの層を通過して減衰する大規模で深いモデルである。

Micikevicius et al. (2018) は、いくつかの ImageNet 分類モデルはスケーリングなしの fp16 でも問題なく訓練できた一方で、物体検出モデルは発散し、大規模な再帰型言語モデル（language model）は損失スケーリングを有効にして初めて fp32 と同等の正解率に到達したことを報告している。

勾配に 2^{16} を掛けると、fp16 のオーバーフロー上限である 65504 より十分下に収まったまま、非正規化数となる要素がすべて解消される。これこそが**損失スケーリング**の本質である。逆伝播の前に、スカラーの損失に大きな定数 S を掛ける。微分の線形性により、誤差逆伝播法（backpropagation）の過程で生じるすべての中間勾配は S 倍に引き上げられる。そしてオプティマイザのステップを実行する前に、微小な値を忠実に表現できる fp32 上で勾配を S で割り戻す。これにより、数学的なパラメータ更新を一切変えることなく、中間的な fp16 勾配をアンダーフロー領域の外に保つことができる。

S を静的に選ぶ方法は脆い。小さすぎれば勾配がアンダーフローし、大きすぎれば大きな勾配が `inf` にオーバーフローする。**動的損失スケーリング**（dynamic loss scaling）はこの選択を自動化する。大きな S から開始して毎ステップ勾配を検査し、`inf` や NaN が現れたらオプティマイザの更新をスキップして S を半分にする。逆に正常なステップが一定回数続いたら、fp16 の表現範囲の上限付近を維持するよう S を 2 倍にする。Figure 5.11 にこのループの 1 ステップを示す。

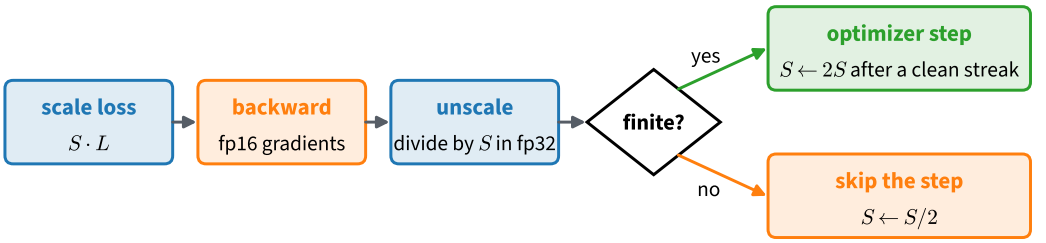


Figure 5.11: 動的損失スケーリングの 1 ステップ。損失に S を掛け、逆伝播を fp16 で実行し、勾配を fp32 で S で割る。すべての勾配が有限であればオプティマイザがステップを進め、そのようなステップが一定回数続くと S を 2 倍にする。いずれかが `inf` または NaN であれば、そのステップをスキップして S を半分にする。

`torch.amp.GradScaler` は fp16 の `autocast` と組み合わせて動的損失スケーリングを自動化する。

```
net = mlp()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
scaler = torch.amp.GradScaler('cpu')
for X, y in batches[:20]:
    with torch.autocast('cpu', dtype=torch.float16):
        loss = F.cross_entropy(net(X), y)
    optimizer.zero_grad()
    scaler.scale(loss).backward()  # Gradients of S * loss
    scaler.step(optimizer)         # Divide by S, skip the step on inf/NaN
    scaler.update()                # Adjust S
print(f'loss {loss.item():.3f}, loss scale {scaler.get_scale():.0f}')
```

```
loss 2.030, loss scale 65536
```

したがって、bf16 をネイティブサポートする最新のハードウェア（近年の GPU、TPU、サーバー CPU）では、損失スケーリングなしの bf16 混合精度が標準的な手法となっており、動的損失スケーリングを伴う fp16 は旧世代の GPU において引き続き不可欠である。

最後に、テンソルの dtype を一切変更せずに fp32 モデルを高速化するハードウェア機能を 1 つ紹介しておく。NVIDIA の Ampere アーキテクチャ以降では、fp32 の行列積をテンソルコア上の **tf32** モードで実行できる。入力は fp32 の 8 ビット指数部を保ったまま fp16 と同じ 10 ビット仮数部に切り詰められ、内積は完全な fp32 で累積されるため、厳密な fp32 と有効数字 3〜4 桁まで一致しつつ大幅な高速化が得られる。

PyTorch のデフォルトは厳密な fp32 の行列積であり、`torch.set_float32_matmul_precision('high')`（5.7 節）によって tf32 が有効になる。

5.5.4 数値計算の失敗モード

16 ビットのコードにおける数値バグの大半は、3 つの具体的な落とし穴で説明できる。上流で起きたオーバーフローが下流で NaN として表面化すること、シフトしていない指数関数が最後の対数で縮められる前に発散すること、そして長いリダクションにおいて個々の項がアキュムレータの丸め閾値を下回った途端に和が増えなくなることである。

5.5.4.1 オーバーフローから NaN へ

損失が NaN になること自体は根本原因ではない場合が多い。たいていは、それより前に起きた `inf` へのオーバーフローの後遺症である。`inf - inf` や `inf / inf` のような不定形の演算が NaN を生み出すためである。

```
s = torch.tensor(60000., dtype=torch.float16) * 2  # Overflows
s, s - s
```

```
(tensor(inf, dtype=torch.float16), tensor(nan, dtype=torch.float16))
```

最初のオーバーフローは NaN が損失に到達するより何層も上流で起きていることがあり、一度でも NaN を含むパラメータ更新が行われると、それ以降のすべてのステップが汚染される。損失が NaN になったときは、学習率やアーキテクチャを疑う前に、まず値の範囲を確認すべきである（fp16 で実行されている演算が上限の 65504 付近に達していないかなど）。

5.5.4.2 Log-Sum-Exp

ソフトマックス交差エントロピー（cross-entropy）では $\log \sum_i \exp(x_i)$ を計算する必要がある。この式を文字どおり評価すると、最終的な答えが小さい場合でさえ fp16 ではオーバーフローする。

```
x = torch.tensor([12.0, 11.0, 10.0], dtype=torch.float16)
x.exp().sum().log(), torch.logsumexp(x, dim=0)
```

```
(tensor(inf, dtype=torch.float16), tensor(12.4062, dtype=torch.float16))
```

真の値は 12.4 という穏やかな大きさであるにもかかわらず、中間項 $e^{12} \approx 163{,}000$ が fp16 の上限である 65504 を軽々と超えてしまう。解決策は 3.3.5.2 節で導入した厳密なシフトであり、指数をとる前に最大要素 $m = \max_i x_i$ を引き、あとから m を足し戻せばよい。

$$\log \sum_i \exp(x_i) = m + \log \sum_i \exp(x_i - m). \tag{5.9}$$

右辺の指数はすべて非正となるためオーバーフローは起こり得ず、最大の項は厳密に $e^0 = 1$ となるため $\log 0$ へのアンダーフローも防げる。実際にはこのシフトを自分で実装する必要はほとんどなく、組み込みの log-softmax や交差エントロピーのルーチンが自動的に適用してくれる。

5.5.4.3 fp32 での累積

16 ビットで長い総和を累積すると、気づかないうちに計算が停滞する。途中の累積和が十分に大きくなると、足し込む小さな各項が局所的な格子間隔の半分を下回り、0 に丸められてしまうからである。以下では、（fp16 で**厳密に**表現できる）2^{-10} を 4096 回、fp16 のアキュムレータと fp32 のアキュムレータにそれぞれ加算して比較する。

```
total16, total32 = np.float16(0), np.float32(0)
term = np.float16(2**-10)
for _ in range(4096):
    total16 += term  # fp16 accumulator
    total32 += term  # fp32 accumulator, same fp16 terms
total16, total32
```

```
(np.float16(2.0), np.float32(4.0))
```

厳密な合計は 4 であり、fp32 のアキュムレータはピタリとこの値に一致する。一方、fp16 のアキュムレータは 2048 ステップ後に 2 で凍りつく。2 と 4 の間では隣り合う fp16 の値の間隔が 2^{-9} であるため、$2 + 2^{-10}$ はちょうど中点に落ち、最近傍偶数丸め（round-to-nearest-even）によって永久に 2 へと丸め戻されるからである。バッチ平均、エポック全体の損失合計、正規化層（normalization layer）における分散計算はすべてこの落とし穴を抱えている。混合精度のポリシーがリダクションと正規化を fp32 に保つ理由も、16 ビットテンソルに対するカスタムのリダクションを fp32 で累積すべき理由もここにある。

これら 3 つの失敗モードはすべて同じ根源に行き着く。最終的な答えが余裕をもって収まる場合でも、中間量が 16 ビット形式の範囲や精度を超えてしまうことである。また、混合精度と tf32 は設計上、演算の最下位ビット群を変化させるため、純粋な fp32 とビット単位で一致することは決してない点も覚えておこう。実行結果をいつビット単位で再現**できる**のか、そしてそれをどう実現するかについては 5.8 節で扱う。

5.5.5 まとめ

浮動小数点形式はダイナミックレンジ（指数ビット）と精度（仮数ビット）のトレードオフの上にある。fp16 は局所的な刻み幅が細かいものの表現範囲が狭く（65504 を超えるとオーバーフローし、微小な勾配はアンダーフローする）、一方で bf16 は精度が 10 進で 2〜3 桁にとどまる代わりに fp32 の 8 ビット指数部をすべて維持する。重みを直接 16 ビットで保存すると小さな勾配更新が丸めで消えて訓練が停滞するため、混合精度訓練では fp32 のマスター重みを保持しつつ行列積の実行と活性化値の保存を 16 ビットで行う。これにより、純粋な bf16 重みが遅れをとる Fashion-MNIST の MLP でも fp32 と同等の正解率を達成できる。fp16 では動的損失スケーリングによって誤差逆伝播法の間、勾配を非正規化数・アンダーフロー領域の外へ引き上げるが、bf16 ではこの手順自体が不要となる。最後に、オーバーフロー起因の NaN から低精度コードを守るには、シフト付きの log-sum-exp 演算を用い、長いリダクションや正規化には fp32 のアキュムレータを使えばよい。

5.5.6 演習問題

1. **混合精度のメモリ**。fp32 で Adam を用いて訓練すると、パラメータあたり 16 バイトが保存される（5.2 節）。`train` が SGD の代わりに Adam を使うと仮定する。
   1. 使用しているフレームワークの混合精度の手法では、ステップ間において重み、勾配、および Adam の移動平均はそれぞれどの dtype で保存されるか。混合精度によってパラメータあたりのバイト数は減少するか。
   2. 各線形層の入力とロジットを保存対象の活性化値として数えるものとする。`train` の MLP について、1 例あたりの要素数を fp32 の場合と `train` が用いる 16 ビット形式の場合のバイト数へそれぞれ換算せよ。どのバッチサイズで、活性化メモリの削減量がこの MLP の永続的な訓練状態のサイズと等しくなるか。
2. **オーバーフロー時の損失スケーリング**。動的損失スケーリング（5.5.3.3 節）のもとでは、スケール解除後の勾配に `inf` または NaN を含むステップはスキップされ、スケールが半分にされる。正常なステップが N 回連続すると、スケールは 2 倍にされる。
   1. スキップされたステップでは、Adam の移動平均を一切変更してはならない。仮に非有限の勾配でそのまま更新した場合に何が問題になるか、また非有限の要素を 0 に置き換えて更新した場合に何が問題になるかをそれぞれ説明せよ。
   2. スケールを縮小するだけでなく、正常なステップが続いた後に再び増大させなければならない理由を説明せよ。
   3. スケールがある固定の閾値を超えると毎ステップ必ずオーバーフローし、それを下回ると一切オーバーフローしないと仮定する。スケールが落ち着いた後、スキップされるステップの割合を N の関数として求めよ。固定閾値の仮定を置かない場合、N を大きくするとどのようなコストが生じるか。
3. [code] **fp16 での正規化**。混合精度の手法では正規化層を fp32 で実行する。その理由を見るため、ゲインを 1 に設定した RMSNorm を考える（5.4 節でモジュールとして実装した）。これは入力行 $\mathbf{x}$（幅 d）をその二乗平均平方根で再スケーリングする。

$$\frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}}, \tag{5.10}$$

   ここで、小さな定数 ϵ は 0 除算を防ぐためのものである。要素の標準偏差が約 100 である幅 $d = 1024$ の行 64 個からなるバッチに対し、この式をすべて fp16 で計算せよ。
   1. どの中間量が最初に `inf` になり、影響を受けた行に対してこの層は何を返すか。
   2. 2 乗の計算自体は厳密であるとする。入力の標準偏差がいくらになるとその平均が fp16 の最大値を超え、またいくらになると d 個の要素にわたる総和が fp16 の最大値を超えるか。
4. [code] **e4m3fn 形式**。●`torch.finfo(torch.float8_e4m3fn)` を出力し、その `max`、`tiny`、`eps` を fp16 および bf16 の値と比較せよ。また、`e4m3fn` という名前の意味を説明せよ。4 ビットの指数部と 3 ビットの仮数部を持ち IEEE の規約に従う形式であれば、すべて 1 の指数部を `inf` と NaN のために予約するはずである。その形式における最大の有限値を計算し、`e4m3fn` がどのようにして `max` として 448 を達成しているかを説明せよ。

5.6 保存、読み込み、事前学習済みの重み

訓練済みのニューラルネットワークは驚くほど脆いものであり、それを訓練したプロセスのメモリ上にしか存在しない。スクリプトが正常終了したか、エラーに遭遇したか、サーバーの電源コ

ードが抜けたか、あるいはプリエンプティブルなクラウドインスタンス上で実行されていたかにかかわらず、そのプロセスが終了した瞬間に、学習したすべての重みはディスクに保存していない限り消えてしまう。これまでの章で扱ったトイモデルであれば、途中経過を失っても数分を失うだけで済む。しかし現代の大規模な訓練では数週間分の計算が水泡に帰しかねず、しかもハードウェア障害は珍しい例外ではなく日常茶飯事である。実際、最大 16,384 基の GPU を用いて Llama 3 405B を事前学習（pre-training）した 54 日間のうちに、Meta は予期しない中断を 419 回、すなわち約 3 時間に 1 回の頻度で記録している (Grattafiori et al. 2024)。信頼できるチェックポイント保存（checkpointing）と再開の仕組みがなければ、この規模の実行は決して完走できないだろう。

重みの保存には、もう 1 つ同様に広く浸透した目的がある。それは訓練済みモデルを共有し、他者が一から訓練することなく推論（inference）を実行したりファインチューニングしたりできるようにすることである（32.2 節（英語版））。どちらのワークフローでも、実務上同じ疑問が生じる。ディスクには具体的に何を保存すべきか、どのファイル形式なら安全に読み込めるか、そして重みが対象のアーキテクチャと一致していることをどう検証すべきか。

何を保存すべきかという問いへの答えは、5.2 節から直接導かれる。モデルはコードと状態からなり、コードがアーキテクチャと順伝播（forward pass）を定義する一方、**状態**（state）は訓練によって生成される名前付きテンソル（パラメータとバッファの両方）の集まりである。シリアライズするのは名前をテンソルに対応付けるこの状態のみであり、モデルの再構築はコード上でアーキテクチャをインスタンス化して、そこにテンソルを読み込み直すことで行う。中断された訓練を再開するには、さらに多くの情報が必要になる。モデルの重みに加え、オプティマイザ状態（optimizer state、たとえば Adam の移動モーメントなど）、乱数生成器（random-number generator、RNG）の状態、ステップカウンタ、およびアーキテクチャ設定も保存しなければならず、そのため訓練チェックポイントは推論用の重みファイルよりも数倍大きくなる。

この実践的な節では、まずモデルの状態を確認してシリアライズし、次に危険性を伴う pickle ファイルと安全な `safetensors` 形式を対比する。続いて、ビット単位で完全に再開可能なアトミックな訓練チェックポイントを構築し、最後に事前学習済み ResNet-50 の重みを改変したアーキテクチャへ読み込みつつ、欠落キーと予期しないキーを検証する。

```
import json
import os
import struct
from collections import Counter
from dataclasses import asdict, dataclass
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from torchvision.models import resnet50, ResNet50_Weights
from safetensors.torch import load_file, save_file
```

5.6.1 モデルの状態

モデルは数十から数百ものテンソルを含むため、その保存には名前からテンソルへの対応付けを記録する必要がある。各フレームワークはモジュールツリー（module tree）を走査することでこの対応付けを自動生成し（5.1 節）、すべてのテンソルにルートからのパスをキーとして割り当てる。以下では、256 個の隠れユニットを持つ単純な単一隠れ層 MLP の状態を確認する。

`collect_params()` と `state_dict()` は、ドット区切りのパス（`hidden.weight`、`output.bias`）をパラメータやバッファに対応付ける。

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(2, 20)
Y = net(X)
{name: tuple(t.shape) for name, t in net.state_dict().items()}
```

```
{'hidden.weight': (256, 20),
 'hidden.bias': (256,),
 'output.weight': (10, 256),
 'output.bias': (10,)}
```

この状態辞書（state dictionary）には名前、形状、数値のみが記録されており、それ以外の情報は一切含まれない点に注目してほしい。`MLP` クラスや順伝播のロジック、活性化関数（activation function）の痕跡は残っていないのである。したがってモデルを復元するには、完全に一致する名前と形状を再構築する Python コードが必要になる。属性名を `output` から `head` に変更しただけでも、計算する関数は同一であるにもかかわらず対応付けは壊れてしまう。このような不一致を検出し診断する方法は、本節の後半で示す。

5.6.2 Pickle と safetensors

チェックポイントファイルが自身のディスク外に出ない限り、ほぼどのようなシリアライズ形式でも事足りる。しかし、公開ハブから事前学習済みの重みをダウンロードしたり、他者からチェックポイントを受け取ったりした瞬間、セキュリティ上の疑問が生じる。そのファイルを読み込むだけで、手元のマシン上で任意のコードが実行されうるのだろうか。残念ながら、Python のデフォルトのシリアライズ形式では、その答えは「イエス」である。

5.6.2.1 読み込み時のコード実行

Python 標準の `pickle` モジュールは、受動的なデータを保存しているわけではない。Python オブジェクトを再構築するバイトコードプログラムを保存しており、pickle の復元処理はそのプログラムを**実行する**ため、インポート可能な任意の関数を呼び出せてしまう。どのクラスも `__reduce__` を通じて呼び出し可能オブジェクトとその引数を返すことで、pickle が実行する内容をカスタマイズできる。以下の `Tripwire` の例は無害に `print` を呼び出すだけだが、悪意あるチェックポイントであれば同じくらい容易に `os.system` を呼び出せる。

```
class Tripwire:
    def __reduce__(self):  # pickle calls print(...) to rebuild the object
        return (print, ('*** payload executed while loading ***',))
```

`torch.save` は pickle アーカイブを書き出す。`weights_only=False`（PyTorch 2.6 より前のデフォルト）を指定して読み込むと、ペイロードが即座に実行される。

```
torch.save(Tripwire(), 'tripwire.pt')
_ = torch.load('tripwire.pt', weights_only=False)  # the payload runs here
```

```
*** payload executed while loading ***
```

デフォルトでは、各ローダは任意の Python オブジェクトを拒否する。

`torch.load` は PyTorch 2.6 以降、デフォルトで `weights_only=True` となっている。

```
try:
    torch.load('tripwire.pt')  # weights_only=True, the default
except Exception as e:
    print(type(e).__name__, str(e).split('.')[0])
```

```
UnpicklingError Weights only load failed
```

制限付き Unpickler による保護は許可リスト（allowlist）にバグがない範囲でしか機能せず、過去には強化された制限付きローダさえ回避する脆弱性が発見されている。実行可能な命令をそもそもエンコード**できない**ファイル形式であれば、設計上その攻撃対象領域を丸ごと排除できる。

PyTorch の `weights_only` モードは、こうした制限付きローダの一種である。

5.6.2.2 safetensors 形式

safetensors 形式は生のテンソルデータとメタデータのみを厳密に格納し、コードは一切含まない。そのバイナリレイアウトは連続した 3 つのセグメントだけで構成されている（Figure 5.12）。すなわち、ヘッダ長を表す 8 バイトのリトルエンディアン整数、各テンソル名をその dtype・形状・バイトオフセットに対応付ける UTF-8 の JSON ヘッダ、そして隙間なく詰め込まれた生のテンソルバッファである。読み込み時にコードは一切実行されず、リーダは JSON ヘッダを検証してすべてのバイト範囲がファイル内に収まっていることを確認するだけである。システム面のさらなる利点として、正確なバイトオフセットがあらかじめわかっているため、メモリマッピング（`mmap`）や、ファイル全体を読み込まずに個々のテンソルへランダムアクセスすることが可能になる。フレームワークに依存せず高速な safetensors は、Hugging Face Hub における標準の重み配布形式となっている。

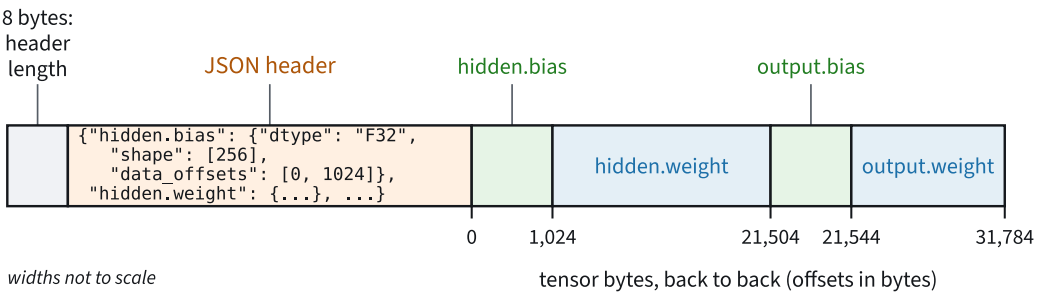


Figure 5.12: 後述する MLP の safetensors ファイルを 1 本のバイト列として示したもの。8 バイトの長さ、各テンソルの dtype・形状・バイト範囲を記した JSON ヘッダ、そして 4 つのテンソルの生バイト列が名前順に隙間なく並ぶ。データの下のオフセットは、ヘッダが記録している範囲を表す。

この MLP の状態を `mlp.safetensors` に保存し、新しく作成したクローンに読み込んで、両方のインスタンスが同一の出力を生成することを確認しよう。

```
save_file(net.state_dict(), 'mlp.safetensors')
clone = MLP()
clone.load_state_dict(load_file('mlp.safetensors'))
torch.equal(clone(X), Y)
```

```
True
```

保存と読み込みの往復変換はビット単位で完全に一致する。さらに都合がよいことに、ヘッダは固定オフセットにあるプレーンな JSON であるため、Python の標準ライブラリだけでファイルの中身を調べられる。

```
with open('mlp.safetensors', 'rb') as f:
    n = struct.unpack('<Q', f.read(8))[0]  # header length, little-endian
    header = json.loads(f.read(n))
print('header length:', n, 'bytes')
for name, entry in sorted(header.items(), key=lambda e: e[1]['data_offsets']):
    print(f"{name:18s} {entry['dtype']} {str(entry['shape']):10s} "
          f"bytes {entry['data_offsets']}")
```

```
header length: 296 bytes
hidden.bias        F32 [256]      bytes [0, 1024]
hidden.weight      F32 [256, 20]  bytes [1024, 21504]
output.bias        F32 [10]       bytes [21504, 21544]
output.weight      F32 [10, 256]  bytes [21544, 31784]
```

どのバイト区間も、テンソルの要素数に 4 バイト（`F32` = float32）を掛けた長さに等しい。256 要素の隠れ層バイアスが最初の 1,024 バイトを占め、$256 \times 20 = 5{,}120$ 要素の隠れ層重みが続く 20,480 バイトを占め、各ブロックは直前のブロックが終わった位置からすぐ始まる。隠れたコードを一切含まない、純粋なテンソルデータである。

手元のマシン上で一時的に使う個人用のチェックポイントであれば、フレームワーク標準の形式でまったく問題ない。しかし、ユーザー向けに公開したりウェブからダウンロードしたりするなど、重みが信頼境界（trust boundary）を越える場合には、safetensors のようなコードを含まない形式を用い、受け手が完全性を検証できるよう暗号学的チェックサムを併記すべきである。

5.6.3 訓練チェックポイント

単なる重みファイルは推論には十分だが、訓練の再開には**不十分**である。その理由は、状態を持つ最適化にある。現代の深層学習の実行の多くは **Adam**（9.5 節）を用いており、パラメータごとに 2 つの移動平均を追跡する（5.2 節）。すなわち、$\mathbf{m}$（勾配 $\mathbf{g}$ 用）と $\mathbf{v}$（要素ごとの二乗 $\mathbf{g} \odot \mathbf{g}$ 用）であり、直近の勾配が大きかった箇所では更新幅を抑えつつ、平滑化された勾配と逆方向へ各パラメータを次の式で更新する。

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \, \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}. \tag{5.11}$$

ここで、η は学習率（learning rate）、ϵ は小さな定数（通常は 10^{-8}）であり、除算と平方根は要素ごとに作用し、ハット記号はゼロ初期化を補うバイアス補正（bias correction）項を表す（導

出は 9.5 節を参照)。重要なのは、**m** と **v** が最適化の履歴を保持している点である。これらを破棄すれば、Adam はあたかも訓練が一から始まったかのように振る舞う。

したがって、真に再開可能な訓練チェックポイントには 5 つの異なる要素が詰め込まれる(Figure 5.13)。すなわち、(1) モデル状態(パラメータとバッファ)、(2) オプティマイザ状態(**m** と **v**)、(3) ミニバッチのシャッフルやドロップアウトを切れ目なく継続するための乱数生成器の状態、(4) Adam のバイアス補正や学習率スケジュール(learning-rate schedule)を制御するステップカウンタ、そして (5) アーキテクチャの再構築に必要なモデル設定である。勾配は次のステップですぐに再計算されるため省略される。結果として、fp32 での Adam 訓練中に保持されるパラメータあたり 16 バイトのうち 12 バイト(重みと 2 つのモーメント)が残る(5.2 節)。そのため、80 億パラメータのモデルに対する再開可能なチェックポイントは 96 GB に達し、これは 32 GB の fp32 重みの 3 倍、16 GB の bf16 推論用ファイルの 6 倍にあたる(5.5 節)。

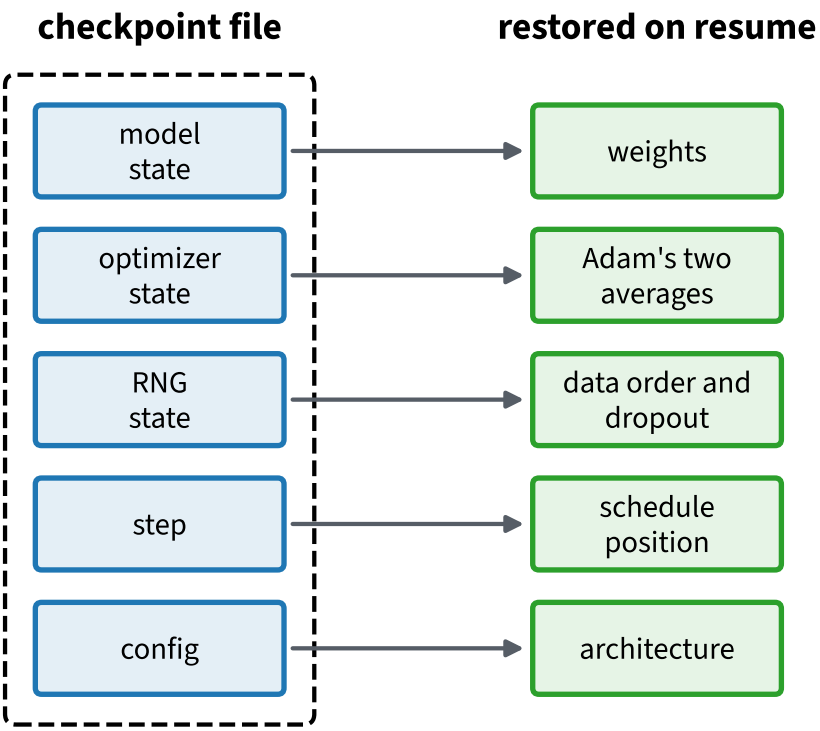


Figure 5.13: チェックポイントファイルの 5 つの区画と、再開時にそれぞれが復元する対象を矢印で対応付けた図。モデル状態は重み、オプティマイザ状態は Adam の 2 つの移動平均、RNG 状態はデータ順序とドロップアウト、ステップはスケジュール内の位置、設定はアーキテクチャを復元する。

チェックポイント保存は**クラッシュセーフ**でもなければならない。既存のチェックポイントをその場で上書きすると、書き込み途中でジョブが異常終了した際に新旧両方のバージョンを失う危険がある。そこで代わりに一時ファイルへ書き出し、完了後に保存先へアトミックにリネームすることで、どのリーダから見ても完全な前回のチェックポイントか完全な新しいチェックポイントのいずれか一方だけが見えることを保証する。このリネームによりプロセスのクラッシュからは保護される。電源喪失にまで耐えるにはファイルとそのディレクトリの両方に対する `os.fsync` も必要になるが、これについては演習問題で扱う。

モデル状態、オプティマイザ状態、ステップ、`asdict(cfg)`、および CPU・CUDA・`torch.Generator` の RNG 状態を、`weights_only=True` と互換性のある純粋なテンソルとプリミティブ型のみの辞書にまとめ、`.tmp` ファイルと `os.replace` を介して保存する。

```
def save_checkpoint(path, model, optimizer, step, cfg=None, rng=None):
    """Atomically write a resumable training checkpoint."""
    ckpt = {'model': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'step': step,
            'cpu_rng': torch.get_rng_state()}
    if cfg is not None:
        ckpt['cfg'] = asdict(cfg)
    if rng is not None:                          # a torch.Generator
        ckpt['rng'] = rng.get_state()
    if torch.cuda.is_available():
        ckpt['cuda_rng'] = torch.cuda.get_rng_state_all()
    tmp = path + '.tmp'
    torch.save(ckpt, tmp)
    os.replace(tmp, path)

def load_checkpoint(path, model, optimizer=None, rng=None):
    """Restore the state written by save_checkpoint; return the raw dict."""
    ckpt = torch.load(path, weights_only=True)
    model.load_state_dict(ckpt['model'])
    if optimizer is not None:
        optimizer.load_state_dict(ckpt['optimizer'])
    if rng is not None:
        rng.set_state(ckpt['rng'])
    torch.set_rng_state(ckpt['cpu_rng'])
    if torch.cuda.is_available() and 'cuda_rng' in ckpt:
        torch.cuda.set_rng_state_all(ckpt['cuda_rng'])
    return ckpt
```

チェックポイントからの再開をテストするため、64 個の隠れユニットを持つ小さな MLP 回帰モデルを Adam（$\eta = 0.05$）により 20 次元のノイズ付き線形データ 256 例で訓練する。その際、シードを 0 に固定した専用の乱数生成器を使って 64 例ずつのミニバッチを抽出し、ミニバッチの順序もチェックポイントの状態に含まれるようにする。5.1 節と同様、`Config` データクラスにアーキテクチャと訓練のすべてのハイパーパラメータをまとめ、チェックポイント内部に直接同梱できるようにする。

```
@dataclass
class Config:
    num_inputs: int = 20
    num_hiddens: int = 64
    num_outputs: int = 1
    batch_size: int = 64
    lr: float = 0.05
```

まず 200 ステップ訓練してチェックポイントを書き出し、そのまま同じ実行をさらに 5 ステップ続けて、中断なしの 205 ステップ分の正解を作成する。`train` の各呼び出しは、各更新ステップの直前にミニバッチ損失を記録する。

```
def build(cfg):
    net = nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                        nn.ReLU(),
                        nn.Linear(cfg.num_hiddens, cfg.num_outputs))
    for layer in (net[0], net[2]):
        nn.init.xavier_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
    return net

cfg = Config()
data = torch.randn(256, cfg.num_inputs)
target = data @ torch.randn(cfg.num_inputs, 1) + 0.1 * torch.randn(256, 1)
loss = nn.MSELoss()

def train(model, opt, rng, num_steps):
    losses = []
    for _ in range(num_steps):
        idx = torch.randint(len(data), (cfg.batch_size,), generator=rng)
        opt.zero_grad()
        l = loss(model(data[idx]), target[idx])
        l.backward()
        opt.step()
        losses.append(l.detach())
    return [l.item() for l in losses]

net = build(cfg)
opt = torch.optim.Adam(net.parameters(), lr=cfg.lr)
rng = torch.Generator().manual_seed(0)      # draws the minibatches
train(net, opt, rng, 200)
path = 'run.pt'
save_checkpoint(path, net, opt, step=200, cfg=cfg, rng=rng)
uninterrupted = train(net, opt, rng, 5)
```

ここで、新しいプロセスでの再起動をシミュレートしよう。resume は保存された設定からモデルとオプティマイザを再構築し、重みとあわせてオプティマイザ状態や RNG 状態を任意で復元したうえで、num_steps に達するまで残りのステップを実行する。また、max_diff は 2 つのモデル間における要素ごとの重みの最大誤差を返す。

```
def resume(path, num_steps, restore_optimizer=True, restore_rng=True):
    cfg = Config(**torch.load(path, weights_only=True)['cfg'])  # saved config
    model = build(cfg)
    opt = torch.optim.Adam(model.parameters(), lr=cfg.lr)
    rng = torch.Generator().manual_seed(0)  # seeded as at the start
    ckpt = load_checkpoint(path, model, opt if restore_optimizer else None,
                           rng=rng if restore_rng else None)
    return model, train(model, opt, rng, num_steps - ckpt['step'])

def max_diff(a, b):
    return max((p - q).abs().max().item()
               for p, q in zip(a.parameters(), b.parameters()))
```

完全なチェックポイントを復元した場合と、乱数生成器だけを省いた場合（restore_rng=False、

スクリプトが単にシード `0` を再設定したかのような状態）とを比較し、中断なしで 205 ステップ実行した `net` に対する重みの最大誤差を測定する。

```
for name, kwargs in (('full checkpoint', {}),
                     ('without the generator', {'restore_rng': False})):
    resumed, _ = resume(path, 205, **kwargs)
    print(f'{name:22s} max |difference| {max_diff(resumed, net):.3g}')
```

```
full checkpoint        max |difference| 0
without the generator  max |difference| 0.00397
```

完全なチェックポイントを復元すると、中断なしの重みが誤差ゼロ（ビット単位の一致）で再現される。一方、RNG 状態を省くと、再開後のループは次のミニバッチを抽出する代わりに最初のミニバッチを再生してしまい、重みに約 0.004〜0.02 の差が生じる（なお、ビット単位の完全一致には決定論的な演算も前提となるが、これについては 5.8 節で詳しく調べる）。

オプティマイザ状態の破棄ははるかに破壊的である。以下では、保存された重みと RNG 状態はそのまま引き継ぎつつ、新しく構築した Adam オプティマイザを使って再開し、再開後 5 ステップのミニバッチ損失を並べて比較する。

```
_, full = resume(path, 205)
_, fresh = resume(path, 205, restore_optimizer=False)
print('saved optimizer:', [round(l, 4) for l in full])
print('fresh optimizer:', [round(l, 4) for l in fresh])
```

```
saved optimizer: [0.0009, 0.0017, 0.0022, 0.001, 0.0019]
fresh optimizer: [0.0009, 2.1294, 0.6979, 0.7146, 0.9511]
```

保存されたオプティマイザ状態を用いれば、訓練は滑らかに再開し、損失は 0.01 前後かそれ以下で推移する。ところが新規の Adam オプティマイザを用いると、**たった 1** ステップで損失が約 1,000 倍に跳ね上がり、4 ステップ後でも再開前の 100 倍以上の水準にとどまってしまう。

なぜ新規の Adam を 1 ステップ実行しただけで、ほぼ収束したモデルが壊れてしまうのだろうか。Equation 5.11 がその理由をすべて物語っている。$\mathbf{m} = \mathbf{0}$ と $\mathbf{v} = \mathbf{0}$ から出発すると、ステップ 1 における Adam のバイアス補正が初期化の減衰因子をちょうど打ち消すため、$\hat{\mathbf{m}} = \mathbf{g}$ および $\hat{\mathbf{v}} = \mathbf{g} \odot \mathbf{g}$ となる。したがって、非ゼロの勾配を持つすべてのパラメータは、勾配がどれほど微小であっても $\eta\,|g_i|/(|g_i| + \epsilon) \approx \eta = 0.05$、すなわち学習率そのものの大きさで移動してしまう。対照的に、保存された $\mathbf{v}$ は訓練序盤のはるかに大きな二乗勾配を記憶しているため、訓練終盤の微小な勾配に対して $\sqrt{\hat{\mathbf{v}}}$ が大きく保たれ、更新は穏やかなまま維持される。モデルが収束（convergence）に近づいているほど、未初期化の Adam による再開の衝撃は激しくなる。重みだけでは、訓練の再開には決して十分ではないのである。

5.6.4 事前学習済みの重み

ここからは、保存された状態のもう 1 つの主要な用途である、他者が訓練した重みからの初期化に目を向けよう。ImageNet（1,000 カテゴリにわたる 120 万枚の画像）で訓練されたバックボーンは、新しいタスクへきれいに転移できる豊かな視覚表現を学習している。これを 10 クラスの分類問題に適応させるのは簡単で、最終の分類器**ヘッド**を除くすべての層を読み込み、新しい 10 出力の線形層に差し替えるだけでよい。以下では ResNet-50（約 2500 万個のパラメータ、1,000 クラスのヘッドを除くと 2350 万個）を使ってこれを実演する。畳み込みネットワークについては第 6 章と第 7 章で学ぶが、重みの読み込みという観点では、ResNet-50 は単に数百個の名前付きテンソルからなる状態辞書にすぎない。

```
net = resnet50(weights=ResNet50_Weights.DEFAULT)  # ~100 MB once
net.fc = nn.Linear(net.fc.in_features, 10)        # new 10-class head
net.fc
```

```
Linear(in_features=2048, out_features=10, bias=True)
```

外部の重みファイルが独自の対象モデルと 1 対 1 で一致することはまれであり、不一致を黙って無視するのは悪名高いバグの温床である。慎重なローダが部分的な転移をどのように監査するかを見るため、事前学習済み ResNet-50 の状態から 2 つのテンソル群を取り除く。1 つは置き換えを**意図して**いる分類ヘッド、もう 1 つは破損または不一致のチェックポイントを模したバックボーンの最終ステージ全体である。残りのテンソルを新規の 10 クラス ResNet-50 に読み込むと、2 つの診断用集合を確認できる。すなわち、**欠落**キー（missing keys、ファイルに存在せずランダム初期化のまま残されるモデルパラメータ）と、**予期しない**キー（unexpected keys、モデル側で使われないファイル内のエントリ）である。

`load_state_dict(..., strict=False)` は一致するキーを読み込み、`(missing_keys, unexpected_keys)` を直接返す。

```
damaged = {k: v for k, v in net.state_dict().items()
           if not k.startswith(('fc.', 'layer4.'))}
target = resnet50(weights=None)
target.fc = nn.Linear(target.fc.in_features, 10)
report = target.load_state_dict(damaged, strict=False)
print('missing by block:', dict(Counter(
    k.split('.')[0] for k in report.missing_keys)))
print('unexpected:', report.unexpected_keys)
```

```
missing by block: {'layer4': 50, 'fc': 2}
unexpected: []
```

新しい 10 クラス用ヘッドの重みとバイアスに対する欠落キーは、そのヘッドがランダム初期化から始まるため想定どおりである。一方、バックボーン最終ステージの欠落キーは危険信号である。もし部分一致のまま漫然と読み込んでいれば、「事前学習済み」ResNet-50 の大部分が人知れずランダムノイズのまま残るところだった。今回の例では予期しないキーの集合は空だが、これが空でない場合は通常、モジュール名の変更によって重みがまったく読み込まれなかったことを示唆する。本番コードでは、欠落キーと予期しないキーの集合が置き換えや削除を意図した層**のみ**と一致することを必ず明示的にアサートし、それ以外が含まれていればエラーを送出すべきである。

検証が済めば、転移学習（transfer learning）では通常、まず事前学習済みバックボーンを**凍結**（freezing）して（5.2.3.2 節）初期の更新で新しいヘッドのみを訓練し、その後より深い層の凍結を選択的に解除していく（32.2 節（英語版））。

本章でここまで扱ってきたものはすべて、ホスト CPU メモリ上に置かれていた。続く 5.7 節では、状態と計算の両方を GPU 上へ移す。

5.6.5 まとめ

モデルの永続化とはその**状態**（モジュールツリーのパスをパラメータやバッファのテンソルに対応付ける辞書）を保存することであり、アーキテクチャ自体はコード内に存在する。Python の `pickle` は読み込み時に任意のコードを実行するため、信頼境界を越えて共有する重みには `safetensors`（8 バイトの長さプレフィックス、JSON ヘッダ、および隙間なく詰め込まれたテンソルバイト列）のようなコードを含まない形式を用いるべきである。クラッシュセーフな訓練チ

ェックポイントは一時ファイルを介してアトミックに書き出され、モデル状態、オプティマイザ状態、RNG 状態、ステップカウンタ、設定という 5 つの要素を保持する。5 つすべてを復元すれば訓練はビット単位で再開されるが、RNG 状態を省くとデータストリームがドリフトし、Adam のモーメントを省くと再開後の最初のステップですべてのパラメータが学習率そのものの大きさで移動して損失が何桁も急騰する。事前学習済みの重みを改変したアーキテクチャへ読み込む際は、不一致を黙って無視するのではなく、欠落キーと予期しないキーの集合を自身の想定と照らし合わせて必ず監査すべきである。

5.6.6 演習問題

1. [code] **Safetensors ヘッダ**。MLP の safetensors ファイルにおけるヘッダ長は、ファイル先頭にある 8 バイトのリトルエンディアン整数である（Figure 5.12）。

    1. 隠れ層の幅を 2 倍にしたとき、および 3 番目の層を追加したときに、その長さがどう変化するかを予測したうえで実測せよ。ヘッダのどの部分がテンソルのサイズに依存するかを説明せよ。
    2. safetensors ライブラリを使わず、ヘッダとファイルのシーク操作のみを用いてファイルから隠れ層のバイアスを読み取れ。ヘッダに記録された dtype と形状を使ってバイト列をデコードし、その結果が `load_file` の返すテンソルと一致することを確かめよ。
    3. pickle ベースのチェックポイントでは、オブジェクトグラフ全体をデシリアライズせずに同様の単一テンソル読み取りを行えない理由を説明せよ。

2. [code] **部分的な読み込み**。`MLP2` を、`MLP` の出力層の直前に幅 256 の 2 番目の隠れ層 `hidden2` を挿入し、属性 `output` を `head` に改名したコピーとして定義せよ。

    1. `mlp.safetensors` を `MLP2` に `load_state_dict(..., strict=False)` で読み込め。返されたレポートを確認し、各欠落キーと各予期しないキーについて説明せよ。
    2. 読み込む前に、ファイルのキーを `MLP2` が使う名前に変更せよ。`hidden2` のみが新規の初期化を保っていることを確認し、予期しないキーや `hidden2` 以外の欠落キーがあれば読み込み時に例外を送出するようにせよ。

    Matthew Inkawhich「Saving and Loading Models」内の「Warmstarting Model Using Parameters from a Different Model」を改変。

3. [code] **dtype の往復変換**。●`float16` の数値は指数部に 5 ビット、仮数部に 10 ビットを格納するのに対し、`float32` ではそれぞれ 8 ビットと 23 ビットを格納する（5.5 節）。`MLP` のパラメータを `float16` にキャストして保存し、それを `float32` の `MLP` に読み込み直せ。

    1. 仮数部のビット幅から、復元されたパラメータの最大相対誤差を予測せよ。復元後の値と元の値のあいだで、最大絶対誤差、および非ゼロのパラメータ全体にわたる最大相対誤差を測定せよ。相対誤差が予測を上回るパラメータがあれば、その理由を説明せよ。
    2. 元のモデルと復元された `MLP` の `X` に対する出力を比較し、この往復変換が推論において許容できるかどうかを判断せよ。
    3. Adam は二乗勾配の移動平均を保持する。`float16` が勾配要素 10^{-4} の二乗をどのような値として格納するかを計算し、再開可能なチェックポイントがオプティマイザ状態を `float32` で保持する理由を説明せよ。

4. [extended] **クラッシュセーフなマニフェスト**。●

    1. 新しいステップ番号のプレフィックスのもとでモデル状態、オプティマイザ状態、JSON メタデータファイルを書き出し、各ファイルの名前・サイズ・チェックサムを

記録した 1 つのマニフェストを所定の位置へリネームすることでこれら 3 つを公開するユーティリティを作成せよ。

2. 2 種類の障害をシミュレートせよ。1 つ目は 3 つのファイルを書き出した後、マニフェストをリネームする前のクラッシュであり、2 つ目は公開済みチェックポイントに含まれる 1 ファイルの切り詰め（破損）である。再読み込みを行うと、1 つ目のケースでは前回の完全なチェックポイントから再開し、2 つ目のケースでは破損したチェックポイントを拒否することを示せ。
3. リネームはアトミックであるため、リーダからは古いマニフェストか新しいマニフェストのいずれか一方だけが見える。しかし POSIX は、リネームされたファイルのデータがリネーム自体より先にディスクへ到達することを保証していない。`os.fsync` は、ファイルのバッファリングされた書き込みやディレクトリのエントリを強制的にディスクへ書き出す。電源喪失が起きても前回のチェックポイントか新しいチェックポイントのいずれかが無傷で残るよう、このユーティリティのどこで `os.fsync` を呼び出すべきかを述べ、各呼び出しの役割を説明せよ。

5.7 GPU、デバイス、メモリ

2.4 節以降、`Trainer` は `num_gpus` 引数を受け取りつつも、それを黙って無視してきた。これまでの章で扱った小さなモデルでは、CPU での訓練が数秒で終わるため、それでもほとんど問題にならなかった。しかし、モデルが大規模になると、デバイス配置（device placement）が数秒で済むか数時間かかるかの違いを生む。現代のデスクトップ GPU は 2 つの 4096×4096 行列の積を 2 ms 未満で計算するが、CPU では 0.1 秒以上かかるからである（5.7.2.1 節でベンチマークする）。この圧倒的な計算速度には、厳しいハードウェア上の制約が伴う。GPU は専用のメモリ（たとえばデスクトップ向けカードなら 24 GB）を備えており、GPU が演算するすべてのテンソルはそのメモリ上に置かれていなければならない。

そもそも目的のモデルはメモリに収まるのだろうか。ここまで例に挙げてきた 80 億パラメータのモデルは、bf16 の重みに 16 GB を必要とするため、推論であれば余裕をもって収まる。しかし、このモデルを fp32 の Adam で訓練するには、活性化値（activation）を 1 つ保存する前でも 128 GB の静的な状態が必要となり（5.2 節）、単一の 24 GB や 80 GB のカードの容量をはるかに超えてしまう。モデルがデバイスに収まるかどうか、そして GPU が休みなく計算を続けられるかデータ待ちで停止してしまうかは、すべてのテンソルがどこに存在し、何バイトを消費するかに帰着する。

本節では、3 つの中心的な問いに答える。すなわち、テンソルとモデルをどのように CPU や GPU 上に配置し、デバイス間でコピーするか、GPU の非同期実行（asynchronous execution）はどのように動作するか（また GPU カーネルを正確にベンチマークするにはどうすればよいか）、そして訓練中に実際何が GPU メモリを占めるのか（余分な計算と引き換えにメモリ使用量を劇的に削減する**活性化チェックポイント**（activation checkpointing）の仕組みを含む）である。そのうえで、これらすべてを本書の残りの章で用いるデバイス対応の `Trainer` に組み込む。5.2 節と 5.5 節ではパラメータあたり・数値あたりのバイト数を数えたが、本節ではそれらのバイトを実際のハードウェア上に載せる（マルチ GPU 訓練は第 13 章に譲る）。すべてのコードセルは、GPU が利用できない環境でも CPU へスムーズにフォールバックし、実行時間の計測には GPU が必要であっても動作の正しさは確認できるようになっている。

```
import time
from torch.utils.checkpoint import checkpoint
```

```
torch.set_float32_matmul_precision('high')  # allow tf32 in fp32 matmuls
```

5.7.1 デバイスと配置

5.7.1.1 デバイスオブジェクト

すべてのテンソルは特定の**デバイス**上に存在する。すなわち、ホスト RAM と対になった CPU か、オンボードメモリと対になった個々の GPU のいずれかである。ターミナルで `nvidia-smi` を実行すると、マシン上のすべての NVIDIA GPU、その総メモリ量と使用中メモリ量、および実行中のプロセスが一覧表示される。デバイスの命名法はフレームワークごとに多少異なるものの、4 つのフレームワークはすべて 1 つの基本原則に従っている。それは、演算は単一のデバイス上で実行されるため、そのオペランドはすべて同じデバイス上に存在しなければならないという原則である。

`torch.device('cpu')` と `torch.device(f'cuda:{i}')` は、それぞれ CPU と GPU i を表す。

デバイスの取得処理を、本書の以降の章でインポートする 3 つの小さなヘルパ関数にまとめておこう。CPU 用のもの、GPU i 用のもの、そして利用可能な GPU の数を数えるものの 3 つである。

```
def cpu():
    """Get the CPU device."""
    return torch.device('cpu')

def gpu(i=0):
    """Get a GPU device."""
    return torch.device(f'cuda:{i}')

def num_gpus():
    """Get the number of available GPUs."""
    return torch.cuda.device_count()
```

デバイスオブジェクトの構築は常に成功し、ハードウェアが存在するかどうかはテンソルをその上に配置した時点で初めて確認される。

`num_gpus` を土台として、GPU があればそれを選択し、なければ CPU にフォールバックする 2 つのヘルパ関数を定義する。これにより、ノート PC でも GPU サーバーでも同じノートブックのコードを一切変更せずに実行できる。

```
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    devices = [gpu(i) for i in range(num_gpus())]
    return devices if devices else [cpu()]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

Apple シリコンの GPU（mps）や Google TPU などの他のアクセラレータも独自のデバイス識別子を用いるが、以下で述べる配置の原則はそれらすべてに等しく当てはまる。本書全体を通して、CPU と CUDA GPU に焦点を当てる。

5.7.1.2 デバイス上の配列

すべてのテンソルは、device 属性を通じて自身の配置先を公開している。

PyTorch のテンソルは、device= を渡さない限りデフォルトで CPU に配置される。

目的のデバイス上に直接メモリを確保すれば、CPU から GPU への余分なコピーを避けられる。以下では、X を 1 番目の GPU に、Y を 2 番目の GPU に配置し（GPU が 2 基未満の場合は CPU にフォールバックする）、両者の加算を試みる。

```
X = torch.ones(2, 3, device=try_gpu())
Y = torch.rand(2, 3, device=try_gpu(1))
try:
    print(X + Y)
except RuntimeError as e:
    print(str(e).split(', but')[0])
```

```
Expected all tensors to be on the same device
```

異なるデバイス上のテンソル同士を加算しようとすると即座にエラーが送出され、意図しないデバイス間転送を防ぐ。

5.7.1.3 デバイス間のコピー

デバイス間でテンソルを移動する処理は、デバイス上での計算よりはるかに遅いため（この差は 5.7.2.1 節で計測する）、異なるデバイス上のオペランドを暗黙的にコピーしてしまうと深刻な速度低下のバグが隠れてしまう。そこで、X を Y のデバイス上へ明示的にコピーしてから（Figure 5.14）加算する。

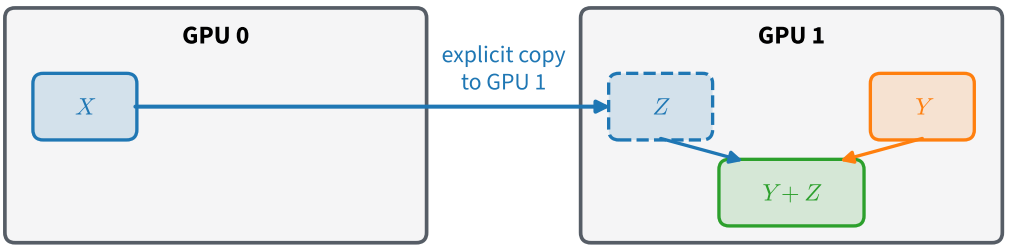


Figure 5.14: X は GPU 0 上に、Y は GPU 1 上に存在する。GPU 1 上に X のコピー Z（破線）を作ることで、Y + Z をすべて GPU 1 上で実行できる。

```
Z = X.to(try_gpu(1))
(Y + Z).device
```

```
device(type='cuda', index=1)
```

これで加算が実行され、結果も両オペランドと同じ 2 番目の GPU 上に置かれる。実践上の原則は単純である。計算の境界で一度だけデータを転送し（すでにそのデバイス上にあるテンソルの移動は実質的に何もしない操作である）、内側ループの演算はすべてその単一のデバイス上に留めることである。

5.7.1.4 デバイス上のモデル

順伝播（forward pass）では入力とモデルパラメータを組み合わせるため、モデル全体がミニバッチと同じデバイス上に存在しなければならない。5.1 節で見たモジュールツリーのおかげで、どのフレームワークでも 1 回の呼び出しですべてのパラメータを移動または初期化できる。

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
net(X)
```

```
tensor([[-0.2254],
        [-0.2254]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

入力とパラメータが同じデバイス上に揃っていれば、順伝播はそのデバイス上で実行され、出力も同じデバイス上に返される。

配置に関する 2 つの原則を心に留めておこう。1 つ目は、オプティマイザを構築する**前**にモデルを目的のデバイス上へ移動または初期化すること（これによりオプティマイザのモーメントも同じデバイス上に確保される）、2 つ目は、`forward` 内で新たに生成するテンソルをデフォルトの CPU ではなく `X.device` 上に確保することである。

5.7.2 非同期実行

5.7.2.1 実行キュー

GPU は、Python から演算が呼び出された瞬間にそれを同期的に実行するわけではない。その代わり、Python は各カーネル起動（kernel launch）をデバイスのコマンドキューに追加して即座に制御を返し、GPU がキュー内の処理を順番にこなしている間に CPU が後続の処理を準備できるようにする。このパイプライン化によって Python のオーバーヘッドは隠蔽されるが、時間計測においては重要な落とし穴がある。Python ループの所要時間を測っても、計測されるのは**実行**ではなく**キューへの投入**にかかる時間なのである。以下では、4096×4096 行列の積を 32 回キューに投入し、実時間（ウォールクロック時間）を 2 回確認する。Python がキューへの投入を終えた直後と、デバイスが計算を終えた直後の 2 回である。

```
if torch.cuda.is_available():
    A = torch.randn(4096, 4096, device=gpu())
    A @ A
    torch.cuda.synchronize()  # Warm up and wait
    t = time.perf_counter()
    for _ in range(32):
        B = A @ A
```

```
    print(f'time to queue 32 matrix products: {time.perf_counter() - t:.4f} sec')
    torch.cuda.synchronize()
    print(f'time until they all finished:     {time.perf_counter() - t:.4f} sec')
else:
    print('No CUDA device: operations on the CPU run synchronously.')
```

```
time to queue 32 matrix products: 0.0010 sec
time until they all finished:     0.0535 sec
```

GPU カーネルは非同期に実行される一方、CPU 演算は同期的に実行される。`torch.cuda.synchronize()` はキューに入ったすべての GPU 処理が完了するまで待機する。

Python は、実際の計算時間のほんの一瞬のうちに 32 個すべての行列積をキューに投入し終える。では、CPU が実際にブロックして待機するのはいつだろうか。それは、テンソルの表示、スカラーから Python の浮動小数点数への変換、NumPy への変換、あるいは `if` 文でのテンソル値に基づく分岐など、Python がホスト側で具体的な値を必要とするときだけである。このような**同期ポイント**（synchronization point）のたびに、GPU がその値までのキューを消化し終えるまで Python は停止し、その後 Python が追いつくまでの間、今度は GPU がアイドル状態になる（Figure 5.15）。ミニバッチごとに損失を Python の数値として記録すると、毎ステップ同期の空白時間（バブル）が生じてしまう。`Trainer` が `d2l.Averager` 内で損失をデバイス上に保持し、エポックごとに 1 回だけ Python の浮動小数点数に変換しているのはこのためである（2.4 節）。

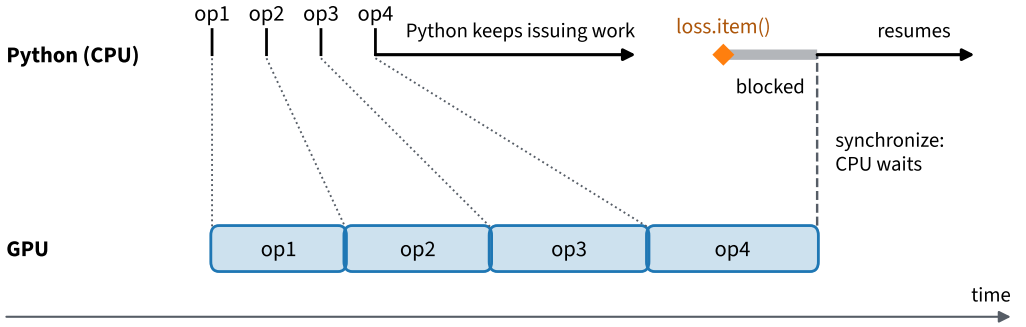


Figure 5.15: Python は演算 op1 から op4 までをキューに投入して先へ進み、その間に GPU がそれらを順番に実行する。損失を Python の数値として読み出す処理は同期ポイントとなる。GPU がキューを消化し終えるまで CPU はブロックし、その後 Python が再開する間、GPU は次の処理を受け取るまでアイドル状態になる。

非同期ディスパッチの仕組みから、信頼できる GPU ベンチマークのための 4 つの原則も導かれる。（1）初回限りのメモリ確保、ライブラリの初期化、JIT コンパイルを除外するため、最初にウォームアップ呼び出しを実行すること、（2）タイマの開始**直前**と停止**直前**の両方でデバイスを同期すること、（3）複数回の反復にわたって平均をとること、（4）サブミリ秒の分解能を得るために `time.perf_counter` を使用することである。（上の実演セルではウォームアップと同期は実施したものの、32 回の積の合計時間を報告していた。以下で定義する `Benchmark` ヘルパは、本章および以降の章のためにこれら 4 つの原則をすべて自動化する。）

```
class Benchmark:
    """Time a callable: warmup, then device-synchronized average seconds.
```

```
        `devices` lists the GPUs to wait for, by default the current one."""
        def __init__(self, f, warmup=3, repeats=10, desc='time', devices=None):
            def wait():  # CUDA runs asynchronously; the CPU does not
                if torch.cuda.is_available():
                    for device in devices or [None]:
                        torch.cuda.synchronize(device)
            self.desc = desc
            for _ in range(warmup):
                f()
            wait()
            t0 = time.perf_counter()
            for _ in range(repeats):
                f()
            wait()
            self.time = (time.perf_counter() - t0) / repeats

        def __repr__(self):
            return f'{self.desc}: {1000 * self.time:.2f} ms/call'
```

さっそく Benchmark を使って、先ほどの 2 つの主張を検証しよう。すなわち、大規模な行列積において GPU が CPU よりどれだけ高速か、そして計算に比べてホストからデバイスへの転送がいかに高コストかである。以下では、4096×4096 行列の積を CPU（fp32）、GPU の tf32（5.5 節）および 16 ビット精度で計測し、あわせて単一の 4096×4096 行列を CPU RAM から GPU へコピーする時間も計測する。

```
if torch.cuda.is_available():
    A_cpu = A.cpu()  # The same matrix in main memory
    A16 = A.bfloat16()
    print(Benchmark(lambda: A_cpu @ A_cpu, warmup=1, repeats=3,
                    desc='product on the CPU, fp32'))
    print(Benchmark(lambda: A @ A, desc='product on the GPU, tf32'))
    print(Benchmark(lambda: A16 @ A16, desc='product on the GPU, bf16'))
    print(Benchmark(lambda: A_cpu.to(gpu()),
                    desc='copy from CPU to the GPU'))
else:
    print('No CUDA device: there is nothing to compare the CPU with.')
```

```
product on the CPU, fp32: 1423.37 ms/call
product on the GPU, tf32: 1.68 ms/call
product on the GPU, bf16: 0.91 ms/call
copy from CPU to the GPU: 5.32 ms/call
```

現代の GPU では、4096×4096 の積は 2 ms 未満で完了するのに対し、CPU では約 100 ms から 2 s を要する（CPU のコア数に応じて約 $100\times$ から $1000\times$ の高速化である）。16 ビット精度に落とすと GPU の計算時間はさらにほぼ半減し、5.5 節で予告したテンソルコアによる高速化が実現される。

最後の行に特に注目してほしい。行列の積ではコピーされた要素 1 つあたり 4096 回の積和演算が実行されるにもかかわらず、CPU メモリから GPU へ行列を単に**コピーする**だけで、到着後に積を計算するよりも 3〜6 倍**長く**かかっている。通常のホストからのコピーでは、GPU の DMA エンジンが直接データを引き出せるよう、まず**ピン留め**（pinned、ページロック）ホストメモリにデータを中継する。バッチをあらかじめピン留めしておけば、この余分なコピーが不要になり、

データ転送と計算をオーバーラップさせることができる（13.2 節）。

5.7.3 GPU メモリ

5.7.3.1 メモリの 3 つのビュー

訓練スクリプトがどれだけの GPU メモリを使用しているかを尋ねると、ソフトウェアスタックのどの層に問い合わせるかによって 3 つの異なる数値が返ってくる。CUDA ドライバにメモリを直接要求する処理は遅いため、各フレームワークは内部で**キャッシングアロケータ**を動かし、ドライバから大きなメモリ領域（スラブ）をまとめて取得してテンソルへ細分化して割り当て、解放されたスラブも即座に返却せず再利用のために保持する。これにより、入れ子になった 3 つのメモリビューが生じる（Figure 5.16）。すなわち、生存中のテンソルが現在占有しているバイト数、フレームワークのアロケータが予約しているより大きなプール、そして CUDA ドライバがそのプロセスに割り当てた総メモリ量（`nvidia-smi` に表示される数値）である。

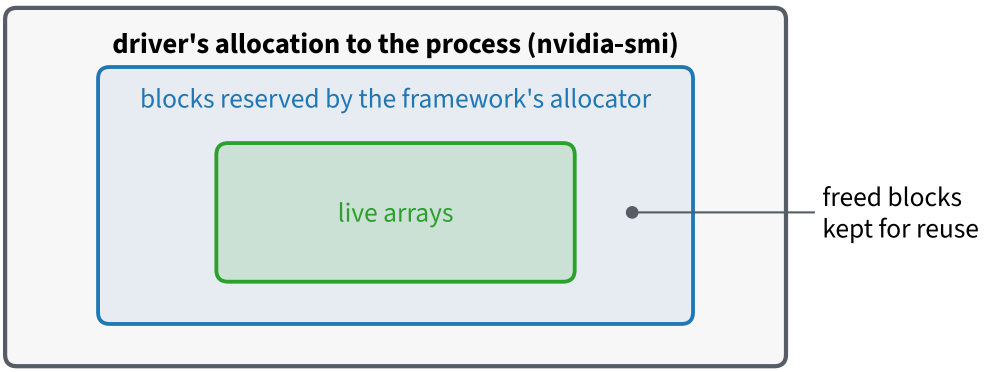


Figure 5.16: 3 つのメモリビューは入れ子になっている。nvidia-smi が報告するプロセスへのドライバ割り当ての中にアロケータの予約済みブロックが含まれ、さらにその中に生存中の配列が含まれる。

`torch.cuda.memory_allocated()` は生存中のテンソルのバイト数を報告し、`torch.cuda.memory_reserved()` はキャッシングアロケータのプール量を報告する。

1 GiB の float32 テンソルを確保して直ちに削除し、これらのカウンタがどう反応するか見てみよう。

```
if torch.cuda.is_available():
    def report(tag):
        print(f'{tag}: {torch.cuda.memory_allocated() / 2**20:7.1f} MiB '
              f'allocated, {torch.cuda.memory_reserved() / 2**20:7.1f} MiB '
              'reserved')
    big = torch.zeros(256, 1024, 1024, device=gpu())  # 1 GiB of float32
    report('after creating big')
    del big
    report('after deleting it ')
else:
    print('No CUDA device: both counters read 0 on the CPU.')
```

```
after creating big:  1192.1 MiB allocated,  1302.0 MiB reserved
after deleting it :   168.1 MiB allocated,  1302.0 MiB reserved
```

`big` を削除すると生存中のテンソル 1 GiB 分が解放されるが、フレームワークのキャッシングアロケータは解放された 1 GiB のスラブを予約プール内に保持し続けるため、`nvidia-smi` の表示はピクリとも動かない。（訓練ループの内部でスラブを強制的に CUDA ドライバへ返却してはならない。毎ステップ低速なドライバ割り当てが発生してしまうからである。）最も外側のドライバビューは、メモリ不足の絶対的な上限を定める。ドライバはカードが物理的に搭載している以上の RAM を貸し出せないためである。コード内の**どこに**メモリが使われているかを診断するには、生存バイト数のカウンタ、とりわけ呼び出し中に達した**ピーク**メモリを調べる。このピーク値が、ステップが成功するかメモリ不足（out-of-memory、OOM）でクラッシュするかを決定する。

`torch.cuda.reset_peak_memory_stats()` と `torch.cuda.max_memory_allocated()` を使えば、任意の呼び出しにおける生存メモリの正確なピーク増分を計測できる。

```
def peak_memory(f, *args):
    """Extra peak GPU memory allocated by f(*args), in bytes."""
    torch.cuda.synchronize()
    torch.cuda.reset_peak_memory_stats()
    base = torch.cuda.memory_allocated()
    f(*args)
    torch.cuda.synchronize()
    return torch.cuda.max_memory_allocated() - base
```

5.7.3.2 訓練時のメモリ

5.2 節では、fp32 の Adam 訓練における静的メモリの下限を $16N$ バイト（パラメータ数を N として、重みに $4N$、勾配に $4N$、Adam のモーメントに $8N$）と導出した。実際の訓練ステップでは、ここに 4 つ目の項として**活性化値**が加わる。誤差逆伝播法（backpropagation）では、順伝播の中間出力を逆伝播（backward pass）で消費するまで保持しなければならず、静的な $16N$ とは異なり、活性化メモリはミニバッチサイズと層の幅に比例して増大する。実測する前に計算してみよう。1024 次元の入力を受け取る幅 4096・隠れ層 3 層の MLP（パラメータ数は $N \approx 3.8 \times 10^7$、したがって重みは $4N \approx 144$ MiB）を、ミニバッチサイズ 4096 で動かすとする。3 つの隠れ層の出力はそれぞれ 64 MiB の 4096×4096 fp32 テンソル（合計 192 MiB）となり、この時点ですでに重み自体より 3 分の 1 大きい。

```
if torch.cuda.is_available():
    start = torch.cuda.memory_allocated()
    def allocated():  # Memory in use beyond that at the start of the cell
        return f'{(torch.cuda.memory_allocated() - start) / 2**20:7.1f} MiB'
    net = nn.Sequential(nn.Linear(1024, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Linear(4096, 10)).to(gpu())
    optimizer = torch.optim.Adam(net.parameters())
    X = torch.randn(4096, 1024, device=gpu())
    y = torch.randint(0, 10, (4096,), device=gpu())
    print('weights           ', allocated())
    loss = F.cross_entropy(net(X), y)
    print('+ activations     ', allocated())
    loss.backward()
```

```
        print('+ gradients       ', allocated())
        optimizer.step()
        print('+ optimizer state ', allocated())
    else:
        print('No CUDA device: run this cell on a GPU to see the four readings.')
```

```
weights              160.2 MiB
+ activations        353.4 MiB
+ gradients          313.6 MiB
+ optimizer state    602.0 MiB
```

出力された数値は、訓練ステップのライフサイクルを正確にたどっている。まず重み（$4N$ バイト）と入力がメモリ上に置かれ、順伝播によって保存された活性化値（約 192 MiB、重みを上回る）が加わり、逆伝播がその活性化値を消費・解放しつつ勾配（$4N$）を実体化し、最初のオプティマイザステップが Adam の 2 つのモーメント（$8N$）を確保する。Figure 5.17 は、16 MiB の入力バッチを除いたこれら 4 つの段階を示したものである。2 ステップ目以降は、毎ステップの中でメモリが増減する。順伝播で重みと Adam のモーメントの上に活性化値が保存され、逆伝播でそれが勾配に置き換わる。ピーク使用量は高々、静的な $16N$ に活性化値を加えた量であり、次の順伝播の前に古い勾配を解放するループでは最大 $4N$ だけ低くなる。活性化値が原因でメモリ上限を超えてしまう場合、アーキテクチャを変えずに活性化メモリを削減できるミニバッチサイズの縮小が最も手軽な対策となる。対照的に、メモリが**ステップを追うごとに**着実に増え続ける場合は、古い計算グラフを生かし続けているメモリリークを疑うべきである。典型的な原因は、生の損失テンソルを切り離さずそのまま Python のリストに追加してしまうことである。

`optimizer.zero_grad()` はデフォルト（`set_to_none=True`）で勾配を解放するため、順伝播の前にこれを呼び出すループでは、活性化値を保持している間のパラメータ関連メモリは $12N$ バイト（$16N$ ではなく）にとどまる。

5.7.3.3 活性化チェックポイント

活性化メモリを抑えるには、アーキテクチャ面での 2 つ目の手段がある。通常の誤差逆伝播法は、逆方向の計算で一度読み出すためだけに、すべての層の中間出力を保存している。**活性化チェックポイント**（**勾配チェックポイント**（gradient checkpointing）や**再実体化**（rematerialization）とも呼ばれる (Chen et al. 2016)）は、順伝播中に大半の中間活性化値を破棄して選択したセグメントの境界にある入力だけを保持し、逆伝播中に各セグメント内部の活性化値をその場で再計算する。（同じ「チェックポイント」という語を含むが、このメモリ上の手法は 5.6 節で扱ったディスクへのチェックポイント保存とは無関係である。）

均一な K 個の層を s 層ずつのセグメントに分割すると、保持されるのは K/s 個のセグメント境界と、処理中のセグメント内部で生存している s 個の活性化値のみとなる（約 $K/s + s$ 個のテンソルであり、K 個すべてではない）。$s = \sqrt{K}$ と選べば、ピークの活性化メモリは $O(K)$ から $O(\sqrt{K})$ へと減少する。その代償は、誤差逆伝播法におけるセグメントあたり 1 回分の追加の順伝播である（Figure 5.18）。線形層の逆伝播が 2 回の行列積（入力勾配と重み勾配）を計算するのに対し、順伝播は 1 回の行列積で済むため、順伝播が 1 回増えてもステップ全体の計算量は順伝播換算で約 3 回分から 4 回分に増えるだけであり、ハードウェアや層の構成にもよるが大まかな上限として約 33% の増加にとどまる。

活性化チェックポイントが真価を発揮するのは、5.1 節の残差 MLP や現代の Transformer のように、同一構造のブロックを深く積み重ねたモデルである。以下では、重みを $\mathcal{N}(0, 0.01^2)$、バイアスを 0 で初期化した `ResidualBlock` と、ブロックをセグメントにまとめて任意でチェックポイントを適用する `run_stack` ヘルパを定義する。

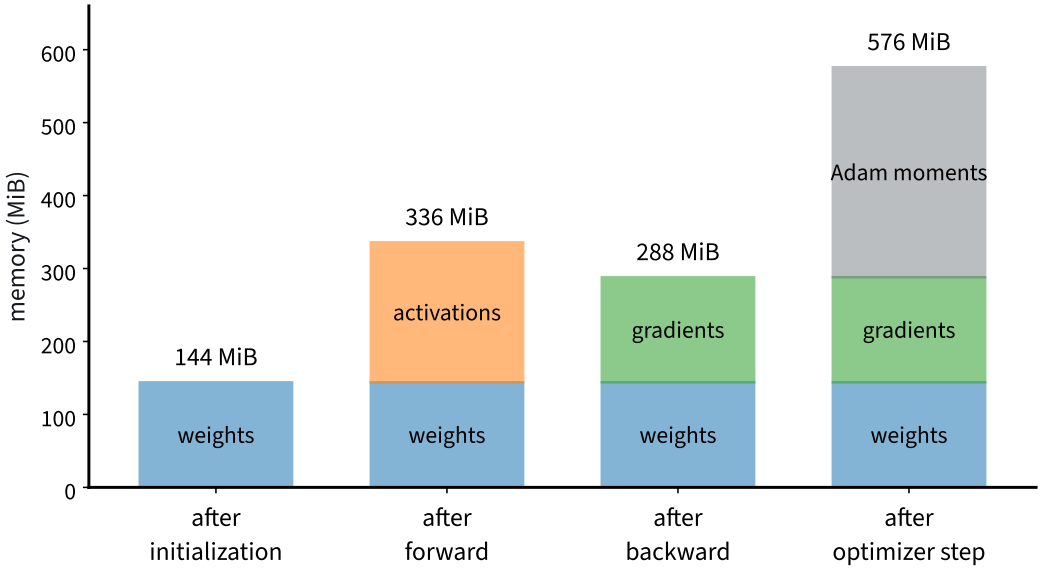


Figure 5.17: 幅 4096 の MLP をミニバッチサイズ 4096 で訓練した際の、最初の訓練ステップの各段階後における使用中メモリ（入力バッチは含めない）。順伝播では重み（$4N$）に 3 つの保存された活性化値が加わる。逆伝播では活性化値が解放されて勾配（$4N$）が残る。オプティマイザステップでは Adam の 2 つのモーメント（$8N$）が加わる。

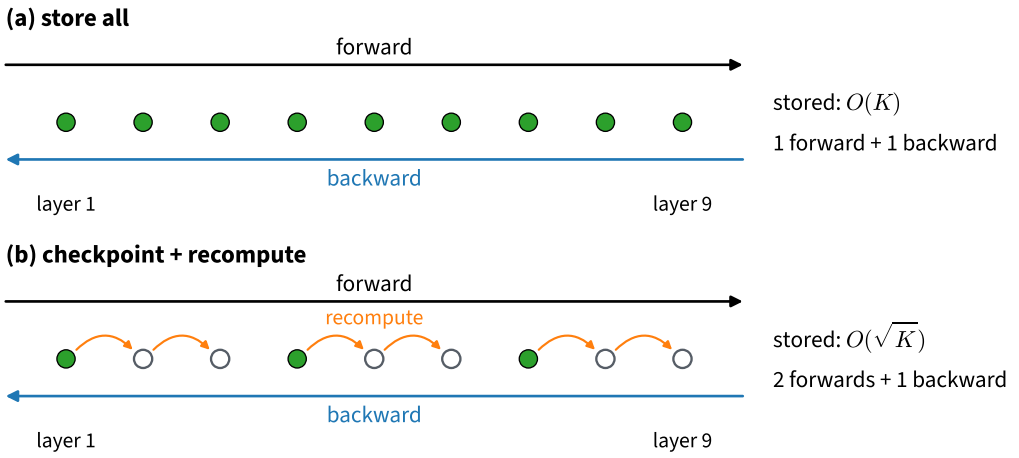


Figure 5.18: (a) 通常の誤差逆伝播法はすべての層の活性化値を保存する。(b) チェックポイントを用いると、セグメントの入力（塗りつぶし）のみを保存し、各セグメント内部の活性化値（白抜き）は逆伝播中に再計算する。K 個の層を $\sqrt{K}$ 層ずつのセグメントに分けると、1 回分の追加の順伝播をコストとして、保存される活性化値は $O(K)$ から $O(\sqrt{K})$ に減少する。

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))
        for layer in self.body[0], self.body[2]:
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)

def run_stack(blocks, X, use_checkpoint=False, segment_size=4):
    for start in range(0, len(blocks), segment_size):
        stop = min(start + segment_size, len(blocks))
        # Bind this segment's range now: the backward pass calls segment
        # again after the loop has moved on
        def segment(X, start=start, stop=stop):
            for i in range(start, stop):
                X = blocks[i](X)
            return X
        X = checkpoint(segment, X, use_reentrant=False) \
            if use_checkpoint else segment(X)
    return X
```

`torch.utils.checkpoint.checkpoint(segment, X, use_reentrant=False)` は `segment` 内部の中間活性化値を破棄し、乱数生成器の状態を保存・復元して（ドロップアウトの一貫性を保ちつつ）、`segment` を `backward()` の実行中に再実行する。

まず健全性チェックとして、活性化値の再計算によって数学的な計算結果が一切変わらないことを確認しておこう。8 ブロックのスタックを 2 つのチェックポイント付きセグメントに分割し、すべての勾配が通常の誤差逆伝播法と一致することを検証する。

```
blocks = nn.Sequential(*[ResidualBlock(64) for _ in range(8)])
X = torch.randn(32, 64)

def stack_grads(use_checkpoint):
    blocks.zero_grad(set_to_none=True)
    run_stack(blocks, X, use_checkpoint).sum().backward()
    return [p.grad.clone() for p in blocks.parameters()]

for g, g_ckpt in zip(stack_grads(False), stack_grads(True)):
    assert torch.allclose(g, g_ckpt)
print('checkpointed gradients match the ordinary ones')
```

```
checkpointed gradients match the ordinary ones
```

この正しさの確認はどの環境でも実行できるが、メモリと計算量のトレードオフを実測するため、次は GPU 上でより深い 16 ブロックのスタック（幅 1024、バッチサイズ 8192）を 4 ブロックずつのセグメントにまとめ、**ステップメモリ**（パラメータと入力を除いたピークメモリ）と `Benchmark` による実時間の両方を計測する。

```
if torch.cuda.is_available():
    blocks = nn.Sequential(*[ResidualBlock(1024) for _ in range(16)]).to(gpu())
    X = torch.randn(8192, 1024, device=gpu())
    def step(use_checkpoint):
        blocks.zero_grad(set_to_none=True)
        run_stack(blocks, X, use_checkpoint).sum().backward()
    for use_checkpoint in (False, True):
        timing = Benchmark(lambda: step(use_checkpoint))
        blocks.zero_grad(set_to_none=True)  # Gradients count as step memory
        peak = peak_memory(step, use_checkpoint)
        print(f'checkpointing={use_checkpoint!s:5}  step memory '
              f'{peak / 2**20:5.0f} MiB, {timing.time * 1e3:5.1f} ms per step')
else:
    print('No CUDA device: the peak-memory comparison needs a GPU.')
```

```
checkpointing=False  step memory  1096 MiB,  25.5 ms per step
checkpointing=True   step memory   452 MiB,  32.3 ms per step
```

チェックポイントなしでは、逆伝播は 16 ブロックすべての活性化値を同時に保持する。4 ブロックずつのセグメントにまとめると、保持されるのは 4 つのセグメント境界と処理中の 1 セグメントのみとなり、計算時間が約 4 分の 1 増える（先ほどの概算による上限 33% の範囲内である）だけで、ステップメモリは約 1.1 GiB から 0.4〜0.5 GiB へと削減される。この有利なトレードオフにより、大規模な Transformer や深い視覚モデルの訓練において、活性化チェックポイントは標準的な手法となっている。

5.7.4 デバイス対応の Trainer

これでようやく `num_gpus` を `d2l.Trainer` に組み込むことができる。ここまでの原則を実践に移すには、たった 2 つの手順で済む。訓練が始まる前にモデルパラメータを目的のデバイス上に配置すること、そして毎ステップの冒頭で各ミニバッチをそのデバイスへ移動することである。

```
@d2l.add_to_class(d2l.Trainer)
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]
    self.board = d2l.ProgressBoard(xlabel='epoch', ylabel='loss')

@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    if self.gpus:
        batch = [d2l.to(a, self.gpus[0]) for a in batch]
    return batch

@d2l.add_to_class(d2l.Trainer)
def prepare_model(self, model):
    self.model = model
    if self.gpus:
        model.to(self.gpus[0])
        # Pick convolution algorithms by timing, unless determinism is wanted
        torch.backends.cudnn.benchmark = not (
            torch.are_deterministic_algorithms_enabled()
```

```
            or torch.backends.cudnn.deterministic)
    # Lazy parameters get their shapes from a dry pass on the device, then
    # the model's weight_init, if it has one, before the optimizer exists
    if any(isinstance(p, nn.parameter.UninitializedParameter)
           for p in model.parameters()):
        loader = (self.train_dataloader if self.train_dataloader is not None
                  else self.val_dataloader)  # `evaluate` has no training data
        batch = self.prepare_batch(next(iter(loader)))
        model.apply_init(batch[:-1], getattr(model, 'weight_init', None))
```

`min(num_gpus, d2l.num_gpus())` を使うことで、`num_gpus` は必須条件ではなく上限として扱われる。CPU のみのマシンでは `self.gpus` が空になり、訓練はそのままシームレスに CPU 上で実行される。本書の残りの章におけるすべての `Trainer(num_gpus=1)` 呼び出しの背後にあるのは、まさにこのフォールバック機構である。

小さなモデルの訓練は、GPU によって実際どれほど速くなるのだろうか。4096×4096 の行列積は約 $100\times$ 以上高速化されたが、エンドツーエンドの訓練ループにはデータ読み込み、Python のディスパッチ、そしてより小さな層が含まれる。以下では、幅 256 の残差ブロック（5.7.3.3 節）を 2 つ重ね、重みを $\mathcal{N}(0, 0.01^2)$、バイアスを 0 で初期化した Fashion-MNIST 分類器で計測してみよう。

```
class ResMLPClassifier(d2l.Classifier):
    def __init__(self, num_hiddens=256, num_blocks=2, lr=0.1):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.Linear(784, num_hiddens), nn.ReLU(),
            *[ResidualBlock(num_hiddens) for _ in range(num_blocks)],
            nn.Linear(num_hiddens, 10))
        for layer in self.net[1], self.net[-1]:  # The blocks set their own
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)
```

Fashion-MNIST におけるウォームアップ後の 1 エポックの実行時間を、`num_gpus=0` と `num_gpus=1` で比較する。

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

def epoch_time(num_gpus):
    model = ResMLPClassifier()
    trainer = d2l.Trainer(max_epochs=1, num_gpus=num_gpus)
    trainer.board.display = False
    trainer.fit(model, data)  # The first epoch warms up
    return Benchmark(trainer.fit_epoch, warmup=0, repeats=1).time

for gpus in (0, 1):
    print(f'num_gpus={gpus}: {epoch_time(num_gpus=gpus):.1f} sec per epoch')
```

```
num_gpus=0: 2.9 sec per epoch
num_gpus=1: 1.1 sec per epoch
```

この高速化倍率は良い現実確認になる。この小さな Fashion-MNIST ネットワークでは、GPU で

の 1 エポックは CPU より高々数倍速いだけであり、$100\times$ の向上（4096×4096 の行列積での倍率）には遠く及ばない。ここでは最大の行列でも 784×256 にすぎないため、GPU の計算能力の大部分が遊んでしまい、データの読み込みとカーネル起動のオーバーヘッドが実行時間の大半を占めている。次章以降で扱うように、大きな畳み込み、アテンションヘッド、幅の広い線形層が計算負荷の主役になって初めて、GPU は本領を発揮する。

メモリの観点では、この小さな SGD モデルは数 GB もの余裕を残して楽に収まる。しかし、80 億パラメータのモデルに立ち返ってみよう。バッチサイズを縮小し活性化チェックポイントを有効にすれば活性化メモリは抑えられるものの、fp32 の重み・勾配・Adam のモーメントが占める 128 GB の静的な下限は手つかずのままであり、しかも `Trainer` は OOM に直面する前に事前警告を出すことはできない。概算によるメモリ見積もりは依然として不可欠であり、単一デバイスのメモリを超えてスケールさせるには第 13 章で扱うマルチ GPU の並列性（parallelism）へと進むことになる。

5.7.5 まとめ

すべてのテンソルとモデルパラメータは特定のデバイス上に存在し、あらゆる演算はそのオペランドが同じデバイス上にあることを要求する。ホスト・デバイス間の転送は GPU の計算よりはるかに遅いため、モデルは最初に一度だけ配置し、バッチはステップの境界で転送し、内側ループはデバイス上に留める必要がある。GPU はコマンドキューから非同期に実行するため、Python が停止するのは同期ポイント（スカラーから Python の浮動小数点数への変換や NumPy への変換など）のみであり、信頼できるベンチマークにはウォームアップ、明示的な同期、および反復試行が欠かせない。訓練時のメモリは静的なモデル状態・オプティマイザ状態とバッチ依存の活性化値からなり、活性化チェックポイントは約 1 回分の追加の順伝播をコストとして活性化メモリを $O(K)$ から $O(\sqrt{K})$ へと削減する。最後に、デバイス対応の `Trainer` は単一 GPU への配置を自動化する一方で、GPU がない環境では CPU へとスムーズにフォールバックする。

5.7.6 演習問題

1. [code] **ピークメモリ**。幅が 1024、4096、4096、4096 で出力数が 10 の 4 層 MLP を、5.7.3.2 節と同様に Adam で訓練する場合を考える。同項の計算方法を用いて、重み、勾配、オプティマイザ状態、および逆伝播が保持する活性化値を数え上げ、バッチサイズ 64、256、1024 における 1 訓練ステップのピークメモリを予測せよ。次に、GPU 上で実際のピーク値を計測せよ。この予測はどこで崩れ、何を見落としていたか。

2. **チェックポイントの計算**。幅 1024 の `ResidualBlock` を 16 個積み重ねたスタックをバッチサイズ 8192 で動かす場合について、Figure 5.18 のスケジュールを紙の上で書き出せ。

    1. 1 つのブロック内部で逆伝播が保持しなければならない配列を挙げ、その要素数を数えよ。
    2. チェックポイントなしの場合と、セグメントサイズを 1、2、4、8 ブロックとした場合について、逆伝播のピーク時に保持される活性化メモリを計算せよ。また、各スケジュールがどれだけの順伝播計算を繰り返すか述べよ。
    3. この設定ではどのセグメントサイズを選ぶべきか。また、その選択は 5.7.3.3 節で述べた約 $\sqrt{K}$ 層ごとのセグメントという規則とどのように関係するか。

3. [code] **非同期ディスパッチ**。●以下の両問とも、5.7.2.1 節の時間計測の原則を用いよ。

    1. 256 から 8192 までの n について、$n \times n$ 行列を CPU から GPU へ送り再び戻す往復時間と、デバイス上での 1 回の $n \times n$ 行列積の時間を計測せよ。両者の時間が等しくなる n を求めよ（この範囲内で交差しない場合は、各時間が n に対してどう増加するかから外挿せよ）。使用している GPU の公称ホスト・デバイス間帯域幅と行列積スル

ープットからこの交差点を予測し、予測値と実測値の間に差があればその理由を説明せよ。

2. GPU が 2 基ある場合、2 つの 4096×4096 行列の積を 1 基の GPU 上で 1000 回実行する時間と、同じループから 2 基の GPU にそれぞれ 500 回ずつ発行して実行する時間を計測せよ。スケーリングを計測し、非同期ディスパッチによって Python がどのように両デバイスへ処理を発行できるかを説明したうえで、線形スケーリングから乖離する要因を特定せよ。実験コードは `num_gpus() >= 2` でガードせよ。

4. [code] **同期ポイント。●**GPU 上における `ResMLPClassifier` の訓練実行（5.7.4 節）を考える。

   1. モデルのコードに加えて `Trainer.fit_epoch`、`Trainer.prepare_batch`、`Averager` を読み、1 訓練ステップ内のすべての同期ポイントを列挙せよ。それぞれを「正しいループにとって不可避なもの」か「回避可能な計測・記録上の副産物」かに分類し、その分類の根拠を述べよ。
   2. ループが毎ステップ後に損失を Python の浮動小数点数へ読み出すとする。作成したリストをもとに、この読み出しによってデフォルトの `ResMLPClassifier` と `ResMLPClassifier(num_hiddens=2048, num_blocks=8)` がそれぞれどれだけ遅くなるかを予測せよ。次に、各モデルについて読み出しあり・なしの 1 エポックを計測し、4 つの計測結果を説明せよ。

PyTorch ドキュメントの CUDA semantics に基づき改変。

5.8 再現性と検査

訓練スクリプトを 1 行だけ書き換えたところ、Fashion-MNIST における検証正解率が 0.3 パーセントポイント上がったとしよう。ぬか喜びする前に、スクリプトを**変更しない**まま 2 回実行しただけでどの程度結果がばらつくかを知る必要がある。5.8.1.5 節では、乱数シード（random seed）だけを変えてソフトマックス回帰（softmax regression）を 5 回実行する。すると、初期重みとバッチのシャッフルが異なるだけで最終的な正解率には 1〜4 パーセントポイントもの幅が生じ、0.3 ポイントの改善など完全に埋もれてしまうことがわかる。

クラッシュする訓練のデバッグは、これと表裏一体の問題である。数時間の訓練の後に損失が爆発して NaN になった場合、クラッシュに至るまでのステップ列を正確に再現し、実行中のネットワークの内部を調べて、どの層が最初に非有限値を出力したのかを突き止めたい。

本節では、この両面をまとめて扱う。まず、擬似乱数生成器（pseudorandom number generator）、すなわちグローバルシード、専用のジェネレータオブジェクト、マルチワーカーのデータローダを制御して実行を再現可能にし、続いて GPU 特有のさらに厄介な問題である、並列アトミックリダクションのもとでの浮動小数点演算の非結合性に取り組む。次に、**フック**（および各フレームワークにおける同等の仕組み）を用いて、モデルのコードを変更することなく実行中のモデルを検査し、活性化値（activation）のスケールの記録、最初に NaN を出す層の捕捉、勾配ノルム（gradient norm）の監視を行う。この両面は互いに補完し合う関係にある。内部の診断結果を 2 つのコードバージョン間で比較できるのは、実行間で他の条件が一切変わらない場合に限られるからである。初読時はジェネレータオブジェクトとデータローダのワーカーに関する項を軽く読み流し、データパイプラインでランダムなデータ拡張（data augmentation）を扱うようになってから読み返してもよい。

```
from torch.utils.data import DataLoader, Dataset
```

5.8.1 シードと乱数生成器

訓練パイプラインには、多くの実務者が意識している以上に多様な箇所からランダム性が入り込む。たとえば、重みの初期化（5.3 節）、毎ステップのドロップアウトマスク、エポックごとのミニバッチのシャッフル、ランダムなデータ拡張（切り抜き、反転、色ジッター）、そして並列に動作するデータローダのワーカープロセスなどである。すべての乱数は**擬似乱数生成器**（PRNG）から生成される。これは、初期**シード**によって出力系列全体が決定される決定論的なアルゴリズムである。したがって、パイプラインが利用するすべての乱数生成器にシードを設定すれば、あらゆる乱数抽出を固定できる。

5.8.1.1 グローバルシード

深層学習フレームワークは、共有されたグローバルな乱数生成器から乱数を引き出すか、あるいは生成器の状態を引数として明示的に渡すことを要求する。以下では、重みと合成入力のすべてをシード付きの 1 つの乱数源から生成する、5 ステップの小規模な訓練ループでシード設定の実例を示す。

`torch.manual_seed(k)` は、フレームワークの CPU および GPU 用のグローバルな乱数生成器にシードを設定する（Python の `random` や NumPy の `np.random` を使う場合は、別途シードを設定する必要がある）。

```
def train_once(seed):
    torch.manual_seed(seed)  # seeds the CPU and every GPU
    net = nn.Sequential(nn.Linear(20, 32), nn.ReLU(), nn.Linear(32, 1))
    opt = torch.optim.SGD(net.parameters(), lr=0.1)
    X, y = torch.randn(128, 20), torch.randn(128, 1)
    for _ in range(5):
        opt.zero_grad()
        loss = ((net(X) - y) ** 2).mean()
        loss.backward()
        opt.step()
    return loss.item(), net[0].weight.detach().clone()

loss_a, w_a = train_once(seed=0)
loss_b, w_b = train_once(seed=0)
loss_c, _ = train_once(seed=1)
print('same seed, identical loss and weights:',
      loss_a == loss_b, torch.equal(w_a, w_b))
print('different seed:', loss_a, 'vs', loss_c)
```

```
same seed, identical loss and weights: True True
different seed: 0.8643209338188171 vs 0.9312677383422852
```

`seed=0` を指定した 2 回の実行は**ビット単位で**一致する（すべての重みと最終損失が同一になる）。同一の初期重み、データ、および決定論的な算術演算からは同一の勾配が生じるからである（演算順序については 5.8.2 節で調べる）。シードを 1 に変更すると、異なる軌跡（trajectory）と損失が得られる。

5.8.1.2 ジェネレータオブジェクト

単一のグローバル PRNG に依存すると、そこから乱数を引くすべてのコードが結合してしまう。層を 1 つ追加したり、デバッグ中に乱数を伴う一時的な表示処理を入れたりするだけで、後続の乱数ストリーム全体がずれ、訓練・検証データの分割、重みの初期化、ドロップアウトマスクが

気づかないうちに一斉に変わってしまう。専用の**ジェネレータオブジェクト**を使えば、独自のシードの背後にストリームを分離できるため、他の箇所での無関係な乱数抽出によって乱されることは決してない。

`torch.Generator().manual_seed(42)` は、グローバルな乱数抽出の影響を受けない独立した PRNG ストリームを生成する。

```
g = torch.Generator().manual_seed(42)
split = torch.randperm(10, generator=g)
_ = torch.randn(1000)  # unrelated draws from the global generator
split_again = torch.randperm(10, generator=g.manual_seed(42))
print(torch.equal(split, split_again), split)
```

```
True tensor([2, 6, 1, 8, 4, 5, 0, 9, 3, 7])
```

5.8.1.3 データローダのワーカー

データローダが並列の**ワーカー**プロセスでミニバッチを準備する場合、ランダムなデータ拡張には気づきにくい 2 つの失敗モードがある。ワーカーが OS の `fork` によってメインプロセスのクローンとして起動され、シードが再設定されない場合、すべてのワーカーがまったく同じ生成器の状態を引き継ぎ、同一の「ランダム」なデータ拡張を生成してしまう。逆に、各ワーカーが OS のエントロピーから初期化されると、2 回の実行が一致することは二度となくなる。

`DataLoader` は各ワーカーの PyTorch、`random`、およびグローバルな NumPy の RNG に `base_seed + worker_id` からシードを設定する。しかし、専用の `np.random.default_rng()` 属性を `Dataset` に持たせた場合、`worker_init_fn` で `torch.utils.data.get_worker_info().seed` を使ってシードを再設定しない限り、fork されたワーカー間で同一のクローンが作られてしまう。

```
class NoisyData(Dataset):
    """Eight examples whose values are random draws, as in augmentation."""
    def __init__(self):
        self.rng = np.random.default_rng()  # the dataset's own generator

    def __len__(self):
        return 8

    def __getitem__(self, i):
        return self.rng.random()

def seed_worker(worker_id):
    info = torch.utils.data.get_worker_info()
    info.dataset.rng = np.random.default_rng(info.seed)  # this worker's seed

for init_fn in (None, seed_worker):
    loader = DataLoader(NoisyData(), batch_size=4, num_workers=2,
                        worker_init_fn=init_fn,
                        multiprocessing_context='fork',  # copies of this one
                        generator=torch.Generator().manual_seed(0))
    print([[round(x, 3) for x in batch.tolist()] for batch in loader])
```

```
[[0.61, 0.817, 0.884, 0.742], [0.61, 0.817, 0.884, 0.742]]
[[0.777, 0.641, 0.352, 0.467], [0.653, 0.267, 0.932, 0.823]]
```

ワーカーごと、または例ごとのシード再設定がない場合、1 行目では両方のワーカーが同一の「ランダム」な数値を出力する。適切にシードを設定した 2 行目では、互いに異なりかつ再現可能な乱数が得られている。

5.8.1.4 値としてのランダム性

上述のワーカーに関するバグはいずれも、隠れた可変な生成器の状態が裏側で暗黙のうちにコピーまたは共有されることに起因する。関数型の乱数 API は、ランダム性を明示的な値として扱うことで隠れた状態を排除する。すなわち、すべての乱数呼び出しはキーを受け取り、同じキーで 2 回呼び出せば同一の数値が返り、独立したストリームはキーを明示的に分割することで得る。誤ってキーを再利用してしまう可能性は残るものの、その再利用はコード上にそのまま現れる。その代償は、確率的なすべての関数にキーを引き回す手間である。グローバルシードとはこの同じ契約を暗黙に略記したものにすぎず、専用のジェネレータオブジェクトを使えば必要な箇所で明示的な分離を取り戻せる。

5.8.1.5 シード間のばらつき

ここで、本節の冒頭で触れたシード間の分散を実際に測定してみよう。以下では、Fashion-MNIST に対するソフトマックス回帰（3.3.5 節）を GPU 上で 3 エポック、5 つのシード（0〜4）にわたって訓練する。関係するすべての乱数生成器にシードを設定し、各実行の最終的な検証正解率と重みを返す。

```
def train_classifier(seed, data):
    torch.manual_seed(seed)  # initialization and shuffle order
    model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
    trainer = d2l.Trainer(max_epochs=3, num_gpus=1)
    trainer.board.display = False
    trainer.fit(model, data)
    return trainer.history['val_acc'][-1], model.net[1].weight.detach().cpu()

data = d2l.FashionMNIST(batch_size=256)
runs = [train_classifier(seed, data) for seed in range(5)]
accs = 100 * np.array([acc for acc, _ in runs])
print('validation accuracy (%):', ' '.join(f'{a:.1f}' for a in accs))
print(f'mean {accs.mean():.1f}, max - min {np.ptp(accs):.1f} points')
```

```
validation accuracy (%): 79.8 81.8 81.9 78.3 82.1
mean 80.8, max - min 3.7 points
```

5 回の実行すべてで同一のコードとハイパーパラメータを実行しているにもかかわらず、最終的な正解率は 81% 前後で 1〜4 パーセントポイントの広がりを見せており、1 回の実行で得られた 0.3 ポイントの上昇は単なるノイズにすぎないことが確認できる。信頼できる実験比較では、必ず複数のシードにわたる平均とばらつきを報告する。

この差がシードの選択だけによって生じたことを確かめるため、`seed=0` を再実行し、正解率とすべての重みを最初の実行と比較してみよう。

```
acc, w = train_classifier(0, data)
print('same accuracy:', acc == runs[0][0],
      '| same weights:', torch.equal(w, runs[0][1]),
      f'| largest difference {(w - runs[0][1]).abs().max().item():.2g}')
```

```
same accuracy: True | same weights: True | largest difference 0
```

再実行の検証正解率は最初の実行と完全に一致する。個々の重みが最下位ビットまで一致するかどうかは、GPU が浮動小数点数を一定の順序で足し合わせるかどうかに依存しており、これはシードだけでは制御できない（5.8.2 節）。

5.8.2 決定論的な算術演算

シードの設定は、実行時に**どの**乱数を引くかを固定するものであり、浮動小数点演算がそれらを**どのように**組み合わせるかまで固定するわけではない。浮動小数点の加算は毎回、表現可能な最も近い値へと丸められるため、結合法則を満たさ**ない**。すなわち、足し合わせる順序を変えると結果が変わり得る。

3 つの float32 の数値を使えば、この影響は一目瞭然となる。10^8 付近では、隣り合う float32 の数値の間隔が 8 になるため（5.5 節）、$-10^8 + 1$ は丸められて元の -10^8 に戻ってしまう。

```
a, b, c = [torch.tensor(v, dtype=torch.float32) for v in (1e8, -1e8, 1.0)]
print(float((a + b) + c), float(a + (b + c)))  # same numbers, regrouped
```

```
1.0 0.0
```

括弧の位置をずらすだけで、答えが `1.0` から `0.0` に変わってしまう。長いリダクションではこのずれはより微妙になるが、至る所で生じる。100 万個の浮動小数点数を先頭から足す場合と末尾から足す場合とでは、最下位ビット付近が日常的に食い違う。

スレッド数が固定された CPU 上では、リダクションの順序は通常一定に保たれるため、シードを設定した実行はきれいに再現される。一方 GPU 上では、数千ものスレッドが並列に競合しており、いくつかの高速なカーネル（埋め込みの勾配、スキャッタ加算（scatter-add）、一部の畳み込みの逆伝播、各種リダクションなど）は、スレッドが完了した順に到着する**アトミック加算**（atomic addition）によって部分結果を累積する。そのため、同じ GPU 上で同一のシードを設定した 2 回の実行であっても最下位ビットに差異が生じ、それがカオス的な勾配ステップによってやがて目に見えるほど異なる損失曲線へと増幅され得る。どのフレームワークにも、通常いくらかの速度コストと引き換えに一定の加算順序を強制する決定論モード（または代替カーネル）が用意されている。以下では、GPU 上の**スキャッタ加算**で 100 万個の乱数を 10 個のビンに 2 回続けて足し合わせ、この現象を直接観察してみよう。

`torch.use_deterministic_algorithms(True)` は、`index_add_` のような演算を決定論的カーネルに切り替える（該当するカーネルが存在しない場合は `RuntimeError` を送出する）。

```
if torch.cuda.is_available():
    src = torch.randn(2**20, device='cuda')  # a million numbers
    idx = torch.randint(0, 10, (2**20,), device='cuda')  # their bins
    for deterministic in (False, True):
        torch.use_deterministic_algorithms(deterministic)
        a, b = [torch.zeros(10, device='cuda').index_add_(0, idx, src)
                for _ in range(2)]
        print(f'deterministic {deterministic!s:5}: bitwise equal '
              f'{torch.equal(a, b)}, largest difference '
              f'{(a - b).abs().max().item():.2g}')
    torch.use_deterministic_algorithms(False)
else:
    print('no GPU: on a CPU this sum adds in a fixed order')
```

```
deterministic False: bitwise equal False, largest difference 0.0051
deterministic True : bitwise equal True, largest difference 0
```

決定論モードを有効にしない場合、まったく同じ GPU データに対する連続した 2 回のスキャッタ加算では、数百規模の和に対して数千分の 1 程度の差が生じる。決定論的カーネルを使えば、両方の呼び出しはビット単位で一致する。

ビット単位の再現性が成り立つ範囲を正しく認識しておく必要がある。それが保証されるのは同一のマシン、ソフトウェアスタック、ハードウェア構成の上だけであり、ライブラリのバージョンアップや異なる GPU アーキテクチャ、あるいは CPU と GPU の間でビット単位の一致を約束するフレームワークは存在しない。ビット単位の決定論は、2 回の実行が分岐する正確なステップを二分探索で特定するための非常に有用な**デバッグ**手段である。一方、科学的な再現性はこれより緩やかであると同時に、より強い要請でもある。本質的な改善（「このアーキテクチャ変更によって正解率が向上する」といった主張）は複数のシードにわたって頑健に成り立たなければならず、5.8.1.5 節と同様に平均とばらつき（約 81%、幅 1〜4 ポイント）として報告すべきである。16 ビット形式の場合（5.5 節）と同じく、シードの変更や最下位ビットの丸めによって覆るような主張には、より強い根拠が必要である。

5.8.3 フックによるモデルの検査

バグを確実に再現できたとしても、それで証明されるのはバグが実在することだけであり、ネットワーク内部の**どこで**発生しているかまではわからない。すべての層の `forward` メソッドに `print` 文を挿入するのは不格好であり、ソースコードをフォークしたくないサードパーティ製ライブラリのモデルでは現実的でもない。**フック**を使えば、この問題をきれいに解決できる。フックとはモデルの外部から登録するコールバックであり、層の実行前または実行後にフレームワークが自動的に呼び出す仕組みである。

`module.register_forward_hook(f)` は `f(module, inputs, output)` を `__call__` 内で `forward` の後に実行する（新しいテンソルを返すと出力が置き換わる）。

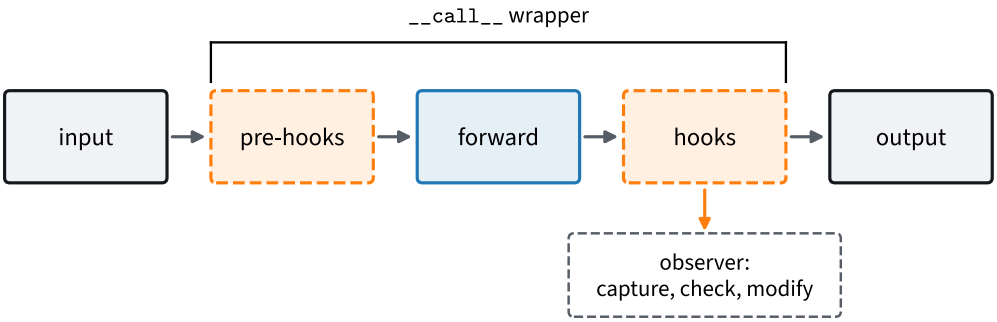


Figure 5.19: パイプラインとしての `__call__` ラッパー。入力は事前フック、forward、フックを順に通って出力へと流れる。forward の青い実線に対し 2 つのフック段階はオレンジ色の破線で示されており、フック段階から横に出る矢印は値の取得・検査・変更を行うオブザーバを表す。

検査の例として、入力数 20 の層と出力数 10 のヘッドの間に幅 64 の残差ブロック（residual block）を 8 個積み重ねたネットワークを構築する（5.1 節）。各ブロックは $\mathbf{h} = \mathbf{x} + \mathrm{body}(\mathbf{x})$ を計算し、重みには Xavier 初期化（4.4.2.1 節）、バイアスにはゼロを用いる。実行ごとに新しいランダムな重みが引かれるよう、このネットワークにはあえてシードを設定しないでおく。以下の診断結果の測定値を典型的な範囲として示しているのはそのためである。

ここまでのシード付き実行によって PyTorch のグローバルな乱数生成器が固定された状態になっており、そのままではこのモデルの重みまで固定されてしまうため、このセルではまず `torch.seed()` によってオペレーティングシステムのエントロピーから生成器のシードを再設定している。

```
torch.seed()  # a fresh seed from the operating system

class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(num_hiddens, num_hiddens),
                                  nn.ReLU(),
                                  nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)

net = nn.Sequential(nn.Linear(20, 64),
                    *[ResidualBlock(64) for _ in range(8)],
                    nn.Linear(64, 10))
for m in net.modules():
    if isinstance(m, nn.Linear):  # Xavier weights, zero biases
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)
```

5.8.3.1 活性化値の統計量

5.3 節では、ブロックを手動でループして深さ方向の活性化値の標準偏差を測定した。フックを使えば、モデルに一切手を加えることなく、標準正規分布に従う 256 個の入力からなるミニバッチに対してわずか数行でまったく同じプロファイルを収集できる。

```
stats, handles = [], []

def record(module, inputs, output):
    stats.append((type(module).__name__, output.detach().std().item()))

for m in net:
    handles.append(m.register_forward_hook(record))
net(torch.randn(256, 20))
for h in handles:
    h.remove()
for name, std in stats:
    print(f'{name:15s} std {std:.2f}')
```

```
Linear          std 0.72
ResidualBlock   std 0.88
ResidualBlock   std 1.09
ResidualBlock   std 1.31
ResidualBlock   std 1.62
ResidualBlock   std 1.98
ResidualBlock   std 2.28
ResidualBlock   std 2.93
```

```
ResidualBlock   std 3.58
Linear          std 3.51
```

活性化値の標準偏差は深さとともに着実に増加し、入力層の直後では約 0.7 だったものが 8 番目のブロックの後では 3〜4.5 前後に達しており、理論値とよく一致している。Xavier 初期化により $20 \to 64$ の入力層の重みの分散は $2/84$ となり、出力の分散は $20 \cdot 2/84 \approx 0.48$（標準偏差 0.69）となる。続く各残差ブロックは、入力とほぼ無相関で分散が入力分散の半分となる body 出力を加算するため（最初の線形層は分散を保ち、ReLU が二乗平均を半分にし、2 番目の線形層がその半分を保つ）、ブロックごとに全体の分散は 1.5 倍（標準偏差は $\sqrt{1.5} \approx 1.22$ 倍）になる。8 ブロック全体では $0.69 \times 1.5^4 \approx 3.5$ となり、実測値のちょうど中央に収まる。これは 5.3 節で行った分散増加の解析そのものであり（ここでは係数が 1.5 であり、He 初期化のもとでの 2 とは異なる）、深い残差ブランチにおいて出力射影（output projection）のスケールを縮小することが有効な理由を改めて裏づけている。

フックによるメモリリークを防ぐには 2 つの原則がある。フック内で保存するテンソルは必ず `.detach()` すること（さもないと自動微分の計算グラフ全体がステップをまたいでメモリ上に残り続ける）、そして検査が終わったら必ずフックのハンドルを削除すること（`h.remove()`）である。

5.8.3.2 NaN 検出器

ステップ 40,000 で訓練損失（training loss）が突然 NaN になったとき、最初の診断上の疑問は**どの層が最初に非有限値を出力したのか**である。フックは順伝播（forward pass）の実行順に呼び出されるため、すべてのリーフ層に有限性チェックのフックを取り付ければ、1 回の順伝播で原因箇所を特定できる。この検出器を試すため、3 番目の残差ブロック（インデックス 3）の重みの 1 つに意図的に NaN を注入してみよう。

```
def make_finite_check(name):
    def check(module, inputs, output):
        if not torch.isfinite(output).all():
            raise RuntimeError(f'first non-finite output in {name}')
    return check

handles = [m.register_forward_hook(make_finite_check(name))
           for name, m in net.named_modules()
           if len(list(m.children())) == 0]  # leaf modules only
saved = net[3].body[0].weight[0, 0].item()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = float('nan')  # break one layer
try:
    net(torch.randn(2, 20))
except RuntimeError as e:
    print(e)
for h in handles:
    h.remove()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = saved  # restore the weight
```

```
first non-finite output in 3.body.0
```

層のクラス定義の中に print 文を 1 つも追加することなく、フックによってブロック 3 内の破損した層が即座に特定される。もちろん、意図的に仕込んだ NaN を捕まえるのはわかりやすいデモの例にすぎない。実際の訓練クラッシュでは、この検出器が示すのは最初の**症状**であり、根本的

な原因（活性化値の暴走的な増大が最終的にオーバーフローに至るなど）はさらに数層手前から始まっていることもある。

5.8.3.3 勾配の検査

逆伝播（backward pass）も同じ精度で監視できる。8 個の残差ブロックを通るにつれて順伝播の活性化値が増大したのだから、勾配も逆方向に増大するのだろうか。標準正規分布に従う 256 個の入力に対する出力の二乗平均を損失として用い、各残差ブロックの出力における勾配ノルムと、ブロック 3 の最初の線形層における重みおよびバイアスの勾配ノルムを記録してみよう。

`module.register_full_backward_hook(f)` は、`f(module, grad_input, grad_output)` を `backward()` の実行中に呼び出す（最後のブロックから最初のブロックへと、逆トポロジカル順に発火する）。

```
X = torch.randn(256, 20)
grad_norms, handles = [], []
def record(module, grad_input, grad_output):
    grad_norms.append(grad_output[0].norm().item())  # store a number
for block in net[1:-1]:
    handles.append(block.register_full_backward_hook(record))
net.zero_grad()
(net(X) ** 2).mean().backward()
for h in handles:
    h.remove()
# The hooks fire in backward order, last block first
print('block output gradients:',
      ' '.join(f'{g:.3g}' for g in reversed(grad_norms)))
layer = net[3].body[0]  # block 3, first layer
print(f'weight {layer.weight.grad.norm().item():.3g}, '
      f'bias {layer.bias.grad.norm().item():.3g}')
```

```
block output gradients: 1.04 0.83 0.659 0.489 0.396 0.317 0.257 0.194
weight 12.9, bias 2.52
```

8 番目のブロックから 1 番目のブロックへと逆向きにたどると、出力の勾配ノルムはブロックごとに約 1.2× から 1.4× ずつ（スタック全体では 5〜6 倍）増大しており、順伝播での 1.22× の増大とちょうど対称になっている。理由もまったく同じで、逆伝播において各残差ブロックがブランチ側の勾配をスキップ接続（skip connection）の勾配に足し戻すからである。個々のパラメータの勾配ノルムを単独で見てもあまり意味はないが、深さ方向に比較したり訓練ステップにわたって追跡したりすれば、勾配の消失や爆発の挙動が即座に浮かび上がる（演習問題を参照）。

これらのデバッグツールが何を解決し、何を解決しないのかを押さえておこう。異なる 2 台のマシンをビット単位で一致させることはできないし、1 回の実行だけで実験上の改善が本物かどうかを決着させることもできない（そのためには複数のシードにわたる分布を比較する）。しかし、再開可能なチェックポイント（5.6 節）と組み合わせれば、失敗するジョブをクラッシュ直前まで正確に再生し、診断用のフックを取り付けて原因となっている層をピンポイントで特定するために必要な道具はすべて揃ったことになる。

本章の冒頭で取り上げた 80 億パラメータの言語モデルが小さくなったわけではない（fp32 では依然として 32 GB を占める）。しかし、フレームワークがそれをどのように扱うかについて、もはやブラックボックスは何もない。設定からのアーキテクチャ構築、パラメータとバッファの監査、深いスタックの初期化、カスタム層とカスタム勾配の実装、混合精度（mixed precision）での訓練、クラッシュに強いチェックポイントの保存、GPU メモリの管理、そして実行中のモデルの検査は、すべてまったく同じモジュール木の上で動作している。このエンジニアリングの道具立て

を備えた今、実世界のデータの構造に合わせて設計されたアーキテクチャへと進む準備が整った。まずは第 6 章の畳み込みニューラルネットワーク（convolutional neural network）から始めよう。

5.8.4 まとめ

訓練におけるあらゆるランダムな選択（初期化、シャッフル、ドロップアウト、データ拡張）は、シードによって制御される擬似乱数生成器に行き着く。また、専用のジェネレータオブジェクトや明示的なキーを用いれば、無関係なコードによってストリームが乱されるのを防げる。浮動小数点の加算は結合法則を満たさず、並列な GPU のアトミック演算は任意のスレッド順で和を計算するため、ビット単位の再現性を得るには決定論的カーネルも必要であり、しかもそれが成り立つのは同一のマシンとソフトウェアスタックの上だけである。したがって、ビット単位の決定論はデバッグのための手段であり、科学的な結論は複数のシードにわたって成り立たなければならない。最後に、順伝播および逆伝播のフック（ならびに各フレームワークにおける同等の仕組み）を使えば、モデルに手を加えることなく、活性化値のスケール、勾配ノルム、最初に発生した NaN を内部から検査できる。

5.8.5 演習問題

1. [code] **残差の分散収支。**$\mathbf{x}_k$、$\mathbf{b}_k$、$\mathbf{h}_k$ をそれぞれブロック k の入力、body の出力、ブロックの出力とし、各分散は全要素にわたって計算するものとする。1 つのバッチ上で、すべての `ResidualBlock` について、$\mathbf{x}_k$ の分散を `register_forward_pre_hook` で記録し（事前フックは引数のタプルを受け取るためアンパックすること）、$\mathbf{h}_k$ の分散をブロックに対する順伝播フックで、$\mathbf{b}_k$ の分散をその `body` に対する順伝播フックで記録せよ。5.8.3.1 節の一覧における標準偏差が増大するのは、各ブロックが自身の body の出力をストリームに加算するためである。次の等式に注目する。

$$\mathrm{Var}[\mathbf{h}_k] = \mathrm{Var}[\mathbf{x}_k] + \mathrm{Var}[\mathbf{b}_k] \tag{5.12}$$

   すべてのブロックについてこの等式がどの程度よく成り立つかを測定し、この等式が省略している項を答えよ。

2. [code] **シードとデータ読み込み。**●`train_once` を拡張し、`DataLoader`（`shuffle=True`、`num_workers=4`）を通じてデータを読み込むようにせよ。ただし、先頭の `torch.manual_seed(seed)` の呼び出し 1 つはそのまま残すこと。データセットは `__getitem__` 内で各例にノイズを加えるものとし、そのノイズはデータセットのコンストラクタで生成した `np.random.default_rng()` オブジェクトから引く。macOS や Windows のデフォルトである `spawn` で起動されたワーカープロセスはノートブック内で定義されたデータセットクラスをインポートできないため、コードはスタンドアロンのスクリプトとして実行せよ。

   1. 同じシードによる 2 回の実行でシャッフル順序とノイズがそれぞれ一致するか、また 4 つのワーカーが同じノイズを引くかを予測せよ。それぞれのランダム性がどこから引かれるかに基づいて各予測の根拠を述べ、3 つすべてを検証せよ。
   2. 本節の `seed_worker` とシードを設定した `generator` をローダに追加し、2 回の実行が一致すること、および各ワーカーが異なるノイズを引くことを検証せよ。ある例が 2 エポック目に受けるノイズは、1 エポック目と同じになるか。ローダがいつベースシードを引き、いつ初期化関数を呼び出すかという観点から答えを説明せよ。

   PyTorch ドキュメントの マルチプロセスでのデータ読み込み をもとに作成。

3. [code] **発散モニタ。**●5.8.3.2 節の有限性チェックを、ステップごとの訓練モニタに作り変えよ。`net` を再構築し、標準正規分布に従う 256 個の入力からなる固定した 1 つのバッチを

用い、出力と標準正規分布に従うターゲットとの平均二乗誤差を損失として、学習率 1 の SGD で訓練せよ。いずれかの層の出力が最初に非有限になるステップと、そのステップで深さ順で最初に非有限となる層を報告せよ。毎ステップの損失も確認し、いずれかの層の出力より先に損失が非有限になるかどうかを報告せよ。

4. [code] **勾配ノルム。**●`net` を再構築し、残差ブロック内のすべての線形層の重みを 3 倍にし、固定したバッチに対する出力の二乗平均を損失とせよ。
    1. 本節と同様にして各ブロックの出力勾配のノルムを記録せよ。逆伝播の後、各ブロックのパラメータ勾配のノルムを計算し、両方のノルムを深さに対してプロットせよ。
    2. 線形層の重みの勾配を、その層の入力と出力勾配を用いて表せ。この式を使って 2 つの深さ方向プロファイルが異なる理由を説明し、逆伝播信号の爆発を監視するにはどちらの系列を見るべきかを述べよ。

Chapter 6

畳み込みニューラルネットワーク

これまでに構築してきた画像分類器はすべて、画像の空間的な幾何構造を無視し、2 次元のピクセルグリッドを 1 次元のベクトルへと平坦化していた。第 4 章の MLP は、28×28 の Fashion-MNIST 写真を順序のない 784 個の数値のリストとして扱う。そのため、すべての画像のピクセルに同じ固定のランダム置換を施して再訓練しても、並べ替えられた画像が人間の目にはただの砂嵐にしか見えないにもかかわらず、MLP はまったく同じ正解率に到達する。人間はスニーカーを輪郭やテクスチャ、すなわち**隣接する**ピクセルが形作るパターンによって認識しており、輪郭はフレーム内のどこに現れても同じ意味を持つ。

初期の手がかりは神経科学から得られた。ネコとサルを用いた古典的な実験において、Hubel と Wiesel は初期視覚野のニューロンが視野内の局所的な領域、すなわち各ニューロンの**受容野**（receptive field）に現れる特定の向きを持つエッジやバーに応答することを発見した (Hubel and Wiesel 1959, 1962, 1968)。その後、Field (1987) は画像上のあらゆる位置にわたって走査する少数の局所フィルタからなるバンクを用いて、自然画像に対するこれらの応答をモデル化した（Figure 6.1）。

これらの知見に触発され、Fukushima の Neocognitron (Fukushima 1980) は局所的な検出器を空間全体にわたって繰り返される階層的な層へと組織化し、続いて LeCun らは手書きの郵便番号を認識するために、そのようなアーキテクチャを誤差逆伝播法（backpropagation）によってエンドツーエンドで訓練する方法を示した (LeCun et al. 1989, 1995)。さらに、訓練された視覚ネットワークのより深い層は、霊長類の視覚経路における高次の段階と表現上の類似性さえ示す (Kuzovkin et al. 2018)。もっとも、生物学が果たしたのはアーキテクチャ上の着想を与える役割であり、シリコン上の計算に対する設計図ではなかった。

これらのアーキテクチャの中核にあるのが**畳み込み層**（convolutional layer）であり、共有された小さな重みのカーネルを画像のあらゆる近傍にわたってスライドさせる。各出力ユニットは近くのピクセルのみに依存し、入力を平行移動させれば出力もそのまま平行移動する。これにより、局所性（locality）と平行移動同変性（translation equivariance）をネットワークに直接組み込みつつ、パラメータ数を何桁も削減できる。こうした**畳み込みニューラルネットワーク**（convolutional neural network、CNN）は、ImageNet における分水嶺となるブレークスルーを牽引し (Deng et al. 2009; Krizhevsky et al. 2012)、ビジョン Transformer（第 11 章）と並んで今なおコンピュータビジョンの礎であり続けている。

本章では、3 つのステップで第一原理から CNN を導出し構築する。まず、全結合層に局所性と平行移動同変性を課すと、線形演算としては畳み込み（convolution）が唯一の選択肢になることを示す（6.1 節）。続いて 2 次元の相互相関（cross-correlation）を実装し、勾配降下法（gradient descent）によってエッジ検出カーネルを学習させ、畳み込みを行列積へと書き換え（**im2col**）、

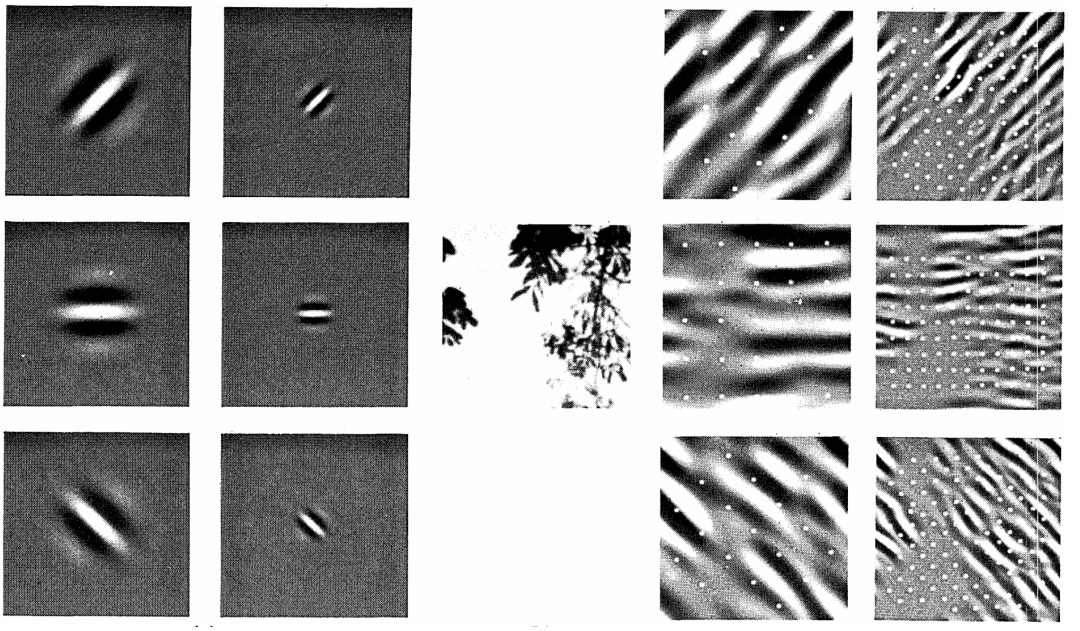


Figure 6.1: 6 つの受容野フィルタと自然画像に対するその応答（Field (1987) より改変）。左のパネルはフィルタ、中央のパネルは入力画像、右のパネルは各フィルタのサイズに比例した間隔でサンプリングされたフィルタ応答を示す。異なるフィルタは、それぞれ異なる局所的なエッジパターンに応答する。

層を積み重ねることで受容野がどのように広がるかをたどる（6.2 節）。次に、パディング・ストライド・ダイレーション（6.3 節）、マルチチャネルのフィルタバンクと 1×1 ボトルネック（6.4 節）、および局所プーリングとグローバルプーリング（6.5 節）を通じて、空間解像度、視野、特徴の表現容量を制御する。最後に、これらの要素を LeNet（6.6 節）へと組み立て、Fashion-MNIST 上で MLP と比較し、学習された最初の層のフィルタを観察する。さらに、対称性から MLP の正解率は変化しないピクセルシャッフル画像上で LeNet を再訓練することで、空間的な帰納バイアス（inductive bias）の寄与を切り分ける。現代の深層畳み込みアーキテクチャは、続く第 7 章で扱う。

参考資料と関連文献

以下の資料は本章の各節に対応しており、それぞれ関連する節を併記している。特記のない限り、すべてオンラインで無料公開されている。

書籍

- Deep Learning (Goodfellow, Bengio & Courville)：無料の HTML 版。第 9 章で疎な相互作用、パラメータ共有、同変性を詳しく展開している（6.1 節、6.5 節）。
- Understanding Deep Learning (Simon J. D. Prince)：無料の PDF 版。第 10 章で畳み込み、ストライド、ダイレーション、受容野を明快な幾何学的図解とともに解説している（6.2 節、6.3 節）。
- Neural Networks and Deep Learning (Michael Nielsen)：オンラインで無料公開。第 6 章で畳み込み層、共有された重み、プーリングを第一原理から導入している（6.2 節、6.5 節）。

講義コースと講義動画

- Stanford CS231n: Deep Learning for Computer Vision：無料の講義ノート。「Convolutional Networks」モジュール（層のサイズ計算、パラメータ数の計算、層の配置パターン）は、本章とあわせて広く利用されているオンライン副教材である（6.3 節、6.4 節）。
- Michigan EECS 498-007: Deep Learning for Computer Vision (Justin Johnson)：無料の講義動画。本章と同じペースで畳み込みとプーリングの計算を順を追って解説している（6.2 節、6.5 節）。

チュートリアル・解説記事・インタラクティブ教材

- A guide to convolution arithmetic for deep learning (Dumoulin & Visin, 2016)：無料、付属のアニメーションあり。パディング、ストライド、ダイレーション、転置畳み込みの視覚的なリファレンスである（6.3 節）。
- CNN Explainer (Wang et al., Polo Club)：活性化値、カーネル、受容野をブラウザ上で対話的に確認できる畳み込みネットワークの可視化ツールである（6.4 節、6.6 節）。
- Computing Receptive Fields of Convolutional Neural Networks (Araujo, Norris & Sim, 2019, **Distill**)：ストライドを含む場合や複数経路を持つ場合も含め、6.2 節で導入する受容野の閉形式の計算式を導出している。
- Conv Nets: A Modular Perspective (Chris Olah)：構造化された重み共有としての畳み込みについて直感を養う、簡潔な定番記事である（6.1 節）。
- Image Kernels Explained Visually (Victor Powell)：人手で設計したカーネルを画像上でスライドさせて試せるインタラクティブツールであり、6.2 節を補完する。
- Feature Visualization (Olah, Mordvintsev & Schubert, 2017, **Distill**)：畳み込みネットワークの各ユニットが何に応答するかを視覚的に探求する記事であり、6.6 節に対応する。

基礎となった論文

- Gradient-Based Learning Applied to Document Recognition (LeCun, Bottou, Bengio & Haffner, 1998, **Proc. IEEE**)：無料の PDF 版。6.6 節の背景にある LeNet-5 の論文である。
- Backpropagation Applied to Handwritten Zip Code Recognition (LeCun et al., 1989, **Neural Computation**)：誤差逆伝播法によってエンドツーエンドで訓練された最初の畳み込みネットワークの論文である（購読制限あり）（6.6 節）。
- Neocognitron (Fukushima, 1980, **Biological Cybernetics**)：畳み込みとプーリングの段階を交互に重ねる構造の、誤差逆伝播法以前における先駆的研究である（購読制限あり）（6.5 節）。
- Receptive fields, binocular interaction and functional architecture in the cat's visual cortex (Hubel & Wiesel, 1962, **J. Physiology**)：無料公開（PMC）。局所受容野と階層的な特徴検出の生物学的起源となった論文である（6.1 節）。

6.1 全結合層から畳み込みへ

第 4 章の多層パーセプトロン（multilayer perceptron、MLP）は、あらゆる画像を順序のない数値ベクトルへと平坦化する。これは表形式データには適しているが、どのピクセル同士が隣接しているかという情報を捨ててしまうため、画像に対しては無駄が多い。1 メガピクセル（1000×1000）のグレースケール写真では、わずか 1,000 ユニットの控えめな隠れ層でさえ $10^6 \times 10^3 = 10^9$ 個の重み（fp32 で 4 GB）を必要とし、元の解像度を保つ 1000×1000 の隠れマップともなれば 10^{12} 個もの重みを要求する。しかし、自然画像は任意の 100 万次元ベクトルとはまったく異なる。隣接するピクセル同士が強く相関する**局所性**（locality）があり、動物のひげのような視覚的特徴がフレーム内のどこに現れても同じ意味を持つ**平行移動同変性**（translation equivariance）を備えているからである。

2 次元のピクセルグリッドを 2 次元の隠れグリッドへ写す密結合層（dense layer）に、これら

2 つの幾何学的原理を直接課すとどうなるだろうか。平行移動同変性だけでも 10^{12} 個の重みは約 4×10^6 個へと激減し、さらに各隠れユニットが参照する範囲を小さな局所近傍に制限すれば、重みは数百個にまで削減される。この両方の制約を課して残るものこそが、**畳み込み層**（convolutional layer）である。

6.1.1 同変性と不変性

子どもの遊びである「ウォーリーをさがせ !」（Figure 6.2）は、視覚認識における本質的な課題を的確に捉えている。ごちゃごちゃした各場面には気を散らすものが溢れており、ウォーリーは思いがけない場所に隠れているが、**ウォーリーがどう見えるか**は**どこに立っているか**に依存しない。したがって、単一のウォーリー検出器で画像全体を走査し、あらゆる位置でまったく同じ規則を用いて、各パッチがどれだけ強くウォーリーと一致するかのスコアを計算できる。現代の物体検出（object detection）(Girshick et al. 2014) やセマンティックセグメンテーション (Long et al. 2015) のパイプラインは、この考え方を直接の土台としている。

Figure 6.2: ウォーリーを見つけられるだろうか（画像提供：William Murphy (Infomatique)）。

この直観から、視覚ネットワークの初期の層に対する 2 つの設計指針が導かれる。

1. 同じ局所パッチに対しては、それがどこに現れても同一の応答を返すべきである。すなわち、入力をシフトさせれば出力も同じ量だけシフトする。
2. 画像の離れた領域は気にせず、小さな局所近傍のみに着目すべきである。これが**局所性**の原理である。

後続の層はこれらの局所的な応答を組み合わせることで、段階的により大きく抽象的なパターンを構成し、最終的に画像全体の予測へと至る。

第 1 の原理を定式化しよう。シフト作用素 T_v を、画像をオフセット $v = (v_1, v_2)$ だけ平行移動させる作用素とし、ピクセル (i, j) を $(i + v_1, j + v_2)$ へ移すことで $[T_v \mathbf{X}]_{i,j} = [\mathbf{X}]_{i-v_1, j-v_2}$ が成り立つものとする。画像から画像への写像 f は、入力をシフトさせると出力もまったく同じベクトルだけシフトするとき**平行移動同変**であり、一方で画像からラベルへの写像は、入力をシフトさせても出力がまったく変化しないとき**平行移動不変**（translation invariant）である。すなわち、次式が成り立つ。

$$\begin{aligned} f(T_v \mathbf{X}) &= T_v f(\mathbf{X}) \quad \text{(equivariance)}, \\ f(T_v \mathbf{X}) &= f(\mathbf{X}) \quad \text{(invariance)}. \end{aligned} \tag{6.1}$$

Figure 6.3 は、平行移動同変性を可換図式として示したものである。ここで f は、画像のあらゆるウィンドウに対して小さなテンプレートを照合する演算であり、これこそが本節でこれから導出しようとしている演算にほかならない。先にシフトしてから照合しても、先に照合してからシフトしても、まったく同じマップが得られる。

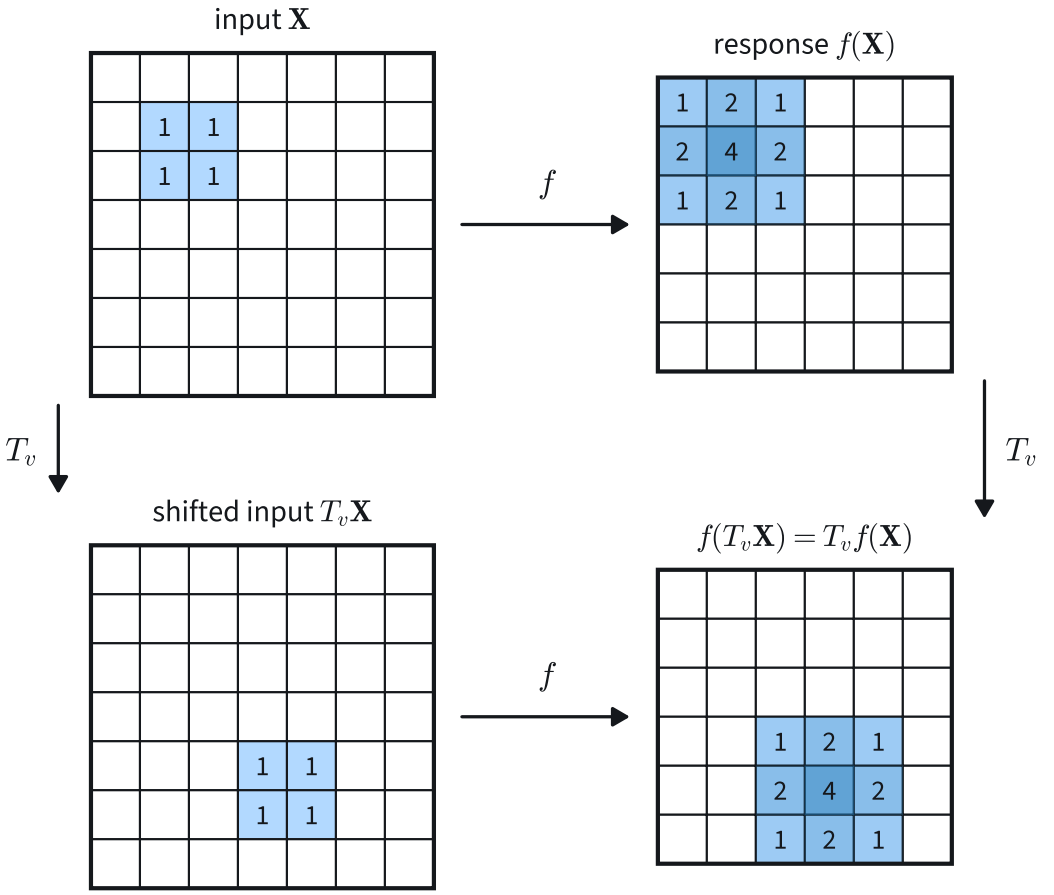


Figure 6.3: 平行移動同変性。入力にはすべて 1 からなる 2×2 のパッチが含まれており、f はすべての 2×2 ウィンドウをそのパッチと比較するため、ウィンドウがパッチ全体を覆う位置で応答は 4 になる。入力を $v = (3, 2)$ だけシフトしてから f を適用したマップ（下段）は、先に f を適用してから応答をシフトしたマップ（右列）と一致する。

ここで、明確な役割分担に注目しよう。**同変性**はパターンが**どこに**現れるかを保持するのに対し、**不変性**は位置情報を捨てる。ウォーリー検出器は、そのスコアマップが場面内のウォーリーを追跡できるよう同変であるべきだ。一方、「この画像にウォーリーはいるか」というイエス・ノーの問いへの回答は不変であるべきであり、これは最後に同変なマップを空間全体にわたって集約する（たとえば最大値をとる）ことで達成される。

6.1.2 全結合層への制約

重み行列全体を図示できる 1 次元で考えると、これら両方の制約はきわめて明快になる。8 サンプルの信号 $x_0, \ldots, x_7$ が全結合層（fully connected layer）によって 8 個の隠れユニットへ

$h_i = u_i + \sum_k W_{i,k} x_k$ と写される場合を考えよう。この 8×8 の重み行列は 64 個の独立な成分を持ち、Figure 6.4 のパネル (a) に示されている。

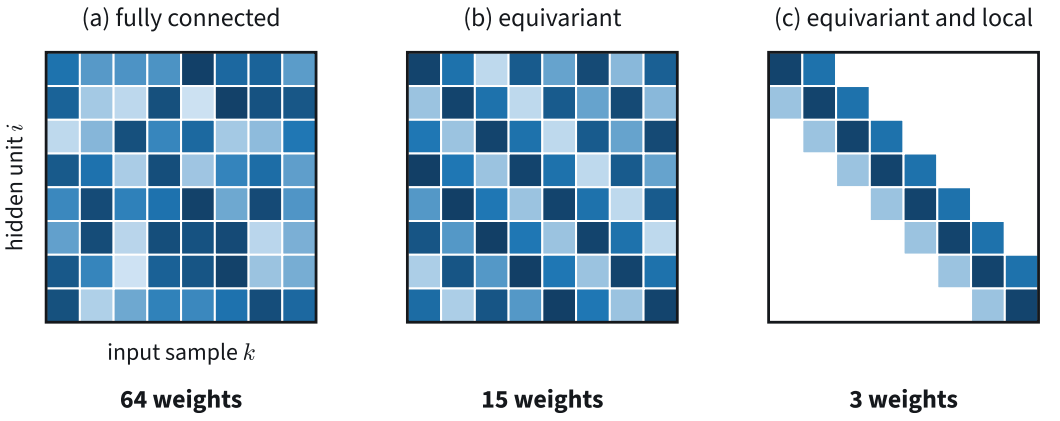


Figure 6.4: 8 サンプル（列）の 1 次元信号を 8 個の隠れユニット（行）へ写す線形層の重み。同じ色は等しい重みを表し、白は 0 を表す。(a) 全結合：すべての重みが自由パラメータである。(b) 平行移動同変：重みは入力と出力のオフセットのみに依存するため、行列は各対角線に沿って一定となる。(c) 同変かつ局所的：オフセット -1、0、1 のみが非ゼロの重みを持つ。

平行移動同変性は $\mathbf{W}$ にどのような条件を課すだろうか。いったんバイアスを無視して（後述の証明で、すべてのバイアスが一致しなければならないことを示す）、サンプル k にのみ単一の 1 を持つ単位インパルスを入力すると、層は列番号 k に対応する $\mathbf{W}$ の列を出力する。このインパルスを $k+1$ へシフトさせると、層は第 $k+1$ 列を出力する。同変性により、第 $k+1$ 列は第 k 列を 1 行下にシフトしたものと等しくなければならない。すなわち $W_{i+1,k+1} = W_{i,k}$ である。その結果、パネル (b) のように、$\mathbf{W}$ はすべての対角線に沿って一定でなければならない（**テプリッツ**行列）。各対角線は単一の相対オフセット $a = k - i$ に対応するため、64 個の自由な重みはわずか 15 個（-7 から 7 までのオフセット）へと縮約される。

局所性は、残った 15 個の重みのほぼすべてを刈り込む。各隠れユニットが距離 1 以内の近傍のみを参照するように制限すると、$\{-1, 0, 1\}$ 以外のすべてのオフセットが 0 になり、パネル (c) のようにすべての行で共有されるわずか 3 個の重みだけが残る。これにより、この層は重み 3 個からなる 1 つの小さなテンプレートで信号全体を走査することになる。これこそが導出全体の縮図であり、2 次元画像への拡張は 2 つ目の空間インデックスを追加するだけでよい。

画像の場合、入力 $\mathbf{X}$ と隠れ表現 $\mathbf{H}$ はどちらも 2 次元グリッドであり、入力ピクセル $[\mathbf{X}]_{k,l}$ が (k,l) に、隠れユニット $[\mathbf{H}]_{i,j}$ が (i,j) に位置する。すべての隠れユニットをすべてのピクセルに結合すると、重みは 4 階テンソル W となり、これにバイアス行列 $\mathbf{U}$ が加わる。

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l}.$$

両方の制約とも、隠れユニット (i,j) に対するピクセルの**相対的な**位置に依存するため、オフセット $a = k - i$ および $b = l - j$ によってインデックスを張り替え、$[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$ と定義する。

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.2}$$

ここで、a と b は正と負の両方のオフセットにわたる。このインデックスの張り替えは純粋に表記上のものであり、W を対角線に沿って読み替えているだけで、パラメータはまだ削減されていない。1000×1000 の画像を 1000×1000 の隠れグリッドへ写す場合、この単一の層は依然として $10^6 \times 10^6 = 10^{12}$ 個の重みを必要とする。

6.1.2.1 平行移動同変性

1 次元の場合と同様に、Equation 6.2 の形をした層が平行移動同変となるのは、その重みとバイアスが位置 (i, j) に依存しないとき、かつそのときに限る。重みの共有から同変性が導かれることの確認は 1 行で済むが、重要なのはその逆である。同変性を課すと線形な選択肢がほかに残らないため、畳み込みが必然的に導かれるからである。

境界（シフトによって有限フレームの端からピクセルが押し出される箇所）を見通しよく扱うため、台（support）の外側を 0 でパディングした無限グリッド上に画像を埋め込み、情報がグリッドの外へ失われないようにする。有限サイズの画像では、同変性は境界から離れた領域で厳密に成り立つ。この詳細については 6.3 節で扱う。

命題： 入力（非ゼロのピクセルを有限個のみ持つ）と隠れ表現が無限グリッド上にあるとする。このとき、Equation 6.2 の層があらゆるシフト v に対して平行移動同変となるのは、あるスカラー u と行列 $\mathbf{K}$ が存在して、$[\mathbf{U}]_{i,j} = u$ かつ $[\mathsf{V}]_{i,j,a,b} = [\mathbf{K}]_{a,b}$ がすべての i, j, a, b について成り立つとき、かつそのときに限る。

証明： 重みもバイアスも (i, j) に依存しないならば、入力を v だけシフトさせると出力もそのまま v だけシフトする。

$$
\begin{aligned}
[f(T_v \mathbf{X})]_{i,j} &= u + \sum_a \sum_b [\mathbf{K}]_{a,b} [\mathbf{X}]_{i - v_1 + a, j - v_2 + b} \\
&= [f(\mathbf{X})]_{i - v_1, j - v_2} = [T_v f(\mathbf{X})]_{i,j}.
\end{aligned}
$$

逆に、この層が同変であると仮定する。すべて 0 の画像はあらゆるシフトに対して不変であり、$\mathbf{U}$ へと写されることから、$\mathbf{U} = T_v \mathbf{U}$ がすべての v について成り立つ必要がある。したがって、バイアスは定数 u でなければならない。両辺から u を引けば、線形部分単独でも同変であることがわかる。ここで、(k, l) のみが単一の 1 で他がすべて 0 である画像を入力すると、(i, j) における応答は $[\mathsf{W}]_{i,j,k,l}$ となる。このインパルスを v だけシフトさせると $(k + v_1, l + v_2)$ へ移り、同変性から $[\mathsf{W}]_{i,j,k+v_1,l+v_2} = [\mathsf{W}]_{i-v_1,j-v_2,k,l}$ が要請される。$(k, l) = (a, b)$ かつ $v = (i, j)$ とおけば、$[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b} = [\mathsf{W}]_{0,0,a,b}$ が得られ、これは (i, j) に依存しない。■

要するに、単一の明るいピクセルでこの層を探ることで、すべての位置がまったく同じ重みを共有せざるを得なくなり、層は次式へと縮約される。

$$
[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a, j+b}. \tag{6.3}
$$

これにより、すべての隠れユニットは自身の座標を中心として同じパターン $\mathbf{K}$ を適用するようになる。ゼロパディングされた 1000×1000 の画像では、非ゼロのピクセルに触れうるのはオフセット $a, b \in [-999, 999]$ のみであるため、重みの数は 10^{12} 個から $1999^2 \approx 4 \times 10^6$ 個へと激減する。

6.1.2.2 局所性

単一のフィルタに 400 万個もの重みがあるのは依然として多すぎるうえ、その大半は隠れユニットを画像の反対側にある遠いピクセルと結びつけている。そこで**局所性**（$[\mathbf{H}]_{i,j}$ が各軸に沿って距離 Δ 以内のピクセルのみに依存するという条件）を課すと、$[\mathbf{K}]_{a,b} = 0$ が $|a| > \Delta$ または $|b| > \Delta$ のときに成り立ち、次式が得られる。

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.4}$$

これにより、各隠れユニットはサイズ $(2\Delta+1) \times (2\Delta+1)$ の (i,j) を中心とする局所ウィンドウ上で重み付き和を計算するようになり、使用するパラメータはわずか $(2\Delta+1)^2$ 個の共有された重みと 1 個のバイアスだけになる。すなわち、重みの数は $\Delta = 1$（3×3 ウィンドウ）なら 9 個、$\Delta = 5$ なら 121 個である。通常 Δ は 10 よりずっと小さいため、隠れマップの空間解像度を完全に保ったまま、4×10^6 個の重みが数十〜数百個の重みへと置き換わる。

Equation 6.4 は、**畳み込みニューラルネットワーク**（convolutional neural network、CNN）の基本構成要素である**畳み込み層**を定義している。共有される行列 $\mathbf{K}$ は**畳み込みカーネル**（convolution kernel）または**フィルタ**と呼ばれる。空間的および時間的な重み共有（weight sharing）の先駆的な研究としては、音声認識のための時間遅れニューラルネットワーク (Waibel et al. 1989) や、Zhang et al. (1988) によるシフト不変ネットワークが挙げられる。

重要な点として、パラメータ数が 10^{12} から $(2\Delta+1)^2$ へと縮小したのは、仮説クラス（hypothesis class）を局所的かつ平行移動同変な写像に制限した結果である。この組み込まれた**帰納バイアス**（inductive bias、4.5 節）により、エッジやテクスチャのように特徴が局所的で位置に依存しない場合にはサンプル効率が劇的に向上する。もっとも、単一の層だけでは、同一のパッチを絶対位置のみによって区別することはできない（たとえば、パスポート写真の上部付近にある目と、下の方にある鼻孔の区別など）。非線形性を挟んで複数の層を積み重ねることで、より深い層のユニットは段階的に広い領域にわたる文脈を統合できるようになる。

6.1.3 畳み込みと相互相関

厳密に言うと、深層学習における「畳み込み」という名称は少し不正確である。数学では、2 つの 1 次元関数の離散**畳み込み**（convolution）は $(f * g)(i) = \sum_a f(a) g(i-a)$（$\mathbb{R}^d$ 上では積分）で定義され、f と、**反転**させてシフトした g のコピーとの重なり具合を測る。2 次元では次のようになる。

$$(f * g)(i,j) = \sum_a \sum_b f(a,b) g(i-a, j-b). \tag{6.5}$$

これを Equation 6.4 と比較してみよう。そこでは $\mathbf{X}$ のインデックスが $(i+a, j+b)$ となっており、$(i-a, j-b)$ ではない。$-a$ を a に、$-b$ を b に代入すると、Equation 6.4 は $\mathbf{X}$ と反転させたカーネル $[\mathbf{K}']_{a,b} = [\mathbf{K}]_{-a,-b}$ との数学的な畳み込みであることがわかる。

$$\sum_a \sum_b [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a,j+b} = \sum_a \sum_b [\mathbf{K}']_{a,b} [\mathbf{X}]_{i-a,j-b} = (\mathbf{K}' * \mathbf{X})(i,j). \tag{6.6}$$

この反転操作はカーネルを両方の軸に沿って逆向きにするため、左上の重みが右下の重みへと移る（Figure 6.5）。

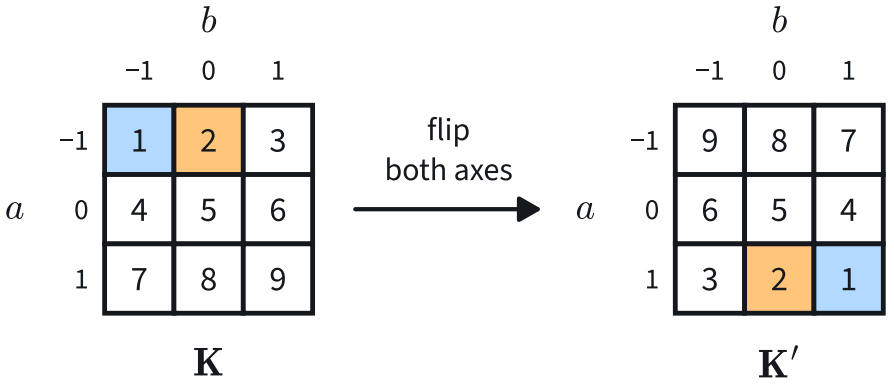


Figure 6.5: 3×3 カーネル $\mathbf{K}$ とその反転コピー $\mathbf{K}'$。行と列はオフセット a と b でインデックス付けされている。反転によって両方の軸が逆向きになり、色付きの 2 つの重みはオフセット $(-1,-1)$ と $(-1,0)$ から $(1,1)$ と $(1,0)$ へと移動する。$\mathbf{K}$ との相互相関は $\mathbf{K}'$ との畳み込みに等しい。

反転を伴わないこの演算は、正しくは**相互相関**（cross-correlation）と呼ばれる。CNN のカーネルはデータから学習されるため、この違いは表現力にまったく影響しない。すなわち、$\mathbf{K}$ との相互相関を計算する層は、$\mathbf{K}'$ との畳み込みを計算する層とまったく同じ関数族を学習する。深層学習フレームワークは相互相関を計算してそれを畳み込みと呼んでおり、本書でも全体を通してこの標準的な慣習に従う。

6.1.4 チャネル

ここまでは、入力と隠れマップの両方を単一のグレースケールグリッドとして扱ってきた。実際のカラー写真は 1 ピクセルあたり 3 つの輝度値（赤、緑、青）を持つため、画像は 3 階テンソル X となり、その形状は $3 \times \text{height} \times \text{width}$、各成分は $[\mathsf{X}]_{c,i,j}$ で表される。先頭の**チャネル**軸は各空間ピクセルに特徴ベクトルを対応付け、カーネルは局所的な空間ウィンドウとすべての入力チャネルの両方にわたって和をとる。

同様に重要な点として、単一のカーネルは単一の応答マップ（Figure 6.6 のウォーリーのスコアマップのようなもの）しか生成せず、1 つの視覚パターンしか捉えられない。豊かな視覚表現を得るには、方向を持つエッジ、コーナー、色のコントラスト、テクスチャなど、多数のパターンを並列に検出する必要があるため、隠れ表現 H もまた**出力チャネル**（output channel）軸に沿って積み重ねられた 3 階テンソルとなる。各出力チャネル o はそれぞれ固有の学習されたカーネルスライスとバイアス u_o を持ち、これによって K は 4 階テンソルとなる。

$$[\mathsf{H}]_{o,i,j} = u_o + \sum_c \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathsf{K}]_{o,c,a,b} [\mathsf{X}]_{c,i+a,j+b}. \tag{6.7}$$

ここで、c は入力チャネル、o は出力チャネルのインデックスであり、本章全体を通してこの「出力チャネルが先、入力チャネルが後」という順序の慣習を用いる。各出力チャネルは特定の学習された特徴がグリッド上のどこに現れるかを表すため、隠れ層のチャネルは**特徴マップ**（feature map）とも呼ばれ、そのスタック全体が次の層への入力となる。入力チャネル数を c_i、出力チャネル数を c_o とすると、この層は $(2\Delta+1)^2 c_\text{i} c_\text{o}$ 個の重みと c_o 個のバイアスを必要と

する。したがって、RGB 画像を 64 枚の特徴マップへ写す 3×3 の層が持つパラメータはわずか $9 \cdot 3 \cdot 64 + 64 = 1{,}792$ 個であり、画像の解像度にはまったく依存しない。

Figure 6.6: ウォーリー検出器：ウォーリーに一致するカーネルの応答は、ウォーリーの位置でピークに達する（画像提供：William Murphy (Infomatique)）。

この導出には 2 次元に固有の要素は一切ない。まったく同じ論理から、音声サンプルやテキストのトークン上をスライドする 1 次元畳み込みや、動画フレームや 3 次元医用スキャン上をスライドする 3 次元畳み込みが導かれる。

Equation 6.7 は、本章の残りの節すべての基礎となる。ただし、これは理想化された無限グリッド上で導出されたものであるため、実用上は 3 つの課題が残っている。すなわち、有限な画像境界の扱い、より深い層のユニットが見渡せる画像領域の拡大、そして空間的な特徴マップを画像全体の単一の判定（たとえば、ウォーリーがシーン内の**どこかに**現れているかどうか）へと集約する方法である。しかしその前に、まずは Equation 6.4 をコードとして実装してみよう。

6.1.5 まとめ

画像を構造のないベクトルへと平坦化すると、密結合層は扱いきれないほど巨大になる（1 メガピクセルの画像を同サイズの隠れグリッドへ写すのに 10^{12} 個の重みが必要となる）。画像に関する 2 つの自然な仮定がこのパラメータ数を数百個へと激減させる。すなわち、**平行移動同変性**（無限グリッド上で証明したとおり）によってすべての空間位置が 1 つのカーネルを共有し、**局所性**によってそのカーネルが小さなウィンドウに制限される。結果として得られる層は相互相関（深層学習では慣例的に畳み込みと呼ばれる）を計算し、複数の入力チャネルと出力チャネルを重ねることで、パラメータ数を画像サイズに依存させないまま単一の層で多数の視覚パターンを同時に検出できるようになる。これらの利点は帰納バイアスから直接もたらされるものである。畳み込み層は、位置に依存する任意の写像を表現する自由度を手放す代わりに、空間的構造を持つデータに対する高い汎化性能を獲得しているのである。

6.1.6 演習問題

1. **円環上の重み共有**。最後のサンプルの次が再び最初のサンプルに戻る周期グリッド上の 4 サンプルの信号（シフトによって情報が一切失われない）に対して、Figure 6.4 の議論を繰り返せ。
    1. 各サンプルの位置に順番に単一の 1 を層へ入力し、平行移動同変な層の 4×4 重み行列を書き下せ。自由な重みはいくつあるか。周囲を 0 で囲まれた 4 サンプルの場合の

$2 \cdot 4 - 1 = 7$ 個のオフセットと比較せよ。
2. オフセット -1、0、1 の局所性を課せ。行列のどの成分が非ゼロとなり、そのうち図のパネル (c) に対応する成分がないものはどれか。

2. **畳み込みの可換性**。
1. Equation 6.5 の畳み込みが可換であること、すなわち $f * g = g * f$ を証明せよ。
2. 相互相関 $(f \star g)(i,j) = \sum_a \sum_b f(i+a, j+b)\, g(a,b)$（入力 f とカーネル g に対する Equation 6.4 の演算）を定義する。$(g \star f)(i,j)$ を $f \star g$ の値を用いて表し、その式から相互相関が可換であるかどうかを判定せよ。

3. **音声に対する畳み込み**。音声データは多くの場合、サンプルの 1 次元系列として表現される。
1. Equation 6.4 の 1 次元版を導出せよ。
2. 時間方向に局所的かつ平行移動同変な層が適している音声タスクを 1 つ、適していない音声タスクを 1 つ挙げよ。それぞれの理由を説明せよ。
3. スペクトログラムは、録音を時間と周波数の 2 軸からなる 2 次元配列として表現する（周波数はヘルツ単位の線形スケールとする）。それぞれの軸について、平行移動同変性が妥当な仮定であるかどうかを述べ、その軸に沿ったシフトが音にどのような変化を与えるかという観点から理由を説明せよ。さらに、周波数軸が対数スケールである場合についても同じ問いに答えよ。

4. **同変性と不変性**。畳み込み分類器が、Equation 6.7 の層と非線形性を交互に積み重ねて特徴マップ $h(\mathsf{X})$ を計算し、続いて予測 $g(h(\mathsf{X}))$ をヘッド g によって計算するものとする。
1. 最後の列の次が再び最初の列になるよう折り返す周期グリッドを仮定すると、この上では命題の議論により h は平行移動同変となる。ここで g は $h(\mathsf{X})$ の各チャネルをすべての位置にわたって平均するものとする。Equation 6.1 を用いて、この予測が平行移動不変であることを示せ。
2. ある画像には 3×3 グリッドの 9 つのセルのいずれか 1 つに単一の数字が描かれており、どのセルに描かれているかを当てるタスクを考える。前半の分類器がこのタスクを解けない理由を説明せよ。また、ヘッド g として、同じ特徴マップ $h(\mathsf{X})$ からこのタスクを解けるものを記述せよ。

5. **回転同変性**。●本節の命題では、シフトに対する同変性から重み共有を導出した。回転についても同様の問いを考えよう。命題と同じ無限グリッド上で、R を原点まわりに画像を 90° 回転させる作用素 $[R\mathbf{X}]_{i,j} = [\mathbf{X}]_{-j,i}$ とする。
1. Equation 6.4 の畳み込み層において、$f(R\mathbf{X}) = Rf(\mathbf{X})$ がすべての画像 $\mathbf{X}$ について成り立つのは、$[\mathbf{K}]_{a,b} = [\mathbf{K}]_{-b,a}$ がすべてのオフセット (a,b) について成り立つとき、かつそのときに限ることを示せ。
2. $\Delta = 1$ のとき、このようなカーネルは自由な重みをいくつ持つか。評価対象のピクセルを中心に画像を 90° 回転させてもこのカーネルの応答が変化せず、したがって垂直エッジと水平エッジを区別できないことを示せ。
3. 1 つのカーネルに制約を課す代わりに、制約のない 1 つのカーネル $\mathbf{K}$ をそれぞれ 0°、90°、180°、270° 回転させたコピーをカーネルとして持つ 4 つの出力チャネルを用いるとする。入力を回転させると、各出力マップが回転するとともに 4 つのチャネルが巡回置換されることを示せ。前半の制約付きカーネルでは失われていたが、この層では保持されているものは何か。

6.2 画像に対する畳み込み

6.1 節では、平行移動同変性（translation equivariance）と局所性（locality）という 2 つの幾何学的原理から、畳み込み層（convolutional layer）を数式の上で導出した。ここでは、その式が画像上で実際に何を計算しているのか、そして勾配降下法によってデータから有用なカーネルを自動的に見つけ出せるのかを見ていく。

明るい領域と暗い領域が接する垂直方向のエッジを見つけることを考えよう。各ピクセルから右隣のピクセルを引くと（$[\mathbf{X}]_{i,j} - [\mathbf{X}]_{i,j+1}$）、一様な領域の内部では 0 になり、白から黒へ変化する箇所では $+1$、黒から白へ戻る箇所では -1 になる。2 つの重みからなる検出器 $(1, -1)$ を画像のすべての有効な位置にわたって走査すると、**エッジマップ**が得られる。本節では、2 次元相互相関（two-dimensional cross-correlation）を実装し、手作業で設計したエッジ検出器を試し、勾配降下法によってデータから直接 $(1, -1)$ を**学習**させる。さらに、畳み込みが単一の行列積（**im2col**）に帰着されることを示し、層を積み重ねることでユニットの受容野（receptive field）がどのように広がるかを導出する。

6.2.1 相互相関

6.2.1.1 演算の定義

ひとまずチャネルのことは脇に置き、入力 $\mathbf{X}$ を $n_\mathrm{h} \times n_\mathrm{w}$ 行列、カーネル $\mathbf{K}$ を $k_\mathrm{h} \times k_\mathrm{w}$ 行列とする。このカーネルは、形状 $k_\mathrm{h} \times k_\mathrm{w}$ のスライディング**ウィンドウ**を定める。ウィンドウの左上隅を (i, j) に置き、重なった入力パッチと $\mathbf{K}$ を要素ごとに掛けてその総和をとることで、位置 (i, j) の出力要素が得られる。

$$[\mathbf{Y}]_{i,j} = \sum_{a=0}^{k_\mathrm{h}-1} \sum_{b=0}^{k_\mathrm{w}-1} [\mathbf{K}]_{a,b} \, [\mathbf{X}]_{i+a,j+b}. \tag{6.8}$$

これが $\mathbf{X}$ と $\mathbf{K}$ の **2 次元相互相関**であり、コードに合わせて 0 始まりの添字を用い、$\mathbf{Y} = \mathbf{X} \star \mathbf{K}$ と表記する。6.1 節ではオフセットを $-\Delta$ から Δ まで対称に中心化したが、左上隅を基準点とする方式は一定のシフト Δ だけが異なるにすぎず、中心ピクセルを持たない偶数サイズのカーネル（2×2 カーネルや先ほどの 1×2 エッジ検出器など）も自然に扱える。

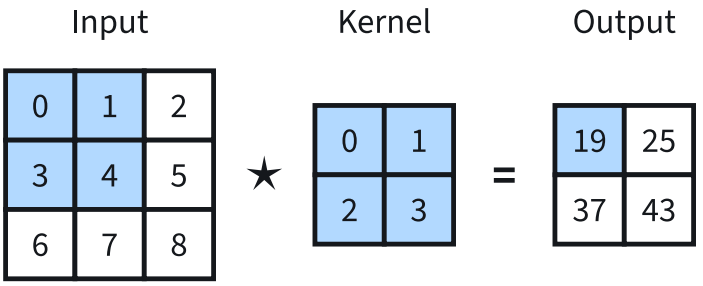


Figure 6.7: 2 次元相互相関。網掛け部分は、最初の出力要素と、その計算に使われる入力要素およびカーネル要素である（$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$）。

Figure 6.7 では、2×2 カーネルを 3×3 の入力上で左から右、上から下へと 1 ピクセルずつスライドさせている。ウィンドウは 4 つの位置にぴったり収まり、次の 2×2 の出力が得られる。

$$\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19, \\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25, \\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37, \\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.
\end{aligned}$$

カーネルは入力の内部に完全に収まらなければならないため、出力は各軸に沿ってわずかに縮小する。幅 k のウィンドウを長さ n の範囲に収めると有効な位置は $n-k+1$ 個残るため、出力の形状は次のようになる。

$$(n_{\mathrm{h}}-k_{\mathrm{h}}+1)\times(n_{\mathrm{w}}-k_{\mathrm{w}}+1). \tag{6.9}$$

したがって、3×3 の入力と 2×2 カーネルからは 2×2 が、6×8 の画像と 1×2 カーネルからは 6×7 が、28×28 の Fashion-MNIST 画像と 5×5 カーネルからは 24×24 が得られる。空間次元を保ちたい場合は、入力の周囲を 0 の枠で囲む（**パディング**）。これについては、ストライドやダイレーションとあわせて 6.3 節で詳しく調べる。

6.2.1.2 実装

すべての出力座標を回る二重ループを用いて、Equation 6.8 を `corr2d` として直接実装する。

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = d2l.reduce_sum((X[i: i + h, j: j + w] * K))
    return Y
```

`corr2d` を Figure 6.7 の手計算と照合するため、3×3 の入力テンソル `X` と 2×2 のカーネル `K` を作成し、`corr2d(X, K)` を評価する。

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

出力は Figure 6.7 と要素ごとに一致する（また、6.1 節で示したように、反転させたカーネル $\mathbf{K}'$ を用いる数学的に厳密な畳み込みもまったく同じ層の族を学習するため、本書でも深層学習の標準的な用語法に従い、相互相関を「畳み込み」と呼ぶ）。

6.2.2 畳み込み層

6.2.2.1 層の定義

2 次元畳み込み層は、入力と学習されるカーネルの相互相関を計算し、スカラーのバイアスを加える。MLP の重み（第 4 章）と同様に、カーネルは $\mathcal{N}(0,0.01^2)$ から初期化し、バイアスは 0 に初期化する。以下の `Conv2D` は両方のパラメータを登録し、順伝播（forward pass）で `corr2d` を呼び出す。高さ h、幅 w のカーネルを持つ層は、$h\times w$ 畳み込み層と呼ばれる。

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
```

```
            self.weight = nn.Parameter(0.01 * torch.randn(kernel_size))
            self.bias = nn.Parameter(torch.zeros(1))

        def forward(self, x):
            return corr2d(x, self.weight) + self.bias
```

全結合層とは異なり、畳み込み層のパラメータはカーネルサイズのみに依存するため、固定された入力解像度に縛られない。まったく同じ 2×2 の層（パラメータ数は合計 5 個）で、先ほどの 3×3 テンソル `X` と 6×8 の画像の両方を処理でき、それぞれ形状 2×2 と 5×7 の出力を返す。

```
layer = Conv2D((2, 2))
layer(X).shape, layer(d2l.ones((6, 8))).shape
```

```
(torch.Size([2, 2]), torch.Size([5, 7]))
```

6.2.2.2 エッジ検出

ここで、冒頭で述べた垂直エッジ検出器を 6×8 の合成画像で試してみよう。この画像は、中央の 4 列が黒（`0`）で外側の列が白（`1`）になっている。

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

次の 1×2 カーネルを定義する。

$$\mathbf{K} = \begin{bmatrix} 1 & -1 \end{bmatrix}. \tag{6.10}$$

すべての位置 (i, j) において、このカーネルは水平方向の**有限差分**（finite difference）である $[\mathbf{X}]_{i,j} - [\mathbf{X}]_{i,j+1}$ を計算する。これは輝度の水平方向の導関数を（符号を除いて）離散近似したものであり、平坦な領域ではゼロになり、画素の強度が急変する箇所で鋭く立ち上がる。

```
K = d2l.tensor([[1.0, -1.0]])
```

6×8 の縞模様の画像 `X` と 1×2 の水平差分カーネル `K` の相互相関を `corr2d(X, K)` で計算すると、6×7 のエッジマップ `Y` が得られる。

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
```

```
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

予想どおり、`Y` は白から黒への境界に沿って +1、黒から白への境界に沿って -1 を出力し、それ以外の場所ではすべて `0` になる。まったく同じカーネルを転置した画像（縦縞を横縞に変えたもの）に適用すると、出力はすべて 0 になる。

```
corr2d(d2l.transpose(X), K)
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

明らかに、`[1, -1]` は垂直方向のエッジにのみ反応する。水平方向のエッジや斜め線、コーナー、テクスチャを検出するには、まったく別のカーネルが必要である。

6.2.2.3 カーネルの学習

欲しい特徴が正確にわかっていれば `[1, -1]` を手作業で作るのは簡単だが、深いネットワーク全体にわたる何千もの中間フィルタを人手で指定することは誰にもできない。では、勾配降下法はカーネル `[1, -1]` を入出力の組 `(X, Y)` だけから復元できるだろうか。各フレームワークの組み込み 2 次元畳み込み層（バイアスは省略し、重みは $\mathcal{N}(0, 0.01^2)$ から初期化）を用い、全 42 個の出力要素にわたる二乗誤差（squared error）の和を、10 回の勾配ステップにわたり学習率 0.03 で最小化してこれを検証する。組み込みの層はバッチ軸とチャネル軸を含む 4 次元テンソルを入力として受け取るため（6.4 節）、それに合わせて `X` と `Y` の形状を変換する。

```
conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
nn.init.normal_(conv2d.weight, std=0.01)
# Layers take (example, channel, height, width): one example, one channel
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    l = ((conv2d(X) - Y) ** 2).sum()
    conv2d.zero_grad()
    l.backward()
    with torch.no_grad():
        conv2d.weight -= lr * conv2d.weight.grad
    if i % 3 == 0:
        print(f'step {i}, loss {l:.3f}')
```

```
step 0, loss 12.216
step 3, loss 0.840
step 6, loss 0.058
step 9, loss 0.004
```

初期状態では損失は 12 付近にある（ゼロに近いランダムなカーネルは、12 個ある ± 1 のエッジピクセルすべてを約 1 だけ外すためである）。しかし、わずか 9 ステップ以内に損失は 0.01 未満まで低下する。学習されたカーネルを調べると、何が発見されたかを確認できる。

```
d2l.reshape(conv2d.weight.data, (1, 2))
```

```
tensor([[ 0.9891, -0.9876]])
```

どちらの要素も `[1, -1]` から約 0.02 以内に収まっている。このループにはエッジに関する事前知識は何も組み込まれていない。損失の信号を与えるだけで、勾配降下法が自力でフィルタを見つけ出したのである。ここでは単一の層に `Y` を直接教師信号として与えたが、深い CNN では、誤差逆伝播法（backpropagation）によって最終的なタスク損失だけからすべての層のカーネルがエンドツーエンドで学習される。

6.2.3 行列積としての畳み込み

`corr2d` の入れ子ループは数式の意味を理解しやすいものの、Python で出力ピクセルを 1 つずつ処理するのでは実用的なワークロードには遅すぎる。古典的な解決策であり、現在でもライブラリが選択するアルゴリズムの 1 つとなっているのが、相互相関を 1 回の大きな行列積に書き換える方法である。行列積こそ、GPU が最高速度で実行できる演算にほかならない。Figure 6.7 の各出力要素は、平坦化した $k_\mathrm{h} k_\mathrm{w}$ 要素のカーネルと、対応する平坦化した入力ウィンドウとのドット積にすぎないことに注目しよう。$(n_\mathrm{h} - k_\mathrm{h} + 1)(n_\mathrm{w} - k_\mathrm{w} + 1)$ 個すべてのウィンドウを取り出して行として積み重ねるとパッチ行列が得られ、これに平坦化したカーネルを掛ければ相互相関全体を一度に計算できる。この標準的なメモリ配置の変換は **im2col**（「image to columns」の略だが、ここでは行に並べている）と呼ばれる。Figure 6.8 は、Figure 6.7 の例に対してこの変換を実行した様子である。

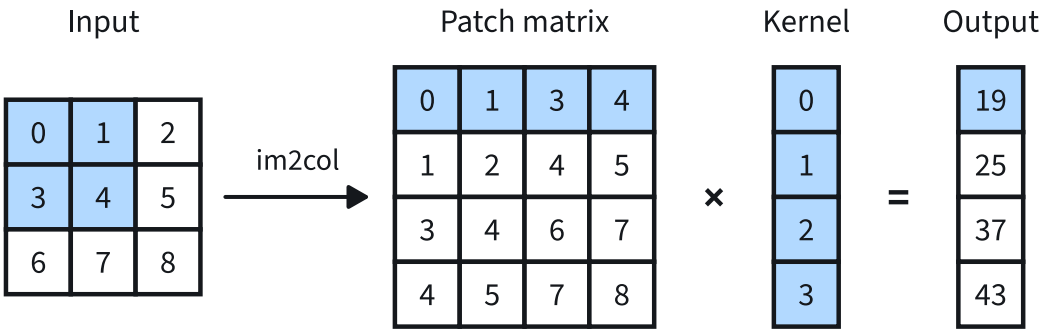


Figure 6.8: 1 回の行列積として表した Figure 6.7 の相互相関。入力の 4 つの 2×2 ウィンドウがそれぞれパッチ行列の 1 行になり、平坦化したカーネルを掛けることで平坦化された出力が得られる。網掛けのセルは、最初のウィンドウから対応する行、そして出力 19 までの対応を追ったものである。中央のピクセル 4 は 4 つすべてのウィンドウに含まれるため、すべての行に現れる。

引き換えとなるのは追加の作業用メモリである。内部の各ピクセルが最大 $k_\mathrm{h} k_\mathrm{w}$ 個のパッチに重複してコピーされるからである。以下では Figure 6.7 の例を用い、この 1 回の行列積が `corr2d` と完全に一致することを確かめる。

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
h, w = K.shape
h_o, w_o = X.shape[0] - h + 1, X.shape[1] - w + 1
patches = d2l.stack([d2l.reshape(X[i:i + h, j:j + w], (-1,))
                     for i in range(h_o) for j in range(w_o)])
Y_mat = d2l.reshape(d2l.matmul(patches, d2l.reshape(K, (-1, 1))), (h_o, w_o))
Y_mat, d2l.reduce_sum(d2l.abs(Y_mat - corr2d(X, K)))
```

```
(tensor([[19., 25.],
         [37., 43.]]),
 tensor(0.))
```

差は恒等的に 0 であり、1 回の行列積によって `corr2d` が完全に再現されている。

これと等価な方法として、入力ではなく**カーネル**の方をスパースな帯行列（banded matrix）$\mathbf{W}$ に展開し、平坦化した画像に掛けることもできる。これは Figure 6.4 のパネル (c) のちょうど 2 次元版にあたり、どの行にも同じ少数のカーネル重みが繰り返し現れる。畳み込みを構造化された全結合層とみなすことで、6.1 節の導出へと一周して戻ってきたことになる（演習問題では $\mathbf{W}$ を具体的に構成する）。

6.2.4 特徴マップと受容野

単一の畳み込み層が見ているのはカーネルサイズのパッチだけだが、物体全体を認識するには、はるかに広い領域にわたって文脈を統合する必要がある。6.1.4 節で述べたように、畳み込み層の出力は**特徴マップ**（feature map）と呼ばれる（先ほどの垂直エッジマップ `Y` もその一例である）。特徴マップ内の任意の要素 x に対し、その**受容野**とは、順伝播において x の値に影響を与えうる入力画像（またはより前の層）の領域のことである。これは神経生理学の用語を借りたものである（Figure 6.1）。

6.2.4.1 層の積み重ね

Figure 6.7 では、網掛けされた出力 19 は 2×2 の受容野を持ち、$\mathbf{X}$ の左上の 4 ピクセルをカバーしている。ここで、**2 層目**の 2×2 畳み込み層を、2×2 の出力 $\mathbf{Y}$ の上に重ねて単一のスカラー z を生成してみよう。z は $\mathbf{Y}$ の 4 要素すべてに依存し、それらは合わせて $\mathbf{X}$ の 9 要素すべてに依存する。層を積み重ねることで、深さとともに受容野が滑らかに広がっていく。

では、どのくらいの速さで広がるのだろうか。各 $k \times k$ 畳み込み層は、その入力の 1 要素よりも $k-1$ ピクセルだけ遠くまで届くため、L 個の層（カーネルサイズ $k_1, \ldots, k_L$）を積み重ねると、受容野の 1 辺の長さは次のようになる。

$$r = 1 + \sum_{i=1}^{L} (k_i - 1). \tag{6.11}$$

連続する L 個の 3×3 層を積み重ねた場合、各層が 2 ピクセルずつ範囲を広げ、$(2L+1)\times(2L+1)$ 個の入力をカバーする。すなわち、2 層後には 5×5、3 層後には 7×7 となる（Figure 6.9）。この線形な拡大は、巨大なカーネルに頼る理由ではなく、むしろ大きな利点である。3 層の 3×3 層を積み重ねれば、同じ 7×7 の受容野を単一の 7×7 層よりもはるかに少ないパラメータ数（チャネル対あたりの重みが $3 \times 9 = 27$ 個に対し 49 個）で達成しつつ、非線形性を 1 回ではなく 3 回挟み込めるからである。

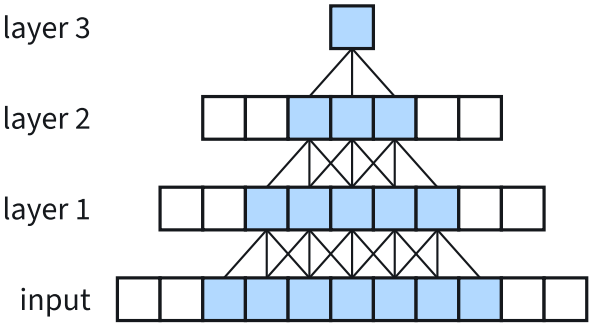


Figure 6.9: カーネルサイズ 3 の 1 次元の層を 3 層重ねた後の 1 ユニットの受容野。網掛けのセルは最上部のユニットが依存するユニットであり、各線はそれぞれのユニットとその入力を結んでいる。層ごとに受容野が 2 ずつ広がり、7 個の入力に達する。

6.2.4.2 有効受容野

ただし、r が測っているのは**理論的**受容野（theoretical receptive field）、すなわち出力ユニットへの**何らかの**経路を持つすべての入力ピクセルの範囲であり、各ピクセルが実際にどれほど強く寄与するかは表していないことに留意したい。入力勾配を直接計測することで、Luo et al. (2016) は、未訓練のネットワークと訓練済みのネットワークの両方において、**有効**受容野（effective receptive field）が中心から離れるにつれてほぼガウス型の釣鐘曲線を描いて急激に減衰することを示した。深い層のユニットは理論上は画像の広い範囲を見渡せるものの、その感度は中心付近に強く集中しているのである。

経路の数を数えれば、この釣鐘型の形状を説明できる。受容野の端にある入力は単一の結合の連鎖を通ってのみ出力ユニットに到達するのに対し、中央の入力は中間の層を通る数多くの経路をたどることができる。Figure 6.10 は、Figure 6.9 の積層構造について経路数を数えたものである。最上部のユニットに至る経路は、中央の入力では 7 本あるのに対し、両端の入力ではそれぞれ 1 本しかない。すべての重みが同程度の大きさであれば、ある入力の影響力は経路の数とともに大きくなる。

この基本的な層には、実用上の課題がまだ 2 つ残っている。パディングなしの畳み込みを繰り返すと空間次元が縮小し、かつ境界付近のピクセルがあまり使われないこと、そして単一のカーネルでは 1 つのパターンしか検出できないことである。空間サイズの調整と境界の扱いについては続く 6.3 節で、マルチチャネルの特徴マップについては 6.4 節で取り上げる。

6.2.5 まとめ

畳み込み層は、入力のすべての有効なウィンドウにわたってカーネルをスライドさせながら、学習されたカーネルとの 2 次元相互相関にスカラーのバイアスを加える。単純な 1×2 の有限差分カーネル `[1, -1]` は画像内のどこにある垂直エッジも検出でき、勾配降下法は入出力データだけからその重みを正確に復元する。これと同様に、深い CNN はフィルタの階層全体をエンドツーエンドで学習する。数学的な畳み込みはカーネルを反転させるが、カーネルをデータから学習する以上、実質的な違いはない。また、畳み込みは単一の行列積（**im2col**）として計算でき、L 個の畳み込み層を積み重ねると、各ユニットの受容野は層ごとに $k_i - 1$ ずつ線形に広がる。

6.2.6 演習問題

1. [code] **手作業によるカーネル設計**。手作業でいくつかのカーネルを設計せよ。

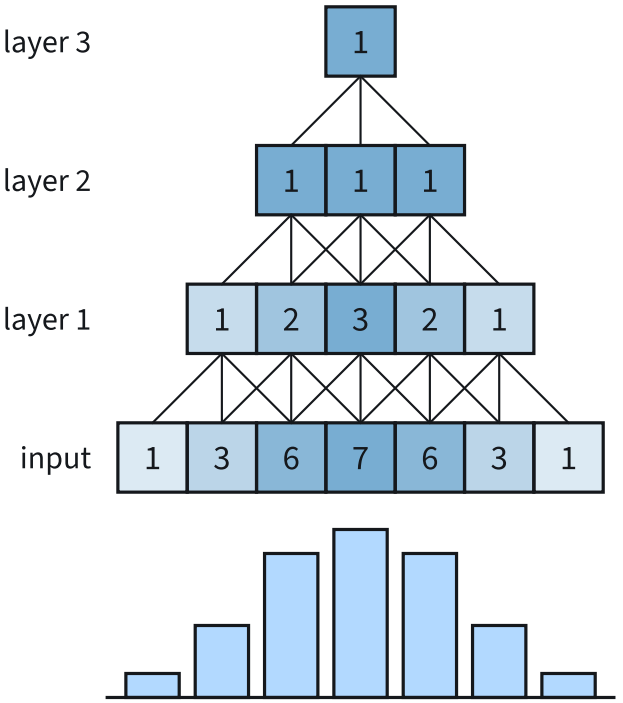


Figure 6.10: Figure 6.9（カーネルサイズ 3 の 1 次元の層 3 層）における各ユニットから最上部のユニットへの経路数。各セルの数値は、そのセルが入力を送る先のユニットの経路数の和になっている。色が濃いセルほど同じ層の他のセルより経路が多く、下部の棒グラフは入力行の経路数（1、3、6、7、6、3、1）をそのまま表している。

1. 方向ベクトル $\mathbf{v} = (v_1, v_2)$ が与えられたとき、$\mathbf{v}$ に直交するエッジ、すなわち方向 $(v_2, -v_1)$ のエッジを検出するエッジ検出カーネルを導出せよ。
2. 二階導関数に対する有限差分作用素を導出せよ。画像内のどのような構造がこれに最も強く反応するか。
3. d 階の導関数を得るために必要なカーネルの最小サイズはいくらか。
4. ぼかしカーネルをどのように設計するか。また、そのようなカーネルを使いたくなるのはなぜか。
5. 写真を 1 枚読み込み、1 チャネルに変換して約 100×100 ピクセルにダウンサンプリングせよ（`corr2d` は Python でループ処理を行うため）。エッジ検出器、ぼかしカーネル、および先鋭化カーネル「恒等変換の 2 倍からぼかしを引いたもの」を適用し、それぞれの結果を元の画像の隣に表示せよ。写真のエッジマップには、黒い帯のエッジマップでは見られなかった何が現れているか。

Stanford CS231n の Assignment 2（ConvolutionalNetworks ノートブック）を改変。

2. [code] **帯行列としての畳み込み。** 6.2.3 節のパッチ行列は入力を展開するものであった。これとは逆の構成として、カーネルを展開する。$\mathbf{X}$ を 4×4 の入力、$\mathbf{K}$ を 3×3 のカーネルとし、$\mathrm{vec}(\cdot)$ を行ごとの平坦化とする。

   1. 4×16 行列 $\mathbf{W}$ を、$\mathbf{W}\,\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{Y})$ を満たすように構成せよ。ここで $\mathbf{Y}$ は $\mathbf{X}$ と $\mathbf{K}$ の相互相関である。各行には非ゼロ要素がいくつあり、$\mathbf{W}$ には何種類の異なる値が現れるか。
   2. $\mathbf{W}$ を `K` からコード上で構築し、その行列積が `corr2d(X, K)` を再現することを確かめよ。
   3. $n \times n$ の入力と $k \times k$ のカーネルに対し、密配列として格納した $\mathbf{W}$ の要素数、$\mathbf{W}$ の非ゼロ要素数、およびパッチ行列の要素数を数えよ。それぞれの数は n に伴ってどのように増加するか。
   4. $\mathbf{W}$ のスパース性のパターンを保ったまま、各行に固有の k^2 個の値を持たせる。すると局所結合層が得られる。これは 4 階テンソル V（6.1 節の $[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_{a,b}[\mathsf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}$ におけるもの）を、平行移動同変性によって重みを共有させる前に、$k \times k$ ウィンドウのオフセット (a, b) に制限したものにあたる。$n \times n$ の入力に対し、この層、同じ入出力形状を持つ全結合層、および畳み込みのパラメータ数をそれぞれ数えよ。

   CMU 11-785 の Homework 2 Part 1 を改変。

3. [code] **有効受容野。●** 本節の理論的受容野 r はどの入力が出力に**影響を与えうる**かの範囲を与えるが、有効受容野は実際にどれだけ影響するかを測る (Luo et al. 2016)。L 個の単一チャネル 3×3 畳み込み層（パディングなし、バイアスなし、非線形性なし）を積み重ねたネットワークを 32×32 の入力上に構築せよ（6.2.2.3 節の組み込み層を用いること）。カーネルの各要素は標準正規分布から独立にサンプリングせよ。

   1. $L = 3$ とし、1 回サンプリングしたカーネルに対して、最終層の中央付近にある出力ユニットから入力まで単位勾配を逆伝播させ、入力勾配の大きさをプロットせよ。その台を 7×7 の理論的受容野と比較し、大きさがどこで最大になるかを答えよ。
   2. カーネルを 500 回サンプリングして入力勾配の二乗を平均し、これを $L = 3$ と $L = 10$ のそれぞれについて求めよ。平均した各マップを行方向に足し合わせ、得られた列方向のプロファイルを総和が 1 になるように正規化せよ。プロファイルの標準偏差と理論上の 1 辺の長さ $2L + 1$ との比は、深さに伴ってどのように変化するか。
   3. Figure 6.10 は 1 次元における $L = 3$ の経路数を数えたものである。一般の L に対し、行方向に足し合わせた入力勾配の二乗の期待値が、L 個のベクトル $(1, 1, 1)$ の畳み込みに比例することを示せ。また、正規化されたプロファイルの標準偏差が L に伴ってどのように増大するかを導出せよ。

4. [code] **畳み込みの逆伝播。●Y を入力 X とカーネル K の単一チャネル相互相関（corr2d が計算するもの）とし、ℓ を X と K に Y を介してのみ依存するスカラー損失とする。**
    1. **$\partial\ell/\partial\mathbf{K}$ と $\partial\ell/\partial\mathbf{X}$ を $\partial\ell/\partial\mathbf{Y}$ を用いて表す式を、4.3 節の誤差逆伝播法の方程式の形式で導出せよ。**
    2. **導出した $\partial\ell/\partial\mathbf{K}$ の式を、損失 $\ell = \sum_{i,j} G_{ij} Y_{ij}$ に対して自動微分を使わずに実装せよ（ここで G は Y と同じ形状の固定されたランダム行列である）。ステップサイズ 10^{-2} の中心有限差分による勾配と小さなランダムな X および K 上で比較し、最大の相対誤差を報告せよ。**
    3. **導出した $\partial\ell/\partial\mathbf{K}$ の式を、6.2.2.3 節の実験で自動微分が計算するカーネル勾配と照合して確認せよ。**

    Michael Nielsen『Neural Networks and Deep Learning』第 6 章、および Stanford CS231n の Assignment 2（ConvolutionalNetworks ノートブック）を改変。

6.3 パディング、ストライド、ダイレーション

パディングなしの $k \times k$ 畳み込みは、入力を各軸に沿って $k-1$ ピクセル縮小する（6.2 節）。5×5 の層を 10 層重ねると、240×240 の画像は 200×200 に縮小し、境界に沿って面積の 30% が失われる。しかし、ストライド 1 ですべての層を元の解像度に保つと、今度は逆の問題が生じる。各 3×3 層はユニットの受容野（receptive field）を 2 ピクセルしか広げないため、224×224 の画像全体をカバーするには 100 層以上が必要になるのである。さらに、セマンティックセグメンテーションのような密な予測（dense prediction）タスクでは、巨大なカーネルによる二乗オーダーのコストを避けつつ、広い視野と元の空間解像度（spatial resolution）の**両方**を確保したい。

3 つの幾何学的な制御が、これらのトレードオフを調整する。**パディング**は入力の周囲に枠を付加することで、カーネルの中心を端のピクセルに合わせ、空間次元を維持できるようにする。**ストライド**はウィンドウを 1 ピクセルより大きい幅で移動させることで、特徴マップ（feature map）をダウンサンプリングし、受容野の拡大を加速させる。そして**ダイレーション**はカーネルの重みの間隔を広げることで、小さなカーネルでも元の解像度のままより広い領域をカバーできるようにする。

6.3.1 パディング

6.3.1.1 境界ピクセル

層を重ねて画像の外周が削られていく以前に、境界付近のピクセルはそもそもほとんど活用されない。Figure 6.11 は、パディングなしの畳み込みにおいて各ピクセルを覆うスライディングウィンドウの数を示したものである。$k \times k$ カーネルでは、すべての境界から少なくとも $k-1$ ステップ離れた内部ピクセルは k^2 個のウィンドウに含まれるのに対し、辺上のピクセルは高々 k 個、角のピクセルはわずか 1 個のウィンドウにしか現れない。5×5 カーネルの場合、内部ピクセルは 25 個の出力位置に寄与するが、角のピクセルは 1 か所にしか寄与しないため、角の近くにある物体は次の層にほとんど反映されない。何らかの対処をしなければ、各層は入力の中心部に偏って情報を抽出することになる。

6.3.1.2 ゼロパディング

最も単純な解決策は、カーネルをスライドさせる前に、入力の周囲を余分な詰め物ピクセル（通常はゼロ）の枠で囲むことである。この手法を**パディング**と呼ぶ。Figure 6.12 では、上下にゼロの行を 1 行ずつ、左右に列を 1 列ずつ追加することで、3×3 の入力を 5×5 のグリッドへと拡張している。これにより、2×2 カーネルによって 4×4 の出力が得られる（パディングなしで

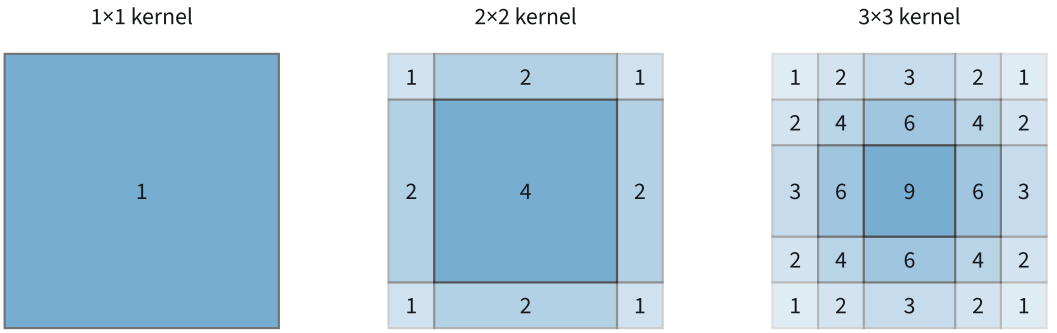


Figure 6.11: パディングなしで大きな入力上を 1×1、2×2、または 3×3 のカーネルがスライドするとき、各ピクセルを含むウィンドウの数。内部ピクセルは k^2 回、角のピクセルは 1 回覆われる。細長い各セルは、個数にかかわらず境界からその距離にあるすべてのピクセルを表す。

は 2×2 にとどまる)。網掛けされた左上の出力要素は、パディングされた 3 個のゼロと元の入力の角の要素を組み合わせたものであり、$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$ となる。

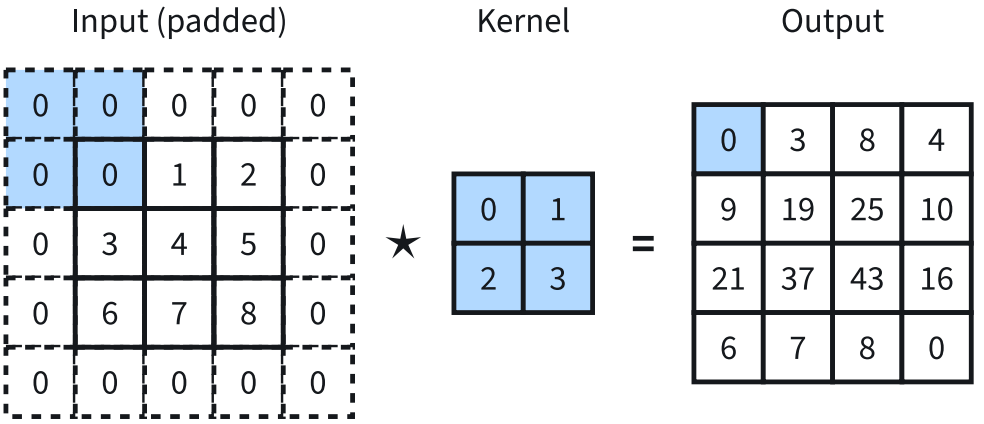


Figure 6.12: 四方に 1 ピクセルのゼロパディングを施した 2 次元相互相関。

深層学習フレームワークの慣例に従い、パディングは片側あたりの大きさで数える。すなわち、入力の上に p_h 行、下に p_h 行を追加し、左右にはそれぞれ p_w 列を追加する。このとき、パディング後の入力の形状は $(n_\mathrm{h} + 2p_\mathrm{h}) \times (n_\mathrm{w} + 2p_\mathrm{w})$ となり、出力の形状は次のようになる。

$$(n_\mathrm{h} + 2p_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} + 2p_\mathrm{w} - k_\mathrm{w} + 1). \tag{6.12}$$

上下どちらの側であってもパディングを 1 行追加するごとに出力は 1 行増え、1 列追加するごとに出力は 1 列増える。カーネルによって $k_\mathrm{h} - 1$ 行と $k_\mathrm{w} - 1$ 列が削られるため、$p_\mathrm{h} = (k_\mathrm{h} - 1)/2$ および $p_\mathrm{w} = (k_\mathrm{w} - 1)/2$ と設定すれば、入力の高さと幅がそのまま維持される。実務ではこれが圧倒的に最もよく使われる選択である。形状を維持する層を積み重ねれば、どの深さでも同じ空間解像度が保たれるため、ネットワークアーキテクチャの設計や考察がはるかに容易になる。

$k - 1$ を 2 で割ることから、CNN でほぼ常に 1、3、5、7 といった奇数のカーネルサイズが使われ、それに 0、1、2、3 のパディングが組み合わされる理由も説明できる。奇数サイズのカーネル

には明確な中心ピクセルが存在するため、対称なパディングを行えば、出力 `Y[i, j]` を計算するウィンドウの中心が入力ピクセル `X[i, j]` の真上に置かれ、入力と出力のグリッドの位置が揃ったまま保たれる。一方、偶数サイズのカーネルでは $k_{\mathrm{h}} - 1$ が奇数になるため、空間次元を維持するには非対称なパディング（片側に他方より 1 行多く追加すること）が必要になり、出力グリッドが入力に対して半ピクセルずれてしまう。

これらの出力形状の公式をコードで確認するため、まず小さなヘルパ関数を定義する。組み込みの畳み込み層（convolutional layer）はバッチ軸とチャネル軸を持つ 4 次元の入力を前提としているため、`comp_conv2d` は 2D 入力の先頭にサイズ 1 の次元を 2 つ挿入して層を適用し、結果からそれらの余分な次元を取り除く。

```
def comp_conv2d(conv2d, X):
    """Apply a convolutional layer to a single-channel 2D input."""
    Y = conv2d(X.reshape((1, 1) + X.shape))  # input (1, 1, h, w)
    return Y.reshape(Y.shape[2:])
```

3×3 カーネルの場合、形状を維持するパディングには $p = (3-1)/2 = 1$（コードでは `padding=1`）が必要である。この層を 8×8 の入力に適用すると、空間形状は 8×8 のまま変化しないはずである。

```
X = torch.rand(size=(8, 8))
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

6.3.1.3 パディングの値

ゼロパディングは、フレームの外側が黒いピクセルで囲まれていると暗黙のうちに仮定している。より形式的に言えば、パディングとは**境界条件**（boundary condition）、すなわち観測値が存在しない画像境界の外側に値を与える規則を指定するものである。ゼロパディングが出力にどう影響するかを見るため、すべての要素が 1 である一様な 5×5 画像の四方に 1 ピクセルのゼロをパディングし、すべての要素が 1 の 3×3 カーネルとの相互相関を 6.2 節の `corr2d` で計算してみよう。このとき、出力の各要素は、ウィンドウ内に含まれる本来の画像ピクセルの数を表す。

```
img = d2l.ones((5, 5))
d2l.corr2d(F.pad(img, (1, 1, 1, 1)), d2l.ones((3, 3)))
```

```
tensor([[4., 6., 6., 6., 4.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [4., 6., 6., 6., 4.]])
```

内部の出力は 9 であるのに対し、辺では 6、角では 4 に低下する。元の画像が完全に一様であっても、層にはその周囲に暗い枠があるように見えているのである。この人工的な枠は後続の層を通じて内側へと伝播し、暗黙的な位置の手がかりとして学習に利用されうる。同時に、境界のアーティファクトによって端の近くに死角が生じ、ネットワークの反応が弱くなることもある (Alsallakh et al. 2020)。ゼロを導入しない代表的な代替手法が 2 つある。**反射パディング**（reflect padding）は境界を軸にピクセルを反転させ、**複製パディング**（replicate padding）は境界の値を外側へ繰り返す。どちらも定数画像を定数出力に写すため、境界の暗転が生成ピクセルに直接現れてしま

う画像生成や画像復元のネットワークでよく使われる。しかし、分類ネットワークでは依然としてゼロパディングが標準的な選択肢である。単純であるうえ、パディング済みの大きなテンソルを別途確保することなく高速なカーネル内で暗黙的に処理できるからである。

6.3.2 ストライド

6.3.2.1 ストライド付き畳み込み

224×224 の画像をネットワーク全体にわたって元の解像度のまま保持すると、計算コストが高くつくうえ、大域的なコンテキストを集約するのにも時間がかかる。1 層あたり各チャネルに 50,176 個の空間位置が含まれ、各 3×3 畳み込みがユニットの視野を 2 ピクセルしか広げないため、1 つのユニットが画像全体を見渡せるようになるまでに 100 層以上が必要になるからである。畳み込みウィンドウをより大きな刻み幅で入力上に移動させれば、空間解像度を下げるとともに大域的なコンテキストを得るのに必要な深さも抑えられ、両方の問題を一挙に解決できる。

ここまでは、スライディングウィンドウを各行・各列に沿って 1 ピクセルずつ進めてきた。この高さ方向および幅方向のステップサイズを**ストライド**と呼ぶ。縦方向のストライドを s_h、横方向のストライドを s_w とすると、層は s_h 行ごと、s_w 列ごとにのみカーネルを評価するため、出力要素の数はおよそ $s_\mathrm{h}s_\mathrm{w}$ 分の 1 に減る。Figure 6.13 は、Figure 6.12 のパディング済み入力に対して縦方向のストライドを 3、横方向のストライドを 2 とした例を示している（2 つの軸を区別しやすいよう非対称な値を選んでいる）。網掛けされた各出力は、それを生成したウィンドウと同じ色で表されている。青い要素は $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ であり、最初のウィンドウから右に 2 列ずらした位置から得られる。橙色の要素は $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ であり、下に 3 行ずらしたウィンドウから得られる。さらに右へ 2 列進めるとウィンドウがパディング済み入力の右端をはみ出してしまうため、1 行あたりの出力は 2 個となる。

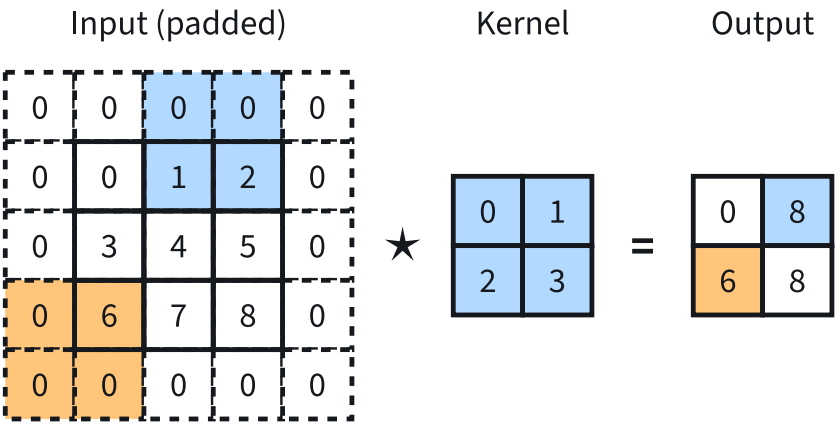


Figure 6.13: 高さ方向に 3、幅方向に 2 のストライドを用いた相互相関。網掛けされた各出力は、それを生成するウィンドウと同じ色になっている。

一般の場合の出力の高さを求めるため、ウィンドウがパディング済み入力の第 0 行から始まり、パディング済みの $n_\mathrm{h} + 2p_\mathrm{h}$ 行の内部に完全に収まらなければならない点に注目しよう。有効な開始行は $0, s_\mathrm{h}, 2s_\mathrm{h}, \ldots$ であり、上限は $n_\mathrm{h} + 2p_\mathrm{h} - k_\mathrm{h}$ となる。幅についても同様に考えると、出力の形状は次の式で与えられる。

$$\left(\left\lfloor \frac{n_\mathrm{h} + 2p_\mathrm{h} - k_\mathrm{h}}{s_\mathrm{h}} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{n_\mathrm{w} + 2p_\mathrm{w} - k_\mathrm{w}}{s_\mathrm{w}} \right\rfloor + 1\right). \tag{6.13}$$

ここで、床関数の項は最初の配置の後にウィンドウが進めるステップ数を表し、$+1$ はその開始位置の分である。Figure 6.13 に当てはめて確認すると、$\lfloor 3/3 \rfloor + 1 = 2$ 行、$\lfloor 3/2 \rfloor + 1 = 2$ 列となり、$s_\mathrm{h} = s_\mathrm{w} = 1$ と置けば先ほどのパディングの公式に戻る。ストライドを形状維持パディング（$2p_\mathrm{h} = k_\mathrm{h} - 1$）と組み合わせると、出力の高さは $\lfloor (n_\mathrm{h} - 1)/s_\mathrm{h} \rfloor + 1 = \lceil n_\mathrm{h}/s_\mathrm{h} \rceil$ に簡約される。すなわち、ストライド s は空間解像度を s で割り、端数を切り上げる。

ResNet 族（7.4 節）全体を通して使われている非常に一般的なダウンサンプリング層は、パディング 1、ストライド 2 の 3×3 畳み込みである。先ほどの 8×8 の入力に適用すると、両方の空間次元が半分になり 4×4 となる。

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

高さと幅で同じカーネルサイズ、パディング、ストライドを共有する必要はなく、出力形状の公式は各軸に独立して適用される。

ストライドがカーネルサイズに等しく、パディングがゼロである場合は重要な特殊ケースとなる。このとき、ウィンドウは入力を重なりのないタイルへと分割する。たとえば、ストライド 16 の 16×16 畳み込みは、224×224 の画像を 14×14 のパッチのグリッドに分割し、各パッチをチャネル出力のベクトルへ射影する。この**パッチ化**（patchify）ステムは、ビジョン Transformer が画像をトークンの系列に変換する仕組みであり、近年のいくつかの CNN も同じ演算から始まる。

高さと幅で同じ設定を共有する場合は、添字を省略して単に p および s と書く。特に断らない限り、畳み込み層のデフォルトはストライド 1、パディング 0（$p = 0$）である。

6.3.2.2 ストライドと受容野

6.2 節で見たように、ストライド 1 では各 $k \times k$ 層が各軸に沿って受容野を $k - 1$ ピクセルずつ広げるため、深さに対して線形にしか拡大しない。ストライドはこの拡大を劇的に加速させる。ストライド 2 の層を通過すると、出力の隣接する要素は元の画像で 2 ピクセル離れた位置に対応する。そのため、後続の 3×3 カーネルは、受容野を 2 ピクセルではなく $(3-1) \cdot 2 = 4$ 入力ピクセル広げる。2. より一般に、第 i 層への入力における 1 ステップは元の画像の $\prod_{j<i} s_j$ ピクセルにまたがるため、L 層のスタック（各層のカーネルサイズ k_i、ストライド s_i）の最上位にあるユニットの受容野の幅は次の式で与えられる。

$$r = 1 + \sum_{i=1}^{L} (k_i - 1) \prod_{j=1}^{i-1} s_j, \tag{6.14}$$

ここで、$i = 1$ に対する空積は 1 と定義する。各層は、元の画像のピクセル単位で測ったカーネルの広がり分だけ寄与する。すべてのストライドを 1 に設定すると各項は $k_i - 1$ になり、6.2 節で導いた線形の式に戻る。この公式にパディングが含まれないことに注目しよう。パディングはどの出力位置が存在するかを決めるものであり、各ユニットが観測するパッチの広さを変えるものではないからである。3×3 畳み込みの 3 層スタックを 2 通り比較すると、ストライド $(1, 1, 1)$ では $r = 1 + 2 + 2 + 2 = 7$ であるのに対し、ストライド $(2, 2, 1)$ では到達範囲が 2 倍以上の $r = 1 + 2 + 4 + 8 = 15$ に広がる。

各ストライドはそれより上位にあるすべての層の実効的な到達範囲を定数倍するため、ダウンサンプリングが続く限り受容野は幾何級数的に拡大する。ストライド 2 のステージを 5 回経ると

(224×224 の入力を扱う画像分類器（image classifier）で典型的な構成である）、特徴マップ上の 1 ステップは 32 入力ピクセルにまたがるようになり、最上位のユニットはわずか数層で画像全体を見渡せるようになる。

6.3.3 ダイレーション

6.3.3.1 ダイレーション畳み込み

ストライド付き畳み込み（strided convolution）は、空間解像度を犠牲にして広い受容野を得る。しかしタスクによっては、広いコンテキストとすべての入力ピクセルに対する出力の両方が求められる。単にカーネルを大きくすれば視野は広がるものの、パラメータ数（parameter count）と計算コストがカーネルサイズの二乗で増大してしまう。パディングとストライドがカーネルを**どこに**配置するかを制御するのに対し、ダイレーションは**カーネル自体の間隔**を変化させる。すなわち、**ダイレーション畳み込み**（dilated convolution）は隣接する重みの間に隙間を挿入し、重みを d ピクセル間隔で評価する (Yu and Koltun 2016)。ここで d は**ダイレーション**率である。$k \times k$ カーネルは、ダイレーション d を用いても学習可能な重みを k^2 個しか含まないが、次の式で与えられる**実効カーネルサイズ**（effective kernel size）を持つ。

$$k' = k + (k-1)(d-1) = d(k-1) + 1 \tag{6.15}$$

これは各軸に沿ったピクセル数であり、連続するタップの間で $d-1$ ピクセルをスキップする。$d = 1$ と設定すれば通常の密な畳み込みに戻る。Figure 6.14 は、3×3 カーネルにおける $d = 1$ と $d = 2$ を対比したものである。ダイレーション 2 では、1 ピクセルおきにサンプリングすることで同じ 9 個の重みが 5×5 の領域をカバーしつつ、出力位置ごとに必要な乗算は依然として 9 回だけで済む。

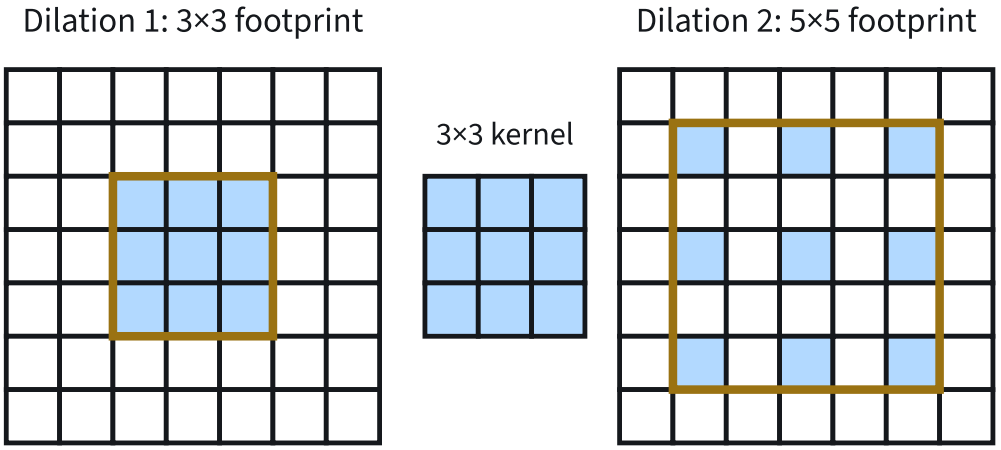


Figure 6.14: ダイレーション 1 およびダイレーション 2 における 3×3 カーネル。ダイレーション 2 では、9 個の重みが 5×5 の領域に広がる。

出力次元に関しては、ダイレーションを施したカーネルはサイズ k' の密なカーネルとまったく同じように振る舞う。ストライド付きの公式において k を $k' = d(k-1)+1$ に置き換えると、任意のパディング、ストライド、ダイレーションに対する軸方向の出力サイズを表す単一の統一式が得られる。

$$\left\lfloor \frac{n + 2p - d(k-1) - 1}{s} \right\rfloor + 1, \tag{6.16}$$

この式は、それぞれのパラメータ n、k、p、s、d を用いて高さと幅に独立して適用される。$d = 1$ と置けばストライド付きの公式が再現され、さらに $s = 1$ かつ $p = 0$ と置けば最初の $n - k + 1$ に戻る。したがって、先ほどの 8×8 の入力に対し、ダイレーション 2 の 3×3 カーネルは 4×4 の出力形状を生成し、これは密な 5×5 カーネルの出力形状と一致するはずである。

```
conv2d = nn.LazyConv2d(1, kernel_size=3, dilation=2)
conv5 = nn.LazyConv2d(1, kernel_size=5)
comp_conv2d(conv2d, X).shape, comp_conv2d(conv5, X).shape
```

```
(torch.Size([4, 4]), torch.Size([4, 4]))
```

6.3.3.2 ダイレーションと受容野

ストライド 1 では、ダイレーションを用いた層は受容野を $k' - 1 = d(k-1)$ ピクセル広げるため、各 k_i を実効サイズ k'_i に置き換えれば Equation 6.14 がそのまま成り立つ。各層でダイレーション率を $d_i = 1, 2, 4, \ldots, 2^{L-1}$ のように倍増させていくと、L 層の 3×3 カーネルからなるスタックは幅 $1 + 2(1 + 2 + \cdots + 2^{L-1}) = 2^{L+1} - 1$ の受容野を達成する。ストライド付きスタックとまったく同様に受容野は深さに対して指数関数的に拡大するが、すべての層で入力の空間解像度がそのまま維持される。

Figure 6.15 は、これら 2 つの戦略を並べて示したものである。*パネル* (a) はストライド $(2, 2, 1)$ のストライド付きスタックを示しており、3 つの層が最上位ユニットの視野にそれぞれ 2、4、8 ピクセルを加え、合計 15 入力ピクセルに達する。*パネル* (b) はストライド 1 とダイレーション $(1, 2, 4)$ を用いてまったく同じ 15 ピクセルの受容野を実現しており、最上位ユニットと入力を同一の接続パターンで結んでいる。唯一の違いは空間的な密度である。ストライド付きスタックが空間位置を 4 つごとにしか保持しないのに対し、ダイレーションスタックはすべての位置を評価する。実際、パディングなしで重みを共有すれば、各層の後に要素ごとの非線形性を適用した場合であっても、ストライド付きスタックの出力はダイレーションスタックの出力を 4 要素ごとに取り出したものにほかならない。このため、ダイレーション畳み込みはセマンティックセグメンテーションのような**密な予測**タスクで特に威力を発揮する。こうしたタスクでは、周囲の広いコンテキストを参照しつつ、すべてのピクセルにクラスラベルを割り当てなければならないからである (Yu and Koltun 2016)。トレードオフとなるのは計算量とメモリであり、すべての層が元の解像度の特徴マップを処理しなければならない。もう 1 つの選択肢は、ストライドによって一度ダウンサンプリングし、後から 32.10 節（英語版）で扱う転置畳み込み（transposed convolution）のような学習可能なアップサンプリング層を用いて空間解像度を復元する方法である。

本節のここまでの例はすべて、単一チャネルの入力に対して単一のカーネルを作用させるものであった。実世界の画像は複数の色チャネルを含み、実用的な隠れ層（hidden layer）は数十から数百もの異なるパターンを同時に検出する。6.4 節では、この両方の制約を取り除く。

6.3.4 まとめ

パディングを行わない場合、畳み込みは入力を各軸に沿って $k - 1$ ピクセル縮小させ、境界ピクセルを内部ピクセルよりはるかに少ないウィンドウでしかサンプリングしない。奇数サイズのカーネルの両側に $(k-1)/2$ ピクセルのパディングを追加すれば、入力の形状が維持され、各出力の中心が対応する入力ピクセルの上に保たれる。ただし、ゼロパディングは深い層が検出しうる境界条件を導入する。ストライド $s > 1$ を用いると s 番目ごとのウィンドウ位置のみが評価さ

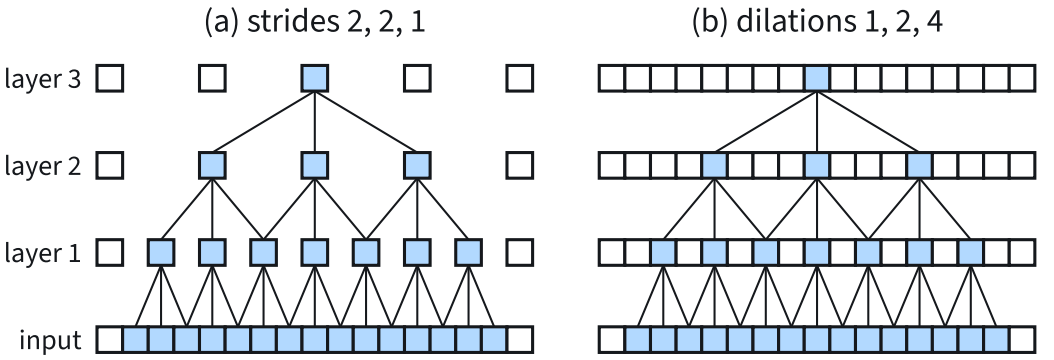


Figure 6.15: 1 次元においてカーネルサイズ 3 の層を 3 層重ねたときの、最上位ユニットの受容野。各セルは各層が計算する位置、網掛けのセルは最上位ユニットが依存する位置、線はカーネルのタップを表す。(a) ストライド 2、2、1 の場合、第 1 層は 2 つごとの位置を保持し、第 2 層と第 3 層は 4 つごとの位置を保持する。(b) ストライド 1、ダイレーション 1、2、4 の場合、すべての層が全位置を保持する。どちらの最上位ユニットも同じ 15 個の入力ピクセルを参照する。

れ、空間解像度と計算量が各軸あたりおよそ s 分の 1 に減る一方で、上位層の受容野は幾何級数的に拡大する。最後に、ダイレーション d はカーネルの重みを実効サイズ $d(k-1)+1$ の領域にわたって配置し、パラメータを増やすことなく元の解像度のまま受容野を指数関数的に拡大する。これら 3 つの仕組みはすべて、単一の出力形状公式 Equation 6.16 に集約される。

6.3.5 演習問題

1. [code] **ミラーパディング**。ミラーパディング（反射パディングとも呼ばれる）は、境界の要素を繰り返さずに境界を軸として要素を反転させることでテンソルを拡張する。たとえば、ベクトル $(1,2,3)$ の両側に 1 要素ずつパディングすると $(2,1,2,3,2)$ になる。
   1. 2 次元テンソルの四方に 1 要素ずつミラーパディングを行う処理を実装し、互いに異なる要素を持つ 3×3 テンソルに対して手計算で結果を確かめよ。
   2. ゼロパディングのかわりにミラーパディングを用いて 6.3.1.3 節の計算を繰り返し、その結果を説明せよ。
   3. 5×5 の傾斜（ランプ）データ $x_{ij}=j$ と、3×3 カーネル（すべての要素が $\frac{1}{9}$）を考える。内部では出力が入力に一致することを示せ。また、ゼロパディングおよびミラーパディングのもとで最初と最後の列の出力を計算し、この傾斜データを厳密に再現するようなパディングの方法を見つけよ。
2. **ダイレーションとグリッディング**。あるネットワークが、ストライド 1 の 3×3 畳み込みを 4 層積み重ねているとする。ダイレーションを倍増させるスケジュール $(1,2,4,8)$ では、1 つの出力要素は 1 辺が 31 の受容野を持つ。これを一定のスケジュール $(4,4,4,4)$ と比較せよ。
   1. 各カーネルをその実効サイズに置き換えた Equation 6.14 を用いて、一定のスケジュールにおける受容野を計算せよ。
   2. 各スケジュールについて、受容野内のどのピクセルに出力が実際に依存するかを特定し、その個数を数えよ。受容野内の疎な格子上のピクセルにしか出力が依存しないスケジュールは、**グリッディング**を起こすという。
   3. セグメンテーションのような密な予測タスクにおいて、グリッディングが有害となる理由を 1 つ挙げよ。
   4. ●ダイレーションを $d_1,\dots,d_L$ とするストライド 1 の 3×3 畳み込みのスタックにお

いて、出力が受容野内のすべてのピクセルに依存することを保証する条件を述べ、その条件が十分であることを示せ。

3. **ストライドとダイレーションの比較**。@fig-conv-rf-towers の 2 つの 1 次元スタックを、パディングなし、両スタックで共有されるサイズ 3 のカーネル $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$、および各層の後の要素ごとの非線形性のもとで考える。
   1. 各スタックの位置 i における出力を入力の関数として書き下し、ストライド付きスタックの位置 i における出力が、ダイレーションスタックの位置 $4i$ における出力に等しいことを示せ。
   2. 長さ n の入力に対し、境界の影響を無視して各スタックが実行する乗算の回数を数えよ。
   3. 最初の設問を一般化せよ。ストライド $s_1, \ldots, s_L$ を持つスタックを再現するには、ダイレーションをどのように設定すればよいか。
4. **ステムを通過する形状**。224×224 の画像が 4 つの層を通過するとする。すなわち、パディング 3・ストライド 2 の 7×7 畳み込み、パディング 1・ストライド 2 の 3×3 畳み込み、およびパディング 1・ストライド 1 の 2 つの 3×3 畳み込みである。
   1. すべての層について出力の高さを求めよ。
   2. Equation 6.14 を用いて、最終層のユニットの受容野を計算せよ。そのユニットは画像の幅のどれだけの割合を見ているか。
   3. 最初の 2 層を、パディングなし・ストライド 4 の単一の 4×4 畳み込み（本節で扱ったパッチ化層）に置き換える。改めて出力の高さと受容野を求めよ。第 1 層の隣接するウィンドウは依然として重なり合うか。

6.4 複数の入力チャネルと複数の出力チャネル

ここまでの畳み込みは、単一の 2 次元グリッドを単一の 2 次元グリッドへ写すものであった。しかし実際の画像や中間表現は、入出力の両側に複数の**チャネル**を持つ。たとえば RGB 写真は $3 \times h \times w$ のテンソルであり（等輝度で赤と緑が接する境界を検出するにはチャネル間の比較が必要になる）、隠れ層は何十、何百もの異なる特徴マップ（feature map）を並列に抽出する。

6.1 節で導出したように、c_i 個の入力チャネルと c_o 個の出力チャネルを持つ層は、出力チャネルごとに $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ のカーネルを 1 つずつ保持し、すべての入力チャネルにわたって相互相関をとってその結果を足し合わせる。本節では、多チャネル畳み込みを実装し、パラメータ数・FLOP・活性化メモリのコスト（いずれも積 $c_\mathrm{i} c_\mathrm{o}$ に支配される）を整理したうえで、1×1 ボトルネック畳み込みを導入する。

6.4.1 複数の入力チャネル

まず入力側から見ていこう。入力が c_i 個のチャネルを持つとき、カーネルは各入力チャネルに対して $k_\mathrm{h} \times k_\mathrm{w}$ のフィルタを 1 つずつ用意する必要があり、その結果、形状 $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ の 3 階カーネルテンソルとなる。各入力チャネル $\mathbf{X}_c$ と対応するカーネルスライス $\mathbf{K}_c$ について 2 次元の相互相関（6.2 節、`corr2d` で計算）をとり、得られた c_i 個の中間グリッドを足し合わせる。

$$\mathbf{Y} = \sum_{c=1}^{c_\mathrm{i}} \mathbf{X}_c \star \mathbf{K}_c. \tag{6.17}$$

したがって、得られる 2 次元出力の各要素は、すべての入力チャネルにわたる同じ空間ウィンドウ上の重み付き和であり、$c_\mathrm{i} k_\mathrm{h} k_\mathrm{w}$ 個のスカラー積を合わせたものになる。Figure 6.16 に 2 つの入力チャネルに対するこの計算を示す。網掛けされた左上の出力要素は $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ に等しい。

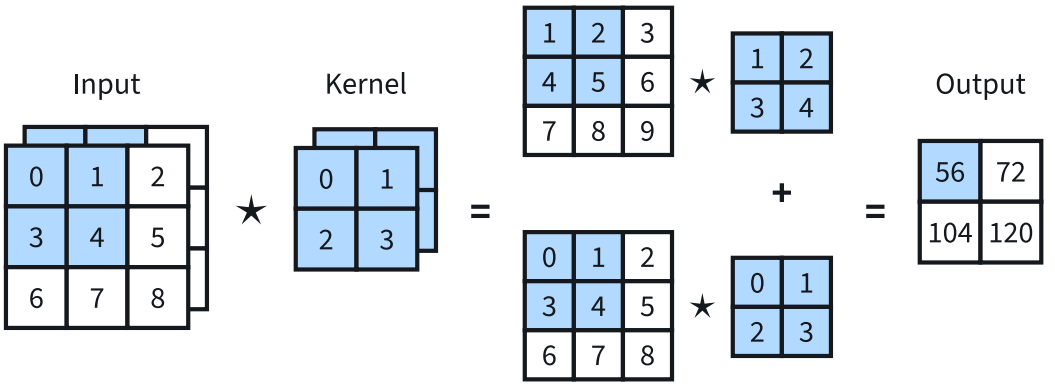


Figure 6.16: 2 つの入力チャネルに対する相互相関。チャネルごとに相互相関を計算し、それらを足し合わせる。

チャネル間で和をとることこそが、色のように異なるチャネルが担う情報をカーネルが比較できる理由である。例として、1×1 カーネルで赤チャネルの重みを $+1$、緑チャネルの重みを -1 としたものを考えてみよう。このカーネルは各ピクセルで「赤マイナス緑」を計算するため、明るさにかかわらず、赤い領域では正の値、緑の領域では負の値、無彩色の灰色の領域では 0 を出力する。隣接するピクセル間で符号が反転する場所では、赤の領域と緑の領域が接している。たとえ両領域の輝度がまったく同じで、グレースケールの明るさフィルタではエッジが一切見えない場合でも、この境界を検出できる。

多入力の相互相関の実装は、この数式をそのまま Python に書き下すだけである。チャネルごとに `corr2d` を 1 回呼び出し、その結果を足し合わせればよい。

```
def corr2d_multi_in(X, K):
    """Cross-correlate each channel of X with its channel of K and add."""
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

Figure 6.16 の入力テンソルとカーネルテンソルを使って、この実装を検証しよう。左上の出力要素は 56 になるはずである。

```
X = d2l.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = d2l.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

6.4.2 複数の出力チャネル

どれほど多くの入力チャネルを読み込もうとも、単一の多入力カーネルが生成するのは 1 枚の 2 次元**特徴マップ**（6.1.4 節）だけである。異なる c_{o} 枚の特徴マップを同時に抽出するため、畳み込み層はそのようなカーネルを c_{o} 個保持する（各カーネルの形状は $c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ である）。これらを新しい軸に沿って積み重ねると、形状 $c_{\mathrm{o}} \times c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ の 4 階カーネルテンソルが得られ、

出力チャネル o は次の式で与えられる。

$$\mathbf{Y}_o = \sum_{c=1}^{c_\mathrm{i}} \mathbf{X}_c \star \mathbf{K}_{o,c}, \quad o = 1, \ldots, c_\mathrm{o}. \tag{6.18}$$

ここで、各出力チャネルは固有の $k_\mathrm{h} \times k_\mathrm{w}$ スライス $\mathbf{K}_{o,c}$ を介して**すべての**入力チャネルと結合している。この密な全対全のチャネル結合こそが、この層の計算コストを押し上げる主な要因である。出力チャネルごとにスカラーのバイアス u_o を加えれば、6.1 節で導出した多チャネル畳み込み層が得られる。典型的な CNN アーキテクチャでは、ストライドやプーリングによって空間解像度が縮小するにつれてチャネル数を増やし、きめ細かい空間配置と引き換えに、各位置で共同して学習されたより豊かな特徴記述を得る。

PyTorch はカーネルをチャネルファーストの $(c_\mathrm{o}, c_\mathrm{i}, k_\mathrm{h}, k_\mathrm{w})$ として格納する。

多出力チャネルの相互相関を実装するには、出力カーネルを順に取り出してそれぞれに `corr2d_multi_in` を適用し、得られた特徴マップを軸 0 に沿って積み重ねる。

```
def corr2d_multi_in_out(X, K):
    """Apply corr2d_multi_in once per output channel of K and stack."""
    return d2l.stack([corr2d_multi_in(X, k) for k in K], 0)
```

この関数をテストするため、`K`、`K + 1`、`K + 2` を積み重ねて 3 つの出力チャネルを持つカーネルテンソルを作ってみよう。

```
K = d2l.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

`corr2d_multi_in_out` を 2 チャネルの入力 `X` と積み重ねた $3 \times 2 \times 2 \times 2$ のカーネル `K` に適用すると、形状 $(3, 2, 2)$ の 3 チャネル出力テンソルが得られる。

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

1 番目の出力チャネルは、カーネルが元の `K` のままであるため先ほどの結果と一致する。2 番目のチャネルも暗算で確かめられる。カーネルのすべての重みに 1 を加えると、各出力要素は、両方の入力チャネルにわたる対応ウィンドウ内の全要素の和だけ増加する。左上のウィンドウではその和が 20 なので、2 番目のチャネルの先頭要素は $56 + 20 = 76$ となる。

6.4.2.1 パラメータとコスト

チャネルを増やすとネットワークの表現力は高まるが、同時にメモリ使用量と計算量も急激に増大する。出力チャネルごとに 1 つのスカラーバイアスを含めると、畳み込み層が持つ要素数は次の式で表される。

$$c_{\mathrm{o}} c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}} + c_{\mathrm{o}} \tag{6.19}$$

これが学習可能なパラメータの数である。ネットワークの入力付近では、この数は控えめである。たとえば、3 つの RGB チャネルを 64 チャネルへ写す 3×3 の層（6.1 節）が持つパラメータは 1,792 個にすぎない。しかしネットワークの深い位置ではチャネル数が大きくなり、256 チャネルを 256 チャネルへ写す 3×3 の層では、589,824 個の重みと 256 個のバイアスが必要になる。

次に計算コストを考えると、各出力要素の計算には、その出力チャネルの各重みにつき 1 回の積和演算（multiply-add）が必要になる（合計 $c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}}$ 回）。空間サイズ $h_{\mathrm{o}} \times w_{\mathrm{o}}$、出力チャネル数 c_{o} の出力特徴マップ全体にわたってこの層が実行する演算回数は、次の式で与えられる。

$$h_{\mathrm{o}} w_{\mathrm{o}} \cdot c_{\mathrm{o}} c_{\mathrm{i}} k_{\mathrm{h}} k_{\mathrm{w}} \tag{6.20}$$

これだけの積和演算が実行される。56×56 の特徴マップに適用した場合、先ほどの 256 チャネルから 256 チャネルへの層は、画像 1 枚あたり $56^2 \cdot 589{,}824 \approx 1.8 \times 10^9$ 回の積和演算を実行する。パラメータ数と演算コストはいずれも積 $c_{\mathrm{i}} c_{\mathrm{o}}$ に比例するため、各層の幅を 2 倍にすると両コストとも 4 倍になる。また、同じ重みがすべての空間位置で再利用されるため、演算コストは出力面積 $h_{\mathrm{o}} w_{\mathrm{o}}$ にもそのまま比例する。

3 つ目のリソースコストは、訓練時に同等に重要でありながら見落とされがちな**活性化メモリ**（activation memory）である。逆伝播で勾配を計算するため、フレームワークは各層が出力した活性化値を保持しなければならず、訓練例 1 つあたり $h_{\mathrm{o}} w_{\mathrm{o}} c_{\mathrm{o}}$ 個の数値を保存する（5.7.3.2 節）。パラメータのメモリとは異なり、活性化メモリは**最も初期の**層で最大になる。そこではパラメータ数こそごくわずかだが、空間解像度が最も高いからである。たとえば、3 チャネルから 64 チャネルへ写して 224×224 の出力を生成する層は、訓練例 1 つあたり $224^2 \cdot 64 \approx 3.2 \times 10^6$ 個の活性化値をキャッシュする必要があり、これは 1,792 個のパラメータの約 2,000 倍もの数値にあたる。

6.4.3 1×1 畳み込み

一見すると、1×1 畳み込み（$k_{\mathrm{h}} = k_{\mathrm{w}} = 1$）は無意味に思えるかもしれない。空間ウィンドウが 1 ピクセルしか覆わないため、エッジやコーナーなど近傍の位置にまたがるパターンを一切検出できないからである。しかし複数のチャネルが存在するときには、先ほどの「赤マイナス緑」の色フィルタと同様に、きわめて有用な演算となる。各空間座標 (i, j) において、1×1 畳み込みは c_{i} 次元のチャネルベクトルに共有された $c_{\mathrm{o}} \times c_{\mathrm{i}}$ の重み行列を掛け、c_{o} 次元の出力ベクトルを生成する。言い換えれば、すべての空間位置で重みを共有しつつ、各ピクセルに独立して適用される全結合層（fully connected layer）として機能するのである。Figure 6.17 に、高さと幅を保ったまま 3 つの入力チャネルを 2 つの出力チャネルへ写す 1×1 畳み込みを示す。

すべてのピクセルでまったく同じ線形変換が適用されるため、1×1 畳み込み全体は 1 回の行列積で計算できる。すなわち、入力を $c_{\mathrm{i}} \times hw$ の行列に変形し、左から $c_{\mathrm{o}} \times c_{\mathrm{i}}$ のカーネル行列を掛け、得られた出力を (c_{o}, h, w) の形状に戻せばよい。

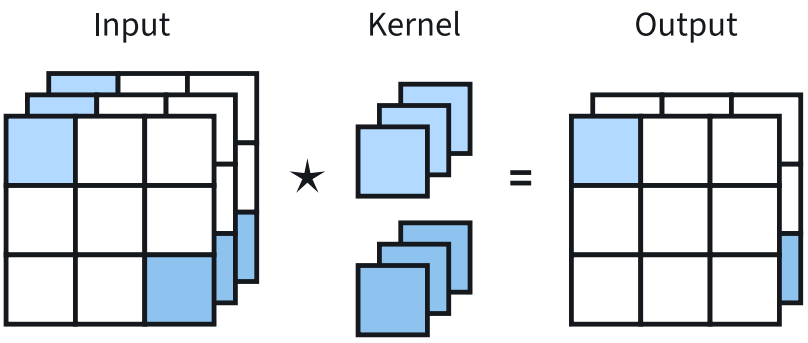


Figure 6.17: 3 つの入力チャネルと 2 つの出力チャネルを持つ 1×1 畳み込み。各出力要素は、同じ位置にある入力要素の線形結合である。

```
def corr2d_multi_in_out_1x1(X, K):
    """A 1x1 convolution as one matrix product."""
    c_i, h, w = X.shape
    c_o = K.shape[0]
    Y = d2l.matmul(d2l.reshape(K, (c_o, c_i)), d2l.reshape(X, (c_i, h * w)))
    return d2l.reshape(Y, (c_o, h, w))
```

ランダムな入力を用いて、この行列積による定式化が `corr2d_multi_in_out` の結果と一致することを確かめよう。

```
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
float(d2l.reduce_sum(d2l.abs(Y1 - Y2)))
```

```
0.0
```

期待どおり、2 つの結果は浮動小数点の丸め誤差の範囲内で一致する。

なぜ現代のアーキテクチャでは 1×1 畳み込みがこれほど広く使われているのだろうか。最大の強みは、チャネル次元を低コストで調整し、高コストな空間畳み込みをより狭い表現の上で実行できるようにする点にある。先ほどの 256 チャネルから 256 チャネルへの 3×3 層に戻ろう。3×3 カーネルを 256 チャネルすべてに直接適用するかわりに、まず 1×1 畳み込みで 256 チャネルを 64 チャネルへ射影し、その 64 チャネルに対して 3×3 畳み込みを適用したのち、2 つ目の 1×1 畳み込みで 256 チャネルへ戻すことができる。3 つの層を合わせても使用する重みは $16{,}384 + 36{,}864 + 16{,}384 = 69{,}632$ 個にとどまり、$589{,}824$ 個と比べてパラメータ数と積和演算の両方で約 $8.5\times$ の削減となる。このような**ボトルネック**は中間での空間的な混合を 64 チャネルの部分空間に制限するが、ResNet（7.4 節）などのアーキテクチャは、はるかに深いネットワークを実現する引き換えとしてこのトレードオフを積極的に受け入れている。また、1×1 畳み込みの直後に活性化関数を置けば、各ピクセルで評価される小さな非線形 MLP となる (Lin et al. 2013)。非線形性がなければ、連続する線形な畳み込みは単一の線形変換に縮退してしまう。

どの入力チャネルを各出力チャネルに入力するかを制限することで $c_i c_o$ の因子を削減するアーキテクチャ上の仕組みは、ほかにも 2 つある。ResNeXt（7.4.5.1 節）で広く知られるようになった

グループ畳み込み（grouped convolution）と、軽量なモバイルモデル（7.5.1.1 節）の基盤をなす**深さ方向分離可能畳み込み**（depthwise-separable convolution）である。これらについては次章で、それぞれを用いるアーキテクチャを構築しながら導入する。

なお、チャネルを導入しただけでは、1 つの基本的な性質は変わらないままである。すなわち、入力がずれると畳み込みの出力もやはりピクセル単位でそのまま移動する。6.5 節では、局所近傍を要約して小さな空間的ずれに対する耐性を獲得するために、プーリング層を導入する。

6.4.4 まとめ

多チャネル入力を扱うため、畳み込み層は各入力チャネルと固有の空間カーネルとの相互相関をとってその結果を足し合わせる。これにより、単一のフィルタで色のようなチャネル間の手がかりを統合できる。c_o 枚の出力特徴マップを生成するため、この層はそのような多入力カーネルを c_o 個積み重ねて形状 $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ のテンソルを構成する。これには $c_\mathrm{o} c_\mathrm{i} k_\mathrm{h} k_\mathrm{w} + c_\mathrm{o}$ 個のパラメータと $h_\mathrm{o} w_\mathrm{o} c_\mathrm{o} c_\mathrm{i} k_\mathrm{h} k_\mathrm{w}$ 回の積和演算（いずれも $c_\mathrm{i} c_\mathrm{o}$ に比例）を要し、さらに訓練時には訓練例 1 つあたり $h_\mathrm{o} w_\mathrm{o} c_\mathrm{o}$ 個の活性化値を保存しなければならない。1×1 畳み込みは各ピクセルで共有される全結合層として機能し、チャネルを混合してパラメータ効率・計算効率の高いボトルネックを構築することを可能にする。

6.4.5 演習問題

1. **畳み込みの合成**。ストライド 1、パディングなしで、間に非線形性を含まない 2 つの畳み込み層があり、それぞれのカーネルの形状を $c_\mathrm{m} \times c_\mathrm{i} \times k_1 \times k_1$ および $c_\mathrm{o} \times c_\mathrm{m} \times k_2 \times k_2$ とする。
    1. これら 2 層の合成が単一の畳み込みになることを証明し、そのカーネルの形状を求めよ。
    2. この 2 層の組が持つ重みの数と、単一の畳み込みが持つ重みの数を比較せよ。どのような中間幅 c_m のときに 2 層の組のほうが低コストになるか。
    3. 6.2 節で示したように、積み重ねた 3 つの単一チャネル 3×3 カーネルは 7×7 の受容野を持つ。すべての単一チャネル 7×7 カーネルが 3 つの 3×3 カーネルの合成として表せるかどうかを判定し、その理由を述べよ。
2. **3 つのコストと 3 つの層**。224×224 の RGB 画像を入力とするネットワークが、パディング 1 の 4 つの 3×3 畳み込みから始まるとする。すなわち、3 チャネルから 64 チャネル（ストライド 1）、64 チャネルから 128 チャネル（ストライド 2）、128 チャネルから 128 チャネル（ストライド 1）、そして 128 チャネルから 256 チャネル（ストライド 2）である。
    1. 各層について、出力の形状、重みの数、画像 1 枚あたりの積和演算の回数、および逆伝播のために訓練時に保持しなければならない出力値の個数を求めよ。
    2. 重みの数が最も多い層、積和演算の回数が最も多い層、保持する出力値の個数が最も多い層はそれぞれどれか。これらがすべて異なる 3 つの層になる理由を説明せよ。
    3. 入力画像を 448×448 の画像に置き換える。3 つのコストのうちどれが変化し、何倍になるか。
3. [code] **多チャネルの im2col**。6.2.3 節のパッチ行列を複数の入力チャネルと出力チャネルに拡張せよ。本節の後半のセルで上書きされているため、Figure 6.16 の 2 チャネル入力 `X` と 6.4.2 節の 3 出力チャネルカーネル `K` を再構築せよ。形状 $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ の入力パッチをその軸順で平坦化したものを各行とするパッチ行列を作成し、同様に `K` も同じ順序で行を並べた $c_\mathrm{i} k_\mathrm{h} k_\mathrm{w} \times c_\mathrm{o}$ のカーネル行列に並べ替えよ。両者の積を出力の形状に並べ替えると `corr2d_multi_in_out(X, K)` が再現されることを確認せよ。

    Simon Prince『Understanding Deep Learning』問題 10.6、10.7 より改作。

6.5 プーリング

畳み込み層は（ピクセル (i, j) にエッジがあるかといった）局所的な問いに答えるのに対し、分類器は画像全体に関する大域的な問いに答えなければならない。局所的な特徴マップから大域的な予測へと橋渡しをするには、広い空間的到達範囲（6.3.2.2 節）と、カメラの 1 ピクセルの手ぶれで予測が変わらないような微小な空間的シフトに対する頑健性（robustness）の両方が求められる。

プーリング層（pooling layer）は、学習可能なパラメータを追加することなくこの両方の目的を果たす。各チャネルの上で独立にウィンドウをスライドさせ、パッチごとにその**最大値**または**平均**によって要約するのである。小さなブロック内の最大値はピーク応答がそのブロック内にとどまる限り変化しないため、局所プーリングは小さなシフトに対する**耐性**（tolerance）をもたらす。一方、特徴マップ全体にわたる**グローバルプーリング**は、空間座標を潰してチャネルごとの単一のベクトルに集約する。

6.5.1 最大プーリングと平均プーリング

畳み込み層と同様に、プーリング演算子は固定形状のウィンドウ（**プーリングウィンドウ**）を入力上でスライドさせ、各位置で単一のスカラー出力を計算する。しかし畳み込みとは異なり、プーリング層は学習可能なカーネルの重みを持たず、ウィンドウ内の要素に対して固定された決定論的な縮約を適用する。**最大プーリング**（maximum pooling、略して **max-pooling**）はウィンドウ内の最大の要素を返し、**平均プーリング**（average pooling）はそれらの算術平均を返す。

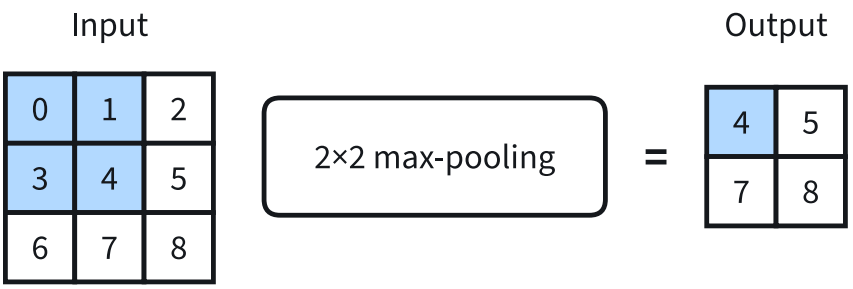


Figure 6.18: プーリングウィンドウの形状が 2×2 の最大プーリング。網掛け部分は最初の出力要素と、その出力の計算に使われる入力テンソルの要素を表す（$\max(0, 1, 3, 4) = 4$）。

Figure 6.18 では、2×2 のウィンドウが 3×3 の入力上をストライド 1 で左から右、上から下へとスライドする。ウィンドウは各軸に沿って 2 つの位置に収まるため、出力の形状は 2×2 となり、各要素はそれぞれのウィンドウの最大値で与えられる。

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned}$$

9 個の入力要素のうち 5 個（0、1、2、3、6）は、どのウィンドウでも最大値にならないことに注目しよう。最大プーリングは各ウィンドウで最大となる要素のみに依存するため、これらの要素をわずかに摂動させても出力はまったく変化しない。同じスライディングウィンドウの構成は、任意の $k_{\mathrm{h}} \times k_{\mathrm{w}}$ プーリングウィンドウへそのまま一般化できる。

以下では、2 次元プーリングの順伝播を pool2d として実装する。その入れ子のループは corr2d（6.2 節）と同じ構造であり、重み付き内積を max または mean に置き換えている。また、次にダウンサンプリングを調べられるよう stride 引数も用意しておく。

```
def pool2d(X, pool_size, mode='max', stride=1):
    k_h, k_w = pool_size
    Y = d2l.zeros(((X.shape[0] - k_h) // stride + 1,
                   (X.shape[1] - k_w) // stride + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            window = X[i * stride: i * stride + k_h,
                       j * stride: j * stride + k_w]
            Y[i, j] = window.max() if mode == 'max' else window.mean()
    return Y
```

pool2d を 3×3 テンソル（Figure 6.18）でテストすると、最大プーリングによって 4、5、7、8 が得られることが確認できる。

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

同じ入力に対して平均プーリングに切り替えると、4 つのウィンドウの平均である 2、3、5、6 を返す。

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

6.5.2 小さなシフトに対する耐性

プーリングは空間的なシフトをどの程度まで吸収できるのだろうか。最も単純な設定で試してみよう。ストライド 2 の 2×2 プーリング層は、入力を重なりのない 2×2 のブロックに分割し、各ブロックを単一の値に要約する。（孤立したエッジにエッジ検出器が反応したときのように）1 という非ゼロの応答をただ 1 つだけ含む 4×4 の特徴マップを考え、その応答を行 1 に沿って 1 ピクセルずつスライドさせてみよう。以下では、連続する 3 つの列位置にわたって最大プーリングと平均プーリングの出力を比較する。

```
def single_response(i, j, n=4):
    """An n x n feature map that is 1 at (i, j) and 0 elsewhere."""
    return d2l.tensor([[float(r == i and c == j) for c in range(n)]
                       for r in range(n)])

for j in range(3):
    Z = single_response(1, j)
    print(f'response at (1, {j}): max',
          d2l.numpy(pool2d(Z, (2, 2), stride=2)).tolist(),
```

```
        'avg', d2l.numpy(pool2d(Z, (2, 2), 'avg', stride=2)).tolist())
```

```
response at (1, 0): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 1): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 2): max [[0.0, 1.0], [0.0, 0.0]] avg [[0.0, 0.25], [0.0, 0.0]]
```

応答を列 0 から列 1 へシフトさせても、両方の位置が同じ 2×2 ブロック内に収まるため、どちらのプーリング出力もまったく変化しない。さらに 1 ピクセル進めて列 2 へシフトさせるとブロック境界をまたぐため、プーリングされた応答は隣接する出力セルへとジャンプする。したがって、水平方向の 1 ピクセルのシフトは、応答が元のブロック内にとどまる場合（全開始列の半分）には吸収され、そうでない場合には出力の位置を 1 つずらす。ゆえに局所プーリングは、表現にある種の小さなシフトに対する**耐性**を持たせるのであり、厳密に不変にするわけではない。Figure 6.19 に、4 つの列位置すべてにわたる最大プーリングの出力を示す。

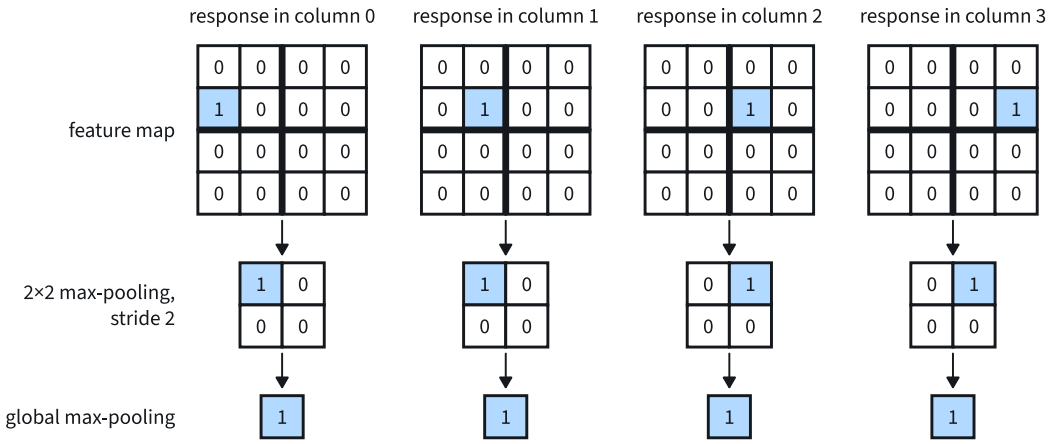


Figure 6.19: 4×4 の特徴マップの 4 つの列を移動する単一の応答（上段）、ストライド 2 による 2×2 最大プーリング後のマップ（中段）、およびその大域的最大値（下段）。太線はプーリングブロックを表す。プーリングされた応答は、元の応答がブロック内にとどまる間は同じ位置に保たれ、次のブロックへ境界を越えるとジャンプするが、大域的最大値は決して動かない。

また、最大プーリングと平均プーリングとでブロックの要約の仕方が異なることにも注目しよう。最大プーリングはブロック内のどこであっても活性化値（activation）の強さ（1）をそのまま保持するため、実質的にその近傍のどこかに特徴が現れたかどうかを問うていることになる。一方、平均プーリングでは特徴が 4 つのセルのうち 1 つしか占めていないため 0.25 が出力され、特徴が空間的に占める割合を測るとともに、周囲の 0 によって強い孤立応答を薄めてしまう。

また、ストライド 2 によるサブサンプリングには、特徴がサンプリング境界をまたぐ際に信号処理上の厄介な落とし穴が生じる。極端な例として、高周波で振動する信号 $1, 0, 1, 0, \ldots$ から 1 つおきにサンプルを取り出すと、1 ピクセルのシフト次第で出力がすべて 1 になるかすべて 0 になるかが切り替わってしまう。サンプリングレートで分解できる限界より速く信号が変化するときに生じるこの歪みは**エイリアシング**と呼ばれ、プーリングか畳み込みかを問わず、ストライド 2 のあらゆるダウンサンプラに影響を及ぼす。サブサンプリングの前に小さな平滑化フィルタを適用するとエイリアシングは軽減される (Zhang 2019)。この手法は本節末の演習問題で取り上げる。

歴史的には、シフト耐性こそがプーリングの主な動機であった。Fukushima の Neocognitron (Fukushima 1980) は、特徴を検出する細胞の層ごとに、局所近傍内のどこかに特徴が現れれば

発火するプーリング様の細胞を対にして配置した。最大プーリングの初期の形態は音声認識で登場し (Yamaguchi et al. 1990)、その後、計算論的神経科学において視覚野における階層的な証拠集約のモデルとして最大プーリングが提案された (Riesenhuber and Poggio 1999)。

6.5.3 グローバルプーリング

画像内の**どこかに** Waldo が現れているかを問う際、6.1 節で求めた完全な平行移動不変性（translation invariance）はどうなるだろうか。今見たとおり、小さな固定ウィンドウではそれを保証できない。しかし、各特徴マップの空間領域**全体**を覆うようにプーリングウィンドウを広げれば実現できる。すべての空間位置にわたる最大値や平均は、応答がどこにあってもまったく同じスカラーになるからである（Figure 6.19 の下段）。この演算は**グローバルプーリング**と呼ばれ、各チャネルを単一の数値へと潰す。

しかし CNN 全体の中では、グローバルプーリングはゼロパディングされた畳み込み層の上に置かれており、そうした畳み込み層が厳密にシフト同変であるのは画像の境界から離れた領域に限られる（6.1 節）。6.3.1.3 節で見たように、内部のピクセルは 3×3 カーネルの 9 つのウィンドウすべてに関与するのに対し、境界のピクセルは 6 つ（四隅のピクセルは 4 つ）にしか関与しない。そのため、孤立した応答に対する大域的最大値は列によらず一定に保たれる一方、大域的な平均は特徴がパディングされた境界へ移動すると低下する。したがって、最後にグローバルプーリングを置くストライド 1 の畳み込み層の積み重ねは、顕著な内容が境界から離れたままであるようなシフトに対しては厳密に不変であり、それ以外では近似的に不変となる。ストライドを持つ層はこの保証を弱める。上で見たエイリアシングのため、不変性が厳密に成り立つのは各層のストライドの積の倍数だけシフトした場合に限られるからである。

第 7 章の現代的なアーキテクチャでは、プーリングが担う 2 つの役割は切り離されることが多い。中間層での空間的ダウンサンプリングは、固定された最大値や平均に決めてしまうのではなく各ウィンドウへの重み付けを学習するストライド 2 の畳み込みで処理されることが多く、ネットワークの末尾では最終の線形分類器の直前に**グローバル平均プーリング**（global average pooling）を置く (Lin et al. 2013)。次節で構築する古典的アーキテクチャである LeNet は、これら両方の慣行より前の時代のものである。

6.5.4 パディングとストライド

プーリングは畳み込みとまったく同じ方法で空間グリッド上にウィンドウをスライドさせるため、深層学習フレームワークでは同じウィンドウ形状・パディング・ストライドの引数でプーリング層を指定でき、出力形状も 6.3 節と同一の式に従う。すなわち、両端に p ピクセルずつパディングした長さ n の軸に対し、ウィンドウサイズ k とストライド s を適用すると、その軸に沿った出力の数は $\lfloor (n + 2p - k)/s \rfloor + 1$ となる。唯一の微妙な違いは、パディングされた境界をどの**値**で埋めるかである。組み込みのプーリング層でこれらの挙動を確認するため、まず 0 から 15 までの要素を持つ単一の 4×4 特徴マップを格納した 4 次元テンソルを構築しよう。

```
# (example, channel, height, width)
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

プーリングは通常、重なりのない領域を要約するために使われるので、大半のフレームワークではウィンドウが重ならずに入力を敷き詰めるよう、デフォルトでストライドをプーリングウィン

ドウのサイズと等しく設定する。この 4×4 の入力に対しては、ストライド 3 の 3×3 ウィンドウは 1 回しか収まらず、左上の 3×3 ブロックの最大値（10）を返す。プーリングには学習可能な重みがないため、この層ではパラメータの初期化が不要である。

```
pool = nn.MaxPool2d(3)  # The stride defaults to the window shape
pool(X)
```

```
tensor([[[[10.]]]])
```

重なりのあるウィンドウや明示的なパディングを使いたい場合は、これらのデフォルト設定を上書きできる。パディングを 1、ストライドを 2 に設定すると、高さと幅の両方に沿ってサイズ $\lfloor (4+2-3)/2 \rfloor + 1 = 2$ の出力が得られる。

```
pool = nn.MaxPool2d(3, padding=1, stride=2)
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

プーリング層のパディングされた境界には、どのような値を入れるべきだろうか。畳み込みでは、0 を掛けても重み付き和に何も寄与しないため、0 が自然な単位元（neutral element）となる。しかし最大プーリングでは、単位元は $-\infty$ である。パディングされたセルが実際の活性化値を上回ることは決してないため、各出力はウィンドウ内にある本来の入力要素の最大値と必ず一致する。上の例では、ウィンドウが覆う本来の行および列は $\{0, 1\}$ と $\{1, 2, 3\}$ であり、出力として 5、7、13、15 が得られる。また、$-\infty$ によるパディングは**値**に関するシフト同変性も保つ。すなわち、すべての入力要素に定数を加えると、プーリング後の各出力もまったく同じ定数だけシフトする。要素がすべて負である `X - 16` でこれを確認できる。

```
pool(X - 16)
```

```
tensor([[[[-11.,  -9.],
          [ -3.,  -1.]]]])
```

予想どおり、すべての出力がちょうど 16 だけ減少する。もし仮にこの層が 0 でパディングしていたなら、境界に接する 3 つのウィンドウは 0 を返してしまい、単に 0 がすべての負の入力より大きいというだけの理由で結果が損なわれていたはずである。

平均プーリングには、これに相当する中立な定数が存在しない。パディングされた 0 を分母に含めると、境界での平均に 0 へ引き寄せるバイアスが生じるため、各ウィンドウ内に収まる本来の入力要素の個数だけで割る方がすっきりとした方法である。

ここでの平均プーリングはデフォルトでパディングされた 0 を個数に含めるが（`count_include_pad=True`）、`count_include_pad=False` を指定すると本来の要素のみの平均をとる。

6.5.5 複数チャネル

畳み込み層がすべての入力チャネルにわたって和をとることでチャネル間の情報を混合するのに対し（6.4 節）、プーリング層は各チャネルに対して独立に作用する。その結果、出力のチャネル数は常に入力のチャネル数と完全に一致する。この挙動を確認するため、チャネル軸に沿って `X` と `X + 1` を連結し、2 チャネルのテンソルを作成しよう。

```
X = d2l.concat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

先ほどのプーリング演算（パディング 1、ストライド 2）を適用すると、2 つの出力チャネルが得られる。1 つ目のチャネルは 5、7、13、15 を再現し、2 つ目のチャネルはそれらに 1 を加えた値を返す。

```
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

プーリングは学習可能な重みを持たないため、それ自体でパターンを見つけ出すことはできず、先行する畳み込み層が検出した特徴を要約することしかできない。畳み込み、パディング、ストライド、ダイレーション、複数チャネルの特徴マップ、そしてプーリングがすべて揃ったことで、完全な畳み込みニューラルネットワークを組み立てて訓練し、これまでの MLP と直接比較するために必要な構成要素がすべて手に入ったことになる。

6.5.6 まとめ

プーリング層は、入力特徴マップの各空間ウィンドウをその最大値または平均によって要約する。学習可能なパラメータを一切追加せず、各チャネルを独立に処理し、畳み込みとまったく同じパディングおよびストライドの出力形状の規則に従う。また、最大プーリングではパディングされた位置が最大値に決して影響しないよう、境界を $-\infty$ でパディングする。ストライド 2 の 2×2 プーリング層は各軸に沿った空間解像度を半分にし、ブロック内にとどまる小さなシフトを吸収するが、応答がブロック境界をまたぐとやはりジャンプする。最大プーリングは近傍内のピークの活性化値を保持するのに対し、平均プーリングは孤立したピークをウィンドウ全体で薄めてしまう。最後に、グローバルプーリングは各特徴マップ全体を単一のスカラーへと潰す。ストライド 1 の層の上に置けば、内部のパターンに対して完全な平行移動不変性（パディングされた境界付近では近似的な不変性）が得られ、ストライドを持つ層と組み合わせた場合、不変性が厳密に成り立つのは各層のストライドの積の倍数だけシフトした場合に限られる。

6.5.7 演習問題

1. **プーリングの代替手法**。
    1. 独立した最小プーリング層を用意する必要があるか、それとも本節ですでに扱った演算を使って表現できるか。

2. 温度 $\tau > 0$ を持つ**ソフトマックスプーリング**は、ウィンドウ内の要素 $x_1, \ldots, x_m$ を次のソフトマックス重み付き平均で置き換える。

$$\sum_{i=1}^{m} x_i \frac{\exp(x_i/\tau)}{\sum_{j=1}^{m} \exp(x_j/\tau)}. \tag{6.21}$$

   $\tau \to 0$ および $\tau \to \infty$ としたときの極限を求めよ。また、最大プーリングと比べてソフトマックスプーリングが追加で要するコストを 1 つ挙げよ。

2. **エイリアシングとブラープール**。単純なストライド 2 のダウンサンプリングは、入力の要素を 1 つおきに残す。周波数を入力 1 サンプルあたりの周期数で測ることにすると、1 次元信号 $(1, 0, 1, 0, 1, 0)$ と $(0, 1, 0, 1, 0, 1)$ の周波数はいずれも $\frac{1}{2}$ である。
   1. 1 マス分のシフトだけが異なるこれら両方の信号に単純なストライド 2 のダウンサンプリングを適用し、2 つの出力を比較せよ。なぜこの挙動は**エイリアシング**と呼ばれるのか。
   2. ストライド 2 のサブサンプラが忠実に表現できる入力周波数はどの範囲か。
   3. ブラープール (Zhang 2019) は、サブサンプリングの前に小さな平均化フィルタを適用することでエイリアシングを抑える。2 タップのボックスフィルタ $(\frac{1}{2}, \frac{1}{2})$ を用いたブラープールが上の 2 つの信号に対して何を計算するか求めよ。これは本節のどのプーリング演算と一致するか。

3. [code] **最大プーリングの逆伝播**。$k_\mathrm{h} \times k_\mathrm{w}$ ウィンドウおよびウィンドウサイズと等しいストライドを持つ 2 次元最大プーリングについて、入力に関する勾配を導出し、自動微分を使わずに実装せよ。本節の組み込み最大プーリング演算に対して自動微分が計算する勾配と、小さなランダムテンソル上で相対誤差 10^{-4} 未満で一致することを検証せよ。

   Stanford CS231n、課題 2、ConvolutionalNetworks の素朴な最大プーリングの逆伝播より改変。

4. **ReLU による最大プーリング**。●畳み込みは線形であるため、それ単独では最大値を計算できない。
   1. $\max(a, b)$ を、アフィン演算と $\mathrm{ReLU}(x) = \max(0, x)$ のみを用いて表せ。
   2. これを利用して、畳み込みと ReLU 層の積み重ねとして最大プーリングを実装せよ。
   3. その構成が 2×2 ウィンドウおよび 3×3 ウィンドウに対して使用する層ごとのチャネル数と ReLU 層の数を数えよ。ReLU 層の数はウィンドウ内の要素数に伴ってどのように増加するか。

6.6 LeNet

ここまでで、畳み込みニューラルネットワーク（convolutional neural network）の構成要素がすべて出そろった。すなわち、畳み込み層（6.2 節）、パディングとストライド（6.3 節）、複数チャネル（6.4 節）、およびプーリング（6.5 節）である。いよいよこれらを組み合わせて完全な画像分類器を構築し、これまでの議論を検証してみよう。2 次元の空間構造を尊重することで、実際に標準的な MLP を上回る性能が得られるのだろうか。

ベースラインとするのは 4.2 節の MLP であり、Fashion-MNIST において 30 エポック後に約 87% の検証正解率に達する。MLP が各画像を順序のない 784 個の数値へと平坦化する（6.1 節）のに対し、CNN はピクセルの格子構造を保ったまま、共有された局所カーネルをすべての空間位置に適用する。2 つのアーキテクチャを公平に比較するため、同じ交差エントロピー損失（cross-entropy loss）、SGD オプティマイザ、30 エポックという計算予算で CNN を訓練する。正解率という 1 つの数値だけでは全体像は見えないため、第 1 畳み込み層が学習したフィルタを観

察するとともに、ピクセルをシャッフルした画像を用いた対照実験によって CNN の優位性がどこから生じるのかを確かめる。

本節で実装するアーキテクチャは **LeNet** である。手書き数字を認識するために AT&T Bell Labs の同僚らとともにこれを開発した Yann LeCun にちなんで名付けられた (LeCun, Léon Bottou, et al. 1998)。LeNet は、誤差逆伝播法（backpropagation）で訓練された最初期のエンドツーエンド畳み込みネットワークの 1 つに端を発する、10 年におよぶ研究の集大成であった (LeCun et al. 1989)。1998 年までに LeNet-5 は手書き数字に対して 1% 未満の誤り率を達成し、当時の機械学習を席巻していたサポートベクターマシンに匹敵する性能を示した。またその派生モデルは、銀行の小切手や ATM 預け入れ時の手書き金額を読み取るシステムとして商用展開された。エンドツーエンドで学習したネットワークが、人手で入念に設計されたビジョンパイプラインに取って代われることを、LeNet は早い段階で説得力をもって実証したのである。

6.6.1 LeNet のアーキテクチャ

大まかに見ると、LeNet は Figure 6.20 に示すように 2 つの段階へ明確に分かれる。2 つの畳み込みブロックから構築された**畳み込みエンコーダ**（convolutional encoder）と、それに続く 3 つの全結合層（fully connected layer）からなる**全結合分類ヘッド**（dense classification head）である。

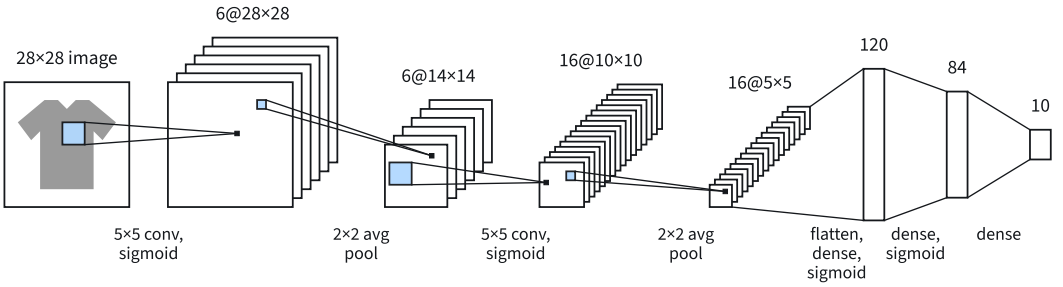


Figure 6.20: 1 枚の 28×28 Fashion-MNIST 画像に対する LeNet 内のデータフロー（特徴マップは実寸比で描画）。網掛けされた各ウィンドウから、矢印が指す次の層の濃いユニットが計算される。3 つの全結合層はそれぞれ 120、84、10 個の出力を持つ。

各畳み込みブロックは、5×5 の畳み込み層およびシグモイド活性化関数と、ストライド 2 の 2×2 平均プーリング層を組み合わせたものである。第 1 畳み込み層は 6 チャネルを出力し、第 2 畳み込み層は 16 チャネルを出力する。

エンコーダの出力は、チャネルファーストの形状 (`batch_size, channels, height, width`) を持つ。

畳み込みエンコーダを全結合ブロックに接続するため、各データ例の空間的な特徴マップを 1 本のベクトルへと平坦化し、ミニバッチ内の例ごとに 1 行を持つ 2 次元行列を作る。続いて 3 つの全結合層がこのベクトルを 120 個および 84 個の隠れユニットを経由して、ソフトマックス回帰と同様にクラスごとの 10 個の出力ロジットへと射影する。

なお、ここで扱うのは LeNet-5 の標準的な教育用バージョンであり、当時の実装を細部まで忠実に再現したものではない。1998 年のネットワークにおける畳み込みとプーリングの交互配置や各層の幅（6、16、120、84）は維持しつつ、今日ではほとんど使われない当時の細かな仕様（スケーリングされた tanh 活性化関数、学習可能なプーリングのスケールとバイアス、2 つの畳み込み間の疎なチャネル結線、動径基底関数（radial basis function）による出力ユニットなど）は簡略化している。ただし、コードには歴史的な名残が 1 つ現れている。元の LeNet-5 は 32×32 の入力を受け取っていたという点である。MNIST の数字画像は 28×28 であるが、1998 年の論文

では数字の端近くにある筆画も上位の特徴検出器の受容野（receptive field）に収まるよう、四方を 2 ピクセル分の背景で囲んでいた。本節の第 1 畳み込み層における 2 ピクセルのパディングはこれと同じ役割を果たし、最初の 5×5 畳み込みは再び 28×28 の出力を生成する。

ネットワークを定義する前に、`init_cnn` を作成し、すべての畳み込み層と線形層の重みを Xavier 一様初期化（4.4.2.1 節）で初期化する。

```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

現代の深層学習フレームワークを使えば、`LeNet` の組み立てに必要なのは、2 つの畳み込み・プーリング段階と 3 つの全結合層を連結し、すべてのバイアスを 0 に設定する単一の `Sequential` ブロック `self.net` だけである。

```
class LeNet(d2l.Classifier):
    """The layout of LeNet-5, with sigmoid activations and average pooling."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        init_cnn(module)  # Xavier-uniform weights
        if type(module) in (nn.Linear, nn.Conv2d) and module.bias is not None:
            nn.init.zeros_(module.bias)
```

6.6.2 各層の形状

パディングの値が 1 つずれたり平坦化層を入れ忘れたりするだけで後続すべてのテンソル形状が変わってしまうため、訓練を始める前に中間層の形状を確認しておくのがよい習慣である。Figure 6.21 は、簡略記法で表した LeNet の図に、各境界で想定されるテンソル形状を添えたものである。実装がこの図と一致するか確かめるため、単一チャネルの 28×28 ダミーテンソルをモデルに通し、各層を通過した後の出力形状を表示する。次章のより深いアーキテクチャでも再利用できるよう、この診断処理を `layer_summary` メソッドとして `d2l.Classifier` に実装しておこう。

```
@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = d2l.randn(*X_shape)
    for layer in self.net:
```

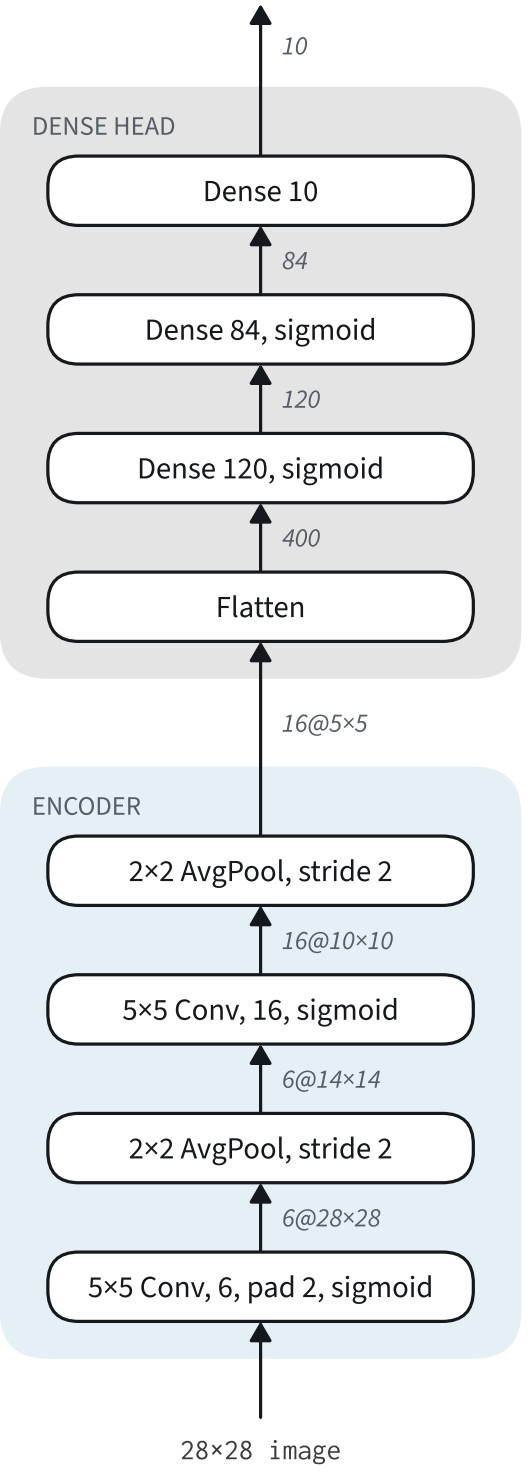


Figure 6.21: LeNet の簡略記法。各矢印の横には、そこを流れるテンソルの形状が「チャネル数 @ 高さ × 幅」の形式で記されている。

```
            X = layer(X)
            name = getattr(layer, '__name__', type(layer).__name__)
            print(f'{name:<22} {tuple(X.shape)}')

    model = LeNet()
    model.layer_summary((1, 1, 28, 28))
```

```
Conv2d                 (1, 6, 28, 28)
Sigmoid                (1, 6, 28, 28)
AvgPool2d              (1, 6, 14, 14)
Conv2d                 (1, 16, 10, 10)
Sigmoid                (1, 16, 10, 10)
AvgPool2d              (1, 16, 5, 5)
Flatten                (1, 400)
Linear                 (1, 120)
Sigmoid                (1, 120)
Linear                 (1, 84)
Sigmoid                (1, 84)
Linear                 (1, 10)
```

表示された形状は Figure 6.21 と完全に一致しており、6.3 節で導いた出力形状の規則に従っている。パディングなしの 5×5 カーネルは高さと幅をそれぞれ 4 減らすため、第 1 畳み込み層では各辺に 2 ピクセルのパディングを加えて 28×28 の空間解像度を維持し、第 2 畳み込み層ではパディングを行わずに 14×14 を 10×10 へと縮小する。各 2×2 平均プーリング層は両方の空間次元を半分にする（$28 \to 14$ および $10 \to 5$）。その一方で、エンコーダが空間解像度と引き換えにより豊かな特徴表現を獲得するにつれ、チャネルの深さは 1 から 6、さらに 16 へと拡大する。最後の $16 \times 5 \times 5$ テンソルを平坦化すると 400 次元のベクトルが得られ、全結合ブロックがこれを段階的に 10 個のクラスロジットへと圧縮する。

6.6.3 パラメータと積和演算

すべての層の出力形状が確認できたところで、パラメータのメモリ量と演算コストの両方を集計してみよう。6.4.2 節で見たように、入力チャネル数 c_{i}、出力チャネル数 c_{o}、カーネルサイズ $k \times k$ の畳み込み層は、$c_{\mathrm{i}} c_{\mathrm{o}} k^2$ 個の重みと c_{o} 個のバイアスを保持し、出力の空間位置ごとにすべての重みを 1 回ずつ再利用する。対照的に、全結合層は 1 つの例につき各重みを 1 回しか使わない。Table 6.1 に LeNet 全体での内訳を示す。

Table 6.1: LeNet の各層における 1 例あたりのパラメータ数と積和演算数。プーリング層と活性化層はパラメータを持たない。

層	出力	パラメータ数	積和演算数
5×5 畳み込み、1 から 6 チャネル	6@28×28	$150 + 6 = 156$	$150 \times 784 = 117{,}600$
5×5 畳み込み、6 から 16 チャネル	16@10×10	$2{,}400 + 16 = 2{,}416$	$2{,}400 \times 100 = 240{,}000$
全結合、400 から 120	120	$48{,}000 + 120 = 48{,}120$	$48{,}000$
全結合、120 から 84	84	$10{,}080 + 84 = 10{,}164$	$10{,}080$
全結合、84 から 10	10	$840 + 10 = 850$	840
合計		$61{,}706$	$416{,}520$

この内訳から、CNN の設計全般に繰り返し現れるメモリと計算量の際立った非対称性が浮かび上

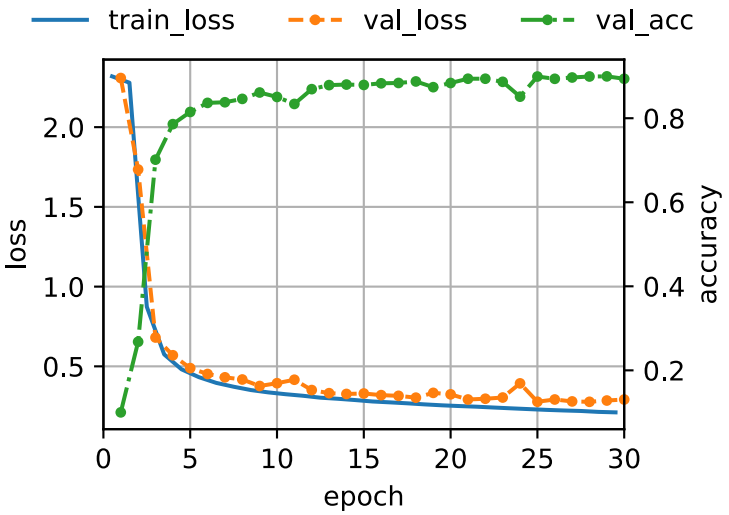
がる。2 つの畳み込み層が占めるのは LeNet のパラメータのわずか 4% にすぎないが、カーネルの各重みが空間格子全体にわたって評価されるため、積和演算の 86% を担っている。逆に、最初の全結合層だけでパラメータの 78% を保持する一方、演算量への寄与は 12% にとどまる。先に扱った Fashion-MNIST の MLP（パラメータ数 203,530、画像あたりの積和演算数約 203,000）と比べると、LeNet はパラメータ数が 3 分の 1 未満である一方、約 2 倍の計算を実行する。

6.6.4 訓練

MLP のときと同様に、ミニバッチサイズ 128（MLP の 256 の半分）のミニバッチ SGD によって交差エントロピー損失を最小化し、LeNet を 30 エポック訓練する。ただし、1 つだけ大きく変更しなければならないハイパーパラメータがある。学習率（learning rate）である。シグモイド活性化関数の導関数は決して $1/4$ を超えないため（4.1.4 節）、第 1 畳み込み層へ逆伝播する勾配は 4 つの連続するシグモイドによって減衰する。MLP と同じ 0.1 の学習率では訓練が極端に遅くなり、検証正解率は最初の 2〜4 エポックの間ランダム予測の水準付近に留まり、10 エポック後でも約 73% までしか上がらない。したがって、学習率を 2.0（より小さな勾配スケールを反映し、経験的に調整した 20 倍の値）へと引き上げる。

5.7.4 節と同様、GPU が利用可能な場合には `d2l.Trainer` が自動的に GPU 上で訓練を実行する。

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=2.0)
trainer.fit(model, data)
```



30 エポック後、LeNet はパラメータ数が 3 分の 1 未満であるにもかかわらず、MLP の約 87% を上回る約 88〜90% の検証正解率を達成する。

訓練曲線の立ち上がりにも注目してほしい。ランダム初期化（random initialization）の違いだけで、正解率が直ちに上昇することもあれば、損失が $\ln 10 \approx 2.3$ 付近に張り付いたまま数エポックにわたってランダム予測の水準（10%）付近をさまよい、その後急激に立ち上がることもある。この初期のプラトーの間、ネットワークはどの画像に対してもほぼ同一の一様な確率を予測している。4 つのシグモイド層を通過するうちに、ロジットにおける入力依存の変動が当初は小

さくなりすぎて、クラスを分離できないからである。この信号消失の効果については後述の演習問題で定量化する。また、同じプラトーがピクセルシャッフル実験でも再び現れることになる。

6.6.4.1 学習された特徴

ソフトマックス回帰では各クラスの重みをそのまま 28×28 のテンプレート画像として可視化でき（3.3 節）、同様に MLP の各隠れユニットも固定されたピクセル位置に重みを付けていた（4.2 節）。対照的に、畳み込みカーネルは特定のピクセルに紐づいておらず、単なる 5×5 の重みパッチであるため単独では解釈しにくい。そこで第 1 畳み込み層が何を発見したのかを探るため、まず検証ミニバッチからスニーカー（クラス 7）1 枚とシャツ（クラス 6）1 枚を取り出し、2 枚の画像からなるテンソル `X` に格納する。

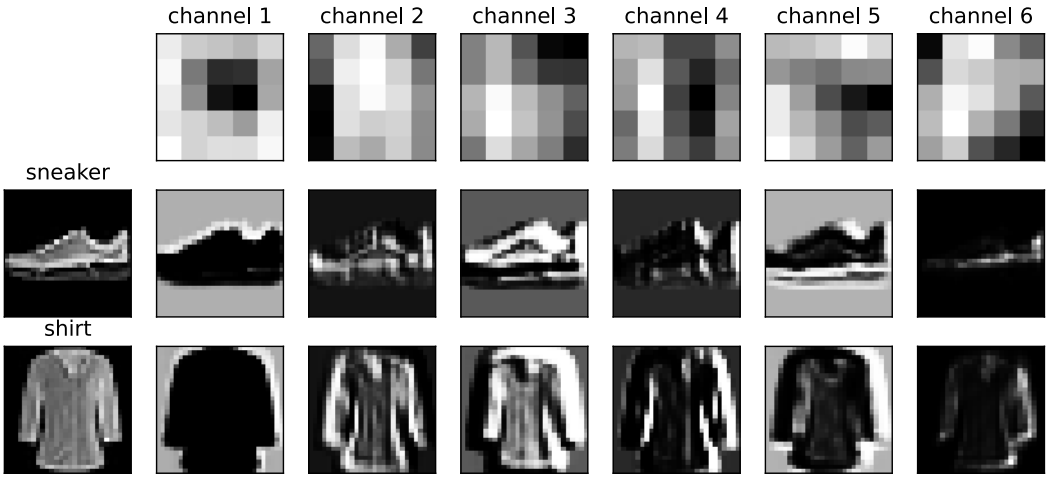
```
X, y = trainer.prepare_batch(next(iter(data.val_dataloader())))
labels = d2l.numpy(y)
X = d2l.stack([X[int(np.argmax(labels == c))] for c in (7, 6)], 0)
```

次に、`X` を第 1 畳み込み層とそのシグモイド活性化関数に通し、6 枚の 28×28 特徴マップ `maps` と学習済みの 5×5 カーネル `kernels` を抽出する。

```
maps = model.net[1](model.net[0](X))  # The first convolution and its sigmoid
images, maps = d2l.numpy(X[:, 0]), d2l.numpy(maps)
kernels = d2l.numpy(model.net[0].weight[:, 0])  # Six 5x5 kernels
```

これで、最上段に 6 個の `kernels` を並べ、その下に入力画像 `images` とシグモイド通過後の 6 枚の特徴マップ `maps` を表示できる。各パネルはそれぞれの最小値（黒）から最大値（白）まで独立にスケーリングしている。

```
names = data.text_labels([7, 6])
axes = d2l.show_images(
    [kernels[0], *kernels, images[0], *maps[0], images[1], *maps[1]], 3, 7,
    titles=['', *[f'channel {c}' for c in range(1, 7)],
            names[0], *[''] * 6, names[1], *[''] * 6], cmap='gray')
axes[0].set_visible(False)  # No kernel above the input images
```



訓練を繰り返しても、一貫して 2 種類の明確に異なる特徴マップ群が立ち現れる。一部のチャネルは局所的な明るさや前景の検出器として働き、衣類の塗りつぶされたシルエット（暗い背景に明るい領域、あるいはその反転）を再現する。残りのチャネルは衣類の輪郭、とりわけ左右の境界に沿って選択的に強く反応する。これらの 5×5 カーネルを調べるとその仕組みがわかる。明るい列と暗い列が隣り合って対をなしており、6.2.2.2 節で人手により設計した $(1, -1)$ 垂直エッジカーネルの複数行版を学習によって獲得しているのである。以前は勾配降下法（gradient descent）にエッジフィルタを学習させるため明示的な正解エッジマップを与える必要があったが、ここでは生の衣類ピクセルとクラスラベルだけから、エッジとシルエットが視覚的分類に有用な構成要素であることを LeNet 自らが発見している。（6 つのチャネルへの具体的な役割分担は実行ごとに異なる。）

6.6.4.2 シャッフルされたピクセル

MLP に対する 1〜3 パーセントポイントの改善を、どの程度重く受け止めるべきだろうか。このマージンだけを見ればノイズのせいかもしれない。MLP の検証正解率はエポックごとに約 1 ポイント変動するうえ（4.2 節）、LeNet は深さ、活性化関数（activation function）、学習率も異なっているからである。より純粋な対照実験は、畳み込みだけが活用している唯一の情報、すなわち空間的隣接性を取り除くことである。本章の冒頭で述べた思考実験に立ち返り、784 個のピクセル位置に対する固定されたランダムな置換を 1 つ生成し、まったく同じ並び替えをすべての訓練画像と検証画像に適用しよう。

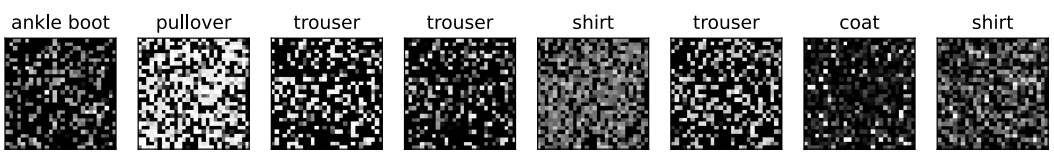
```
perm = np.random.permutation(28 * 28)  # One fixed order for every image

def shuffle_pixels(images):
    """Rearrange the 784 pixels of every image into the order perm."""
    return images.reshape(len(images), -1)[:, perm].reshape(images.shape)
```

それでは `shuffle_pixels` を使い、この共通の置換 `perm` を `data_shuffled` の訓練データと検証データのすべての画像に適用しよう。

```
data_shuffled = d2l.FashionMNIST(batch_size=128)
for split in (data_shuffled.train, data_shuffled.val):
    split.data = shuffle_pixels(split.data)  # The stored 8-bit images
```

`data_shuffled` から検証ミニバッチを取り出して可視化すると、784 個のピクセル座標を置換したことで判別可能な衣類の輪郭がすべて失われ、人間の目には純粋な砂嵐ノイズにしか見えないパターンになっていることが確認できる。

```
data_shuffled.visualize(next(iter(data_shuffled.val_dataloader())))
```

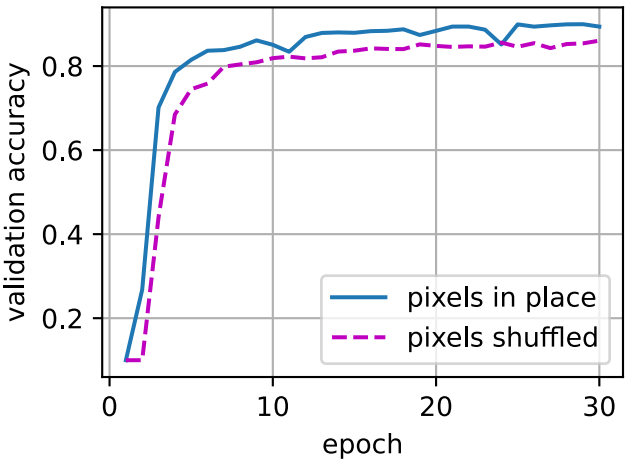


MLP はこの置換の影響をまったく受けない。その理由を証明するのに再訓練すら不要である。入力座標を置換しても、第 1 層の重み行列 $\mathbf{W}^{(1)}$ の行が入れ替わるにすぎないからである。各重みは独立同分布で初期化されるため、シャッフルされたピクセルでの訓練は元の画像での訓練と統計的に同一であり、同じ約 87% の正解率をもたらす。しかし LeNet にとっては、シャッフルによ

って空間的局所性が破壊され、すべての 5×5 カーネルが元の衣類上にランダムに散らばった 25 ピクセルを読み取ることになる。まったく同じ設定のままシャッフル後のデータセットで LeNet を再訓練し、両方の検証曲線を並べてプロットしてみよう。

```
curves = {'pixels in place': trainer.history['val_acc']}
trainer.board.display = False
trainer.fit(LeNet(lr=2.0), data_shuffled)
curves['pixels shuffled'] = trainer.history['val_acc']
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
for name, acc in curves.items():  # A single epoch can land a point low
    print(f'{name}: {sum(acc[-5:]) / 5:.3f} averaged over the last 5 epochs')
```

```
pixels in place: 0.897 averaged over the last 5 epochs
pixels shuffled: 0.853 averaged over the last 5 epochs
```



シャッフルされたピクセル上では、LeNet の正解率は典型的な実行で数パーセントポイント低下して 85〜86% 前後となり、MLP と同等かやや下回る水準に落ち込む。空間的な近傍構造が破壊されると LeNet の優位性は消失し、MLP に対するリードが畳み込みの帰納バイアス（inductive bias）から直接もたらされていたことが裏付けられる。

なお、初期のランダム予測水準のプラトーはここでも同様に予測不能な形で現れ得る。通常の画像では数エポックを超えることはまれだが、シャッフルされた画像では運悪く初期化が偏ると 30 エポックの大半にわたってランダム予測の水準付近から抜け出せなくなり、最終的な正解率がさらに押し下げられることがある。この不安定性はピクセルの順序そのものではなく 4 つのシグモイド活性化関数を重ねていることに起因しており、プラトーを即座に脱出した実行であっても、シャッフルされたピクセル上の LeNet が MLP を上回ることはない。

最後に、LeNet を 6.1 節の対称性の原理と改めて結びつけよう。理想的な画像分類器は厳密に平行移動同変な特徴層と厳密に平行移動不変な出力を備えるはずだが、LeNet はどちらも厳密には達成していない。パディングを伴う畳み込みは境界付近で同変性を破り、プーリングが吸収できるのはプーリングブロック内に収まるシフトに限られ（6.5 節）、最後の 5×5 マップを平坦化する処理はすべての空間セルに別々の全結合の重みを割り当てるからである。したがって LeNet のシフト耐性は部分的なものにとどまる。すべての衣類がきれいに中央配置された Fashion-MNIST には十分だが、制約のない一般のシーンには不十分である。

2 つの実験を総合すると、CNN が 3 分の 1 未満のパラメータ数で MLP に匹敵し、さらには上回ることができるのは、まさに重み共有（weight sharing）によって各カーネルが局所的な視覚パターンを一度学習するだけで格子上のどこででも検出できるからだとわかる。小さな 28×28 の画像ではこの効率性はささやかな利点にすぎないが、6.1 節で議論したメガピクセル級の写真（単一の全結合層だけで 10 億個の重みを要する）においては、同じ原理がそもそもモデルを構築できるかどうかを左右する。深い畳み込みネットワークを大規模環境で**訓練しやすくする**ことが次の課題であり、LeNet の 4 つのシグモイド層が見せた鈍く不安定な収束は、現代のアーキテクチャがなぜ 1998 年の姿から大きく進化しなければならなかったのかをすでに示唆している。

6.6.5 まとめ

LeNet は、空間解像度を段階的に縮小してより深いチャネル表現へと変換する畳み込みエンコーダと、3 層の全結合分類ヘッドを組み合わせたモデルである。その畳み込み層はパラメータのわずか 4% しか持たないにもかかわらず積和演算の 86% を実行し、一方で最初の全結合層だけでパラメータの 78% を占める。4.2 節の MLP と同じ 30 エポックの SGD 設定で訓練すると、LeNet は 3 分の 1 未満のパラメータ数で Fashion-MNIST において約 88〜90% の検証正解率（MLP は約 87%）に到達し、クラスラベルのみからシルエットフィルタや垂直エッジフィルタを自動的に学習する。ピクセル位置をシャッフルするとこの改善が消えることから、優位性の源泉が空間的局所性にあると特定できる。同時に、4 つのシグモイド層を通して勾配を伝播させるため、はるかに大きな学習率（0.1 ではなく 2.0）が必要となり、訓練初期に予測不能なプラトーが生じる。

エンコーダとヘッドを組み合わせる全体の基本設計は今なお礎であり続けているが、LeNet を構成する個々の部品はその後の発展でほぼすべて現代化された。ReLU がシグモイドに取って代わり、最大プーリングやストライド付き畳み込みが平均プーリングを置き換え、グローバル平均プーリングと単一の線形層の組み合わせが重い全結合ヘッドに取って代わり、正規化層が深いモデルの訓練を安定させたのである。次章では AlexNet（7.1 節）を皮切りに、ネットワークが深さ・幅・解像度の面でスケールアップしていく過程に沿って、こうしたアーキテクチャ上の進歩を 1 つずつたどっていく。

6.6.6 演習問題

1. [code] **現代的な構成要素**。本節の訓練設定を用い、以下の変更を加えて `LeNet` を再訓練し、30 エポック後の各変種の検証正解率を `LeNet` と比較せよ。
    1. 平均プーリングを最大プーリングに置き換えよ。
    2. シグモイド活性化関数を ReLU に置き換えよ。損失が発散するか正解率がランダム予測の水準にとどまる場合は、訓練が成功するまで学習率を下げ、使用した学習率を報告せよ。
    3. 両方の変更を同時に行い、その結果得られるモデルを現代化版 LeNet と呼ぶことにする。正解率の差の大部分をもたらしているのは、2 つの変更のうちどちらか。
    4. ●訓練開始時点の重みを用いて、`LeNet` と現代化版 LeNet のそれぞれについて、第 1 畳み込みカーネルに関する損失の勾配のノルムと、ミニバッチ全体にわたるロジットの標準偏差を測定せよ。4.4.2.1 節における Xavier 初期化の導出では、平均ゼロの入力と非線形性がないことを仮定している。シグモイド出力の平均とその傾き $\sigma'(0) = 1/4$ を用いて、測定結果、`LeNet` が必要とする学習率、および訓練初期にランダム予測の水準にとどまるエポックが生じる理由を説明せよ。
2. [code] **元の MNIST**。`MNIST` データクラスを `d2l.FashionMNIST` を改変して作成し、それぞれ Fashion-MNIST で用いた設定のまま、`LeNet` と 4.2 節の MLP の両方をそのデータ上で訓練せよ。LeCun, Léon Bottou, et al. (1998) は、MNIST における LeNet-5 のテスト誤差率が 1% 未満であると報告している。

    1. LeNet の検証誤差率をこの数値と比較せよ。両者は同じ画像上で測定されている（Fashion-MNIST と同様、本書の検証セットはこのデータセットの公式テストセットである）。
    2. 各データセットについて、2 つのネットワークの検証誤差率の差と比を計算せよ。3.2 節では、MNIST ではモデルクラス間の違いが見えにくくなると述べられている。2 つの指標はそれぞれこの記述を裏付けているか。

3. [code] **より深い層の特徴マップ**。本文では第 1 畳み込み層の特徴マップを描画した。

    1. プルオーバーとコートのように混同されやすい 2 つのクラスの検証画像について、訓練済み LeNet の第 2 畳み込み層のシグモイド通過後の特徴マップ 16 枚を表示せよ。各チャネルが何に反応しているかをまだ説明できるか。2 つのクラス間で最も違いが大きいのはどのチャネルか。
    2. 問題 1 の現代化版 LeNet を訓練し、Fashion-MNIST の検証画像、猫や自動車などの分布外の写真、および一様ランダムノイズのそれぞれに対して、2 つの畳み込み層それぞれの ReLU 通過後における活性化値の最大値を測定せよ。各写真は 28×28 のグレースケール画像に変換し、すべての入力を $[0, 1]$（d2l.FashionMNIST のピクセル値の範囲）に収めること。写真およびノイズに対する最大値は、検証画像で観測される範囲の内側に収まるか、それとも明確に外れるか。

4. [code] **シフトさせた衣類画像**。本文では、LeNet のシフトに対する耐性は部分的なものにすぎないと論じた。

    1. すべての検証画像を右に 1、2、4 ピクセルシフトさせ（空いた列はゼロで埋める）、シフトさせた各データセット上で訓練済み LeNet の検証正解率を測定せよ。ネットワークが耐性を持つのはどのシフトか。得られた結果を 2 つのプーリング層および全結合ヘッドと関連づけて考察せよ。
    2. LeNet の畳み込み層はすべての位置で重みを共有しているにもかかわらず、平行移動させた訓練画像のコピーを用いて訓練すると、シフトさせた検証画像に対する正解率が向上し得る理由を説明せよ。

    Michael Nielsen『Neural Networks and Deep Learning』第 6 章より改変。

5. [code] **全結合ヘッド**。●Table 6.1 が示すように、最初の全結合層は LeNet のパラメータの 78% を保持している。

    1. 平坦化層から出力層までの全結合ヘッド全体を、グローバル平均プーリングとそれに続く 16 ユニットから 10 ユニットへの単一の線形層に置き換えよ。このネットワークのパラメータ数はいくつになるか。訓練する前に、検証正解率が LeNet と比べて 1 パーセントポイント以上変化するかどうかを予測し、そのうえで再訓練して比較せよ。
    2. 今度は、平坦化された 400 次元の畳み込み出力を 10 出力の線形出力層へ直接接続せよ。このネットワークのパラメータ数はいくつになるか。再訓練し、縮小された 2 種類のヘッドの正解率を互いに、また事前の予測と比較せよ。この比較から、LeNet の表現能力がどこで使われているかについて何が言えるか。

    Michael Nielsen『Neural Networks and Deep Learning』第 6 章より改変。

Chapter 7

現代の畳み込みニューラルネットワーク

畳み込み層と LeNet の仕組みを押さえたところで（第 6 章）、本章では 2012 年の ImageNet における AlexNet のブレークスルーから、2020 年代初頭の畳み込みネットワークと Vision Transformer の対話に至るまで、10 年にわたる畳み込みアーキテクチャ設計の歩みをたどる。節目となった各モデルを孤立した産物として扱うのではなく、どのようなボトルネックがそれぞれの設計ステップを促したのかを問い、章全体を通して累積的なツールキットを引き継いでいく。

冒頭の各節では、個別に手作業で組まれた層からモジュール式の構成ブロックへと進む。AlexNet（7.1 節）は、大規模なラベル付きデータセットと GPU が結びついたことで、エンドツーエンドの特徴学習（feature learning）がいかに人手で設計されたビジョンパイプラインを凌駕したかを実証している。同モデルは ReLU 活性化関数、ドロップアウト、データ拡張（data augmentation）、および本章全体で再利用するウォームアップ付きコサイン学習率スケジュールを導入した一方で、畳み込みの計算量と全結合ヘッドのパラメータ数との間にある際立った不均衡も浮き彫りにした。続く VGG、Network in Network（NiN）、GoogLeNet（7.2 節）は、繰り返し可能な 3×3 ブロックによるネットワークの構成、1×1 畳み込みによるピクセルごとのチャネル混合、巨大な全結合ヘッドからグローバル平均プーリング（global average pooling）への置き換え、そしてステム・ボディ・ヘッド構造の内部での並列マルチスケール分岐の組み合わせによって、この不均衡に対処する。

中盤の各節が扱うのは、深さとハードウェア効率のスケーリングである。正規化層（7.3 節）は、ミニバッチ全体または個々のデータ例の内部で中間の活性化値を標準化し、隠れ層を重みのスケールに対して不感にすることで、実用可能な学習率の幅を何桁も広げる。残差ネットワーク（7.4 節）は、加算ショートカット $\mathbf{x} + g(\mathbf{x})$ により $g = 0$ で恒等写像（identity mapping）を保つことで、深いプレーンな積層ネットワークにおける最適化の劣化を解消し、さらにグループ畳み込み（ResNeXt）および特徴の連結（DenseNet）によってブロックを拡張する。効率的な畳み込みネットワーク（7.5 節）では、正解率と並ぶ重要な設計目標としてレイテンシ、メモリ、エネルギーを位置づけ、深さ方向分離可能な分解（depthwise-separable factorization）、逆ボトルネック（inverted bottleneck）、複合スケーリング（compound scaling）、Squeeze-and-Excitation ゲート、構造的再パラメータ化（structural re-parameterization）を導入するとともに、演算回数が GPU の実経過時間としばしば乖離する理由を明らかにする。

終盤の各節では、訓練手法とアーキテクチャを切り分ける。現代の訓練レシピ（7.6 節）では、AdamW、コサインウォームアップ、ラベル平滑化（label smoothing）、Mixup、CutMix、確率的深さ（stochastic depth）、重み平均化といった現代的な最適化と正則化だけでも、アーキテク

チャを変えない ResNet において ImageNet の正解率の数パーセントポイント分を担っていることを示す。このようなレシピを固定したうえで、ConvNeXt（7.7 節）は ResNet を段階的に現代化し、アテンションを用いずに階層型 Vision Transformer に匹敵する性能を達成する。そして畳み込みネットワークの設計空間（7.8 節）では、サンプリングされたモデル集団にわたる経験誤差の分布を比較することで、RegNet ファミリーの簡潔な構造規則を抽出する。

7.1 AlexNet と学習された画像表現

6.6 節では、LeNet が畳み込みフィルタから最終的な分類器に至るパイプラインのすべての段階を、生のピクセルからエンドツーエンドで訓練した。しかし、1989 年の LeCun の成果から 20 年間、主流のコンピュータビジョンでは特徴抽出（feature extraction）と分類を切り離していた。すなわち、SIFT (Lowe 2004)、SURF (Bay et al. 2006)、HOG (Dalal and Triggs 2005)、Bag-of-Visual-Words (Sivic and Zisserman 2003) といった人手で設計された記述子がピクセルを固定の特徴ベクトルへと写像し、それを線形分類器やカーネル分類器に入力していたのである。訓練損失は分類器のみを更新し、特徴を決して更新しなかったため、進歩は人間による特徴量エンジニアリングに依存していた。

転換点は 2012 年の ImageNet コンペティション（検証画像のうち正解ラベルがモデルの上位 5 件の予測に含まれない割合である **top-5 誤差**（top-5 error）によって 1,000 個の物体カテゴリを評価する競技会）で訪れた。**AlexNet** (Krizhevsky et al. 2012) は、8 層の畳み込みネットワークを生のピクセルからエンドツーエンドで訓練したモデルである。コンペティションへの出場システムはこのようなネットワーク 7 個のアンサンブルであり、top-5 誤差を（人手で設計された最良のパイプラインによる）26.2% から一躍 15.3% へと引き下げた。単一のネットワークだけでもすでに約 18% に達していた。アーキテクチャの面では AlexNet は LeNet を大規模化したものであるため、この劇的な勝利は 2012 年に揃った 3 つの推進要因を直接示している。すなわち、100 万枚規模のラベル付きデータセット（ImageNet）、数千万個の重みを数か月ではなく数日で訓練できるほど高速な GPU、そして ReLU 活性化関数、ドロップアウト、データ拡張（data augmentation）といった、深いネットワークを訓練可能にし暗記しにくくした重要なモデリング手法の 3 つである。

7.1.1 学習された特徴、データ、ハードウェア

人手で設計された記述子が見落とし、誤差逆伝播法（backpropagation）が ImageNet 上で発見したものは何だったのだろうか。Figure 7.1 に、AlexNet の最初の畳み込み層が学習した 96 個の 11×11 カーネルを示す。方向選択性を持つエッジ検出器や周波数検出器と並んで反対色ブロブ（color-opponent blob）が見られ、これらは分類損失の最小化のみによって見いだされ、より深い層によってテクスチャ、部品、物体カテゴリへと合成される。

7.1.1.1 データとハードウェア

6000 万個のパラメータを持つモデルは、PASCAL VOC（約 10^4 枚の画像）のような小規模なベンチマークをすぐに暗記してしまう。2012 年の ImageNet の訓練データ分割 (Deng et al. 2009; Russakovsky et al. 2015) は 1,000 カテゴリにわたる 120 万枚のラベル付き写真を提供し、さらに AlexNet はデータ拡張によって実効的なサンプル多様性を何倍にも高めた。具体的には、224×224 のランダムクロップと水平反転を 256×256 の画像から抽出することで、写真 1 枚あたり最大 $32 \cdot 32 \cdot 2 = 2{,}048$ 通りのビューを生成した。

ブレークスルーのもう一方の立役者はハードウェアであった。あらゆる空間位置とチャネルにわたってフィルタを畳み込む処理では、数百万回の独立した積和演算（multiply-add）を実行する。これはまさに、高帯域幅メモリに接続された数百個の並列コアで実行するために GPU

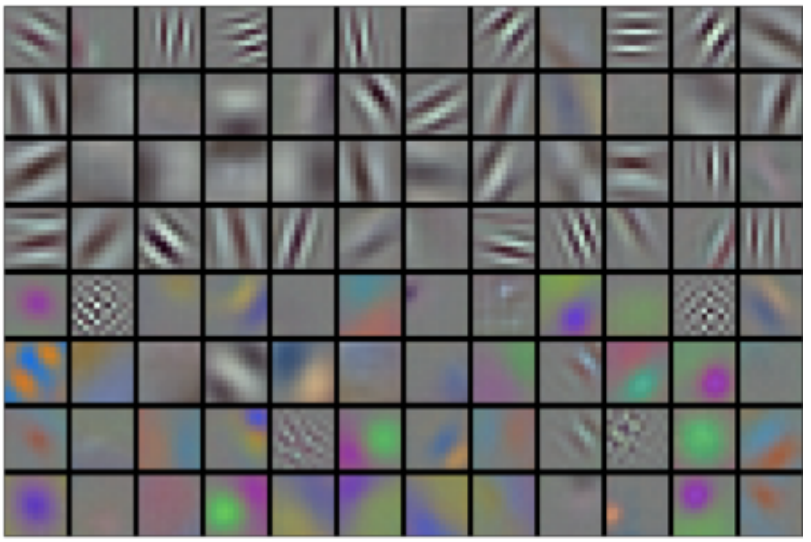


Figure 7.1: ImageNet 上で学習された、AlexNet の最初の畳み込み層における 96 個のカーネル。Krizhevsky et al. (2012) の厚意により転載。

（graphics processing unit）が設計されたワークロードそのものである。AlexNet に使われた 2 基の NVIDIA GTX 580 GPU は、それぞれ約 1.6 TFLOPS の演算性能と 192 GB/s のメモリ帯域幅を備えていたのに対し、当時のクアッドコア CPU は約 0.2 TFLOPS と 20 GB/s にとどまっていた。224×224 の RGB 画像 1 枚に対する AlexNet の 1 回の順伝播には約 10^9 回の積和演算がかかる。逆伝播を順伝播の 2 倍、積和演算 1 回あたり 2 FLOPs と数えると、90 エポックにわたって 120 万枚の画像を訓練するには次の演算量が必要になる。

$$10^9 \cdot 3 \cdot 2 \cdot 1.2 \times 10^6 \cdot 90 \approx 6.5 \times 10^{17} \text{ FLOPs} \tag{7.1}$$

これには 2 基の GTX 580 上で Krizhevsky のカスタム CUDA カーネル（cuda-convnet）を用いて 5〜6 日を要したが、CPU では優に 1 か月以上かかる計算量であった。

7.1.2 AlexNet のアーキテクチャ

Figure 7.2 では、LeNet（6.6 節）と、以下で Fashion-MNIST 上で訓練する AlexNet の変種を並べて比較している。どちらも畳み込みステージの積み重ねから密結合の分類ヘッドへとつなぐ同じ大局的な設計に従っているが、AlexNet はあらゆる次元を拡大するとともに、活性化関数、プーリング、正則化という 3 つの重要な構成要素を近代化している。

7.1.2.1 深さ、幅、ReLU、ドロップアウト

AlexNet はパラメータを持つ層の数を LeNet の 5 層から、5 つの畳み込み層と 3 つの密結合層からなる 8 層へと拡張している（Figure 7.2）。224×224 の入力は 28×28 のサムネイルより幅が 8 倍大きいため、第 1 層にはストライド 4 の 11×11 カーネル、第 2 層には 5×5 カーネル、第 3〜5 層には 3×3 カーネルを用い、第 1・第 2・第 5 の畳み込みの後に領域が重なり合う 3×3（ストライド 2）の最大プーリングを配置する。チャネル幅は 96 から 384 へと増え（LeNet では 6 と 16）、2 つの隠れ密結合層はそれぞれ 4096 ユニットへと広がる（LeNet では 120 と 84）。本書の実装では、2012 年当時の歴史的な事情による 2 つの要素は省略する。3 GB の GPU 2 基にわた

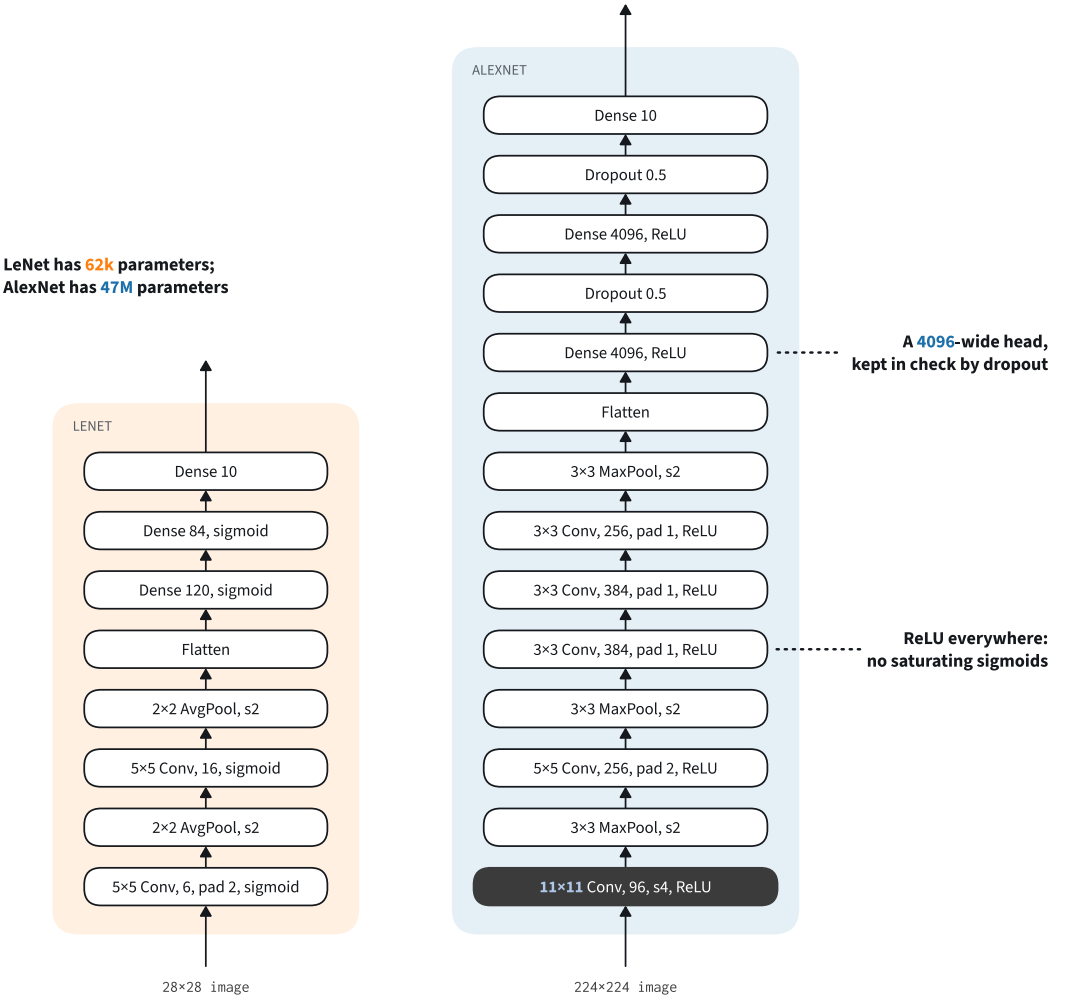


Figure 7.2: 6.6 節で構築した LeNet（左）と本節で構築する AlexNet（右）。いずれも単一チャネルの Fashion-MNIST 画像と 10 クラスを対象としており、入力は下端にある。AlexNet はより深く幅広であり、ReLU 活性化関数と最大プーリングを用い、密結合層をドロップアウトで正則化する。

る層の分割と、局所応答正規化（local response normalization）であり、後者は 7.3 節で扱うバッチ正規化（batch normalization）に取って代わられている。

この大規模化したネットワークを有効に機能させているのは、3 つのモデリング上の選択である。最初の 2 つは AlexNet 自身のものであり、3 つ目は本書による改良である。

1. **ReLU 活性化関数：**飽和するシグモイド（導関数が $\leq 1/4$ であるため深い層の勾配を 4^{-K} 倍に減衰させる）を $\mathrm{ReLU}(x) = \max(x, 0)$ に置き換えることで、活性なユニットでの導関数が 1 に保たれ、CIFAR-10 の訓練誤差 25% に tanh より 6 倍速く到達した (Krizhevsky et al. 2012)。
2. **ドロップアウト：**4096 ユニットの密結合層の両方の後にドロップアウト（$p = 0.5$、4.6 節）を適用することで、分類器ヘッドが訓練セットを暗記するのを防ぐ。
3. **He 初期化：**AlexNet は標準偏差 0.01 に固定したガウス分布から重みを抽出していた。本書ではこれを、3 年後に He et al. (2015) が ReLU ネットワーク向けに導出した初期化に置き換える。ReLU は入力の約半分を 0 にするため、重みを $\mathcal{N}(0, 2/n_{\text{in}})$（4.4.2.2 節、畳み込みでは $n_{\text{in}} = c_{\text{i}}k^2$）から抽出し、バイアスは 0 に初期化する。

遅延初期化（lazy）層は、最初のミニバッチによって形状が確定するまで重みを持たない。LeNet と同様、その後に `Trainer` がモデルの `weight_init` をすべての層に対して呼び出す。

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))

    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')
            nn.init.zeros_(module.bias)
```

コードが Figure 7.2 と一致していることを確認するため、単一チャネルの 224×224 のダミー画像を `layer_summary` に通し、各出力テンソルの形状を調べる。

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d                  (1, 96, 54, 54)
ReLU                    (1, 96, 54, 54)
MaxPool2d               (1, 96, 26, 26)
Conv2d                  (1, 256, 26, 26)
ReLU                    (1, 256, 26, 26)
```

```
MaxPool2d             (1, 256, 12, 12)
Conv2d                (1, 384, 12, 12)
ReLU                  (1, 384, 12, 12)
Conv2d                (1, 384, 12, 12)
ReLU                  (1, 384, 12, 12)
Conv2d                (1, 256, 12, 12)
ReLU                  (1, 256, 12, 12)
MaxPool2d             (1, 256, 5, 5)
Flatten               (1, 6400)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 10)
```

すべての次元が 6.3 節の公式と一致している。最初のストライド 4 の畳み込みは 224 を $\lfloor (224-11)/4 \rfloor + 1 = 54$ に写し、領域が重なり合う 3×3（ストライド 2）の各最大プーリング層は n を $\lfloor (n-3)/2 \rfloor + 1$ に写す（$54 \to 26 \to 12 \to 5$）。中間のパディング付き畳み込みは空間次元を保つため、最終的な畳み込みステージは平坦化されて $256 \times 5 \times 5 = 6400$ 要素のベクトルとなり、密結合の分類器ヘッドに入力される。

7.1.2.2 パラメータと積和演算

各層の形状が確認できたところで、LeNet（6.6 節）のときと同様にパラメータ数と算術演算コストを集計しよう。畳み込み層は $c_{\mathrm{i}} c_{\mathrm{o}} k^2$ 個の各重みをすべての空間出力位置で再利用するのに対し、全結合層（fully connected layer）は 1 例あたり各重みを 1 回だけ乗算する。単一チャネル・10 クラスの AlexNet についての結果を Table 7.1 にまとめる。

Table 7.1: 単一チャネルの 224×224 画像 1 枚・10 クラスに対する AlexNet の各層のパラメータ数と 1 例あたりの積和演算数。プーリング層、活性化層、ドロップアウト層にはパラメータがない。

層	出力	パラメータ数	積和演算数
11×11 畳み込み、1 → 96、ストライド 4	96@54×54	11,712	33.9M
5×5 畳み込み、96 → 256	256@26×26	614,656	415.3M
3×3 畳み込み、256 → 384	384@12×12	885,120	127.4M
3×3 畳み込み、384 → 384	384@12×12	1,327,488	191.1M
3×3 畳み込み、384 → 256	256@12×12	884,992	127.4M
全結合、6400 →4096	4096	26,218,496	26.2M
全結合、4096 →4096	4096	16,781,312	16.8M
全結合、4096 →10	10	40,970	0.04M
合計		46,764,746	938.1M

パラメータと計算量の配分は、LeNet よりもさらに極端に偏っている。5 つの畳み込み層が占めるのは AlexNet の 4700 万個のパラメータのうちわずか 8% であるが、積和演算の 95% を実行し

ている。一方、3 つの全結合層はパラメータの 92%（32 ビット浮動小数点で約 170 MB）を消費しながら、算術演算のわずか 5% しか担っていない。画像 1 枚あたり 61,706 個のパラメータと 416,520 回の積和演算であった LeNet と比べると、AlexNet はパラメータ数を約 $760\times$、算術演算量を $2{,}250\times$ に拡大している。ImageNet（RGB 3 チャネル、1000 クラス）では、完全なモデルは約 6000 万個のパラメータに達するが、依然としてその大半が同じ密結合層に集中している。

7.1.3 Fashion-MNIST での訓練

現代の GPU でも ImageNet 上で AlexNet を訓練するには数時間かかり、さらに約 150 GB のデータセットと、GPU にデータを供給し続ける高速な入力パイプライン（input pipeline）が必要になる。そこで本節では、対話的に実験を進められるよう Fashion-MNIST 上で訓練する。Fashion-MNIST 本来の 28×28 画像は小さすぎて、11×11（ストライド 4）の第 1 層と 3 回のプーリングステージを通過できないため、224×224 へと `resize` 引数（`d2l.FashionMNIST` の指定）によってアップサンプリングする。実応用においては、画素数を 64 倍にアップサンプリングしても視覚的な情報は一切増えず計算量だけが増大するが、ここでは変更なしの AlexNet アーキテクチャをそのまま動かすためだけにこのオーバーヘッドを受け入れる。

バッチサイズ 128 のミニバッチ SGD を用いて、10 エポック（LeNet の 30 エポックの 3 分の 1）訓練する。ReLU 活性化関数のおかげでシグモイドによる減衰なしに勾配が流れるため、LeNet の 2.0 よりはるかに小さい学習率できれいに学習が進む。

7.1.3.1 学習率スケジュール

固定された学習率が訓練の初期と終盤の両方で最適になることはまれである。重みがランダムな初期段階では、過度に大きなステップは最適化を不安定にしやすく、終盤近くでは大きなステップによってパラメータが最小点の周囲で跳ね回り、そこへ落ち着かなくなる。AlexNet の原論文の著者らは、検証誤差（validation error）が頭打ちになるたびに手動で学習率を 10 で割って減衰させていた。その代わりに本書では、現代の標準となっている滑らかで人手のいらない学習率スケジュール（learning-rate schedule）を採用し、すべてのモデルを公平に比較できるよう本章の残り全体でもこれとまったく同じスケジュールを再利用する。具体的には、線形の**ウォームアップ**によって学習率を 0 からピーク値 η_0 まで最初の t_{w} ステップかけて引き上げ、続いて半コサイン減衰（half-cosine decay）により 0 へと最終ステップ T で戻す。

$$\eta_t = \frac{\eta_0}{2}\left(1 + \cos\frac{\pi(t - t_{\mathrm{w}})}{T - t_{\mathrm{w}}}\right) \quad \text{for } t \geq t_{\mathrm{w}}. \tag{7.2}$$

$t = t_{\mathrm{w}}$ ではコサイン項が 1 に等しくなり $\eta_t = \eta_0$ となる。減衰の中間地点では $\eta_0/2$ に下がり、$t = T$ で 0 に達する。緩やかにウォームアップすることで、ステップ 1 からいきなり使うには危険なほど高いピーク学習率を利用できるようになる。ここではピーク学習率 $\eta_0 = 0.03$ で AlexNet を訓練する（7.2 節の各アーキテクチャでも同様に、安定して訓練できる最大のピーク学習率を用いる）。以下の `cosine_warmup` はステップ t における乗算スケジュール係数を計算する関数である。本章全体で再利用するためこれを `d2l` に保存しておくが、学習率スケジュールについては 9.7 節で体系的に学ぶ。Figure 7.3 にこのスケジュールの形状を示す。

```
def cosine_warmup(t, num_steps, warmup_steps):
    """Learning-rate factor: linear warmup from 0, then a half-cosine to 0."""
    if t < warmup_steps:
        return t / warmup_steps
    progress = min((t - warmup_steps) / (num_steps - warmup_steps), 1)
    return 0.5 * (1 + math.cos(math.pi * progress))
```

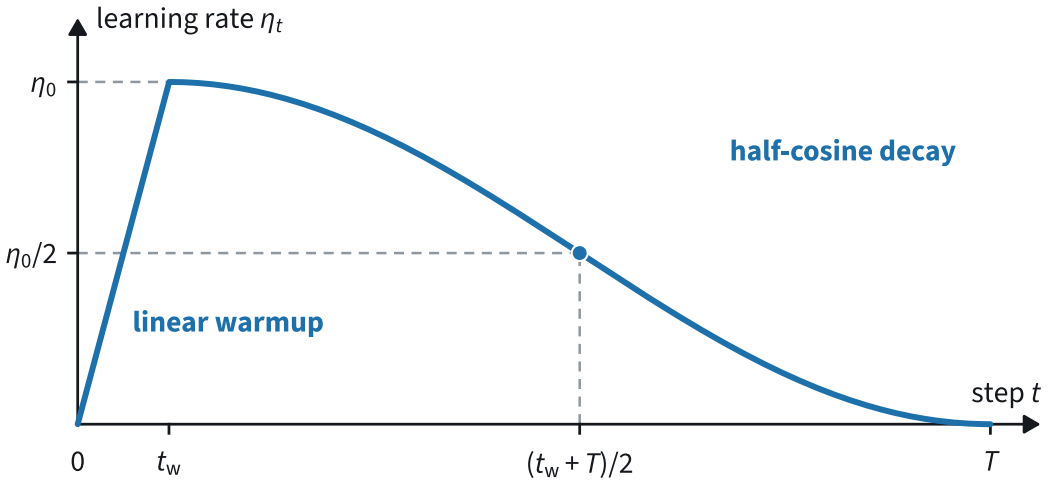


Figure 7.3: 本章全体で使用する学習率スケジュール。線形ウォームアップが学習率を 0 からピーク値 η_0 まで最初の t_w ステップ（ここでは訓練全体の 10 分の 1）かけて引き上げる。続いて Equation 7.2 の半コサイン減衰がこれを 0 へと最終ステップ T で引き下げ、減衰の中間地点で $\eta_0/2$ を通る。

AlexNet はミニバッチごとにこのスケジュールを更新する。num_steps は実行全体の総ステップ数 T を指定し、ウォームアップは訓練の最初の 10 分の 1（10 エポック中の 1 エポック）を占める。

```
@d2l.add_to_class(AlexNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

7.1.3.2 LeNet との比較

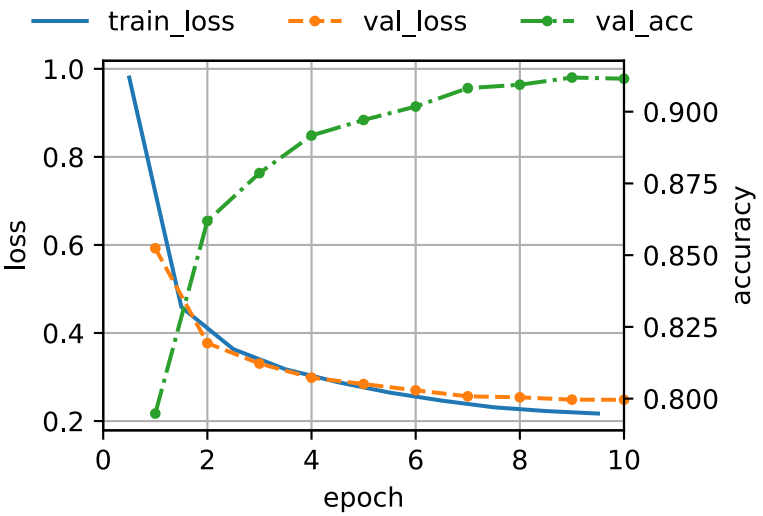
訓練ループを実行する前に、あらかじめ見通しを立てておこう。Fashion-MNIST には AlexNet の 4700 万個のパラメータを十分に活用するだけのデータセット規模も視覚的な複雑さもないため、LeNet に対する正解率（accuracy）の向上はわずかなものにとどまると予想すべきである。ここでの目的は、実装を検証し、計算プロファイルを確認し、第 1 層が何を学習するかを調べることにある。その観察を可能にするため、訓練開始前に第 1 層の 96 個の初期カーネルをスナップショットとして保存し、最適化によってそれらがどう変化するかを正確に測れるようにしておく。

```
def first_kernels(model):
    """The 96 kernels of 11 x 11 weights of the first layer, as NumPy."""
    return d2l.numpy(model.net[0].weight)[:, 0]

data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = AlexNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
model.apply_init([next(iter(data.train_dataloader()))[0][:1]],
                 model.weight_init)  # Create the weights now
```

```
K0 = first_kernels(model)
```

初期の 11 × 11 カーネルを K0 に記録したところで、AlexNet をリサイズ済みの Fashion-MNIST データセット data 上で 10 エポック訓練する。

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



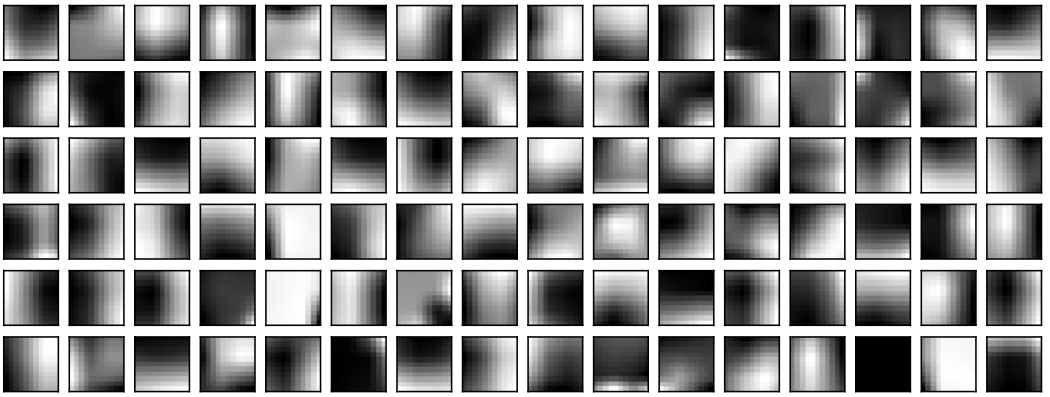
10 エポック後、AlexNet は約 91% の検証正解率に達し、訓練損失は 0.22 付近、検証損失（validation loss）は 0.25 付近となる。コサインスケジュールが終盤のエポックで学習率を 0 に向けて徐々に減衰させるため、終盤での正解率の変動は 1 パーセントポイント未満に収まり、最終エポックの値が実行全体の信頼できる要約となる。30 エポック後に約 88〜90% であった LeNet と比べると、AlexNet は 3 分の 1 のエポック数で 1〜2 パーセントポイント上回るが、その代償として画像 1 枚あたり 2,250× の算術演算を要している。しかも、そのわずかな改善でさえ、モデル容量（model capacity）の大きさと同じくらい ReLU 活性化関数、He 初期化、学習率スケジュールに負うところが大きい。

7.1.3.3 学習された第 1 層

LeNet の 5 × 5 カーネルは小さすぎて視覚的に読み取れなかったが（6.6 節）、AlexNet のサイズ 11 × 11 の 96 個のカーネルは、ImageNet 上の Figure 7.1 で見たように、小さな画像として直接解釈できる十分な大きさがある。このモデルが Fashion-MNIST 上で何を学習したかを見るため、訓練後の第 1 層のカーネルからランダムな初期重み K0 を引き、各重みの差分をそれぞれのグレースケール範囲でプロットする。

```
change = first_kernels(model) - K0
d2l.show_images(change, 6, 16, scale=0.6, cmap='gray')
print(f'mean size of the change: {np.abs(change).mean() / np.abs(K0).mean():.3f}'
      ' of the mean size of the initial weights')
```

```
mean size of the change: 0.044 of the mean size of the initial weights
```



10 エポックの訓練で第 1 層の重みが変化する量は初期の大きさの約 4% にすぎないため、訓練後のカーネルをそのままプロットしても、大部分は初期のランダムノイズのように見えてしまう。しかし、**重みの更新量**そのものには明確な構造が現れている。滑らかな輝度勾配や幅広の方向性バーであり、これらは Figure 7.1 で見た鮮明な ImageNet のエッジフィルタに対応する低周波版である。欠けている 2 つの要素（細かな高周波エッジと色ブロブ）はいずれも入力データから直接説明できる。ここでの画像はグレースケールであり、また縦横 8 倍のアップサンプリングによって約 8 ピクセルより細かい本来のディテールが存在しないため、11×11 カーネルからは緩やかな輝度勾配として捉えられるのである。要するに、第 1 層はそのデータセットの空間統計量に忠実に適応している。

最後に、Table 7.1 で見たいくつかの数値を振り返ろう。AlexNet のパラメータの 92% は巨大な全結合ヘッドに存在し、算術演算にはほとんど寄与しない一方で暗記を抑えるために強いドロップアウトを必要としていた。次節では、再利用可能なブロックから深いアーキテクチャを体系的に構築する方法、そしてあの巨大な密結合ヘッドがそもそも必要なのかどうかを問う。

7.1.4 まとめ

エンドツーエンドで学習される視覚表現が人手で設計された特徴パイプラインを凌駕したのは、2012 年に 3 つの要因が揃ったときであった。大規模なラベル付きデータセット（ImageNet）、数千万個のパラメータを数日で訓練できる超並列 GPU、そして近代的な訓練の構成要素（ReLU 活性化関数、ドロップアウト、データ拡張）である。これらの道具を用いて LeNet の基本テンプレートを拡大した AlexNet の出場システムは、ImageNet の top-5 誤差を 26.2% から 15.3% へと大幅に引き下げた。層ごとの内訳を集計すると際立った非対称性が明らかになり、畳み込みの幹線部が算術演算の 95% を実行する一方、密結合ヘッドがパラメータの 92% を保持している。これほど膨大なモデル容量を使い切れない Fashion-MNIST 上では、AlexNet は画像あたり約 $2{,}250\times$ の算術演算を費やしつつ、3 分の 1 のエポック数で LeNet を 1〜2 パーセントポイント上回る。

7.1.5 演習問題

1. **メモリと計算量の内訳**。32 ビット浮動小数点において Table 7.1 を `AlexNet` が必要とするメモリ量へと拡張せよ。
    1. 224×224 画像 1 枚の順伝播が保存する活性化値の個数を、すべての畳み込み層、プーリング層、全結合層の出力を数えて層ごとに計算せよ。活性化メモリはどこに集中しており、パラメータのメモリはどこに集中しているか。
    2. 訓練中には、パラメータ、その勾配、およびミニバッチ全体の保存された活性化値がメモリに収まらなければならない。どのバッチサイズで、活性化メモリがパラメータ

と勾配のメモリを上回るか。

3. Table 7.1 のパラメータ数の計算を ImageNet 版（3 入力チャネル、1000 クラス、227×227 入力により最終畳み込みステージが 6×6（5×5 ではなく）になる設定）に対して繰り返し、合計を Krizhevsky et al. (2012) で報告されている 6000 万個のパラメータと比較せよ。

2. **初期層の受容野**。カーネルサイズ k_i とストライド s_i を持つ層の積み重ねにおいて、最上位のユニットが持つ受容野の一辺は $r = 1 + \sum_i (k_i - 1) \prod_{j<i} s_j$ で与えられる（6.3 節）。各最大プーリング層もそれぞれのカーネルサイズとストライドを持つ層として数え、`AlexNet` の 5 つの畳み込み層それぞれの直後における 1 ユニットの受容野を計算せよ。各受容野は 224×224 の入力の一辺に対してどの割合を占めるか。また、いずれかの畳み込み層のユニットは画像全体を見ているか。

3. **演算密度**。●層の**演算密度**とは、メモリから読み出すパラメータ 1 バイトあたりにその層が実行する積和演算の回数である。Table 7.1 を用いて次を答えよ。

   1. 画像 1 枚のバッチと画像 128 枚のバッチのそれぞれについて、`AlexNet` のすべての層の演算密度を計算せよ（各パラメータは 1 バッチにつき 1 回読み出されると仮定する）。
   2. メモリから 1 バイトを読み出す時間に 10 回の積和演算を実行するアクセラレータを考え、各層の処理時間は計算時間とパラメータ読み出し時間の最大値で決まるとする。画像 1 枚の推論（$B = 1$）をより高速化するのは、計算スループットを 2 倍にすることか、それともメモリ帯域幅を 2 倍にすることか。また、$B = 128$ での訓練をより高速化するのはどちらか。

4. [code] **バッチサイズ、スループット、メモリ**。●バッチサイズのみを変えながら、本節と同様に `AlexNet` を訓練せよ。

   1. 問題 1 を用いて、手元の GPU のメモリに収まる最大の訓練バッチサイズを予測せよ。
   2. バッチサイズ 32 から始めて 2 倍ずつ増やし、メモリ不足にならずに訓練できる最大のバッチサイズを見つけ、予測との差が生じる主な要因を説明せよ。
   3. その上限までの少なくとも 4 通りのバッチサイズにわたって訓練スループット（枚/秒）を測定せよ。どのバッチサイズを超えるとスループットが頭打ちになるか。また、それはなぜか。

7.2 ブロック、ボトルネック、ブランチ：VGG、NiN、GoogLeNet

AlexNet（7.1 節）は 1 層ずつ手作業で設計されていた。ここには 11×11 カーネル、あそこには 384 チャネルを置き、1 番目、2 番目、5 番目の畳み込みの後に最大プーリングを挿入するといった具合である。8 層のモデルであれば全層を手動で調整することも可能だが、20 層以上へとスケールさせるには体系的な設計規則が求められる。同様に注目すべき点として、AlexNet は 4700 万個のパラメータのうち 92% を全結合分類ヘッドに集中させているが、この部分は演算量のわずか 5% しか担っておらず、訓練セットの丸暗記を防ぐために強いドロップアウトに頼っている。2013〜2014 年にかけて、3 つの画期的なアーキテクチャがこれらのボトルネックに順次取り組んだ。今日の実務でこれらのモデルがそのまま使われることはほとんどないが、そこで導入された構造上のアイデアは、現代のほぼすべての畳み込みネットワークに受け継がれている。

第 1 に、**VGG** は設計の単位を個々の層から、小さな 3×3 畳み込みとそれに続くプーリングからなる反復可能な**ブロック**へと移行させ、アーキテクチャ全体をブロック仕様の簡潔なリストとして表現できるようにした (Simonyan and Zisserman 2015)。第 2 に、**Network in Network**（NiN）は肥大化した全結合ヘッドを完全に排除し、1×1 畳み込みによって各ピクセルでチャネルを混合したうえで、グローバル平均プーリング（global average pooling）により最終の特徴マップを

直接クラススコアへと縮約した (Lin et al. 2013)。第 3 に、**GoogLeNet** は単一のカーネルサイズに固定するのを避け、各ブロック内で複数のフィルタサイズを並列に実行しつつ、チャネルを削減する 1×1 ボトルネックによって大きな畳み込みの計算コストを抑えた (Szegedy et al. 2015)。以下では、これら 3 つのアーキテクチャをすべて構築し、AlexNet と同様にパラメータ数と積和演算（multiply-add）の回数を数え、同一のコサインスケジュールのもとで Fashion-MNIST 上で訓練して横並びで比較する。

7.2.1 VGG：設計単位としてのブロック

7.2.1.1 3×3 畳み込みの積み重ね

もしすべての畳み込み層の後にストライド 2 のプーリングを置いたとすると、幅 d の入力は高々 $\lfloor \log_2 d \rfloor$ 層で 1×1 に潰れてしまう（$224 \to 112 \to 56 \to 28 \to 14 \to 7 \to 3 \to 1$）。ネットワークをより深くするには、ダウンサンプリングの前に、各空間解像度（spatial resolution）で形状を保つストライド 1 の畳み込みを複数層積み重ねる必要がある。

VGG は、左右および上下の隣接ピクセルを捉える最小の対称フィルタである 3×3 カーネルに標準化した (Simonyan and Zisserman 2015)。2 つ積み重ねた 3×3 層は、同じ 5×5 の受容野（receptive field）を 1 つの 5×5 層と同様にカバーしつつ、重みの数を $2 \cdot 9c^2 = 18c^2$ 個に抑えられる（単一層では $25c^2$ 個）。また、3 つ積み重ねた 3×3 層は、7×7 のウィンドウを $27c^2$ 個の重みでカバーし（単一層では $49c^2$ 個）、同時に ReLU の非線形性を 3 つ挿入できる（単一層では 1 つ）。**VGG ブロック**は、パディング付きの 3×3 畳み込みと ReLU を `num_convs` 個積み重ね、その後に 2×2（ストライド 2）の最大プーリングを続ける構造である。

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

7.2.1.2 VGG ネットワーク

完全な VGG ネットワークは、これらのブロックの列を連結し、その末尾に AlexNet の密な分類ヘッド（ドロップアウト付きの 4096 ユニットの隠れ層 2 つと、それに続く出力層）を接続する。Figure 7.4 に VGG-11 と AlexNet の対比を示す。5 段階のプーリングステージによって 224×224 の入力は半分ずつ縮小されて 7×7 になり、一方でチャネル幅はステージごとに倍増して 512 に達する。

すべての VGG ブロックは 2 つの整数（畳み込み層の数と出力チャネル数）で完全に指定できるため、畳み込みトランク全体は `(num_convs, out_channels)` というタプルのリスト（`arch`）として渡すことができる。VGG は単一の固定されたアーキテクチャを定義するのではなく、**パラメータ化されたモデル族**（parameterized family）を定義する。すなわち、VGG-11、VGG-16、VGG-19 はまったく同じクラスを共有し、`arch` だけが異なる。ネットワークのトポロジを手書きのコードではなくモジュール化されたデータとして扱うことこそ、VGG が残した最大の遺産である。7.1 節と同様に、本節のすべてのネットワークは重みを He 正規初期化し、バイアスを 0 に初期化する。

AlexNet と同様に、最初のミニバッチによって遅延初期化（lazy）層の形状が確定した後、`Trainer` がすべての層に対して `weight_init` を呼び出す。

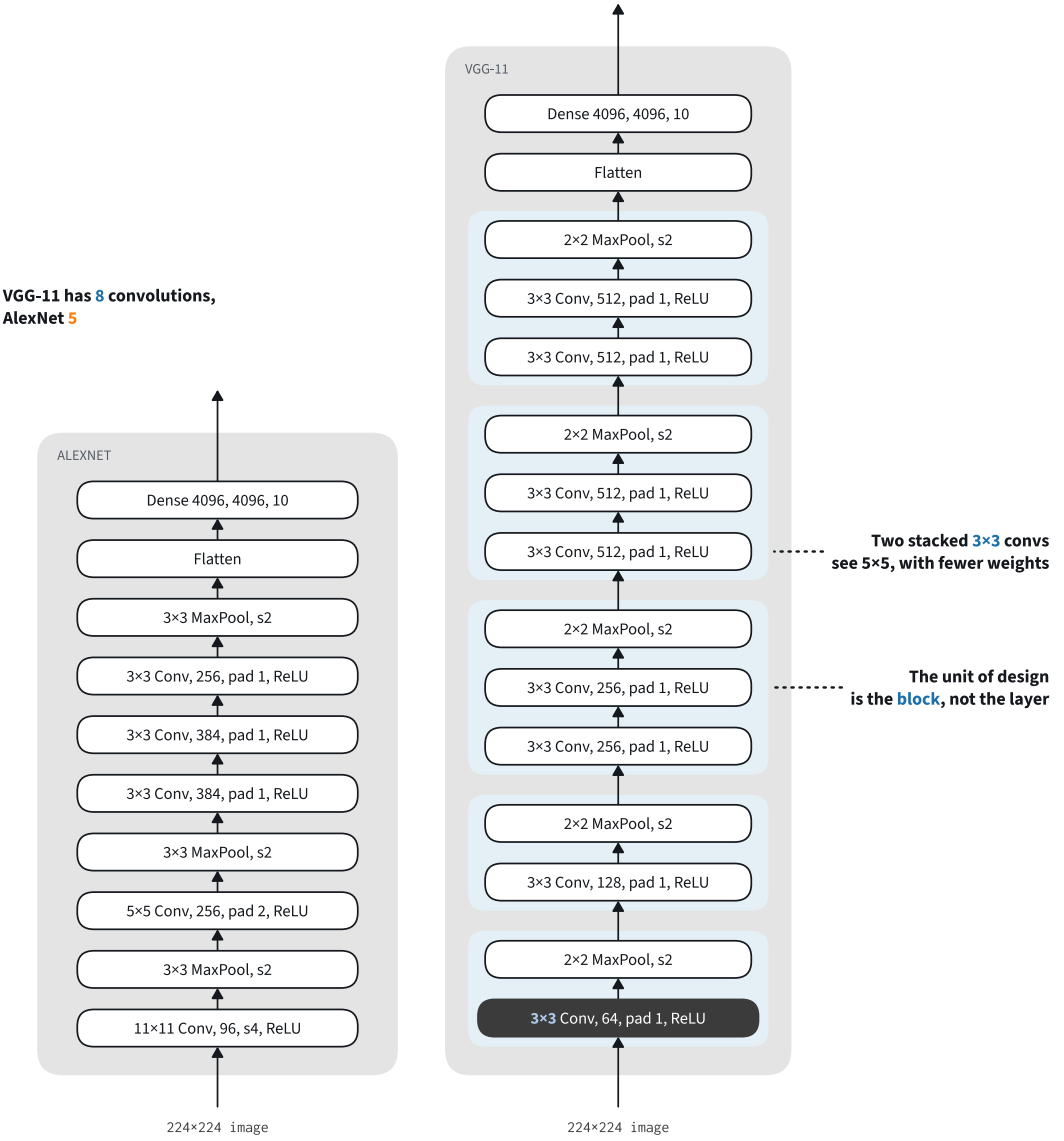


Figure 7.4: 単一チャネルの 224×224 画像と 10 クラスに対する AlexNet（左）と VGG-11（右）。入力は下側にある。AlexNet の層は 1 つずつ個別に選ばれているのに対し、VGG は複数の 3×3 畳み込みと最大プーリングからなる 1 種類のブロックを、チャネル数を増やしながら繰り返す。

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = [vgg_block(num_convs, out_channels)
                     for (num_convs, out_channels) in arch]
        self.net = nn.Sequential(*conv_blks, *self.head())

    def head(self):  # AlexNet's fully connected head
        return [nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(self.num_classes)]

    def weight_init(self, module):  # He-normal weights, zero biases
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

元論文で最小の変種である VGG-11 は、(1, 1, 2, 2, 2) 個の畳み込み層と (64, 128, 256, 512, 512) チャネルを持つ 5 つのブロックを積み重ねており、8 つの畳み込み層と 3 つの全結合層を合わせて計 11 個のパラメータ付き層からなる。単一の 224×224 画像に対する中間テンソルの形状を確認してみよう。

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential            (1, 64, 112, 112)
Sequential            (1, 128, 56, 56)
Sequential            (1, 256, 28, 28)
Sequential            (1, 512, 14, 14)
Sequential            (1, 512, 7, 7)
Flatten               (1, 25088)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 10)
```

最終の畳み込みブロックは $512 \times 7 \times 7$ のテンソルを出力し、これが 25,088 個の特徴量へと平坦化されて密なヘッドに入力される。AlexNet と同様に各層を集計すると、VGG-11 は 1 億 2880 万個のパラメータを持ち、画像 1 枚あたり 75 億回の積和演算（AlexNet の計算量の 8 倍）を実行することがわかる。ここでもやはり、パラメータと計算量はまったく別の場所に偏在している。密なヘッドが全重みの 93% を占めており、最初の $25{,}088 \times 4096$ の行列だけで 1 億 300 万個のパラメータに達する。

7.2.1.3 訓練

Fashion-MNIST の実演に AlexNet の 8 倍もの計算量を費やすのは不必要に重いため、チャネル数を 16、32、64、128、128 とした 1/4 幅の VGG-11 を訓練する。このモデルの画像 1 枚あたり

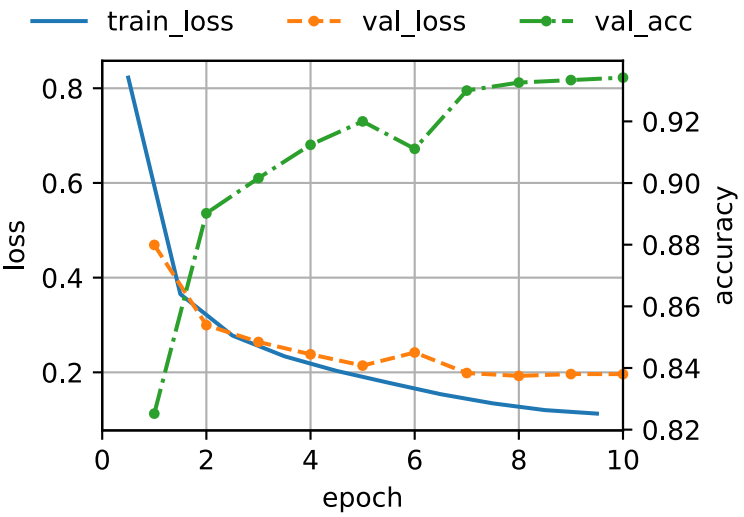
の積和演算は 5 億 1200 万回であり、AlexNet の 9 億 3800 万回の約半分に収まる。その他の訓練設定はすべて 7.1 節と同一とし、224 × 224 にアップサンプリングした画像、バッチサイズ 128、および 7.1 節のウォームアップ付きコサインスケジュールによる 10 エポックの SGD を用い、AlexNet の `configure_optimizers` を再利用する。

本節のネットワーク間で変更する唯一のハイパーパラメータは、ピーク学習率である。ピーク値を 0.1 にすると、以下で扱う正規化なしのアーキテクチャのいくつかは不安定になる寸前の状態に陥り、ときおり発散したりランダム推測の水準で停滞したりするため、それらのモデルは代わりに 0.03 で訓練する。1/4 幅の VGG は 0.1 でも安定して訓練でき、0.03 に比べて 1.5〜2 パーセントポイント高い精度が得られるため、0.1 で実行する。

```
@d2l.add_to_class(VGG)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

コサインウォームアップスケジュールを設定したうえで、1/4 幅の `VGG` を 224 × 224 の Fashion-MNIST 上で 10 エポック訓練し、検証正解率の軌跡を `val_acc['VGG']` に保存する。

```
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.1,
            num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc = {'VGG': trainer.history['val_acc']}  # for the comparison below
```



この 1/4 幅の VGG は約 93% の検証正解率に達し、約半分の演算量で AlexNet を 2 パーセントポイント上回る。もっとも公平を期すなら、この差の多くは学習率の高さによるものである。後ほど確認するように、AlexNet と同じピーク値 0.03 では、同じ VGG でも 92% 弱にとどまる。しかし、畳み込みトランクの幅を狭めても解決しなかった問題に注目してほしい。畳み込み層のパラメー

タはわずか 60 万個まで縮小したにもかかわらず、密なヘッドは依然として $128 \times 7 \times 7 = 6272$ 個の活性化値を 4096 個の隠れユニットへと写像しており、モデル全体の 4310 万パラメータのうち 4250 万個（実に 99%）を占めている。ブロックによってネットワークのボディは規則的に整理されたが、ヘッドは依然として肥大化したままである。

7.2.2 NiN：1×1 畳み込みとグローバル平均プーリング

密な分類ヘッドが数千万個ものパラメータを消費するのは、すべてのチャネルのすべての空間座標と、すべての隠れユニットを結ぶ個別の重みを割り当てるからである。しかも、その膨大な容量は、最終の特徴マップをクラスのロジットへと要約するというただ 1 つの役割のためだけに使われている。**Network in Network**（NiN）は、この密なヘッドを完全に廃止しつつ、同時に各畳み込みブロック内の非線形性を高められないかと問う。標準的な畳み込みの後に ReLU を続けただけでは、各局所パッチにしきい値付きの線形フィルタを適用しているにすぎなかった。カーネルを大きくせずに同じ空間ウィンドウ上でより豊かな非線形関数を計算するには、ピクセルごとにチャネルを混合する層、まさに 1×1 畳み込み（6.4.3 節）を積み重ねればよい。

7.2.2.1 NiN ブロック

1×1 畳み込みはすべてのピクセルに対して同一の全結合層を独立に適用するため、空間畳み込みの後に 2 つの 1×1 畳み込み（それぞれ ReLU を伴う）を続けると、すべての空間位置に隠れ層 1 層の小さな MLP を配置することになる。これが **Network in Network** という名前の由来である。Figure 7.5 に VGG の密なヘッドと NiN の末尾の対比を示す。

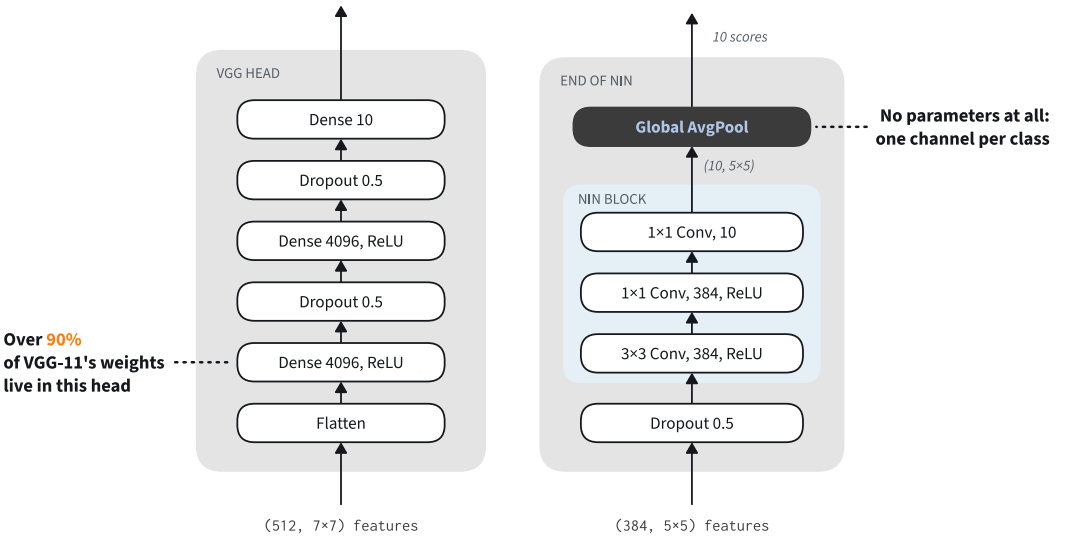


Figure 7.5: いずれも 10 クラス分類に対する VGG のヘッド（左）と NiN の末尾（右）。入力は下側にある。VGG は最終の特徴マップを平坦化して全結合層に入力する。NiN の最終ブロックは、ドロップアウトの後に 3×3 畳み込みと 2 つの 1×1 畳み込み（最後の畳み込みの後には ReLU を置かない）を適用してクラスごとに 1 チャネルを出力し、グローバル平均プーリングによって各チャネルを 1 つのスコアに変換する。

```
def nin_block(out_channels, kernel_size, strides, padding, num_classes=None):
    layers = [
```

```
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    if num_classes is None:
        layers += [nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    else:  # Last block: one map of scores per class, no ReLU
        layers.append(nn.LazyConv2d(num_classes, kernel_size=1))
    return nn.Sequential(*layers)
```

7.2.2.2 グローバル平均プーリング

バックボーンについては、NiN は AlexNet の序盤のカーネルサイズとチャネル数を踏襲しており、幅 96、256、384 の 11×11、5×5、3×3 畳み込みから始まるブロックを並べ、それぞれの後にストライド 2 の 3×3 最大プーリングを置く。しかし、ネットワークの最上部において NiN は AlexNet や VGG と決定的に袂を分かつ。ドロップアウトの後、4 番目の 384 チャネル NiN ブロックは出力チャネル数がクラス数（Fashion-MNIST では 10）に等しい 1×1 畳み込みで終わり、続いて**グローバル平均プーリング**がそれら 10 枚の特徴マップをそれぞれ全空間座標にわたって平均し、10 個のクラスロジットを直接生成する。プーリングヘッドにはパラメータが 1 つもないため、正解クラスに対応するチャネルで高い空間平均応答を出すよう、最終ブロック自体が学習しなければならない。

ここで、実装上の重要な細部に注目してほしい。`num_classes` が `nin_block` に渡されたとき、最後の 1×1 畳み込みの出力は生のクラスロジットであるため、ReLU 活性化関数を省いている。もしロジットに ReLU を適用すると、あるクラスが永久に死んでしまう危険がある。あるクラスチャネルのすべての空間位置がデータセット全体にわたって負になった場合、そのロジットと勾配は恒等的に 0 となって消失し、SGD がそのクラスを二度と復活させられなくなるからである。

グローバル平均プーリングには、パラメータの削減に加えて概念上の第 2 の利点がある。畳み込みは平行移動同変（translation-equivariant）であり、全空間座標にわたる平均は置換に対して不変であるため、得られるクラススコアは（境界処理やストライドの影響を除けば）物体が画像内の**どこに**現れるかに依存しない。対照的に、平坦化された密なヘッドは空間セルごとに異なる重みを割り当てるため、あらゆる物体をあらゆる位置で学習し直さなければならない。

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(384, kernel_size=3, strides=1, padding=1,
                      num_classes=num_classes),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten())

    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

単一の 224×224 入力に対するテンソルの形状を追ってみよう。

```
NiN().layer_summary((1, 1, 224, 224))
```

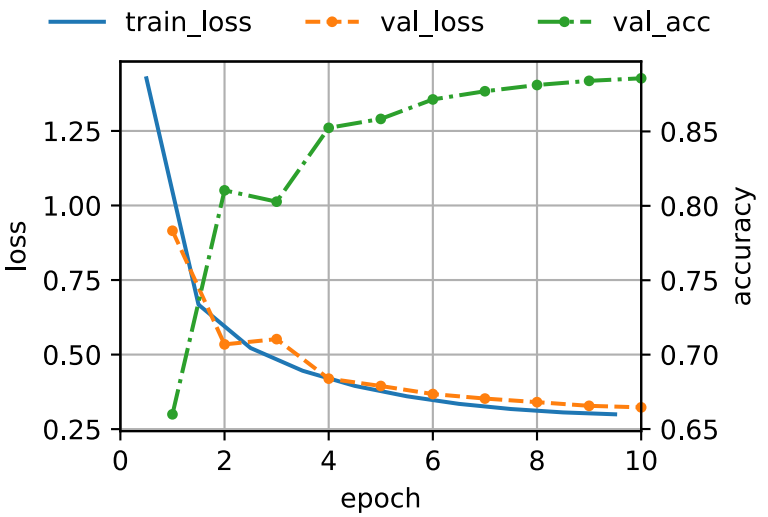
```
Sequential              (1, 96, 54, 54)
MaxPool2d               (1, 96, 26, 26)
Sequential              (1, 256, 26, 26)
MaxPool2d               (1, 256, 12, 12)
Sequential              (1, 384, 12, 12)
MaxPool2d               (1, 384, 5, 5)
Dropout                 (1, 384, 5, 5)
Sequential              (1, 10, 5, 5)
AdaptiveAvgPool2d       (1, 10, 1, 1)
Flatten                 (1, 10)
```

AlexNet と同様に、空間次元は 224 から 54、26、12、5 へと段階的に縮小し、その後に最終の NiN ブロックが 10 枚の 5×5 スコアマップを出力して、グローバル平均プーリングがそれらを 10 個のロジットへと縮約する。密なヘッドを取り除くことで、パラメータ数はわずか 340 万個（AlexNet のサイズの 7% にすぎない）まで激減する一方、追加された 1×1 畳み込みによって計算が畳み込みボディへと移るため、画像 1 枚あたりの総演算量は積和演算 7 億 9800 万回と、AlexNet の 9 億 3800 万回と同程度にとどまる。

7.2.2.3 NiN の訓練

VGG と同じオプティマイザスケジュールで NiN を訓練する。ただし、0.1 ではときおり発散するため、ピーク学習率には 0.03 を用いる。

```
NiN.configure_optimizers = VGG.configure_optimizers  # AlexNet's schedule
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = NiN(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['NiN'] = trainer.history['val_acc']
```



10 エポック後、NiN の検証正解率は約 89% に達するが、これは AlexNet より約 2 ポイント、1/4

幅の VGG より 4 ポイント低い。訓練損失は 0.30 付近で終了し（AlexNet では約 0.22）、検証損失もそれにほぼ追随していることから、10 エポックでは過学習を起こさずわずかに学習不足の状態にあることがわかる。単独のアーキテクチャとして見れば、NiN の性能は演算コストの割に控えめである。その永続的な影響力は、チャネルを混合する 1×1 畳み込みとグローバル平均プーリングヘッドという、再利用可能な 2 つの基本要素を広めた点にある。

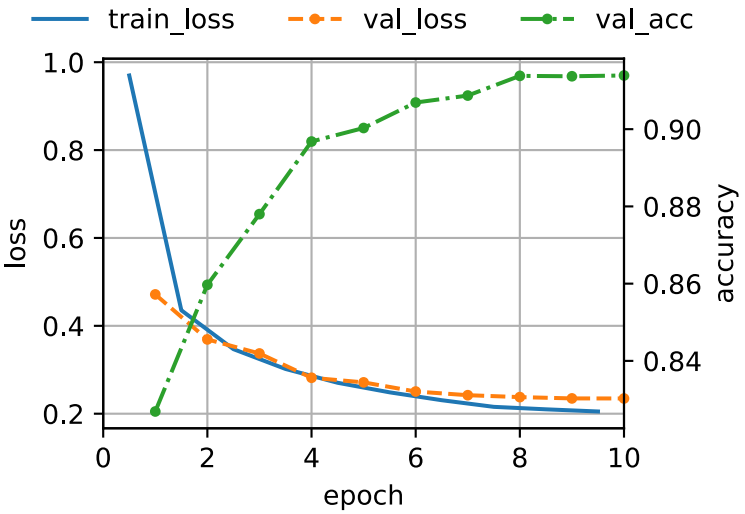
7.2.2.4 プーリングヘッドを持つ VGG

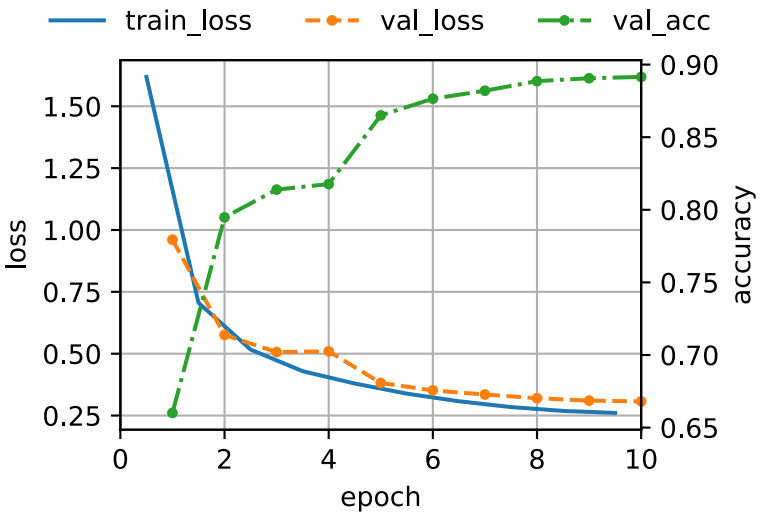
NiN は畳み込みボディと分類ヘッドの両方を同時に変更しているため、そのスコアだけでは密なヘッドが実際にどれほど寄与していたのかを切り分けられない。ヘッドの影響だけを純粋に取り出すため、1/4 幅の VGG ボディをまったく同じまま保ち、`VGG.head` のみをオーバーライドする。`PooledVGG` では、グローバル平均プーリングが最終ブロックの 128 個の出力チャネルを要約し、単一の線形層がそれら 128 個の特徴量を 10 クラスへ直接写像する。これにより、ネットワーク全体の 4310 万パラメータのうち 4250 万個（および同数の積和演算）が削減される。

```
class PooledVGG(VGG):
    def head(self):  # Global average pooling and one fully connected layer
        return [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                nn.LazyLinear(self.num_classes)]
```

`PooledVGG` も `NiN` と同様に 0.1 では発散や崩壊を起こすことがあるため、`VGG` と `PooledVGG` の両方を揃えたピーク学習率 `lr=0.03` で訓練し、最終的な検証正解率を `heads` に保存する。

```
heads = {}  # Both heads at peak 0.03, the rate the pooled head tolerates
for net in (VGG, PooledVGG):
    model = net(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)),
                lr=0.03, num_steps=10 * len(data.train_dataloader()))
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data)
    heads[net.__name__] = round(trainer.history['val_acc'][-1], 3)
val_acc['PooledVGG'] = trainer.history['val_acc']
heads
```






```
{'VGG': 0.914, 'PooledVGG': 0.891}
```

PooledVGG の検証正解率は 89〜90% となり、密なヘッドを持つ VGG（同じピーク値 0.03 で 91〜92%）を約 2 パーセントポイント下回り、訓練損失も高くなる（約 0.25 対約 0.20）。したがって、Fashion-MNIST 上で 10 エポック訓練したこの幅の狭い畳み込みボディにおいては、密なヘッドの追加容量はモデル全体のパラメータの 99% と引き換えに、正解率で約 2 ポイント分の働きをしていることになる。

7.2.3 GoogLeNet：マルチブランチブロック

ここまでのアーキテクチャでは、設計者は各層で単一のカーネルサイズを選ばなければならなかった。1×1 は局所的にチャネルを混合し、3×3 は細かいテクスチャを捉え、5×5 はより広い近傍をカバーするが、実際の写真に写る物体はあらゆるスケールで現れる。GoogLeNet はこの選択そのものを回避する。その **Inception ブロック**は、同じ入力を複数のカーネルサイズで並列に処理し、それらの出力をチャネル軸に沿って連結するのである。複数のフィルタサイズをそのまま同時に走らせればパラメータも計算量も爆発してしまうため、GoogLeNet は高コストな各空間フィルタの手前に、次元削減（dimensionality reduction）のボトルネックとして NiN の 1×1 畳み込みを取り入れている。

7.2.3.1 Inception ブロック

Figure 7.6 に示すように、各 Inception ブロックは入力を 4 つの並列ブランチに分岐させる。(1) 1×1 畳み込み、(2) チャネル削減の 1×1 畳み込みとそれに続く 3×3 畳み込み、(3) チャネル削減の 1×1 畳み込みとそれに続く 5×5 畳み込み、(4) ストライド 1 の 3×3 最大プーリング層とそれに続く 1×1 畳み込みである。4 つの出力をチャネル次元に沿ってきれいに連結できるよう、すべてのブランチで空間的な高さと幅を保つように入力をパディングする。ブランチごとのチャネル数は、空間スケール間で容量を配分する調整つまみの役割を果たす（このブロック名は、映画 **Inception** と深層学習のミーム「もっと深くする必要がある（we need to go deeper）」に掛けた遊び心のある命名である）。

```
class Inception(nn.Module):
    # c1--c4 are the numbers of output channels of the four branches
```

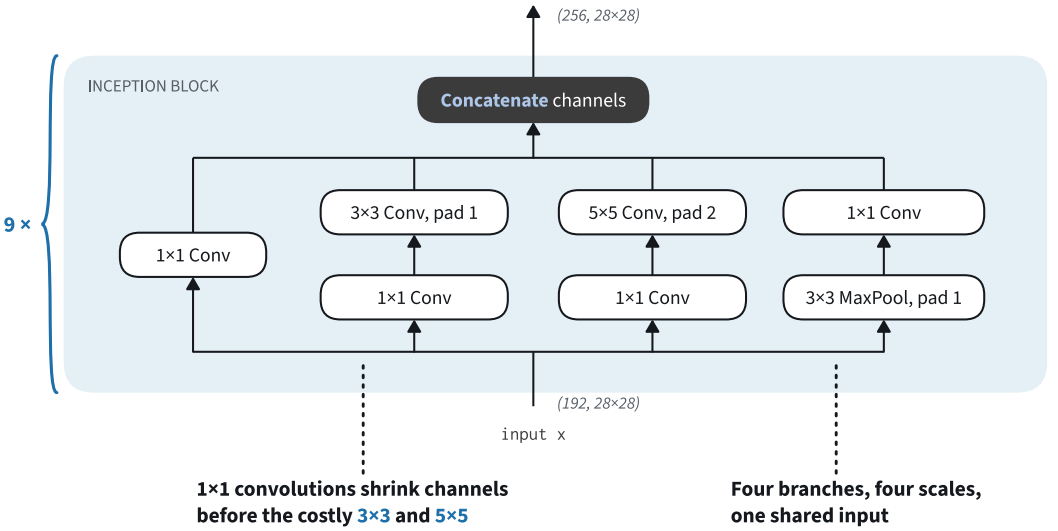


Figure 7.6: Inception ブロック（入力は下側）。4 つの並列ブランチが同じ入力を異なるカーネルサイズで読み込み、1×1 畳み込みが 3×3 および 5×5 畳み込みの手前でチャネル数を削減し、各ブランチの出力がチャネル次元に沿って連結される。図中の形状は、224×224 入力に対する GoogLeNet の最初の Inception ブロックのものである。

```
    def __init__(self, c1, c2, c3, c4):
        super().__init__()
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

GoogLeNet の最初の Inception ブロックは、1×1 によるチャネル削減の効果を鮮やかに示している。224×224 の入力に対し、このブロックは解像度 28×28 で 192 チャネルを受け取る（Figure 7.6）。ブランチ幅を $c_1 = 64$、$c_2 = (96, 128)$、$c_3 = (16, 32)$、$c_4 = 32$ とすると、$64 + 128 + 32 + 32 = 256$ チャネルを同じ解像度 28×28 で出力する。もし 192 チャネルから 32 チャネルへ 5×5 畳み込みを直接適用すれば、$25 \cdot 192 \cdot 32 = 153{,}600$ 個の重みが必要になる。しかし、先に 192 チャネルを 16 チャネルへ射影しておけば、重みは $192 \cdot 16 = 3{,}072$ 個（1×1 による削減分）と $25 \cdot 16 \cdot 32 = 12{,}800$ 個（5×5 フィルタ分）の合計 15,872 個で済み、パラメータ数と積和演算の両方で約 10 分の 1 に削減される。高コストな空間畳み込みの手前に幅の狭い 1×1 の射影（projection）を置くこの構造は**ボトルネック**と呼ばれ、深層視覚モデルで最も広く採用される基本要素の 1 つとなり、7.4.5.1 節や 7.5 節でも再び登場する。

7.2.3.2 ステム、ボディ、ヘッド

マクロな視点で見ると、GoogLeNet は現代のほぼすべての視覚バックボーンに共通する 3 部構成の**ステム・ボディ・ヘッド**という設計図を確立した。**ステム**は標準的な畳み込みとプーリング層の短い積み重ね（ここでは 3 つの畳み込みと 2 つの最大プーリング）を通じて生のピクセルを取り込み、空間解像度を素早く 8 分の 1 に縮小しつつ 192 チャネルへと拡張する。**ボディ**は 2 個、5 個、2 個の 3 ステージに配置された計 9 個の Inception ブロックを連結し、ステージ間をストライド 2 の最大プーリングで区切る。最後に**ヘッド**は NiN のコンパクトな設計を採用し、グローバル平均プーリングの後に単一の線形分類層を置く。

9 個の Inception ブロックにおける各ブランチの幅は、単純な数式から導かれたものではなく、元論文で手作業により調整されたものである。VGG と同様に、この構成全体を構造化データ、すなわち各 Inception ブロックにつき 1 つの設定タプルを収めたステージのタプルとして記述する。

```
googlenet_arch = (
    ((64, (96, 128), (16, 32), 32), (128, (128, 192), (32, 96), 64)),
    ((192, (96, 208), (16, 48), 64), (160, (112, 224), (24, 64), 64),
     (128, (128, 256), (24, 64), 64), (112, (144, 288), (32, 64), 64),
     (256, (160, 320), (32, 128), 128)),
    ((256, (160, 320), (32, 128), 128), (384, (192, 384), (48, 128), 128)))
```

`arch` を `GoogLeNet` に渡すことで、ステム、最大プーリングで区切られた 3 つの Inception ステージ、およびグローバルプーリングヘッドが組み立てられる。

```
class GoogLeNet(d2l.Classifier):
    def __init__(self, arch=googlenet_arch, lr=0.1, num_steps=4690,
                 num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        pool = lambda: nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        stem = nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3), nn.ReLU(),
            pool(),
            nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
            nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(), pool())
        body = [nn.Sequential(*[Inception(*c) for c in group])
                for group in arch]
        head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                             nn.LazyLinear(num_classes))
        self.net = nn.Sequential(stem, body[0], pool(), body[1], pool(),
                                 body[2], head)

    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

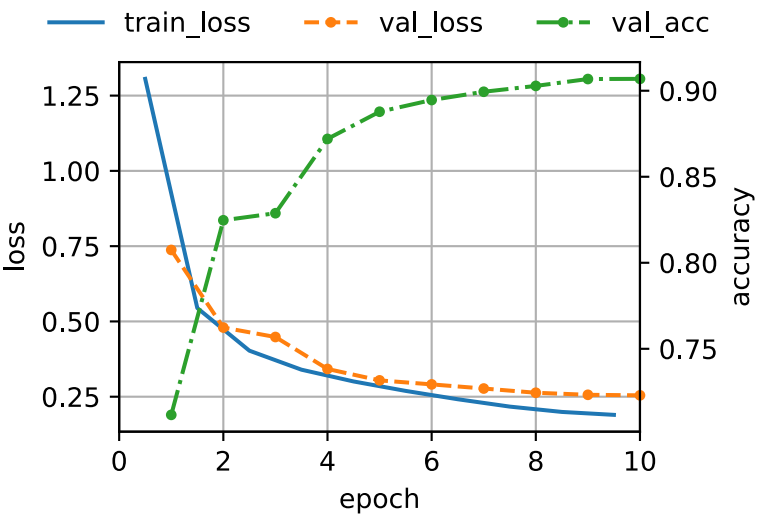
ステム内の 3 回（最初の 7×7 畳み込みと 2 回の最大プーリング）と Inception ステージ間の 2 回を合わせた計 5 回のストライド 2 のダウンサンプリング演算により、GoogLeNet は空間解像度を 32 分の 1 に縮小する。Fashion-MNIST の訓練を軽快に保つため、入力を 96×96 にリサイズすると、3 つの Inception ステージはそれぞれ 12×12、6×6、3×3 で動作し、チャネル数はステム直後の 192 から 480、832、1024 へと増加する。

```
GoogLeNet().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 192, 12, 12)
Sequential              (1, 480, 12, 12)
MaxPool2d               (1, 480, 6, 6)
Sequential              (1, 832, 6, 6)
MaxPool2d               (1, 832, 3, 3)
Sequential              (1, 1024, 3, 3)
Sequential              (1, 10)
```

GoogLeNet は本節で群を抜いて深いアーキテクチャであり、最長経路に沿って 22 個のパラメータ付き層が連なる（ステムに 3 層、9 個の Inception ブロックに各 2 層、そして線形ヘッド）。NiN と同様に、0.1 ではときおりランダム推測の水準で停滞するため、ピーク学習率 0.03 で訓練する。以下の比較にあたり、実験上の細部を 1 点念頭に置いてほしい。入力を 96×96 にして訓練するため（224×224 ではなく）、GoogLeNet の 600 万個のパラメータが画像 1 枚あたりに要する積和演算はわずか 2 億 7600 万回にとどまる。

```
GoogLeNet.configure_optimizers = VGG.configure_optimizers
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = GoogLeNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['GoogLeNet'] = trainer.history['val_acc']
```



10 エポック後、GoogLeNet は検証正解率 91% 付近に達して同じピーク学習率の AlexNet に並び、訓練損失はわずかに低く（0.20 弱）、検証損失は同程度（約 0.25）となる。

7.2.4 ネットワークの比較

コサインスケジュールは学習率を滑らかに 0 まで減衰させるため、最終エポックの検証正解率は 4 つのモデルすべてで安定しており、直接比較できる。

```
{name: round(acc[-1], 3) for name, acc in val_acc.items()}  # After 10 epochs
```

```
{'VGG': 0.934, 'NiN': 0.886, 'PooledVGG': 0.891, 'GoogLeNet': 0.907}
```

Table 7.2 に、LeNet、AlexNet、および本節の 4 つのアーキテクチャをまとめ、入力解像度、パラメータ数、画像 1 枚あたりの積和演算数、ピーク学習率、Fashion-MNIST の検証正解率を一覧で示す。

Table 7.2: 各ネットワーク（6.6 節、7.1 節、および本節）のパラメータ数、積和演算数、および Fashion-MNIST 検証正解率。積和演算数は訓練時の入力サイズにおける画像 1 枚あたりの回数である。正解率は示されたピーク学習率で 10 エポック訓練した後の結果であり、複数回の実行にわたる範囲として示す（単一の値は、各実行の差が 0.5 ポイント以内に収まったことを意味する）。LeNet の結果は一定の学習率で 30 エポック訓練した後の値である。

ネットワーク	入力	パラメータ数	積和演算数	ピーク学習率	検証正解率
LeNet	28×28	0.06M	0.4M	2.0	88〜90%（30 エポック）
AlexNet	224×224	46.8M	938M	0.03	91%
VGG（1/4 幅）	224×224	43.1M	512M	0.1	93%
VGG（1/4 幅）	224×224	43.1M	512M	0.03	91〜92%
VGG（1/4 幅、プーリングヘッド）	224×224	0.6M	470M	0.03	89〜90%
NiN	224×224	3.4M	798M	0.03	89%
GoogLeNet	96×96	6.0M	276M	0.03	91%

Table 7.2 からは、3 つの教訓が浮かび上がる。

1. **単純な演算量は正解率を決定しない：**1/4 幅の VGG は AlexNet の半分の積和演算で最高の正解率を達成する一方、NiN は AlexNet とほぼ同等の計算量を費やしながら 2 ポイント低い結果に終わる。
2. **ボトルネックとステムのダウンサンプリングがパラメータとピクセルの予算を大幅に削る：**GoogLeNet は $8\times$ のパラメータ削減（6.0M 対 46.8M）と 96×96 の入力（積和演算 276M 回。ただし 224×224 では約 1.5B 回の積和演算を実行する）により、AlexNet の 91% の正解率に並ぶ。
3. **グローバル平均プーリングは、幅の狭いトランクでのわずかな正解率のマージンと引き換えに膨大なパラメータを削減する：**`VGG` の密なヘッドをグローバル平均プーリングに置き換える（`PooledVGG`）と、Fashion-MNIST で約 2 パーセントポイントのコストを払う代わりにパラメータの 99%（43.1M $\rightarrow$ 0.6M）が削減される。一方 ImageNet では、より深いボディにプーリングヘッドを組み合わせた GoogLeNet が、144M パラメータの VGG モデルに圧勝した。

また、4 つのネットワークすべてに共通する限界にも注目してほしい。いずれも中間の活性化値を正規化しておらず、補助ヘッドや段階的な事前学習なしには約 22 層を超えられない。さらに、ピーク学習率を 0.03 から 0.1 に上げると、NiN、`PooledVGG`、GoogLeNet はいずれも不安定になる。7.3 節では、このスケールに対する脆弱性を直接解決する。

7.2.5 まとめ

AlexNet 以降、深層視覚モデルの設計は、個々の層を手作業で設計する手法からモジュール化されたブロックを組み合わせる手法へと移行した。VGG はパディング付き 3×3 畳み込みとプーリングの反復可能な積み重ねを導入し、小さなカーネルの積み重ねによってより少ない重みとより多くの非線形性で広い受容野を捉えられることを示すとともに、ネットワークトポロジをデー

タへと変えた。NiN はピクセルごとにチャネルを混合する 1×1 畳み込みを導入し、AlexNet や VGG で重みの 90% 超を占めていた巨大な密な分類ヘッドを、パラメータ不要のグローバル平均プーリングに置き換えた。GoogLeNet は、明快なステム・ボディ・ヘッド構成の内部で、マルチスケールの並列ブランチと 1×1 ボトルネックを融合させた。共通のコサインウォームアップスケジュールを用いた Fashion-MNIST 上での比較では、1/4 幅の VGG が最も高い正解率を示し、GoogLeNet は 8 分の 1 のパラメータ数で AlexNet に並び、VGG の密なヘッドをグローバル平均プーリングに置き換えると約 2 パーセントポイントの低下と引き換えにパラメータの 99% を削減できる。

7.2.6 演習問題

1. **設定から見る VGG-16**。VGG-16 は、224×224 の RGB 画像に対し、`arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 512))` の 5 つのブロックで 13 個の畳み込み層を積み重ね、その後に 4096、4096、1000 ユニットの 3 つの全結合層を続ける。
   1. 畳み込みボディと全結合ヘッドのそれぞれについてパラメータ数と積和演算数を手計算で求め、Simonyan and Zisserman (2015) が報告している 1 億 3800 万パラメータと合計が一致することを確認せよ。
   2. 単一の層として最も多くのパラメータを持つのはどの層か。最も多くの積和演算を実行するのはどの層か。また、複数の畳み込み層が同数で並ぶのはなぜか。
2. [code] **元の解像度**。Fashion-MNIST の画像は 28×28 であるため、224×224 へのアップサンプリングは情報を増やすことなく計算量を消費している。
   1. 1/4 幅の `VGG`（`arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128))`、`lr=0.1`）を、56×56 および 84×84 にリサイズした画像で訓練せよ。各解像度において密なヘッドに入力される特徴マップの空間サイズを示し、検証正解率と 1 エポックあたりの実行時間を 224×224 での実行結果と比較せよ。
   2. `GoogLeNet` はすべてのストライド 2 のステップで解像度が半分（切り上げ）になるようパディングしているため、変更なしで 28×28 の入力を受け付ける。28×28 入力に対して、3 つの Inception ステージがそれぞれどの空間解像度で動作するか計算せよ。最終ステージにおいて、3×3 ブランチと 5×5 ブランチは何を計算しているか。
3. **削減効果の源泉**。●本節の層ごとの集計と Table 7.1 を用いて、以下の問いに答えよ。
   1. 10 クラスの単一チャネル 224×224 画像に対する VGG-11、`NiN`、`GoogLeNet` について、パラメータ数と積和演算数を畳み込み層と分類ヘッドにそれぞれ切り分けよ。
   2. VGG-11 と比べて、`NiN` がより多くのパラメータを削減しているのはボディか、それともヘッドか。`GoogLeNet` についてはどうか。積和演算数についても同じ問いに答え、それぞれの削減をもたらしている設計上の選択を挙げよ。
4. [code] **単一ブランチ Inception のアブレーション**。●Inception ブロックにおいて 3×3 ブランチ（1×1 による削減を含む）のみを残した `GoogLeNet` の変種を構築せよ。その際、3×3 畳み込みには元の 4 ブランチブロック全体と同じチャネル数を出力させ、各ブロックのパラメータ数が元の 10% 以内に収まるように 1×1 削減の幅を選べ。両方のネットワークを 96×96 の Fashion-MNIST 上で訓練し、検証正解率を比較せよ。この結果から、1 つのブロック内で複数のカーネルサイズを混合することについて何が言えるか。

7.3 正規化層

6.6 節で LeNet を訓練した際、MLP では問題なく機能した学習率が通用しなくなった。0.1 では検証正解率が数エポックにわたりランダム推測レベルに留まり、10 エポック後でも約 73% にしか達しなかったため、学習率を 20 倍の 2.0 に引き上げ、30 エポック訓練してようやく約 89% に到達させる必要があった。同様の脆さは 7.2 節でも見られた。そこではピーク学習率 0.1 で VGG は順調に訓練できた一方、NiN と GoogLeNet は停滞または発散してしまった。5 層のネットワ

ークがステップサイズに対してこれほど敏感であるはずがない。なぜ実用的な学習率がアーキテクチャにこれほど微妙に依存するのか、そしてネットワーク内部に層を挿入してその脆さを取り除くことはできるのだろうか。

根本的な原因は、信号と勾配のスケールが深さ方向に伝播するにつれてドリフトすることにある。LeNet の 4 つのシグモイド層はそれぞれ、逆伝播される勾配を元の $1/4$ 以下に縮小するため、浅い層の更新量は深い層に比べて極端に小さくなり、単一の学習率ではネットワーク全体に適合しなくなる。各層が次の層へ入る活性化値のスケールを変化させるため、深さを増すほどこの問題は深刻になる。ReLU 活性化関数と慎重な初期化（4.4.2.2 節）によってステップ 0 における活性化値の分散は均衡させられるものの、勾配更新によって重みが動き始めると、それらのスケールを揃え続ける仕組みは何もない。

バッチ正規化（batch normalization）は、訓練全体を通じてこの問題を動的に解決する (Ioffe and Szegedy 2015)。層の非線形性の直前に配置されたバッチ正規化は、現在のミニバッチの平均と分散を使って各特徴量を標準化し、さらに学習されたスケールとシフトを適用する。この単純な変換によって順伝播の活性化値と逆伝播の勾配の両方が安定し、正規化された各層は重みのリスケーリングに対して不変になる。以下で確認するように、バッチ正規化を加えることで、LeNet は MLP と同じ学習率だけでなく、300 倍もの幅にわたるステップサイズで順調に訓練できるようになる。次節で扱う残差接続と並び、バッチ正規化は 100 層を超える深さのネットワークを実現可能にした。ただし、その高い経験的性能を支える正確なメカニズムについては、現在も研究者の間で議論が続いている。一方で、バッチ正規化はミニバッチ内の複数の例を結合するため、ミニバッチが非常に小さい場合には性能が低下する。そこで本節の後半では、各例を単独で正規化する**層正規化**（layer normalization）と**グループ正規化**（group normalization）を導入する。

7.3.1 バッチ正規化

7.3.1.1 ミニバッチにわたる標準化

ミニバッチ $\mathcal{B}$（例数 m）にわたって全結合層が計算する活性化前の単一の特徴量を考え、得られるスカラー値を $x_1, \ldots, x_m$ とする。バッチ正規化は、このミニバッチにわたる標本平均と標本分散を次式のように計算する。

$$\hat{\mu}_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i,$$
$$\hat{\sigma}^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \hat{\mu}_{\mathcal{B}})^2,$$

そして、各値を次の式で変換する。

$$\mathrm{BN}(x_i) = \gamma \frac{x_i - \hat{\mu}_{\mathcal{B}}}{\sqrt{\hat{\sigma}^2_{\mathcal{B}} + \epsilon}} + \beta. \tag{7.3}$$

この分数項は、ミニバッチにわたって特徴量を平均 0、分散 $\hat{\sigma}^2_{\mathcal{B}}/(\hat{\sigma}^2_{\mathcal{B}} + \epsilon)$（1 よりわずかに小さい値）へと標準化する。微小な定数 $\epsilon > 0$（通常は 10^{-5}）は、ミニバッチ内のすべての値が一致した際のゼロ除算を防ぐためのものである。複数の特徴量を持つ層では、各特徴量がそれぞれのミニバッチ平均と分散を用いて独立に正規化される。

具体的な確認として、4 つの値 $(2, 4, 6, 8)$ は平均が 5、分散が $(9 + 1 + 1 + 9)/4 = 5$ であり、（ϵ を無視すれば）$(x_i - 5)/\sqrt{5} \approx (-1.34, -0.45, 0.45, 1.34)$ に標準化される。ここで、

$(20, 40, 60, 80)$ や $(102, 104, 106, 108)$ もまったく同じ 4 つの数値に写されることに注目しよう。標準化は、入力される特徴量の任意のスケールとオフセットの両方を取り除くのである。

しかし、平均 0・分散 1 という分布が、後続のあらゆる活性化関数にとって理想的とは限らない。たとえばシグモイドは、分散 1 の入力領域ではほぼ線形に振る舞ってしまう。表現の自由度を完全に回復するため、Equation 7.3 では特徴量ごとに学習可能な**スケール** γ と**シフト** β を導入し、$\gamma = 1$、$\beta = 0$ に初期化してネットワークの重みとともに訓練する。γ と β をデータセットの真の標準偏差と平均にそれぞれ設定すれば（ミニバッチのサンプリングによる変動を除いて）正規化前の元の特徴量が復元されるため、バッチ正規化がネットワークの表現力を損なうことはない。

7.3.1.2 スケール不変性

特徴量のスケールを取り除くことは、最適化に対して 2 つの強力な効果をもたらす。1 つ目は直接的な効果であり、バッチ正規化がなぜ LeNet を救うのかを端的に説明する。ミニバッチ内で特徴量がほとんど変化しない場合（たとえば飽和しかけたシグモイドを通過して $1/2$ 付近に集中している場合）、その微小な標準偏差 $\hat{\sigma}_{\mathcal{B}}$ で割ることで順伝播において分散 1 が回復され、連鎖律によって逆伝播される勾配も同じ係数 $1/\hat{\sigma}_{\mathcal{B}}$ 倍に増幅される。信号の消失と勾配消失の双方が自動的にリスケーリングされるのである。

2 つ目の効果はより巧妙で、層の重み自体に作用する。$x_i = \mathbf{w}^\top \mathbf{h}_i + b$ を例 i の活性化前の値とする。$\mathbf{w}$ と b を任意の正の定数 $a > 0$ 倍にスケーリングすると、すべての x_i、その平均、および標準偏差が a 倍になり、（ϵ を無視すれば）正規化された出力は変化しない。

$$\mathrm{BN}(a\mathbf{w}^\top \mathbf{h}_i + ab) = \mathrm{BN}(\mathbf{w}^\top \mathbf{h}_i + b).$$

その結果、損失は $\mathbf{w}$ の**向き**のみに依存し、ユークリッドノルムには依存しない。両辺を $\mathbf{w}$ に関して微分すると、$a\mathbf{w}$ における勾配は、係数 $1/a$ を $\mathbf{w}$ における勾配に掛けたものに等しいことがわかる (Ioffe and Szegedy 2015)。すなわち、小さい重みは大きな勾配を受け取り、大きい重みは小さな勾配を受け取る。

過度に大きな学習率 η を選んだ場合に何が起きるか考えてみよう。損失は $\mathbf{w}$ の動径方向の半直線上で一定であるため、勾配 $\mathbf{g}$ は $\mathbf{w}$ と直交し、SGD の各ステップは重みのノルムを厳密に増加させる。すなわち、$\|\mathbf{w} - \eta\mathbf{g}\|^2 = \|\mathbf{w}\|^2 + \eta^2\|\mathbf{g}\|^2$ が成り立つ。過大なステップは $\|\mathbf{w}\|$ を急速に膨張させ、それによって後続の勾配はすべて同じ比率で縮小する。こうして、大きな学習率に対する自己安定化のフィードバックループが自然に形成されるのである (Arora et al. 2019)。

ただし、この解析の前提条件には留意しておく必要がある。厳密には、この議論は重み減衰（weight decay）なしの単純な SGD を仮定しており（重み減衰の下では、外向きの勾配による増大と内向きの減衰が釣り合う平衡状態に重みノルムが落ち着く）、直後に正規化が続く層のみに適用でき（分類器の最終出力層には当てはまらない）、さらに重みノルムが成長する前のごく初期の数ステップにおける発散を防ぐことはできない。こうした留保はあるものの、スケール不変性（scale invariance）は、正規化されたネットワークがはるかに広い学習率の範囲を許容できる理由を最も明快に説明してくれる。

7.3.1.3 全結合層と畳み込み層

重み $\mathbf{W}$、バイアス $\mathbf{b}$、非線形性 ϕ を持つ全結合層では、Ioffe and Szegedy (2015) で提案されたように、アフィン変換と活性化関数の間にバッチ正規化を配置する。

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.4}$$

ここでは、各出力ユニットを独立した特徴量として扱い、それぞれ固有のミニバッチ平均、分散、γ_j、β_j を持たせる。

畳み込み層では、バッチ正規化は平行移動同変性（translation equivariance）を保つため、各出力**チャネル**を 1 つの特徴量として扱い、m 個のミニバッチ例**および** $p \times q$ 個のすべての空間位置の両方にわたって単一の平均と分散を計算する（チャネルあたり $m \cdot p \cdot q$ 個の値）。LeNet の最初の畳み込み層（チャネル数 6、空間サイズ 28×28、バッチサイズ 128）では、各チャネルの統計量は $128 \cdot 28 \cdot 28 = 100{,}352$ 個の要素を集約するため、全結合層のユニットあたり 128 個の要素よりもはるかに安定する。同様に、正規化された層が平行移動同変であり続けるよう、各チャネルはすべての空間座標にわたって単一のスカラー γ と β を共有する。

7.3.1.4 訓練と予測

推論時には、デプロイされたモデルが画像を 1 枚ずつ評価することも多く、ある入力に対する予測は、たまたま同じミニバッチに入った他の画像に依存するのではなく決定論的でなければならない。そのため、バッチ正規化は 2 つの異なるモードで動作する。**訓練モード**（training mode）では、Equation 7.3 のように現在のミニバッチの統計量を用いて正規化しつつ、各ステップの後に指数移動平均（exponential moving average）である $\hat{\mu}$ と $\hat{\sigma}^2$（それぞれ 0 と 1 に初期化）を次式のように更新する。

$$\begin{aligned} \hat{\mu} &\leftarrow \rho\, \hat{\mu} + (1-\rho)\, \hat{\mu}_{\mathcal{B}}, \\ \hat{\sigma}^2 &\leftarrow \rho\, \hat{\sigma}^2 + (1-\rho)\, \hat{\sigma}^2_{\mathcal{B}}. \end{aligned} \tag{7.5}$$

減衰（decay）を表す係数 ρ は、この移動統計量の実効的なウィンドウ幅を決める。k ステップ前のミニバッチは重み $(1-\rho)\rho^k$ を持つ。$\rho = 0.9$ のとき、重みは約 7 ステップごとに半減し、直近の約 $1/(1-\rho) = 10$ 個のミニバッチ（Fashion-MNIST の 1 エポックあたり 469 個あるサイズ 128 のミニバッチのうち）にわたって平均することになる。**予測モード**（prediction mode）では、層はこれらの移動推定値を固定し、それを用いてすべての入力を正規化する。これにより、バッチ正規化は固定された決定論的なアフィン変換に帰着する。このように、ドロップアウト（4.6 節）と同様、バッチ正規化は訓練時と評価時で異なる振る舞いをする。

ライブラリ間における悪名高い API の不統一には注意が必要である。パラメータ ρ を `momentum` と呼ぶフレームワークがある一方、$1-\rho$ を `momentum` と呼ぶフレームワークもあり、しかもそのどちらも最適化のモメンタムとは何の関係もない。本章全体を通して $\rho = 0.9$ および $\epsilon = 10^{-5}$ に統一し、フレームワークのデフォルト値が異なる箇所ではこれらを明示的に渡すことにする。

7.3.2 ゼロからの実装

以下の `batch_norm` は、全結合層（2D）と畳み込み層（4D）の両方の入力に対し、訓練モードと予測モードにわたって Equation 7.3 と Equation 7.5 の双方を実装したものである。

画像は（例、チャネル、高さ、幅）の形式で格納されるため、チャネル軸は 1 である。

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum,
               training):
    if not training:
        # Prediction mode: normalize with the running estimates
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        # Fully connected: average over examples. Convolutional: over
```

```
        # examples and positions, one mean and variance per channel
        dims = 0 if len(X.shape) == 2 else (0, 2, 3)
        mean = X.mean(dim=dims, keepdim=True)
        var = ((X - mean) ** 2).mean(dim=dims, keepdim=True)
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the running estimates; `momentum` is the decay rho
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.detach(), moving_var.detach()
```

batch_norm を BatchNorm 層にラップするには、2 種類の異なる状態を管理する必要がある。1 つはオプティマイザによって更新される学習可能なパラメータ（gamma と beta）であり、もう 1 つはオプティマイザからは無視されるものの、重みとともにモジュールがディスクへ保存し GPU へ転送する訓練対象外のバッファ（moving_mean と moving_var）である。

```
class BatchNorm(nn.Module):
    # `num_features`: outputs of a fully connected layer or channels of a
    # convolutional layer. `num_dims`: 2 or 4, the number of input axes
    def __init__(self, num_features, num_dims):
        super().__init__()
        shape = (1, num_features) if num_dims == 2 else (1, num_features, 1, 1)
        # Scale and shift, learned, start at 1 and 0
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # Running estimates, not learned, start at 0 and 1
        self.register_buffer('moving_mean', torch.zeros(shape))
        self.register_buffer('moving_var', torch.ones(shape))

    def forward(self, X):
        Y, moving_mean, moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean, self.moving_var,
            eps=1e-5, momentum=0.9, training=self.training)
        self.moving_mean.copy_(moving_mean)
        self.moving_var.copy_(moving_var)
        return Y
```

register_buffer は、model.to(device) で移動され state_dict() で保存されるが、parameters() からは返されないテンソルを登録する。属性 self.training は model.train() と model.eval() によって設定され、Trainer が訓練と検証の前にこれらを呼び出す。

ゼロから構築した実装を検証するため、BatchNorm とフレームワークの組み込み層（いずれも $\rho = 0.9$、$\epsilon = 10^{-5}$）の両方を、平均 2、標準偏差 3 で生成した 128 個の 6 チャネル 28×28 特徴マップからなる同一のミニバッチに対して実行する。以下では、すべての出力チャネルが平均 0、分散 1 になること、出力がフレームワークの組み込み層と一致すること、そして 1 ステップ後に移動平均が適切に更新されることを確認する。

```
X = 2 + 3 * torch.randn(128, 6, 28, 28)
bn, bn_ref = BatchNorm(6, num_dims=4), nn.BatchNorm2d(6)  # rho 0.9, eps 1e-5
with torch.no_grad():
    Y, Y_ref = bn(X), bn_ref(X)  # New layers are in training mode
```

```
{'largest channel mean': float(Y.mean(dim=(0, 2, 3)).abs().max()),
 'average channel variance': float(Y.var(dim=(0, 2, 3),
                                         unbiased=False).mean()),
 'largest difference': float((Y - Y_ref).abs().max()),
 'running means': bn.moving_mean.reshape(-1)}
```

```
{'largest channel mean': 1.8056253736631334e-07,
 'average channel variance': 0.9999989867210388,
 'largest difference': 7.152557373046875e-07,
 'running means': tensor([0.1994, 0.1993, 0.2000, 0.2007, 0.1991, 0.1981])}
```

単精度の丸め誤差の範囲内ですべてのチャネルが平均 0、分散 1 となり、実装した層の出力は組み込み層と 10^{-6} 以内で一致している。1 回のミニバッチ処理の後、移動平均は初期値の 0 からミニバッチ平均の 2 に向けて $1 - \rho = 0.1$ の割合だけ進んでいる。

`nn.BatchNorm2d(6)` は PyTorch のデフォルトである `momentum=0.1` を使用するが、これは本節の $\rho = 0.9$ に相当する。

7.3.3 バッチ正規化を用いた LeNet

ここで LeNet（6.6 節）に戻り、バッチ正規化の効果を試してみよう。まったく同じ Xavier 初期化を保ったまま、`BNLeNet` では LeNet の 4 つの隠れ層（6 チャネルと 16 チャネルの畳み込み、および 120 ユニットと 84 ユニットの全結合層）のそれぞれの直後、各シグモイド活性化関数の直前に組み込みのバッチ正規化層を挿入する。これら 4 つの層ではバイアスを無効化している点に注目してほしい。ミニバッチ平均を引く操作によってバッチ正規化の直前にある加法的なバイアスは完全に相殺され、β がその役割を引き継ぐからである。正規化される全 $6 + 16 + 120 + 84 = 226$ 個の特徴量にわたり、226 個のバイアスを 226 組の (γ, β) ペアに置き換えることで、LeNet の 61,706 個のパラメータに加わる訓練可能なパラメータはわずか 226 個にとどまる。

```
class BNLeNet(d2l.Classifier):
    """LeNet with batch normalization before each hidden sigmoid."""
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(  # No biases before batch normalization
            nn.LazyConv2d(6, kernel_size=5, padding=2, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(84, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    weight_init = d2l.LeNet.weight_init  # Xavier weights, zero biases
```

`LazyBatchNorm2d` は 4 軸入力の各チャネルを正規化し、`LazyBatchNorm1d` は 2 軸入力の各特徴量を正規化する。これらのデフォルトである `momentum=0.1` と `eps=1e-5` は、本節の $\rho = 0.9$ と ϵ に相当する。`BNLeNet` は LeNet の `weight_init` を再利用しており、0 に設定すべきバイアスを持つのは出力層のみである。

訓練に入る前に、予測モードと学習率の間の微妙な相互作用に注目しておこう。移動平均は約 10 ミニバッチ分の遅れを伴うため、ネットワークの重みがまだ急速に変化している間は、少し古い時点のネットワークのスナップショットを表すことになる。これは単なる理論上の話ではない。一定の学習率 0.1 の下では、BNLeNet の予測モードにおける検証正解率が、最終エポックを含む個々のエポックで突然 10 パーセントポイント以上も急落することがある。学習率を滑らかに 0 へと減衰させれば、この問題はきれいに解消する。ステップサイズが縮むにつれて重みが落ち着き、移動平均が最終的なネットワークに収束するからである。そこで、7.1 節で用いたウォームアップ付きコサインスケジュール（詳しくは第 9 章で学ぶ）を再利用し、num_steps の最初の 10 分の 1 でウォームアップを行うとともに、両モデルがまったく同じスケジュールを共有できるよう d2l.LeNet をサブクラス CosineLeNet でラップする。

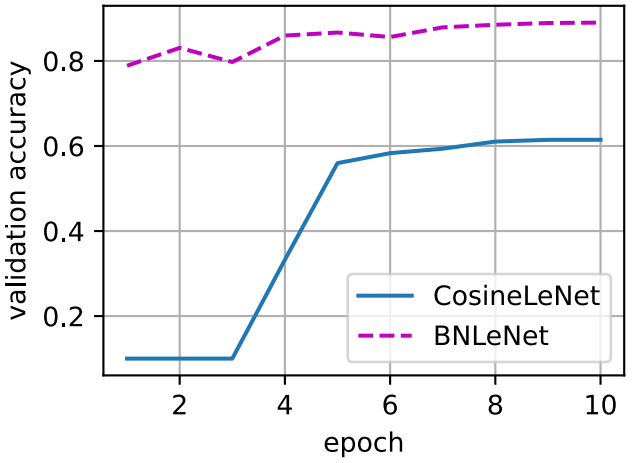
```
@d2l.add_to_class(BNLeNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch

class CosineLeNet(d2l.LeNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(lr, num_classes)
        self.num_steps = num_steps

    configure_optimizers = BNLeNet.configure_optimizers
```

CosineLeNet と BNLeNet の両方を同一の設定（ピーク学習率 0.1、バッチサイズ 128、10 エポック）で訓練し、エポックごとの検証正解率を追跡してみよう。

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 10 * len(data.train_dataloader())
curves = {}
for model in (CosineLeNet(0.1, num_steps), BNLeNet(0.1, num_steps)):
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False  # We plot the accuracies ourselves
    trainer.fit(model, data)
    curves[type(model).__name__] = trainer.history['val_acc']
```

curves に記録されたエポックごとの検証正解率の推移をプロットすると、CosineLeNet と BNLeNet の収束速度を直接対比できる。

```
d2l.plot(list(range(1, 11)), list(curves.values()), 'epoch',
         'validation accuracy', legend=list(curves))
```



また、最終的な到達性能を比較するため、両モデルの 10 エポック後の検証正解率 `acc[-1]` を `curves` から取り出して表示する。

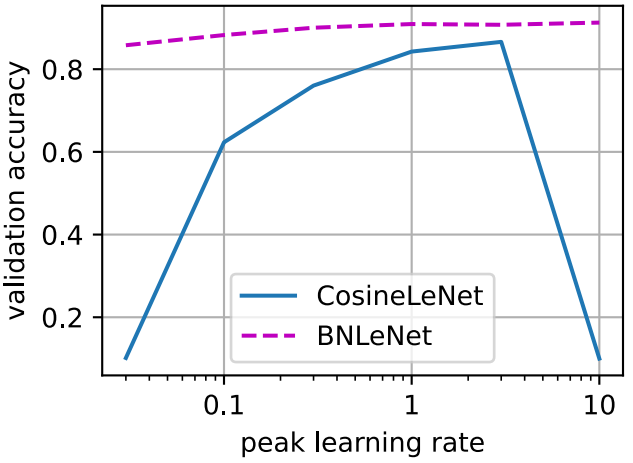
```
{name: round(acc[-1], 3) for name, acc in curves.items()}  # After 10 epochs
```

```
{'CosineLeNet': 0.615, 'BNLeNet': 0.89}
```

バッチ正規化なしの LeNet は、最初の 2〜5 エポックにわたりランダム推測レベル（10%）で停滞し、10 エポック後でも約 60% にしか達しない（6.6 節の 73% より低いのは、コサイン減衰によって平均ステップサイズが半分になるためと考えられる）。一方、バッチ正規化を加えると、まったく同じアーキテクチャが最初の 1 エポック目で 70% を超え、10 エポック後には約 88% まで上昇する。これは、正規化なしの LeNet が 20 倍の学習率と 3 倍のエポック数を費やしてようやく到達した 88〜90% にほぼ匹敵する水準である。また、学習率が 0 へと減衰するにつれて、終盤のエポックで曲線が滑らかに平坦化していることにも注目してほしい。わずか 226 個のパラメータを追加したことでネットワークの表現容量が広がったのではなく、単にすべての層の活性化値と勾配が健全なスケールに保たれるようになったのである。

しかし、単一の学習率で比較しただけでは重要な疑問が残る。バッチ正規化は最適なステップサイズの位置を単にずらしただけなのか、それとも実用的な学習率の範囲を本当に広げたのだろうか。これを確かめるため、ピーク学習率を 0.03 から 10 までの 300 倍の幅にわたって変化させ、両モデルの 10 エポック後の最終検証正解率をプロットする。

```
lrs = (0.03, 0.1, 0.3, 1, 3, 10)
final = {'CosineLeNet': [], 'BNLeNet': []}
for lr in lrs:
    for cls in (CosineLeNet, BNLeNet):
        trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
        trainer.board.display = False
        trainer.fit(cls(lr, num_steps), data)
        final[cls.__name__].append(trainer.history['val_acc'][-1])
d2l.plot(list(lrs), list(final.values()), 'peak learning rate',
         'validation accuracy', legend=list(final), xscale='log')
```



このスイープ実験は決定的な答えを示している。通常の LeNet がうまく訓練できるのは 1〜3 付近の狭い帯域（84〜87% に到達）に限られる。この範囲より下では学習が遅すぎて 10 エポックでは収束せず、10 では挙動が不安定になってランダム推測レベルで停滞することもある。対照的に、`BNLeNet` は 0.03 から 10 までの**あらゆる**学習率で約 85% を超え、1 以上の学習率では約 91% に達する（0.1 のときより丸々 3 ポイント高い）。正規化なしのネットワークが 3 倍以内の範囲で学習率を調整しなければならないのに対し、正規化されたネットワークはスケール不変性の解析が予測したとおり、300 倍もの範囲にわたって安定して機能する（ただし、最終出力層は正規化されていないため、上限が際限なく伸びるわけではない）。

バッチ正規化が深いネットワークにこれほど劇的な効果をもたらす理由は、当初提唱された内部共変量シフト（internal covariate shift）の軽減にとどまらない (Ioffe and Szegedy 2015)。その後の研究により、損失ランドスケープと勾配を滑らかにすること (Santurkar et al. 2018)、深さ方向にわたって活性化値と勾配のスケールを均衡させること、隠れ層の重みをスケール不変にして大きな学習率を自己安定化させること、そしてドロップアウトのように正則化として働く適度なミニバッチノイズを注入すること (Luo et al. 2018; Brock et al. 2021) が明らかになっている。

7.3.4 バッチ統計量を使わない正規化

ある例の出力を同じミニバッチ内の他の例と結合させるというのは、ニューラルネットワークの層としては異例の性質であり、実用上 2 つの弱点をもたらす。第 1 に、バッチサイズが小さくなるにつれてミニバッチ統計量の質が悪化する。極端な例として、1 例のみのミニバッチに対する全結合層では、すべての要素が自身の平均と一致するため、標準化された活性化値はすべて強制的に 0 となり、層は入力にかかわらず定数 β を出力してしまう。物体検出やセマンティックセグメンテーションのようにメモリ消費の大きいタスクでは、GPU あたり 1〜2 枚の高解像度画像しか載せられないことが多く、まさにバッチ統計量のノイズが最も大きくなる領域にぶつかる。第 2 に、バッチ正規化は訓練モードと予測モードで 2 つの異なる関数を計算する。たとえばドメインをまたぐファインチューニングの後など、移動推定値がテスト時の入力分布とずれてしまうと、気づかないうちに正解率が低下し、デバッグが極めて困難な不具合を引き起こすことがある。

正規化の統計量を**各例の内部で独立に**計算すれば、これら 2 つの制約はともに解消する。Figure 7.7 は、バッチ軸 N、チャネル軸 C、空間軸 $H \cdot W$ を持つ畳み込みの活性化値テンソル上で、こうして得られる正規化層の族を比較したものである。どの手法もまったく同じ標準化の式を用いており、どの要素をまとめて 1 つの平均と分散を共有させるかという点のみが異なる。

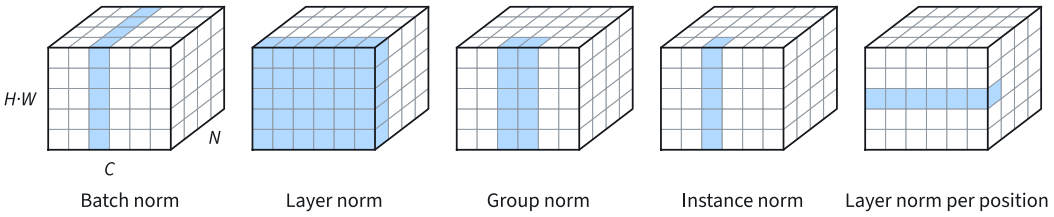


Figure 7.7: 例 N、チャネル C、空間位置 $H \cdot W$ を持つ活性化値に対する各種正規化手法。色付きの要素が 1 つの平均と 1 つの分散を共有する。バッチ正規化は各チャネルをミニバッチと全位置にわたって平均し、層正規化は各例を全チャネルと全位置にわたって平均し、グループ正規化はチャネルのグループと全位置にわたり、インスタンス正規化は単一チャネルにわたって平均する。Transformer や ConvNeXt における層正規化は、1 つの位置におけるチャネル方向にわたって平均する。

7.3.4.1 層正規化

層正規化 (Ba et al. 2016) は、単一の例が持つ全 n 個の特徴量 $\mathbf{x} = (x_1, \ldots, x_n)$ を独立に標準化する。

$$\mathrm{LN}(\mathbf{x})_j = \gamma_j \frac{x_j - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta_j,$$
$$\text{with} \quad \hat{\mu} = \frac{1}{n}\sum_{j=1}^{n} x_j \quad \text{and} \quad \hat{\sigma}^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j - \hat{\mu})^2.$$

これは Equation 7.3 を転置したものにあたる。1 つの特徴量を複数の例にわたって平均する代わりに、特徴量ごとの学習可能なスケール γ_j とシフト β_j は保ったまま、1 つの例をその特徴量方向にわたって平均するのである。計算に他の例が一切関与しないため、層正規化はあらゆるバッチサイズにおいて、また訓練モードと予測モードの両方でまったく同じように動作し、移動統計量を一切必要としない。なお、実際に「層正規化」が集約する軸はアーキテクチャによって異なる。畳み込みテンソルでは 1 つの例の全チャネルと全空間位置にわたって集約する場合がある（Figure 7.7 の左から 2 番目のパネル）のに対し、Transformer は各トークンの特徴ベクトルを独立に正規化し、ConvNeXt（7.7 節）も同様に空間方向に集約せず各ピクセルのチャネルベクトルを個別に正規化する（右端のパネル）。

7.3.4.2 グループ正規化とインスタンス正規化

畳み込み層に特化して設計された**グループ正規化** (Wu and He 2018) は、c 個のチャネルを G 個のグループ（各グループ c/G チャネル）に分割し、各例をグループごとの $(c/G) \cdot p \cdot q$ 個の要素にわたって標準化した後、チャネルごとに学習されたスケールとシフトを適用する。これは 2 つの両極端を滑らかに補間する。$G = 1$ では全チャネルと全位置にわたって集約し（Figure 7.7 の空間的な層正規化）、$G = c$ では各例の各チャネルを単独で正規化する**インスタンス正規化**（instance normalization）になる。中間の値（通常は $G = 32$）を選べば、無関係な特徴量のチャネルに単一の平均と分散の共有を強いることなく、信頼できる統計量を得るのに十分なグループあたりの要素数を確保できる。以下では、32 個のテンソルからなるバッチ（16 チャネル、4 グループ）を用いて、すべての `(example, group)` スライスが平均 0、分散 1 に標準化されること、そして先頭の 1 例のみを単独で正規化した結果がバッチ全体の中で正規化した結果と完全に一致することを確認する。

```
X = torch.randn(32, 16, 8, 8)
gn = nn.GroupNorm(4, 16).requires_grad_(False)  # 4 groups of 4 channels
Y = gn(X)
G = Y.reshape(32, 4, -1)  # Collect each (example, group) pair's numbers
(G.mean(dim=2).abs().max(), G.var(dim=2, unbiased=False).mean(),
 (gn(X[:1]) - Y[:1]).abs().max())
```

```
(tensor(3.3528e-08), tensor(1.0000), tensor(0.))
```

予想どおり、丸め誤差の範囲内ですべてのグループが平均 0、分散 1 となり、先頭の例の出力はバッチの残りの例から厳密に独立している。この独立性は正解率とのトレードオフを伴うのだろうか。バッチサイズが大きい場合、その差はごくわずかである。ImageNet 上の ResNet-50 を GPU あたり 32 枚の画像で訓練した実験では、Wu and He (2018) は top-1 誤り率がバッチ正規化の 23.6% に対してグループ正規化では 24.1% であると報告している。しかし、GPU あたり 2 枚の画像に減らすと、バッチ正規化の誤り率が 34.7% へと急増する一方、グループ正規化は 24.1% のまま安定している。このため、GPU あたりの小バッチで訓練される物体検出モデルやセグメンテーションモデルでは、グループ正規化が標準的な選択肢となっている。

7.3.4.3 ノーマライザフリーネットワーク

深い畳み込みネットワークにおいて信号のスケールを適切に保つ方法は、活性化値の正規化だけではない。**ノーマライザフリーネットワーク**（normalizer-free network、NFNet）は、正規化層を 1 つも使わずに深い ResNet 風のアーキテクチャを訓練し、（ImageNet のみで訓練されたモデルの中で）ImageNet の最先端の正解率を達成した (Brock et al. 2021)。NFNet は活性化値を正規化する代わりに、各畳み込みカーネルの重みを直接標準化することで、初期化（4.4.2.2 節）がステップ 0 でしか確立できない分散の均衡を訓練全体にわたって維持し、さらに各重みテンソルのノルムに応じた適応的勾配クリッピング（adaptive gradient clipping）を組み合わせる。本節で NFNet の実装は扱わないが、その成功は示唆に富む証拠を与えてくれる。活性化値ではなく重みを通じて信号スケールを制御しても深い畳み込みネットワークが同等にうまく訓練できるという事実は、正規化層の核心的な役割がスケール制御にあるという見方を強く裏付けている。

7.3.5 まとめ

バッチ正規化は、現在のミニバッチの平均と分散を用いて全結合層の各特徴量または畳み込み層の各チャネルを標準化し、続いて学習されたスケールとシフトを適用する。順伝播の信号と逆伝播の勾配を適切なスケールに保ち、正規化された層を重みノルムに対して不変にすることで、LeNet は 0.03 から 10 にわたるピーク学習率（正規化なしの 3 倍の帯域に対し 300 倍の帯域）で安定して訓練できるようになる。もっとも、深いネットワークにおけるその効果を完全に説明する理論は今なお未解決のままである。推論時には、バッチ正規化は訓練統計量の固定された指数移動平均へと切り替わる。例をまたぐ結合は極小のバッチサイズで破綻するため、層正規化とグループ正規化は各例の内部のみで厳密に統計量を計算し、バッチサイズへの感度と訓練・テスト間のモード分離の両方を解消する。次節では、バッチ正規化を残差接続と組み合わせ、ネットワークの深さを 100 層超へと押し広げる。

7.3.6 演習問題

1. **バッチ正規化直前のバイアス。**`BNLeNet` は正規化を伴う 4 つの層をバイアスなしで構築している。Equation 7.3 から、バッチ正規化の前にバイアス $\mathbf{b}$ を加えても訓練モードの出力も重みの勾配も変化せず、$\partial L/\partial \mathbf{b} = \mathbf{0}$ となることを示せ。また、予測モードではこのようなバイアスはどうなるか。

2. [code] **学習率の上限**。本節のスイープ実験では、BNLeNet は 10 までのすべてのピーク学習率で順調に訓練できた。
    1. BNLeNet のスイープをピーク学習率 30、100、300 まで拡張せよ。どこで訓練が破綻するか。
    2. 出力の線形層は、直後にバッチ正規化を持たない唯一の重み層である。出力層の学習率を 0.1 に固定したまま、正規化された各層のピーク学習率を 30、100、300 にわたってスイープし、この出力層が上限を決めているかどうか検証せよ。
    3. CosineLeNet と BNLeNet の両方でシグモイドを ReLU に置き換え（He 初期化を使用）、学習率のスイープを再度実行せよ。正規化なしのネットワークの実用的な帯域はどのように変化するか。また、それはなぜか。
3. [code] **移動平均と再計算された統計量**。@eq-bn-running の減衰 ρ は、移動統計量が直近いくつのミニバッチにわたって平均するかを決定する。
    1. $\rho = 0.9$ と $\rho = 0.99$ のそれぞれについて、移動平均におけるあるミニバッチの重みが最新のミニバッチの重みの 1% 未満になるのは何ステップ後か。両者の期間を Fashion-MNIST の 1 エポックにあたる 469 ステップと比較せよ。
    2. BNLeNet を一定の学習率 0.1（`d2l.Classifier.configure_optimizers`）で 10 エポック訓練せよ。各エポックの後にモデルを 2 通りの方法で評価せよ。1 つは移動平均を用いた予測モードでの評価、もう 1 つは重みを固定した状態で 100 個の訓練ミニバッチから統計量を新たに再計算した後の評価である。両方の検証曲線をコサインスケジュールの曲線と並べてプロットし、その差が生じる理由を説明せよ。
4. [code] **ミニバッチ内の結合**。バッチ正規化はミニバッチ内の複数の例を結合するため、訓練済みの BNLeNet を検証セット上で次の 3 通りの方法で評価せよ（訓練モードの実行で移動統計量が上書きされないよう、予測モードを最初に計測すること）。すなわち、(i) 予測モード、(ii) 検証バッチサイズ 128 での訓練モード、(iii) 検証バッチサイズ 2（または 1）での訓練モードである。得られた 3 つの正解率について説明せよ。
5. [code] **小バッチ訓練：BatchNorm 対 GroupNorm・LayerNorm**。●BNLeNet の派生モデルとして、2 つの畳み込み層の後の BatchNorm をグループ正規化（2 グループ）に、2 つの隠れ全結合層の後の正規化を層正規化（7.3.4.1 節）に置き換えたものを構築せよ。BNLeNet とこの派生モデルの両方を、バッチサイズ 4 およびバッチサイズ 128 で 2 エポック訓練せよ。バッチサイズが 128 から 4 に下がると各モデルの検証正解率はどのように変化するか。また、それはなぜか。
6. [code] **バッチ正規化の逆伝播**。●Equation 7.3 の $\hat{\mu}_{\mathcal{B}}$ と $\hat{\sigma}^2_{\mathcal{B}}$ はミニバッチのすべての要素に依存するため、バッチ正規化の逆伝播は全 m 個の例を結合する。
    1. 1D バッチ正規化層における勾配 $\partial L/\partial x_i$（スカラー損失 L の各入力 x_i に関する偏微分）を手計算で導出し、得られた式から恒等的に $\sum_{i=1}^{m} \partial L/\partial x_i = 0$ となることを示せ。
    2. 導出した逆伝播の式を実装し、自動微分の結果とマシン精度で一致することを検証せよ。

7.4 残差ネットワーク

ImageNet においてはここまで、AlexNet の 8 層から GoogLeNet の 22 層に至るまで、ネットワークを深くすることは一貫して高い正解率を意味しており、バッチ正規化（7.3 節）によって深さをさらに拡大するうえでの残りの障壁も取り除かれたかのように見えた。当然ながら、さらに多くの層を積み重ねれば、悪くとも過学習が始まるまでは訓練誤差が下がり続けると予想されるだろう。ところが、He et al. (2016a) がバッチ正規化と ReLU を備えた VGG 風の**プレーン**な畳み

込みの積層を深さを増しながら訓練してこの直感を検証したところ、まったく逆の結果が明らかになった。CIFAR-10 では、56 層のプレーンネットワークが 20 層のネットワークよりも大幅に高い**訓練**誤差で訓練を終えたのである。訓練セットへの当てはまりさえ悪化しているため、この**劣化問題**（degradation problem）は過学習ではなく、最適化の失敗である。この謎は深刻だ。原理的には、56 層のネットワークは 20 層の解をそのままコピーし、残りの 36 層にわたって恒等写像（identity mapping）として振る舞うだけで、少なくとも同じくらい低い訓練損失を保証できるはずである。なぜ SGD はそのような解を見つけられないのか、そして見つけられるようにするには基本ブロックをどう再設計すればよいのだろうか。

残差ネットワーク（residual network、ResNet）は、拍子抜けするほど単純な再パラメータ化（reparameterization）によってこの逆説を解決する。各ブロックに出力変換全体をゼロから学習させる代わりに、残差ブロック（residual block）は学習した補正項を入力に加算する。そのため、この補正項を 0 に近づけるだけで直ちに恒等写像が復元される。たった 1 本のこのショートカットが前例のない深さを可能にした。ImageNet では、34 層のプレーンネットワークが 18 層のモデルに遅れをとったのに対し、34 層の ResNet は 18 層の ResNet を 3 パーセントポイント近く上回り、152 層の ResNet が 2015 年の ImageNet コンペティションで優勝した。

本節ではまず、入れ子の関数クラスを通じて深さを捉え直し、残差ブロックを設計する。次に、Fashion-MNIST 上で 20 層と 56 層のプレーンネットワークおよび残差ネットワークを並べて訓練することで劣化現象を直接再現し、劣化が現れては消える様子を目の当たりにする。続いて ResNet-18 を組み立てて訓練し、最後に影響力の大きい 2 つの後継モデルを取り上げる。すなわち、グループ畳み込み（grouped convolution）を用いて同等のコストでブロックの幅を広げる **ResNeXt** と、特徴マップを加算するのではなく連結する **DenseNet** である。

7.4.1 関数クラスと深さ

あるアーキテクチャの**関数クラス**（function class）$\mathcal{F}$、すなわちパラメータを変化させたときに達成可能な入出力写像の集合を考えよう。より大きなアーキテクチャ $\mathcal{F}'$ が、$\mathcal{F}$ 内の最良の解と少なくとも同じくらい低い訓練損失をもつ解を必ず含むのはどのようなときか。それは、関数クラスが**入れ子**（nested）になっており、$\mathcal{F} \subseteq \mathcal{F}'$（Figure 7.8）を満たすときである。もし $\mathcal{F} \not\subseteq \mathcal{F}'$ であれば、クラスを大きくしても目標の関数 f^* からかえって遠ざかることさえある。

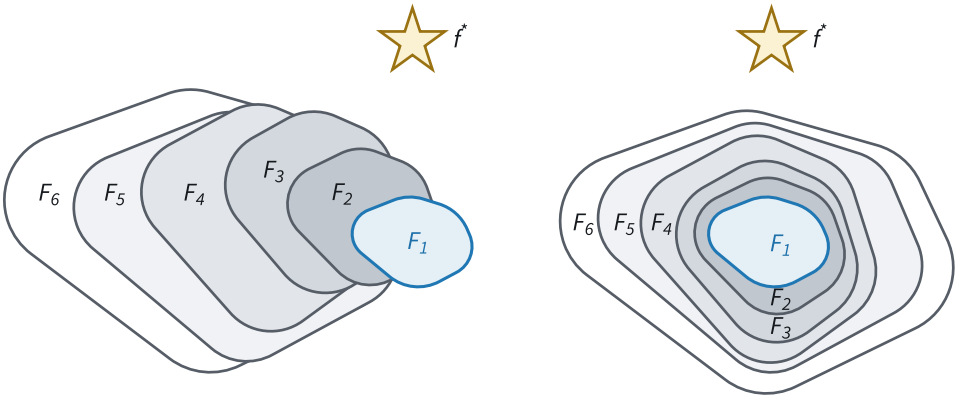


Figure 7.8: 次第に大きくなる関数クラス。左は入れ子になっていない場合であり、大きなクラスが小さなクラスの関数を含むとは限らず、目標 f^* に近づくとも限らない。右は各クラスが前のクラスをすべて含んでいるため、クラスが大きくなるにつれて利用可能な最良の関数は改善する一方である。

追加した層が恒等写像 $f(\mathbf{x}) = \mathbf{x}$ を表現できる限り、層を追加することで関数クラスは入れ子に

なる。プレーンブロック $\mathbf{x} \mapsto \mathrm{ReLU}(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}))$ は非負の入力に対して恒等写像を表現**できる**ため、プレーンネットワークはすでに Figure 7.8 の右パネルに示した入れ子の状況にある。つまり、その劣化は表現能力の不足ではなく最適化の失敗であり、SGD が何十もの非線形層にわたって恒等写像を近似するのに苦戦しているのである。残差ブロックは層を再パラメータ化し、残差分岐の重みがゼロになる原点 $g(\mathbf{x}) = 0$ に恒等写像が位置するようにする。

7.4.2 残差ブロック

目標の写像 $f(\mathbf{x})$ を直接学習する代わりに、残差ブロック（Figure 7.9）は**残差**（residual）$g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ を学習し、**ショートカット**（または**スキップ**）接続を通じて入力を足し戻すことで $\mathbf{x} + g(\mathbf{x})$ を出力する。このブロックを恒等写像にするには、g の最終層をゼロにするだけでよい。

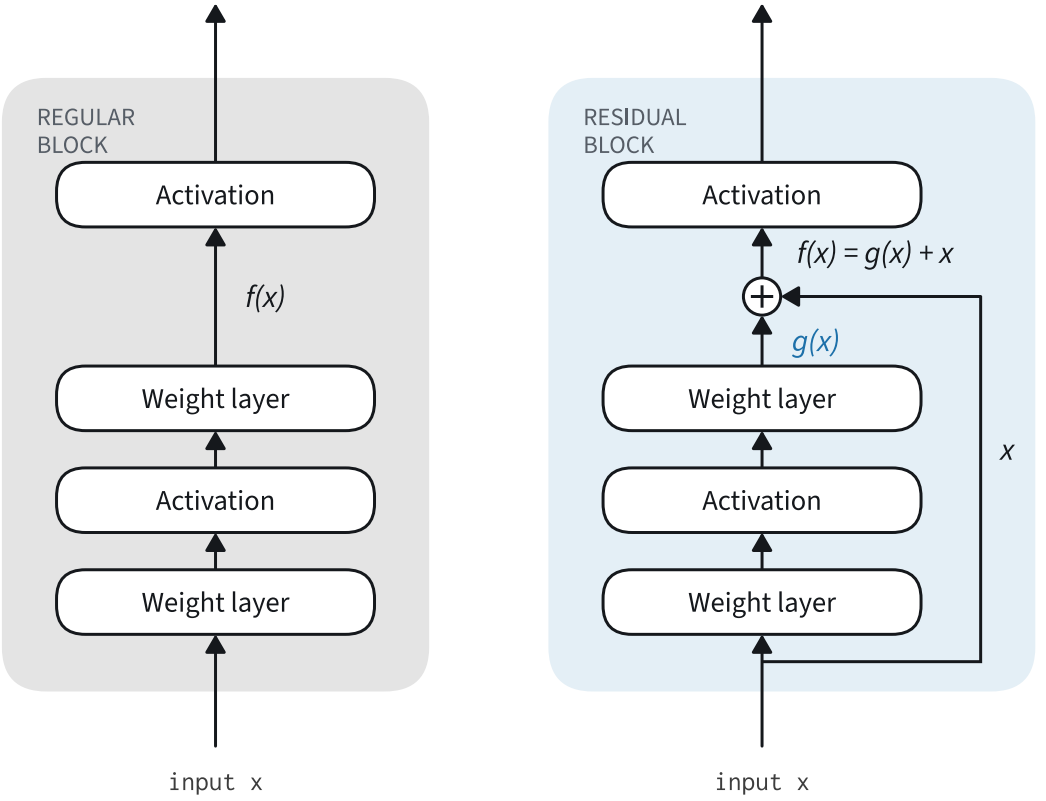


Figure 7.9: 通常のブロック（左）は写像 $f(\mathbf{x})$ を直接学習しなければならない。残差ブロック（右）は残差写像 $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ を学習して入力を足し戻すため、恒等写像 $f(\mathbf{x}) = \mathbf{x}$ は $g(\mathbf{x}) = 0$ に対応する。

ショートカットは勾配の伝わり方も変える。積み重ねた 2 つのブロック $\mathbf{x}_1 = \mathbf{x}_0 + g_1(\mathbf{x}_0)$ と $\mathbf{x}_2 = \mathbf{x}_1 + g_2(\mathbf{x}_1)$ に対して、ヤコビ行列 $\partial \mathbf{x}_2 / \partial \mathbf{x}_0 = (\mathbf{I} + \partial g_2 / \partial \mathbf{x}_1)(\mathbf{I} + \partial g_1 / \partial \mathbf{x}_0)$ を展開すると単位行列 $\mathbf{I}$ と分岐項の和になるため、損失の勾配は途中のすべての重み行列との積を経ることなく初期のブロックに到達する（連続時間の観点からは、$\mathbf{x}_{l+1} = \mathbf{x}_l + g_l(\mathbf{x}_l)$ は常微分方程式（ODE）$\dot{\mathbf{x}} = g(\mathbf{x}, t)$ の単位ステップのオイラー離散化にあたる。38.3 節を参照）。

プレーンネットワークが劣化するのは勾配が消失するためだと結論づけたくなるかもしれない。

しかしそうではない。He et al. (2016a) は、バッチ正規化を施したプレーンネットワークがどの深さでも健全な勾配ノルムを維持していることを確認した。破綻の原因はより微妙であり、その後の研究によって 2 つの相補的な説明が与えられている。深いプレーンネットワークでは、深さが増すにつれて近傍の入力における勾配の相関が失われて白色雑音に近づき、SGD が従うべき一貫した方向を見失うのに対し、ショートカットがあると相関の減衰ははるかに緩やかになる (Balduzzi et al. 2017)。また重みの側から見ると、深いプレーンネットワークの損失ランドスケープは粗くカオス的であるのに対し、ショートカットは最小点の周囲でそれを滑らかに保つ (Li et al. 2018)。上のヤコビ行列はその理由を示唆している。L 個のブロックにわたって展開すると、因子 $\mathbf{I} + \partial g_l / \partial \mathbf{x}_{l-1}$ の積は 2^L 通りの経路にわたる和となり、勾配の大半は短い経路を通って流れるため、ResNet は比較的浅いネットワークのアンサンブルのように訓練される (Veit et al. 2016)。このようなアンサンブルから 1 つのブロックを取り除いてもほとんど悪影響はなく、7.6 節で扱う確率的深さ（stochastic depth）はこの性質を正則化手法へと転用している。

具体的には、ResNet の基本的な `Residual` ブロックは、バイアスなしの 2 つの 3×3 畳み込みをバッチ正規化および中間の ReLU とともに積み重ね、2 番目のバッチ正規化の出力にショートカット入力 $\mathbf{x}$ を加算し、最後に ReLU を適用する。2 番目のバッチ正規化で $\gamma = \beta = 0$ と設定すれば分岐の出力はゼロになり、非負の入力に対して $\mathrm{ReLU}(\mathbf{x}) = \mathbf{x}$ が復元される。大規模訓練のレシピではこれを初期化に利用し、すべての残差分岐の最後の γ を 0 から始めることで、最初のステップでは各ブロックを恒等写像とし、分岐が徐々に育っていく浅いネットワークとして深いネットワークの訓練を開始する (Goyal et al. 2017; He et al. 2019)。本節で扱うネットワークはデフォルトの $\gamma = 1$ でも十分に訓練できる浅さであるため、そのままにしておく。`use_1x1conv=True` または `strides != 1` によってチャネル幅や空間解像度（spatial resolution）が変化するときは、加算が適切に定義されるよう、ショートカット側にもストライドを合わせた 1×1 畳み込みとバッチ正規化を適用する。

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides, bias=False)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   bias=False)
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides, bias=False)
            self.bn3 = nn.LazyBatchNorm2d()
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.bn3(self.conv3(X))
        Y += X
        return F.relu(Y)
```

Figure 7.10 に両方の構成を示す。use_1x1conv=True を渡すか、1 より大きいストライドを設定すると、右側の射影ショートカットが有効になる。

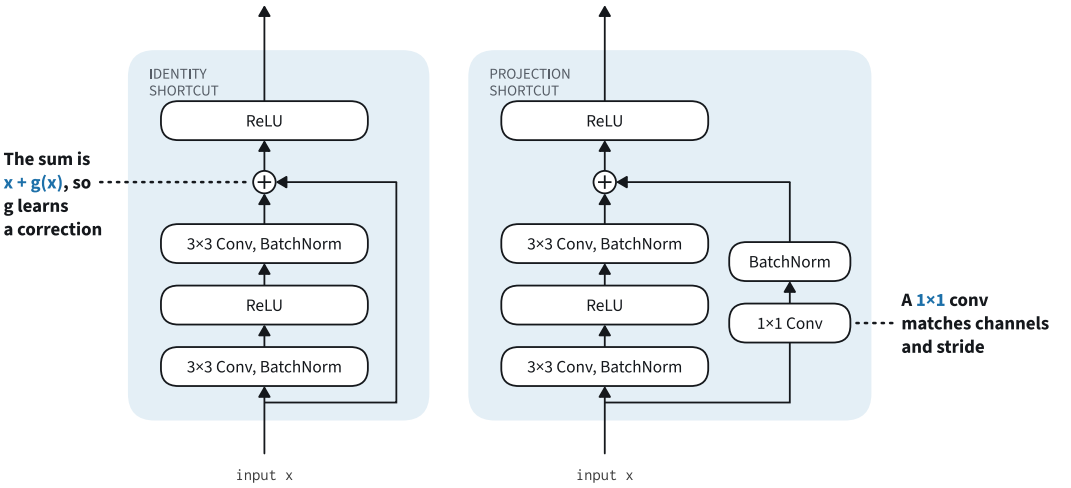


Figure 7.10: ResNet ブロックの 2 つの変種。入出力の形状が一致するときは恒等ショートカットを用い（左）、形状が異なるときはショートカット上の 1×1 畳み込みとバッチ正規化によってチャネル数と解像度を調整し、加算が定義されるようにする（右）。

出力の形状を確認してみよう。入出力のチャネル数が一致しストライドが 1 のとき、このブロックはテンソルの形状 (4, 3, 6, 6) をそのまま保つ。

```
blk = Residual(3)
X = d2l.randn(4, 3, 6, 6)
blk(X).shape
```

```
torch.Size([4, 3, 6, 6])
```

num_channels=6、use_1x1conv=True、strides=2 と設定すると、チャネル数が 2 倍になると同時に高さと幅がいずれも半分になる。これは、以下で構築する ResNet のステージ 2、3、4 の先頭に置かれるダウンサンプリングブロックそのものである。

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

7.4.3 プレーンネットワークと残差ネットワークの比較

Fashion-MNIST 上で劣化問題を直接再現し、残差ショートカットがそれを解消する様子を観察できるだろうか。条件を揃えた明確な対照実験を行うため、ショートカット接続の有無**のみ**が異なる 2 つのネットワーク族を構築する。He et al. (2016a) による CIFAR-10 向けの設計を Fashion-MNIST 本来の 28×28 画像に適合させることで、深いモデルでも高速に訓練できる。**プレーンブロック**は、迂回経路の加算を除けば Residual と層単位で同一であり、2 つの 3×3 畳み込みのそれぞれにバッチ正規化と ReLU が続く構成をとる。

```
def plain_block(num_channels, strides=1):
    """The layers of `Residual` without the shortcut."""
    return nn.Sequential(
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, stride=strides,
                      bias=False), nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, bias=False),
        nn.LazyBatchNorm2d(), nn.ReLU())
```

各ネットワークは、16 チャネルの 3×3 畳み込み、バッチ正規化、ReLU から始まり、チャネル数が 16、32、64 である n ブロックずつの 3 つのステージが続き（ステージ 2 と 3 の最初のブロックはストライド 2 でダウンサンプリングする）、最後はグローバル平均プーリングと線形分類器で終わる。ステム、ブロックあたり 2 つの畳み込み、および線形のヘッドを数えると合計 $6n + 2$ 層になり、$n = 3$ では 20 層、$n = 9$ では 56 層となる。`plain_block` または `Residual` のいずれかを `block` として渡すことで、条件を揃えた対となるモデルを構築できる。これまでと同様に、すべての重みを He 初期化（4.4.2.2 節）で初期化し、`d2l.cosine_warmup`（7.1 節）を用いてピーク学習率 0.1 で訓練する。この学習率は 7.2 節では正規化なしのネットワークを不安定化させたが、ここではすべての畳み込みの後にバッチ正規化（7.3 節）があるため滑らかに訓練が進む。

```
class DeepNet(d2l.Classifier):
    """6n + 2 layers: a 3x3 stem, three stages of n blocks, a linear head."""
    def __init__(self, block, n, lr=0.1, num_steps=2345, num_classes=10):
        super().__init__()
        self.save_hyperparameters(ignore=['block'])
        layers = [nn.LazyConv2d(16, kernel_size=3, padding=1, bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU()]
        for i, num_channels in enumerate((16, 32, 64)):
            for j in range(n):  # The first block of stages 2 and 3 downsamples
                strides = 2 if i > 0 and j == 0 else 1
                layers.append(block(num_channels, strides=strides))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    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)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr)
        schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
        return optim, schedule  # The Trainer steps it after every minibatch
```

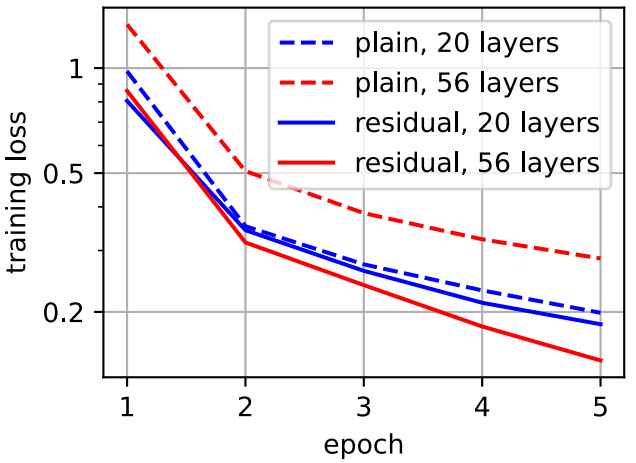
6.6 節の LeNet と同様に、最初のミニバッチによって遅延初期化層の形状が確定した時点で、Trainer がモデルの `weight_init` を呼び出す。これにより He 初期化（`kaiming_normal_`）が、`Residual` ブロックの内部も含むすべての畳み込み層と全結合層に適用される。

4 つの構成すべて（20 層および 56 層のプレーンネットワークと残差ネットワーク）をバッチサイズ 128 で 5 エポック訓練し、訓練履歴をすべて記録する。

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 5 * len(data.train_dataloader())
history = {}
for name, block in (('plain', plain_block), ('residual', Residual)):
    for n in (3, 9):
        model = DeepNet(block, n, lr=0.1, num_steps=num_steps)
        trainer = d2l.Trainer(max_epochs=5, num_gpus=1)
        trainer.board.display = False  # We plot the losses ourselves
        trainer.fit(model, data)
        history[f'{name}, {6 * n + 2} layers'] = trainer.history
```

4 本すべての訓練損失の曲線を対数スケールの y 軸上に重ねてプロットしてみよう（青が 20 層、赤が 56 層であり、破線がプレーンネットワーク、実線が残差ネットワークを表す）。

```
d2l.plot(list(range(1, 6)), [h['train_loss'] for h in history.values()],
         'epoch', 'training loss', legend=list(history), yscale='log',
         fmts=('b--', 'r--', 'b-', 'r-'))
```



また、20 層と 56 層の 4 つの構成すべてについて、最終エポックの `train_loss` と `val_acc` を `history` から確認する。

```
{name: (round(h['train_loss'][-1], 3), round(h['val_acc'][-1], 3))
 for name, h in history.items()}  # Training loss, validation accuracy
```

```
{'plain, 20 layers': (0.199, 0.913),
 'plain, 56 layers': (0.285, 0.879),
 'residual, 20 layers': (0.184, 0.907),
 'residual, 56 layers': (0.145, 0.916)}
```

プレーンネットワークには劣化現象が紛れもなく現れている。56 層のプレーンネットワークは最終的な訓練損失が 0.25〜0.30 前後と、20 層のプレーンネットワークの約 0.20 よりも高く、すべてのエポックにおいて浅いモデルの曲線よりも厳密に上に位置している。訓練損失そのものが悪化しているため、過学習の可能性は除外される。一方、残差ショートカットを有効にすると順序は完全に逆転する。56 層の残差ネットワークは訓練損失が約 0.15〜0.17 まで下がり、20 層の両

ネットワーク（ショートカットの有無にかかわらず 0.19 付近で終了）を上回る。20 層ではショートカットの影響はほとんど見られないが、56 層では追加の 36 層が最適化を妨げるか改善するかをショートカットが決定づけている。これら 4 回の実験で変えたのは深さとショートカット接続だけであるため、この差がショートカットによるものであることは明白だ。検証正解率にも同じ傾向がより緩やかに現れており、5 エポック後の時点で 56 層のプレーンネットワークが 88〜90% と遅れをとる一方、残りの 3 つは 90.5〜91.5% の狭い範囲に集まっている。

7.4.4 ResNet-18

7.4.4.1 ステム、ステージ、ヘッド

本格的な ImageNet スケールにおいて、ResNet は GoogLeNet（7.2.3 節）が確立したステム・ボディ・ヘッドのアーキテクチャを採用している。その**ステム**は、ストライド 2 で 64 チャネルの 7×7 畳み込み、バッチ正規化、ReLU、およびストライド 2 の 3×3 最大プーリングから始まり、空間次元をいきなり 4 分の 1 に縮小する。以下の `ResNet.stem` はこの入り口のモジュールを構築する。

```
class ResNet(d2l.Classifier):
    def stem(self):
        return 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))

    weight_init = DeepNet.weight_init  # He initialization
```

ボディは 4 つの**ステージ**からなり、各ステージは一定のチャネル幅で `num_residuals` 個の残差ブロックを積み重ねる。ステムがすでに入力を 4 分の 1 にダウンサンプリングしているため、ステージ 1 は解像度と 64 チャネルをそのまま維持し、それ以降の各ステージは先頭にストライド 2 の射影ブロックを置いて高さと幅を半分にしつつチャネル数を 2 倍にする。`stage` メソッドはそのようなステージを 1 つ構築する。

```
@d2l.add_to_class(ResNet)
def stage(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

モデルを締めくくる**ヘッド**は NiN や GoogLeNet に倣い、グローバル平均プーリングに単一の線形分類層が続く構成をとる。`VGG` と同様に、引数 `arch` で各ステージの `(num_residuals, num_channels)` を指定し、すべての畳み込み層と線形層の重みには He 初期化を用い、出力層にある唯一のバイアスは 0 で初期化する。

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
    super(ResNet, self).__init__()
```

```
        self.save_hyperparameters()
        self.net = nn.Sequential(self.stem())
        for i, b in enumerate(arch):
            self.net.add_module(f'stage{i+1}', self.stage(*b, first_block=(i==0)))
        self.net.add_module('head', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
```

4 つの各ステージに 2 つずつ残差ブロックを配置すると **ResNet-18** になる。4 ステージ × 4 つの 3×3 畳み込み（ショートカットの 1×1 射影は除く）に、ステムの 7×7 畳み込みと最終の線形層を加えると、パラメータをもつ層は合計 18 層になる。ResNet-34 はステージごとに `(3, 4, 6, 3)` 個のブロックを用い、さらに深いモデル（ResNet-50、ResNet-101、ResNet-152）では 3 層の**ボトルネック**ブロック（$1 \times 1 \to 3 \times 3 \to 1 \times 1$）に置き換える。これは後述する ResNeXt ブロックでグループ数を 1 とした場合に一致する。Figure 7.11 に ResNet-18 の全体構造を示す。

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_steps, num_classes)
```

1 チャネルの 96×96 画像を ResNet-18 に通し、グローバル平均プーリングに向かって解像度が段階的に下がりチャネル幅が広がっていく際のテンソル形状をたどってみよう。

```
ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 64, 24, 24)
Sequential              (1, 64, 24, 24)
Sequential              (1, 128, 12, 12)
Sequential              (1, 256, 6, 6)
Sequential              (1, 512, 3, 3)
Sequential              (1, 10)
```

7.4.4.2 訓練

`ResNet18` を訓練するため、まず `DeepNet` のコサインウォームアップ付き SGD スケジュールを `configure_optimizers` によって設定する。

```
ResNet.configure_optimizers = DeepNet.configure_optimizers  # Same schedule
```

それでは、`ResNet18` を 96×96 にリサイズした Fashion-MNIST 上で、バッチサイズ 128、ピーク学習率 `lr=0.1` で 10 エポック訓練しよう。

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```

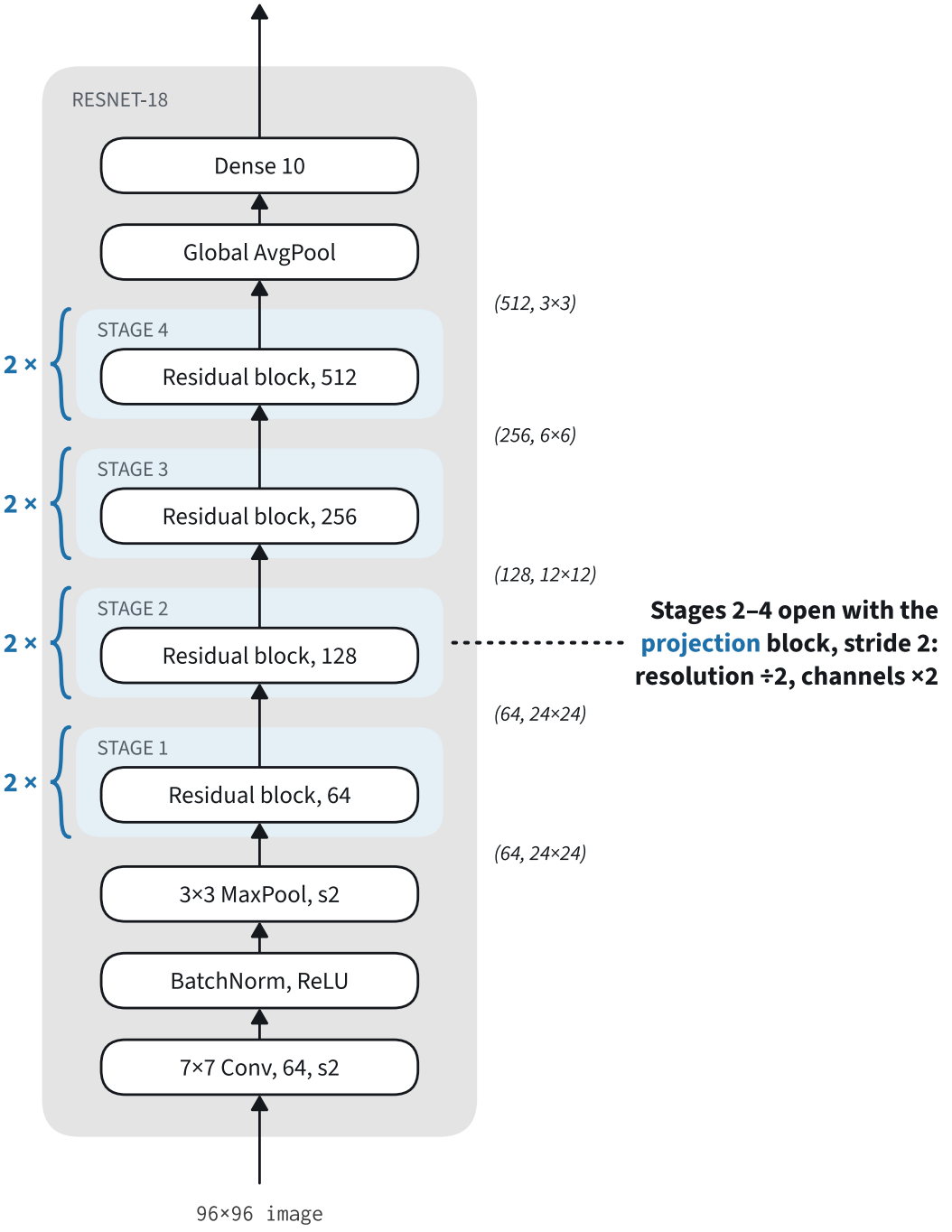
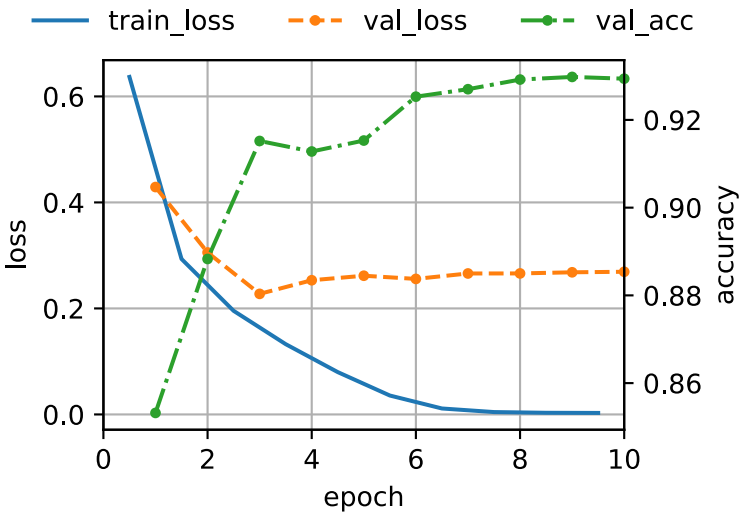


Figure 7.11: ResNet-18 の構造。7 × 7 畳み込みのステム、残差ブロックを 2 つずつもつ 4 つのステージ（ステージ 2、3、4 の最初のブロックで解像度を半分にしチャネル数を 2 倍にする）、およびグローバル平均プーリングと 1 つの全結合層からなるヘッドで構成される。形状は 96 × 96 入力に対するものである。



`trainer.history['val_acc']` の最後の要素を読み取ることで、10 エポックのコサインスケジュール終了時点での検証正解率が得られる。

```
round(trainer.history['val_acc'][-1], 3)  # Validation accuracy after 10 epochs
```

```
0.929
```

10 エポック後、ResNet-18 は約 93% の検証正解率に達し（最後の 3 エポックでの変動は 1 ポイント未満）、7.2 節の 4 分の 1 幅の VGG と同等の精度を、わずか 4 分の 1 のパラメータ数、かつ 96×96 ピクセル（224×224 ピクセルではなく）の入力で達成している。もっとも、アーキテクチャ、正規化、入力解像度がすべて同時に変化している異なるモデル族間の単一の比較を過大解釈しないよう注意すべきである。残差接続（residual connection）の真価を示す明確な証拠は、7.4.3 節で行った 20 層対 56 層の対照実験にある。

また、過学習の典型的な兆候にも注目しよう。ResNet-18 は訓練損失を 0.01 未満まで引き下げる一方、検証損失は最初の数エポック以降 0.27 前後で頭打ちになる。60,000 枚のグレースケール衣類画像に対して 1,120 万個のパラメータを訓練しているため、訓練セットの丸暗記は容易である。この汎化ギャップ（generalization gap）を縮めるにはより強力なデータ拡張（data augmentation）と正則化が必要であり、これについては 7.6 節で体系的に取り組む。

7.4.5 ResNeXt と DenseNet

残差接続は、ブロックが学習した変換に入力を加算するという形で、ブロックが入力を**どのように**先へ伝えるかを規定する。代表的な 2 つの後継アーキテクチャは、ショートカット配線という思想を受け継ぎながらブロック自体を改良している。**ResNeXt** はブロック内部の畳み込みを再設計して計算量を増やさずにチャネル幅を広げ、**DenseNet** は加算型のショートカットをチャネルの連結（concatenation）に置き換える。ResNeXt ブロックについては、7.8 節で体系的なネットワーク設計空間を探究する際に再び取り上げる。

7.4.5.1 ResNeXt

幅広い層はより豊かな表現を学習できるが、c_i チャネルを c_o チャネルに写す標準的な畳み込みのコストは $c_\text{i}c_\text{o}$ に比例して増大するため（6.4 節）、幅を 2 倍にすると重みと演算量の双方が 4 倍に

なる。**ResNeXt** (Xie et al. 2017) は、GoogLeNet の多分岐 Inception の発想（7.2.3 節）を、すべての分岐がまったく同じトポロジーをもつ完全に均一な構造へと発展させたものである。

具体的には、**グループ畳み込み**は c_i 個の入力チャネルを互いに素な g 個のグループ（各 c_i/g チャネル）に分割し、各グループを独立に畳み込んで c_o/g 個の出力チャネルを生成する（ただし g は c_i と c_o の両方を割り切ると仮定する）。各グループに必要なカーネルのサイズは $(c_\text{i}/g) \times (c_\text{o}/g) \times k \times k$ であり、全 g グループにわたる総パラメータ数と積和演算（multiply-add）の回数は g 分の 1 に削減され、$c_\text{i}c_\text{o}k^2$ から $c_\text{i}c_\text{o}k^2/g$ へと減る（実際には、メモリアクセス量が演算量と同じ割合では減らないため、実時間での高速化は通常 g 倍未満にとどまる。この違いについては 7.5 節で詳しく調べる）。歴史的には、グループ畳み込みは AlexNet (Krizhevsky et al. 2012) で初めて導入され、2 基の GPU に収めるために 5 つの畳み込み層のうち 3 つを 2 グループに分割するのに使われた。

独立したグループ同士はそれ自体では情報を交換できないため、ResNeXt ブロック（Figure 7.12）はグループ化した 3×3 畳み込みを 2 つの密な 1×1 畳み込みで挟み込む。1 つ目の畳み込みが c 個の入力チャネルを b 個の**ボトルネック**チャネルへと射影し、2 つ目の畳み込みが再び c チャネルへ射影して全グループ間の情報を混合する。2 つの 1×1 層は $2cb$ 個の重みを用い、グループ化した 3×3 層は $9b^2/g$ 個の重みを用いる。ダウンサンプリング時には、ストライド 2 の処理を 3×3 畳み込みに持たせ（最初の 1×1 射影ではなく）、ショートカットは `Residual` と同じ形にする。

「同等のコストでより幅広く」が数値的に何を意味するかを見るため、標準的な ResNet-50 のボトルネックブロック（$c = 256$、$b = 64$、$g = 1$）で使用される $2 \cdot 256 \cdot 64 + 9 \cdot 64^2 = 69{,}632$ 個の重みと、Figure 7.12 に示す ResNeXt ブロック（$b = 128$、$g = 32$）で使用される $2 \cdot 256 \cdot 128 + 9 \cdot 128^2/32 = 70{,}144$ 個の重みを比較してみよう。重みの増加を 1% 未満に抑えながら、内部のボトルネック幅が 2 倍になっている。ImageNet では、この単純なトレードオフによって ResNet-50 の top-1 誤差が 23.9% から 22.2% へと低下する (Xie et al. 2017)。

以下の `ResNeXtBlock` は、出力幅 `num_channels`（c）、グループ数 `groups`（g）、およびボトルネック比 `bot_mul`（b/c）を受け取る。`Residual` と同様に、`strides=2` を渡すと高さと幅が半分になり、ショートカットの 1×1 射影が自動的に有効になる。

```
class ResNeXtBlock(nn.Module):
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3,
                                   stride=strides, padding=1,
                                   groups=groups, bias=False)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,
```

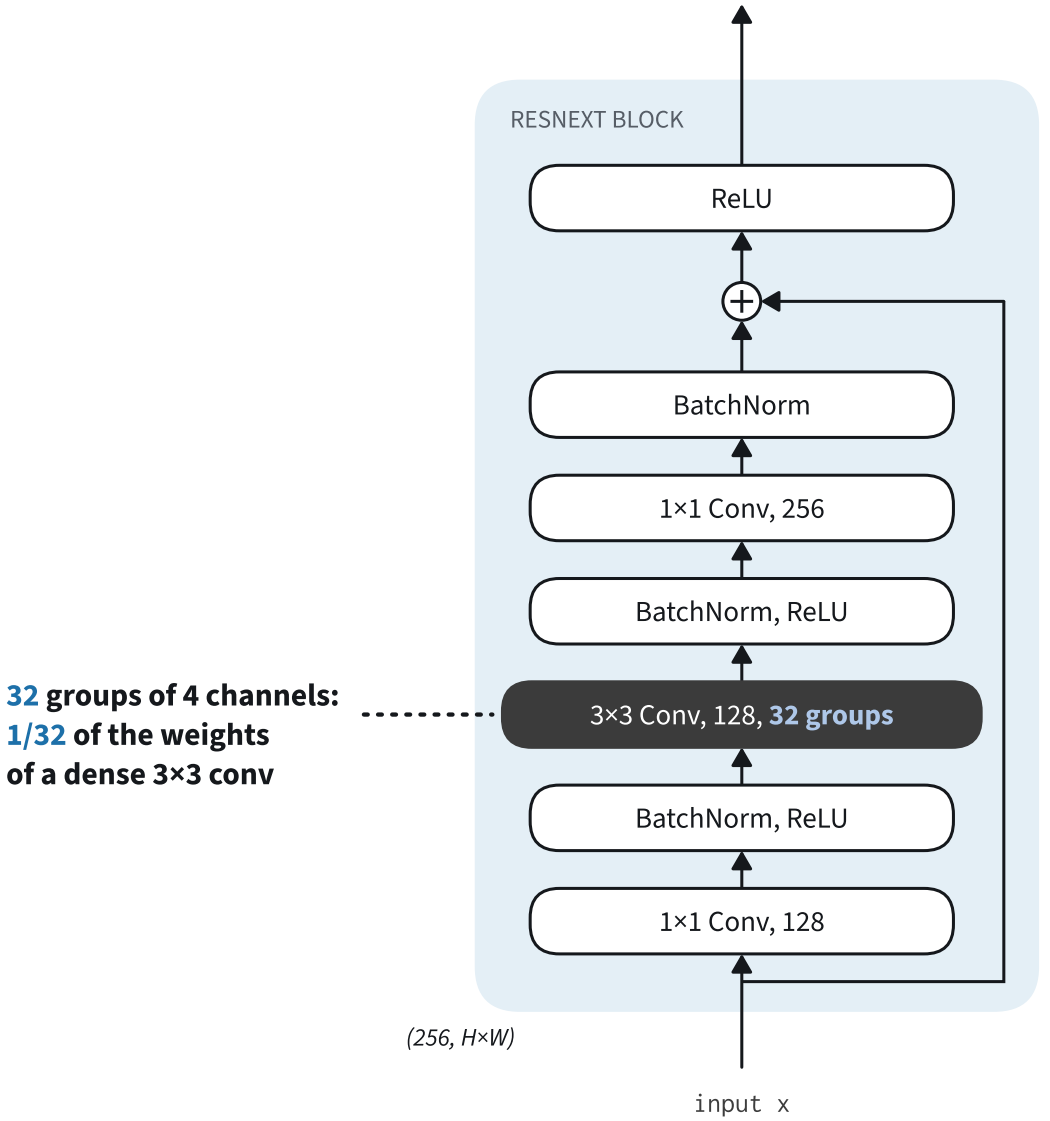


Figure 7.12: ResNeXt ブロック。3×3 のグループ畳み込み（グループ数 g）を、グループ間で情報を混合する 2 つの 1×1 畳み込みで挟む。図のブロックはチャネル数 $c = 256$、ボトルネックチャネル数 $b = 128$、4 チャネルずつのグループ数 $g = 32$ である。

```
                                        stride=strides, bias=False)
            self.bn4 = nn.LazyBatchNorm2d()
        else:
            self.conv4 = None

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return F.relu(Y + X)
```

32 チャネル、16 グループ、bot_mul=1 のブロックが入力テンソルの形状を保つことを確認しよう。

```
blk = ResNeXtBlock(32, 16, 1)
X = d2l.randn(4, 32, 96, 96)
blk(X).shape
```

```
torch.Size([4, 32, 96, 96])
```

7.4.5.2 DenseNet

残差ブロックが出力を入力に加算し（$\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$）、チャネル幅を一定に保ちながら古い特徴と新しい特徴を混ぜ合わせるのに対し、**DenseNet**（dense convolutional network、密結合畳み込みネットワーク）(Huang, Liu, et al. 2017) は各層の出力を入力に対してチャネル次元に沿って**連結**する。

$$\mathbf{x}_k = [\mathbf{x}_{k-1}, f_k(\mathbf{x}_{k-1})]\,,$$

ここで、$[\cdot, \cdot]$ はチャネル方向の連結を表す。いかなる情報も上書きされることはない。$\mathbf{x}_k$ は元のブロック入力と先行する k 個すべての層の出力をそのまま保持するため、ブロック内のすべての層がそれ以前のすべての層と直接接続される。Figure 7.13 に 2 層の密ブロック（dense block）を示す。

密ブロックの内部では、各畳み込みはプレアクティベーションの順序（バッチ正規化 → ReLU → バイアスなしの畳み込み）を採用する。この順序を ResNet に適用した場合については後述の演習問題で調べる。各層が新たに追加するチャネル数（Figure 7.13 では 32）は**成長率**（growth rate）と呼ばれる。どの密ブロックの内部でもチャネル幅が線形に増加するため、DenseNet は連続するブロックの間に**遷移層**（transition layer）を挿入し、1×1 畳み込みでチャネル数を圧縮するとともにストライド 2 の平均プーリングで高さと幅を半分にする。完全な DenseNet は遷移層を挟んで 4 つの密ブロックを積み重ねており、ResNet-18 の 4 ステージ構成と対応している（その構築と訓練の手順は 7 で扱う）。

DenseNet はより少ないパラメータ数で ResNet に匹敵する ImageNet 正解率を達成し、著者らはその要因を徹底した特徴の再利用に帰した。しかし実用上の欠点は活性化メモリ（activation memory）にある。**固有の**特徴マップの数はブロックの深さに対して線形にしか増えないにもかかわらず、素朴な実装では層ごとにより幅の広い新たな連結テンソル（およびその正規化後のコピー）を確保して誤差逆伝播法のためにキャッシュするため、逆伝播の際に中間テンソルを再計算する特殊な実装を用いない限り、メモリ使用量が深さの二**乗**に比例して増大してしまう (Pleiss et al. 2017)。加算型の残差ショートカットはこのメモリオーバーヘッドを完全に回避できる。こ

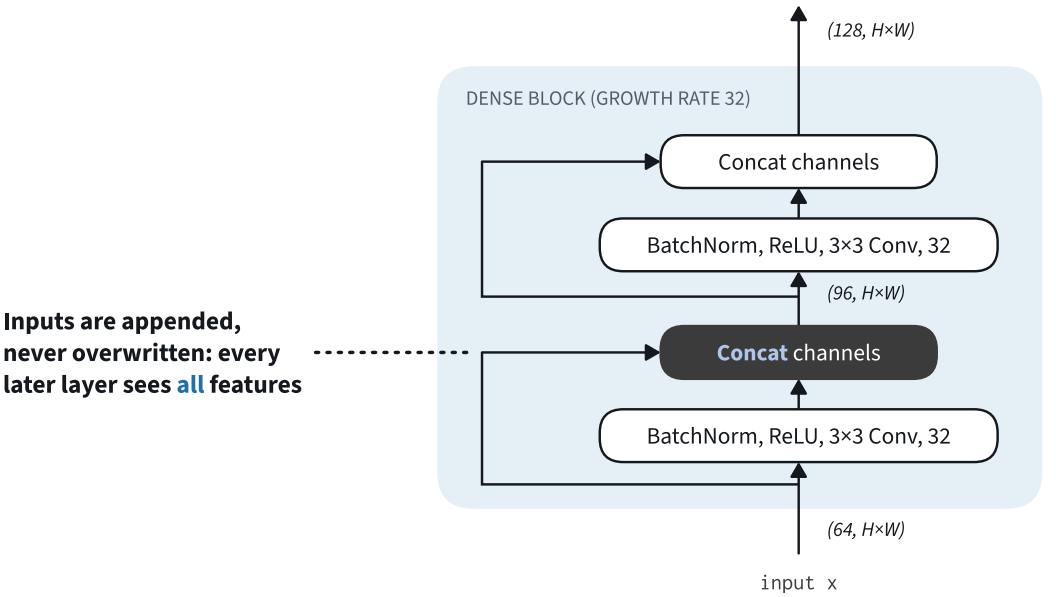


Figure 7.13: 2 層で成長率が 32 の密ブロック。各層が 32 個の出力チャネルを入力に追加するため、64 チャネルの入力は 96 チャネル、次いで 128 チャネルへと増加し、すべての層がそれ以前の全層の出力を参照する。

れが、加算型のスキップ接続（skip connection）が深層学習全般で普遍的な標準となった主要な理由の 1 つである。

それでもなお、連結型のスキップ接続は 1 つの主要なアーキテクチャ族において中心的な役割を担い続けている。それが画像セグメンテーション（image segmentation）向けの **U-Net** (Ronneberger et al. 2015) であり、画像を段階的にダウンサンプリングしたのち元の解像度までアップサンプリングし、アップサンプリングされた各特徴マップをエンコーダ経路の対応する高解像度マップと連結する。連結は層ごとではなく空間スケールごとに 1 回しか行われないため、メモリ消費は控えめに抑えられる。この同じ U-Net バックボーンはのちに現代の画像拡散モデル（diffusion model、17.4 節）の原動力ともなった。

7.4.6 まとめ

驚くべきことに、バッチ正規化を施したプレーンネットワークでは、深いアーキテクチャの関数クラスが浅いモデルの関数クラスを厳密に含んでいるにもかかわらず、層を積み増すことで浅いモデルよりも訓練損失が**増加**してしまうことがある。残差ブロックは加算的な補正 $\mathbf{x} + g(\mathbf{x})$ を学習することでこの劣化問題を解決する。これにより恒等写像が $g = 0$ に位置するようになり、深さが増してもショートカットが勾配の一貫性と損失ランドスケープの滑らかさを保つ。Fashion-MNIST では、56 層のプレーンネットワークは 20 層のモデルより訓練結果が悪化するのに対し、残差ショートカットを導入すると 56 層のモデルの方が良好に訓練される。残差ブロックをバッチ正規化とともに 4 つのステージに配置することで ResNet-18（Fashion-MNIST で 10 エポック後に検証正解率約 93%）が得られ、150 層を超える深さへも無理なくスケールする。ResNeXt はグループ畳み込みによって同等のコストでボトルネックブロックの幅を広げ、DenseNet はメモリコストの増加と引き換えに加算型ショートカットを特徴の連結へと置き換える。以来、残差接続は第 11 章の Transformer をはじめ、現代のほぼすべてのアーキテクチャに不可欠な基盤となっている。

7.4.7 演習問題

1. **Inception ブロックと残差ブロックの比較**。GoogLeNet の `arch`（7.2.3 節）において、中間ステージの 2 番目のブロックは分岐幅 $c_1 = 160$、$c_2 = (112, 224)$、$c_3 = (24, 64)$、$c_4 = 64$ をもち、512 個の入力チャネルを 512 個の出力チャネルへ写像する。これを恒等ショートカット付きの 512 チャネル `Residual` ブロックと比較せよ。
   1. 両ブロックの畳み込みの重み数と出力ピクセルあたりの積和演算回数を数えよ（バイアスとバッチ正規化のパラメータは無視してよい）。
   2. 分岐の出力がゼロである `Residual` ブロックは、非負の入力（ブロックが ReLU の直後に続く場合は常に成り立つ）に対して恒等写像を計算する。訓練モードと予測モードにおいて `bn2` が何を出力するかを `conv2` をゼロにした場合について確認し、両モードでブロックを厳密な恒等写像にする `bn2` のパラメータを述べよ。また、4 分岐の Inception ブロックが非負の入力に対して恒等写像を計算するようなパラメータの選び方は存在するか。

2. **プレアクティベーションの順序**。@{He.Zhang.Ren.ea.2016**1**} **はバッチ正規化と ReLU を各畳み込みの**前 * に移し（BN $\rightarrow$ ReLU $\rightarrow$ Conv）、加算後の ReLU を取り除くことで、各ブロックが手つかずの恒等ハイウェイを保ったまま $\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$ を計算するようにした（7.4.5.2 節の DenseNet ブロック内部と同じ順序である）。
   1. 積み重ねた K 個のプレアクティベーションブロックについて、$\mathbf{x}_K$ を $\mathbf{x}_0$ と各分岐出力を用いて表し、ヤコビ行列 $\partial \mathbf{x}_K / \partial \mathbf{x}_0$ が加算項として厳密な単位行列 $\mathbf{I}$ を含むことを示せ。
   2. 標準的なポストアクティベーション型の `Residual` ブロック（それぞれ加算後に ReLU を適用する）を K 個積み重ねた場合、$\mathbf{I}$ に代わって $\partial \mathbf{x}_K / \partial \mathbf{x}_0$ に現れる因子は何か。

3. [code] **プレーンネットワークの勾配**。●He et al. (2016a) は、バッチ正規化を施したプレーンネットワークが深さとともに劣化するにもかかわらず勾配消失を起こしていないことを指摘した。7.4.3 節の 20 層および 56 層の 4 つのネットワークでこのことを確認せよ。
   1. 初期化直後と 1 エポックの訓練後に、1 つのミニバッチ上で損失の勾配を計算し、4 つのネットワークすべてについて各畳み込み層の重み勾配のフロベニウスノルムを層のインデックスに対してプロットせよ。56 層のプレーンネットワークの勾配は入力層に向かうにつれて消失しているか。
   2. 56 層のプレーンネットワークを 5 エポックではなく 20 エポック訓練せよ。その訓練損失は 20 層のプレーンネットワークに追いつくか。

4. [code] **ResNet-34 と ResNet-50 のボトルネック**。●He et al. (2016a) の表 1 は、ステージの深さとブロックの種類によって ResNet-18 から ResNet-152 までを定義している。
   1. ResNet-34（`arch=((3, 64), (4, 128), (6, 256), (3, 512))`）を `ResNet` を用いて構築し、そのパラメータ数を `ResNet18` と比較せよ。
   2. ResNet-50 は 2 層の `Residual` ブロックを 3 層のボトルネックブロック（1×1 で幅を $4\times$ 縮小し、3×3 を経て、1×1 で幅を復元する構造であり、`ResNeXtBlock` で `groups=1, bot_mul=0.25` としたもの）に置き換え、ステージ出力幅を $4\times$ 広い（`256, 512, 1024, 2048`）にする。幅 $4c$ の ResNet-50 ボトルネックブロック 1 個のパラメータ数と積和演算回数を幅 c の ResNet-34 の `Residual` ブロック 1 個と比較し、続いて `stage` を `ResNet` のサブクラスでオーバーライドして ResNet-50 を構築し、総パラメータ数を確認せよ。

5. [code] **ResNet-D のダウンサンプリングショートカット**。ストライド 2 の `Residual` ブロックでは、ショートカットの 1×1 でストライド 2 の畳み込みは 4 つの空間位置のうち 1 つしかサンプリングせず、残りの 3 つを捨ててしまう。He et al. (2019) はこれを、2×2 でス

トライド 2 の平均プーリングに続けてストライド 1 の 1×1 畳み込みとバッチ正規化を行う構成に置き換えている。この ResNet-D ショートカットを `Residual` のサブクラスとして実装し、出力の形状とパラメータ数が変わらないことを確認したうえで、Fashion-MNIST における 10 エポック訓練後の検証正解率を複数回の実行にわたって `ResNet18` と比較せよ。

6. [code] **プレアクティベーション ResNet-18**。●He et al. (2016b) のプレアクティベーションブロックを `Residual` の変種として実装し、`stage` を `ResNet` のサブクラスでオーバーライドして積み重ね、その訓練曲線を `ResNet18` と比較せよ。

7. [code] **DenseNet**。●7.4.5.2 節の DenseNet を構築して訓練し、そのパラメータ数と活性化メモリを計算せよ。

    1. `DenseBlock`（`num_convs` 個の (BN $\rightarrow$ ReLU $\rightarrow$ $\text{Conv}_{3\times3}$ without bias) ユニットを適用し、`growth_rate` チャネルの各出力を入力にチャネル軸に沿って連結するブロック）と、遷移層 (BN $\rightarrow$ ReLU $\rightarrow$ $\text{Conv}_{1\times1}$ without bias $\rightarrow$ $\text{AvgPool}_{2\times2,s=2}$) を実装せよ。3 チャネルの入力に対し、畳み込みが 2 個で成長率 10 の密ブロックが $3 + 10 + 10 = 23$ チャネルを返し、10 チャネルの遷移層がそれを 10 チャネルに減らしつつ高さと幅を半分にすることを確認せよ。
    2. `ResNet` のステム、畳み込み 4 個・成長率 32 の 4 つの密ブロック、連続する密ブロック間でチャネル数を半分にする遷移層、および BN $\rightarrow$ ReLU に続く `ResNet` のヘッドから DenseNet を組み立てよ。これを 96×96 の Fashion-MNIST 上で訓練し、パラメータ数、検証正解率、1 エポックあたりの実行時間を `ResNet18` と比較せよ。
    3. n 個の畳み込みがそれぞれ r チャネルを出力する密ブロック（入力は c チャネル）と、n 個の `Residual` ブロック（一定の幅 c）を比較せよ。連結されたすべての入力テンソルを逆伝播のために保持する素朴な実装において、i 番目の畳み込みは何チャネルを入力として消費するか。また $i = 1, \ldots, n$ にわたるその総和はいくらか。この総和を、保持される $2nc$ チャネル（n 個の残差ブロックの場合）および $c + nr$ 個の固有チャネル（Pleiss et al. (2017) の手法による場合）と、$c = 64, r = 32, n = 6$ のもとで比較せよ。

7.5 効率的な畳み込みネットワーク

これまでに学んだアーキテクチャは、データセンターの GPU を用いて学術ベンチマークの正解率リーダーボードで首位に立つよう設計されたものであった。しかし実環境での運用では、膨大な数の畳み込みネットワークがスマートフォン、スマートカメラ、自動運転車、組み込みボード上で動作しており、そこではレイテンシ、メモリ使用量、バッテリー消費が厳格な予算制約となる。ベンチマークの正解率がどれほど高くても、予算を超過するモデルは製品として出荷できないのである。カメラからのフレームをリアルタイムに分類するスマートフォンを考えてみよう。VGG-16 は 224×224 の単一フレームに対して 153 億回の積和演算を必要とし、毎秒 30 フレームでは毎秒 4600 億回の積和演算に相当する。さらに、1 億 3800 万個の単精度パラメータは、端末上の他のあらゆるアプリと並行して 550 MB の RAM を消費する。畳み込みネットワークのコストを 1 桁削減するにはどうすればよいか、またそれによって正解率にどのようなトレードオフが生じるのだろうか。

本節では、効率的な畳み込みネットワークの設計を支える 3 つの主要なアイデアを探る。**因子分解**（factorization）は、各密な畳み込みを、軽量なチャネルごとの空間フィルタと 1×1 のチャネル混合層のペアへと分解する。**スケーリングと探索**は、強力な基本ブロックを固定したうえで、各ステージの深さ、幅、入力解像度を体系的に最適化する。**構造的再パラメータ化**（structural re-parameterization）は、最適化しやすいマルチブランチ構造を訓練し、その後それらのブランチを代数的に統合して、実運用向けの高速な単一の畳み込みへと変換する。その過程で、因子分解を正解率と実経過時間の両面から直接ベンチマークし、示唆に富む教訓を明らかにする。すな

わち、理論上の積和演算数だけでは実際の速度はほとんど語れず、因子分解が単に密なモデルの幅を狭める手法を上回るかどうかは、タスクの規模に強く依存するのである。

```
import time
```

7.5.1 デプスワイズ分離可能ネットワーク

密な空間畳み込みは畳み込みネットワークの演算の大半を占めるため、効率化への最も直接的な道筋は畳み込み層そのものを因子分解することである。

7.5.1.1 デプスワイズ分離可能畳み込み

標準的な密な畳み込みでは、すべての出力チャネルが独自の空間フィルタを通してすべての入力チャネルを読み取るため、計算量と重み数は $c_\mathrm{i}c_\mathrm{o}$ に比例して増加する。7.4.5.1 節で見た ResNeXt のグループ畳み込み（grouped convolution）は、チャネルを互いに素な g 個のグループに分割し、各出力グループを対応する入力グループのみに制限することでこの全対全の結合を断ち切り、重みと積和演算の両方を g で割った規模に削減した。Figure 7.14 に示すように、グループ化によって密なチャネル接続行列はブロック対角パターンへと変わる。

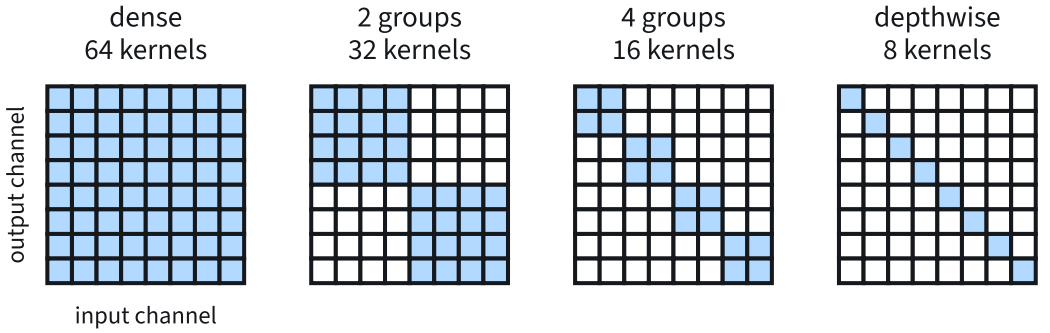


Figure 7.14: 8 個の入力チャネルと 8 個の出力チャネルにおいて、各出力チャネルが読み取る入力チャネル。網掛けの各セルは 1 つの $k_\mathrm{h} \times k_\mathrm{w}$ カーネルを表し、g 個のグループに分けると全体の $1/g$ だけが残る。

ここで、グループ化を極限である $g = c_\mathrm{i} = c_\mathrm{o}$ まで推し進め、各グループが単一のチャネルのみを含むようにしてみよう。この**デプスワイズ畳み込み**（depthwise convolution）は、チャネル間の混合を一切伴わずに入力チャネルごとに 1 つの $k \times k$ 空間フィルタを独立に適用する。これは、空間的な相互作用を一切伴わずに各ピクセルですべてのチャネルを混合する 1×1 畳み込み（6.4.3 節）とちょうど相補的な関係にある。デプスワイズな $k \times k$ 畳み込みと**ポイントワイズ**な 1×1 畳み込みを連結すれば、2 つの低コストなステップで空間フィルタリングとチャネル混合の両方を回復でき、**デプスワイズ分離可能畳み込み**（depthwise-separable convolution）が得られる (Chollet 2017; Howard et al. 2017)。Figure 7.15 に、この 2 段階の因子分解と置き換え元の密な畳み込みの対比を示す。

削減効果を定量化してみよう。出力を $h \times w$、カーネルを $k \times k$ とすると、密な畳み込みは $hw \cdot c_\mathrm{i}c_\mathrm{o}k^2$ 回の積和演算を必要とするのに対し、デプスワイズ分離可能ペアが必要とするのは空間フィルタリングの $hw \cdot c_\mathrm{i}k^2$ 回とポイントワイズ混合の $hw \cdot c_\mathrm{i}c_\mathrm{o}$ 回の和である。両者の比は次の式で与えられる。

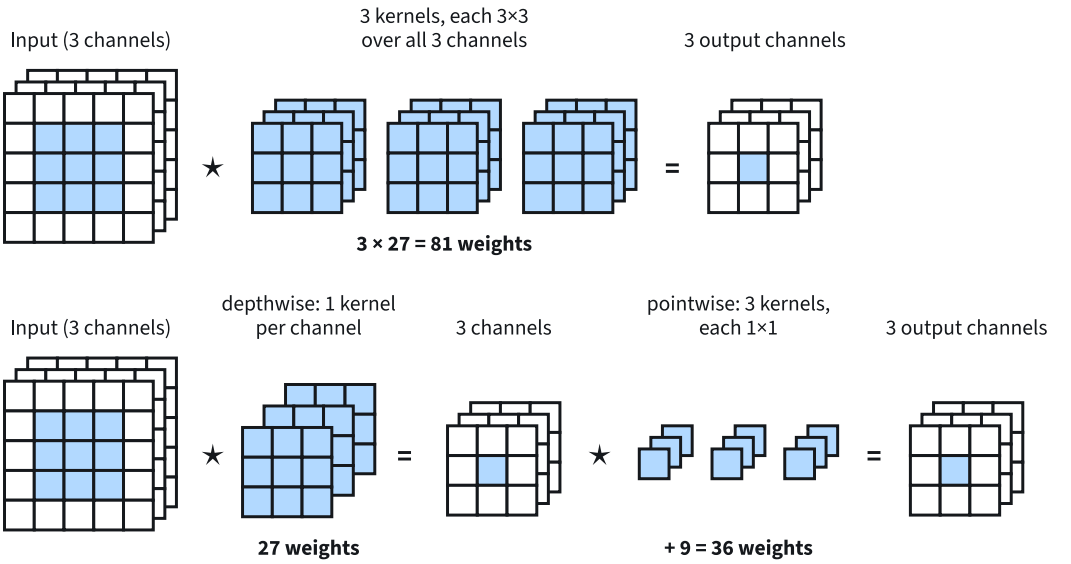


Figure 7.15: 3 個の入力チャネルと 3 個の出力チャネルに対する 3×3 カーネル。上：密な畳み込みでは、各出力チャネルが独自のカーネルスタックを通してすべての入力チャネルを読み取る。下：デプスワイズ畳み込みが全出力チャネルに共通して各チャネルを個別にフィルタリングし、続いてポイントワイズ畳み込みが出力チャネルごとに 1 つの 1×1 カーネルスタックを用いてチャネルを混合する。

$$\frac{hw \cdot c_{\mathrm{i}} k^2 + hw \cdot c_{\mathrm{i}} c_{\mathrm{o}}}{hw \cdot c_{\mathrm{i}} c_{\mathrm{o}} k^2} = \frac{1}{c_{\mathrm{o}}} + \frac{1}{k^2}, \tag{7.6}$$

パラメータ数についてもまったく同じ比が成り立つ。Figure 7.15（$c_{\mathrm{i}} = c_{\mathrm{o}} = k = 3$）では、密な層が $3 \cdot 27 = 81$ 個の重みを使うのに対し、分離可能ペアは $27 + 9 = 36$ 個で済む（比は $1/3 + 1/9 = 4/9$）。出力チャネルをもう 1 つ増やすのに必要な新しい重みは、密な層では 27 個であるのに対し、分離可能層ではわずか 3 個である。c_{o} が大きく $k = 3$ のとき、この比は $1/9$ に近づき、重みと演算量の両方でほぼ 9 分の 1 への削減となる。

むろん、この削減は無償ではなく、実効的なフィルタテンソルに対する構造的制約から生じている。分離可能ペアは単一の空間フィルタ $\mathbf{D}_c$ を入力チャネル c に適用し、フィルタリングされたマップをスカラーのポイントワイズ重み $P_{o,c}$ で線形結合するため、カーネルが $\mathbf{K}_{o,c} = P_{o,c}\,\mathbf{D}_c$ という分解形式に制約された密な畳み込みと等価になる。すべての出力チャネルが入力チャネル c 上で同じ空間フィルタを（スカラー倍を除いて）共有しなければならないのに対し、密な畳み込みは独立した任意の $k \times k$ フィルタをすべての (o, c) の組に対して学習できる。効率的なネットワークはこのランク 1 の制約を受け入れ、浮いた計算予算をより深い、あるいはより広いステージへと再投資するのである。

バイアスなしの 3×3 層で 128 チャネルを 128 チャネルに写像する場合について、パラメータ比をコードで確認してみよう。この設定では、Equation 7.6 から削減倍率は $(1/128 + 1/9)^{-1} \approx 8.4$ と予測される。

```
c_i, c_o, k = 128, 128, 3
X = d2l.normal(0, 1, (1, c_i, 32, 32))
dense = nn.LazyConv2d(c_o, kernel_size=k, padding=1, bias=False)
depthwise = nn.LazyConv2d(c_i, kernel_size=k, padding=1, groups=c_i,
                          bias=False)
pointwise = nn.LazyConv2d(c_o, kernel_size=1, bias=False)
assert dense(X).shape == pointwise(depthwise(X)).shape
p_dense = dense.weight.numel()
p_sep = depthwise.weight.numel() + pointwise.weight.numel()
p_dense, p_sep, p_dense / p_sep
```

```
(147456, 17536, 8.408759124087592)
```

密な層が 147,456 個の重みを使用するのに対し、分離可能ペアは 17,536 個であり、ちょうど 8.4× の削減となっている。

PyTorch では、`groups=c_i` を指定すると `nn.Conv2d` がデプスワイズ畳み込みになる。また、入出力両方のチャネル数を割り切る任意の `groups` を指定するとグループ畳み込みが得られる。

7.5.1.2 MobileNet

MobileNet は、デプスワイズ分離可能畳み込みをモバイル向けの完全なバックボーンへと発展させた (Howard et al. 2017)。最初の単一の密なステム畳み込みの後、すべての層で Figure 7.16 のブロック、すなわちデプスワイズ 3×3 畳み込みに続けてポイントワイズ 1×1 畳み込みを配置し、それぞれにバッチ正規化（batch normalization）と ReLU を組み合わせた構造を用いる。空間的なダウンサンプリングはストライド 2 のデプスワイズ畳み込みで直接処理し（独立したプーリング層を省く）、モデルの末尾には NiN のグローバル平均プーリング（global average pooling）と単一の線形層を置く（7.2.2 節）。ImageNet における効果は劇的であった。MobileNet は VGG-16 と 1 パーセントポイント以内の差（top-1 正解率で 70.6% 対 71.5%）に迫りながら、わずか 5 億 7000 万回の積和演算と 420 万個のパラメータ、すなわち演算量で 27× 減、パラメータ数で 33× 減を達成したのである。

7.5.1.3 逆ボトルネック

古典的な ResNet や ResNeXt のボトルネック（7.4.5.1 節）では、空間フィルタの手前でチャネルを**圧縮**していた。すなわち、1×1 層が広い表現を狭い部分空間へと絞り込むことで高コストな密な 3×3 畳み込みの計算量を抑え、2 つ目の 1×1 層で再び元の幅へと拡張していた。しかし、3×3 層が**デプスワイズ**である場合、この論理は完全に逆転する。デプスワイズ畳み込みは広い特徴マップ上でも低コストであり、しかもチャネルごとに 1 つの空間フィルタしか学習しないため、むしろ広い幅で実行する方が有利になる。

そのため、**MobileNetV2** はボトルネックを**反転**させる (Sandler et al. 2018)。残差ハイウェイは狭いまま（たとえば 24 チャネル）に保ち、ブロック内部では 1×1 の拡張によって 6 倍の 144 チャネルへと引き上げ、デプスワイズ 3×3 畳み込みがその拡張された空間をフィルタリングした後、線形の 1×1 射影で再び 24 チャネルへと圧縮する。ここで決定的に重要なのは、最終の射影で ReLU を省く点である。きわめて細いボトルネックに ReLU を適用すると負の座標が 0 に潰され、後続の層では回復できない情報が永久に失われてしまう。Figure 7.17 に標準的なボトルネックと逆ボトルネックを並べて示す。

残差ショートカットが広い内部ではなく細い両端を結ぶため、ブロック間でメモリ上に保持されるのはコンパクトなテンソルだけで済む。これは、重み用のフラッシュストレージよりも活性化用の RAM の方が逼迫しやすいモバイルプロセッサにおいて大きな利点となる。この逆ボトルネ

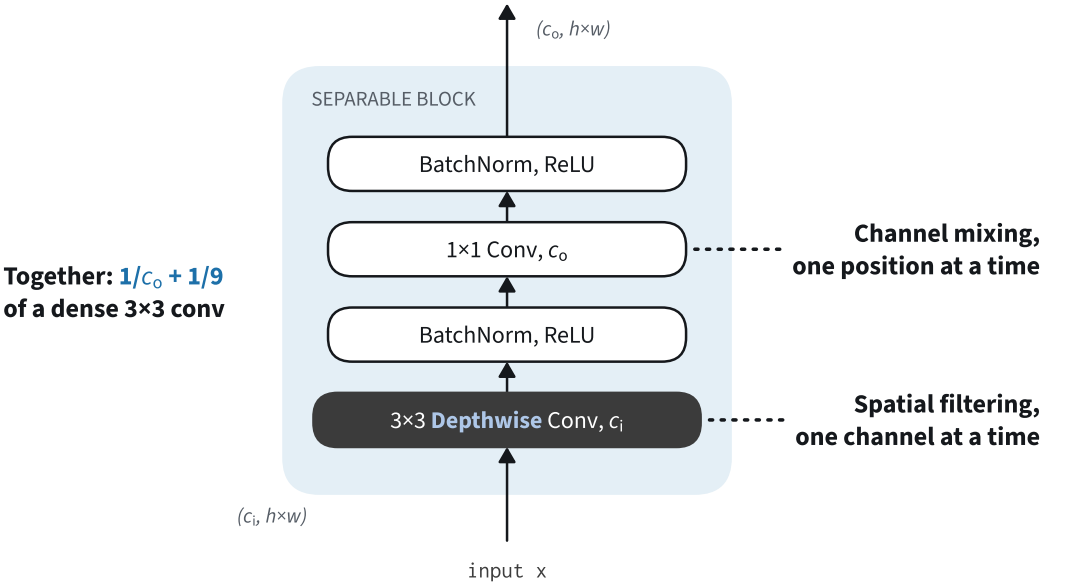


Figure 7.16: c_i チャネルから c_o チャネルへのデプスワイズ分離可能ブロック。デプスワイズ 3×3 畳み込みが各入力チャネルを個別にフィルタリングし、ポイントワイズ 1×1 畳み込みが各位置でチャネルを混合する。各畳み込みの後にはバッチ正規化と ReLU が続く。

ック（inverted bottleneck）とまったく同じ構造には、より大きな 7×7 デプスワイズカーネルを先頭に移動させた形で、ConvNeXt（7.7 節）において再び出会うことになる。

7.5.1.4 Mini-MobileNet

デプスワイズ分離可能因子分解の効果を Fashion-MNIST 上で直接確かめるため、因子分解そのものに焦点を絞れるよう Figure 7.16 のシンプルなブロックを用いてコンパクトな MobileNetV1 風ネットワークを構築する。すべての畳み込みの直後にバッチ正規化が続くため、畳み込みのバイアスはすべて省略し、畳み込み層と線形層の重みにはすべて He 初期化（4.4 節）を用いる。

```
def dws_block(in_channels, out_channels, stride=1):
    """Depthwise 3x3 and pointwise 1x1 convolutions, each with BN and ReLU."""
    return nn.Sequential(
        nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=stride,
                  padding=1, groups=in_channels, bias=False),
        nn.BatchNorm2d(in_channels), nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
        nn.BatchNorm2d(out_channels), nn.ReLU())

def he_init(module):
    """Normal weights with variance 2/n_in and zero biases."""
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

VGG（7.2.1 節）と同様に、アーキテクチャは `(out_channels, stride)` のペアのリストとして簡潔

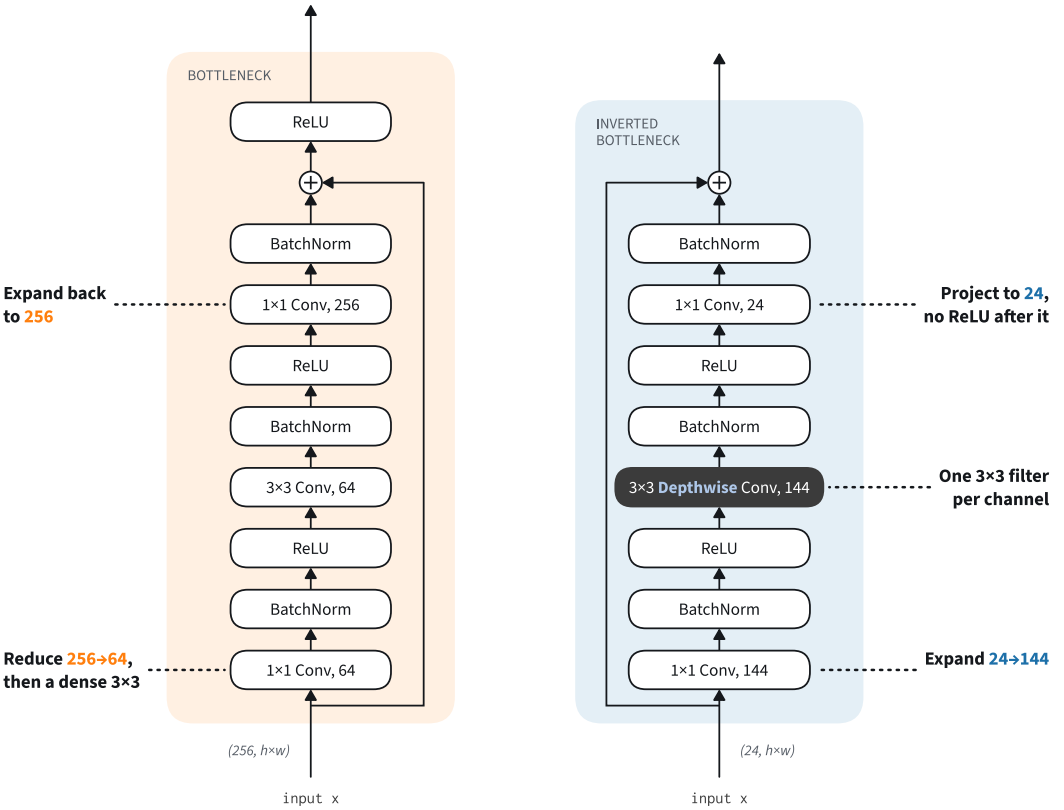


Figure 7.17: ResNet のボトルネックブロック（左）と MobileNetV2 の逆ボトルネック（右）。入力は下側にある。通常のボトルネックは密な 3×3 畳み込みのために 256 チャネルを 64 チャネルに狭めてから再び広げる。逆ボトルネックはチャネルごとに 1 つのフィルタを学習するデプスワイズ 3×3 畳み込みのために 24 チャネルを 144 チャネルに広げ、ReLU なしで 24 チャネルへ射影して戻す。残差接続は細い両端どうしを結ぶ。

に指定する。ストライド 2 の 3×3 ステムが単一チャネルの入力を半分の解像度で 32 チャネルへと引き上げ、続く 7 個のデプスワイズ分離可能ブロックがチャネル数の倍増と空間解像度の半減を 3 回繰り返し、最後にグローバル平均プーリングと線形ヘッドが 10 クラスのロジットを出力する。

ここではオプティマイザもモメンタム 0.9 付きの SGD へと強化する。これは過去の勾配の指数移動平均を蓄積することで、同じ学習率でも実効的なステップサイズを通常の SGD の最大 10 倍にまで高められる手法である（9.4 節）。このオプティマイザの変更があるため、ここでの生の正解率を 7.4 節における通常の SGD を用いた ResNet-18 の実行結果と直接比較してはならない。これまでと同様、学習率は半コサイン減衰（7.1 節）に沿って `num_steps` 回の訓練ステップ（10 エポック分で 4,690 ミニバッチ）にわたり滑らかに 0 へと減衰させるが、ここではウォームアップなしで各フレームワーク組み込みのコサインスケジューラを使用する。

```
class MiniMobileNet(d2l.Classifier):
    def __init__(self, arch=((64, 1), (128, 2), (128, 1), (256, 2), (256, 1),
                             (512, 2), (512, 1)),
                 lr=0.05, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1,
                            bias=False),
                  nn.BatchNorm2d(32), nn.ReLU()]
        c = 32
        for c_out, stride in arch:
            layers.append(dws_block(c, c_out, stride))
            c = c_out
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

`MiniMobileNet` の訓練可能なパラメータ数を数え、対応する密なネットワークと比較してみよう。なお、後の節で再利用するため `num_params` を `d2l` に保存しておく。

```
def num_params(model):
    """The number of trainable parameters."""
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

model = MiniMobileNet()
assert model(torch.zeros(1, 1, 96, 96)).shape == (1, 10)
num_params(model)
```

```
542474
```

`MiniMobileNet` は 542,474 個の訓練可能なパラメータを持つ。各分離可能ブロックを同じ幅どうしの通常の密な 3×3 畳み込みに置き換えると、7 個のブロックだけでも $9\sum_l c_{\mathrm{i},l}\,c_{\mathrm{o},l} = 4{,}663{,}296$ 個の重みが必要になる。これはこのモデルの分離可能層が持つ 530,528 個の重みの $8.8\times$ にあた

り、チャネル幅が 512 に達するにつれて漸近的な上限である $9\times$ に近づいている。

7.5.1.5 演算量・正解率・実行時間

因子分解はどれだけの正解率を引き換えにし、実際にどれだけの実測速度をもたらすのだろうか。それを確かめるため、まず `VGGSmall` を構築する。これは密な VGG 風ネットワーク（7.2.1 節）であり、`MiniMobileNet` の**パラメータ数**に合わせたものである。密な 3×3 層はそれぞれ分離可能ブロックの約 9 倍の重みを使うため、パラメータ予算を揃えると、`VGGSmall` の上限は 128 チャネルとなるのに対し、`MiniMobileNet` は 512 チャネルに達する。両モデルはバッチ正規化、He 初期化、グローバル平均プーリングヘッド、およびまったく同じ訓練スケジュールを共有する。なお、パラメータ数を揃える比較では、層の種類とチャネル幅の両方が同時に変化する点に留意しておこう。

```
class VGGSmall(d2l.Classifier):
    """A dense VGG-style network with the same parameter budget."""
    def __init__(self, arch=((2, 32), (2, 64), (2, 128), (2, 128)),
                 lr=0.05, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers, c = [], 1
        for num_convs, c_out in arch:
            for _ in range(num_convs):
                layers += [nn.Conv2d(c, c_out, kernel_size=3, padding=1,
                                     bias=False),
                           nn.BatchNorm2d(c_out), nn.ReLU()]
                c = c_out
            layers.append(nn.MaxPool2d(2))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

訓練を開始する前に、96×96 の入力に対する正確な積和演算数を集計しておこう。「same」パディングとストライド s を用いるとき、$h \times h$ の入力は一辺 $\lceil h/s \rceil$ へと写像され、$k \times k$ のグループ畳み込み（グループ数 g）は出力要素あたり $k^2 c_\mathrm{i}/g$ 回の積和演算を要する。

```
def conv_macs(h, c_in, c_out, k=3, stride=1, groups=1):
    """Output side and multiply-adds of a k x k convolution, 'same' padding."""
    h = -(-h // stride)  # ceil(h / stride)
    return h, h * h * k * k * c_in * c_out // groups

def mobilenet_macs(arch, h=96, num_classes=10):
    h, total = conv_macs(h, 1, 32, stride=2)  # The stem
    c = 32
    for c_out, stride in arch:
        h, depthwise = conv_macs(h, c, c, stride=stride, groups=c)
        h, pointwise = conv_macs(h, c, c_out, k=1)
```

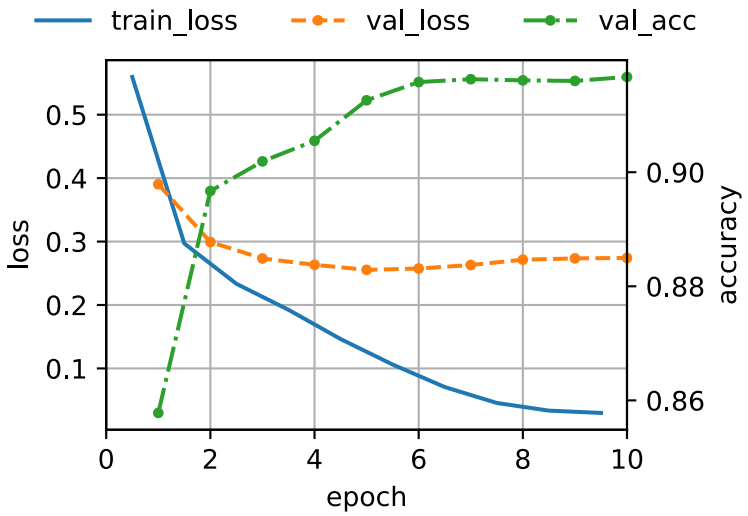
```
            total, c = total + depthwise + pointwise, c_out
        return total + c * num_classes

    def vgg_macs(arch, h=96, num_classes=10):
        total, c = 0, 1
        for num_convs, c_out in arch:
            for _ in range(num_convs):
                h, macs = conv_macs(h, c, c_out)
                total, c = total + macs, c_out
            h //= 2  # 2 x 2 max-pooling
        return total + c * num_classes

    narrow = ((2, 11), (2, 23), (2, 46), (2, 46))  # VGGSmall's widths / 2.8
    macs = {'Mini-MobileNet': mobilenet_macs(MiniMobileNet().arch),
            'VGG-style': vgg_macs(VGGSmall().arch),
            'VGG-style, narrow': vgg_macs(narrow)}
    macs, macs['VGG-style'] / macs['Mini-MobileNet']
```

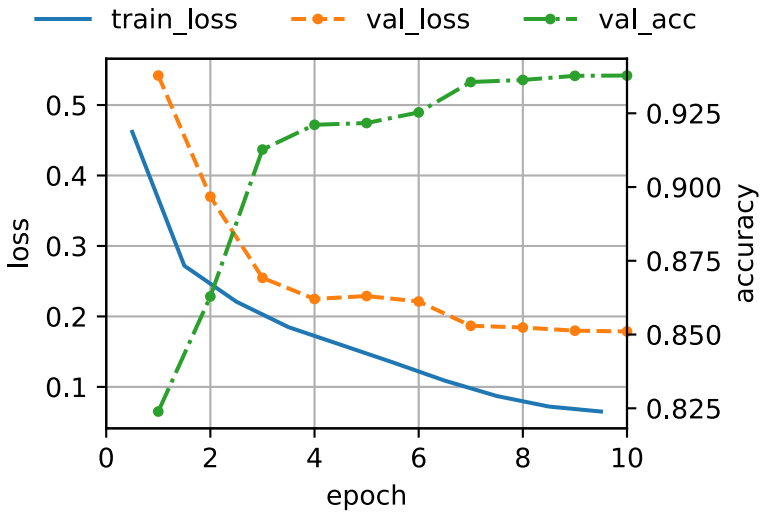
```
({'Mini-MobileNet': 50259968,
  'VGG-style': 384861440,
  'VGG-style, narrow': 49103308},
 7.657415142007253)
```

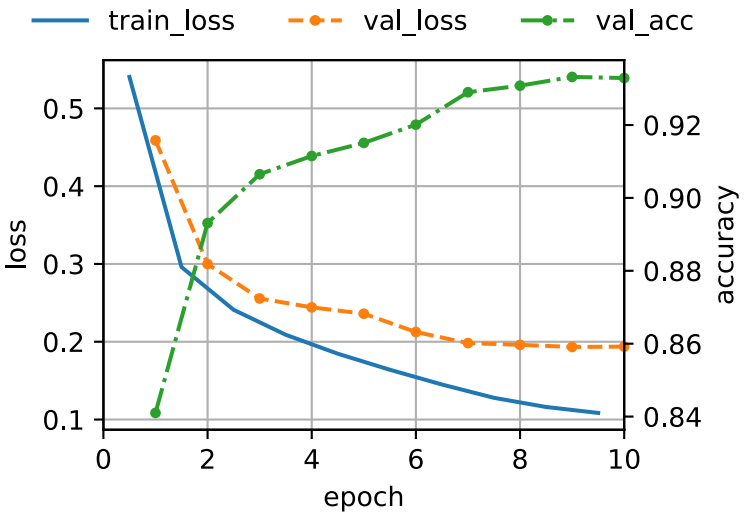
パラメータ数が揃っているにもかかわらず、画像 1 枚あたりの積和演算数は VGGSmall が 3 億 8500 万回を実行するのに対し、MiniMobileNet はわずか 5000 万回、すなわち 7.7× 少ない演算量で済む。この差は、各モデルが重みを**どこに**配置しているかを反映している。VGGSmall が 96×96 から 24×24 までの高解像度の特徴マップにわたって密なフィルタを適用するのに対し、MiniMobileNet の重みのほぼ 4 分の 3 は、小さな 6×6 のグリッド上で作用する最後の 2 つのポイントワイズ層に集中しているためである。

しかしエッジハードウェアにおいて決定的な意味を持つのは、**同一の演算量**での比較である。VGGSmall を MiniMobileNet と同じ 5000 万回の積和演算まで抑えるため、すべてのチャネル数を約 2.8 分の 1 に縮小して (11, 23, 46, 46)（上の narrow）とすると、パラメータ数はわずか 76,000 個、すなわち MiniMobileNet の重み数の 7 分の 1 になる。これら 3 つのモデルすべてを 10 エポックにわたり 96×96 の Fashion-MNIST 上で訓練してみよう。

```
    data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
    num_steps = 10 * len(data.train_dataloader())  # 10 epochs of 469 steps
    models = {'Mini-MobileNet': MiniMobileNet(lr=0.05, num_steps=num_steps),
              'VGG-style': VGGSmall(lr=0.05, num_steps=num_steps),
              'VGG-style, narrow': VGGSmall(narrow, lr=0.05, num_steps=num_steps)}
    accuracy = {}
    for name, model in models.items():
        trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
        trainer.fit(model, data)
        accuracy[name] = trainer.history['val_acc'][-1]
```


train_loss
val_loss
val_acc
loss
accuracy
epoch
0.1
0.2
0.3
0.4
0.5
0.86
0.88
0.90
0
2
4
6
8
10


train_loss
val_loss
val_acc
loss
accuracy
epoch
0.1
0.2
0.3
0.4
0.5
0.825
0.850
0.875
0.900
0.925
0
2
4
6
8
10



演算回数が測るのは算術演算の量であり、経過時間ではない。積和演算数が実際のレイテンシにどう結びつくかを調べるため、GPU 上で 128 枚の画像からなるバッチに対する推論の順伝播をベンチマークする。この際、1 回ウォームアップを実行し、実経過時間を読み取る前にデバイスを同期させる（5.7.2.1 節）。

```
def forward_ms(model, num_reps=50):
    """Milliseconds per forward pass of a batch of 128 images."""
    X = torch.randn(128, 1, 96, 96, device=d2l.try_gpu())
    model.eval()
    with torch.inference_mode():
        sync = torch.cuda.synchronize if X.is_cuda else lambda: None
        model(X)
        sync()  # Warm up and wait
        start = time.perf_counter()
        for _ in range(num_reps):
            model(X)
        sync()
    return 1000 * (time.perf_counter() - start) / num_reps

for name, model in models.items():
    print(f'{name}: {num_params(model):,} parameters, '
          f'{macs[name] / 1e6:.1f}M multiply-adds, '
          f'accuracy {accuracy[name]:.3f}, '
          f'{forward_ms(model):.1f} ms/batch')
```

```
Mini-MobileNet: 542,474 parameters, 50.3M multiply-adds, accuracy 0.917, 1.8 ms/batch
VGG-style: 583,594 parameters, 384.9M multiply-adds, accuracy 0.938, 4.9 ms/batch
VGG-style, narrow: 75,854 parameters, 49.1M multiply-adds, accuracy 0.933, 1.3 ms/batch
```

Table 7.3 に、複数回の実行にわたる 3 つのモデルすべての結果をまとめる。検証正解率とあわせて、MiniMobileNet を基準に正規化した各モデルの順伝播レイテンシを示している。

Table 7.3: パラメータ数をほぼ揃えた密な VGG 風ネットワークおよび積和演算数をほぼ揃えた狭幅の VGG 風ネットワークと Mini-MobileNet の比較（96×96 の Fashion-MNIST、10 エポック。複数回の実行にわたる検証正解率と順伝播時間の比）。

モデル	パラメータ数	積和演算数	検証正解率	順伝播時間
Mini-MobileNet	542,474	5030 万	91〜92%	1
VGG 風	583,594	3 億 8490 万	93.5〜94%	2〜3.5
VGG 風（狭幅）	75,854	4910 万	93%	0.5〜0.8

Table 7.3 からは 3 つの教訓が浮かび上がる。

1. **パラメータ数を揃えても計算量は揃わない：**約 55 万パラメータにおいて、`VGGSmall` は `MiniMobileNet` を約 2 パーセントポイント上回るが、同時に $7.7\times$ 多い積和演算を実行している（最初のステージで 96×96 の完全な解像度を保ち、制約のない密なカーネルを学習するためである）。
2. **小規模なデータセットで演算量を揃えた場合、因子分解より幅の縮小が勝る：**`VGGSmall` を 5000 万回の積和演算まで縮小したモデル（`VGG-style, narrow`、約 7.6 万パラメータ）は、`MiniMobileNet` を約 1 ポイント上回り、**しかも**高速に動作する。Equation 7.6 によれば、デプスワイズ因子分解が最も効果を発揮するのはチャネル幅が広い領域である（削減倍率は $5\times$ にとどまる $c = 11$ に対し、ほぼ $9\times$ に達する $c = 512$ で大きく開く）。10 クラスの Fashion-MNIST は幅の狭い密な層でも十分に表現できるのに対し、1,000 クラスの ImageNet では広いステージが不可欠であり、密な畳み込みネットワークの幅を狭めると表現容量が不足してしまう。
3. **積和演算数は GPU の実測レイテンシの指標としては不十分である：**演算量を $7.7\times$ 削減しても速度向上が 2–3.5× にとどまるのは、密な 3×3 層が入力の読み出し 1 回あたり $9c_\text{o}$ 回の積和演算を実行する（演算密度（arithmetic intensity）が高い）のに対し、デプスワイズ 3×3 層は入力の読み出し 1 回あたりわずか 9 回の積和演算しか実行せず、GPU がメモリ帯域幅律速（memory-bandwidth bound）となるからである。

7.5.2 スケーリングと探索

MobileNet は効率的な基本ブロックを固定し、各ステージの幅を人手で選択している。ここから自然な疑問が 2 つ生じる。利用可能な計算資源が増えたとき、ベースとなるアーキテクチャをどのようにスケールアップすべきだろうか。また、自動化された最適化によって、人間の試行錯誤よりも優れたステージの幅・深さ・解像度を選択できるのだろうか。

7.5.2.1 複合スケーリング

計算量を数えてみれば、スケーリングの問題はただちに明確になる。畳み込みネットワークの演算量は、深さに対して線形に、チャネル幅に対して 2 次（$c_\text{i}c_\text{o}$ 経由）に、入力解像度に対しても 2 次（hw 経由）に増加する。どれか 1 つの次元だけを伸ばしても、すぐに収穫逓減に突き当たる。極小のサムネイル画像に対して非常に深いモデルを適用しても分析すべき細部がそもそも欠けており、逆に高解像度の画像では受容野が大きな物体全体を覆えるようにより多くの層が必要になるからである。**複合スケーリング**（compound scaling）は、これら 3 つの次元すべてを歩調を揃えて拡大する (Tan and Le 2019)。各スケーリングステップで深さを 1.2 倍、幅を 1.1 倍、解像度を 1.15 倍にすると、総計算量はおよそ 2 倍になる（$1.2 \cdot 1.1^2 \cdot 1.15^2 \approx 1.92$）。これにより、単一の整数のつまみだけでモデル族全体を生成できるようになる。たとえば 3 ステップ進めると、深さが $1.7\times$、幅が $1.3\times$ になり、$1.5\times$ 大きい画像を入力として計算量が約 $7\times$ となるネットワークが得られる。

7.5.2.2 アーキテクチャ探索

ベースとなるネットワーク自体の設計には、EfficientNet は逆ボトルネックブロックの積層を対象とした**ニューラルアーキテクチャ探索**（neural architecture search、NAS）を用い、正解率と積和演算数のバランスをとる目的関数を最適化した。しかし、先ほどの GPU 実行時間ベンチマークが示したとおり、理論上の積和演算数を最適化すると実ハードウェアでは誤った結論を招きかねない。**EfficientNetV2** は探索の目的関数において積和演算数を実測の訓練速度とパラメータ数に置き換えた (Tan and Le 2021)。注目すべきことに、探索の結果、初期の高解像度ステージにあるデプスワイズブロックは、演算量が多くともアクセラレータ上で高速に動作する標準的な密な 3×3 畳み込みへと自動的に置き換えられた。いずれの世代でも、中核となる基本演算は人間が設計し、各ステージへの幅・深さ・解像度の配分を自動探索が担っている。

7.5.2.3 Squeeze-and-Excitation ゲート

EfficientNet が探索した逆ボトルネックブロックには、デプスワイズ畳み込みの直後にもう 1 つ軽量なモジュールとして **Squeeze-and-Excitation**（SE）ゲートが含まれている (Hu et al. 2018)。畳み込みが局所的な空間ウィンドウ内でのみ情報を集約するのに対し、SE ゲートは大域的な画像コンテキストを用いて特徴チャネル全体の重みを動的に付け直すことができる。具体的には、グローバル平均プーリングによって c 個の各チャネルを単一のスカラーへと**圧縮**（squeeze）し、その c 次元の要約を小さな 2 層のボトルネック MLP（ReLU を伴う隠れ層の幅 c/r、シグモイドを伴う出力幅 c による**励起**（excitation））に通して、特徴マップの各チャネルを $(0, 1)$ 内の予測ゲート値でスケーリングする。縮小比を r とすると、この MLP が追加するパラメータは約 $2c^2/r$ 個にとどまり、全ピクセルではなく画像あたり単一のベクトルに対して作用するため、演算コストは無視できるほど小さい。概念的には、SE はチャネルごとの単純なアテンション機構として機能しており、第 10 章で扱う一般的なアテンション層の先駆けとなっている。

SE ゲートはほぼゼロの計算コストで一貫した正解率の向上をもたらすため、MobileNetV3 や RegNetY（7.8 節、その演習問題 3 では RegNet に SE を追加する）の随所で採用されている。SE ブロックの実装はほんの数行で済み、グローバルプーリング、2 つの線形層、およびチャネルごとの再スケーリングからなる。

```
class SE(nn.Module):
    def __init__(self, num_channels, ratio=4):
        super().__init__()
        self.fc = nn.Sequential(
            nn.LazyLinear(num_channels // ratio), nn.ReLU(),
            nn.LazyLinear(num_channels), nn.Sigmoid())

    def forward(self, X):
        s = self.fc(X.mean(dim=(2, 3)))
        return X * s[:, :, None, None]

SE(32)(d2l.randn(2, 32, 16, 16)).shape
```

```
torch.Size([2, 32, 16, 16])
```

$c = 32$、$r = 4$ のとき、SE モジュールが追加するのはわずか $2 \cdot 32^2/4 = 512$ 個の重みと 40 個のバイアスだけであり、テンソルの形状を完全に保つため、任意のブロックにそのまま組み込むことができる。

7.5.3 構造的再パラメータ化

7.5.3.1 RepVGG ブロック

ResNet（7.4 節）は、ショートカット経路を持つマルチブランチブロックが、単純な直列積層よりも劇的に最適化しやすいことを確立した。しかし推論時には、並列ブランチはシステム上の実質的なオーバーヘッドを伴う。各ブランチを個別に起動し、加算地点までメモリ上に保持し続けなければならず、ハードウェアの利用効率が低下するためである。対照的に、3×3 畳み込みの単純な直列積層はアクセラレータにとって理想的であり、層ごとに高度に最適化された密なカーネル 1 つと、保持すべき単一の活性化バッファだけで済む。**RepVGG** は、訓練時のトポロジーと推論時のトポロジーを切り離すことで両者の長所を兼ね備える (X. Ding et al. 2021)。訓練時には、すべての層が ReLU の手前で 3 つの並列ブランチ、すなわち 3×3 畳み込み、1×1 畳み込み、恒等経路（それぞれ独自のバッチ正規化を伴う）の和をとる（Figure 7.18）。訓練が完了すると、各ブランチは入力に関して純粋に線形となるため、3 つすべてを代数的に**単一の 3×3 畳み込みへと融合**し、まったく同じ関数を計算できる。つまり、訓練時にはマルチブランチの残差ネットワークを用い、実運用時には高速な単純直列の VGG スタックを配備するのである。

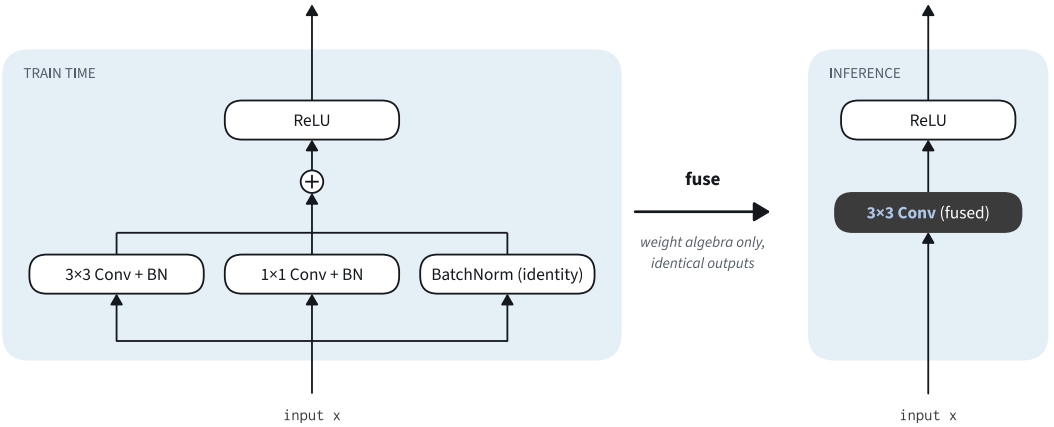


Figure 7.18: 構造的再パラメータ化。訓練時には、RepVGG の各層はそれぞれバッチ正規化を伴う 3×3 畳み込み、1×1 畳み込み、恒等写像の 3 つのブランチの和をとる。推論時には、これらのブランチはバイアス付きの 1 つの 3×3 畳み込みへと折り込まれ、2 つのネットワークは同一の関数を計算する。

```
class RepVGGBlock(nn.Module):
    """Train-time block: 3x3, 1x1, and identity branches, each with BN."""
    def __init__(self, num_channels):
        super().__init__()
        self.conv3 = nn.Conv2d(num_channels, num_channels, kernel_size=3,
                               padding=1, bias=False)
        self.conv1 = nn.Conv2d(num_channels, num_channels, kernel_size=1,
                               bias=False)
        self.bn3, self.bn1, self.bn0 = (nn.BatchNorm2d(num_channels)
                                        for _ in range(3))

    def forward(self, X):
        return F.relu(self.bn3(self.conv3(X)) + self.bn1(self.conv1(X))
                      + self.bn0(X))
```

7.5.3.2 ブランチの融合

3 つのブランチすべてを 1 つの畳み込みに集約するには、線形代数の単純な 3 つのステップを踏めばよい。第 1 に、推論モードにおけるバッチ正規化は、移動平均 μ_o を引き、$\sqrt{\sigma_o^2+\epsilon}$ で割り、γ_o を掛けて β_o を足すという、チャネルごとの固定されたアフィン写像である。このアフィン写像を直前の畳み込みのカーネル $\mathbf{W}$ とバイアス b_o（出力チャネル o で添字付けされる）に直接吸収させると、等価な融合後のカーネルとバイアスが次の式で得られる。

$$\begin{aligned}\hat{\mathbf{W}}_o &= \frac{\gamma_o}{\sqrt{\sigma_o^2+\epsilon}}\mathbf{W}_o,\\ \hat{b}_o &= \beta_o + \frac{\gamma_o(b_o-\mu_o)}{\sqrt{\sigma_o^2+\epsilon}}.\end{aligned} \tag{7.7}$$

各出力チャネルのカーネルは正規化係数によってスケーリングされるだけであり、融合後のバイアスが入力に依存しないオフセットをすべて吸収する（たとえば、$\gamma_o=2$、$\sigma_o^2+\epsilon=16$、$\mu_o=1$、$\beta_o=1$、$b_o=0$ のとき、$\hat{\mathbf{W}}_o=\mathbf{W}_o/2$ および $\hat{b}_o=1-0.5=0.5$ となる）。`RepVGGBlock` の畳み込みはバイアスを持たないため、以下の `fuse_bn` は $b_o=0$ の場合を実装している。なお、BN の折り込み（BN folding）は完全に一般的な手法であり、推論コンパイラによって任意の `Conv + BN` ペアに日常的に適用されている。

第 2 に、残りの 2 つのブランチを 3×3 畳み込みとして表す。1×1 カーネルは中心の周囲を 1 ピクセル分ゼロパディングした 3×3 カーネルにほかならず、恒等写像は 3×3 畳み込みとして、カーネル $\mathbf{W}^{\text{id}}_{o,i}$ が $i=o$ の中心タップで 1、それ以外のすべての位置で 0 となるものに一致する。第 3 に、畳み込みの線形性により、同一の入力に対する 3 つの並列な畳み込みの出力を足し合わせることは、折り込み後の 3 つのカーネルの和をカーネルとし、折り込み後の 3 つのバイアスの和をバイアスとする単一の畳み込みと完全に一致する（Figure 7.19）。

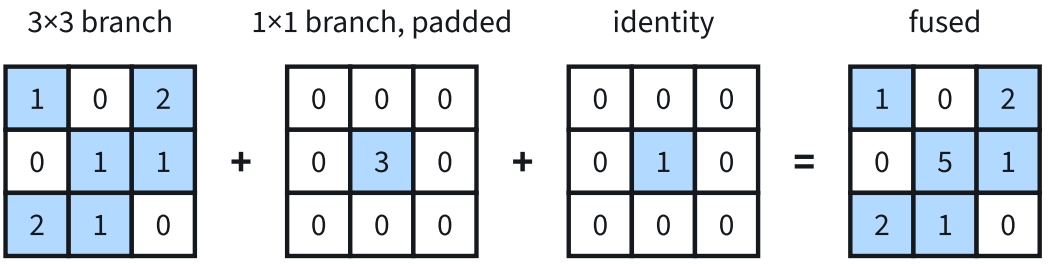


Figure 7.19: バッチ正規化を各ブランチに折り込んだ後の、1 つの出力チャネル o が入力チャネル $i=o$ に適用するカーネル（数値は例示用。折り込み後、恒等ブランチの中心成分は 1 ではなく $\gamma_o/\sqrt{\sigma_o^2+\epsilon}$ になる）。パディングされた 1×1 カーネルと恒等カーネルは中心のみで非ゼロとなるため、融合後のカーネルは中心のみで 3×3 ブランチと異なる。$i\neq o$ では恒等カーネルは 0 である。

```
def fuse_bn(W, bn):
    """Fold inference-mode batch normalization into a bias-free conv."""
    scale = bn.weight / torch.sqrt(bn.running_var + bn.eps)
    return W * scale.reshape(-1, 1, 1, 1), bn.bias - bn.running_mean * scale

@torch.no_grad()
```

```
def fuse_block(blk):
    c = blk.conv3.weight.shape[0]
    W3, b3 = fuse_bn(blk.conv3.weight, blk.bn3)
    W1, b1 = fuse_bn(F.pad(blk.conv1.weight, (1, 1, 1, 1)), blk.bn1)
    I = torch.zeros_like(W3)  # On the device of the block
    I[torch.arange(c), torch.arange(c), 1, 1] = 1.0
    W0, b0 = fuse_bn(I, blk.bn0)
    fused = nn.Conv2d(c, c, kernel_size=3, padding=1, device=W3.device)
    fused.weight.copy_(W3 + W1 + W0)
    fused.bias.copy_(b3 + b1 + b0)
    return fused
```

融合は推論モードの移動統計量に依存するため、fuse_block を厳密にテストする。まず、いくつかの訓練バッチ (平均 1、標準偏差 2) を RepVGGBlock に通して移動統計量を 0 と 1 から変化させ、γ_o と β_o をランダム化して Equation 7.7 のすべての項が働くようにしたうえで、新たなデータに対して推論モードのマルチブランチブロックと単一の融合済み畳み込みの最大絶対誤差を比較する。

```
blk = RepVGGBlock(8)
for _ in range(4):  # Move the BN running statistics
    blk(2 * torch.randn(32, 8, 16, 16) + 1)
with torch.no_grad():  # Nontrivial scales and shifts
    for bn in (blk.bn3, blk.bn1, blk.bn0):
        bn.weight.uniform_(0.5, 1.5)
        bn.bias.normal_()
blk.eval()
fused = fuse_block(blk)
X = 2 * torch.randn(32, 8, 16, 16) + 1
with torch.no_grad():
    error = (blk(X) - F.relu(fused(X))).abs().max()
float(error)
```

```
5.7220458984375e-06
```

最大誤差は 10^{-6} のオーダー（純粋な単精度浮動小数点の丸め誤差）であり、関数として厳密に等価であることが確認できる。重み数の削減は 10 分の 1（$10c^2$ から $9c^2$）にとどまるものの、実行時の処理は劇的に単純化される。2 つの畳み込み、3 つの正規化、2 つの加算、および同時に保持される 3 つの特徴バッファが、1 つの密な畳み込みと 1 つの保持される活性化テンソルへと置き換わるからである。

7.5.3.3 融合後の量子化

実環境への配備では、厳密な浮動小数点の代数だけでは見落とされる繊細な注意点が 1 つ生じる。消費エネルギーとメモリ帯域幅を節約するため、エッジアクセラレータでは重みと活性化値を 8 ビット整数へと量子化（quantization）し、各テンソルのダイナミックレンジを等間隔なわずか 256 段階へと写像することが広く行われている。RepVGG の融合カーネルは独立にバッチ正規化された 3 つのブランチの和をとり、折り込み後の各スケールが何桁も異なりうるため、融合後の重みや活性化値は裾の重いダイナミックレンジを持ちやすく、256 個の離散ビンではうまく表現できなくなる。その影響は壊滅的になりうる。単純な訓練後の 8 ビット量子化を適用すると、融合済み RepVGG-B0 の ImageNet top-1 正解率は 75.1% から 40.2% へと急落するのに対し、標準的な ResNet の低下は 1 ポイント未満にとどまる (Chu et al. 2024)。訓練中に各ブランチのスケールを正則化すればこの問題は解消されるが、ここには重要な工学的教訓がある。すなわち、

FP32 における厳密な代数的等価性は、低精度量子化のもとでの等価性を保証しないのである。

7.5.4 まとめ

効率的な畳み込みネットワークでは、正解率と並んでレイテンシ、メモリ、消費エネルギーが最優先の設計目標へと引き上げられる。デプスワイズ分離可能畳み込みは空間フィルタリングとチャネル混合を分離することで層のコストを密な畳み込みの $1/c_{\mathrm{o}} + 1/k^2$ に削減し、MobileNetV2 の逆ボトルネックは残差ハイウェイを細く保ってピーク時の活性化メモリを最小化する。Fashion-MNIST 上では、`MiniMobileNet` はパラメータ数を揃えた密なネットワークに対して積和演算数が 7.7× 少ない一方で正解率は約 2 ポイント劣り、演算密度の低いデプスワイズ層がメモリ帯域幅のボトルネックとなるため実行速度の向上も 2–3.5× にとどまる。さらにこの小規模なタスクでは、密なネットワークの幅を同じ演算予算まで狭めた方が、正解率と速度の両面で `MiniMobileNet` を上回る。層の因子分解にとどまらず、複合スケーリングとレイテンシ考慮型アーキテクチャ探索は各ステージの深さ・幅・解像度を配分し、Squeeze-and-Excitation ゲートは無視できるほどの演算コストで大域的なチャネル重み付けを追加し、構造的再パラメータ化は訓練時のマルチブランチブロックを推論時に高速な単純畳み込みへと折り込む（ただし融合後の量子化には注意を要する）。人手による設計と単一モデルの探索による効率化を見てきたところで、次節以降では訓練レシピ、ResNet の体系的な現代化、および集団レベルのアーキテクチャ設計空間へと進む。

7.5.5 演習問題

1. **グループ間のチャネル置換（ShuffleNet）**。●グループ畳み込みは、密な 1×1 重みを用いずとも混合できる (X. Zhang et al. 2018)。$k \times k$ のグループ畳み込み（$c_{\mathrm{i}} \to c_{\mathrm{o}}$ チャネル、g グループ）の後に、c_{o} 個の中間チャネルに対する固定の置換と、2 層目のグループ畳み込み（$c_{\mathrm{o}} \to c_{\mathrm{o}}$、$g$ グループ）を続けるとする。g と c_{o} について、2 層目のすべての出力チャネルがある置換のもとで全 c_{i} 個の入力チャネルに依存できるようになる条件を求め、その置換を具体的に構成せよ。

2. [code] **幅乗数**。すべてのステージのチャネル数を m 倍にスケールする幅乗数 m を `MiniMobileNet` に追加せよ (Howard et al. 2017)。パラメータ数と積和演算数が m に対してどのようにスケールするかを予測し、$m = 0.5$ において `num_params` と `mobilenet_macs` を用いてその予測を検証せよ。

3. [code] **バイアス付きバッチ正規化の折り込み**。@eq-bn-fold を用いて `fuse_bn` を非ゼロのバイアス b_o を持つ直前の畳み込みへと拡張し、推論モードの `Conv + BN` ペアが折り込み後の単一の畳み込みと 10^{-6} 以内で数値的に一致することを検証せよ。バッチ正規化が**訓練**モードで動作しているとき、いかなる固定のカーネルとバイアスによっても `Conv + BN` の出力を再現できないのはなぜか。

4. [code] **MobileNetV2 の逆ボトルネック**。●MobileNetV2 (Sandler et al. 2018) は Figure 7.16 のブロックを逆ボトルネック、すなわち 1×1 による倍率 $t = 6$ の拡張、デプスワイズ 3×3 畳み込み、**線形**の 1×1 射影（ReLU なし）、および入出力の形状が一致する場合の残差ショートカットに置き換えている。

    1. $t = 6$ の逆ボトルネックブロックを実装して `MiniMobileNet` の各ブロックを置き換え、総パラメータ数が 542,474 付近に保たれるよう各ステージの幅を選べ。Table 7.3 の設定のもとで訓練し、検証正解率と順伝播レイテンシを `MiniMobileNet` と比較せよ。
    2. 最終の 1×1 射影の後に ReLU を追加し、線形版と ReLU 版の両方をそれぞれ 3 回ずつ訓練して、この規模でも線形射影が ReLU 付き射影を上回るかどうかを確認せよ。
    3. 96×96 の入力に対し、各ブロックの入力テンソル・出力テンソル・拡張された内部テンソルの要素数を表にまとめよ。拡張とデプスワイズフィルタリングをチャネルご

とに融合またはタイリングする推論エンジンでは、拡張されたテンソル全体を RAM 上に実体化する必要が一切ない理由を説明せよ。

5. [code] **演算密度とバッチサイズ。**●`MiniMobileNet` の積和演算数は $7.7\times$ も `VGGSmall` より少ないにもかかわらず、Table 7.3 における GPU 順伝播は 2–$3.5\times$ しか高速にならない。
   1. ある層の演算密度を、その積和演算数を読み書きするテンソル要素の総数（重み + 入力活性化値 + 出力活性化値）で割った値として定義せよ。デプスワイズ畳み込み、ポイントワイズ畳み込み、および等価な密な 3×3 畳み込みの演算密度を `MiniMobileNet` のすべてのステージについて計算せよ。デプスワイズ畳み込みの演算密度が最も低くなるのはどのステージか。また、そのことから EfficientNetV2 (Tan and Le 2021) が初期のデプスワイズステージを密な畳み込みに置き換える理由をどのように説明できるか。
   2. `forward_ms` を `MiniMobileNet` と `VGGSmall` についてバッチサイズ $B \in \{1, 16, 128, 512\}$ にわたり計測し、画像 1 枚あたりのレイテンシを B に対してプロットせよ。両者の速度比は B とともにどのように変化するか、またそれはなぜか。

6. [code] **RepVGG ブロックの訓練と量子化。**●推論時に単一の 3×3 畳み込みへと融合されるにもかかわらず、なぜ 3 ブランチの `RepVGGBlock` を訓練するのだろうか。
   1. チャネル幅を変更するストライド 2 のダウンサンプリングブロックに対応できるよう `RepVGGBlock` と `fuse_block` を拡張せよ（入出力の形状が異なる場合は 3×3 ブランチと 1×1 ブランチのみを残す (X. Ding et al. 2021)）。`RepVGGBlock` から小規模な畳み込みネットワークを構築するとともに、訓練中も単一の 3×3 ブランチのみを残す対応する単純な畳み込みネットワークを構築せよ。両者を Fashion-MNIST 上で訓練して収束を比較し、訓練済みの RepVGG ネットワークを融合しても検証正解率が変化しないことを確認せよ。
   2. 単純なネットワークのバッチ正規化を `fuse_bn` で折り込んだうえで、配備された両ネットワークの重みをテンソルごとの対称スケーリング（$\hat{w} = s \cdot \mathrm{round}(w/s)$、ただし $s = \max|w|/127$）を用いて 8 ビット整数へと量子化せよ。融合済み RepVGG ネットワークの正解率低下を単純なネットワークと比較し、Chu et al. (2024) の重み分布解析と関連づけて論じよ。

7.6 畳み込みネットワークの現代的な訓練レシピ

本章全体を通して、正解率の差をアーキテクチャの違いに帰属できるよう、各節の中で訓練手順を一定に保つよう努めてきた。そのような統制された条件下でさえ、ピーク学習率を変更しただけで VGG の正解率は 1.5 パーセントポイント変化した（7.2 節）。公開されているベンチマークは、これよりはるかに統制されていないことが多い。新たに提案されたネットワークは最新の最適化や正則化の工夫を用いて訓練され、何年も前に書かれた論文から正解率を転記した古典的なベースラインと比較されるのである。報告される飛躍的な性能向上のうち、アーキテクチャそのものではなく**訓練レシピ**（training recipe：オプティマイザ、学習率スケジュール、データ拡張、正則化、訓練期間）に由来する部分は、実際にはどれほどあるのだろうか。

7.4 節で構築した ResNet-18 をさらに深くした兄弟モデルである ResNet-50 が、驚くべき答えを与えてくれる。2015 年のアーキテクチャの層を 1 つも変更することなく、Wightman et al. (2021) は 2020 年代初頭の訓練手法を用いて ResNet-50 を再訓練し、ImageNet の top-1 正解率を 76.1% から 80.4% へと引き上げた。純粋に訓練レシピだけで 4.3 ポイントもの向上を達成したのである。この伸びは、その間に発表された数多くの新規アーキテクチャが主張する改善幅に匹敵するか、それを上回る。したがって、時代遅れのレシピで訓練されたベースラインに対して 2 ポイントの優位を主張する論文があったとしても、その功績がアーキテクチャにあるのかレシピにあるのかは判別できない。

本節では、2015 年から 2022 年の間に何が変わったのかを調べ、中核となる各構成要素を数行のコードで実装したうえで、同一の ResNet-18 を古典的なレシピと現代的なレシピの両方で訓練してその違いを直接観察する。これらの手法はいずれも畳み込みに固有のものではない。まったく同じツール群が現代の Transformer の訓練を支えており、現代の深層学習の実践における汎用的な基盤となっている。

```
import random
```

7.6.1 2015 年から 2022 年にかけての訓練レシピ

2015 年のオリジナルの ResNet レシピは、その時点ですでに入念に調整されていた。モメンタム付き SGD（7.5 節）、重み減衰、段階的な学習率減衰（学習率を 2 回にわたり 10 分の 1 にする）、そしてランダムクロップと水平反転（horizontal flip）である (He et al. 2016a)。その後の数年間にわたり、研究者たちはこのパイプラインのあらゆる要素を、まずは個別に、続いて大規模かつ体系的な組み合わせによって見直していった (He et al. 2019; Wightman et al. 2021)。Table 7.4 に、2015 年のベースラインと 2020 年代初頭の代表的なレシピの対比を示す。直後の節では、それぞれの新しい構成要素を紐解きながら実装していく。

Table 7.4: 2015 年の ImageNet レシピと、2020 年代初頭の代表的な高正解率レシピ。右の列は万能の標準規格ではなく、強力なレシピは依然としてモデルや計算予算によって異なる。

構成要素	2015 年のレシピ	2022 年のレシピ
オプティマイザ	モメンタム付き SGD	AdamW または LAMB（調整された SGD も依然として有力）
スケジュール	ステップ減衰（10 分の 1 を 2 回）	線形ウォームアップ付きコサイン減衰
データ拡張	ランダムクロップ、水平反転	RandAugment、Mixup、CutMix、ランダム消去
正則化	重み減衰	+ ラベル平滑化、確率的深さ
訓練期間	90 エポック	300〜600 エポック
評価時の重み	最終時点の重み	重みの指数移動平均

Table 7.4 を読む際には、2 つの重要な点がある。第 1 に、表の各行は独立に働くのではなく密接に相互作用する。データ拡張と正則化を強めると訓練目的関数への当てはめが難しくなり、その分だけはるかに多くの訓練エポックが必要になる。訓練スケジュールが 90 エポックから 300〜600 エポックへと伸びた理由はここにある。正則化強度と訓練期間は合わせて拡大しなければならない。短い訓練に強力なデータ拡張を詰め込むと、軽めのレシピに比べて正解率をかえって**損なう**ことが少なくない。

第 2 に、オプティマイザの行は万能の必須要件ではなく、選択肢の広がりを反映している。Adam のような適応的オプティマイザ（詳細は第 9 章で扱う）は、各パラメータの勾配をその大きさの移動推定値で正規化し、すべての座標がおよそ学習率のスケールで更新されるようにする。従来の実装ではこの再スケーリングの前に重み減衰を勾配に組み込んでいたため、意図せずペナルティが歪められていた。**AdamW** は重み減衰を適応的な勾配ステップから切り離し（分離型重み減衰、decoupled weight decay）、すべての重みを「学習率 × 減衰係数」に等しい割合で直接縮小する (Loshchilov and Hutter 2019)。この明快な修正により、画像系の各種ベンチマークにおいて適応的オプティマイザは入念に調整された SGD と互角に戦えるようになった。（LAMB は大バッチ向けの AdamW の派生手法である。前述の正解率 80.4% の ResNet-50 は、LAMB を用い、二値交差エントロピー（binary cross-entropy）の損失のもと重みの平均化なしで 600 エポック訓練されたものである。本節では LAMB は使用しない。）

7.6.2 各構成要素の実装

現代的な各構成要素は、ほんの数行のコードで記述できる。以下では、ラベル平滑化、ウォームアップ付きコサインスケジュール、Mixup、確率的深さ、そして指数移動平均による重みの平均化を実装する。（RandAugment は標準的な画像変換のランダムな系列を調整された強度で適用する手法であるが (Cubuk et al. 2020)、実務上は前処理ライブラリの呼び出しとして利用され、その真価は実装よりも強度調整にあるため、ここでは省略する。）

7.6.2.1 ラベル平滑化

ワンホットな正解ターゲットを用いた標準的な交差エントロピーのもとでは、正しく分類された例に対する損失は、正解クラスのロジットが $+\infty$ へ向かって大きくなるにつれて際限なく減少し続ける。有限のロジットでは目的関数が最小化されないため、ネットワークは較正（calibration）の不十分な極端な確信度へと絶えず押しやられる。**ラベル平滑化**（label smoothing）は、重み $1-\epsilon$ を正解クラス y に割り当て、残りの ϵ を全 K クラスに一様に分配したソフトな分布へとハードなワンホットターゲットを置き換えることで、この過信を抑止する (Szegedy et al. 2016)。予測されたソフトマックス確率を p_k と書くと、1 例あたりの損失は次のようになる。

$$l_\epsilon = -(1-\epsilon)\log p_y - \frac{\epsilon}{K}\sum_{k=1}^{K}\log p_k. \tag{7.8}$$

第 1 項は通常の交差エントロピーを $1-\epsilon$ 倍したものであり、第 2 項はいずれかのクラス確率を 0 に近づけることに対してペナルティを課す。この損失は $p_y = 1-\epsilon+\epsilon/K$ かつ $p_k = \epsilon/K$（$k \neq y$）のときに最小値をとり、これはロジット間の明確な有限のマージンに対応する。すでに高い確信度で正解している予測に対し、平滑化なしの交差エントロピーと平滑化ありの交差エントロピーのもとで何が起きるか見てみよう。

```
logits, y = d2l.tensor([[10.0, 0.0, 0.0]]), d2l.tensor([0])
for epsilon in (0.0, 0.1):
    loss = F.cross_entropy(logits, y, label_smoothing=epsilon)
    print(f'epsilon={epsilon}: loss={float(loss):.4f}')
```

```
epsilon=0.0: loss=0.0001
epsilon=0.1: loss=0.6668
```

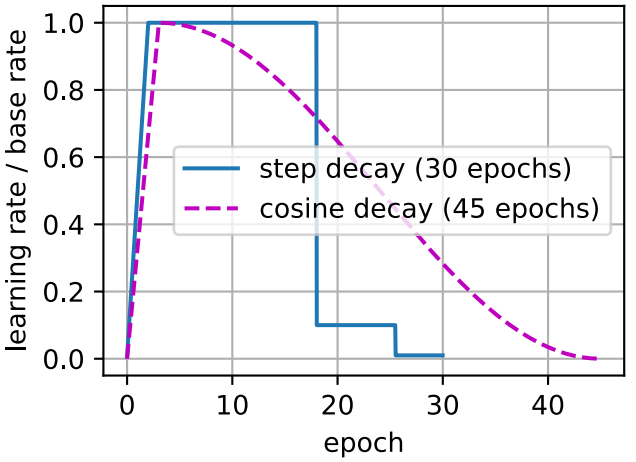
$\epsilon = 0.1$ とすると、損失は 0.0001 から 0.67 へと跳ね上がる。3 クラスの場合、平滑化された損失の最小値は約 0.29 であり、最大ロジットが他のロジットを約 3.3 上回るときに達成される。したがってマージン 10 は過信としてペナルティを受け、勾配の向きが反転して、ロジットをさらに引き離すのではなく適度な差へと引き戻すように働く。

7.6.2.2 ウォームアップ付きコサインスケジュール

従来のステップスケジュールは、数十エポックにわたって学習率を一定に保ったのち突如として 10 分の 1 に引き下げるため、減衰させるエポック数が調整すべき追加のハイパーパラメータとして生じてしまう。現代の標準手法 (Loshchilov and Hutter 2016) は、AlexNet 以降（7.1 節）ですでにおなじみのものである。すなわち、0 からピーク学習率 η_0 まで t_{w} ステップかけて線形に引き上げる**ウォームアップ**に続き、滑らかな半コサイン曲線でステップ T において 0 になるまで**減衰**させる（コサイン減衰、cosine decay）。ウォームアップは初期の発散を防ぐだけでなく、本格的なステップ幅での更新が始まる前にバッチ正規化の移動統計量や適応的オプティマイザのモーメントが落ち着く時間を与え、コサイン減衰は急激な減衰の閾値そのものを不要にする。以下

では、1 エポックあたり 79 ミニバッチとして、step_decay（30 エポック、2 エポックのウォームアップ付き）と d2l.cosine_warmup（45 エポック、3 エポックのウォームアップ付き）を並べてプロットする。これは本節後半の実験で用いるホライズンと完全に一致している。

```
def step_decay(t, num_steps, warmup_steps):
    """Recipe A: linear warmup, then divide by 10 at 60% and 85% of steps."""
    warmup = min((t + 1) / warmup_steps, 1)
    return warmup * 0.1 ** sum(t >= int(f * num_steps) for f in (0.6, 0.85))

steps_per_epoch = 79
steps_a = list(range(30 * steps_per_epoch))  # Recipe A: 30 epochs
steps_b = list(range(45 * steps_per_epoch))  # Recipe B: 45 epochs
d2l.plot([[t / steps_per_epoch for t in steps_a],
          [t / steps_per_epoch for t in steps_b]],
         [[step_decay(t, len(steps_a), 2 * steps_per_epoch) for t in steps_a],
          [float(d2l.cosine_warmup(t, len(steps_b), 3 * steps_per_epoch))
           for t in steps_b]],
         xlabel='epoch', ylabel='learning rate / base rate',
         legend=['step decay (30 epochs)', 'cosine decay (45 epochs)'])
```



7.6.2.3 Mixup

Mixup は、訓練例のペアの凸結合（convex combination）を用いて訓練することで分類器を正則化する手法である (H. Zhang et al. 2018)。ミニバッチごとに混合係数 $\lambda \sim \text{Beta}(\alpha, \alpha)$ をサンプリングし、各データ例 i をランダムにシャッフルした相手のデータ例 j と組にして次を構成する。

$$\begin{aligned}\tilde{\mathbf{x}} &= \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j,\\ \tilde{\mathbf{y}} &= \lambda \mathbf{y}_i + (1-\lambda)\mathbf{y}_j,\end{aligned} \tag{7.9}$$

ここで、$\mathbf{y}_i$ と $\mathbf{y}_j$ はワンホットなクラスベクトルであるため、$\tilde{\mathbf{y}}$ はソフトなターゲット分布となる。交差エントロピーはターゲット分布に関して線形であるため、$\tilde{\mathbf{y}}$ に対する損失は、λ で重み付けした y_i および y_j に対する交差エントロピーの和に等しくなり、より簡潔かつ高速に計算でき

る。訓練例のペア間で予測が線形に補間されるよう促すことで、個々のデータ点周辺での急峻な丸暗記が抑制される。典型的な小さい値（$\alpha \in [0.1, 0.2]$）ではベータ分布が 0 と 1 の近くに集中するため、大半のミニバッチには軽い摂動が加わるにとどまり、一部のミニバッチだけが強く混合される。以下では、ミニバッチごとに共通の λ を 1 つ用いる `mixup` を実装し、7.7 節で再利用できるよう `d2l` に保存する。

```
def mixup(X, y, alpha):
    """Mix a minibatch with a shuffled copy; return both label sets."""
    lam = random.betavariate(alpha, alpha)
    perm = torch.randperm(X.shape[0], device=X.device)
    return lam * X + (1 - lam) * X[perm], y, y[perm], lam
```

効果を視覚的にわかりやすく確認するため、Fashion-MNIST の衣類画像のミニバッチを $\alpha = 2$ で混合する。この設定では λ が 0.5 付近に集中する（実際の訓練ではるかに小さい α を用いる）。上段は元の画像、下段は重ね合わされた混合画像である。

```
data = d2l.FashionMNIST(batch_size=8)
X, y = next(iter(data.train_dataloader()))
X_mix, y_a, y_b, lam = mixup(X, y, alpha=2.0)
d2l.show_images(torch.cat([X, X_mix]).squeeze(1), 2, 8)
print(f'lambda = {lam:.2f}')
```

```
lambda = 0.81
```

Mixup は実際の写真には決して現れない幽霊のような二重露光画像を作り出すため、**CutMix** は代わりに空間的な混合を行う (Yun et al. 2019)。すなわち、一方の画像から矩形パッチを切り出して他方の画像に貼り付け、パッチの面積に比例してラベルを混合することで、すべてのピクセルを局所的に自然なまま保つ。現代の画像系レシピの多くは、ミニバッチごとに Mixup と CutMix をランダムに切り替えている（後述の演習問題を参照）。

7.6.2.4 確率的深さ

ドロップアウト（4.6 節）が訓練中に個々の活性化値をランダムに 0 にするのに対し、**確率的深さ**（stochastic depth）は残差分岐全体をランダムに無効化する (Huang et al. 2016)。確率 p でブロックの残差分岐がバイパスされて入力がそのまま通過するため、実質的により浅いサブネットワークのアンサンブルを訓練することになる。恒等ショートカット（identity shortcut）が常に信号を通し続けるため、残差接続のもとではこの操作を安全に行える。$\mathbf{x} + f(\mathbf{x})$ を計算するブロックに対し、確率的深さは分岐を次の式に置き換える。

$$\mathbf{x} + \frac{b}{1-p} f(\mathbf{x}), \qquad b \sim \text{Bernoulli}(1-p). \tag{7.10}$$

逆ドロップアウト（inverted dropout）と同様に、$1/(1-p)$ 倍でスケーリングすることで訓練中の分岐出力の期待値が不変に保たれるため、評価時の再スケーリングは不要である。現代の実装では、ミニバッチ内のデータ例ごとに独立して b をサンプリングする（以下の `drop_path` であり、ConvNeXt 用に `d2l` に保存する）。`d2l.Residual`（7.4 節）をサブクラス化し、ショートカットを加算する直前で残差分岐をドロップする。訓練モードで $p = 0.5$ とし、1,000 個のランダムな入力をこのブロックに通すと、全出力のおよそ半分が $\mathrm{ReLU}(\mathbf{x})$ と完全に一致するはずである。

```
def drop_path(Y, p, training):
    """Zero each example of Y with probability p, rescale the rest."""
    if not training or p == 0:
        return Y
    shape = (Y.shape[0],) + (1,) * (Y.ndim - 1)  # One draw per example
    keep = torch.rand(shape, device=Y.device) > p
    return Y * keep / (1 - p)

class StochasticResidual(d2l.Residual):
    """A residual block whose branch is dropped with probability p."""
    def __init__(self, num_channels, p):
        super().__init__(num_channels)
        self.p = p

    def forward(self, X):
        Y = self.bn2(self.conv2(F.relu(self.bn1(self.conv1(X)))))
        return F.relu(X + drop_path(Y, self.p, self.training))

blk, X = StochasticResidual(3, p=0.5), torch.randn(1000, 3, 8, 8)
Y = blk(X)  # A module is in training mode until .eval() is called
dropped = (Y == F.relu(X)).flatten(1).all(1).float().mean()
print(f'fraction of examples with a dropped branch: {float(dropped):.3f}')
```

```
fraction of examples with a dropped branch: 0.473
```

深いアーキテクチャでは、ドロップ確率 p を最初のブロックの 0 から最深ブロックのピーク値 0.1〜0.5 まで線形に増加させるのが一般的である。なお、データ例ごとのマスク処理では、ドロップされた例を 0 にする前にすべての分岐をひととおり計算するため、正則化にはなるが訓練の高速化にはつながらない。これに対し、元のミニバッチ単位の定式化ではブロックごとに単一の b を引くため、ドロップされた分岐の計算を完全にスキップして計算量を節約できる。

7.6.2.5 重みの平均化

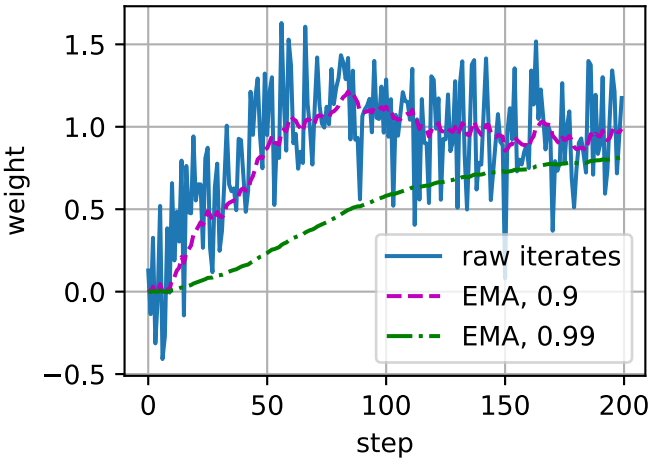
Table 7.4 の最後の行にある手法は、追加コストをほぼゼロに抑えつつ、より良好な汎化をもたらす。訓練の終盤では、ミニバッチノイズによってパラメータが低損失な盆地の底で跳ね回るため、直近の反復点を平均化すると、単一のチェックポイントよりも盆地の中心に近い位置に到達できる。**確率的重み平均化**（stochastic weight averaging、SWA）が訓練終盤のスナップショットを一様に平均するのに対し (Izmailov et al. 2018)、現代の画像系レシピでは一般に**指数移動平均**（exponential moving average、EMA）を保持する。これはバッチ正規化が移動統計量の更新に用いる漸化式（7.3 節）とまったく同じものである。オプティマイザの各ステップのあと、訓練可能なパラメータの**シャドウコピー** $\bar{\theta}$ に最新の重み θ を減衰率 ρ で混合する。

$$\bar{\theta} \leftarrow \rho\bar{\theta} + (1-\rho)\theta. \tag{7.11}$$

k ステップ前のパラメータは重み $(1-\rho)\rho^k$ を持ち、実効的な平均化ホライズンは約 $1/(1-\rho)$

ステップ（$\rho = 0.99$ なら 100 ステップ）となる。最適化は通常どおり θ を更新し続け、評価時には $\bar{\theta}$ に差し替える。（バッチ正規化の移動統計量は訓練可能な重みではなくバッファであるため、EMA 重みとどう組み合わせるのが最善かは後述の演習問題で検討する。）ρ におけるトレードオフを確認するため、50 ステップかけて 0 から 1 まで上昇したのち 1 の周りでノイズを伴って振動する単一の合成重みを追跡してみよう。

```
class EMA:
    """A shadow copy of a model's parameters, updated after each step."""
    def __init__(self, model, decay=0.99):
        self.decay = decay
        self.shadow = {k: p.detach().clone()
                       for k, p in model.named_parameters()}

    @torch.no_grad()
    def update(self, model):
        for k, p in model.named_parameters():
            self.shadow[k].lerp_(p, 1 - self.decay)  # beta*avg + (1-beta)*p

net = nn.Linear(1, 1, bias=False)
nn.init.zeros_(net.weight)  # Training starts at 0
emas = {decay: EMA(net, decay) for decay in (0.9, 0.99)}
raw, avg = [], {decay: [] for decay in emas}
for t in range(200):  # Iterates rise to 1 over 50 steps, then stay noisy
    with torch.no_grad():
        net.weight.copy_(min(t / 50, 1) + 0.3 * torch.randn(1, 1))
    raw.append(float(net.weight.detach()))
    for decay, ema in emas.items():
        ema.update(net)
        avg[decay].append(float(ema.shadow['weight']))
d2l.plot(list(range(200)), [raw, *avg.values()], xlabel='step',
         ylabel='weight', legend=['raw iterates', 'EMA, 0.9', 'EMA, 0.99'])
```



$\rho = 0.9$（ホライズン約 10 ステップ）では、シャドウ重みは約 10 ステップ遅れて上昇に追従し、その後はノイズを劇的に減衰させる。$\rho = 0.99$（ホライズン約 100 ステップ）ではさらに滑らかになるものの、遅れがあまりに大きく、200 ステップ経過後でも依然として 1 より約 20% 低

い位置にとどまっている。これは EMA の主な落とし穴を示している。実行ステップ数に比べて $1/(1-\rho)$ が大きいと、シャドウコピーは改善途上の重みに遅れをとり、最終チェックポイントよりも性能が劣ってしまうのである。したがって、0.999 や 0.9999 のような非常に大きい減衰率は、長期の訓練にのみ適している。

7.6.3 1 つのネットワークに対する 2 つのレシピ

ここまでの要素を組み合わせ、単一の固定アーキテクチャ上で直接比較テストを行ってみよう。用いるのは 7.4 節の ResNet-18 であり、`d2l.Residual` と He 初期化（4.4 節）を用いて再構築する。96×96 にリサイズした Fashion-MNIST 画像からランダムに抽出した 10,000 枚のサブセット（各試行内で両レシピが共有）で訓練する。訓練例を絞ることで、反復実験を高速に保ちつつ正則化の役割を際立たせることができる。両レシピとも、`lr` と `num_steps` から直接ステップスケジュールを計算する。

```
class FashionMNIST10k(d2l.FashionMNIST):
    """Fashion-MNIST with a random subset of num_train training images."""
    def __init__(self, batch_size=128, resize=(96, 96), num_train=10000):
        super().__init__(batch_size, resize)
        idx = torch.randperm(len(self.train))[:num_train]
        self.train.data = self.train.data[idx]  # get_dataloader reads these
        self.train.targets = self.train.targets[idx]

class ResNet18(d2l.Classifier):
    """The ResNet-18 of the residual-networks section, from d2l.Residual."""
    def __init__(self, lr, num_steps, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [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)]
        for i, c in enumerate((64, 128, 256, 512)):
            for j in range(2):
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):  # The Trainer applies it after a dry run
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            nn.init.kaiming_normal_(module.weight)  # Variance 2/n_in
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

両モデルはまったく同じアーキテクチャと訓練データ分割を共有する一方、最適化と正則化のレシピを全面的に変更している。**レシピ A** は 2015 年当時の実践を踏襲したもので、モメンタム 0.9 の SGD、結合型重み減衰 5×10^{-4}、ピーク学習率 0.05（2 エポックの線形ウォームアップののち、訓練の 60% と 85% の時点で 10 分の 1 に減衰）、通常の交差エントロピー損失、30 エポックという設定である。（2 エポックの短いウォームアップは、Goyal et al. (2017) が大規模ミニバッチ SGD 向けに導入した現代の軽微な安定化処置である。これがないと、学習率 0.05 のバッチ正

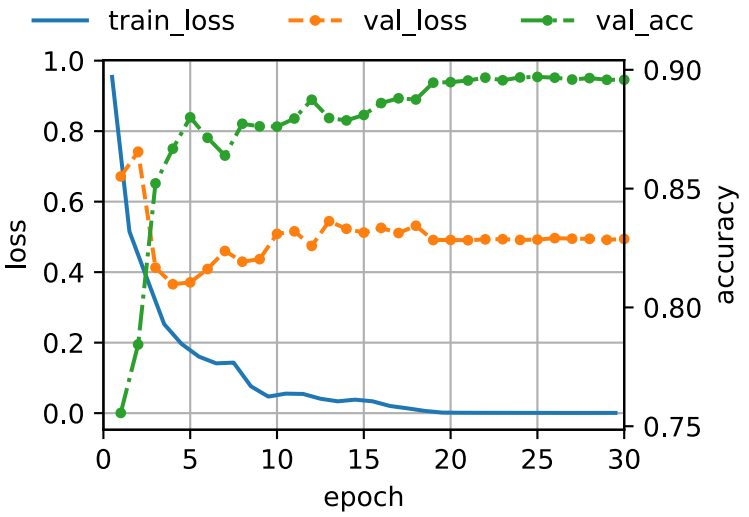
規化付き SGD は試行のおよそ 3 分の 1 でエポック 1 のうちに発散した。）一方、本節の**簡潔な現代的レシピ**では、重み減衰 0.05 の AdamW、ピーク学習率 0.002（3 エポックのウォームアップとコサイン減衰）、ラベル平滑化（$\epsilon = 0.1$）、Mixup（$\alpha = 0.2$）、そして（現代的な正則化が必要とする長いホライズンを反映した）45 エポックを用いる。実行時間を現実的な範囲に収めるため、RandAugment、CutMix、確率的深さ、EMA は省き、フルメニューから厳選したサブセットを検証する。

2 つの重み減衰の数値にある 100 倍の差（0.05 vs. 5×10^{-4}）に惑わされてはならない。1 ステップあたりの実際の縮小率は「学習率 × 重み減衰」であり、ピーク学習率においては AdamW が $0.002 \times 0.05 = 10^{-4}$ であるのに対し、SGD は $0.05 \times 5 \times 10^{-4} = 2.5 \times 10^{-5}$ となる。さらに SGD ではモメンタムがこれを最大 10 倍まで増幅するため、実効的な減衰はむしろ SGD の方が強い。最後に、どちらのレシピも混合していない検証画像に対する通常の交差エントロピーで検証損失を評価する。ラベル平滑化と Mixup が変更するのは訓練目的関数のみである。

```
class ClassicRecipe(ResNet18):
    """Recipe A: SGD with momentum, weight decay, warmup, step decay."""
    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                                weight_decay=5e-4)
        warmup = 2 * self.num_steps // 30  # Two of the 30 epochs
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: step_decay(t, self.num_steps, warmup))

class ModernRecipe(ResNet18):
    """Compact modern recipe: AdamW, warmup and cosine, smoothing, Mixup."""
    def configure_optimizers(self):
        warmup = 3 * self.num_steps // 45  # Three of the 45 epochs
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))

    def training_step(self, batch):
        X, y_a, y_b, lam = mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        smooth = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * smooth(y_a) + (1 - lam) * smooth(y_b)
```

まず、レシピ A を 30 エポック訓練し、10,000 枚の検証データ全体で評価する。

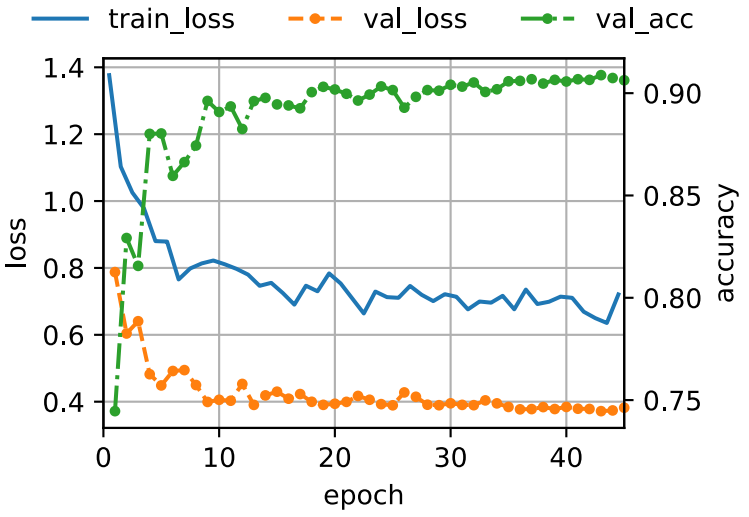
```
data = FashionMNIST10k()
steps_per_epoch = len(data.train_dataloader())  # 79 minibatches of <= 128
model = ClassicRecipe(lr=0.05, num_steps=30 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```




```
0.8958
```

次に、まったく同じアーキテクチャをまったく同じ訓練サブセット上で、簡潔な現代的レシピを用いて 45 エポック訓練する。

```
model = ModernRecipe(lr=2e-3, num_steps=45 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=45, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```




```
0.9063
```

反復試行（毎回新たな 10,000 枚の画像サブセット、初期化、ミニバッチ順序を抽選）にわたる検証正解率の範囲を Table 7.5 に示す。

Table 7.5: 両レシピのもとでの同一の ResNet-18（96×96 の Fashion-MNIST 訓練画像 10,000 枚、反復試行にわたる検証正解率の範囲）。

レシピ	エポック数	検証正解率
A（2015 年）：SGD、ウォームアップ、ステップ減衰	30	89〜90.5%
簡潔な現代的レシピ：AdamW、コサイン、平滑化、Mixup	45	90.5〜91%

同じ 10,000 枚の画像サブセットを用いたすべての対比較において、（RandAugment、CutMix、確率的深さ、EMA を省いているにもかかわらず）`ModernRecipe` は `ClassicRecipe` を 0.5〜1.5 パーセントポイント上回る。この比較からは 3 つの実践的な教訓が得られる。

1. **一括比較とアブレーション：**オプティマイザ、スケジュール、損失、データ拡張、エポック予算を一度に変更しているため、どの構成要素が最も寄与したかを切り分けるには 1 要素ずつのアブレーション（演習問題 4）を行うほかない。また、最適な学習率には $25\times$ もの開き（AdamW の 0.002 対 SGD の 0.05）があるため、オプティマイザを入れ替える際には必ず学習率を再調整しなければならない。
2. **レシピ間で生の訓練損失を比較してはならない：**`ClassicRecipe` のもとでは訓練損失が 0 に向かって急落する一方で、ロジットが過信状態になるにつれて検証損失が 0.4〜0.5 へと反発する。対照的に `ModernRecipe` のもとでは、検証損失が 0.4 未満まで下がる一方、Mixup とラベル平滑化がソフトに混合された訓練目的関数を評価するため、訓練損失は検証損失よりも**高い**まま推移する。
3. **公開リーダーボードの読み方：**訓練レシピだけで固定された ResNet-50 の ImageNet 正解率が > 4 ポイントも動く以上 (Wightman et al. 2021)、過去の論文から転記したベースラインに対する 1 ポイントの改善という主張は、レシピとアーキテクチャを混同している。たとえレシピを揃えたとしても、ImageNet にはテストセットのノイズや複数物体の曖昧さがあるため (Recht et al. 2019; Beyer et al. 2020)、1 ポイント未満の ImageNet top-1 正解率の差よりも、物体検出やセグメンテーションへの転移性能の方がバックボーンの品質を鋭く測る指標となることが多い。

7.6.4 まとめ

2015 年から 2020 年代初頭にかけて、畳み込みネットワークの訓練レシピはあらゆる軸で進化を遂げた。分離型重み減衰を備えた適応的オプティマイザ（AdamW、LAMB）、ウォームアップ付きコサインスケジュール、RandAugment・Mixup・CutMix、ラベル平滑化、確率的深さ、より長い訓練ホライズン、そして EMA による重みの平均化である。そうしたレシピの 1 つ（LAMB、コサインスケジュール、RandAugment と Mixup および CutMix、確率的深さ、600 エポック、重み平均化なし）は、無変更の ResNet-50 の ImageNet 正解率を 76.1% から 80.4% へと引き上げた。各構成要素はわずか数行のコードで実装でき、その簡潔なサブセットだけでも `FashionMNIST10k` 上の ResNet-18 を着実に改善する。続く 7.7 節では、現代的なレシピを固定したうえで、ResNet のアーキテクチャ自体を 1 段階ずつ現代化していく。

7.6.5 演習問題

1. **二値交差エントロピーと Mixup**。ImageNet 分類器の訓練において、Wightman et al. (2021) は K クラスのソフトマックス交差エントロピーではなく、二値交差エントロピー（K 個の各クラスを独立したシグモイド二値分類タスクとして扱う）を用いている。混合された画像に異なる 2 つのクラスの物体が同時に含まれるとき、独立したシグモイド出力が Mixup や CutMix と自然に相性がよい理由を説明せよ。

2. [code] **CutMix**。CutMix (Yun et al. 2019) は $\lambda \sim \mathrm{Beta}(1,1) = \mathrm{Uniform}(0,1)$ をサンプリングし、シャッフルした相手の画像から目標面積比 $1-\lambda$ を覆うランダムな矩形パッチを貼り付ける。`cutmix` を `mixup` とそのまま差し替え可能な変種として実装し、矩形を画像境界でクリッピングしたあとに残るピクセルの**厳密な**割合を用いてワンホットラベルを混合するようにせよ。`ModernRecipe` を `FashionMNIST10k` 上で、Mixup の代わりに CutMix を用いた場合と、ミニバッチごとに Mixup と CutMix をランダムに切り替えた場合のそれぞれで訓練し、検証正解率を Table 7.5 と比較せよ。

3. [code] **重み平均化のホライズン（EMA）**。オプティマイザの各ステップ後にシャドウ重みを更新し、シャドウパラメータを用いて検証正解率を評価するように、`EMA` クラスを `ModernRecipe` に組み込め。減衰率 $\rho \in \{0.9, 0.99, 0.999\}$ を Equation 7.11 において変化させて実験せよ。最も性能のよい ρ を実効ホライズン $1/(1-\rho)$ および総ステップ数（$45 \times 79 = 3{,}555$）と関連付けて考察し、EMA 重みを評価する際にバッチ正規化の移動統計量をどのように扱うべきかを説明せよ。

4. [code] **1 要素ずつのレシピ・アブレーション**。●Table 7.5 では、5 つの構成要素をすべて同時に変更したときに `ModernRecipe` が `ClassicRecipe` を 0.5〜1.5 パーセントポイント上回った。

    1. `ClassicRecipe` と `ModernRecipe` の両方を `FashionMNIST10k` 上で 5 回ずつ実行し、試行間の検証正解率の標準偏差を推定せよ。
    2. `ModernRecipe` を出発点として、構成要素を 1 つずつ `ClassicRecipe` の対応する設定に戻せ。すなわち、(i) Mixup を外す、(ii) ラベル平滑化を外す、(iii) コサインウォームアップを `step_decay` に置き換える、(iv) AdamW をモメンタム付き SGD に置き換える（ピーク学習率は 0.05 に再調整する）、(v) 訓練を 45 エポックから 30 エポックに短縮する。試行間のノイズ水準を超える変化をもたらすのはどの個別変更か。
    3. Mixup とラベル平滑化を有効にした状態で、2×2 の「オプティマイザ（SGD vs. AdamW）$\times$ スケジュール（`step_decay` vs. `cosine_warmup`）」の表を完成させよ。オプティマイザとスケジュールの間に相互作用はあるか。

5. [code] **レシピへの確率的深さの導入**。●本節の簡潔な現代的レシピでは確率的深さを省略した。これを `ResNet18` に追加せよ。

    1. 1×1 ショートカットを持つストライド 2 のダウンサンプリングブロックに対応できるよう `StochasticResidual` を拡張し、ドロップ確率がブロック 1 の 0 からブロック 8 の $p_{\max}$ まで線形に増加する `ResNet18` を構築せよ。1 回の順伝播あたりに有効となる残差分岐の個数の期待値を $p_{\max}$ の関数として求めよ。
    2. $p_{\max} \in \{0.1, 0.2, 0.5\}$ を変化させて `ModernRecipe` を `FashionMNIST10k` 上で実行せよ（$p_{\max} = 0$ と比較すること）。8 ブロックの浅いネットワークでは、50 ブロックや 100 ブロックのネットワークに比べて大きい $p_{\max}$ の恩恵が小さいのはなぜか。
    3. Huang et al. (2016) によるミニバッチ単位の変種（ブロックごとに単一のベルヌーイ変数 b をサンプリングし、$b = 0$ のときは残差分岐の評価を完全にスキップする方式）を実装せよ。$p_{\max} = 0.5$ において、スキップされる分岐の理論上の割合と実測された訓練の実時間の短縮率を比較せよ。

7.7 ConvNeXt：ResNet の段階的な現代化

本章でここまで扱ってきたアーキテクチャはすべて畳み込みをベースとしており、2020 年までは畳み込みネットワークがあらゆる主要な画像分類のリーダーボードで頂点に君臨していた。そこへ、重なりのある空間畳み込みを 1 つも持たず、**アテンション**を中心に構築されたアーキテクチャが登場して首位を奪った。最初のビジョン Transformer は 2020 年後半、ImageNet よりはるかに大規模な事前学習データセットの力を借りてこれを成し遂げた。しかし 2021 年の

Swin Transformer (Liu et al. 2021) はそのような助けを必要とせず、ImageNet のみで訓練して 81.3% の top-1 正解率を達成した。これに対し、同規模の ResNet-50 は 76.1% であった。この 5 ポイントもの飛躍はアテンション自体がもたらしたものなのだろうか。そしてもしそうなら、畳み込みネットワークは時代遅れになったのだろうか。

ここではビジョン Transformer の概略を述べるにとどめる（詳細は第 11 章で展開する）。**ビジョン Transformer** (Dosovitskiy et al. 2021) は、画像をパッチの格子に分割し、各パッチを特徴ベクトルとして埋め込み、その系列を同一構造のブロックの積層に通して処理する。各ブロックは 2 つのサブ層を持つ。すなわち、すべてのパッチが他のパッチのデータ依存な重み付き平均を計算できるようにする**アテンション**層と、それに続いて各パッチベクトルに独立に適用されるポイントワイズ MLP である。Swin は、局所的な 7×7 ウィンドウ内でアテンションを計算し、ステージ間でダウンサンプリングすることで、階層的な畳み込みネットワークの構造を復活させている。

7.6 節を踏まえれば、5 ポイントの差をアテンションだけの功績とみなすことにはすぐさま疑問を持つべきだろう。Swin はアテンション以外にも、少なくとも十数個の点で 2015 年の ResNet と異なっている。300 エポックの現代的な訓練レシピ（training recipe）で訓練し、パッチステムで画像をスライスし、ステージ間で深さの配分を変え、さらに異なる活性化関数と正規化を採用しているのである。これらの要因を切り分けるため、Z. Liu et al. (2022) は画期的な対照実験を実施した。標準的な ResNet-50 から出発し、アテンションを一切追加することなく Swin に着想を得た現代化を 1 つずつ導入し、変更のたびに ImageNet の top-1 正解率を測定したのである。こうして得られた純粋な畳み込みネットワーク **ConvNeXt** は 82.0% に達し、本章ですでに構築してきた構成要素、すなわち 7.6 節の現代的な訓練レシピ、7.5 節のデプスワイズ畳み込み（depthwise convolution）と逆ボトルネック（inverted bottleneck）、および 7.3 節の層正規化（layer normalization）だけで Swin-T をわずかに上回った。このアブレーションの各行には ImageNet 全体の訓練が必要となるため、本節ではまず公開されたロードマップを段階的にたどり、続いて ConvNeXt をゼロから実装したうえで、パラメータ数を揃えた ResNet-18 から出発する独自のコンパクトな 3 ステップの現代化ロードマップを Fashion-MNIST 上で実行する。

7.7.1 現代化のロードマップ

Table 7.6 に、Z. Liu et al. (2022) が検証した一連の変更を順番通りに示す。各行はそれより上のすべての変更を累積しつつ、画像あたり約 45 億回の積和演算（multiply-add）という Swin-T の計算予算付近に計算量を保っている（論文中では 4.5 GFLOPs と報告されている）。

Table 7.6: ConvNeXt の現代化ロードマップ (Z. Liu et al. 2022)。設計変更を累積したときの ResNet-50 の ImageNet top-1 正解率であり、各行はほぼ一定のコストのもとでそれより上のすべての行を含む。同程度のコストの Transformer である Swin-T は 81.3% に達する。

変更項目	内容	Top-1
（出発点）	2015 年のレシピによる ResNet-50	76.1%
現代的な訓練レシピ	300 エポック、AdamW、Mixup、ラベル平滑化、確率的深さ（7.6 節）	78.8%
ステージ比	ステージごとのブロック数 $(3,4,6,3) \to (3,3,9,3)$	79.4%
パッチ化ステム	ストライド 2 の 7×7 畳み込みとプーリング $\to$ ストライド 4 の単一の 4×4 畳み込み	79.5%
デプスワイズ畳み込み	3×3 畳み込みをデプスワイズ化し、幅を 64 から 96 チャネルに拡大	80.5%

変更項目	内容	Top-1
逆ボトルネック	ブロック内で圧縮する代わりに 4× に拡大	80.6%
大カーネル	デプスワイズ畳み込みを先頭へ移動し、7×7 に拡大	80.6%
ReLU → GELU	Transformer と同様の滑らかな活性化関数	80.6%
活性化関数の削減	ブロックあたりの活性化関数を 3 個から 1 個に削減	81.3%
正規化の削減	ブロックあたりの正規化を 3 個から 1 個に削減	81.4%
BN → LN	各位置でのチャネル方向の層正規化	81.5%
独立したダウンサンプリング	ステージ間に LN とストライド 2 の 2×2 畳み込みを配置	82.0%

いきなり最初の行が、ネットワークの層に一切手を加えることなく最大の単一の上昇をもたらしている。7.6 節と呼応するように、訓練レシピを刷新するだけで、無変更の ResNet-50 が 2.7 パーセントポイント向上する。2015 年の ResNet-50 と ConvNeXt を隔てる合計 5.9 ポイントの改善のうち、（この適用順序では）2.7 ポイントが訓練、3.2 ポイントがアーキテクチャに由来しており、現代的な Transformer を 2015 年のレシピによる ResNet と競わせていた比較では、レシピとアーキテクチャが交絡していたことが裏づけられる。

7.7.1.1 マクロ設計

続く 2 行は、ネットワークのマクロ構造を組み替える。Swin-T は 4 つのステージに 1:1:3:1 の比率でブロックを配分し、特徴マップが小さく計算コストが安いステージ 3 に深さを集中させている。ResNet-50 の $(3, 4, 6, 3)$ という配置を $(3, 3, 9, 3)$ に変更すると、0.6 ポイント向上する。次はステムである。ResNet が重なりのあるストライド 2 の 7×7 畳み込みに最大プーリングを続けて 4 分の 1 にダウンサンプリングするのに対し、ビジョン Transformer は画像を重なりのないパッチに分割し、各パッチを共有の線形層で射影する。これは代数的には、重なりのないストライド 4 の 4×4 畳み込みと同一である（6.3 節）。この**パッチ化**（patchify）ステムに置き換えても正解率は維持され（79.5%）、Transformer とまったく同様に 4×4 パッチごとに 1 つのトークンベクトルが生成される。

7.7.1.2 デプスワイズ畳み込みと逆ボトルネック

Transformer ブロックは計算を 2 つの段階にきれいに分解する。アテンションが**空間位置にわたって**情報を混合し、MLP が各位置で**チャネルにわたって**情報を混合する。7.5.1.1 節で見たように、畳み込みネットワークもこれとまったく同じ分解を実行できる。すなわち、**デプスワイズ**畳み込みが各チャネル内で空間方向に情報を混合し、1×1 畳み込みが各ピクセルでチャネル方向に情報を混合する。ResNet のボトルネック内の密な 3×3 層をデプスワイズ畳み込みに置き換えると、計算量が大幅に削減されるため、ステージ 1 の幅を 64 から 96 チャネルへと（Swin-T に合わせて）広げることができ、正解率は 80.5% へと跳ね上がる。

空間畳み込みが密ではなくデプスワイズになると、そのコストは幅に対して二乗ではなく線形で増加するため、フィルタリングの前にチャネルを圧縮する理由がなくなる。そこで ConvNeXt は MobileNetV2 の**逆ボトルネック**（7.5 節）を採用し、細い残差ストリームを保ちながらブロック内部で 4× に拡大する（Transformer の MLP と同じ $1\times \to 4\times \to 1\times$ の構造）。これにより正解率は 80.6% へとわずかに上昇する。

空間フィルタリングが軽量になったことで、カーネルサイズの拡大も容易になる。まず、作用対象が c チャネル（$4c$ ではなく）になるようデプスワイズ畳み込みをブロックの先頭へ移動させると、正解率は 0.7 ポイント低下する（79.9% となり、ロードマップ上で唯一の低下である）。しかし、そのカーネルを 3×3 から 7×7 に拡大すると、その低下分は即座に取り戻される。1 つの 7×7 フィルタは、3 層重ねた 3×3 層の受容野と一致する（6.3 節）。どちらも同じ範囲を覆うのに、なぜ単一の大カーネルが有効なのだろうか。積層構造はウィンドウ内を不均一に覆うからである。境界のピクセルよりも中央のピクセルから出力へ至る経路の方がはるかに多いため、積層構造の**有効**受容野（effective receptive field）はほぼガウス分布状になり、名目上の受容野よりずっと小さくなる (Luo et al. 2016)。これに対し、単一の 7×7 カーネルであれば、ウィンドウ内のあらゆる位置に学習した任意の重みを 1 層で与えられる。かつての VGG（7.2.1 節）では、密な 7×7 カーネルは計算コストが高すぎて実用的でなかったが（$k^2c_{\text{i}}c_{\text{o}}$）、デプスワイズカーネル（$k^2c$）であれば、$3\times3$ から 7×7 へ拡大してもブロック全体の計算量のごく一部が $49/9\approx5$ 倍になるにすぎない。正解率はちょうど 7×7 で飽和し、Swin の局所アテンションのウィンドウサイズとぴったり一致する。

7.7.1.3 ミクロ設計

最後の変更群は、ブロック内部の個々の層を洗練させるものである。ReLU を、Transformer で好まれる滑らかな活性化関数である **GELU** (Hendrycks and Gimpel 2016) に置き換えても、それ単独では正解率は変わらない（80.6%）。**実際に**大きな効果をもたらすのは、活性化関数と正規化の**数**を減らすことである。従来の ResNet のボトルネックは 3 つの畳み込みすべてに正規化と活性化関数を続けるのに対し、Transformer ブロックは MLP 内部に単一の活性化関数しか持たず、正規化の数も少ない。3 つの GELU のうち 2 つを取り除き、2 つの 1×1 層の間にある 1 つだけを残すと、正解率は 0.7 ポイント跳ね上がり、表中のミクロ設計として最大の改善となる（2 つの非線形性を取り除くとなぜ正解率が向上するのかについて明快な理論的証明はまだなく、純粋な経験的知見とみなすのが妥当である）。さらに正規化を 3 つから 1 つに減らすと 0.1 ポイント向上し、残った唯一の正規化をバッチ正規化から位置ごとのチャネル方向の**層正規化**（7.3.4.1 節）へ切り替えるとさらに 0.1 ポイント向上する（81.5%）。Figure 7.20 に、元の ResNet のボトルネックと、こうして得られた ConvNeXt ブロック（後述するチャネルごとの学習可能な層スケール（layer scale）γ を含む）の比較を示す。

最後に、ResNet が各ステージの先頭ブロック内部でストライド 2 の畳み込みによりダウンサンプリングするのに対し、Swin はステージ間に専用のパッチ統合層を挿入する。独立したダウンサンプリング層（ステージ間の層正規化とそれに続く 2×2 のストライド 2 畳み込みに加え、訓練を安定させるためのステム直後および線形ヘッド直前の正規化）を採用することで、ロードマップは 82.0% で完結する。Figure 7.21 に、古典的なステム・ボディ・ヘッド構成（7.2.3 節）のもとで $(3,3,9,3)$ 個のブロックを幅 $(96,192,384,768)$ で積み重ねた ConvNeXt-T バックボーンの全体像を示す。

同等の計算量のもとで、ConvNeXt-T はアテンション層を 1 つも使わずに Swin-T を上回る。ここから得られる教訓は、畳み込みが本質的にアテンションより優れているということではなく、2021 年時点での正解率の差はアテンション単独ではなく訓練レシピとアーキテクチャ設計の洗練に起因していたということである。条件を揃えて調整すれば、両方のパラダイムは同等にスケールする。

7.7.2 実装

現代化のロードマップでは追加した層よりも削ぎ落とした層の方が多いため、ConvNeXt ブロックは驚くほど簡潔に実装できる。

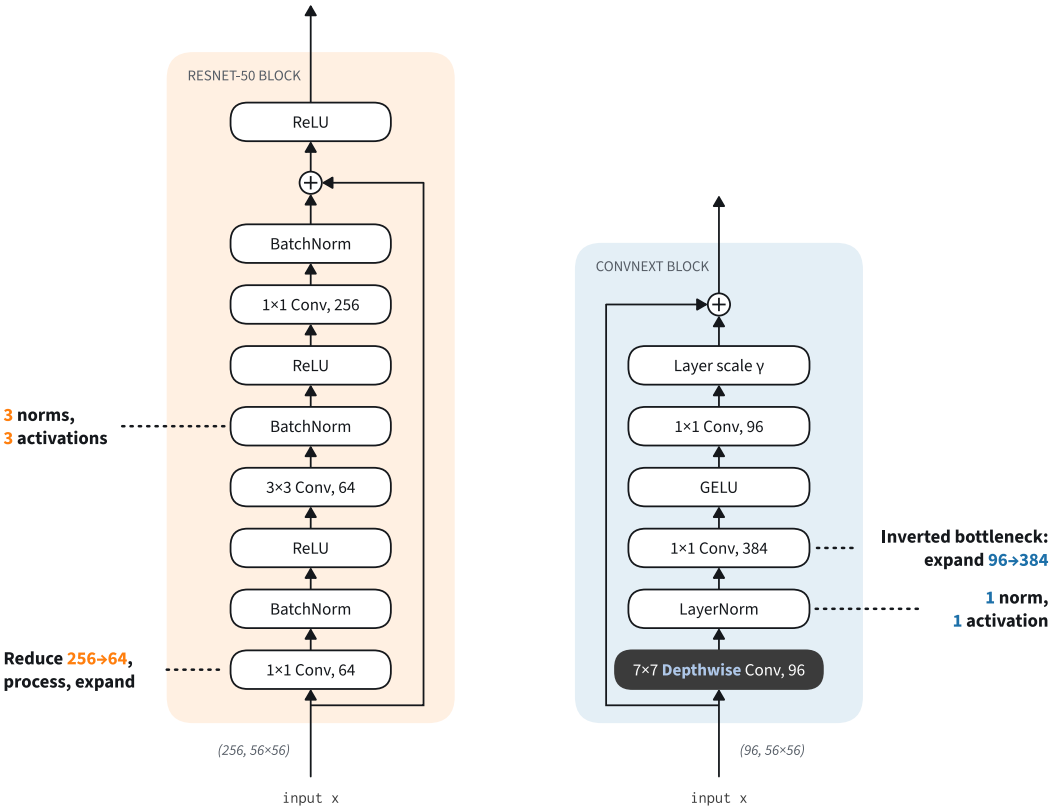


Figure 7.20: ResNet-50 のボトルネックブロックと、ロードマップによって変換された後の ConvNeXt ブロック。3 つの正規化と 3 つの活性化関数がそれぞれ 1 つずつになり、圧縮・処理・拡大のボトルネックが 7×7 デプスワイズ畳み込みを先頭に持つ逆ボトルネックへと変わり、さらにチャネルごとの学習可能な層スケール γ が加算前のブランチに乗算される。

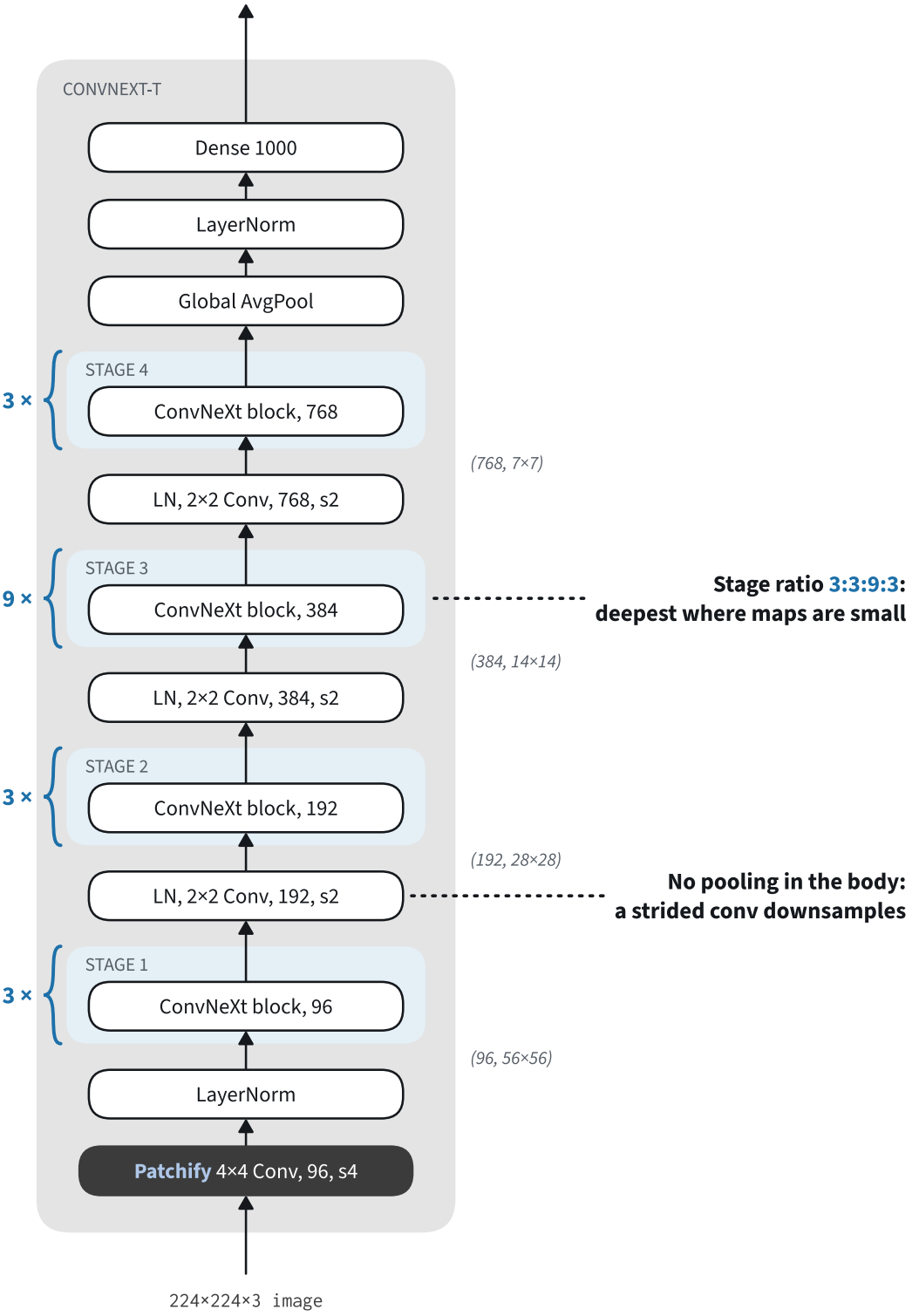


Figure 7.21: ConvNeXt-T。パッチ化ステムが畳み込みとプーリングを置き換え、3、3、9、3 ブロックの各ステージが独立した LN と 2×2 のストライド 2 ダウンサンプリング層で接続される。ボディのどこにもプーリングは存在しない。

7.7.2.1 ConvNeXt ブロック

各ブロックは 7×7 デプスワイズ畳み込み、位置ごとの層正規化、1×1 の拡大 ($c \to 4c$)、GELU、および 1×1 の射影 ($4c \to c$) を順に適用し、最後に残差入力を加算する。このブロックには、論文で導入された訓練安定化の仕組みが 2 つ組み合わされている。**層スケール** (Touvron, Cord, Sablayrolles, et al. 2021) は、学習可能なチャネルごとのベクトル γ (初期値 10^{-6}、バッチ正規化のスケールとは別物) を残差ブランチに乗算することで、すべてのブロックがほぼ恒等写像として始まるようにし、深いネットワークの最適化を滑らかに立ち上げる。これは層正規化を用いるブロックにおいて、ResNet で残差ブランチ最後のバッチ正規化スケールを 0 から始める手法 (7.4.2 節) と同じ役割を果たす。**確率的深さ** (stochastic depth) は、訓練中に `d2l.drop_path` (7.6 節) を用いて残差ブランチ全体をランダムにドロップする。

層正規化は各空間位置のチャネルベクトルに対して作用するため、チャネルラストのテンソル (`batch, height, width, channels`) においては最終軸に対する標準的な `LayerNorm` となり、2 つの 1×1 畳み込みも同じ軸に対する通常の線形層となる。Z. Liu et al. (2022) に従い、すべての畳み込み層と線形層の重みを $\mathcal{N}(0, 0.02^2)$ から初期化し、バイアスは 0 とする。

画像はチャネルファーストで格納されるため、`LayerNorm2d` は正規化の際にチャネルラストへ軸を並べ替えてから元に戻し、ブロックも 2 つの線形層の前後で同様の並べ替えをする。

```
class LayerNorm2d(nn.LayerNorm):
    """LayerNorm over the channel axis of an NCHW tensor."""
    def __init__(self, num_channels, eps=1e-6):
        super().__init__(num_channels, eps=eps)

    def forward(self, X):
        return super().forward(X.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)

class ConvNeXtBlock(nn.Module):
    """Depthwise 7x7, norm, 1x1 expand, GELU, 1x1 project, scaled residual."""
    def __init__(self, dim, drop_prob=0.0, layer_scale=1e-6,
                 norm=LayerNorm2d):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size=7, padding=3,
                                groups=dim)
        self.norm = norm(dim)
        self.pwconv1 = nn.Linear(dim, 4 * dim)  # 1x1 conv in channels-last
        self.pwconv2 = nn.Linear(4 * dim, dim)
        self.gamma = nn.Parameter(layer_scale * torch.ones(dim))
        self.drop_prob = drop_prob

    def forward(self, X):
        Y = self.norm(self.dwconv(X)).permute(0, 2, 3, 1)  # to channels-last
        Y = self.pwconv2(F.gelu(self.pwconv1(Y)))
        Y = (self.gamma * Y).permute(0, 3, 1, 2)  # back to channels-first
        return X + d2l.drop_path(Y, self.drop_prob, self.training)
```

7.7.2.2 ネットワーク全体

VGG (7.2.1 節) と同様に、ネットワーク全体はステージ仕様 (`depth, channels`) のタプルを順にたどり、先頭にパッチ化ステムと層正規化を置き、ステージ間に専用のダウンサンプリング層を挿入する (ボディのどこにも空間プーリングは置かない)。確率的深さのドロップ確率は、最初のブロックの 0 から最後のブロックの `drop_path_max` まで線形に増加させる。Fashion-MNIST に対

しては、幅 $(40, 80, 160, 320)$、ステージ深さ $(2, 2, 6, 2)$ を持つコンパクトな **ConvNeXt-atto** の構成を採用し、ConvNeXt-T と同様に第 3 ステージを最も深く保つ。`norm` を引数として、`stem` をメソッドとして公開しておくことで、後続のアブレーションにおける差し替えが容易になる。

```
class ConvNeXt(d2l.Classifier):
    def __init__(self, arch=((2, 40), (2, 80), (6, 160), (2, 320)),
                 lr=2e-3, num_classes=10, drop_path_max=0.1,
                 norm=LayerNorm2d):
        super().__init__()
        self.save_hyperparameters()
        depths = [d for d, c in arch]
        rates = torch.linspace(0, drop_path_max, sum(depths)).tolist()
        layers = self.stem(arch[0][1], norm)
        c_prev, b = arch[0][1], 0
        for i, (depth, c) in enumerate(arch):
            if i > 0:  # Separate downsampling layer between stages
                layers += [norm(c_prev),
                           nn.Conv2d(c_prev, c, kernel_size=2, stride=2)]
            for _ in range(depth):
                layers.append(ConvNeXtBlock(c, rates[b], norm=norm))
                b += 1
            c_prev = c
        layers += [nn.AdaptiveAvgPool2d((1, 1)), norm(c_prev), nn.Flatten(),
                   nn.Linear(c_prev, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(self.weight_init)

    def stem(self, c, norm):  # Patchify: a 4x4 convolution with stride 4
        return [nn.Conv2d(1, c, kernel_size=4, stride=4), norm(c)]

    def weight_init(self, module):
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.normal_(module.weight, std=0.02)
            nn.init.zeros_(module.bias)
```

64×64 の Fashion-MNIST 入力に対し、パッチ化ステムは 16×16 の格子を生成し、3 つのダウンサンプリング層がグローバル平均プーリングの前に解像度を 8×8、4×4、2×2 へと段階的に縮小する。出力の形状を確認し、訓練可能なパラメータ数を数えてみよう。

```
model = ConvNeXt()
assert model(torch.randn(1, 1, 64, 64)).shape == (1, 10)
d2l.num_params(model)
```

```
3376450
```

この ConvNeXt-atto モデルは正確に 3,376,450 個のパラメータを持ち、これは ResNet-18 の 1120 万個（7.4 節）のおよそ 3 分の 1 である。幅やカーネルサイズの設定ミスがあればこの正確な数値はただちにずれるため、パラメータ数の一致を確認することは再実装における信頼性の高い検証手段となる。

7.7.3 Fashion-MNIST での小規模ロードマップ

Table 7.6 のすべての選択は、130 万枚の 224×224 の ImageNet 写真で検証されたものであった。同じアーキテクチャ上の判断は、小さな 64×64 のグレースケール衣類画像にもそのまま通用するのだろうか。これを確かめるため、独自のコンパクトな累積ロードマップを実行する。パラメータ数を揃えた対照モデルとして、基本幅を 64 から 35 チャネルに狭めた ResNet-18（パラメータ数 3.35M）から出発し、ConvNeXt の変更を 3 つの累積ステップで適用していく。すべてのモデルは、7.6 節のコンパクトな現代的レシピ（分離型重み減衰（decoupled weight decay）を伴う AdamW、10% の線形ウォームアップとコサイン減衰、ラベル平滑化（label smoothing）、および Mixup。AdamW の詳細は第 9 章で扱う）のもとで、Fashion-MNIST 全体に対して 20 エポック訓練する。アーキテクチャの影響を純粋に切り分けるため、ConvNeXt の確率的深さの率は 0 に設定し、両モデル族で正則化を完全に同一にする。

訓練手順を `ModernRecipe` ミックスインとしてまとめておくことで、検証損失を通常の交差エントロピーに保ったまま、ロードマップ内のすべてのモデルにまったく同じオプティマイザ、スケジュール、および混合訓練ステップを継承させられる。

```
class ModernRecipe:
    """AdamW, linear warmup then cosine decay, label smoothing, Mixup."""
    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        warmup = self.num_steps // 10
        schedule = torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
        return optim, schedule  # The Trainer steps it after every minibatch

    def training_step(self, batch):
        X, y_a, y_b, lam = d2l.mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        loss = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * loss(y_a) + (1 - lam) * loss(y_b)

class ModernConvNeXt(ModernRecipe, ConvNeXt):
    def __init__(self, num_steps, lr=2e-3, norm=LayerNorm2d):
        super().__init__(lr=lr, drop_path_max=0.0, norm=norm)
        self.num_steps = num_steps
```

ベースラインとなる `CompactResNet18` は、ResNet 標準の 7×7 ストライド 2 ステムと最大プーリングの後ろに、幅 35、70、140、280 にわたって `d2l.Residual` ブロック（7.4 節）を積み重ねる。

```
class CompactResNet18(ModernRecipe, d2l.Classifier):
    """ResNet-18 with base width 35 instead of 64, trained by the recipe."""
    def __init__(self, num_steps, lr=2e-3, num_classes=10, base=35):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.LazyConv2d(base, kernel_size=7, stride=2, padding=3,
                                bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
        for i, c in enumerate((base, 2 * base, 4 * base, 8 * base)):
            for j in range(2):
```

```
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

論文の変更点をアーキテクチャの部位ごとに 3 つの明快なステップにまとめる。(1) ResNet のステムとバッチ正規化は維持したまま、ブロックとステージを ConvNeXt の 7×7 逆ボトルネック、$(2, 2, 6, 2)$ のステージ比、単一の GELU と正規化、層スケール、および独立したダウンサンプリングに置き換える（Table 7.6 の 8 行分に相当）。(2) バッチ正規化を層正規化に置き換える。(3) ResNet のステムを 4×4 パッチ化ステムに置き換えて完全な ConvNeXt に到達する（元の論文ではステムをより早い段階で置き換えていたが、各ステップへの寄与の割り当てがわずかに変わり得るとしても、最終的なアーキテクチャは順序に依存しない）。`stem` を `ResNetStemConvNeXt` でオーバーライドし、`norm` を切り替えることで、4 つのモデルすべてを簡潔に実装できる。

```
class ResNetStemConvNeXt(ModernConvNeXt):
    """ConvNeXt with the stem of ResNet-18 instead of the patchify stem."""
    def stem(self, c, norm):  # 7x7 convolution with stride 2, max-pooling
        return [nn.Conv2d(1, c, kernel_size=7, stride=2, padding=3), norm(c),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]

roadmap = {'compact ResNet-18': CompactResNet18,
           'ConvNeXt blocks and stages, BN': (
               lambda num_steps: ResNetStemConvNeXt(num_steps,
                                                    norm=nn.BatchNorm2d)),
           '+ BN -> LN': ResNetStemConvNeXt,
           '+ patchify stem = ConvNeXt': ModernConvNeXt}
```

4 つのモデルすべてを順番に訓練し、それぞれのパラメータ数と最終的な検証正解率を記録しよう。

```
data = d2l.FashionMNIST(batch_size=128, resize=(64, 64))
num_steps = 20 * len(data.train_dataloader())
results = {}
for name, make_model in roadmap.items():
    model = make_model(num_steps)
    trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
    trainer.board.display = False  # Four runs; we tabulate the results
    trainer.fit(model, data)
    results[name] = (d2l.num_params(model), trainer.history['val_acc'][-1])
```

続いて `results` を順にたどり、`roadmap` の各ステップにおけるパラメータ数と 20 エポック後の検証正解率を表示する。

```
    for name, (params, accuracy) in results.items():
        print(f'{name:34s} {params:>9,} parameters, '
              f'validation accuracy {accuracy:.3f}')
```

```
compact ResNet-18                  3,346,675 parameters, validation accuracy 0.936
ConvNeXt blocks and stages, BN     3,377,770 parameters, validation accuracy 0.937
+ BN -> LN                         3,377,770 parameters, validation accuracy 0.937
+ patchify stem = ConvNeXt         3,376,450 parameters, validation accuracy 0.918
```

Table 7.7: Fashion-MNIST における本節のロードマップと論文のロードマップの対比。最後の列は 64×64 において現代的なレシピで 20 エポック訓練した後の検証正解率であり、複数回実行した際の安定範囲に丸めている。論文の列は、各ステップが対応する Table 7.6 の行の ImageNet top-1 改善幅を合計したものである。

ステップ	論文の対応行	ImageNet での改善幅	パラメータ数	検証正解率
コンパクトな ResNet-18	（現代的レシピによる ResNet-50）		3.35M	≈94%
ConvNeXt のブロックとステージ、BN	ステージ比〜正規化の削減、独立したダウンサンプリング	+3.0	3.38M	≈94%
BN → LN	BN → LN	+0.1	3.38M	≈94%
パッチ化ステム（本節の ConvNeXt）	パッチ化ステム	+0.1	3.38M	≈92%

Table 7.7 の結果は、64×64 の Fashion-MNIST に何が転移し、何が転移しないかを明確に示している。ResNet の重なりのあるステムを維持すれば、ConvNeXt の現代化されたブロック、ステージ比、および層正規化は、パラメータ数を揃えた `CompactResNet18` と並ぶ（約 94%、画像あたり約 43〜46M 回の積和演算の範囲内）。約 92% への約 2 ポイントの低下はすべて、224×224 の ImageNet では中立だったたった 1 つのステップに起因している。すなわち、重なりのない 4×4 ストライド 4 のパッチ化ステムであり、これは 1 回の線形射影で粗い 16×16 の格子へと一気に飛び越えてしまう。ImageNet の解像度に合わせて調整されたアーキテクチャ上の選択が低解像度の領域へ無条件に転移するとは限らず、累積アブレーションを実施すれば適応させるべき構成要素をただちに特定できる。

7.7.4 限界と未解決問題

ConvNeXt は、レシピを揃えれば純粋な畳み込みネットワークが教師あり分類において階層的なビジョン Transformer に匹敵することを示したが、なお 2 つの活発な研究領域が残されている。

1. **マスク付き自己教師あり事前学習：**マスク付きオートエンコーダ (He et al. 2022) のようにランダムな画像パッチをドロップする処理はパッチトークンの系列には自然になじむが、密にスライドする畳み込みではマスクの穴を越えて情報が染み出してしまう。**ConvNeXt V2** (Woo et al. 2023) は、スパース畳み込み（sparse convolution）と Global Response Normalization（演習問題 4）によってこの隔たりを埋めている。
2. **大域的コンテキスト：**7×7 のデプスワイズフィルタであっても局所的であることに変わりはなく、超大カーネルのデプスワイズ畳み込み (Ding et al. 2022)、変形可能畳み込み（deformable convolution）、および畳み込みとアテンションのハイブリッド積層が最先端のスケールでどう比較されるかは未解決のままである。

7.7.5 まとめ

標準的な ResNet-50 から出発し、ConvNeXt は Swin に着想を得た改良を 1 つずつ適用した。すなわち、300 エポックの現代的なレシピ（+2.7 ポイント）、1:1:3:1 のステージ比、4×4 ストライド 4 のパッチ化ステム、7×7 デプスワイズ畳み込みを先頭に置いた $4\times$ 逆ボトルネック、ブロックあたり単一の GELU と LayerNorm、層スケール、そして独立した 2×2 ストライド 2 のダウンサンプリング層である。これらの変更を組み合わせることで、アテンションを使わずに ImageNet で 82.0% に到達する。64×64 の Fashion-MNIST では、ConvNeXt のボディと LayerNorm はパラメータ数を揃えた ResNet-18 と同等の性能（約 94%）を示す一方、粗い 4×4 パッチ化ステムは約 2 ポイントの低下を招く。

7.7.6 演習問題

1. **デプスワイズ層の移動とカーネル拡大の演算量**。本節の Table 7.6 において、「大カーネル」の行の直前では、各ブロックは $c\to4c$ へ 1×1 畳み込みで拡大し、$4c$ チャネルを 3×3 デプスワイズ畳み込みでフィルタリングし、$4c\to c$ へ 2 つ目の 1×1 畳み込みで射影している。「大カーネル」のステップでは、まずデプスワイズ畳み込みをブロックの**先頭**へ移動させ（作用対象が c チャネルになり、$4c$ ではなくなる）、その後にカーネルを 7×7 へ拡大する。
   1. デプスワイズ層の空間ピクセルあたりの積和演算数を、移動前（3×3 を $4c$ に適用）、移動後（3×3 を c に適用）、および 7×7 へ（c 上で）拡大した後のそれぞれについて数えよ。正味の変化を、7×7 へ**元の位置のまま** $4c$ チャネル上で拡大した場合にかかっていたはずのコストと比較せよ。
   2. `ConvNeXtBlock` 全体のピクセルあたり積和演算数（$49c+8c^2$）のうち、7×7 デプスワイズ畳み込みが占める割合は、$c=96$ および $c=384$ においてそれぞれいくらか。
2. **ConvNeXt におけるパラメータの分布**。
   1. 本節の `ConvNeXt` モデル（パラメータ数 3,376,450）について、(i) 7×7 デプスワイズ畳み込み、(ii) 1×1 の拡大層と射影層、(iii) ブロック内部の正規化および層スケールのベクトル、(iv) ダウンサンプリング層、(v) ステムとヘッドがそれぞれ占める正確なパラメータ割合を計算せよ。
   2. `ConvNeXtBlock` と 2 層の `Residual` ブロック（7.4 節、1×1 ショートカットなし）のパラメータ数を、幅 c の関数として表せ。大きな c に対して `ConvNeXtBlock` は漸近的に何倍パラメータ効率が高いか。また、どの設計上の選択がその倍率をもたらしているか。
3. [code] **層スケール対確率的深さ**。層スケールの初期化を $\gamma_0=10^{-6}$（デフォルト）とした場合、$\gamma_0=1$ とした場合、および層スケールを完全に除去した場合のそれぞれについて、Table 7.7 のロードマップのレシピのもとで `ConvNeXt` を訓練し、訓練曲線を比較せよ。初期値 10^{-6} の層スケールと確率的深さ（7.6 節）はいずれも、訓練開始時に深いネットワークを実質的により浅くする。それぞれの機構がどのように働くか、訓練中のいつ有効であるか、そして評価時のネットワークを変化させるかどうかを対比せよ。
4. [code] **Global Response Normalization（ConvNeXt V2）**。ConvNeXt V2 (Woo et al. 2023) は層スケールを取り除き、各ブロック内部の GELU の直後に Global Response Normalization（GRN）を挿入する。特徴マップ $\mathbf{X}$（チャネル数 C）に対し、$g_c=\|\mathbf{X}_c\|_2$ を空間方向の ℓ_2 ノルム（チャネル c に対するもの）とし、$\bar g=\frac{1}{C}\sum_{c=1}^{C}g_c$ とおく。GRN は学習可能なアフィン変換と残差を用いて、相対的な応答により各チャネルを再スケーリングする。

$$\mathrm{GRN}(\mathbf{X})_c=\gamma_c\frac{g_c}{\bar g+\epsilon}\mathbf{X}_c+\beta_c+\mathbf{X}_c, \tag{7.12}$$

ここで $\epsilon = 10^{-6}$ であり、GRN が恒等写像として始まるよう γ, β は 0 に初期化される。GRN を数行のコードで実装し、層スケールの代わりに `ConvNeXtBlock` へ挿入して、Table 7.7 のロードマップのレシピのもとで検証正解率を比較せよ。

5. [code] **デプスワイズカーネルサイズのスイープ**。●Table 7.7 のレシピのもとで、3×3、7×7、および 11×11 のデプスワイズカーネル（パディングはそれぞれ 1、3、5）を用いて `ConvNeXt` を訓練し、パラメータ数、積和演算数、エポックあたりの実時間、および検証正解率を比較せよ。7×7 から 11×11 にしたとき、エポックあたりの実時間が総積和演算数から予想されるよりもはるかに急激に増大するのはなぜか。

6. [code] **アーキテクチャとレシピの 2×2 アブレーション**。●Table 7.7 の最初と最後の行は、現代的な AdamW・Mixup・平滑化のレシピのもとで `CompactResNet18` と `ModernConvNeXt` を比較している。7.6 節のレシピ A（モメンタム SGD、ステップ減衰、Mixup およびラベル平滑化なし、ピーク学習率は再調整）のもとで両方のアーキテクチャを訓練し、2×2 のアブレーション表を完成させよ。現代的な訓練レシピは両方のアーキテクチャを同程度に押し上げるか。

7.8 畳み込みネットワークの設計空間

本章で構築してきたネットワークの大半は人手で設計されており、各ステージで少数の構造的ハイパーパラメータを固定していた。すなわち、積み重ねるブロックの数、各ステージが出力するチャネル数、そしてどのブロックトポロジを使うかである。これらの数値はどこから来ており、そこには一般的な原理が存在するのだろうか。これまでのモデルを振り返っても、驚くほど手がかりは少ない。ステージごとのブロック数は、ResNet-18 では $(2, 2, 2, 2)$、ResNet-50 では $(3, 4, 6, 3)$、ConvNeXt-T では Swin から借用して 1 度検証されただけの $(3, 3, 9, 3)$ である。さらに厄介なことに、これらのパラメータは密接に相互作用する。固定された計算予算（compute budget）のもとでは、あるステージを深くすれば幅を狭くせざるを得ず、グループ畳み込み（grouped convolution）を使えば計算資源が浮いてチャネル幅を広げられる。網羅的な探索は問題外である。4 つのステージにわたってステージごとに 4 つの数値があるため、各数値にわずか 2 通りの選択肢を設けるだけでも $2^{16} = 65{,}536$ 通りの異なるネットワークが生じ、それぞれに個別の訓練を実行しなければならない。仮に力任せの探索で 1 つの優れたネットワークを見つけ出したとしても、単一の点推定（point estimate）からは**どの**設計上の選択が重要だったのかも、新しい計算予算や構成ブロックへどう適応させればよいのかもわからない。

ニューラルアーキテクチャ探索（neural architecture search、NAS）(Zoph and Le 2016; Liu et al. 2018) は EfficientNet（7.5 節）を生み出し、単一モデルの選択を自動化したものの、人手による設計と同様に再利用可能な科学的規則ではなく 1 つの点解を返すにとどまる。これに対し、Radosavovic et al. (2020) は根本的に異なる道をとった。共通のテンプレートと少数の制約（constraint）から生成されるネットワークの母集団、すなわち**設計空間**（design space）全体を比較するアプローチである。設計空間からネットワークをサンプリングして短時間だけ訓練し、提案した制約によって母集団全体の**誤差分布**（error distribution）が改善されるかを調べることで、単純で一般化可能な設計規則を抽出し、**RegNet** 族を定義した。以下では、この AnyNet テンプレートを実装し、ImageNet における母集団の実験結果（空間あたり 500 個のモデル）をたどったうえで、最も強力な規則を Fashion-MNIST の 2 つの小規模な母集団で自ら検証し、コンパクトな RegNet を ResNet-18 と直接対決させて訓練する。

7.8.1 AnyNet 設計空間

本章を通じて、現代の畳み込みネットワークは 7.2.3 節で導入したステム・ボディ・ヘッドの 3 部構成テンプレートへと収束してきた。**ステム**は 1 つまたは少数の畳み込みで生のピクセルを処理し、通常は直ちにダウンサンプリングする。**ボディ**は一連の**ステージ**を通じて計算の大半を担い、

各ステージは固定された解像度で同一構造の**ブロック**を繰り返しながら、ステージの境界ごとに空間次元を半分にし、チャネル幅を広げる。そして**ヘッド**は、グローバル平均プーリングと線形層を介して最終的な特徴テンソルをクラスのロジットへと写像する。ResNet の 4 つのステージは入力解像度の $1/4$、$1/8$、$1/16$、$1/32$ で特徴を出力する。このマルチスケールなピラミッド構造が ConvNeXt に至るまで受け継がれてきた大きな理由は、後段の物体検出器やセグメンテーションヘッドが 4 つのスケールすべてを同時に利用するためである（32.5 節（英語版））。

7.8.1.1 人手で設計されたネットワークのステージパラメータ

Table 7.8 に、本章で扱った 4 つの代表的なアーキテクチャを並べて比較する。なお、どの行でもチャネル幅はステージが進むごとにきれいに 2 倍になっているのに対し、ステージの深さには明白な規則が見られない。

Table 7.8: 人手で設計された 4 つのネットワークにおけるステージの深さと幅（ImageNet 版）。

ネットワーク	ステージあたりのブロック数	ステージあたりの出力チャネル数	ブロック
ResNet-18（7.4 節）	2, 2, 2, 2	64, 128, 256, 512	2 つの 3×3 畳み込み
ResNet-50 (He et al. 2016a)	3, 4, 6, 3	256, 512, 1024, 2048	1×1、3×3、1×1 ボトルネック
ResNeXt-50（7.4.5.1 節）	3, 4, 6, 3	256, 512, 1024, 2048	32 グループのボトルネック
ConvNeXt-T（7.7 節）	3, 3, 9, 3	96, 192, 384, 768	7×7 デプスワイズ、逆ボトルネック

Radosavovic et al. (2020) による **AnyNet** 設計空間は、このマクロなテンプレートを固定したまま、ステージのハイパーパラメータを自由に残す（Figure 7.22）。ステムはストライド 2 の単一の 3×3 畳み込みで $c_0 = 32$ チャネルを出力し、その後にバッチ正規化（batch normalization）と ReLU が続く（ImageNet では RGB 画像を、Fashion-MNIST では単一のグレースケールチャネルを読み込む）。ボディは 4 つのステージからなり、ステージ i は d_i 個の ResNeXt ブロック（Figure 7.12）を積み重ねる。各ブロックでは、1×1 畳み込みが c_i チャネルを c_i/b_i へ射影し、3×3 グループ畳み込みが g_i チャネルずつのグループを処理し、2 つ目の 1×1 畳み込みが c_i チャネルへと戻す構造になっており、各畳み込みの後にバッチ正規化を備え、残差ショートカットを持つ（なお原論文の記法では、b_i は**ボトルネック比**（bottleneck ratio）を表すため、7.4.5.1 節で b と表記したボトルネック幅はここでは c_i/b_i となる）。各ステージの先頭ブロックはストライド 2 で空間解像度（spatial resolution）を半分にし、ストライド 2 の 1×1 畳み込みでショートカットを射影する。グローバル平均プーリングと線形分類器がヘッドを形成する。

したがって、各ステージには 4 つの自由度がある。深さ d_i、出力幅 c_i、ボトルネック比 $b_i \geq 1$、および**グループ幅**（group width）g_i である（グループあたりのチャネル数であり、**グループ数**（group count）g（7.4.5.1 節）と混同しないこと。`groups` 引数が `d2l.ResNeXtBlock` で表すのは引き続きこのグループ数である）。4 つのステージ全体では、これら 16 個の自由な数値（最大 16 までの深さ、8 刻みで最大 1024 までの幅、$\{1, 2, 4\}$ のボトルネック比、最大 32 までの 2 のべき乗のグループ幅をサンプリングする）によって、およそ $(16 \cdot 128 \cdot 3 \cdot 6)^4 \approx 2 \times 10^{18}$ 通りの候補アーキテクチャが張られる。すべてを列挙することは不可能だが、ランダムにサンプリングするのは容易である。224×224 の ImageNet 入力に対しては、ステムと 4 つのステージで空間解像度を 5 回半分にし、最終的に $224/2^5 = 7 \times 7$ となる。

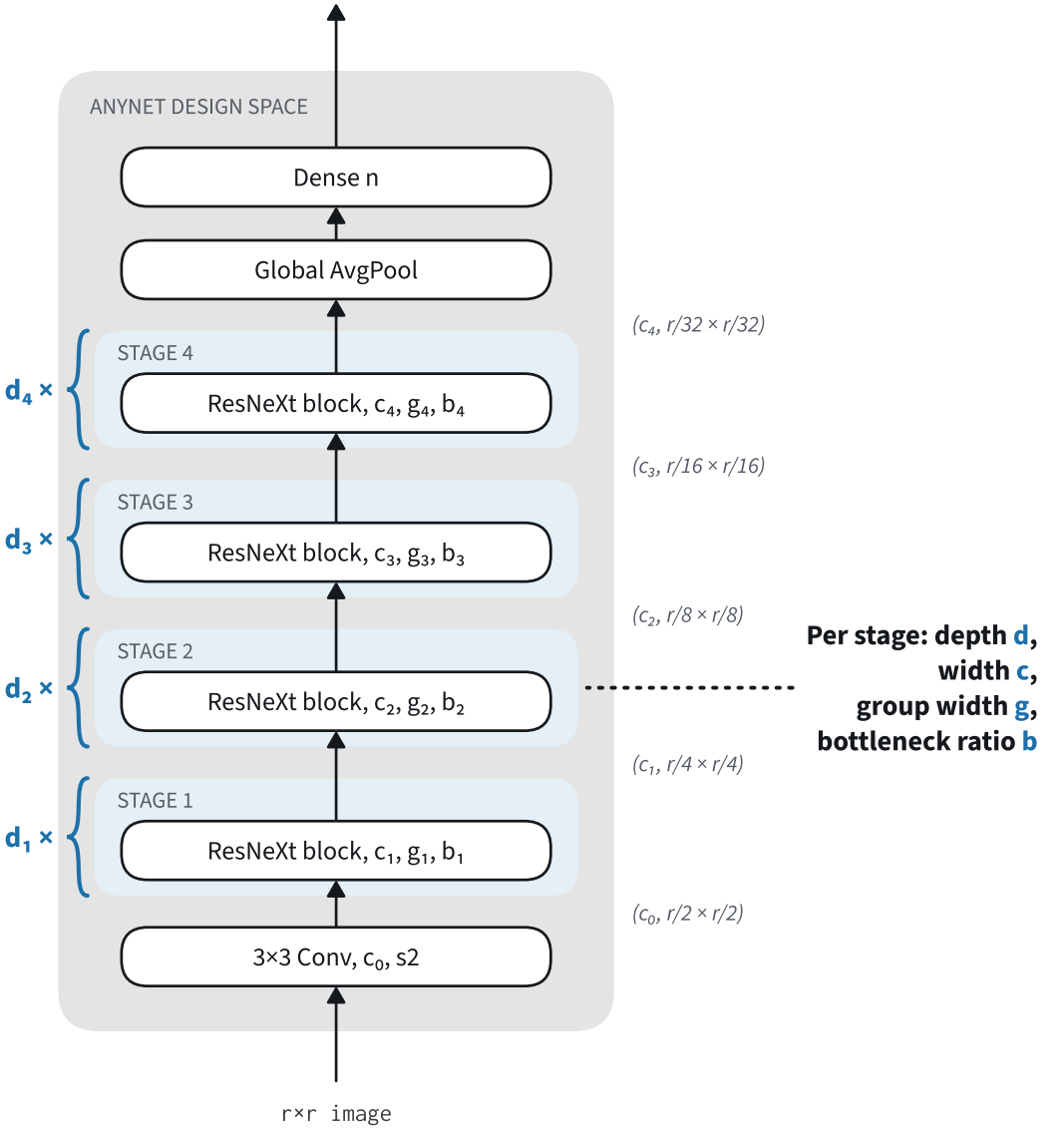


Figure 7.22: AnyNet 設計空間。ステム、4 ステージからなるボディ、およびヘッドで構成される。ステージ i は ResNeXt ブロックを d_i 回繰り返し、各ステージの最初のブロックが解像度を半分にする。図中の注記は $r \times r$ の入力に対する各パート後のチャネル数と解像度を示し、ヘッドは n 個のクラススコアを出力する。ステージごとの自由な選択肢は、深さ d_i、幅 c_i、グループ幅 g_i、ボトルネック比 b_i である。

7.8.1.2 AnyNet の実装

この空間をコードで探索するため、ステージのタプルで駆動される汎用的な AnyNet コンストラクタを構築する。まずは 3 層からなるステムから始めよう。

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=3, stride=2, padding=1,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU())
```

各ステージでは depth 個の d2l.ResNeXtBlock（7.4.5.1 節）を積み重ねる。各ブロックは出力幅 c_i、グループ数、および bot_mul $= 1/b_i$（したがってグループ幅は $(c_i/b_i)/\text{groups}$ に等しい）によってパラメータ化され、先頭のブロックにのみストライド 2 を適用する。

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    blk = []
    for i in range(depth):
        if i == 0:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=2))
        else:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul))
    return nn.Sequential(*blk)
```

ステム、(depth, num_channels, groups, bot_mul) の形式で arch の各要素ごとに作られるステージ、およびグローバルプーリングヘッドを組み立てれば AnyNet が完成する。7.4 節と同様に、すべての畳み込み層と線形層の重みには He 初期化（4.4.2.2 節）を用い、バッチ正規化の直前にある畳み込みではバイアスを省略する。

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.stem(stem_channels))
    for i, s in enumerate(arch):
        self.net.add_module(f'stage{i+1}', self.stage(*s))
    self.net.add_module('head', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))

@d2l.add_to_class(AnyNet)
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')
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

設計空間からサンプリングされたすべてのモデルをまったく同じ訓練プロトコルのもとで評価す

る必要があるため、オプティマイザを直接 AnyNet に組み込む。これは Radosavovic et al. (2020) が用いたプロトコル（および 7.6 節のレシピ A）に合わせたものであり、モメンタム 0.9 の SGD（7.5 節、第 9 章）、結合型重み減衰（coupled weight decay、2.7 節）、そして本章で共通して用いている d2l.cosine_warmup スケジュール（7.1 節、num_steps の最初の 10% で lr までウォームアップし、半コサイン曲線に沿って 0 まで減衰させる）からなる。原論文では ImageNet の 100 エポック訓練に対して重み減衰を 5×10^{-5} に設定しているが、ここでのより短い Fashion-MNIST の実行では 10^{-3} を用いる。この値は RegNetX32（7.8.3 節）に最適でありながら、ResNet18 でも同等に良く機能した。

```
@d2l.add_to_class(AnyNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                            weight_decay=1e-3)  # Decay added to the gradient
    warmup = self.num_steps // 10
    schedule = torch.optim.lr_scheduler.LambdaLR(
        optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
    return optim, schedule  # The Trainer steps it after every minibatch
```

7.8.2 設計空間の比較

2 つの巨大な設計空間を、それぞれの最良アーキテクチャを 1 つも見つけることなく比較するにはどうすればよいだろうか。Radosavovic et al. (2019) は、短時間だけ訓練した適度な数のサンプルモデルで空間全体を忠実に代表させるため、次の 4 つの作業仮説を置いている。

1. **優れた設計原理は豊富に存在する**。妥当な設計規則を満たすネットワークの多くは高い性能を示す（干し草の山の中に良質な針が多数ある）ため、アーキテクチャ上の**分布**を最適化することには意味がある。
2. **短時間訓練のランキングは転移する**。短時間の訓練後の検証正解率は、収束後の最終的な正解率とよく相関する（これにより原論文での ImageNet 10 エポック訓練や、以下で行う Fashion-MNIST 1 エポック訓練が可能になる）。
3. **小規模での知見は外挿できる**。小さな計算予算で見つかった規則は、大規模なモデルへスケールアップしても引き続き成り立つ（最後に大規模環境で検証する）。
4. **設計因子はおおむね分離できる**。個々の構造的制約は、深刻な隠れた相互作用を伴わずに順次評価できる。

これら 4 つの仮説はいずれも先験的に保証されているわけではなく（特に 2 番目と 3 番目は強い仮説である）、末尾の演習問題では小規模環境でこれらを直接検証する。

これらの仮説を受け入れたうえで、ある空間からランダムに n 個のネットワークを抽出し、それぞれを短時間訓練して検証誤差（validation error）$e_1, \ldots, e_n$ を記録する。**経験累積分布関数**（empirical cumulative distribution function、CDF）は、各誤差閾値 e に対して、誤差が高々 e に収まるサンプルモデルの割合を与える。

$$\hat{F}(e) = \frac{1}{n} \sum_{j=1}^{n} \mathbf{1}(e_j \leq e). \tag{7.13}$$

CDF の曲線が左上へ高く立ち上がるほど、その空間には正解率の高いネットワークが豊富に含まれている。ある空間の CDF が別の空間の CDF より一様に上側にあるとき、両空間における最良の単一モデルがたまたま一致していたとしても、最初の空間からランダムに抽出したモデルは任意の誤差閾値をより高い確率で下回る。

この視点を用いて、Radosavovic et al. (2020) は約 4 億回の積和演算（multiply-add、400 MFLOPs）で空間ごとに 500 個のネットワークをサンプリングし、それぞれを ImageNet 上で 10 エポック訓練して得られた CDF を Figure 7.23 にプロットしながら、制約なしの AnyNetX$_A$ 空間を 1 つずつ規則で絞り込んでいった。全 4 ステージで単一のボトルネック比 b を共有し（AnyNetX$_B$）、続いて単一のグループ幅 g も共有しても（AnyNetX$_C$）、誤差分布は変わらない（平均誤差は約 49%）。正解率を損なうことなく 16 個の自由なハイパーパラメータのうち 6 個を削減できるため、これ自体が大きな成果である。劇的な飛躍をもたらすのは**ステージ幅**への制約である。非減少の幅（$c_{i+1} \geq c_i$）を課すと母集団の平均誤差は丸 6 ポイント下がって約 43% になるのに対し、一定または縮小する幅では平均誤差がそれぞれ 56% と 63% へ悪化する。さらに非減少の深さ（$d_{i+1} \geq d_i$）を課すと平均誤差はもう 0.5 ポイント下がり、一定または縮小する深さは後れを取る。

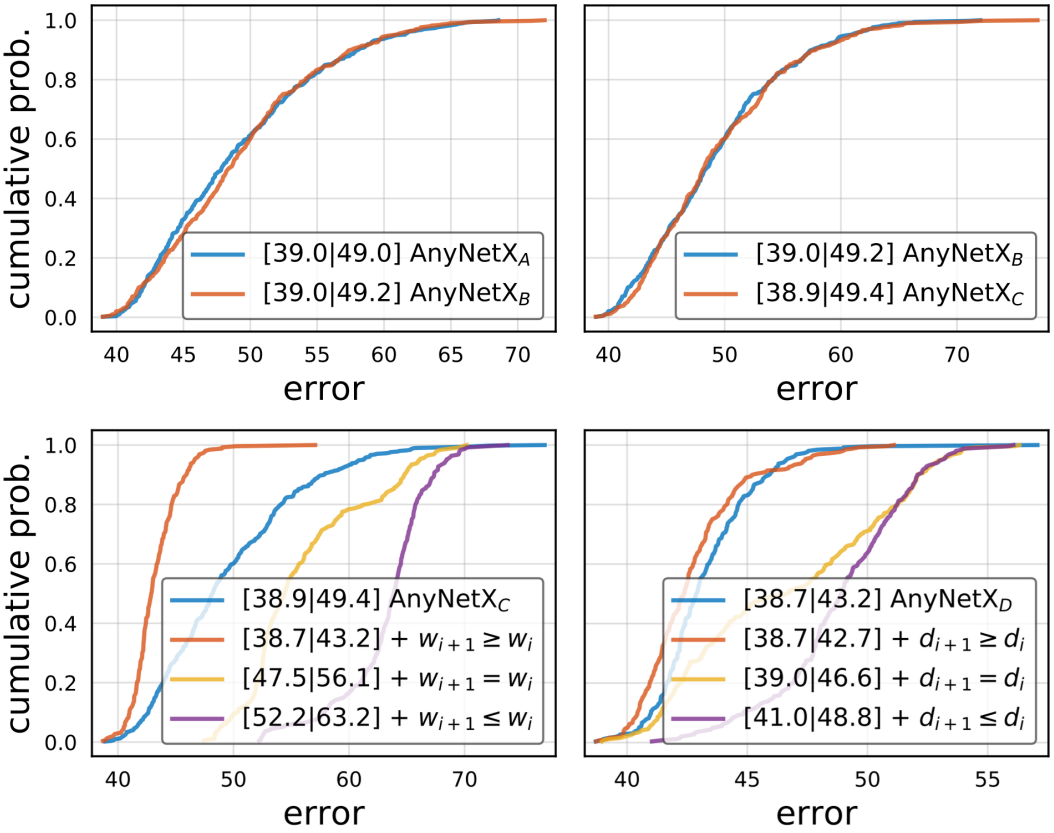


Figure 7.23: Radosavovic et al. (2020) より転載した設計空間の誤差 CDF。各空間から 500 個のネットワークをサンプリングし、それぞれ約 4 億回の積和演算のもと ImageNet 上で 10 エポック訓練したもの。凡例は top-1 誤差の最小値と平均値（パーセント表記）を示し、w_i は幅 c_i に対する原論文の記号である。上段：全ステージで単一のボトルネック比を共有する場合（AnyNetX$_B$）と制約なしの場合（AnyNetX$_A$）の比較、およびグループ幅も単一にした場合（AnyNetX$_C$）。下段：ステージ間で幅が増加・一定・減少する場合の比較、および幅が増加する空間（AnyNetX$_D$）において深さが同様に増加・一定・減少する場合の比較。

経験則はデータセット、計算予算、ブロックの種類に依存するため、最も効果が大きかった「幅の増加対縮小」という単一の規則について、まったく同じ母集団手法を Fashion-MNIST に適用してみるのは有益である。32 チャネルのステムと ResNeXt ブロック（$b = 1$、グループ幅 16）を用いた

2 ステージの設計空間を 2 つ定義し、ステージ深さ $d_i \in \{1, \ldots, 8\}$ と幅 $c_i \in \{16, 32, \ldots, 256\}$ を、一方の空間では $c_1 < c_2$、もう一方の空間では $c_1 > c_2$ という制約のもとで抽出する。原論文と同様に、ステム、すべてのブロック、およびすべての射影ショートカットにわたる積和演算を集計して計算予算を揃える（7.1 節）。

```
def mult_adds(arch, r, stem_channels=32, group_width=16, num_classes=10):
    """Multiply-adds of an AnyNet with k = 1 for one r x r image; arch holds
    (depth, width) for each stage."""
    r //= 2  # The stem halves the resolution
    total, c_in = r * r * 9 * stem_channels, stem_channels  # Grayscale input
    for d, c in arch:
        total += r * r * c_in * c  # First 1 x 1 convolution, before the stride
        r //= 2
        total += r * r * c_in * c  # 1 x 1 convolution of the shortcut
        total += r * r * c * c * (2 * d - 1)  # The other 1 x 1 convolutions
        total += r * r * 9 * group_width * c * d  # Grouped 3 x 3 convolutions
        c_in = c
    return total + c_in * num_classes  # Linear head
```

候補モデルを抽出するには、2 つの深さと互いに異なる 2 つの幅をサンプリングし、空間の制約に従って幅を並べ替え、96×96 画像 1 枚あたりの積和演算が予算である 112M 回の 20% 以内に収まる場合に限り採択する（この予算は 7.8.3 節で訓練する `RegNetX32` モデルに合わせたものである）。

```
def sample_arch(rng, increasing, r=96, budget=112e6):
    """Depths and widths of two stages at about `budget` multiply-adds."""
    while True:
        depths = rng.integers(1, 9, size=2).tolist()
        widths = sorted(rng.choice(np.arange(16, 257, 16), 2, replace=False)
                        .tolist(), reverse=not increasing)
        arch = tuple(zip(depths, widths))
        if abs(mult_adds(arch, r) / budget - 1) <= 0.2:
            return arch

rng = np.random.default_rng()
for increasing in (True, False):
    arch = sample_arch(rng, increasing)
    print(arch, f'{mult_adds(arch, 96) / 1e6:.0f} million multiply-adds')
```

```
((3, 32), (6, 208)) 114 million multiply-adds
((3, 112), (6, 96)) 109 million multiply-adds
```

計算量を揃えた条件では、幅が減少するモデルは初期の高解像度な特徴マップにチャネルを費やし、出力側に向かうにつれて幅を狭めていく。以下では、サンプリングした各アーキテクチャを `AnyNet` のオプティマイザを用いて 1 エポック（7.8.3 節における本訓練の 10 分の 1）だけ訓練し、その検証誤差を記録する。

```
def population_errors(rng, increasing, data, n, num_epochs=1):
    """Validation errors of n networks sampled from one design space."""
    errors = []
    for _ in range(n):
```

```
        arch = sample_arch(rng, increasing)
        model = AnyNet(tuple((d, c, c // 16, 1) for d, c in arch), 32)
        model.num_steps = num_epochs * len(data.train_dataloader())
        trainer = d2l.Trainer(max_epochs=num_epochs, num_gpus=1)
        trainer.board.display = False  # We plot the errors ourselves
        trainer.fit(model, data)
        errors.append(1 - float(trainer.history['val_acc'][-1]))
    return np.array(errors)
```

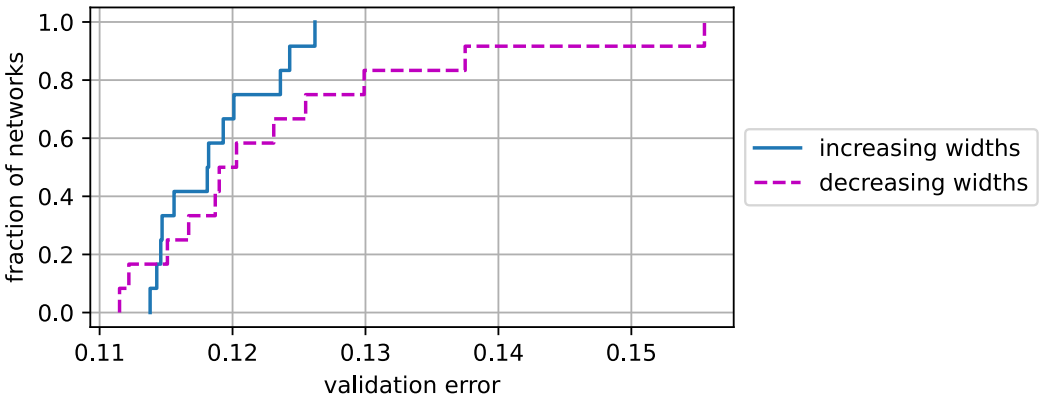
空間ごとに 12 個のモデルを抽出して訓練する（合計 24 回の 1 エポック訓練となる）。

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
spaces = {'increasing widths': True, 'decreasing widths': False}
errors = {name: population_errors(rng, increasing, data, 12)
          for name, increasing in spaces.items()}
```

経験 CDF Equation 7.13 をプロットすると、ソートされた各検証誤差の位置で $1/n$ ずつ跳び上がる階段関数が得られる。

```
def ecdf(e):
    """Corners of the step function of the empirical CDF of e."""
    e = np.sort(e)
    return np.repeat(e, 2), np.repeat(np.arange(len(e) + 1) / len(e), 2)[1:-1]

X, Y = zip(*(ecdf(e) for e in errors.values()))
d2l.plot(list(X), list(Y), xlabel='validation error',
         ylabel='fraction of networks', legend=list(errors),
         legend_loc='outside', figsize=(5, 2.5))
for name, e in errors.items():
    print(f'{name}: mean error {e.mean():.3f}, minimum {e.min():.3f}')
```

```
increasing widths: mean error 0.119, minimum 0.114
decreasing widths: mean error 0.124, minimum 0.112
```



ここでは 2 つの CDF は大きく重なっている。繰り返し抽出を行っても、平均誤差は 12% 付近にとどまって差は 1 パーセントポイント未満であり（多くの場合、幅が増加する空間がわずかに優位となる）、最良モデルはいずれの空間でも 11% 前後に着地する。ImageNet では平均誤差にし

て丸 20 ポイントもの価値があった制約が、Fashion-MNIST では 1 ポイント未満へと縮小しており、空間あたり 12 サンプルでは明確に分離できないほど小さい。これは冷静な見方を促すと同時に示唆に富む結果である。まともな畳み込みネットワークならほぼどれでも 1 エポックで約 88% に達する簡単な 10 クラスのデータセットでは、アーキテクチャ上の制約が母集団を引き離す余地はほとんどない。設計規則は決して普遍的な法則ではなく、その効果の大きさは発見元のタスクの複雑さに直接依存するのである。

7.8.3 RegNet

7.8.3.1 設計原理

ImageNet の検証を勝ち残った制約を組み合わせることで、コンパクトな AnyNetX$_E$ 設計空間が定義される。

- 全ステージにわたって単一のボトルネック比 b を共有する。
- 全ステージにわたって単一のグループ幅 g を共有する。
- ステージ幅を非減少とする（$c_i \leq c_{i+1}$）。
- ステージ深さを非減少とする（$d_i \leq d_{i+1}$）。

AnyNetX$_E$ 内で上位の性能を示すネットワークを詳しく調べると、さらに際立ったパターンがもう 1 つ浮かび上がる。入力側から出力側へ向かってブロックに $j = 0, 1, \ldots$ と順に通し番号を振ると、最適なブロック幅は次のような単純な線形ランプに従う。

$$u_j = u_0 + aj, \tag{7.14}$$

ここで、初期幅 u_0 と傾き $a > 0$ がそのパラメータである。1 つのステージ内のすべてのブロックは同じ幅を共有するため、RegNet は Table 7.8 に示した人手設計ネットワークの倍々になる幅と同様に、各 u_j を等比グリッド上へ量子化（quantization）し、同じ量子化幅に丸められた連続するブロックを 1 つのステージにまとめてきれいな階段構造を作る。Figure 7.24 は RegNet の 1 例における階段構造を示しており、問題 2 では別の例を実際に計算する。興味深いことに、著者らは $b = 1$（内部でのボトルネック圧縮を一切行わない構成）が ResNet-50 や ResNeXt-50 の $b = 4$ のボトルネックを一貫して上回ることも発見した。量子化線形幅と $b = 1$ を組み合わせることで **RegNetX** 族が定義され、16 個の恣意的なステージ数値は計算予算ごとにわずか数個の解釈可能なパラメータへと集約される。

7.8.3.2 Fashion-MNIST における RegNetX32

それでは、これらの規則に従うコンパクトな RegNet 風ネットワークを構築し、ResNet-18 と直接比較してみよう。アップサンプリングされた 96×96 の Fashion-MNIST 画像はもともと 28×28 の縮小画像であるため、4 ステージ（5 回の半減）を実行すると最終盤のステージは 6×6 や 3×3 という極小のグリッドに縮んでしまう。そこで、ここでは 24×24 と 12×12 で動作する 2 ステージ構成を用いる（問題 1 で 4 ステージ構成を検証する）。32 チャネルのステムの後に、増加する深さと幅（ステージ 1 は $d_1 = 4$、$c_1 = 64$、ステージ 2 は $d_2 = 6$、$c_2 = 160$）を人手で選び、$b = 1$ および共通のグループ幅 $g = 16$（グループ数はそれぞれ $64/16 = 4$ と $160/16 = 10$）と組み合わせる。ステムの畳み込み 1 層、3 層構成のブロック 10 個、線形分類器 1 層で合計 32 層となり、これが `RegNetX32` である。

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        stem_channels, group_width, bot_mul = 32, 16, 1
```

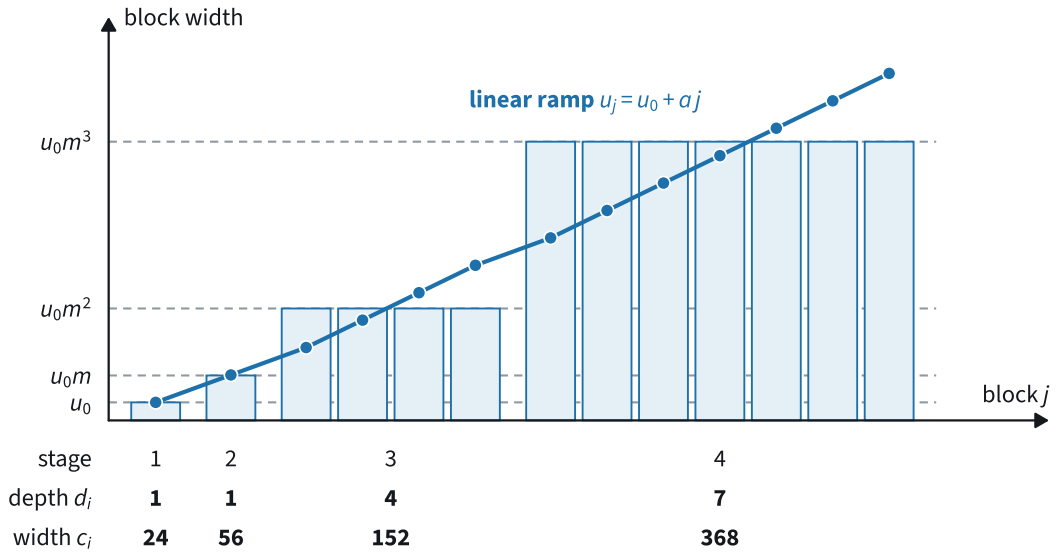


Figure 7.24: 13 ブロックからなるネットワーク RegNetX-200MF における量子化線形幅。線形ランプ $u_j = u_0 + aj$ 上の各ブロックの幅（点）は等比グリッド $u_0, u_0m, u_0m^2, \ldots$（破線）の最も近い水準へ丸められ、同じ水準に属する連続したブロックが 1 つのステージを形成する。$u_0 = 24$、$a = 36.44$、$m = 2.49$ とすると深さ 1、1、4、7 の 4 つのステージが得られ、8 の倍数へ丸めた後の各ステージの幅は 24、56、152、368 チャネルとなる。

```
        depths, channels = (4, 6), (64, 160)
        super().__init__(  # The number of groups is c_i / g
            tuple((d, c, c // group_width, bot_mul)
                  for d, c in zip(depths, channels)),
            stem_channels, lr, num_classes)
        self.num_steps = num_steps  # Length of the learning-rate schedule
```

`layer_summary` を確認すると、チャネル数が増えるにつれて空間解像度が $96 \to 48 \to 24 \to 12$ ときれいに段階的に縮小していくことがわかる。

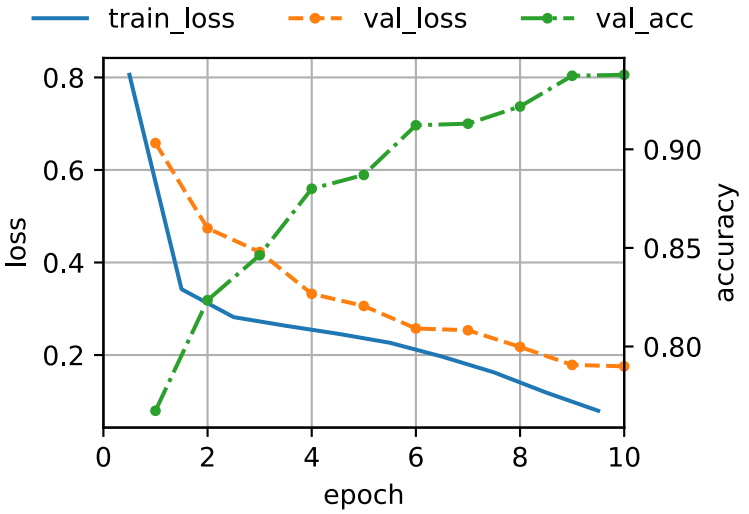
```
RegNetX32().layer_summary((1, 1, 96, 96))
```

```
Sequential            (1, 32, 48, 48)
Sequential            (1, 64, 24, 24)
Sequential            (1, 160, 12, 12)
Sequential            (1, 10)
```

7.1 節と同様に演算数を集計すると、`RegNetX32` は 96×96 画像 1 枚あたり 1 億 1200 万回の積和演算を実行する。これは ResNet-18 の 3 億 1900 万回と比べてほぼ 3 分の 1 の少なさである。

`RegNetX32` を 96×96 の Fashion-MNIST 上で、バッチサイズ 128、`AnyNet` のモメンタム付き SGD オプティマイザ、ピーク学習率 0.1（1 エポックのウォームアップ付き）の設定により 10 エポック訓練しよう。

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = RegNetX32(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



10 エポックの訓練が完了したら、`d2l.num_params(model)` を呼び出して `RegNetX32` の総パラメータ数（parameter count）を最終検証正解率 `trainer.history["val_acc"][-1]` とともに表示する。

```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

```
519,658 parameters, validation accuracy 0.938
```

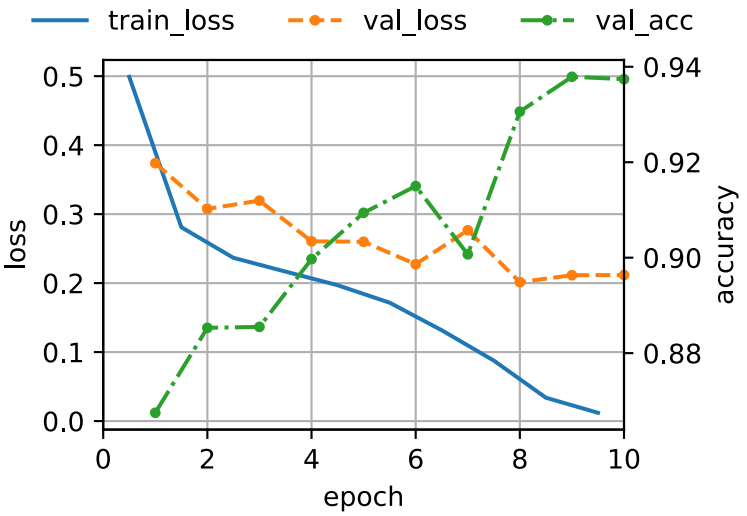
比較条件を厳密に揃えるため、ResNet-18 もまったく同じ `AnyNet` オプティマイザのもとで訓練する。ResNet-18 は同じステム・ボディ・ヘッド構造を共有しているため、単に `AnyNet` をサブクラス化し、`stem` と `stage` を `d2l.Residual`（7.4 節）でオーバーライドして各ステージに `(depth, width, stride)` を渡すだけでよい。

```
class ResNet18(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        super().__init__(((2, 64, 1), (2, 128, 2), (2, 256, 2), (2, 512, 2)),
                         64, lr, num_classes)
        self.num_steps = num_steps

    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=7, stride=2, padding=3,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
    def stage(self, depth, num_channels, stride):
        blk = [d2l.Residual(num_channels, use_1x1conv=stride > 1,
                            strides=stride)]
        blk += [d2l.Residual(num_channels) for _ in range(depth - 1)]
        return nn.Sequential(*blk)
```

それでは、まったく同じデータ上で ResNet18 を 10 エポック訓練する。

```
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



先ほどと同様に d2l.num_params(model) と最終検証正解率 trainer.history["val_acc"][-1] を出力し、ResNet18 を RegNetX32 と直接比較できるようにする。

```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

```
11,175,370 parameters, validation accuracy 0.937
```

どちらのアーキテクチャも検証正解率は 94% 付近（複数回の実行で 93.7〜94.2%）に達するが、パラメータ数は ResNet-18 の 11,175,370 個に対して RegNetX32 はわずか 519,658 個であり、重み数は 5% 未満、積和演算も 3 分の 1（112M 対 319M）にすぎない。この結果を正しく理解するには、次の 3 つの点に留意する必要がある。

1. **アーキテクチャとオプティマイザの結合：**モメンタムも重み減衰も使わない素の SGD（7.4 節）では、ResNet-18 が約 93% に達するのに対し RegNetX32 は約 91% まで低下する。RegNetX32 が肩を並べるのは、RegNet の設計空間が調整された前提であるモメンタム＋重み減衰のプロトコルのもとでのみである。
2. **データセットの上限：**4 ステージの RegNet（RegNetX-400MF、-800MF、-1.6GF、パラメータ数 4.8〜8.3M）へとスケールアップしても、Fashion-MNIST では同じ 93〜94% の上限で頭打ちになる。

3. **演算数対 GPU レイテンシ：**RegNetX32 の 1 エポックには、積和演算が 3 分の 1 であるにもかかわらず ResNet-18 の約 $1.5\times$ の時間がかかる。積和演算が数えるのは算術演算の回数であって実行時間ではない（7.5 節）。RegNetX32 は深さがほぼ 2 倍あり、幅の狭い多数のグループ畳み込み層が画像 1 枚あたり約 $2.5\times$ の活性化値（activation）を書き出すため、算術演算ではなくメモリ転送が律速となる。RegNet の著者らも ImageNet で同じ現象を確認している (Radosavovic et al. 2020)。すなわち、GPU の実行時間は積和演算よりも活性化値の数に強く連動し、活性化値の数が等しい ResNet と比較すると RegNetX モデルはほぼ同等の速度で訓練・実行でき、同程度の積和演算で比較すれば（積和演算に比例して活性化値が増える）EfficientNet より最大 $5\times$ 高速に動作する。

7.8.4 まとめ

設計空間を比較するアプローチは、アーキテクチャ分析の対象を単一の点推定から、パラメータ化されたテンプレート（AnyNet）よりサンプリングした母集団上の誤差分布へと転換させる。空間あたり 500 個の ImageNet サンプルを短時間訓練した検証では、ステージ間で単一のボトルネック比 b とグループ幅 g を共有しても性能は一切損なわれない一方、ステージの幅（階段状に量子化された $u_j = u_0 + aj$）と深さに非減少の制約を課すことで母集団の平均誤差が 49% から 43% へと下がり、RegNetX 族が導かれる。Fashion-MNIST において、RegNetX32 は ResNet-18 の 5% 未満のパラメータ数と 3 分の 1 の積和演算で、その約 94% という正解率に匹敵する。本章全体を通じて見てきたように、少数の基本的な構造プリミティブ（反復可能なブロック、活性化値の正規化、残差ショートカット）と、アーキテクチャによる改善を訓練レシピ（training recipe）による改善から切り分ける規律こそが、現代の視覚バックボーンの礎をなしている。

7.8.5 演習問題

1. [code] **予算を揃えた 4 ステージ構成**。RegNetX32 は 96×96 の Fashion-MNIST に対して 2 つのステージを用いている。ステージ間で幅と深さを非減少に保ち、かつ総パラメータ数を RegNetX32 の 10% 以内に収めつつ、これを Figure 7.22 の 4 ステージ構成へ拡張せよ。検証正解率と 1 エポックあたりの実行時間を RegNetX32 と比較し、4 つの各ステージが動作する特徴マップの空間解像度を述べよ。

2. **量子化線形幅**。@Radosavovic.Kosaraju.Girshick.ea.2020 は、d ブロックからなるネットワークのブロック幅を線形ランプ $u_j = u_0 + a\,j$（$j = 0, \dots, d-1$、7.8.3 節）でパラメータ化し、次の式によって公比 m の等比グリッドへ各ブロックを丸める。

$$s_j = \text{round}\left(\frac{\log(u_j/u_0)}{\log m}\right), \qquad \tilde{u}_j = u_0\, m^{s_j}, \tag{7.15}$$

   そして $\tilde{u}_j$ が等しいブロックをまとめて単一のステージとする。

   1. Figure 7.24 には RegNetX-200MF の結果を示した。$d = 22$、$u_0 = 24$、$a = 24.48$、$m = 2.54$（原論文の記号では w_0, w_a, w_m）を持つ RegNetX-400MF について同じ計算を行い、ステージ数、各ステージの深さ d_i、および最も近い 8 の倍数に丸めたステージ幅を求めよ。
   2. ステージ数と階段近似 $\tilde{u}_j \approx u_j$ は、$m \to 1$ のときと m が大きくなるときとでそれぞれどのように振る舞うか。
   3. (a) で丸めたステージ幅のうち、共通のグループ幅 $g = 16$ で割り切れないものはどれか。また、それらはどのように調整すべきか。

3. [code] **RegNetY（Squeeze-and-Excitation の追加）**。RegNetY は Squeeze-and-Excitation ゲート SE（7.5.2.3 節、縮小比 $r = 4$）を、各 ResNeXtBlock 内の 3×3 グループ畳み込みの直後に挿入する (Radosavovic et al. 2020)。

1. RegNetY32 を RegNetX32 と同じステージ構成で構築せよ。式 $2c^2/r$ を用いて追加されるパラメータ数を予測し、d2l.num_params と照合して検証せよ。
2. RegNetY32 を 96×96 の Fashion-MNIST 上で訓練し、検証正解率と 1 エポックあたりの実行時間を RegNetX32 と比較せよ。積和演算の増加がほぼゼロであるにもかかわらず、SE によって無視できない実時間が増えるのはなぜか。

4. [code] **RegNet の規則を破る。**●パラメータ数をほぼ揃えたまま RegNetX32 の派生モデルを 3 つ実装し、それぞれが AnyNetX$_E$ の規則（7.8.3 節）を 1 つずつ破るようにせよ。すなわち、(i) 減少するステージ深さ（$d_1 > d_2$）、(ii) ステージ間で不均一なグループ幅、(iii) ステージ間で異なるボトルネック圧縮 $b_i > 1$ である。各派生モデルを Fashion-MNIST 上で 3 回ずつ訓練して RegNetX32 と比較し、得られた結果を Figure 7.23 の母集団 CDF と対比せよ。規則違反ごとに人手で選んだ単一のネットワークを検証するだけでは、2 つの経験 CDF を比較する場合よりも証拠として弱いのはなぜか。

5. [extended] **ConvNeXt の設計空間。**●ConvNeXtBlock（7.7 節、拡大比 $t \in \{2, 4\}$ を引数として追加）と ConvNeXt のステムおよびダウンサンプリング層を用いて 2 ステージの AnyNet 設計空間を構築し、ステージ深さ $d_i \in \{1, 2, 4\}$ と幅 $c_i \in \{32, 64, 128\}$ をサンプリングできるようにせよ。

   1. AnyNet におけるステージごとの 4 つの自由度（d_i, c_i, g_i, b_i）のうち、ConvNeXtBlock に直接対応するものがあるのはどれで、デプスワイズ分離構造によって固定されるのはどれか。
   2. 制約なしの空間から 16 個、拡大比を共有した空間（$t_1 = t_2$）から 16 個、さらに $c_1 \leq c_2$ かつ $d_1 \leq d_2$ を課した空間から 16 個のネットワークをサンプリングせよ。それぞれを 96×96 の Fashion-MNIST 上で 2 エポック訓練し、3 つの経験誤差 CDF Equation 7.13 をプロットして、RegNet のどの設計原理が ConvNeXt ブロックにも引き継がれるかを確認せよ。

Chapter 8

系列モデル

ここまで 2 章にわたって画像を扱ってきたが、本書のこのパートの締めくくりとして、ここからは系列データに目を向けよう。これまでに構築してきたすべてのモデルは、属性の表形式ベクトルや 28×28 のピクセル格子のように固定された形状の入力を前提とし、各例を独立に扱っていた。しかし、文、音声録音、動画、あるいはセンサ計測値のストリームは、それとはまったく異なる振る舞いをする。すなわち、長さは例ごとに異なり、各要素は互いに対して、また到着する順序に対して強く依存している。本章全体を通して共通のデータセットとして用いる H. G. Wells の『The Time Machine』の冒頭の一文、「The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us.」（タイム・トラベラー、便宜上彼をそう呼ぶことにするが、彼は難解な事柄を我々に説いていた）を考えてみよう。最後の単語を言い当てるには、その前にある 19 語を順番に読む必要がある。先行するテキストが任意に長くなりうるとき、単一のモデルでどのようにしてその予測ができるだろうか。

Claude Shannon はこの問いを推測ゲームとして捉え (Shannon 1951)、人間の読者に一節を 1 文字ずつ推測させることで、印刷された英語の予測可能性を推定した。**言語モデル**（language model）も同じゲームをプレイする。すなわち、先行するトークンが与えられた下で次の**トークン**（文字、サブワード、単語全体のいずれであってもよい）に確率を割り当て、選ばれた各トークンを次の入力として戻すことで新しいテキストを生成できる。

すでに読み込んだ履歴はステップごとに長くなるため、直近のトークンからなる固定長のウィンドウだけに着目すると、それ以前の情報はすべて捨てられてしまう。そこで**リカレントニューラルネットワーク**（RNN）は、これまでに読み込んだ系列を要約する固定サイズの**隠れ状態**（hidden state）を保持し、トークンを 1 つ処理するたびに同じパラメータでその状態を更新する。現代の大規模言語モデル（large language model）が再帰構造をアテンション（第 10 章）や構造化状態空間モデル（第 12 章）に置き換えている場面でも、サブワードトークン化（subword tokenization）、パープレキシティとバイトあたりビット数（bits per byte）による評価、自己回帰デコーディング（autoregressive decoding）といった本章の中核的な仕組みは、そのまま変わらず引き継がれる。

まず、自己回帰型の系列予測を定式化し、多ステップのロールアウトが 1 ステップ先の予測よりもはるかに速く劣化する理由を確認する（8.1 節）。次に、バイトレベルのバイトペアエンコーディングによって生のテキストをサブワードトークン系列へ変換し（8.2 節）、パープレキシティおよびバイトあたりビット数の評価指標とあわせてカウントベースの n-gram 言語モデルを構築する（8.3 節）。続いて、リカレントニューラルネットワーク言語モデルを定式化してゼロからの実装と高レベル API による実装の両方で構築し（8.4 節）、勾配クリッピング（gradient clipping）によって訓練を安定化させたうえで、学習された表現を Kneser–Ney n-gram と比較する。最後に、

8.5 節では通時的誤差逆伝播法（backpropagation through time）を導出して素朴な RNN の勾配が消失または爆発する理由を説明し、8.6 節ではテキストを生成するための貪欲探索（greedy search）、ビームサーチ、および打ち切りサンプリング手法を検討する。

8.1 系列の扱い

メールクライアントは「Thank you for your」という書き出しを目にすると、その続きとして「help」や「time」といった語を自然に提案する。同様に、1 時間ごとに気温を記録する気象サービスは、直近の観測値をもとに次の値を予測する。いずれの状況でも、良い予測を行うには、それ以前に何が現れたかだけでなく、どの順序で現れたかが重要になる。単語を「your thank for you」と並べ替えれば続く語はまったく変わってしまうし、下降中の気温曲線と上昇中の気温曲線とでは求められる予測が大きく異なる。これまでの章で扱ったモデルは、この構造を直接捉えることができない。固定された形状の単一の入力（ベクトル $\mathbf{x} \in \mathbb{R}^d$）を出力へと写像し、各例を他の例とは独立なものとして扱うからである。文、音声録音、あるいはセンサ計測値のストリームは、この両方の仮定に反している。長さがインスタンスごとに異なり、しかも内部の要素同士が強く依存し合っているためである。

本節では、系列モデリング（sequence modeling）を既知の問題である回帰へと帰着させる。各要素をそれに先行する要素から予測するにあたり、直近の観測値からなるウィンドウを保持するか、それまでに観測したすべてについての逐次的な要約を維持することで、増え続ける履歴を固定サイズの入力へと縮約する。厳密な正解（ground truth）がわかっている設定でこれらの考え方を検証するため、ノイズを含む正弦波に対して固定ウィンドウ上の線形な自己回帰モデル（autoregressive model）を当てはめる。のちに見るように、1 ステップ先を予測するのは容易である一方、モデル自身の予測を入力に戻して何ステップも先を予測するのは格段に難しい。本章の残りの節では、これと同じ原理をテキストへと引き継いでいく。

8.1.1 系列データ

系列（sequence）とは、特徴ベクトル $\mathbf{x}_1, \dots, \mathbf{x}_T$（スカラーの場合は値 $x_1, \dots, x_T$）の順序付きリストである。ここで添字 $t \in \mathbb{Z}^+$ は**タイムステップ**を表し、長さ T は系列ごとに異なってよい。身近な例は枚挙にいとまがない。テキストは文字、サブワード、または単語の系列であり、音声（発話であれ音楽であれ）は波形サンプルやスペクトログラムフレームの系列であり、株価、センサの計測値、患者のバイタルサインといった時系列（time series）では測定値が 1 つずつ順番に届く。

文書全体や入院期間全体といった系列**全体**が、系列上の単一の基底分布から独立に抽出されるという仮定は引き続き置くものの、系列**内部**の要素が互いに独立であると仮定することは到底できない。入院 10 日目に患者へ投与される薬剤は、それまでの 9 日間に起きた経過に大きく依存する。この内部の依存性は厄介どころか、まさに過去を有益な情報源にしている要因そのものである。もし連続する要素が独立であれば、過去の観測値で条件付けても次の要素について何もわからないだろう。また、系列内のすべての位置が同じ周辺分布（marginal distribution）を共有すると仮定する必要もない。文書の結びの段落は書き出しの一文とは大きく異なるのが普通であり、患者の容態も治療の経過とともに回復や悪化へと変化していく。

系列予測のタスクは、いくつかの異なる形で現れる。系列全体に単一のラベルを割り当てる問題（映画レビューの感情分類など）、固定された入力から系列を生成する問題（画像のキャプション生成など）、あるいは入力系列を出力系列へ写像する問題（文の翻訳や音声の書き起こしなど）がある。しかし、いかなる教師ありの目的値を扱うにせよ、まずは同時確率（joint probability）$P(x_1, \dots, x_T)$ を推定することで系列そのものをモデル化してみるのが有益である。各要素がそれに先行する要素に対する目的値として自然に機能するため、この分布の推定には外部ラベルが一切必要ない。さらに、系列に確率を割り当てるモデルは、新しい系列を生成できるだけでなく、

競合する候補を順位付けすることもできる。たとえば、同じ音声クリップに対する 2 つの書き起こしのうち、もっともらしい方を選択できる。

8.1.2 自己回帰モデル

株価指数や 1 時間ごとの気温記録のような数値系列を考えよう。次の測定値に対してどう行動するかを判断する予測者にとって最大の関心事は、これまでに得られたすべての観測値が与えられたもとでの次の値の条件付き分布（conditional distribution）である。

$$P(x_t \mid x_1, \ldots, x_{t-1}). \tag{8.1}$$

プレフィックス $x_1, \ldots, x_{t-1}$ は本章全体を通して登場するため、これを $x_{<t}$ と略記する。条件付き分布全体を推定するのが難しい場合でも、その条件付き期待値（conditional expectation）$\mathbb{E}[x_t \mid x_{<t}]$ だけで十分に有用な要約となり、線形回帰（2.1 節）での当てはめを試みることができる。信号をそれ自身の過去の値に回帰させるモデルは、特徴量と目的値の両方が同じ系列から得られることから、**自己回帰モデル**と呼ばれる。

しかし、ここでただちに問題が生じる。$x_{<t}$ に含まれる条件付け入力の個数は t とともに増大するため、タイムステップごとに特徴量の数が異なり、固定入力のモデルへそのまま渡すことができないのである。この不一致は、履歴を固定サイズの表現へと圧縮する 2 つの一般的な戦略によって解決できる（Figure 8.1）。どちらも自己回帰モデルであることに変わりはなく、過去の情報をどれだけ保持するかという点だけが異なる。

8.1.2.1 固定ウィンドウ

第 1 の戦略は、直近 m 個の観測値 $x_{t-m}, \ldots, x_{t-1}$ のみで条件付け、それ以前の履歴をすべて破棄する方法である。すべての $t > m$ に対して各訓練例はちょうど m 個の特徴量を持つため、線形モデルであれ深層モデルであれ、任意の固定入力アーキテクチャをそのまま適用できる。これは自己回帰の古典的な形式であり、m は**ウィンドウ長**（window length）と呼ばれる。固定ウィンドウは実装が単純であり、各予測が系列の局所的なスライスのみに依存するためタイムステップ間で並列化しやすい。その欠点は明確なホライズンの限界があることで、m ステップより前の過去はモデルからまったく見えなくなる。

8.1.2.2 潜在要約

第 2 の戦略は、固定サイズの逐次的な要約 $\mathbf{h}_t$ によってすべての観測値 $x_1, \ldots, x_t$ の情報を保持し、新しい値が届くたびに $\mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t)$ によって再帰的に更新しながら、その要約から $\hat{x}_{t+1} = f(\mathbf{h}_t)$ によって次の要素を予測する方法である。$\mathbf{h}_t$ は直接観測されないため、このようなモデルは**潜在自己回帰モデル**（latent autoregressive model）と呼ばれる。状態ベクトル $\mathbf{h}_t$ が占めるメモリ量は系列がどれほど長くなっても一定であり、任意の長さの履歴を有界な空間内に要約できる。リカレントニューラルネットワーク（8.4 節）は学習される遷移関数 g を備えた潜在自己回帰モデルであり、第 12 章の状態空間モデル（state space model）も同様である。隠れ状態（hidden state）に何を記憶させ、どのように安定して更新するかが、それらの章における中心的な設計上の問いとなる。

8.1.2.3 条件付き分布から系列へ

どちらの表現を選ぶにせよ、1 ステップの条件付き分布によって系列全体にわたる確率モデルが完全に定まる。**確率の連鎖律**（chain rule of probability）により、任意の同時分布（joint distribution）は、タイムステップごとに 1 つずつ、それ以前のすべての要素で条件付けられた条件付き確率の積へと分解される。

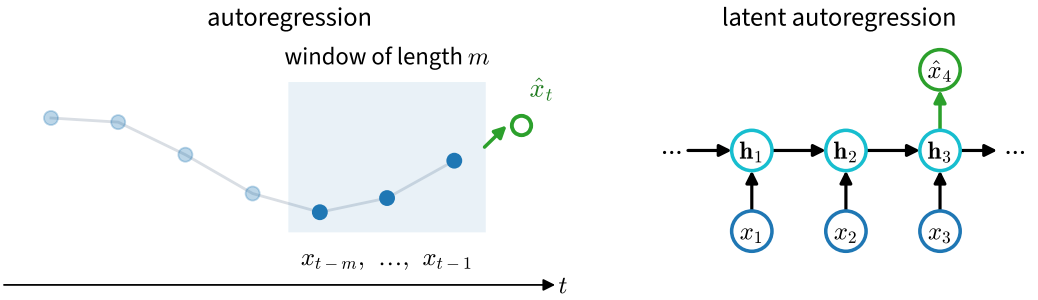


Figure 8.1: 系列モデルに固定サイズの入力を与える 2 つの方法。（左）固定ウィンドウは直近 m 個の観測値で条件付け、残りを破棄する。（右）潜在要約は過去全体を固定サイズに要約する状態 $\mathbf{h}_t$ を引き継ぐ。

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.2}$$

$t = 1$ のときプレフィックス $x_{<1}$ は空であり、最初の因子は単に $P(x_1)$ となる。この恒等式は系列上のあらゆる同時分布に対して厳密に成り立つため、次の要素をモデル化することは、系列全体をモデル化することと数学的に同値である。たとえば 3 語のフレーズであれば、次のようになる。

$$P(\text{the}, \text{cat}, \text{sat}) = P(\text{the})\, P(\text{cat} \mid \text{the})\, P(\text{sat} \mid \text{the}, \text{cat}), \tag{8.3}$$

したがって、モデルがこれら 3 つの因子に確率 0.05、0.01、0.2 を割り当てるなら、この文の同時確率は 10^{-4} となる。実際には、長い系列での数値アンダーフローを避けるため、条件付き確率を直接掛け合わせるのではなく、その対数の和をとって系列にスコアを付ける。新しい系列を生成するには、この手順を逆にたどる。まず x_1 をサンプリングし、次に x_2 を x_1 が与えられたもとでサンプリングし、以降も新たに得られた値を次の条件付き分布へと次々に渡していく。

なお、Equation 8.2 では暗黙のうちに順序を選んでおり、系列を左から右へと分解している。原理的には、添字をどのように置換しても数学的に正しい分解が得られる。しかし因果関係のある領域では、通常は順方向の方がはるかにモデル化しやすい。原因から結果を予測する方が、その過程を逆向きにたどるよりも単純になる傾向があるためである (Peters et al. 2017)。過去から未来を予測することは実務上の目的とも一致するため、本章全体を通して系列は左から右へと分解する。

系列の要素が離散的なトークンである場合、各条件付き分布は語彙（vocabulary）上の分類器となり、$P(x_1, \ldots, x_T)$ を推定することは**言語モデリング**（language modeling、8.3 節）と呼ばれる。本節の残りでは、自己回帰予測の仕組みそのものに直接焦点を当てられるよう、連続値を扱うことにする。

8.1.3 マルコフモデルと定常性

固定ウィンドウを用いることは古い観測値を無視しても問題ないと暗黙に仮定しており、すべてのタイムステップにわたって単一のモデルを当てはめることは根底にある予測規則が時間とともにドリフトしないと仮定している。これら 2 つの仮定はいずれも、確率論と時系列解析における古典的な概念である。

8.1.3.1 マルコフ条件

直近 m ステップでの条件付けが履歴全体での条件付けと同等の予測力を持つとき、その系列は**マルコフ条件**（Markov condition）を満たすという。すなわち、直近の過去が与えられたもとで、未来は遠い過去と条件付き独立である。$m = 1$ のとき、この過程は **1 次**マルコフモデルと呼ばれ、一般には m^{th} 次マルコフモデルと呼ばれる。1 次の条件のもとでは、Equation 8.2 の各因子は直前の要素のみに依存し、同時確率は隣接する対にわたる積へと簡約される。

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}). \tag{8.4}$$

現実世界の系列がマルコフ条件に厳密に従うことはめったにない。自然言語のテキストでは、トークンを追加するにつれて得られる効果は次第に薄れるものの、コンテキストを広げれば有用な情報が得られ続ける。それでもなお、m^{th} 次マルコフ条件を仮定することは実用上不可欠な場合が多い。有限のデータからモデルを計算可能かつ推定可能な範囲に保てるからであり、現代の大規模言語モデル（large language model）でさえ無限の履歴ではなく有界なコンテキストウィンドウ上で動作する。離散系列に対する最も単純なマルコフモデルは、各コンテキストの直後に各後続要素が何回現れるかを数え上げることで $P(x_t \mid x_{t-1})$ を直接推定する。これは、8.3 節で構築する n-gram 言語モデルの原理そのものである。

8.1.3.2 定常性

すべてのタイムステップにわたって単一の予測器を当てはめることは、第 2 の仮定、すなわち先行する要素から各要素を生成する**ダイナミクス**が時間とともに変化しないという仮定に依拠している。時間不変なダイナミクスを持つ過程は**時間斉次**（time-homogeneous）、あるいは機械学習ではより緩やかかつはるかに一般的な呼び方として**定常**（stationary）と呼ばれる。観測される値そのものは大きく変動してもよく、一定に保たれるのは条件付き分布 $P(x_t \mid x_{t-m}, \ldots, x_{t-1})$ である。重要な点として、遷移規則が固定されていても、各要素が同一分布に従うとは限らない。同じ規則が異なるコンテキスト上で評価されるため、文書の末尾は冒頭とは異なる振る舞いを示す。系列に沿ってウィンドウをスライドさせ、単一の系列の異なる部分や多数の系列から例を集めて 1 つの共有訓練セットを構築できるのは、この定常性があるからである。

8.1.4 ノイズを含む正弦波の予測

厳密な正解がわかっている設定でこれらの概念が実際にどう機能するかを確かめるため、合成データ（synthetic data）に単純な自己回帰モデルを当てはめ、1 ステップ先予測と複数ステップのロールアウトを比較してみよう。

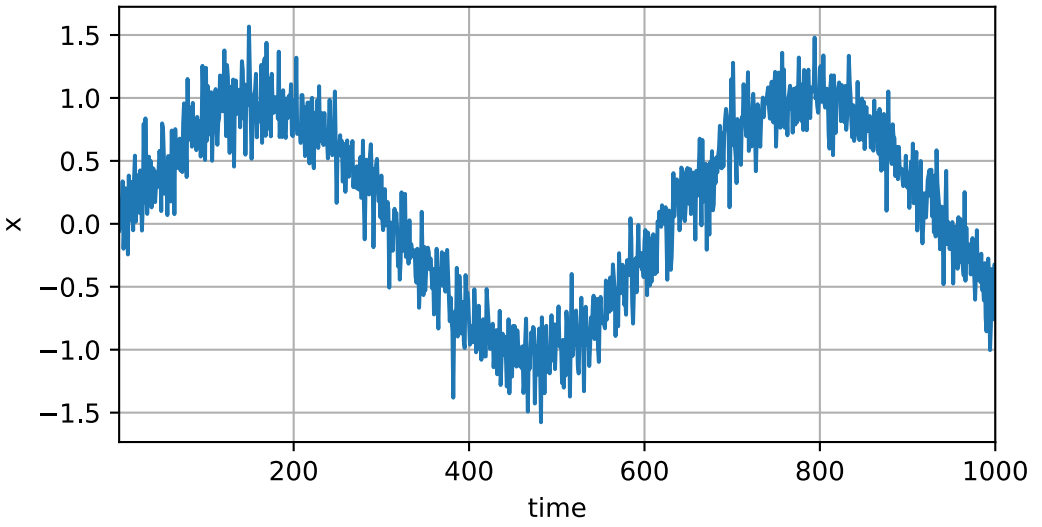
8.1.4.1 データとモデル

$T = 1000$ タイムステップの系列を、ノイズを含む正弦波 $x_t = \sin(0.01\, t) + \epsilon_t$ から生成する。ここで加法ノイズ $\epsilon_t \sim \mathcal{N}(0, 0.2^2)$ は各ステップで独立に抽出される。この周波数では、1 周期は $2\pi/0.01 \approx 628$ ステップに相当する。

```
class Data(d2l.DataModule):
    def __init__(self, batch_size=16, T=1000, num_train=600, m=4):
        self.save_hyperparameters()
        self.time = d2l.arange(1, T + 1, dtype=d2l.float32)
        self.x = d2l.sin(0.01 * self.time) + d2l.randn(T) * 0.2
```

Data をインスタンス化し、data.x を data.time に対してプロットすると、ノイズを含んだ振動が約 1.5 周期分表示される。

```
data = Data()
d2l.plot(data.time, data.x, 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



この系列を教師ありの訓練例へ変換するため、ウィンドウ長 m の固定ウィンドウ戦略を適用する。各例は、特徴ベクトル $(x_{t-m}, \dots, x_{t-1})$ と目的ラベル $y = x_t$ の対からなる。これによって合計 $T - m$ 個の例が得られる。最初の m 個の要素には十分な履歴がないため、ここでは単に破棄する（ゼロパディングもよく使われる代替策である）。lag_features メソッドはラグごとに 1 列を持つ特徴行列全体を一度に構築し、データローダは最初の 600 個の例（正弦波の 1 周期にわずかに満たない長さ）を訓練に使用する。Figure 8.2 に、$m = 4$、$T = 1000$ の設定で得られる表を示す。

```
@d2l.add_to_class(Data)
def lag_features(self):
    """Features (x_{t-m}, ..., x_{t-1}) and labels x_t for t > m."""
    features = [self.x[i : self.T-self.m+i] for i in range(self.m)]
    return d2l.stack(features, 1), d2l.reshape(self.x[self.m:], (-1, 1))

@d2l.add_to_class(Data)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader(self.lag_features(), train, i)
```

予測器には、$m = 4$ 個のラグ特徴量に対する通常の線形回帰を用いる。考えられる限り最も単純な自己回帰モデルといえる。

```
model = d2l.LinearRegression(lr=0.01)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```

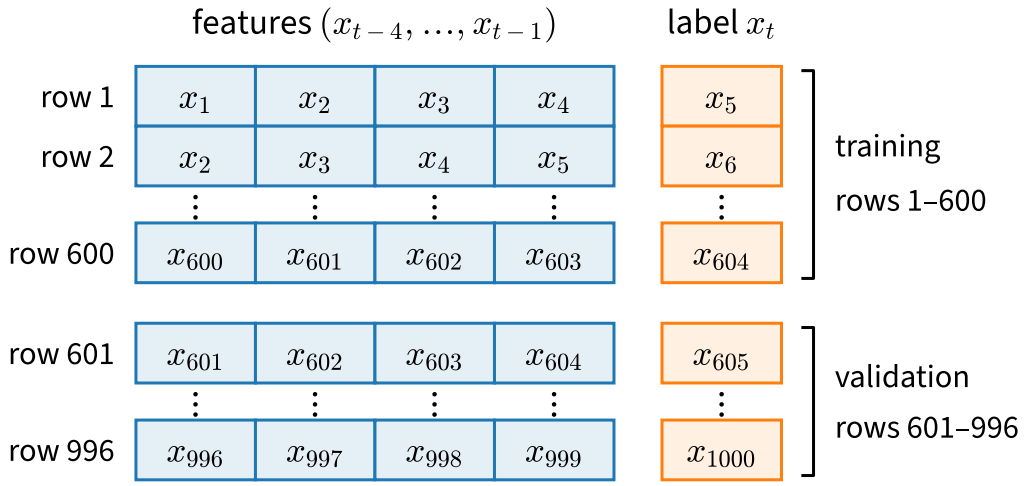
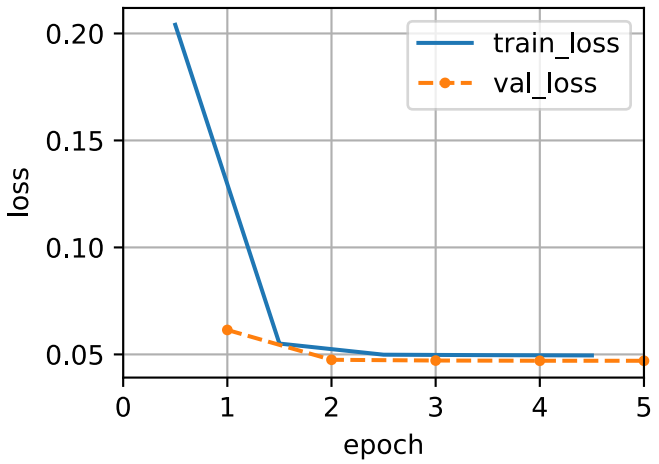


Figure 8.2: $m = 4$、$T = 1000$ におけるラグ特徴量とラベル。第 i 行は 4 つの値 $(x_i, \ldots, x_{i+3})$ とラベル x_{i+4} を対にしており、全 996 行の末尾は x_{1000} となる。最初の 600 行を訓練に、残りの 396 行を検証に使用する。



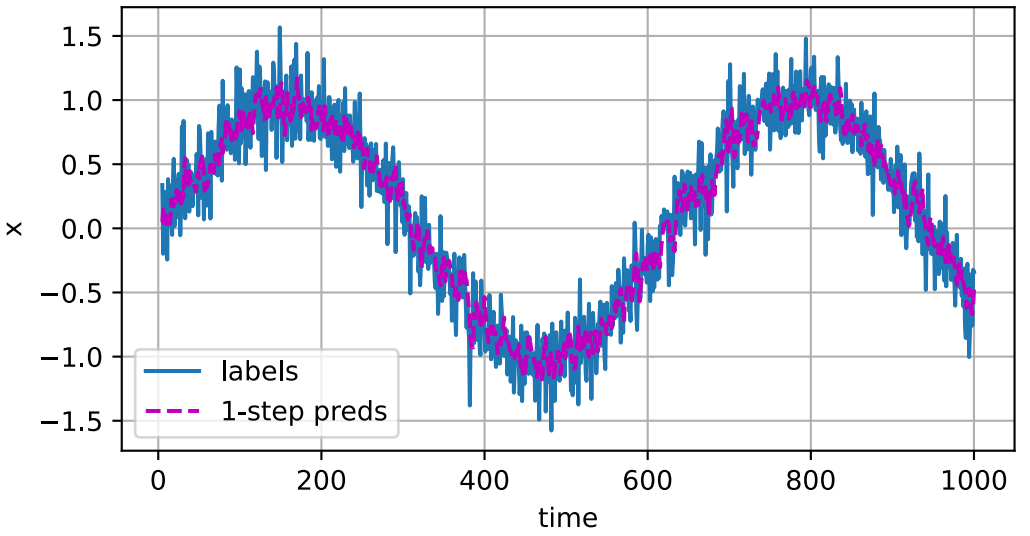
8.1.4.2 1 ステップ先予測

まずは簡単なケースから始めよう。すなわち、$\hat{x}_t$ を、**観測された**直前 m 個の値からすべてのタイムステップ $t > m$ において予測する。

```
features, labels = data.lag_features()
with torch.inference_mode():  # prediction only: record no gradients
    onestep_preds = d2l.numpy(model(features))
```

1 ステップ予測 `onestep_preds` を正解の `labels` に重ねて、全 $T - m = 996$ タイムステップにわたりプロットする。

```
d2l.plot(data.time[data.m:], [labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```



1 ステップ予測は元の波形によく追従しており、訓練分割の外側のデータでモデルが評価されるステップ 600 以降でもその精度は保たれている。

8.1.4.3 複数ステップのロールアウト

複数ステップ先の未来を予測するのは格段に難しくなる。タイムステップ t で観測が途切れ、$\hat{x}_{t+5}$ を予測したいとしよう。$x_{t+1}, \ldots, x_{t+4}$ は観測されていないため、1 ステップ予測器を直接評価することはできない。そこで代わりに、モデル自身の予測をあたかも真の観測値であるかのように入力へ戻し、1 タイムステップずつ先へ進めていく。

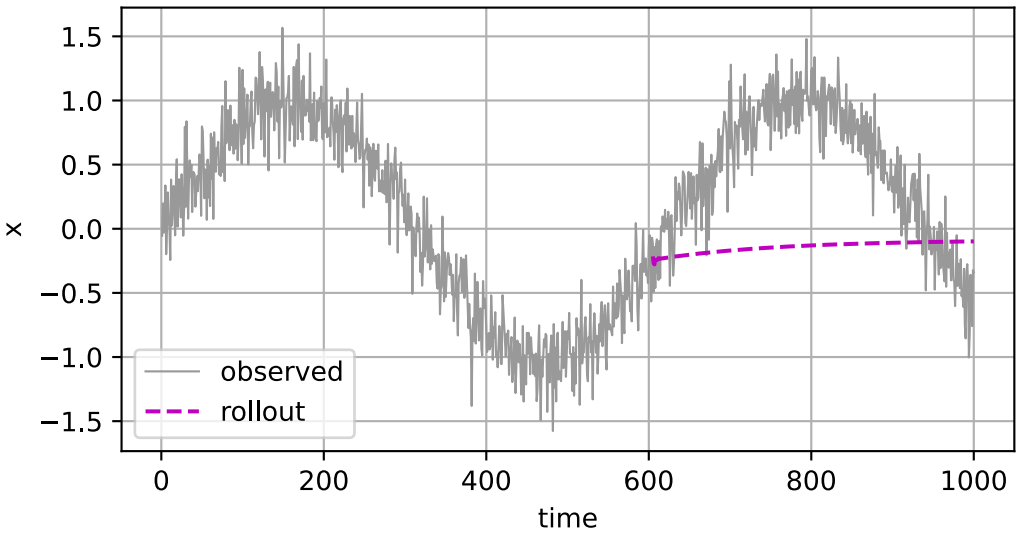
$$\begin{aligned}
\hat{x}_{t+1} &= f(x_{t-3}, x_{t-2}, x_{t-1}, x_t), \\
\hat{x}_{t+2} &= f(x_{t-2}, x_{t-1}, x_t, \hat{x}_{t+1}), \\
\hat{x}_{t+3} &= f(x_{t-1}, x_t, \hat{x}_{t+1}, \hat{x}_{t+2}), \\
\hat{x}_{t+4} &= f(x_t, \hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}), \\
\hat{x}_{t+5} &= f(\hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}, \hat{x}_{t+4}).
\end{aligned}$$

5 ステップ目以降、モデルはそれ以前の自身の予測のみに頼ることになる。$\hat{x}_{t+k}$ を k **ステップ先予測**と呼び、1 ステップ予測を連鎖的に先へとつないでいく過程を**ロールアウト**と呼ぶ。以下のコードは、訓練分割で使われた最後の値である $t = 604$ から系列の末尾まで、線形モデルをロールアウトする。

```
multistep_preds = data.x.clone()
with torch.inference_mode():
    for i in range(data.num_train + data.m, data.T):
        multistep_preds[i] = model(
            multistep_preds[i-data.m : i].reshape(1, -1))
```

```
multistep_preds = d2l.numpy(multistep_preds)
```

```
start = data.num_train + data.m
d2l.plot([data.time, data.time[start:]], [data.x, multistep_preds[start:]],
         'time', 'x', figsize=(6, 3))
ax = d2l.plt.gca()
ax.lines[0].set(color='0.6', linewidth=0.8)  # The series: thin and grey
ax.legend(['observed', 'rollout']);
```



ロールアウトは振動にまったく追従できていない。最後に観測されたウィンドウの水準付近から始まったあと、真の波形が上下し続けるのをよそに、緩やかに定数へとドリフトしていく。1 ステップ誤差がこれほど小さいモデルが、なぜここまで徹底的に破綻するのだろうか。学習された重みを調べれば、その理由はただちに明らかになる。

```
w, b = model.get_w_b()
print(f'weights on x[t-{data.m}], ..., x[t-1]:',
      d2l.numpy(d2l.reshape(w, -1)).round(3))
```

```
weights on x[t-4], ..., x[t-1]: [0.25  0.249 0.23  0.254]
```

4 つの重みはいずれも $1/4$ の近くに収まっており、モデルは単に直前 4 個の平均として次の観測値を予測することを学習している。1 ステップ予測においては、これは理にかなった戦略である。正弦波の変化量は 1 ステップあたり高々 0.01 であるのに対し、加法ノイズの標準偏差は 0.2 もある。ノイズを含む 4 つの測定値を平均すればノイズ分散は 4 分の 1（0.04 から 0.01）に減少し、緩やかに変化する曲線が 1〜2 ステップ遅れることで生じる統計的バイアス（statistical bias）はごくわずかである。ところが、移動平均（moving average）にそれ自身の出力を入力し続けると、現在の水準を維持することしかできない。しかも重みの和が 1 をわずかに下回るため、その水準はロールアウトの不動点（fixed point）$b/(1-\sum_i w_i)$ へとゆっくり減衰していく。ここで b はバイアスである（2 つの小さな数値の比であるため、厳密な漸近先は実行ごとに異なる）。1 ステップの目的関数では正弦波の曲率を学習してもほとんど有利にならないため、モデルはその曲率

をまったく学習しておらず、長いホライズンにわたるとその欠落が誤差の大部分を占めるようになる。重要な点として、線形性そのものが原因ではない。ノイズがなければ、正弦波は厳密な 2 階漸化式 $x_t = 2\cos(0.01)\,x_{t-1} - x_{t-2}$ を満たすため、2 と -1 付近の重みを持つわずか 2 つのラグだけで振動を無限に再現できるはずである。しかしノイズが存在する状況では、そのような重みはノイズ分散を 5 倍に増幅してしまうため、1 ステップの二乗誤差を最小化すると当てはめは平均化へと押しやられるのである（演習問題 1）。

8.1.4.4 ホライズンに伴う誤差の増大

ホライズン k が大きくなるにつれて予測がどれほど急速に劣化するかを測るため、系列内のすべての有効な位置から同時に出発して k ステップ先予測を計算する。補助関数 `k_step_pred` は開始インデックスごとに 1 つの要素を持つ列のリストを保持する。まず観測された m 個のラグから始め、直近 m 列から計算した各段階の予測を順に追加していき、$(j-1)^{\text{th}}$ の要素に j ステップ先予測が格納されたリストを返す。

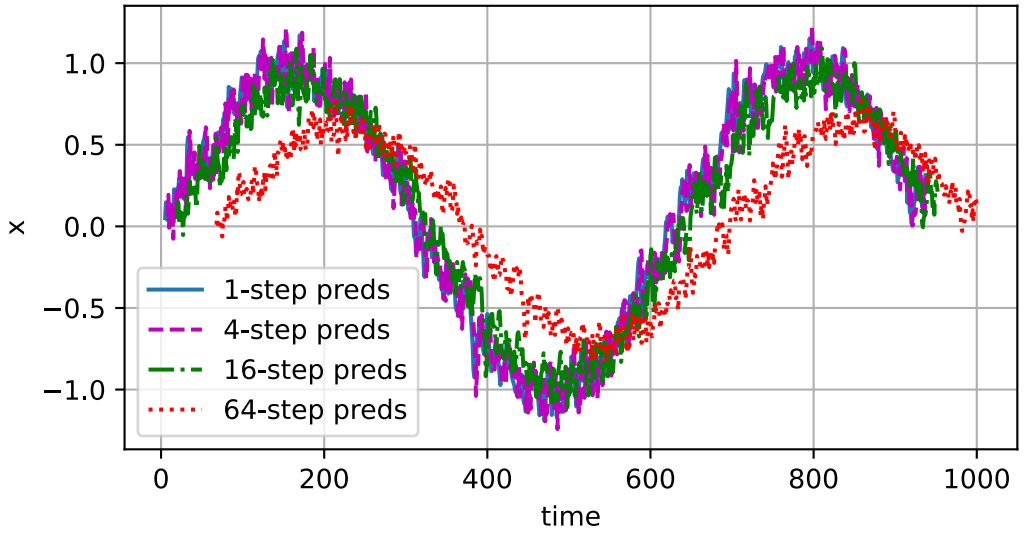
```
def k_step_pred(model, x, m, k):
    """Predictions 1, 2, ..., k steps ahead from every start position."""
    n = len(x) - m - k + 1                      # number of start positions
    columns = [x[i : i + n] for i in range(m)]  # observed lags
    for j in range(k):
        preds = model(d2l.stack(columns[j : j + m], 1))
        columns.append(d2l.reshape(preds, -1))
    return columns[m:]                          # entry j-1: j steps ahead
```

`k_step_pred` を用いて、訓練済みモデルを最大 $k = 64$ ステップ先まで、`data.x` 内のすべての有効な開始ウィンドウからロールアウトする。

```
steps = (1, 4, 16, 64)
with torch.inference_mode():
    preds = k_step_pred(model, data.x, data.m, steps[-1])
```

ホライズン $k \in \{1, 4, 16, 64\}$ における予測曲線を `preds` から取り出してプロットし、ホライズン長とともに誤差がどのように蓄積するかを確認しよう。

```
d2l.plot([data.time[data.m+k-1 : data.T-steps[-1]+k] for k in steps],
         [d2l.numpy(preds[k-1]) for k in steps], 'time', 'x',
         legend=[f'{k}-step preds' for k in steps], figsize=(6, 3))
```



1 ステップ予測と 4 ステップ予測は系列によく追従しているが、16 ステップ予測では遅れが目立ち始め、64 ステップ予測になると波形を約 64 ステップずらして振幅を目に見えて減衰させたコピーのようになる（縮小の度合いは、重みの和が 1 をどれだけ下回るかで決まる）。系列の分散（平均を予測する自明な定数予測の誤差）をベースラインとして、観測値に対する平均二乗誤差（mean squared error）によってこの劣化を定量化できる。

```
x = d2l.numpy(data.x)
for k in steps:
    err = d2l.numpy(preds[k-1]) - x[data.m+k-1 : data.T-steps[-1]+k]
    print(f'{k:2d} steps ahead: mean squared error {(err**2).mean():.3f}')
print(f'variance of the series: {x.var():.3f}')
```

```
 1 steps ahead: mean squared error 0.048
 4 steps ahead: mean squared error 0.049
16 steps ahead: mean squared error 0.066
64 steps ahead: mean squared error 0.246
variance of the series: 0.496
```

1 ステップ先では、平均二乗誤差は 0.05 付近にある。これは削減不可能なノイズ分散 $0.2^2 = 0.04$ に、4 つの観測値の平均に残るノイズ分散 0.01 を加えた値である。k に伴うその後の増大は、ほぼすべて位相遅れによって説明できる。k ステップ予測は k ステップ前の系列の水準をほぼそのまま再現するため、追加で生じる誤差は正弦波が k ステップの間にどれだけ動くかに等しい。波形の位相 a にわたって平均すると、その変化量の二乗は次の式になる。

$$\frac{1}{2\pi}\int_0^{2\pi} \big(\sin(a + 0.01\,k) - \sin a\big)^2\, da = 1 - \cos(0.01\,k), \tag{8.5}$$

この式の値は 0.001（$k = 4$）、0.013（$k = 16$）、0.20（$k = 64$）となる。これにベースラインの下限である 0.05 を加えると、予測される誤差は約 0.05、0.05、0.065、0.25 となり、実測値とよく一致する。64 ステップ先までに誤差は 1 ステップ誤差の 4〜6 倍に達し、すでに系列全体の分散の約半分に及んでいる。

ここで重要な機微に目を向けよう。この誤差は、暴走的な誤差蓄積から生じたものではない。平均化モデルはフィードバックされる入力を常に減衰させるため、ロールアウトは発散するのではなく定数へと滑らかに減衰していく。k ステップ予測が目的値を外すのは、小さな誤差が指数関数的に増幅したからではなく、真の波形が先へ進んでしまったからである。より一般に、1 ステップの目的関数のみで厳密に訓練されたモデルは、1 ステップ先の予測に役立つ特徴しか学習しない。多数のステップにわたってロールアウトしたときの性能は真の基底ダイナミクスを捉えられているかどうかに左右されるが、1 ステップ損失が小さいからといってそのダイナミクスが学習された保証にはならない。

複数ステップのロールアウトは、この減衰型の線形モデルには見られない第 2 の、より広く知られた失敗モードにも直面する。すべての予測誤差がそれ以降の全ステップにおける入力の一部となるため、摂動を減衰させるのではなく増幅してしまうモデルは、データ分布（data distribution）から次第に大きくドリフトしうるのである。言語モデリングでは、運悪くサンプリングされた数個のトークンによって生成器が訓練時には一度も現れなかった不自然な言い回しへと誘導され、話題から逸れたり反復ループに陥ったりすることがある。学習された物理シミュレータが長い軌跡にわたって安定性を失うのも、ほぼ同じ理由による。この不一致を緩和するには、大きく分けて 2 つの戦略が有効である。訓練時には、正解の履歴とあわせてモデル自身の予測にも触れさせることで、不完全なプレフィックスが分布外（out-of-distribution）にならないようにできる。推論時には、各予測分布を具体的な出力へと変換するデコーディング規則を工夫し、ロールアウトを脱線させる低確率な選択を抑制できる。テキストにおけるこの話題には 8.6 節で改めて立ち返るが、その前にまず、生のテキストを系列モデルが処理できる表現へと変換する必要がある。

8.1.5 まとめ

自己回帰モデルは、直近 m 個の観測値からなる固定ウィンドウ（n-gram の背後にある原理）、または過去全体を要約する有界サイズの潜在状態（リカレントネットワークの背後にある原理）のいずれかを用いて各要素を先行する要素から推定することで、系列モデリングを教師あり予測へと帰着させる。確率の連鎖律により、1 ステップの条件付き分布を掛け合わせることで、系列のスコアリングと生成の両方が可能な完全な確率モデル（probabilistic model）が得られる。すべてのタイムステップにわたって単一のモデルを当てはめることは、履歴から各要素を生成する条件付き規則が時間とともに変化しないという定常性の仮定に依拠している。最後に、高い 1 ステップ予測精度は高い複数ステップ予測性能を保証しない。1 ステップの訓練は短距離の特徴しか評価しないためである。ノイズを含む正弦波の例では、平均化を行う線形モデルは 1 ステップのノイズ下限に迫る一方で、64 ステップ先では 4〜6 倍もの誤差を被った。これは、ロールアウト中に誤差が増幅するモデルを悩ませる累積的なドリフトを考慮に入れる前の段階でもすでに生じる現象である。

8.1.6 演習問題

1. [code] **ウィンドウ長**。`Data` は `m=4` 個のラグ特徴量を与え、`d2l.LinearRegression` がそれを当てはめる。
    1. `m` を増やして再訓練せよ。検証誤差の改善が止まるまでに、過去の観測値はいくつ必要か。
    2. `Data` のノイズ項を取り除き、`m=2` で当てはめよ。得られた 2 つの重みを $2\cos(0.01)$ および -1 と比較し、モデルをロールアウトせよ。次に、標準偏差 0.001 から 0.2 までの範囲でノイズを戻せ。どのノイズレベルで重みは曲率の表現を諦めて平均化へと転じるか。
2. [code] **非線形自己回帰器**。同じ 4 つのラグ特徴量上で `d2l.LinearRegression` を小さな MLP に置き換え、必要に応じてエポック数を増やして再訓練せよ。1 ステップ予測と複数ステップ予測はどのように変化するか。

3. [code] **ホライズンに伴う誤差の増大**。各 k について、`k_step_pred(model, data.x, data.m, k)` が返すリストの最後の要素には、`data.x[data.m+k-1:]` に対する k ステップ先予測が格納されている。
    1. $k \in \{1, 2, 4, \dots, 256\}$ について、k ステップの平均二乗誤差を対数軸上の k に対してプロットせよ。曲線はどこで上向きに折れ曲がり、その折れ曲がりは 16 ステップ予測および 64 ステップ予測の減衰とどのように関係しているか。
    2. k ステップの平均二乗誤差が系列の分散を超える最小の k を求めよ。このホライズンを超える予測には、定数予測と比べてどのような価値があるか。
4. [code] **膨張ラグと平均化ラグ**。●`d2l.LinearRegression` の特徴量は 4 つのまま保ちつつ、より過去まで届くようにせよ。1 つはラグ $x_{t-1}, x_{t-2}, x_{t-4}, x_{t-8}$ とし、もう 1 つは 4 つのブロック平均 $\frac{1}{4}\sum_{i=0}^{3} x_{t-4j+i}$ $(j = 1, \dots, 4)$ とする。それぞれの検証誤差を $m = 4$ および $m = 16$ の場合と比較せよ。どちらの構成が有効であり、それはなぜか。
5. [code] **レジームスイッチング**。●`Data` の単一の正弦波を、200 タイムステップごとに $\sin(0.01\,t)$ と $\sin(0.05\,t)$ が交互に切り替わる系列に置き換え、加法ノイズはそのまま残せ。すると訓練例と検証例の両方に 2 つのレジームが含まれることになる。この切り替わり系列は 8.1.3.2 節の定常性の仮定に反している。それでも $m = 4$ の単一の線形モデルで両方のレジームに当てはめられるか、またそれはなぜか。各レジームの訓練誤差と検証誤差を別々に求め、系列上で当てはめが最も悪化する位置を特定せよ。

8.2 テキストからトークンへ

本章でモデル化する系列の大半はテキストである。ここには 8.1 節の合成数値系列にはなかった課題が生じる。ニューラルネットワークは数値のベクトルを扱うのに対し、テキストは生の文字列として与えられるからである。どのような言語モデルを訓練するにせよ、まず文字列をどのように**トークン**（観測 x_t として扱われる最小単位）へ分割し、各トークンをどのように数値インデックスへ対応付けるかを決めなければならない。一見すると地味なこの前処理の選択は、極めて広範な影響を及ぼす。モデルの埋め込み（embedding）層と出力層のサイズ、モデルが読み込むあらゆる文書の系列長（sequence length）、さらには算術のようなタスクの難易度までもがこれによって決まるのである。『The Time Machine』を個々の文字に分割すると、モデルは約 170,000 タイムステップを順に進まなければならなくなる。一方、単語単位に分割すると、約 4,600 種類の異なる単語それぞれに出力クラスが必要になるうえ、訓練時に出現しなかった未知の単語には対応できない。こうした理由から、現代のすべての言語モデルには専用のトークナイザが付属しており、トークナイザはモデル自体の不可分な一部と捉えるのが最善である。

まずは、語彙（vocabulary）サイズと系列長のトレードオフにおいて両極端に位置する、文字レベルのトークン化（tokenization）と単語レベルのトークン化を比較することから始める。続いて、普遍的な基底アルファベットとして生のバイト列へと降り、GPT 系モデルの基盤となっている手法である**バイトペアエンコーディング**（BPE）を用いてデータから直接サブワード語彙を学習する (Sennrich et al. 2016; Radford et al. 2019)。ゼロから構築するこの実装は Python でわずか数十行にすぎないが、GPT-2 の実用トークナイザをトークン単位で完全に再現する。

8.2.1 データセットの読み込み

本章を通して共通のコーパスとして用いるのは、H. G. Wells の『The Time Machine』（30,000 語強の中編小説）である。実世界の事前学習（pretraining）コーパスは数兆トークン規模に及ぶが、このデータセットは自然言語の統計的性質を示すのに十分な豊かさを備えつつ、数秒でトークン化できるほど小さい。以下の `_download` メソッドは、この書籍を取得し、生のテキストを 1 つの文字列として読み込む。

```
import collections
import re
import tiktoken
```

```
class TimeMachine(d2l.DataModule):
    """The Time Machine dataset."""
    def _download(self):
        fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', self.root,
                             '090b5e7e70c295757f55df93cb0a180b9691891a')
        with open(fname) as f:
            return f.read()

data = TimeMachine()
raw_text = data._download()
raw_text[:60]
```

```
'The Time Machine, by H. G. Wells [1898]\n\n\n\n\nI\n\n\nThe Time Tra'
```

古典的な NLP パイプラインでは、句読点を除去し、すべての文字を小文字に統一してから、整形後の文字列をトークンに分割するという積極的な正規化を生テキストに施していた。ここでも、以下で行う文字レベルと単語レベルの比較や、このパイプラインを引き続き使用する後続の節のために、単純な正規化ステップを残しておく。もっとも、このような前処理が大雑把であることは否めない。大文字・小文字の区別（「Bill」と「bill」の違いなど）のような有用な手がかりを捨て去り、どの文字が重要かについて脆い仮定を組み込んでしまうからである。対照的に、本節の後半で構築するバイトレベルのトークナイザは、正規化を一切必要としない。

```
@d2l.add_to_class(TimeMachine)
def _preprocess(self, text):
    return re.sub('[^A-Za-z]+', ' ', text).lower()

text = data._preprocess(raw_text)
text[:60]
```

```
'the time machine by h g wells i the time traveller for so it'
```

8.2.2 文字、単語、バイト

8.2.2.1 文字と単語

系列モデルは 1 タイムステップにつき 1 つのトークンを処理するため、トークンの単位として最も自然な 2 つの選択肢は、個々の文字と単語全体である。まずは文字から順に、両方を見てみよう。

```
@d2l.add_to_class(TimeMachine)
def _tokenize(self, text):
    return list(text)

tokens = data._tokenize(text)
','.join(tokens[:30])
```

```
't,h,e, ,t,i,m,e, ,m,a,c,h,i,n,e, ,b,y, ,h, ,g, ,w,e,l,l,s, '
```

代わりに空白文字で分割すると、単語トークンが得られる。それぞれの方式について、**異なる**トークン種別の数（これが語彙サイズを決める）と、コーパス内の総トークン数（これが系列長を決める）の両方を数えてみよう。

```
words = text.split()
print(f'characters: {len(set(tokens)):>6} distinct, '
      f'{len(tokens):>7} tokens in the corpus')
print(f'words:      {len(set(words)):>6} distinct, '
      f'{len(words):>7} tokens in the corpus')
```

```
characters:     27 distinct,  173428 tokens in the corpus
words:        4579 distinct,   32775 tokens in the corpus
```

正規化したテキストを文字レベルでトークン化する場合、必要な記号はわずか 27 個であり、未知の文字に遭遇することも決してない。しかし、この中編小説は 170,000 以上のタイムステップに引き伸ばされ、主語と動詞の一致のような長距離の依存関係が数十ステップにまたがることになる。単語レベルのトークン化では、同じテキストが約 5 分の 1 のトークン数に圧縮され、各トークンははるかに豊かな意味内容を持つ。しかし、この短い中編小説でさえ、単語に分割するとすでに数千種類の異なるトークン種別が生じ、コーパスが大きくなるにつれて語彙は際限なく膨張する。固有名詞、誤植、語形変化（「traveller」、「travellers」、「travelled」）のすべてが独自の項目を占め、その大半は 1〜2 回しか出現しない（8.3 節）。さらに悪いことに、訓練セットに含まれない単語はすべて、テスト時に**語彙外**（out-of-vocabulary、OOV）となる。Figure 8.3 はこの基本的なトレードオフを示している。語彙を大きくすれば得られる系列は短くなるため、設計上の中心的な問いはこのスペクトラム上のどこで運用するかという点にある。サブワードトークナイザを使えばその動作点を柔軟に選ぶことができ、本節の残りの大部分はその構築に充てる。

8.2.2.2 トークンからインデックスへ

サブワードに進む前に、本書の後続のいくつかの節でも引き続き使用する古典的なパイプラインを完成させておこう。ニューラルネットワークは文字列ではなく整数インデックスを扱うため、文字や単語のパイプラインでは、出現頻度の高い順に各トークン種別へインデックスを割り当て、未知のトークンを予約済みの <unk> 項目に対応付ける**語彙**オブジェクトを用いる。出現回数が `min_freq` 回未満の低頻度トークンも、語彙をコンパクトに保つために枝刈りでき、同様に <unk> へとフォールバックされる。

```
class Vocab:
    """Vocabulary for text."""
    def __init__(self, tokens=(), min_freq=0, reserved_tokens=()):
        # Flatten a 2D list if needed
        if tokens and isinstance(tokens[0], list):
            tokens = [token for line in tokens for token in line]
        # Count token frequencies
        counter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The list of unique tokens, ordered by descending frequency.
        # Reserve <unk> at index 0 so vocab[0] is the unknown token.
        self.idx_to_token = ['<unk>'] + list(reserved_tokens) + [
            token for token, freq in self.token_freqs
```

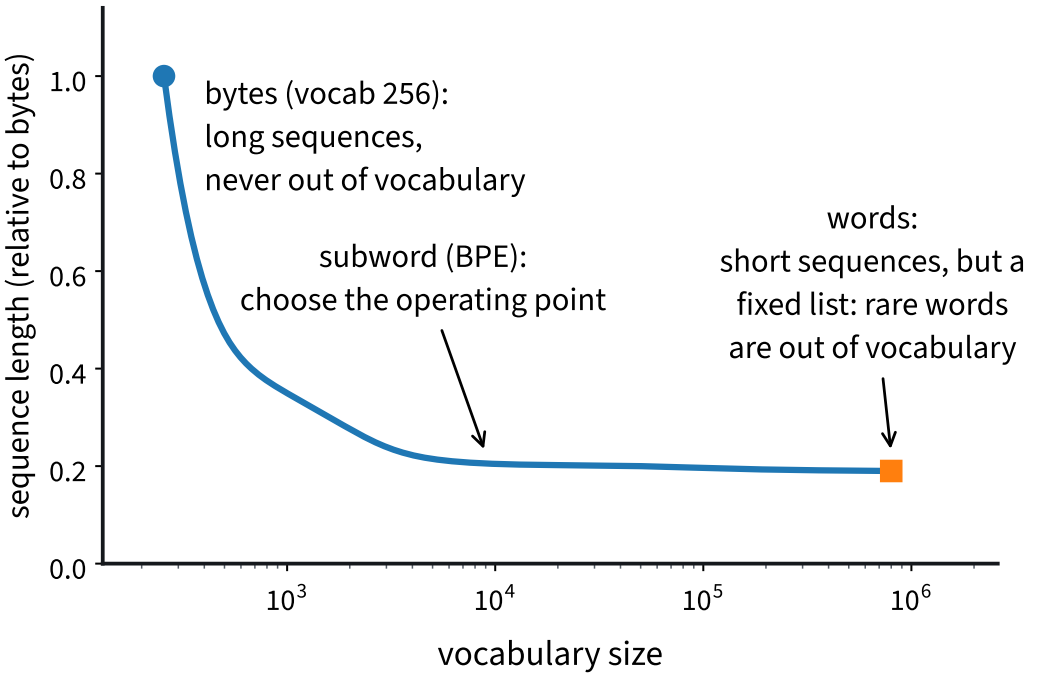


Figure 8.3: 1 本のトレードオフ曲線上に並ぶトークン化の粒度。語彙が大きいほど、同じテキストをより少ないトークンで表現できる。バイト列は決して語彙外にならないが、固定された単語リストでは語彙外が生じる。バイトペアエンコーディングはこの両極端の間を補間する。

```
            if freq >= min_freq and token not in reserved_tokens]
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if isinstance(indices, (list, tuple)) or getattr(indices, 'ndim', 0):
            return [self.idx_to_token[int(index)] for index in indices]
        return self.idx_to_token[int(indices)]

    @property
    def unk(self):  # Index for the unknown token
        return self.token_to_idx['<unk>']
```

Vocab をテストするため、文字のリスト tokens から語彙を構築し、先頭の 10 文字を整数の indices に変換したうえで、vocab.to_tokens によって元のトークンへ戻してみよう。

```
vocab = Vocab(tokens)
indices = vocab[tokens[:10]]
print('indices:', indices)
print('tokens:', vocab.to_tokens(indices))
```

```
indices: [3, 9, 2, 1, 3, 5, 13, 2, 1, 13]
tokens: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm']
```

補助メソッド build は、この古典的なパイプライン全体（前処理、トークン化、インデックス付け）を 1 回の呼び出しにまとめ、整数インデックスからなるコーパスとその語彙を返す。

```
@d2l.add_to_class(TimeMachine)
def build(self, raw_text, vocab=None):
    tokens = self._tokenize(self._preprocess(raw_text))
    if vocab is None: vocab = Vocab(tokens)
    corpus = [vocab[token] for token in tokens]
    return corpus, vocab

corpus, vocab = data.build(raw_text)
len(corpus), len(vocab)
```

```
(173428, 28)
```

8.2.2.3 バイトとしてのテキスト

文字と単語の間を滑らかに補間するには、まずあらゆる文字列を表現できる基底アルファベットが必要になる。驚くべきことに、文字そのものはこの役割に向いていない。Unicode は多様な文字体系、ダイアクリティカルマーク（発音区別符号）、数学記号、絵文字にわたって約 150,000 文

字を定義しているため、平易な英語から一歩踏み出した途端に、文字レベルの語彙はもはや小さなものではなくなってしまう。そこで現代のトークナイザは、もう 1 段階低いレベルへと降りる。コンピュータ上に保存されるあらゆる文字列は、文字エンコーディング（ほぼ例外なく UTF-8）のもとで最終的に**バイト**の系列として表現される。1 バイトが取り得る値はわずか $2^8 = 256$ 通りしかないため、バイトレベルの基底語彙は 256 個の記号しか持たず、語彙外の入力に遭遇することが**決して**ない。どのような言語のどのような文字列であっても、句読点やコード、誤植を含め、余すところなくバイト列へと分解できる。

UTF-8 は、標準的な ASCII 文字には 1 バイトを、あまり一般的でない文字体系や記号には最大 4 バイトを用いる可変長エンコーディングによって、この普遍性を実現している。

```
s = 'Hello, naïve café: 你好  '
print(len(s), 'characters,', len(s.encode('utf-8')), 'bytes')
for ch in 'a é 你  '.split():
    print(repr(ch), '->', list(ch.encode('utf-8')))
n_bytes = len(raw_text.encode('utf-8'))
print(f'The novella: {n_bytes} bytes, {n_bytes / len(words):.1f} per word')
```

```
23 characters, 32 bytes
'a' -> [97]
'é' -> [195, 169]
'你' -> [228, 189, 160]
' ' -> [240, 159, 154, 128]
The novella: 178979 bytes, 5.5 per word
```

この普遍的なアルファベットの代償は系列長である。先ほどの中編小説は約 180,000 バイト（1 単語あたり 5 トークン超）へと膨らみ、さらに「é」のようなアクセント付き文字は 2 バイト、中国語の漢字は 3 バイト、絵文字は 4 バイトを要するため、生のバイト系列では英語以外のテキストがより一層引き伸ばされる。以下で扱う手法はすべて、バイトの普遍性を保ちながら、系列を扱いやすい長さへと圧縮し直すものである。

8.2.3 バイトペアエンコーディング

8.2.3.1 マージ規則

バイトペアエンコーディング（BPE）はもともと、最も頻度の高い隣接バイトのペアを新しい記号に置き換える操作を繰り返すデータ圧縮手法として考案された (Gage 1994)。Sennrich et al. (2016) は機械翻訳（machine translation）向けのサブワード単位を得るために同じ規則を単語内の文字へ適用し、その後 GPT-2 がこれを生のバイト列へと戻した (Radford et al. 2019)。本節で構築するのは、このバイトレベルの変種である。まず 256 個の単一バイト記号から出発し、貪欲に語彙を拡大していく。各反復において、訓練コーパス内のあらゆる隣接トークンの**ペア**を数え上げ、最も頻度の高いペアを 1 つの新しいトークンへとマージする操作を、語彙が目標サイズに達するまで繰り返す。各マージは置き換えた出現回数とちょうど同じ数だけコーパスを短くするため、BPE は貪欲な圧縮アルゴリズムとして機能し、学習された語彙はコーパス内で頻出する断片の辞書を形成する。

アルゴリズムを手作業でトレースしてみると、その動作が明快にわかる。5 種類の単語が計 8 回出現する次のトイコーパスを考えよう。

```
hug pug pun bun hugs hug hug pug
```

生のコーパスの長さは 32 バイトである。最も頻度の高い隣接ペアは (u, g) であり、「hug」、「pug」、「hugs」の中に計 6 回出現するため、最初のマージでは新しいトーク

ン ug が導入される。すべての箇所を ug に置き換えると、最も頻度の高い隣接ペアは (h, ug) となり、以降も同様に処理が続く。

ステップ	最頻出ペア	出現回数	新トークン	コーパス長
0	（バイト列から開始）			32
1	u, g	6	ug	26
2	h, ug	4（同数）	hug	22
3	hug, ' '	3（同数）	'hug '	19

なお、ステップ 2 とステップ 3 には出現回数が同数のペアが存在する。ステップ 2 では (ug, ' ') も 4 回出現し、ステップ 3 では (' ', p) と (' ', hug) も 3 回出現する。3. 標準的な慣例に従い、同数の場合はコーパス内で最も早く出現するペアを優先する。

このトレースから 2 つの重要な性質が直接浮かび上がる。第 1 に、マージは階層的に合成される。hug が h と ug から形成されるように、学習された語彙はバイトアルファベット上の**マージ木**（merge tree）を定義する（Figure 8.4）。これにより、学習済みのマージを順に再生することで、いかなる未知のテキストも分割できる。たとえば「hugs」という単語は hug と s にきれいに分割される一方、低頻度語の「bun」は生の 3 バイトのまま残る。学習されたマージの対象とならなかった部分文字列は、単に個々のバイトへとフォールバックされるのである。第 2 に、ステップ 3 で単語とその**末尾の空白**が結合された点に注目しよう。末尾の空白自体は害がないものの、その後のマージで後続する単語の先頭まで結合されるのを防ぐ仕組みは何もない。実際のコーパスでは、制約のない BPE は複数単語にまたがるトークンを容易に作り出してしまう。この欠点はまもなく解消する。

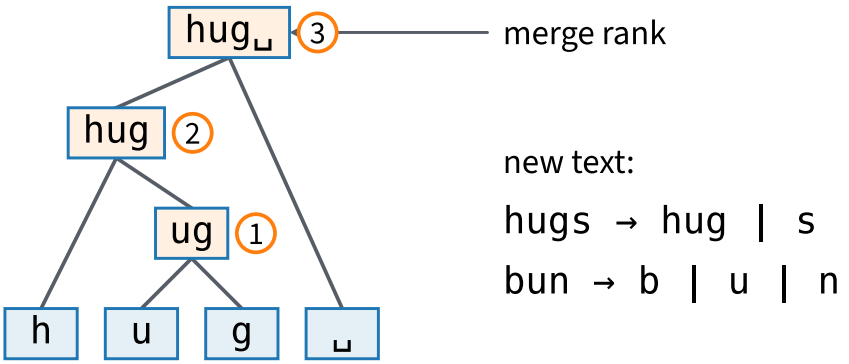


Figure 8.4: トイコーパスで学習された最初の 3 回のマージは、バイトアルファベット上の木を形成する。新しいテキストはこの木を再利用し、「hugs」は 2 トークンになる一方、未知の断片はバイトへとフォールバックされる。

8.2.3.2 実装

BPE アルゴリズムを 3 つの簡潔な要素に分けて実装する。まず、`BPETokenizer` クラスがトークナイザの状態を保持する。トークン ID の 0〜255 は生のバイトを表し、学習された各マージには次に利用可能な整数 ID が順に割り当てられるため、ID はマージの**ランク**も兼ねる（ID が小さいほど早い段階のマージであることを示す）。辞書 `merges` はマージされた ID ペアをその新しい ID に

対応付け、vocab はすべての ID を対応するバイト列に対応付ける。引数 pattern と specials、および byte_ids は本節の後半で登場するため、現時点では無視してよい。また、regex パッケージもインポートする。これは Python 標準の re モジュールを拡張したもので、事前トークン化に必要な Unicode 文字クラス（任意の文字を表す \p{L} など）をサポートしている。

```
import regex

class BPETokenizer:
    """Byte-level BPE tokenizer trained by iterated most-frequent-pair
    merges.

    Token ids 0..255 are raw bytes; learned merges get ids
    256..vocab_size-1; special tokens sit above those and are never
    produced by BPE itself. `pattern` is an optional pre-tokenization
    regex: text is split into chunks and merges never cross chunk
    boundaries.
    """

    GPT2_PATTERN = (r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+"
                    r"| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+")

    def __init__(self, vocab_size=1024, pattern=None,
                 specials=('<pad>', '<bos>', '<eos>')):
        self.vocab_size, self.pattern = vocab_size, pattern
        self.merges = {}                                # (id, id) -> new id
        self.vocab = {i: bytes([i]) for i in range(256)}  # id -> bytes
        self.byte_ids = list(range(256))                # byte value -> id
        self.specials = {s: vocab_size + i for i, s in enumerate(specials)}

    def __len__(self):
        return self.vocab_size + len(self.specials)

    @property
    def pad(self):
        return self.specials['<pad>']

    @property
    def bos(self):
        return self.specials['<bos>']

    @property
    def eos(self):
        return self.specials['<eos>']

    def _chunks(self, text):
        return [text] if self.pattern is None else regex.findall(
            self.pattern, text)
```

訓練とエンコードはいずれも、対象となる隣接 ID ペアのすべての出現箇所をマージ後のトークン ID に置き換えるという単一の基本操作に依存している。補助メソッド _merge は左から右へ走査し、new_id を書き込んで位置を 2 つ進める処理を pair が一致するたびに行い、一致しない場合は ID を 1 つコピーする。その結果、文字列 aaa に対してペア (a, a) は重複して適用されるのではなく、左側で 1 回だけ置き換えられる。

```
@d2l.add_to_class(BPETokenizer)
def _merge(self, ids, pair, new_id):
    """Replace every occurrence of pair in ids by new_id."""
    out, i = [], 0
    while i < len(ids):
        if i + 1 < len(ids) and (ids[i], ids[i + 1]) == pair:
            out.append(new_id)  # Merge the pair and skip past it
            i += 2
        else:
            out.append(ids[i])
            i += 1
    return out
```

訓練では、手作業でトレースしたのと同じループを繰り返す。すなわち、すべての隣接ペアを数え上げ、最頻出ペアをすべての箇所でマージし、得られた規則を記録する。トレースのときと同様、出現回数が同数の場合は最初に出現したペアが優先されるが、これは Python の `max` の挙動そのものである。実用上の最適化として、同一のテキストチャンクは 1 回だけ保持し、そのペア出現回数をコーパス内でのチャンクの出現頻度で重み付けしている。事前トークン化を行わない場合、コーパス全体が重み 1 の単一チャンクとなるが、事前トークン化によってテキストが単語に分割されれば、頻出語は何千回も処理される代わりに 1 回だけ処理される。実用レベルの BPE トレーナはさらに踏み込み、各マージの前後で変化した箇所のみペア出現回数を差分更新するが、ここでは明快さを優先して速度とのトレードオフを受け入れ、各ステップの後に単純に数え直す。

```
@d2l.add_to_class(BPETokenizer)
def train(self, text):
    """Learn vocab_size - 256 merges, most frequent pair first."""
    chunk_freq = collections.Counter(self._chunks(text))
    seqs = [list(chunk.encode('utf-8')) for chunk in chunk_freq]
    for new_id in range(256, self.vocab_size):
        pairs = collections.Counter()
        for seq, w in zip(seqs, chunk_freq.values()):
            for pair in zip(seq, seq[1:]):
                pairs[pair] += w  # A chunk that occurs w times counts w times
        if not pairs:
            break  # nothing left to merge
        pair = max(pairs, key=pairs.get)
        self.merges[pair] = new_id
        self.vocab[new_id] = self.vocab[pair[0]] + self.vocab[pair[1]]
        seqs = [self._merge(seq, pair, new_id) for seq in seqs]
```

トイコーパス上で `BPETokenizer` を 3 回のマージ（`vocab_size=256 + 3`）だけ訓練し、`toy_tok.merges` を確認すると、先ほど手作業でトレースした 3 つのマージステップがそのまま再現される。

```
toy_tok = BPETokenizer(vocab_size=256 + 3)
toy_tok.train('hug pug pun bun hugs hug hug pug')
for (a, b), new_id in toy_tok.merges.items():
    print(f'{toy_tok.vocab[a]} + {toy_tok.vocab[b]} -> '
          f'{toy_tok.vocab[new_id]}')
```

```
b'u' + b'g' -> b'ug'
b'h' + b'ug' -> b'hug'
```

```
b'hug' + b' ' -> b'hug '
```

8.2.3.3 エンコードとデコード

訓練によってマージテーブルが得られる。新しいテキストを**エンコード**するには、生の UTF-8 バイト列から始め、テーブル内に該当する隣接ペアがなくなるまで、最もマージランクの低い隣接ペアを繰り返しマージする。注目すべき点として、エンコード時に参照されるのはマージランクのみであり、コーパスでの出現回数は一切参照されない。つまりエンコードは学習済みテーブルの純粋な関数であり、**トークナイザはそのマージランクによって完全に定義される**。各訓練マージはそれ以前のすべてのマージが適用済みのテキスト上で学習されたため、最もランクの低いマージから順に適用することで、訓練時の過程が忠実に再現される（この性質は演習問題で定式化する）。デコードはさらに単純である。すべてのトークン ID はバイト系列へ直接対応付けられるため、バイトを連結して UTF-8 としてデコードすればよい。この際、生成モデル（generative model）が出力した不完全または不正なバイト系列に対して例外を送出するのではなく穏やかに代替文字へフォールバックできるよう、`errors='replace'` を指定する。

```
@d2l.add_to_class(BPETokenizer)
def _encode_chunk(self, text_bytes):
    ids = [self.byte_ids[b] for b in text_bytes]
    while len(ids) > 1:
        # The lowest-rank merge applicable anywhere in this chunk
        pair = min(zip(ids, ids[1:]),
                   key=lambda p: self.merges.get(p, float('inf')))
        if pair not in self.merges:
            break
        ids = self._merge(ids, pair, self.merges[pair])
    return ids

@d2l.add_to_class(BPETokenizer)
def encode(self, text, allow_special=False):
    if allow_special and self.specials:
        pat = '(' + '|'.join(regex.escape(s) for s in self.specials) + ')'
        ids = []
        for part in regex.split(pat, text):
            if part in self.specials:
                ids.append(self.specials[part])
            elif part:
                ids.extend(self.encode(part))
        return ids
    return [i for chunk in self._chunks(text)
            for i in self._encode_chunk(chunk.encode('utf-8'))]

@d2l.add_to_class(BPETokenizer)
def decode(self, ids):
    specials = {i: s.encode('utf-8') for s, i in self.specials.items()}
    data = b''.join(self.vocab[i] if i in self.vocab else specials[i]
                    for i in ids)
    return data.decode('utf-8', errors='replace')
```

トイトークナイザにおいて、エンコード結果は手作業のトレースと一致し、デコードによって元の文字列がきれいに復元される。

```
print([toy_tok.vocab[i] for i in toy_tok.encode('hugs')])
print([toy_tok.vocab[i] for i in toy_tok.encode('bun')])
toy_tok.decode(toy_tok.encode('hug pug hugs'))
```

```
[b'hug', b's']
[b'b', b'u', b'n']
```

```
'hug pug hugs'
```

8.2.3.4 『The Time Machine』での訓練

実際の散文から BPE が何を学習するかを確認するため、『The Time Machine』で訓練してみよう。トークンのリストや生成された文章は印刷ページの幅を超えることが多いため、まず長い行を見やすく折り返す小さな表示用補助関数を定義し、ここだけでなく本書の以降の箇所でも利用する。

```
import textwrap

def print_wrapped(*args, width=76):
    """Print like `print`, folding each line to the width of a page."""
    for line in ' '.join(str(a) for a in args).split('\n'):
        print(textwrap.fill(line, width, subsequent_indent='    '))
```

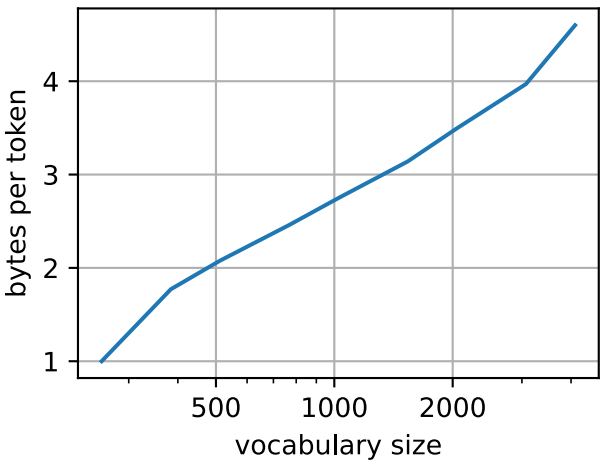
それでは、正規化していないこの中編小説の生テキストから直接、1,024 トークンの語彙を訓練する。初期のマージは頻出する文字ペアを捉えるのに対し、終盤のマージは Wells の散文に登場する単語全体を組み立てるようになる。ただし、以下のエンコードされたサンプルの中に、空白まで含めて 2 単語が結合した複合トークン `"Time Traveller "` が含まれている点に注目してほしい。この現象については次で詳しく調べる。

```
tok = BPETokenizer(vocab_size=1024)
tok.train(raw_text)
merged_ids = list(tok.merges.values())
print('first:', [tok.vocab[i] for i in merged_ids[:8]])
print('last: ', [tok.vocab[i] for i in merged_ids[-8:]])
ids = tok.encode('The Time Traveller smiled. Are you sure?')
print_wrapped(len(ids), 'tokens:', [tok.vocab[i] for i in ids])
```

```
first: [b'e ', b'th', b'd ', b'in', b't ', b's ', b'an', b'er']
last:  [b'appe', b'vo', b'ering ', b'too ', b'door', b'acro', b'clo', b'after ']
13 tokens: [b'The ', b'Time Traveller ', b's', b'mi', b'le', b'd', b'. A',
    b'r', b'e ', b'you ', b'su', b're', b'?']
```

語彙を大きくすると、系列はどれだけ短くなるだろうか。マージには学習された順序どおりに番号が振られるため、同じコーパスで訓練されたより小さな BPE 語彙は、より大きな語彙の**接頭辞**（prefix）となる。すなわち、ID が $|\mathcal{V}|$ 未満のマージだけを残せば、サイズ $|\mathcal{V}|$ で訓練を打ち切ったときに得られるトークナイザが厳密に再現される。したがって、1 回の訓練を実行するだけで、語彙サイズを 256 から 4,096 まで変化させて調べることができる。以下では、中編小説の先頭 10,000 文字に対して 1 トークンあたりのバイト数で圧縮率を測定する（演習問題 5 ではホールドアウトされたテキストで評価する）。

```
big = BPETokenizer(vocab_size=4096)
big.train(raw_text)
excerpt = raw_text[:10000]
sizes = [256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096]
ratios = []
for v in sizes:
    sub = BPETokenizer(vocab_size=v)  # the first v - 256 merges of big
    sub.merges = {p: i for p, i in big.merges.items() if i < v}
    sub.vocab = {i: b for i, b in big.vocab.items() if i < v}
    ratios.append(len(excerpt.encode('utf-8')) / len(sub.encode(excerpt)))
d2l.plot(sizes, [ratios], 'vocabulary size', 'bytes per token',
         xscale='log')
```



語彙を 2 倍にするごとにほぼ一定の圧縮率向上が得られるため、1 トークンあたりのバイト数は語彙サイズに対しておよそ対数的に増加する。このスケーリングは貪欲なマージ規則から直接導かれる。`"e "`、`"th"`、`"d "` のような初期のマージはそれぞれ数千箇所の出現を一気に縮約するのに対し、4,000 番目のマージはわずか数箇所しか縮約しないからである。ただし、この曲線を 50,000〜200,000 トークンの実用語彙へ無批判に外挿しないよう注意が必要である。プロットの上端が良く見えているのは、トークナイザの訓練に使ったのと同じテキスト上で評価しているためであり、そこでは 4,096 トークンの語彙が『The Time Machine』固有の単語やフレーズを記憶してしまっている。一般的な英語の散文では、後ほど検証するように実用語彙はおおむね 1 トークンあたり 4〜5 バイト（およそ 1 単語あたり 1 トークン）となり、余剰の語彙容量はより低頻度な単語やソースコード、そして（`o200k_base` のような多言語語彙では）他言語のために使われる。

8.2.3.5 事前トークン化

複合トークン `"Time Traveller "` は決して例外的な事例ではない。制約のないこのトークナイザは、本来の単語断片と並んで、単語境界をまたぐトークンを何十個も蓄積してしまう。

```
crossers = [v.decode('utf-8', 'replace') for i, v in tok.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print_wrapped(len(crossers), 'tokens cross a word boundary, e.g.',
              crossers[:8])
```

```
78 tokens cross a word boundary, e.g. [', and ', 'of the ', '. I ', '. Th',
    '. A', '. I', 'in the ', 'ed to ']
```

"of the " のような複数単語のマージは訓練テキストの圧縮には役立つものの、脆い複合表現である。固定フレーズに語彙枠を費やす一方で、構成単語の統計的な手がかりを分断してしまうからである（"of" が多数の異なるトークンに分散して出現するようになる）。そのため、すべての実用 BPE トークナイザはマージの前に**事前トークン化**（pre-tokenization）の段階を挿入する。まず正規表現（regular expression）によって入力文字列をチャンクに分割し、チャンク境界を越えてマージが行われないよう、厳密に各チャンクの**内部**でのみ BPE を訓練・適用する。GPT-2 のパターン (Radford et al. 2019) は `BPETokenizer.GPT2_PATTERN` として格納されており、次の 5 つの分岐から構成される。

```
's|'t|'re|'ve|'m|'ll|'d        common English contractions, split off
| ?\p{L}+                      a run of letters, with its leading space
| ?\p{N}+                      a run of digits, with its leading space
| ?[^\s\p{L}\p{N}]+            a run of punctuation or other symbols
|\s+(?!\S)|\s+                 remaining whitespace
```

このパターンには 3 つの意図的な規約が組み込まれている。第 1 に、空白が単語自体と一体になって移動するよう、各単語はその**先頭の空白**を保持する（" the" であり、"the " ではない）。第 2 に、数値や記号が決して単語と融合しないよう、文字・数字・句読点は別々のチャンクに分離される。第 3 に、英語の一般的な短縮形は独立したチャンクとして切り離される。

```
regex.findall(BPETokenizer.GPT2_PATTERN,
              "The traveller's clock struck 12,345 times.")
```

```
['The',
 ' traveller',
 "'s",
 ' clock',
 ' struck',
 ' 12',
 ',',
 '345',
 ' times',
 '.']
```

事前トークン化を有効にするには、`pattern` を `BPETokenizer` に渡すだけでよい。Figure 8.5 に、生テキストからトークン ID までのパイプライン全体を示す。制約なしの実行結果と比べると、単語境界をまたぐ複合トークンはすべて消失し、解放された語彙枠は先頭の空白を伴う完全な単語へと割り当てられる。また、初期のマージは頻出文字を先頭の空白に結合するもの（" t"、" a"）や、頻出する文字ペアを結合するもの（"he"、"in"）になる。

```
bpe = BPETokenizer(vocab_size=1024, pattern=BPETokenizer.GPT2_PATTERN)
bpe.train(raw_text)
crossers = [v for i, v in bpe.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print('boundary-crossing tokens now:', len(crossers))
gained = sorted(set(bpe.vocab.values()) - set(tok.vocab.values()),
                key=lambda v: (-len(v), v))
print_wrapped('longest tokens gained instead:',
              [v.decode('utf-8', 'replace') for v in gained[:6]])
print('first merges:', [bpe.vocab[i] for i in list(bpe.merges.values())[:6]])
```

```
boundary-crossing tokens now: 0
longest tokens gained instead: [' Psychologist', 'sychologist', '
    Traveller', ' Morlocks', ' darkness', ' Machine']
first merges: [b' t', b'he', b' a', b'in', b' the', b' s']
```

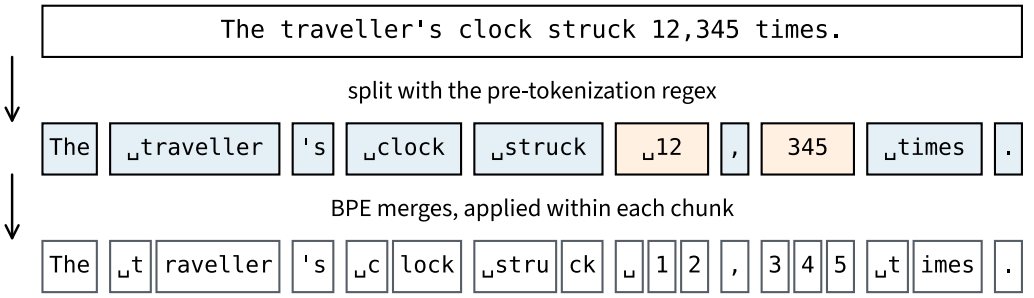


Figure 8.5: 実際の文に対する事前トークン化の動作。正規表現がまずテキストを、先頭の空白を伴う単語、数字の並び、句読点というチャンクに切り分ける。続いて各チャンクの内部で BPE マージが適用され、チャンク境界をまたぐことは決してない。

8.2.4 特殊トークン

訓練やサービングのパイプラインでは、テキストそのものではなく**インフラストラクチャ**にとって意味を持つ少数の制御トークンも必要になる。<pad> は短い系列を共通のバッチ長に揃えるためにパディングし、<bos> は系列の先頭を表し、<eos> は生成が完了したことをモデルが通知するために使われる（8.6 節のデコーダはこれを停止の合図として監視する）。`BPETokenizer` では、これらの特殊 ID を BPE の範囲よりも**上**に配置している。これは、通常のテキストが誤って特殊トークンへとエンコードされるのを防ぐための意図的な安全上の不変条件（invariant）である。そうでなければ、"<eos>" という文字そのものを含む文書やユーザーメッセージがモデルの入力ストリームへ制御信号を直接注入できてしまい、信頼できないユーザーテキストがシステム命令として解釈される**プロンプトインジェクション**の典型的な攻撃経路となってしまう。そのため、`encode` メソッドは呼び出し側が明示的に `allow_special=True` を渡さない限り特殊トークン文字列を通常のバイト列として扱い、`tiktoken` の `allowed_special` ガード（デフォルトではエラーを送出する）と同様の防御を実現している。

```
print('as plain text:  ', bpe.encode('<bos>the time machine'))
print('as control code:', bpe.encode('<bos>the time machine',
                                     allow_special=True))
print('pad/bos/eos ids:', bpe.pad, bpe.bos, bpe.eos, '| len:', len(bpe))
```

```
as plain text:   [60, 98, 111, 115, 62, 341, 513, 680]
as control code: [1025, 341, 513, 680]
pad/bos/eos ids: 1024 1025 1026 | len: 1027
```

8.2.5 実世界のトークナイザ

8.2.5.1 実用トークナイザのランクテーブル

GPT 系やバージョン 3 以降の Llama をはじめ、広く運用されている言語モデルの大半は、これとまったく同じ仕組みで構築され、はるかに大規模なコーパスで訓練されたバイトレベル BPE ト

ークナイザを採用している。GPT-2 のトークナイザ (Radford et al. 2019) は、各トークンのバイト系列をそのマージランクへ対応付けるルックアップテーブルに、事前トークン化用の正規表現と特殊トークンの登録情報を組み合わせた形でコンパクトに保存されており、その内部にニューラルネットワークは存在しない（WordPiece や Unigram などの他のサブワード手法では、これに加えてテキスト正規化規則や、競合する分割候補の中から選択するためのトークン確率も保持する）。OpenAI の `tiktoken` ライブラリには、GPT-2 のランクテーブルがそのまま同梱されている。以下では、単一バイトと学習済みマージの境界をまたぐ項目を確認してみよう。なお、`_mergeable_ranks` の先頭にあるアンダースコアは内部属性であることを示しており、ライブラリのリリースによって名前が変わる可能性がある。

```
enc = tiktoken.get_encoding('gpt2')
print(list(enc._mergeable_ranks.items())[254:260])
print(enc._special_tokens)
```

```
[(b'\xa0', 254), (b'\xad', 255), (b' t', 256), (b' a', 257), (b'he', 258), (b'in', 259)]
{'<|endoftext|>': 50256}
```

ランク 0〜255 は並び替えられた順序での個々のバイトに対応し（GPT-2 は印字可能な ASCII 文字を先に並べている）、ランク 256 以降には学習されたマージが訓練順に続く。これは先ほど作成した `BPETokenizer` とまったく同じ構成である。驚くべきことに、40 GB のウェブテキストから学習された GPT-2 の最初の 4 回のマージ（`" t"`、`" a"`、`"he"`、`"in"`）は、19 世紀の 1 編の中編小説から `bpe` が学習した最初の 4 回のマージと完全に一致しており、英語における最頻出の文字パターンの安定性を物語っている。

8.2.5.2 実装の検証

BPE トークナイザはそのランクテーブルによって完全に規定されるため、公開されている GPT-2 のランクをそのまま自作の `BPETokenizer` エンコーダに読み込むことができる。各ランクがそのままトークン ID となり、256 個の単一バイト項目がバイトの置換順序を定義する。公開されているテーブルには各トークンを形成した親ペアではなく最終的なバイト列のみが記録されているため、各トークンのバイト列に対してよりランクの低いマージをすべて再生し、ちょうど 2 つの断片が残るまで縮約することで各マージペアを再構築する。以下の再構築の詳細は流し読みして構わない。重要なのはその後に続く検証用のセルである。

```
@d2l.add_to_class(BPETokenizer)
@classmethod
def from_tiktoken(cls, name):
    """Load a published bytes->rank table (e.g. 'gpt2') into our encoder."""
    import tiktoken  # lazy: d2l itself does not require tiktoken
    enc = tiktoken.get_encoding(name)
    ranks = enc._mergeable_ranks
    tok = cls(vocab_size=len(ranks), pattern=enc._pat_str, specials=())
    tok.specials = dict(enc._special_tokens)
    tok.vocab = {rank: b for b, rank in ranks.items()}
    tok.byte_ids = [ranks[bytes([b])] for b in range(256)]
    for token, rank in ranks.items():
        if len(token) > 1:  # recover which pair merged into this token
            parts = [bytes([b]) for b in token]
            while len(parts) > 2:
                a, b = min(zip(parts, parts[1:]),
                           key=lambda p: ranks.get(p[0] + p[1],
```

```
                                                float('inf')))
                i = list(zip(parts, parts[1:])).index((a, b))
                parts[i:i + 2] = [a + b]
            tok.merges[ranks[parts[0]], ranks[parts[1]]] = rank
    return tok
```

散文、数値、アクセント付き文字、通貨記号、絵文字を織り交ぜたテスト段落において、このエンコーダは tiktoken とトークン単位で完全に一致する。このアサーションは本書がビルドされるたびに自動的に検証されている。

```
ours = BPETokenizer.from_tiktoken('gpt2')
para = ('The Time Traveller (for so it will be convenient to speak of '
        'him) was expounding a recondite matter to us; his pale grey '
        'eyes shone and twinkled. Prices rose 3.5% in 1895, and naïve '
        'café patrons paid £12,345 more!  ')
assert ours.encode(para) == enc.encode(para)
print(len(ours.encode(para)), 'tokens, identical to tiktoken')
print([ours.vocab[i].decode('utf-8', 'replace')
       for i in ours.encode(para)[:8]])
```

```
60 tokens, identical to tiktoken
['The', ' Time', ' Trave', 'ller', ' (', 'for', ' so', ' it']
```

大規模な実用語彙は『The Time Machine』をどの程度うまく圧縮できるだろうか。以下では、ここで作成した 1,024 トークンのトークナイザを、GPT-2 の 50,257 トークンの語彙、および GPT-4o で採用されている約 200,000 トークンの語彙 o200k_base と比較する。

```
rows = [('ours (1k, in-domain)', len(bpe.encode(raw_text)))]
for name in ('gpt2', 'o200k_base'):
    e = tiktoken.get_encoding(name)
    rows.append((f'{name} ({len(e._mergeable_ranks) // 1000}k)',
                 len(e.encode(raw_text))))
for name, n in rows:
    print(f'{name:>22}: {n:>6} tokens, {n_bytes / n:.2f} bytes/token')
```

```
  ours (1k, in-domain):  65741 tokens, 2.72 bytes/token
           gpt2 (50k):  44073 tokens, 4.06 bytes/token
    o200k_base (199k):  42624 tokens, 4.20 bytes/token
```

ここで作成した語彙はサイズが 50〜200 分の 1 にすぎないにもかかわらず、この中編小説に特化して訓練されているため、実用語彙のおよそ 3 分の 2 の圧縮率を達成している。他のあらゆる統計モデルと同様、トークナイザもその訓練分布（training distribution）に対する圧縮モデルであり、構築元のコーパスに似たテキストに対して最も高い性能を発揮する。

8.2.5.3 肥沃度、数字、グリッチトークン

ある言語においてトークナイザが 1 単語あたりに費やす平均トークン数は、**肥沃度**（fertility）と呼ばれる。GPT-2 のトークナイザは主に英語のウェブテキストで訓練されたため、まったく同じ文を 3 つの言語に翻訳すると、トークン数に際立った差が生じる。

```
sentences = {
    'English': 'The time traveller returned after many years.',
    'German': 'Der Zeitreisende kehrte nach vielen Jahren zurück.',
    'Greek': 'Ο ταξιδιώτης του χρόνου επέστρεψε μετά από πολλά χρόνια.'}
for lang, s in sentences.items():
    gpt2, o200k = [len(tiktoken.get_encoding(e).encode(s)) / len(s.split())
                   for e in ('gpt2', 'o200k_base')]
    print(f'{lang:>8}: {gpt2:.2f} (gpt2), {o200k:.2f} (o200k_base) '
          'tokens per word')
```

```
English: 1.14 (gpt2), 1.14 (o200k_base) tokens per word
 German: 2.86 (gpt2), 1.57 (o200k_base) tokens per word
  Greek: 7.33 (gpt2), 1.89 (o200k_base) tokens per word
```

GPT-2 のトークナイザでは、英単語が 1 語あたり約 1 トークンであるのに対し、ギリシャ語の単語は約 7 トークンも消費する（ギリシャ文字は 1 文字あたり UTF-8 で 2 バイトを要するうえ、ギリシャ語のマージがほとんど学習されなかったためである）。その結果、1,024 トークンのコンテキストウィンドウには英語なら約 900 語収まるのに対し、ギリシャ語は約 140 語しか収まらず、英語圏以外のユーザーにとっては推論がより低速かつ高コストになる。対照的に、多言語対応の語彙 `o200k_base` はギリシャ語の肥沃度を 1 語あたり約 2 トークンまで引き下げる。

数値の例を見ると、トークン化の微妙な仕様がいかにモデルの学習対象を左右するかがわかる。GPT-2 は BPE がたまたま頻出断片を見つけた位置で数字の並びを分割するのに対し、より新しいトークナイザは分割をより均一にするため、事前トークン化の段階で数字チャンクの上限を 3 文字に制限している。

```
for name in ('gpt2', 'o200k_base'):
    e = tiktoken.get_encoding(name)
    print(f'{name}: 12345678 ->',
          [e.decode([t]) for t in e.encode('12345678')])
```

```
gpt2: 12345678 -> ['123', '45', '678']
o200k_base: 12345678 -> ['123', '456', '78']
```

それでもなお、いずれの分割も、右から左へ一の位・十の位・百の位と桁を束ねる位取り記数法の筆算の列とは一致しない。左から右へ 3 桁ずつ区切る上限のもとでは、`"12345678"` は `"123"`、`"456"`、`"78"` へと分割されるため、言語モデルは桁のずれた境界をまたいで繰り上がりを処理しなければならなくなる。この不一致は言語モデルにおける算術誤りのよく知られた一因であり、標準的な 3 桁区切りのカンマ（`"12,345,678"`）を挿入すると数値タスクが容易になり得る理由の説明にもなっている。

最後に、語彙項目がどれほど信頼できるかは、訓練中にモデルがそのトークンにどれだけ触れたかに依存する。初期の GPT-2 や GPT-3 の語彙には `" SolidGoldMagikarp"` のようなトークンが含まれていた。これは Reddit のユーザー名であり、**トークナイザ**の訓練コーパスには専用のマージを獲得するほど頻繁に出現したにもかかわらず、**言語モデル**の事前学習データにはほとんど含まれていなかったものである。このような**グリッチトークン**は、その埋め込みベクトルが勾配更新をほとんど受けていないにもかかわらず、ユーザーのプロンプトから生成され得る。初期のモデルにこのトークンを繰り返すよう促すと、しばしばはぐらかすような応答や無関係な単語を返した。

```
enc.encode(' SolidGoldMagikarp')  # one token for an 18-character string
```

```
[43453]
```

8.2.6 まとめ

トークナイザは、生のテキストとニューラルネットワークが受け取る整数系列との橋渡しをする。語彙サイズは系列長と出力層サイズの間の基本的なトレードオフを支配し、さらに固定された単語語彙は未知語に対して破綻する。バイトレベル BPE は 256 個の単一バイト記号から出発することで語彙外の入力を完全に排除し、目標の語彙サイズに達するまで最頻出の隣接トークンペアを貪欲にマージする。エンコードは学習済みマージを厳密にランク順で再生するため、マージテーブルだけでトークナイザが定義される。実用上は、事前トークン化の正規表現によってマージが単語境界をまたぐのを防ぎ、通常のテキストから合成されないよう特殊な制御トークンには BPE の範囲より上の ID を割り当てる。GPT-2 の実用トークナイザはこの構成をより大規模にそのまま踏襲したものであり、本節の実装はそれをトークン単位で再現する。言語ごとに異なる肥沃度、数字分割の癖、グリッチトークンといった現象はいずれも、訓練データとマージ規則から直接生じた帰結である。次節では、トークナイザが生成するトークン系列のモデル化へと進む。

8.2.7 演習問題

1. **ランクによるエンコード**。`BPETokenizer`（`vocab_size=258`）が、まず ID 256 として (b, c)、次に ID 257 として (a, b) というちょうど 2 つのマージを学習したとする。`_encode_chunk` の**ランク規則**は常に、適用可能なマージのうち最もランクの低いものを適用する。一方、**最左規則**は常に、マージテーブルに存在する隣接ペアのうち最も左にあるものをマージする。

    1. 2 つの規則のそれぞれのもとで、文字列「abc」を分割せよ。
    2. 訓練終了時点において、訓練コーパス内のすべての「abc」の出現箇所がどの分割になっていたかを示し、どちらの規則がそれと一致するかを答えよ。

2. [code] **特殊トークンの往復変換**。通常のテキストから決して生成されないよう、8.2.4 節では特殊トークンを BPE の ID 範囲より上に配置した。`BPETokenizer` を、`specials` がデフォルトの 3 つに加えて文書区切り `<sep>` を含むように構築し、`raw_text` 上で訓練せよ。`x` を、2 つの短い文書を `<sep>` で連結した文字列とする。

    1. `x` を `allow_special=True` ありと、なしのそれぞれで 1 回ずつエンコードし、両方の ID 系列が `x` にデコードされることを確認せよ。
    2. デコードされたテキストを `allow_special=True` ありと、なしのそれぞれで 1 回ずつ再エンコードせよ。それぞれの再エンコードは、前の小問で得られた 2 つの ID 系列のうちどちらを再現するか。モデルの出力をデコードし、後でそのテキストを再び入力として与えるパイプラインにとって、これが何を意味するかを述べよ。
    3. 特殊文字列の混入が許可されていないテキストにその文字列が現れたとき、`tiktoken` はエラーを送出するのに対し、`encode` はその文字列を通常のバイト列として扱う。それぞれの方針が望ましい状況を 1 つずつ挙げよ。

    Andrej Karpathy の minbpe、Step 4 を改変。

3. [code] **数字のチャンク化**。加算プロンプト「1234+5678=」、「1234+567=」、「567+678=」およびそれぞれの和「6912」、「1801」、「1245」を、`gpt2` と `o200k_base` の各エンコーディング（`tiktoken`）でトークン化し、各数値のトークン境界を書き出せ。

    1. `o200k_base` の 3 桁上限のもとで、各数字トークンが担う位取りの値（一の位、十の位、百の位、千の位）をラベル付けせよ。各プロンプトについて、足される 2 つの数と和の対応する位置にあるトークンが同じ位取りの値を担っているかどうかを述べよ。また、同じでない場合、トークン単位で加算するモデルにどのような問題が生じるかを説明せよ。

2. トークンを位取りの値と揃える事前トークン化規則を 1 つ提案し、その規則の欠点を 1 つ述べよ。

4. **グリッチトークン**。グリッチトークンの原因について、8.2.5.3 節ではトークナイザの訓練コーパスでは頻出したものの言語モデルの訓練コーパスではまれであった文字列に帰着させた。

   1. なぜ `BPETokenizer.train` のマージ規則はそのような文字列を複数の断片ではなく単一のトークンにしてしまうのか。また、事前トークン化はそのトークンの範囲を確定するうえでどのような役割を果たすか。
   2. 訓練済みの埋め込みテーブルとモデルの訓練コーパスが与えられたとき、グリッチトークンを検出するテストを 1 つ提案し、2 つの入力がそれぞれ何に寄与するかを述べよ。

5. [code] **事前トークン化のアブレーション**。●`raw_text` を前半の 90% と後半の 10% に分割せよ。前半部分を用いて、パターンの有無のみが異なる 4,096 トークンのトークナイザを 2 つ訓練せよ（1 つは `big` と同様にパターンなし、もう 1 つは `BPETokenizer.GPT2_PATTERN` ありとする）。

   1. 各トークナイザが学習した最も長い 20 個のトークンを列挙し、複数の単語にまたがるものに印を付けよ。
   2. 2 つのうちどちらが前半部分をより高い圧縮率で圧縮するかを測定し、その結果をマージ規則の観点から説明せよ。
   3. 後半の 10% に対して両トークナイザの 1 トークンあたりのバイト数を測定せよ。このホールドアウトされたテキストにおいて、圧縮率の低下割合が大きいのはどちらのトークナイザか。また、より高い圧縮率を達成するのはどちらか。両方の答えを、先ほど列挙したトークンと関連付けて説明せよ。

8.3 言語モデル

次の単語を提案するスマートフォンのキーボードも、話者が「to recognize speech」と言ったのか「to wreck a nice beach」と言ったのかを判定する音声認識器も、同じ能力に依存している。それは、あるテキストの一節がどれほどもっともらしいかを表す数値スコアを割り当てる能力である。これら 2 つのフレーズは発音がほぼ同じであるため、音響情報だけでは区別できない。曖昧さを解消するのは、前者が自然な英語表現であるのに対し、後者が奇抜で不自然だという知識である。**言語モデル**（language model）はこの判断を担う。すなわち、トークン系列 $x_1, \dots, x_T$（語彙 $\mathcal{V}$ から選ばれた、たとえば 8.2 節で構築した BPE トークンなど）が与えられたとき、その同時確率（joint probability）$P(x_1, \dots, x_T)$ を推定する。

一見すると、起こり得るすべての系列に確率を割り当てるのは絶望的に思える。1,024 トークンという控えめな語彙であっても、長さ 10 の系列はすでに $1024^{10} \approx 10^{30}$ 通りも存在するからである。幸いなことに、確率の連鎖律（chain rule、8.1 節）を使えば、この直接扱えない同時分布（joint distribution）を近似なしで次トークンの条件付き確率の積へと変換できる。各項は、直前までのプレフィックス $x_{<t} = x_1, \dots, x_{t-1}$ で条件付けられる。

$$P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.6}$$

したがって、言語モデルは本質的に次トークン予測器である。同じ条件付き分布（conditional distribution）を用いることで、既存の系列に**スコアを付ける**こと（確率を掛け合わせる、またはその対数を足し合わせる）も、新しいテキストを**生成する**こと（まず x_1 をサンプリングし、次に

x_2 を直前の x_1 のもとでサンプリングし、段階的に続けていく）も可能になる。注目すべきことに、幅広いタスクをテキストの続きを書く問題としてそのまま定式化できる。たとえば、「The novella The Time Machine was written by」というプロンプトの続きを補完すれば事実に関する質問への回答となり、これこそが現代の大規模言語モデル（large language model、11.2 節）の根底にあるパラダイムである。

まずは、こうした条件付き確率（conditional probability）に対する最も単純な推定量、すなわちコーパス内での出現回数を数える方法から始め、それがなぜ破綻するのかを調べる。改善を客観的に評価するため、パープレキシティを導入して 2 つの古典的な平滑化手法を比較し、続いて異なるトークナイザを持つモデル同士を公平に比較できるようバイトあたりビット数を定義する。最後に、トークン化された系列を次節以降のニューラル言語モデルに供給するミニバッチデータローダを構築する。

```
import collections
import random
import re
```

8.3.1 n グラム言語モデル

8.3.1.1 マルコフモデルとカウント

Equation 8.6 の各因子はプレフィックス全体 $x_{<t}$ に条件付けられているため、t が大きくなるにつれて異なる文脈の数は際限なく増大する。そこでマルコフ条件（8.1.3.1 節）を導入し、文脈を直近の $n-1$ 個のトークンに打ち切る。

$$P(x_t \mid x_{<t}) \approx P(x_t \mid x_{t-n+1}, \dots, x_{t-1}). \tag{8.7}$$

この仮定が粗いことは否定できないが（文が 2 語進んだだけで主語を忘れることはまずない）、取り得る文脈の集合を有限個に絞ることができる。文脈を持たないモデル（$n=1$）は**ユニグラム**モデル、直前の 1 トークンのみに条件付けるモデル（$n=2$）は**バイグラム**モデル、直前の 2 トークンに条件付けるモデル（$n=3$）は**トライグラム**モデルと呼ばれる。最尤推定（maximum likelihood estimation）のもとでは、各条件付き確率はコーパス内の出現回数の比にほかならない。トークンの組が訓練コーパスに出現する回数を $N(\cdot)$ と書けば、次式を得る。

$$\hat{P}(x_t \mid x_{t-n+1}, \dots, x_{t-1}) = \frac{N(x_{t-n+1}, \dots, x_t)}{N(x_{t-n+1}, \dots, x_{t-1})}, \tag{8.8}$$

これは、文脈 $(x_{t-n+1}, \dots, x_{t-1})$ の直後に x_t が続いた経験的な割合である。n グラム言語モデルとは、こうした条件付き相対頻度のルックアップテーブルにすぎない。

『The Time Machine』を用いてユニグラム、バイグラム、トライグラムのテーブルを推定してみよう。本章の後半で扱うニューラル言語モデルは BPE トークンを対象に動作するが、ここでは生成された文章を目視で確認しやすいよう、この中編小説を単語単位でトークン化する。8.2 節の単純な小文字英字への正規化を再利用し、空白で分割したうえで、テキストの末尾 10% を評価用にホールドアウトする。語彙は訓練分割に出現するすべての単語からなり、訓練時に出現しなかったホールドアウト内の単語はすべて未知トークン（unknown token）<unk> に写像する。

```
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt',
                     sha1_hash='090b5e7e70c295757f55df93cb0a180b9691891a')
```

```
raw_text = open(fname).read()
text = re.sub('[^A-Za-z]+', ' ', raw_text).lower()
split = text.rindex(' ', 0, int(0.9 * len(text)))
words_train = text[:split].split()
known = set(words_train)  # A held-out word never seen in training is <unk>
words_val = [w if w in known else '<unk>' for w in text[split:].split()]
len(words_train), len(words_val)
```

```
(29402, 3373)
```

以下の NGramLM クラスは、観測されたすべての $(n-1)$ トークンの文脈に対し、訓練中にその直後に続いたトークンのカウンタを保持する。これだけで Equation 8.8 を計算するのに必要な情報がすべて揃う（alpha 属性は、後ほど平滑化を導入する際に使用する）。また、sample メソッドは与えられたプレフィックスを延長するため、直近 $n-1$ 個のトークンのもとでの出現回数に比例して次のトークンを繰り返し抽出する。

```
class NGramLM:
    """An n-gram language model estimated from counts."""
    def __init__(self, tokens, n, vocab_size, alpha=0.01):
        self.n, self.vocab_size, self.alpha = n, vocab_size, alpha
        cont = collections.defaultdict(collections.Counter)
        for gram in zip(*(tokens[i:] for i in range(n))):
            cont[gram[:-1]][gram[-1]] += 1
        self.cont = dict(cont)
        self.total = {ctx: sum(c.values()) for ctx, c in self.cont.items()}

    def sample(self, prefix, num_tokens, rng):
        """Extend prefix by sampling from the unsmoothed count model."""
        out = list(prefix)
        for _ in range(num_tokens):
            ctx = tuple(out[len(out) - self.n + 1:])
            counts = self.cont[ctx]
            out.append(rng.choices(list(counts), counts.values())[0])
        return out

vocab_size = len(known) + 1  # The training words and <unk>
models = {n: NGramLM(words_train, n, vocab_size) for n in (1, 2, 3)}
```

これらのテーブルが実際に何を捉えているかを調べるため、タイムトラベルを扱ったこの中編小説全体で頻繁に登場する文脈 "the time" について、トライグラムモデルにクエリを実行してみよう。

```
trigram = models[3]
total = trigram.total[('the', 'time')]
for word, count in trigram.cont[('the', 'time')].most_common(3):
    print(f'P({word} | the time) = {count}/{total} = {count / total:.2f}')
single = sum(len(c) == 1 for c in trigram.cont.values()) / len(trigram.cont)
print(f'word pairs followed by a single word in training: {single:.0%}')
```

```
P(traveller | the time) = 47/87 = 0.54
P(machine | the time) = 27/87 = 0.31
P(i | the time) = 6/87 = 0.07
```

```
word pairs followed by a single word in training: 84%
```

訓練分割において "the time" は 87 回出現し、その直後には約半数の割合で "traveller" が、約 3 分の 1 の割合で "machine" が続いており、このテーブルが作品の核となるフレーズを的確に捉えていることがわかる。しかし、このような高頻度の文脈はまれな例外にすぎない。観測された単語ペアの 5 分の 4 以上では、訓練テーブルに記録された後続語がただ 1 種類しかないのである。

8.3.1.2 n グラムモデルからのサンプリング

テキストの生成には Equation 8.6 をそのまま適用する。すなわち、そこまでのプレフィックスを条件とする条件付き分布から次のトークンを引き、末尾に追加する操作を繰り返す。もっとも、一見流暢な出力であっても丸暗記の丸写しが覆い隠されていることが少なくないため、生成サンプルとあわせて診断関数 longest_copy を用いる。この関数は、生成された単語列のうち訓練セット内に一言一句そのまま現れる最長の連続区間を測定する。

```
def longest_copy(words, train_words):
    """Length of the longest run of words that also occurs in train_words."""
    corpus, best = ' ' + ' '.join(train_words) + ' ', 0
    for i in range(len(words)):
        while (i + best < len(words) and
               ' ' + ' '.join(words[i:i + best + 1]) + ' ' in corpus):
            best += 1
    return best
```

各モデルにプレフィックス "the time" をシードとして与えて 25 語を生成する。実行するたびに新しいサンプルが得られるよう、乱数生成器（random number generator）にはシードを固定しないでおく。

```
rng = random.Random()
for n, model in models.items():
    sample = model.sample(['the', 'time'], 25, rng)
    copied = longest_copy(sample, words_train)
    d2l.print_wrapped(f'{n}-gram (longest copied run: {copied} words): '
                      + ' '.join(sample))
```

```
1-gram (longest copied run: 2 words): the time me and a had i laboratory
    apologize i all seemed potentialities so reached such i assured reached
    little is following of head one however the
2-gram (longest copied run: 3 words): the time and when we began to trouble
    myself but the under world of our friend of flowers brilliant arch of
    the sun belt sandals or other
3-gram (longest copied run: 13 words): the time traveller not a mere block
    but highly decorated with deep framed panels on either hand to pick
    flowers to stick a fork into meat again
```

ユニグラムモデルは文法的な一貫性なしにコーパス内の頻度に従って単語を出力し、バイグラムモデルは各単語が直前の 1 語のみに依存するため、局所的にはもっともらしいペアをつなぎ合わせながらもあてどなくさまよう。一方、トライグラムのサンプルは早くも驚くほど H. G. Wells らしく聞こえる。

しかし、30,000 語のコーパスにおいてトライグラムが Wells らしく聞こえるのは、その大部分が文字どおり Wells の文章**そのもの**だからである。コピーされた最長連続区間を見ればこの振る舞いは一目瞭然であり、ユニグラムでは 2〜3 語、バイグラムでは数語であるのに対し、トライグラ

ムでは典型的に 7〜12 語もの連続した単語に達する。サンプラが訓練時に後続語が 1 種類しかなかった単語ペアに踏み込むたびに、モデルは単一の決定論的な経路へと強制され、小説の節全体をそのまま再生してしまう。モデルは汎化（generalization）しているのではなく、記憶した断片をつぎはぎしているのである（もっとも、数十億語規模で推定すれば、n グラムテーブルは英語の豊かな局所的統語構造を確かに捉えるため、何十年にもわたり音声認識や統計的機械翻訳を支えてきた）。

8.3.1.3 カウントのスパース性

この丸暗記は、より根深い統計的ボトルネックが表面化した症状である。**取り得る** n グラムの総数は $|\mathcal{V}|^n$ であり、n に対して指数関数的に増大するのに対し、コーパス内に実際に現れる n グラムの数はコーパス長に対して線形にしか増えない。結果として生じるスパース性は、ホールドアウトの n グラムのうち訓練時に 1 度も出現しなかったものの割合を調べることで直接測定できる。

```
for n in (1, 2, 3):
    seen = set(zip(*(words_train[i:] for i in range(n))))
    grams = list(zip(*(words_val[i:] for i in range(n))))
    rate = sum(gram not in seen for gram in grams) / len(grams)
    print(f'unseen held-out {n}-grams: {rate:.1%} '
          f'({len(seen)} distinct in training)')
```

```
unseen held-out 1-grams: 8.4% (4315 distinct in training)
unseen held-out 2-grams: 51.5% (18535 distinct in training)
unseen held-out 3-grams: 87.0% (26982 distinct in training)
```

ホールドアウトの単語の約 12 分の 1 は訓練時に 1 度も出現しておらず、ホールドアウトの単語ペアの半数は未知であり、3 語組（トライグラム）に至っては 10 個中ほぼ 9 個がカウント 0 である。最尤推定量（maximum likelihood estimate）Equation 8.8 のもとでは、未知の n グラムにはすべて確率 0 が割り当てられる。そして Equation 8.6 の中に 0 の因子が 1 つでもあれば、英語として完全に妥当な文章全体に確率 0 が割り当てられてしまう。

8.3.1.4 単語頻度とジップの法則

なぜカウントテーブルはこれほど急速にスパースになるのだろうか。根本的な原因は、自然言語における単語頻度（word frequency）の分布が極端に偏っていることにある。トークンを頻度の降順に順位付けする `Vocab` クラス（8.2 節）を用いて、『The Time Machine』における上位 10 個の最頻出語を調べてみよう。

```
words = text.split()
vocab = d2l.Vocab(words)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
```

```
 ('that', 443),
 ('my', 440)]
```

これら上位 10 個の最頻出語からは、タイムトラベルに関する手がかりは何一つ得られない。並んでいるのは冠詞、代名詞、前置詞（しばしば**ストップワード**と呼ばれる）であり、ヴィクトリア朝のどの小説でもほぼ同じリストになるだろう。目を引くのは、頻度がどれほど急激に落ち込むかである。10 番目に多い単語の頻度は、早くも最頻出語の 5 分の 1 未満にすぎない。両対数軸上に頻度と順位をプロットすると、最上位の数語を除けば直線にきわめてよく乗り、べき乗則の特徴が現れる。具体的には、単語の頻度 n_i（最頻出から i^{th} の単語）は**ジップの法則**（Zipf's law）に従う。

$$n_i \propto \frac{1}{i^s}, \quad \text{equivalently} \quad \log n_i = -s \log i + c, \tag{8.9}$$

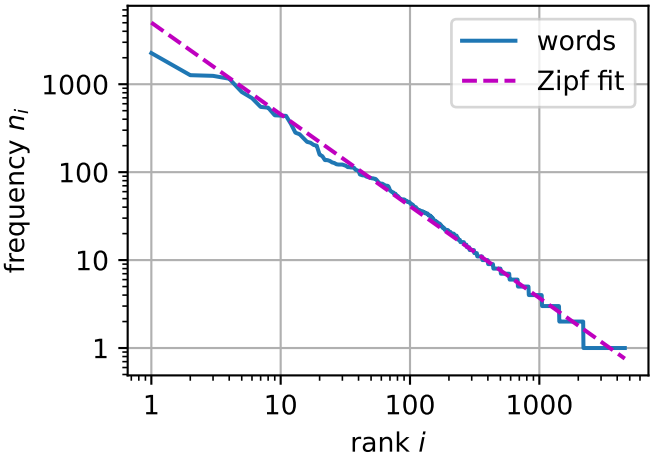
ここで s は指数、c は定数である。以下の補助関数 `zipf_fit` は、順位 10 から 1,000 までの範囲で両対数空間に直線を当てはめることで s と c を推定する。

```
def zipf_fit(freqs, lo=10, hi=1000):
    """Fit log n_i = -s log i + c over ranks lo..hi; return (s, c)."""
    ranks = np.arange(lo, hi + 1)
    slope, c = np.polyfit(np.log(ranks), np.log(freqs[lo - 1:hi]), 1)
    return -slope, c
```

`zipf_fit` を単語頻度 `freqs`（`vocab.token_freqs` から取得）に適用し、推定された指数 s を表示するとともに、経験的な出現回数にべき乗則の直線を重ねて描画する。

```
freqs = [freq for token, freq in vocab.token_freqs]
s, c = zipf_fit(freqs)
print(f'Zipf exponent s = {s:.2f}')
ranks = np.arange(1, len(freqs) + 1)
d2l.plot(ranks, [freqs, np.exp(c) * ranks ** -s], xlabel='rank $i$',
         ylabel='frequency $n_i$', xscale='log', yscale='log',
         legend=['words', 'Zipf fit'])
```

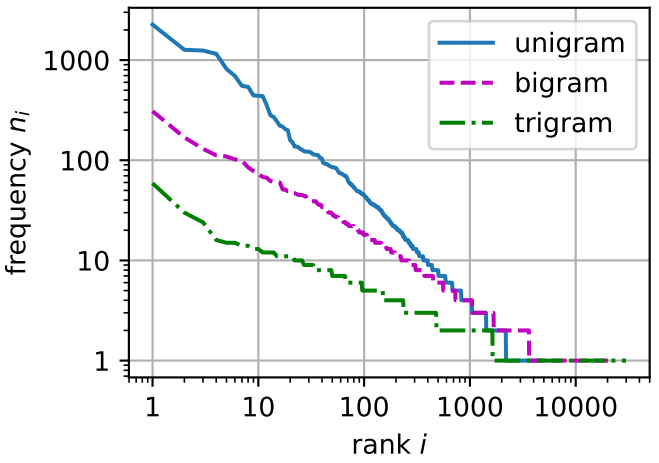
```
Zipf exponent s = 1.04
```



当てはめた指数は 1 に近い値となり、さまざまな自然言語にわたる Zipf の古典的な観測と一致する。べき乗則に従う分布は裾が重い（heavy-tailed）。個々の希少語が出現する頻度は低くとも、異なる希少語があまりに数多く存在するため、それらを合わせるとどのテキストでもかなりの割合を占めるのである。バイグラムやトライグラムでも同様の性質が見られるかを確かめるため、それぞれの頻度曲線を計算してジップ指数を当てはめ、1 回しか出現しない異なる n グラムの割合を測定し、3 本の曲線をまとめてプロットする。

```
counts = [sorted(collections.Counter(zip(*(words[i:] for i in range(n))))
                 .values(), reverse=True) for n in (1, 2, 3)]
for n, f in zip((1, 2, 3), counts):
    print(f'{n}-grams: s = {zipf_fit(f)[0]:.2f}, '
          f'{np.mean(np.array(f) == 1):.0%} of distinct {n}-grams occur once')
d2l.plot([np.arange(1, len(f) + 1) for f in counts], counts,
         xlabel='rank $i$', ylabel='frequency $n_i$', xscale='log',
         yscale='log', legend=['unigram', 'bigram', 'trigram'])
```

```
1-grams: s = 1.04, 52% of distinct 1-grams occur once
2-grams: s = 0.69, 82% of distinct 2-grams occur once
3-grams: s = 0.47, 95% of distinct 3-grams occur once
```



単語の 2 語組や 3 語組もやはりべき乗則に従うが、その指数はより小さくなる（それぞれ約 0.7 と約 0.5）。カウントベースのモデルにとって決定的なのは、n が増えるにつれて分布の裾が劇的に重くなる点である。異なる単語の半数が 1 回しか出現しないのに対し、異なるトライグラムでは 10 個中 9 個以上が、たった 1 回の観測に確率推定を依存するシングルトンとなる。ホールドアウトテキストで未知のトライグラムが大量発生する原因はまさにこの重い裾にあり、各文脈を独立に推定するモデルは、文脈が長くなるにつれて指数関数的に多くのデータを必要とする。

8.3.2 パープレキシティと平滑化

8.3.2.1 パープレキシティ

カウントテーブルを詳しく調べたことで、サンプル生成では隠れていた深刻なスパース性が明らかになった。これらのテーブルを修復する前に、変更によって本当に汎化が改善したかを判定できるよう、原理に基づいた定量的な評価指標が必要である。ホールドアウトテキストの同時確率を評価するのは自然な出発点だが、系列の生の確率は文書長に対して指数関数的に小さくなってしまう。そのため、モデルの良し悪しにかかわらず、どのモデルも Tolstoy の『戦争と平和』には Saint-Exupéry の『星の王子さま』よりはるかに小さな確率を割り当ててしまう。必要なのは、トークンあたりの平均である。

情報理論（information theory）はまさにその指標を与えてくれる。3.1.2.3 節で見たように、確率的分類器の交差エントロピー損失（cross-entropy loss）は、正解ラベルの**サプライザル**、すなわち正解ラベルに割り当てられた確率の負の対数に等しい。言語モデルはあらゆる位置における次トークン分類器であるため、系列の全 T トークンにわたる平均サプライザルによって評価する。

$$\frac{1}{T}\sum_{t=1}^{T} -\log P(x_t \mid x_{<t}), \tag{8.10}$$

ここで、P はモデルの予測分布（predictive distribution）であり、x_t はステップ t における実際のトークンである。この平均は長さで正規化されるため、長さの異なる文書同士を比較できるようになる。自然言語処理では、Equation 8.10 の指数を取った値を報告するのが慣例であり、これを**パープレキシティ**と呼ぶ (Jelinek et al. 1977)。

$$\text{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(x_t \mid x_{<t})\right). \tag{8.11}$$

言い換えれば、パープレキシティとは観測された各トークンに割り当てられた確率の逆数の幾何平均である。すべての目標トークンに確率 $1/4$ を割り当てるモデルはパープレキシティ 4 を達成し、等確率な 4 つの候補から一様に選ぶときの不確実性と一致する。したがって、パープレキシティはトークンあたりのモデルの**実効分岐数**（effective branching factor）として自然に解釈できる。以下の 3 つの極限ケースがこのスケールの基準点となる。

- 常に正解トークンを確率 1 で予測するモデルは、取り得る最小のパープレキシティ 1 を達成する。
- 観測されたトークンのうち 1 つにでも確率 0 を割り当てるモデルは、パープレキシティが無限大になる。
- 語彙全体にわたる一様分布を予測するベースラインモデルはパープレキシティ $|\mathcal{V}|$ となり、有用なモデルであれば必ず下回るべき上界を与える。

平滑化していない最尤法による n グラムモデルは、直ちに 2 番目のケースに直面する。未知の n グラムすべてに確率 0 を割り当てるため、ホールドアウトテキストに未知のトライグラムが 1 つ現れただけでパープレキシティが無限大に発散してしまうからである。カウントモデルを有意義に評価・比較するには、まずカウントを**平滑化**してゼロを取り除かなければならない。

8.3.2.2 ラプラス平滑化

古典的な解決策は**ラプラス平滑化**（Laplace smoothing、加算 α 平滑化とも呼ばれる）である。この手法では、正の定数 $\alpha > 0$ をすべてのカウントに加え、得られる分布が正規化された状態を保つよう分母に $\alpha|\mathcal{V}|$ を加える。

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{N(x_{t-n+1}, \ldots, x_t) + \alpha}{N(x_{t-n+1}, \ldots, x_{t-1}) + \alpha|\mathcal{V}|}. \tag{8.12}$$

これにより、すべての未知の n グラムは小さな正の確率を受け取り、まったく未知の文脈は一様分布 $1/|\mathcal{V}|$ へと帰着する。この発想は言語モデルより 2 世紀も古く、Laplace は今日まで毎日太陽が昇ってきたという事実のもとで明日も太陽が昇る確率を推定するために $\alpha = 1$ としてこれを導入した。

まず、ラプラス平滑化版の `prob` メソッドを `NGramLM` に追加しよう。このメソッドは、モデルに保持された平滑化定数 `alpha` を用いて Equation 8.12 を実装する。

```
@d2l.add_to_class(NGramLM)
def prob(self, context, token):
    """Laplace-smoothed P(token | last n-1 tokens of context)."""
    ctx = tuple(context[len(context) - self.n + 1:])
    num = self.cont.get(ctx, {}).get(token, 0) + self.alpha
    return num / (self.total.get(ctx, 0) + self.alpha * self.vocab_size)
```

`prob` が用意できたので、`eval_nll` でトークン系列全体の負の対数尤度（negative log-likelihood）の総和を計算し、`words_val` 上でのホールドアウト・パープレキシティを各 n グラム次数について求められるようにする。

```
def eval_nll(model, tokens):
    """Total negative log-likelihood of tokens, in nats."""
    return -sum(math.log(model.prob(tokens[max(0, t - model.n + 1):t],
                                    tokens[t]))
                for t in range(len(tokens)))

for n, model in models.items():
    ppl = math.exp(eval_nll(model, words_val) / len(words_val))
    print(f'{n}-gram held-out perplexity, Laplace smoothing: {ppl:7.1f}')
```

```
1-gram held-out perplexity, Laplace smoothing:   840.6
2-gram held-out perplexity, Laplace smoothing:   753.6
3-gram held-out perplexity, Laplace smoothing:  2481.4
```

ラプラス平滑化のもとでは、生成サンプルが最も説得力に富んでいたはずのトライグラムモデルが、実際には大差で**最悪**のホールドアウト・パープレキシティを記録し、バイグラムもユニグラムをわずかに上回るにとどまる。カウントのスパース性がこの両方の結果を説明する。ホールドアウトのトライグラムの 87% が未知であるとき、トライグラムの予測の大部分はほぼ一様な平滑化の当て推量へと潰れてしまうのである。α を調整しても救済はできない。演習問題 2 で求める最適な α を用いてもなお、トライグラムのパープレキシティは 2,400 を超えたままである。流暢に見えるサンプルは、汎化を測る指標としては当てにならないことが判明した。

これらの数値は、ラプラス平滑化が抱える 2 つの根本的な欠陥を浮き彫りにしている。第 1 に、未知の後続語すべてに確率質量を均等に配分してしまい、各単語が一般にどれほど出現しやすいかを無視している。第 2 に、より短い文脈へフォールバックできない。未知のトライグラムは、その末尾 2 語が高頻度のバイグラムをなしていようと不自然なペアであろうと、まったく同じ平坦な推定値を受け取ってしまうのである。

8.3.2.3 Kneser–Ney 平滑化

Kneser–Ney 平滑化 (Kneser and Ney 1995)（ここでは Chen and Goodman (1999) による補間型の定式化を用いる）はこれら両方の欠陥を解消し、20 年にわたり最強のカウントベース言語モデルとして君臨した。記号として、u を $n-1$ トークンの文脈、w を次のトークン、u' を u の最も古いトークンを落として得られるより短い文脈とする。この手法は、互いに連動する 3 つのアイデアを組み合わせている。

第 1 のアイデアは**絶対ディスカウンティング**（absolute discounting）である。経験的なカウントは希少な事象の真の頻度を系統的に過大評価する。希少な n グラムの膨大な母集団から選ばれる際の選択バイアス（selection bias）により、訓練中に 1 回観測された n グラムは、同じサイズの新たなサンプルでは平均して 1 回未満しか出現しないからである。すべての非ゼロカウントから一定のディスカウント $0 < D < 1$ を差し引くことでこの上方向の統計的バイアスを補正でき、歪みが最も大きい小さなカウントほど強く補正されるとともに、合計 $D\,S(u)/N(u)$ の確率質量が文脈 u において解放される。ここで $S(u)$ は、訓練中に u の直後に観測された異なるトークン種類の数である。

第 2 のアイデアは**補間**（interpolation）である。差し引いて得られた確率質量を一様に広げるのではなく、1 つ下の次数の推定値 $P_{\text{KN}}(w \mid u')$ に配分する。この推定値もまったく同じ形で再帰的に定義され、最下層の一様な下限 $1/|\mathcal{V}|$ まで続く。

$$P_{\text{KN}}(w \mid u) = \frac{\max(N(u,w) - D, 0)}{N(u)} + \frac{D\,S(u)}{N(u)}\,P_{\text{KN}}(w \mid u'). \tag{8.13}$$

この再帰のもとでは、頻繁に観測される文脈は主として自身の経験カウントに依拠する一方、多種多様な後続語とともに観測された文脈はより短い文脈へと多くの重みを移す。これは、過去に多様な続き方をした文脈ほど未知の続き方も許容しやすいという直観を反映している。もし文脈 u が訓練時に 1 度も観測されていなければ、確率質量はすべてそのまま下の次数へと引き継がれ、$P_{\text{KN}}(w \mid u) = P_{\text{KN}}(w \mid u')$ となる。

第 3 のアイデアは、低次のモデルで**何を**数えるかを変えることである。低次の分布は主として長い文脈では出現しなかった単語にスコアを付けるために呼び出されるため、コーパスでの生の出現頻度ではなく、その単語がどれほど容易に**新しい**文脈に入り込めるかを測るべきである。訓練テキストにおいて、`"traveller"` は 49 回出現するが直前の単語は 2 種類しかなく（`"time"` が 48 回、`"real"` が 1 回）、一方で `"said"` は 76 回出現し、48 種類の異なる先行語の後に続いている。生のユニグラム頻度を用いると、未知の文脈でも `"traveller"` を `"said"` とほぼ同じくらい積極的に予測してしまうが、異なる先行語の数を数えれば、`"traveller"` が `"time"` なしにはめったに現れないことを見抜ける（自然言語処理における古典的な例は `"Francisco"` であり、ニュース記事で頻出するにもかかわらず、その直前はほぼ例外なく `"San"` である）。そのため Kneser–Ney 平滑化では、最高次より下のすべての次数において $N(u, w)$ を**継続カウント**（continuation count）、すなわち訓練中に (u, w) の直前に現れた異なるトークンの数に置き換え、$N(u)$ と $S(u)$ も同様にその継続カウントから計算する。

さらに都合のよいことに、ディスカウントパラメータ D には Ney et al. (1994) が導出した簡潔な閉形式の推定値があり、各次数で $D = c_1/(c_1 + 2c_2)$ と与えられる。ここで c_1 と c_2 は、その次数においてちょうど 1 回および 2 回出現した異なる n グラムの数である（最高次より下では継続カウントを用いる）。

以下の関数 `kneser_ney` は Equation 8.13 を完全に実装したものである。最高次の n グラムを集計し、1 つ上の次数の異なるタプルからすべての低次の継続カウントを導出し、$N(u)$、$S(u)$、D を計算したうえで、トークン系列全体にわたって再帰式を評価し、利用可能なプレフィックスのもとでの各トークンの対数確率（log probability、単位は nat）を返す。

```
def kneser_ney(train, tokens, vocab_size, n=3):
    """Interpolated Kneser-Ney log-probabilities (nats) of each of tokens."""
    counts = {n: collections.Counter(zip(*(train[i:] for i in range(n))))}
    for k in range(n - 1, 0, -1):  # Lower orders: continuation counts
        counts[k] = collections.Counter(g[1:] for g in counts[k + 1])
    stats = {}
    for k, c in counts.items():
        c1, c2 = (sum(x == r for x in c.values()) for r in (1, 2))
        N, S = collections.Counter(), collections.Counter()
        for g, x in c.items():
            N[g[:-1]] += x
            S[g[:-1]] += 1
        stats[k] = c, N, S, c1 / max(c1 + 2 * c2, 1)
    logp = []
    for i in range(len(tokens)):
        p = 1 / vocab_size  # Below the unigram: uniform
        for k in range(1, min(n, i + 1) + 1):
            c, N, S, D = stats[k]
            g = tuple(tokens[i - k + 1:i + 1])
            if N[g[:-1]]:  # An unseen context passes p down unchanged
                p = (max(c[g] - D, 0) + D * S[g[:-1]] * p) / N[g[:-1]]
        logp.append(math.log(p))
    return logp
```

検証分割全体にスコアを付ける前に、おなじみの文脈 "the time" で Kneser–Ney 平滑化の動きを見てみよう。以下では、トライグラムのディスカウント D を表示し、3 つの後続語候補に対して Kneser–Ney 平滑化とラプラス平滑化が割り当てる確率を比較する。候補は、"traveller"（訓練中に "the time" が 87 回出現したうち 47 回続いた語）と、訓練中に 1 度も後に続かなかった 2 語 "was" および "filby" である。

```
c3 = collections.Counter(zip(words_train, words_train[1:], words_train[2:]))
c1, c2 = (sum(x == r for x in c3.values()) for r in (1, 2))
print(f'trigram discount D = {c1 / (c1 + 2 * c2):.2f}')
for w in ('traveller', 'was', 'filby'):
    lp = kneser_ney(words_train, ['the', 'time', w], vocab_size)[-1]
    print(f'P({w} | the time): Kneser-Ney {math.exp(lp):.2g}, '
          f'Laplace {models[3].prob(["the", "time"], w):.2g}')
```

```
trigram discount D = 0.93
P(traveller | the time): Kneser-Ney 0.53, Laplace 0.36
P(was | the time): Kneser-Ney 0.00037, Laplace 7.7e-05
P(filby | the time): Kneser-Ney 2.3e-05, Laplace 7.7e-05
```

トライグラムのディスカウントは約 0.9 であり、観測されたすべてのトライグラムからほぼ丸々 1 回分を差し引いている。これは 1 回きりの観測がいかにノイズを多く含むかを鮮やかに物語っている。カウントが 47 ある頻出トライグラムでは、0.9 を引いてもほとんど影響はなく、"traveller" は経験頻度 $47/87$ の近くに保たれるのに対し、ラプラス平滑化はその確率質量の 3 分の 1 近くを語彙全体へと浪費してしまう。未知の後続語における対比はさらに鮮明である。ラプラス平滑化が "was" と "filby" をまったく同じに扱うのに対し、Kneser–Ney 平滑化は "was" が幅広い文脈の後に続く一方で "filby" はごく限られた文脈にしか現れないことを見抜き、両者の確率に 1 桁以上の差をつける。これによってホールドアウトテキストでどれほど改善するかを測定してみよう。

```
for n in (1, 2, 3):
    kn = kneser_ney(words_train, words_val, vocab_size, n)
    print(f'{n}-gram held-out perplexity, Kneser-Ney: '
          f'{math.exp(-sum(kn) / len(kn)):7.1f}')
```

```
1-gram held-out perplexity, Kneser-Ney:   592.8
2-gram held-out perplexity, Kneser-Ney:   312.3
3-gram held-out perplexity, Kneser-Ney:   300.1
```

Kneser–Ney 平滑化によってランキングは完全に逆転する。ユニグラムでさえラプラス平滑化を上回り、バイグラムはパープレキシティをほぼ半減させ、トライグラムは他のすべてのカウントモデルを凌駕する。未知のトライグラムが一様な当て推量に潰れるのではなく、適切に較正（calibration）されたバイグラムおよびユニグラムの継続推定値へと滑らかにバックオフするからである。もっとも、トライグラムによるバイグラムからの改善幅は控えめである。ホールドアウトのトライグラムの大半は依然として未知であるため、トライグラムモデルは過半の位置でバイグラムへとフォールバックしているからである。**未知の**テキストで低いパープレキシティを達成することこそ言語モデルの決定的な試金石であり、続くニューラルモデルでもパープレキシティを主要なベンチマークとして用いる。

それでもなお、Kneser–Ney 平滑化はあらゆるカウントベースの手法と共通する根本的な限界を抱えている。平滑化はゼロから確率質量を再配分するだけであり、未知の組み合わせのうちどれが意味的にもっともらしいかを推論することはできない。カウントテーブルは各トークンを分割不能で互いに関係のない記号として扱うため、"cat" と "feline" が統語的・意味的な役割を共有

していることも、"I want to eat, grandma" が無害であるのに対し "I want to eat grandma" が物騒であることも知る由がない。関連する単語や文脈の間で統計的な強みを共有するには、学習された連続表現が不可欠であり、これこそが 8.4 節でニューラルネットワークへと向かう直接の動機となる。

8.3.2.4 バイトあたりビット数

パープレキシティには 1 つ重大な制約がある。それはトークン化の選択に依存するという点である。文字レベルのモデルは各ステップで 27 個の記号から選ぶのに対し、単語レベルのモデルは数千語から選ぶため、生のパープレキシティはまったく異なるスケール上にあり、直接比較することができない。しかしトークン化はモデルの内部設計上の選択にすぎず（8.2 節）、公正な評価指標はそれに対して不変でなければならない。幸い、解決策は単純であり、分母を置き換えればよい。総サプライザルをトークン数で割る代わりに、評価対象テキストの**バイト**数（テキスト自体のみによって決まる量）で割るのである。サプライザルをビット単位（底 2 の対数）で測れば、**バイトあたりビット数**（bits per byte、bpb）が得られる。

$$\text{bpb} = \frac{1}{B}\sum_{t=1}^{T} -\log_2 P(x_t \mid x_{<t}), \tag{8.14}$$

ここで、B は評価対象テキストの UTF-8 バイト長である。バイトあたりビット数は、可逆圧縮（lossless compression）の圧縮率として直ちに実用的な意味を持つ。算術符号器と組み合わせれば、bpb バイトあたりビット数を達成するモデルはテキストを元のバイトサイズの $\text{bpb}/8$ に圧縮できる。

パープレキシティとバイトあたりビット数を直接比較するため、正規化した同じ訓練テキスト上で、まずラプラス平滑化、次に Kneser–Ney 平滑化を用いて、文字、単語、および 1,024 トークンの BPE 語彙（8.2 節のトークナイザを使って正規化済み訓練分割で訓練したもの）という 3 通りのトークン化のもとでトライグラムモデルを当てはめる。ただし、比較を公正に保つには 1 つ注意すべき細部がある。単語モデルが <unk> を予測した時点では、語彙外（out-of-vocabulary）の**どの**単語が出現したのかをまだ特定していないのである。ホールドアウトテキストを可逆的に符号化するには、単語モデルは未知語を 1 文字ずつ綴り出す必要があるため、1 文字あたり（末尾の空白も含めて）$\log_2 27$ ビットを加算する。これは正規化された 27 種類の記号から各文字を一様に符号化するコストである。

```
bpe = d2l.BPETokenizer(vocab_size=1024, pattern=d2l.BPETokenizer.GPT2_PATTERN)
bpe.train(text[:split])
streams = {'character': (list(text[:split]), list(text[split:]),
                         len(set(text))),
           'BPE (1,024)': (bpe.encode(text[:split]), bpe.encode(text[split:]),
                           bpe.vocab_size),
           'word': (words_train, words_val, vocab_size)}
n_bytes = len(text[split:].encode('utf-8'))
spell = sum((len(w) + 1) * math.log(27)  # Nats to spell out the <unk> words
            for w in text[split:].split() if w not in known)
print(f'{"tokenization":>12} {"|V|":>5} {"tokens":>7} {"ppl":>7} '
      f'{"bpb":>5} {"KN bpb":>6}')
for name, (train, val, m) in streams.items():
    nll = eval_nll(NGramLM(train, 3, m), val)
    nll_kn = -sum(kneser_ney(train, val, m))
    extra = spell if name == 'word' else 0
```

```
        print(f'{name:>12} {m:>5} {len(val):>7} {math.exp(nll / len(val)):>7.1f} '
              f'{(nll + extra) / math.log(2) / n_bytes:>5.2f} '
              f'{(nll_kn + extra) / math.log(2) / n_bytes:>6.2f}')
```

```
tokenization   |V|  tokens     ppl   bpb KN bpb
   character    27   17345     6.4  2.68   2.67
 BPE (1,024)  1024    5509   281.6  2.58   1.90
        word  4316    3373  2481.4  2.81   2.22
```

生のパープレキシティで測ると、ラプラス平滑化を用いた 3 つのモデルには 2 桁以上の開きがあるように見える。しかしバイトあたりビット数で測ると、3 つは互いに 3 分の 1 ビット以内の範囲に収まり、パープレキシティでは圧倒的に優位に見えた文字モデルはもはや首位ではなくなる。固定されたマルコフ次数 n に対して、粒度の粗いトークンほどテキスト上のより長い範囲をカバーする。2 語の文脈は約 10 文字先まで遡れるのに対し、2 文字では 1 音節すらほとんどカバーできないからである。もし語彙外単語のペナルティがなければ単語モデルが首位に立つところだが、その綴り出しコストが約 0.6 バイトあたりビット数だけ上乗せされるため、ラプラス平滑化では最下位に転落する。バイトレベルの BPE には語彙外（OOV）のペナルティが一切なく、最も高い圧縮性能を達成しており、現代の言語モデルがサブワードトークン化（subword tokenization）を採用するもう 1 つの理由を裏付けている。

最後の列を見ると、Kneser–Ney 平滑化は BPE と単語のトライグラムの両方でバイトあたりビット数を 0.5 ビット以上削り落としている一方、文字トライグラムはほとんど変化していない。記号が 27 種類しかなければ未知の文字トライグラムはごくわずかであり、平滑化が働く余地がほとんどないためである。約 1.9 バイトあたりビット数を達成する Kneser–Ney 平滑化の BPE トライグラムが、カウントベースの最強のベースラインとなる。

したがって、異なる語彙上に構築された現代の言語モデル同士（たとえば 50,000 トークンのモデルと 200,000 トークンのモデル）を比較する際には、バイトあたりビット数が標準的な評価指標となる。スケール感を掴むには 3 つの基準が参考になる。Shannon による人間の予測実験という古典的研究では、印刷された英語のエントロピーが 1 文字あたり約 1 ビットと推定された (Shannon 1951)。現代のニューラル言語モデルは多様なウェブテキストを 1 バイトあたり 1 ビットを優に下回る水準まで圧縮する。そして本節のカウントベースのモデルは、およそ 1.9〜2.8 バイトあたりビット数の間に位置している。

8.3.3 系列の分割

次節以降のニューラル言語モデルは、8.2 節のトークナイザが生成した固定長の BPE トークン部分系列のミニバッチを入力として受け取る。ここでそのデータパイプラインを組み立てよう。

トークン化されたコーパスを、トークン ID の単一の連続ストリーム $x_1, x_2, x_3, \ldots$ として思い描いてみよう。長さ T でインデックス t から始まる入力ウィンドウは $x_t, \ldots, x_{t+T-1}$ を含み、それと対になる目標は同じウィンドウを右に 1 トークンずらした $x_{t+1}, \ldots, x_{t+T}$ である。モデルが入力ウィンドウを順にたどる際、すべての位置で次のトークンを予測するため、各ウィンドウは教師ありの次トークン予測（next-token prediction）目標を T 個供給する。コーパス末尾の T トークンを除くすべての開始オフセットが有効なウィンドウを定め、隣接するウィンドウ同士は 1 トークンを除いてすべて重なり合う（Figure 8.6）。訓練時には、確率的勾配降下法（stochastic gradient descent）が必要とするシャッフルを実現するため、これらのウィンドウからランダムにミニバッチをサンプリングする。

以下の `TimeMachine` コンストラクタがこのパイプライン全体をまとめる。中編小説をダウンロードし、8.2 節の BPE トークナイザを訓練してテキストを整数 ID に符号化し、長さ `num_steps` の重複する全ウィンドウを対応する入力（`X`）および目標（`Y`）の配列へと積み重ねる。訓練済みトー

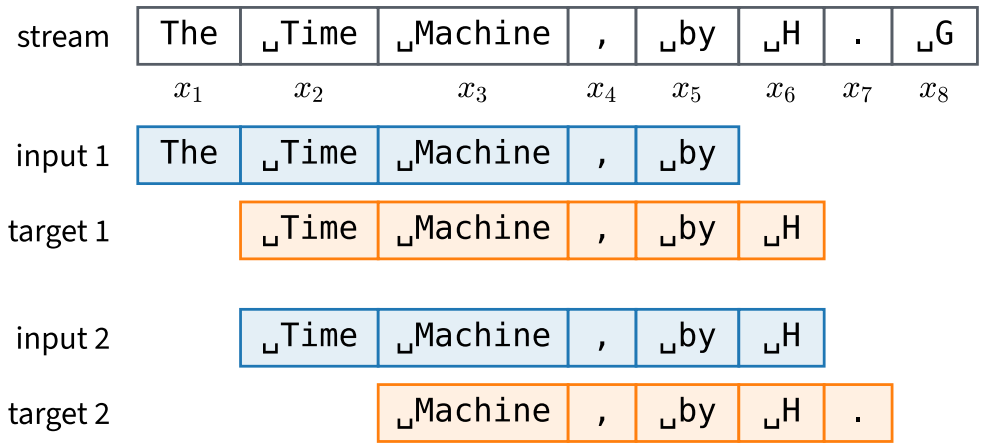


Figure 8.6: 『The Time Machine』冒頭の BPE トークンに対する長さ $T = 5$ の最初の 2 つの入力ウィンドウ（␣は先頭の空白を表す）。各目標は入力を 1 トークン先にずらしたものであり、連続するウィンドウは 1 トークンを除いて重なり合う。ミニバッチはランダムに抽出したウィンドウを集めて構成される。

クナイザは `data.tokenizer` と `data.vocab` の両方に格納されるため、ID をテキストへデコードできるほか、どのトークン化モードでも統一的に `len(data.vocab)` でクエリできる。

```
@d2l.add_to_class(d2l.TimeMachine)
def __init__(self, batch_size, num_steps, num_train=50000, num_val=5000,
             tokenization='bpe', vocab_size=1024):
    super(d2l.TimeMachine, self).__init__()
    self.save_hyperparameters()
    raw_text = self._download()
    if tokenization == 'bpe':
        self.tokenizer = d2l.BPETokenizer(
            vocab_size, pattern=d2l.BPETokenizer.GPT2_PATTERN)
        self.tokenizer.train(raw_text)
        corpus, self.vocab = self.tokenizer.encode(raw_text), self.tokenizer
    else:  # tokenization='char': characters, via TimeMachine.build
        corpus, self.vocab = self.build(raw_text)
    windows = [corpus[i:i+num_steps+1] for i in range(len(corpus)-num_steps)]
    # Drop the windows that straddle the split, so that no validation token
    # is a training target
    windows = windows[:num_train] + windows[num_train+num_steps:]
    array = d2l.tensor(windows)
    self.X, self.Y = array[:,:-1], array[:,1:]
```

先頭の `num_train` 個のウィンドウを訓練用に確保し、続く `num_val` 個のウィンドウを検証用に充てる。隣接するウィンドウは大きく重なり合っているため、訓練分割の直後に続くウィンドウをそのまま使うと、すでに訓練の目標として現れたトークンが含まれてしまう。そこで境界にある `num_steps` 個のウィンドウをバッファとして捨てることで、訓練と検証の間のデータリークを防ぐ。検証時には、ミニバッチはシャッフルせず先頭から順番に反復処理する。

```
@d2l.add_to_class(d2l.TimeMachine)
def get_dataloader(self, train):
    idx = slice(0, self.num_train) if train else slice(
        self.num_train, self.num_train + self.num_val)
    return self.get_tensorloader([self.X, self.Y], train, idx)
```

各ミニバッチは形状（batch_size、num_steps）の 2 つの整数テンソルからなる。サンプルの 1 行をデコードすると、X に Wells の文章が収められており、Y がちょうど 1 トークンだけ先にずれていることが確認できる。

```
data = d2l.TimeMachine(batch_size=2, num_steps=10)
for X, Y in data.train_dataloader():
    print('X:', X, '\nY:', Y)
    break
print('X[0]:', repr(data.tokenizer.decode([int(i) for i in X[0]])))
print('Y[0]:', repr(data.tokenizer.decode([int(i) for i in Y[0]])))
```

```
X: tensor([[ 302,  115,  280,  408,  277,  272,  360,  270,  432,  260],
        [ 353,  258, 1012,  480,  342,   46,  533,  260,   10,   84]])
Y: tensor([[ 115,  280,  408,  277,  272,  360,  270,  432,  260,  335],
        [ 258, 1012,  480,  342,   46,  533,  260,   10,   84,  355]])
X[0]: 'sts cast and filed into the'
Y[0]: 's cast and filed into the re'
```

BPE トークンのウィンドウは、いかなるカウントテーブルが活用できるよりもはるかに長い文脈を提供する（先ほどの単語テーブルは、わずか 2 語の履歴で早くもデータ不足に陥っていた）。次節では、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、任意の長さのプレフィックスを学習された隠れ状態（hidden state）へと圧縮する。

8.3.4 まとめ

言語モデルはトークン系列に確率を割り当て、確率の連鎖律によってこの課題は次トークン予測へと帰着するため、単一のモデルでテキストのスコア付けと生成の両方が可能になる。マルコフ仮定のもとで、n グラムモデルはコーパス内のカウントから次トークンの確率を直接推定する。構築は容易で局所的には流暢なサンプルを生成できるものの、指数関数的に膨張する文脈空間とジップ則に従う重い裾に阻まれ、観測された n グラムの大半がシングルトンとなって汎化ではなく丸暗記を引き起こす。トークンあたりの平均交差エントロピーの指数であるパープレキシティはモデルの実効分岐数を測り、ラプラス平滑化を用いたトライグラムがホールドアウトテキストで振るわない理由を明らかにする。補間型 Kneser–Ney 平滑化は観測カウントをディスカウントし、解放された質量をより低次の継続カウントへと再配分することで最強のカウントベースモデルをもたらすが、意味的に関連するトークン間で汎化することは依然としてできない。また、パープレキシティは語彙の粒度に依存するため、バイトあたりビット数がトークン化に不変な圧縮率としてモデル間の公平な比較を可能にする。最後に、次節以降のニューラル言語モデルへ入力するため、BPE トークン化されたコーパスを固定長の重複する入力・目標ウィンドウへと変換し、検証分割へのリークを防ぐ明確なギャップを設けた。

8.3.5 演習問題

1. **密な格納とスパースな格納**。語彙サイズ $|\mathcal{V}| = 50{,}000$ の単語トークン上の 5 グラムモデルを考える。

1. 条件付き確率 Equation 8.8 の密なテーブルは何個のエントリを含むか。
2. 代わりに、1 兆トークンのコーパスで観測された 5 グラムのみを 1 エントリあたり 16 バイトで格納するとする。必要なメモリは最大でどれほどか。また、取り得るすべての 5 グラムのうちどれだけの割合がカバーされるか。
3. 8.3.1 節の単語語彙は $|\mathcal{V}| = 4{,}316$ エントリであり、そこのコードは訓練テキスト内の異なる単語トライグラムの数を表示している。取り得るすべてのトライグラムのうち、訓練に出現する割合はどれだけか。また、ホールドアウトのトライグラムのうち訓練に出現する割合はどれだけか。したがって、(b) で計算したカバー率は、新しいテキストに対するクエリがテーブル内で該当する 5 グラムを見つけられる頻度の良い推定値といえるか。

2. [code] **平滑化定数**。単語のユニグラム、バイグラム、トライグラムの各モデルについて、Equation 8.12 の α を $\alpha = 10^{k/4}$（$k = -16, -15, \ldots, 2$）の 19 個の値にわたって変化させ、ホールドアウト・パープレキシティを対数軸の α に対してプロットせよ。

   1. 最良の α は n に応じてどのように変化するか。また、それはなぜか。
   2. 3 つのモデルをそれぞれの最良の α におけるホールドアウト・パープレキシティによって順位付けせよ。このランキングは、8.3.2 節で `NGramLM` のデフォルト値 $\alpha = 0.01$ を用いて表示したランキングとどのように異なるか。

3. **偏りのある分布でのパープレキシティ**。10 個の記号からなる訓練コーパスにおいて、1 つの記号が 91% の割合で出現し、残りの 9 個の記号がそれぞれ 1% の割合で出現するとする。

   1. このコーパスから推定されたユニグラムモデルについて、同じ分布から抽出された長いテストテキスト上でのパープレキシティを計算せよ。語彙サイズの 10 と比べてどうなるか。
   2. 8.3.2 節の単語レベルのユニグラムモデルは数千種類の単語から選択する。そのホールドアウト・パープレキシティを等確率な選択肢の実効数として読み取り、ジップの法則 Equation 8.9 を用いて、この値が $|\mathcal{V}|$ よりはるかに小さくなる理由を説明せよ。

   Daniel Jurafsky、James H. Martin『Speech and Language Processing』第 3 版ドラフト、演習問題 3.12 を改変。

4. **重複するウィンドウ**。`d2l.TimeMachine` は、コーパス内の取り得るすべての開始位置に対して連続する `num_steps + 1` トークンからなる行を 1 行ずつ構築する。各行の最初の `num_steps` トークンが `X` の 1 行をなし、最後の `num_steps` トークンが対応する `Y` の 1 行をなす。最初の `num_train` 行が訓練に回り、次の `num_steps` 行は捨てられ、その後の `num_val` 行が検証に回る（8.3.3 節）。ここで `num_steps=32`、`num_train=10000`、`num_val=5000` とする。

   1. 入力トークンと目標トークンを合わせて数えたとき、訓練行はコーパスのどの位置に触れ、検証行はどの位置に触れるか。もし行をまったく捨てなかった場合、両方が触れる位置は何個になり、なぜそれによって検証スコアが不当に押し上げられるのか。
   2. 訓練範囲の内部にあるトークンは、1 エポックあたり何回、目標として使われるか。
   3. 約 66,000 トークンのコーパス全体を重複しない 32 トークンのウィンドウに分割すると、何個のウィンドウが得られるか。この分割を 10,000 個の重複する訓練行と、ウィンドウ数およびカバーされる異なるコーパス位置の数の両面で比較せよ。

5. [code] **異なる長さにわたる正規化**。●`NGramLM` には系列終了トークンがない。3 記号の語彙上の短い文字列で、バイグラムの `NGramLM` を $\alpha = 1$、`vocab_size=3` として訓練せよ。`prob` と連鎖律 Equation 8.6 を用いて、長さ 2 のすべての系列の確率の和と、長さ 3 のすべての系列の確率の和をそれぞれ別個に求めよ。

   1. それぞれの和は何に等しいか。また、この 2 つの和は、長さが事前に固定されていない系列上の分布としてのこのモデルについて何を示しているか。

2. 語彙に系列終了トークンを追加して `vocab_size=4` とし、訓練文字列の末尾にもそのトークンを追加せよ。このトークンを出力した時点で系列が完結するものとして扱う。完結するすべての系列の確率の総和が厳密に 1 になることを示せ。決して終わらない系列に確率質量が残る可能性を排除しているのは、モデルのどの性質か。

Daniel Jurafsky、James H. Martin『Speech and Language Processing』第 3 版ドラフト、演習問題 3.5 を改変。

6. [code] **既知のトライグラムと未知のトライグラム。**●ホールドアウトの 3 語目以降の位置を、その位置で終わる単語トライグラムが訓練中に出現したかどうかによって 2 つのグループに分けよ。バイグラムとトライグラムの単語モデルを、それぞれ問題 2 の探索で得られた最良の α に設定し、各グループ上での各モデルの平均の負の対数尤度を計算せよ。トライグラムの方が優れたモデルとなるのはどちらのグループか。また、Kneser–Ney 平滑化 Equation 8.13 はこれをどのように活用しているか。

8.4 リカレントニューラルネットワーク

8.3 節のカウントに基づく言語モデルは、直前の短いウィンドウ内のトークンから各トークンを予測する。トライグラムモデルが条件づけるのは直前の 2 トークンのみであり、文書内のそれより前の部分は予測に一切影響しない。単にウィンドウを広げようとしてもすぐに壁にぶつかる。20 個の BPE トークンにまたがるカウント表は $1024^{20} \approx 10^{60}$ 個もの文脈を追跡しなければならず、連結した m 個のトークン埋め込みからなる固定ウィンドウを読み取るニューラルネットワーク (Bengio et al. 2003) でさえ、$m+1$ ステップ以上前のトークンはまったく参照できないうえ、パラメータ数が m に対して線形に増加してしまう。

8.1 節では、根本的に異なるアプローチとして、履歴全体を固定サイズの**隠れ状態**（hidden state）ベクトル $\mathbf{h}_{t-1}$ に圧縮し、その要約から次のトークンを予測する**潜在**自己回帰モデル（latent autoregressive model）を導入した。

$$P(x_t \mid x_{<t}) \approx P(x_t \mid \mathbf{h}_{t-1}), \qquad \mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t). \tag{8.15}$$

この漸化式のもとでは、プレフィックスがどれほど長くなっても、1 ステップあたりの計算コストは一定に保たれる。**リカレントニューラルネットワーク**（RNN）は、すべてのタイムステップで重みを共有するニューラルネットワークの層によって遷移関数 g をパラメータ化する (Elman 1990)。（なお、タイムステップをまたいで引き継がれる**隠れ状態**は、単一ステップ内で入力と出力の間にあるフィードフォワードの**隠れ層**（hidden layer）とは異なる。）本節では、RNN 言語モデルを定式化し、そのパラメータ数と計算量を数え上げたうえで、『The Time Machine』を対象にゼロからの実装とフレームワークの高水準層を用いた実装の両方を作成する。

8.4.1 隠れ状態を持つリカレントニューラルネットワーク

RNN は、単一隠れ層の MLP（4.1 節）から出発し、そこにリカレント結合を 1 つ追加する形で考えると最も理解しやすい。b 個の例からなり各例が d 個の特徴量を持つミニバッチ $\mathbf{X} \in \mathbb{R}^{b \times d}$ が与えられたとき（本章を通じてバッチサイズには b を用いるが、これは n が n-gram の次数を表すためである）、活性化関数 ϕ を持つ通常の隠れ層は次式を計算する。

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{\text{xh}} + \mathbf{b}_{\text{h}}), \tag{8.16}$$

ここで、$\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$ および $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$ であり、これに続いて出力射影（output projection）$\mathbf{O} = \mathbf{H}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}$ により q 個のロジットへと写す。

ここで、ミニバッチ入力の系列 $\mathbf{X}_t \in \mathbb{R}^{b \times d}$（タイムステップ t で添字づけられる）を受け取るとしよう。隠れ表現に前のステップを記憶させるため、直前の隠れ状態 $\mathbf{H}_{t-1} \in \mathbb{R}^{b \times h}$ をリカレント重み行列 $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$ を介して層へフィードバックする。

$$\begin{aligned}\mathbf{H}_t &= \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}),\\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}.\end{aligned} \tag{8.17}$$

同じ 5 つのパラメータテンソル（リカレント層の $\mathbf{W}_{\text{xh}}$、$\mathbf{W}_{\text{hh}}$、$\mathbf{b}_{\text{h}}$ と出力層の $\mathbf{W}_{\text{hq}}$、$\mathbf{b}_{\text{q}}$）が（$\mathbf{H}_0 = \mathbf{0}$ から始めて）すべてのステップで再利用されるため、パラメータ数は系列長に一切依存しない（Figure 8.7）。時間方向に展開した形で見ると、T ステップにわたって動作する RNN は、すべての層が同じ重みを共有する T 層のフィードフォワードネットワークである。

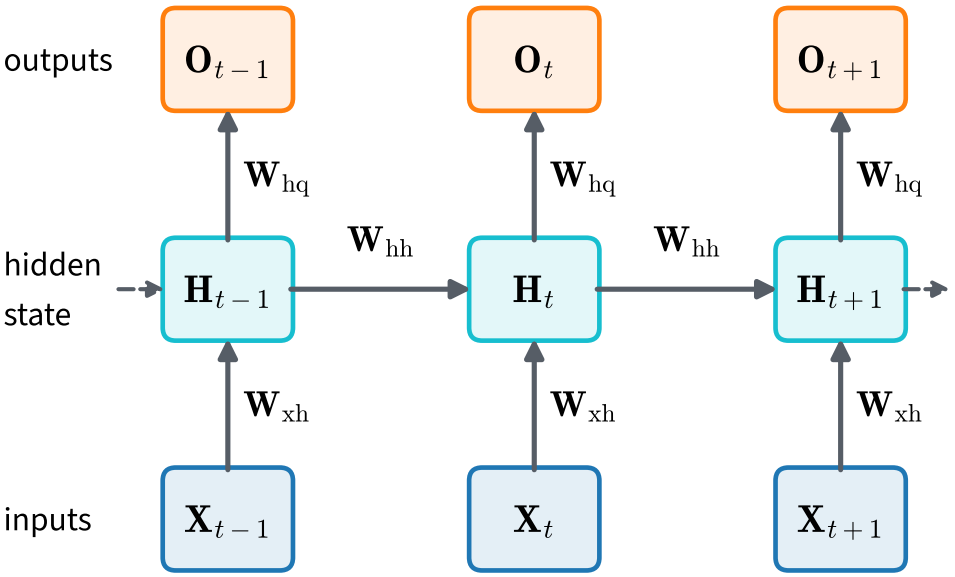


Figure 8.7: 隣接する 3 つのタイムステップにわたって展開した、隠れ状態を持つ RNN。すべてのステップで同じ重みが再利用される。

Equation 8.17 における 2 つの行列積は、水平方向に連結した入力と垂直方向に積み重ねた重みとの単一の積に等しい。

$$[\mathbf{X}_t, \mathbf{H}_{t-1}] \begin{bmatrix}\mathbf{W}_{\text{xh}}\\ \mathbf{W}_{\text{hh}}\end{bmatrix} = \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}}.$$

それにもかかわらずライブラリがこの 2 つの積を分けているのは、訓練時にはすべての入力 $\mathbf{X}_1, \ldots, \mathbf{X}_T$ が事前にわかっているため、$\mathbf{X}_t \mathbf{W}_{\text{xh}}$ を全 T ステップ分まとめて逐次的な時間ループの外でバッチ計算できるからである（演習問題 1）。

さらに、再帰状態（recurrent state）を用いれば、ビット列に含まれる 1 の累積個数が偶数か奇数か（パリティ）を毎ステップ報告する課題のように、**いかなる**固定ウィンドウモデルにも不可能なタスクを解くことができる。任意の固定ウィンドウ長 m より前のビットを 1 つ反転させる

と、直近 m ビットは変わらないまま真のパリティだけが反転してしまうのに対し、2 ユニットの再帰状態であれば任意の長さでパリティを追跡できる。

8.4.1.1 ステップあたり一定のメモリ量

$\mathbf{H}_t$ の計算に必要なのは $\mathbf{H}_{t-1}$ と $\mathbf{X}_t$ だけであるため、1 ステップあたりのメモリ量と計算量は系列長にかかわらず一定に保たれる。$h = 128$ 個の `float32` 隠れユニットを用いる場合、系列の状態が占めるのは、モデルが 10 トークン読んだ後でも 100 万トークン読んだ後でもわずか 512 バイトである。しかし、この一定のメモリ使用量は本質的に**非可逆な**（lossy）ボトルネックでもある。プレフィックスから記憶されるあらゆる詳細がこの 512 バイトを通過しなければならず、系列長とともに増大するコストと引き換えに過去の全トークンの表現を保持する**アテンション**（第 10 章）とは対照的である。

8.4.1.2 RNN 言語モデルと教師強制

RNN 言語モデルは 1 ステップにつき 1 つのトークン ID を受け取り、全 $q = |\mathcal{V}|$ 個の語彙項目にわたる次トークンの分布を予測する（Figure 8.8）。ステップ t では、ミニバッチ内の b 個のトークン ID が学習済みの埋め込みテーブル（embedding table）$\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ から対応する行を参照し、$\mathbf{X}_t \in \mathbb{R}^{b \times d}$ を形成する（これは $b \times |\mathcal{V}|$ のワンホット行列に $\mathbf{E}$ を掛けることと数学的に同一だが、ゼロを実体化したり乗算したりせずに済む）。リカレント層が $\mathbf{H}_t$ を更新し、$\mathrm{softmax}(\mathbf{O}_t)$ が $P(x_{t+1} \mid x_{\le t})$ を定める。これは連鎖律による分解 $P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$ を構成する条件付き確率である（8.3 節）。

パラメータと FLOPs はどこに集中しているのだろうか。BPE 語彙サイズ $|\mathcal{V}| = 1{,}027$ 個の ID（1,024 個の BPE トークンと 3 つの特殊トークン）、埋め込み幅 $d = 64$、隠れ状態のサイズ $h = 128$ の場合、内訳は次のようになる。

構成要素	形状	パラメータ数
埋め込みテーブル $\mathbf{E}$	$1{,}027 \times 64$	65,728
入力重み $\mathbf{W}_{\mathrm{xh}}$	64×128	8,192
リカレント重み $\mathbf{W}_{\mathrm{hh}}$	128×128	16,384
隠れバイアス $\mathbf{b}_{\mathrm{h}}$	128	128
出力重み $\mathbf{W}_{\mathrm{hq}}$	$128 \times 1{,}027$	131,456
出力バイアス $\mathbf{b}_{\mathrm{q}}$	$1{,}027$	1,027
合計		222,915

リカレント層が占めるのはわずか 24,704 パラメータ（モデル全体の 11%）および 1 トークンあたり $24{,}576$ 回の積和演算であるのに対し、語彙サイズの埋め込み層と出力層がパラメータの 89% を保持し、出力射影だけで $128 \cdot 1{,}027 = 131{,}456$ 回の積和演算を要する。

訓練時には、前のステップでモデルが何を予測したかにかかわらず、毎ステップの入力にはコーパスからの**正解**（ground-truth）トークンが与えられる。この方式を**教師強制**（teacher forcing）と呼ぶ。ウィンドウ内のすべての正解入力が事前にわかっているため、ウィンドウ全体の交差エントロピー損失（cross-entropy loss、パープレキシティの対数）を 1 回の順伝播で評価できる。しかし生成時には、サンプリングまたは貪欲選択された各トークンが次の入力としてフィードバックされるため、8.1 節の多段階正弦波予測と同様に、ロールアウトは自身の誤差にさらされることになる。

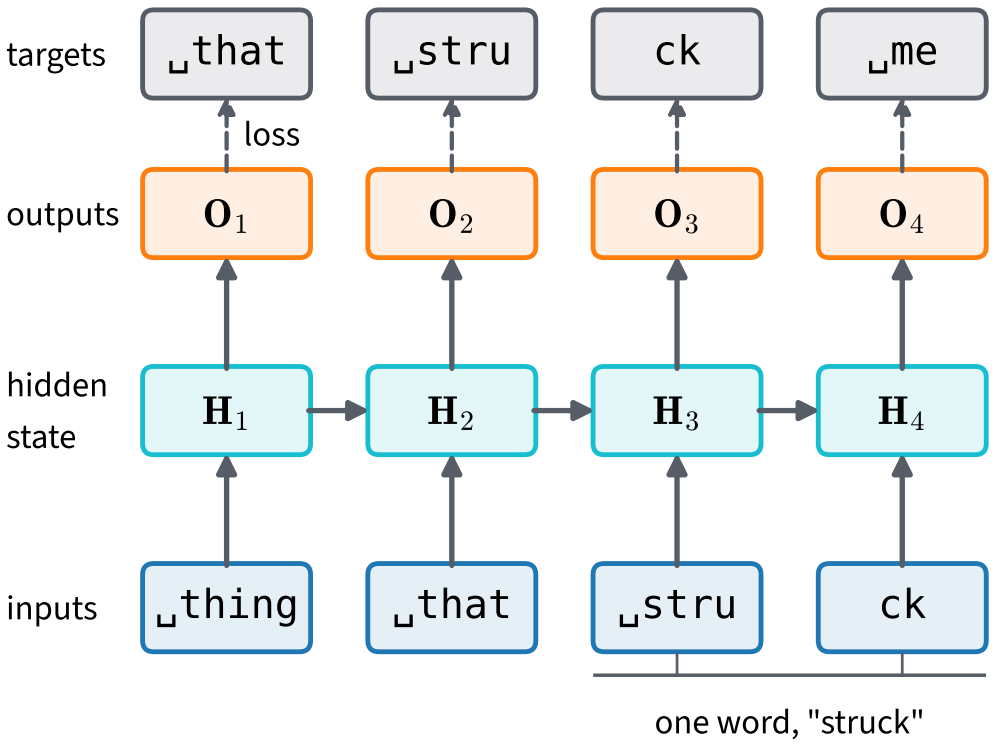


Figure 8.8: 『The Time Machine』の BPE トークン上で教師強制により訓練される RNN 言語モデル。ターゲット系列は入力を 1 トークン前にずらしたものであり、各ステップは次のトークンを予測するよう訓練される。

8.4.2 RNN 言語モデルのゼロからの実装

ここからは、この言語モデルの 3 つの構成要素（リカレントセル、埋め込み参照、出力層）をゼロから実装する。

8.4.2.1 リカレントセル

RNNScratch クラスは Equation 8.17 のパラメータを $\phi = \tanh$ として保持する。$\mathbf{W}_{\text{xh}}$ と $\mathbf{W}_{\text{hh}}$ は $\mathcal{N}(0, \sigma^2)$（ただし $\sigma = 0.01$）から抽出し、$\mathbf{b}_{\text{h}}$ はゼロで初期化する。このスケールでは、$\mathbf{W}_{\text{hh}}$ のスペクトル半径（spectral radius）は約 $\sigma\sqrt{h} \approx 0.11$ となるため、未訓練のセルは過去の状態を急速に減衰させ、必要に応じて訓練によってリカレント重みを成長させる設計になっている。

```
class RNNScratch(d2l.Module):
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W_xh = nn.Parameter(
            d2l.randn(num_inputs, num_hiddens) * sigma)
        self.W_hh = nn.Parameter(
            d2l.randn(num_hiddens, num_hiddens) * sigma)
        self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
```

順伝播では、inputs（形状は（num_steps、batch_size、num_inputs））の先頭にある時間軸に沿ってループを回し、隠れ状態が与えられない場合はゼロで初期化したうえで、ステップごとの全隠れ状態のリストを最終状態とともに返す。

```
@d2l.add_to_class(RNNScratch)
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

リカレントセルの次元を確認するため、まず RNNScratch の順伝播をダミーのミニバッチ X（形状 (num_steps, batch_size, num_inputs)）に対して実行する。

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

ここでヘルパー関数 check_shape（以降の章でも再利用する）を定義し、outputs をスタックすると (num_steps, batch_size, num_hiddens) になり、state の形状が (batch_size, num_hiddens) になることを確認する。

```
def check_shape(a, shape):
    """Check the shape of a tensor."""
    assert a.shape == shape, \
            f'tensor\'s shape {a.shape} != expected shape {shape}'

check_shape(d2l.stack(outputs, 0), (num_steps, batch_size, num_hiddens))
check_shape(state, (batch_size, num_hiddens))
```

8.4.2.2 埋め込み層と出力層

RNNLMScratch は、リカレントセル rnn を埋め込みテーブル E と出力射影（W_hq、b_q）で包み、さらに埋め込みと隠れ状態に対する任意のドロップアウト率を備える。埋め込みの各行は（単位スケールのワンホット入力に合わせて）$\mathcal{N}(0,1)$ から初期化し、出力重みはスケール $\sigma = 0.01$、バイアスはゼロとするため、未訓練のモデルはゼロに近いロジットを出力し、初期パープレキシティは $|\mathcal{V}|$ に近くなる。d2l.Classifier を継承することで平均交差エントロピー損失が提供され、validation_step をオーバーライドしてその損失を直接記録することで、その指数から検証パープレキシティが得られる。

```
class RNNLMScratch(d2l.Classifier):
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01, dropout=0, emb_dropout=0):
        super().__init__()
        self.save_hyperparameters()
        self.sigma = getattr(rnn, 'sigma', 0.01)  # Scale of the output weights
        self.init_params()

    def init_params(self):
        self.E = nn.Parameter(
            d2l.randn(self.vocab_size, self.rnn.num_inputs))
        self.W_hq = nn.Parameter(
            d2l.randn(self.rnn.num_hiddens, self.vocab_size) * self.sigma)
        self.b_q = nn.Parameter(d2l.zeros(self.vocab_size))

    def validation_step(self, batch):
        return self.loss(self(*batch[:-1]), batch[-1])
```

データローダは形状（batch_size、num_steps）のミニバッチを生成するため、embedding はトークン ID を転置してから E を参照し、先頭の軸を時間とする（num_steps、batch_size、num_inputs）テンソルを返す。

```
@d2l.add_to_class(RNNLMScratch)
def embedding(self, X):
    # Output shape: (num_steps, batch_size, num_inputs)
    return self.E[X.T]
```

output_layer はステップごとの状態を形状（batch_size、num_steps、num_hiddens）にスタックし、1 回のバッチ行列積で共有の出力射影を適用する。一方、forward は埋め込み参照、ドロップアウト、リカレントセル、出力射影を順に連結する。

```
@d2l.add_to_class(RNNLMScratch)
def output_layer(self, rnn_outputs):
    # A list of per-step states, or a tensor whose first axis is time
    H = d2l.stack(list(rnn_outputs), 1)  # (batch_size, num_steps, num_hiddens)
    return d2l.matmul(H, self.W_hq) + self.b_q

@d2l.add_to_class(RNNLMScratch)
def forward(self, X, state=None):
    embs = F.dropout(self.embedding(X), self.emb_dropout, self.training)
    rnn_outputs, _ = self.rnn(embs, state)
    H = d2l.stack(list(rnn_outputs), 0)  # (num_steps, batch_size, num_hiddens)
    return self.output_layer(F.dropout(H, self.dropout, self.training))
```

形状 (batch_size, num_steps) のトークンインデックスからなるダミーのミニバッチを RNNLMScratch に通すと、返されるロジットテンソル outputs の形状が (batch_size, num_steps, vocab_size) であることが確認できる。

```
model = RNNLMScratch(rnn, vocab_size=1024)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, 1024))
```

8.4.3 勾配クリッピング

展開された RNN は T ステップにわたって同じ遷移ヤコビ行列を適用するため、勾配は T に対して指数関数的に縮小または増大しうる（8.5 節）。突発的な勾配のスパイクが生じると、パラメータはユークリッド距離で $\eta\|\mathbf{g}\|$ だけ移動し、1 次のテイラー近似が成り立つ局所領域を大きく飛び越えて、何十回分もの更新を 1 ステップで台無しにしてしまう。η を一律に小さくする代わりに、**勾配クリッピング**（gradient clipping）は（すべてのモデルパラメータにわたって一括計算した）大域的な勾配ノルムをしきい値 θ で頭打ちにする。具体的には、$\mathbf{g}$ を半径 θ の球へ射影する。

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{8.18}$$

長さが θ を超える勾配は方向を変えずに長さ θ に縮小され、それより短い勾配はそのまま保たれる（Figure 8.9）。

gradient_clip_val が正のとき、d2l.Trainer（2.4 節）は誤差逆伝播法の直後かつオプティマイザのステップの前に Equation 8.18 を適用する。

以下のメソッドは、各パラメータの勾配をインプレースで再スケーリングする。

```
@d2l.add_to_class(d2l.Trainer)
def clip_gradients(self, grad_clip_val, model):
    params = [p for p in model.parameters() if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum(p.grad ** 2) for p in params))
    scale = torch.clamp(grad_clip_val / norm, max=1.0)  # no `if` on the host
    for param in params:
        param.grad.mul_(scale)
```

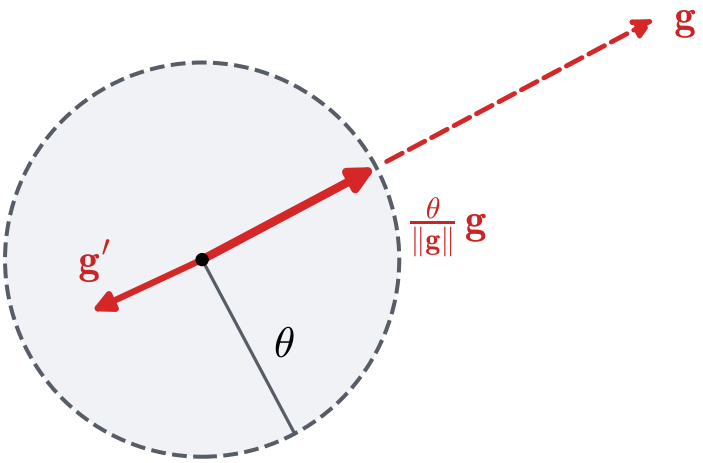


Figure 8.9: 半径 θ の球への射影としての勾配クリッピング。長い勾配 $\mathbf{g}$ は自身の方向に沿って長さ θ に再スケーリングされる一方、球の内部にある勾配 $\mathbf{g}'$ は変化しない。

8.4.4 訓練と評価

`d2l.TimeMachine`（8.3 節）はこの中編小説の 65,741 個の BPE トークンを符号化し、長さ $T = 32$ の重複するウィンドウからなるミニバッチを生成する。訓練用が 50,000 個、検証用が 5,000 個であり、検証トークンが訓練ターゲットに現れないよう両者の間には 32 ウィンドウ分の間隔が空けられている。各ウィンドウは $\mathbf{H}_0 = \mathbf{0}$ から始まるため、誤差逆伝播法は 32 ステップで**打ち切られる**（truncated、8.5 節）。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

8.4.4.1 単純なモデルとクリッピングの役割

まず、222,915 パラメータのベースラインモデル（$d = 64$、$h = 128$、ドロップアウトなし）を SGD の学習率 $\eta = 4$ でインスタンス化する。

```
rnn = RNNScratch(num_inputs=64, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4)
```

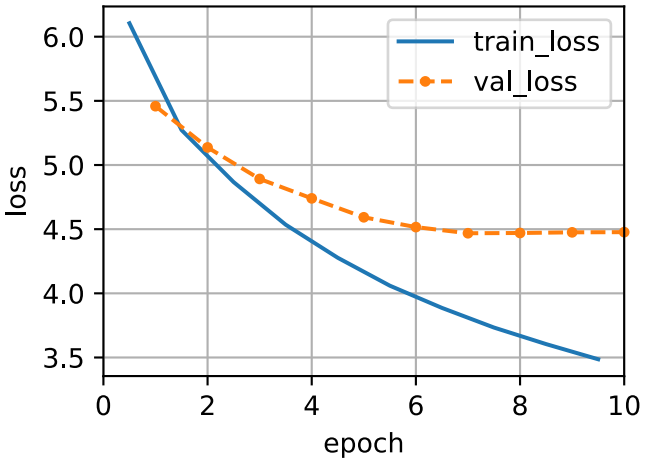
訓練を始める前に、ランダムな初期埋め込み行列 `model.E` の NumPy コピーを `E_init` に保存しておき、SGD によってどれだけ移動するかを測定できるようにする。

```
E_init = d2l.numpy(model.E).copy()
```

次に、未訓練モデルの初期検証パープレキシティ `ppl_init` を評価してから、`model` を 10 エポックにわたり `gradient_clip_val=1` の勾配クリッピング付きで訓練する。

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
ppl_init = math.exp(trainer.evaluate(model, data)['loss'])
print(f'initial validation perplexity {ppl_init:.0f}')
trainer.fit(model, data)
```

```
initial validation perplexity 1027
```



最終的な検証交差エントロピー損失 `trainer.history['val_loss'][-1]` の指数をとると、訓練済みベースラインの検証パープレキシティ `ppl_plain` が得られる。

```
ppl_plain = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_plain:.1f}')
```

```
validation perplexity 88.0
```

初期検証パープレキシティは $|\mathcal{V}| = 1{,}027$ 付近にあり、10 エポック後にクリッピングありのモデルは検証パープレキシティ約 85〜100 に達する。ただし最初のエポックの途中では、損失が初期値を大きく上回って跳ね上がることがよくある（一方、222,915 個のパラメータが 50,000 個の訓練トークンに過学習し始めるため、訓練パープレキシティはその半分以下にまで下がる）。

$\eta = 4$ において勾配クリッピングが不可欠である理由を見るため、まったく同じアーキテクチャと学習率を持つ新しいベースラインインスタンスを構築するヘルパー関数 `unclipped_model()` を定義する。

```
def unclipped_model():
    return RNNLMScratch(RNNScratch(num_inputs=64, num_hiddens=128),
                        vocab_size=len(data.vocab), lr=4)
```

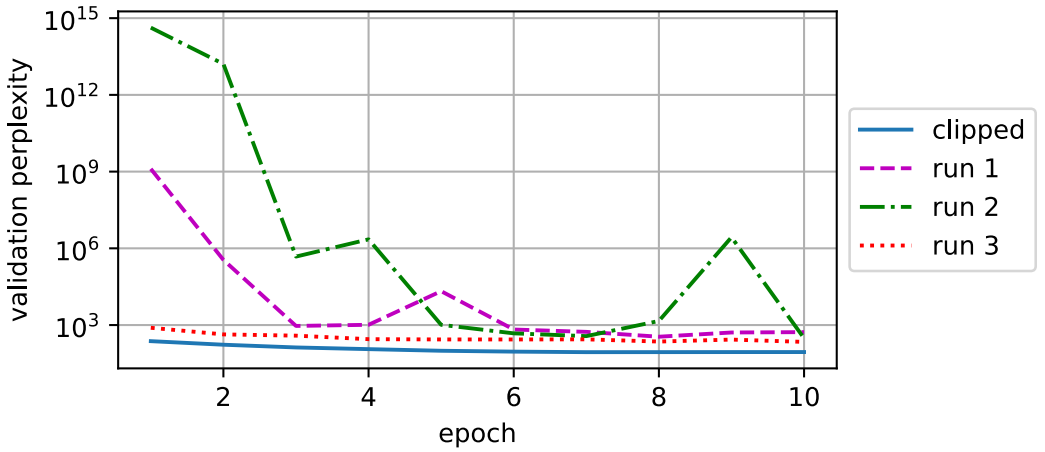
ここで、シード未固定の独立した 3 つのインスタンスを勾配クリッピングなしで訓練し、各実行のエポックごとの検証パープレキシティを `ppl_runs` に記録して、その最大値と最終値を報告する。

```
ppl_runs = []
for run in range(3):  # Three unseeded runs, no clipping
    trainer_unclipped = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer_unclipped.board.display = False  # One plot for all runs below
    trainer_unclipped.fit(unclipped_model(), data)
    ppls = [math.exp(loss) for loss in trainer_unclipped.history['val_loss']]
    ppl_runs.append(ppls)
    print(f'run {run + 1}: highest {max(ppls):.3g}, final {ppls[-1]:.3g}')
finals = [ppls[-1] for ppls in ppl_runs]
print(f'{sum(f > 2 * ppl_plain for f in finals)} of 3 runs end above twice '
      f'the clipped {ppl_plain:.3g}; finals from {min(finals):.3g} to '
      f'{max(finals):.3g}')
```

```
run 1: highest 1.28e+09, final 529
run 2: highest 4.26e+14, final 304
run 3: highest 782, final 219
3 of 3 runs end above twice the clipped 88; finals from 219 to 529
```

`ppl_runs` に格納したクリッピングなしの 3 本の軌跡を、クリッピングありのベースライン `ppl_clipped` とともに対数軸上にプロットすると、その不安定さは一目瞭然となる。

```
ppl_clipped = [math.exp(loss) for loss in trainer.history['val_loss']]
d2l.plot(list(range(1, 11)), [ppl_clipped] + ppl_runs, 'epoch',
         'validation perplexity', yscale='log',
         legend=['clipped', 'run 1', 'run 2', 'run 3'],
         legend_loc='outside', figsize=(5, 2.5))
```



クリッピングなしでは、勾配のスパイクによって検証パープレキシティが激しく跳ね上がり、未訓練ベースラインの 1,027 をはるかに超えることさえある。そして大半の実行は、クリッピングありのモデルの 2 倍を超えるパープレキシティで終了する。クリッピングありの検証パープレキシティをバイトあたりビット数（bits per byte、8.3.2.4 節）に変換するには、$\log_2$ ppl を BPE トークンあたりの平均 UTF-8 バイト数（≈ 2.8）で割る。

```
ids = d2l.numpy(data.X[data.num_train:data.num_train+data.num_val, 0]).tolist()
bytes_per_token = len(data.tokenizer.decode(ids).encode('utf-8')) / len(ids)
print(f'{bytes_per_token:.2f} bytes/token, '
      f'{math.log2(ppl_plain) / bytes_per_token:.2f} bits per byte')
```

```
2.78 bytes/token, 2.32 bits per byte
```

8.4.4.2 SGD 下での埋め込みとドロップアウト付き幅広モデル

10 エポックの SGD を通じて $\mathbf{E}$ の各行がどれだけ移動したかを調べると、第 2 のボトルネックが浮かび上がる。まず、訓練済みの埋め込みテーブルを NumPy 配列 `E_final` に取り出す。

```
E_final = d2l.numpy(model.E)
```

すべての語彙行にわたって相対ユークリッド変位 `change`（`E_final` と `E_init` の差）を計算すると、各トークンベクトルがいかにわずかしか動いていないかが明らかになる。

```
change = (((E_final - E_init) ** 2).sum(1) / (E_init ** 2).sum(1)) ** 0.5
print(f'mean change of an embedding row: {change.mean():.1%} of its length')
```

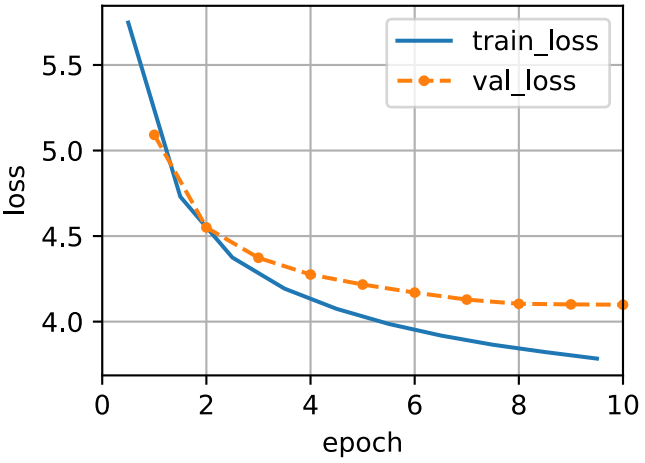
```
mean change of an embedding row: 4.3% of its length
```

第 i 行はトークン i が出現した位置からのみ勾配を受け取るうえ、損失はミニバッチ内の全 $1{,}024 \times 32 = 32{,}768$ トークンにわたって平均されるため、一様な学習率の SGD では低頻度や中頻度のトークンが受け取る勾配はごく小さくなり、$\mathbf{E}$ はほぼ凍結されたままになる。Adam（9.5 節）のような適応的オプティマイザは座標ごとのステップサイズを再スケーリングして埋め込みをはるかに大きく動かすが（演習問題で扱う）、SGD の枠内で直接対処することもできる。埋め込み幅を $d = 64$ から $d = 512$ へと広げれば、密に更新される入力行列 $\mathbf{W}_{\text{xh}}$ にはるかに豊かなランダム特徴量基底が与えられ、さらにドロップアウト（埋め込みに 0.1、隠れ状態に 0.5）を加えることで過学習を抑えられる。

```
rnn = RNNScratch(num_inputs=512, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                     emb_dropout=0.1)
```

この正則化された幅広モデルを、同じ学習率 `lr=4` とクリッピングしきい値 `gradient_clip_val=1` を用いて 10 エポック訓練する。

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



最終的な検証損失から、検証パープレキシティ ppl_scratch と、それに対応する圧縮率であるバイトあたりビット数の両方を計算する。

```
ppl_scratch = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_scratch:.1f}, '
      f'{math.log2(ppl_scratch) / bytes_per_token:.2f} bits per byte')
```

```
validation perplexity 60.3, 2.13 bits per byte
```

検証パープレキシティは 85〜100 から 60 前後へと低下し、バイトあたりビット数は 2.3〜2.4 から約 2.1 へと下がる。

8.4.4.3 Kneser–Ney 法との比較と実効コンテキスト範囲

バイトあたり 2.1 ビットという値は、補間 Kneser–Ney 平滑化（8.3.2.3 節）と比べてどうだろうか。Kneser–Ney トライグラムが達成したバイトあたり 1.9 ビット（8.3 節）は、そのまま比較基準にはできない。この数値は英小文字と空白のみに整形したテキスト上で測定されたものであるのに対し、本節では大文字・句読点・改行を含む元の小説をそのままモデル化しているからである。そこで、Kneser–Ney n-gram（$n = 2, 3, 4$）をまったく同じ 50,000 個の BPE 訓練トークンに当てはめ、val[32:] 上で評価する。ここでは各トークンが、それぞれの 32 トークン検証ウィンドウの最終ターゲットとしてスコア付けされる。

```
first_tokens = d2l.numpy(data.X[:, 0]).tolist()  # First token of each window
train = first_tokens[:data.num_train]
val = first_tokens[data.num_train:data.num_train + data.num_val]
lp_kns = {n: d2l.kneser_ney(train, val, len(data.vocab), n)[32:]
          for n in (2, 3, 4)}
for n, lp in lp_kns.items():
    bits = -sum(lp) / len(lp) / math.log(2)
    print(f'KN {n}-gram: {bits / bytes_per_token:.2f} bits per byte')
lp_kn = lp_kns[3]
```

```
KN 2-gram: 2.24 bits per byte
KN 3-gram: 2.09 bits per byte
KN 4-gram: 2.08 bits per byte
```

訓練済み RNN をまったく同じホールドアウトトークン `val[32:]` 上でスコア付けするため、検証ウィンドウのスライス `[start:stop]`（その最終ターゲットが `val[32:]` を網羅する範囲）を選択する。

```
start, stop = data.num_train, data.num_train + data.num_val - 32  # val[32:]
```

`model` を評価モードにして `data.X[start:stop]` 上で 1,024 ウィンドウずつのバッチとして実行し、ウィンドウ内の全 32 ステップにわたる位置ごとのターゲット対数確率の行列 `logp` を収集する。

```
model.eval()
with torch.inference_mode():  # Log-probability of every target
    logp = d2l.numpy(torch.cat([
        F.log_softmax(model(X.to(d2l.try_gpu())), -1).gather(
            -1, Y.to(d2l.try_gpu())[..., None])[..., 0]
        for X, Y in zip(data.X[start:stop].split(1024),
                        data.Y[start:stop].split(1024))]))
```

RNN とトライグラムはほぼ同じ平均値に到達する。では、両者は同じ予測をしているのだろうか。`logp` の最終列は、トークン列 `val[32:]` に対して `lp_kn` と同じ対象をスコア付けしているため、2 つのモデルを 1 トークンずつ比較できる。あわせて、ニューラル言語モデルとカウントに基づくモデルを組み合わせる定石である等重み混合 $0.5\,P_{\text{RNN}} + 0.5\,P_{\text{KN}}$ もスコア付けする (Mikolov et al. 2011)。

```
lp_rnn, lp_tri = logp[:, -1], np.array(lp_kn)
lp_mix = np.logaddexp(lp_rnn, lp_tri) + math.log(0.5)  # 0.5 P_RNN + 0.5 P_KN
for name, lp in [('RNN', lp_rnn), ('KN trigram', lp_tri), ('mixture', lp_mix)]:
    print(f'{name}: {-lp.mean() / math.log(2) / bytes_per_token:.2f} '
          'bits per byte')
print(f'RNN more confident on {np.mean(lp_rnn > lp_tri):.0%} of the tokens')
```

```
RNN: 2.12 bits per byte
KN trigram: 2.09 bits per byte
mixture: 2.03 bits per byte
RNN more confident on 50% of the tokens
```

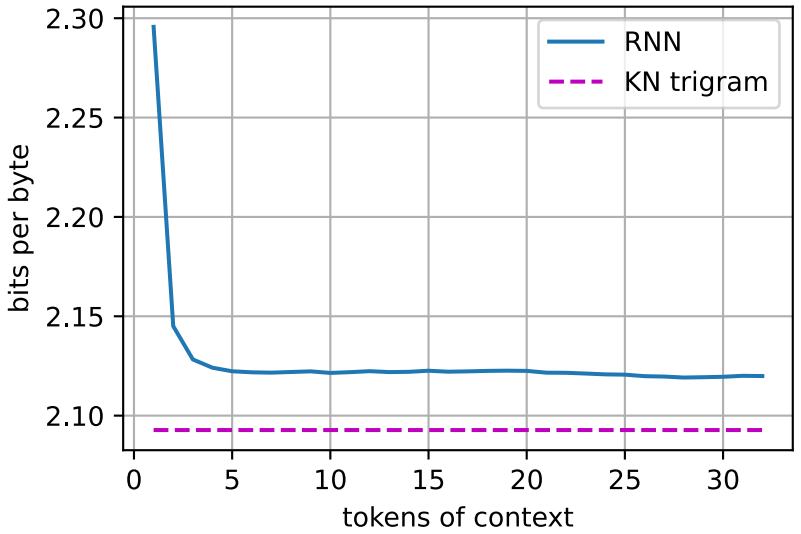
2 つのモデルは平均では引き分けるが、トークン単位ではそうではない。約半数のトークンでは RNN が、残りの半数ではトライグラムがより高い確率を割り当てており、混合モデルはトライグラムよりバイトあたり約 0.06 ビット低くなり、単独のどちらのモデルよりも優位に立つ。

混合モデルが改善を得られるのは構成モデル同士の予測が食い違う箇所に限られるため、2 つのモデルはテキストについて部分的に異なる事柄を学習していることになる。そうなると、この引き分けはいっそう謎めいて見える。32 トークン分の履歴を読み取るリカレントネットワークが、わずか 2 トークンしか振り返らない Kneser–Ney トライグラムと同じバイトあたり ≈ 2.1 ビット付近に着地するのはなぜだろうか。訓練済み RNN が実際にどれだけの文脈を利用しているかを調べるため、各ウィンドウ位置 $1, \ldots, 32$ における平均検証バイトあたりビット数を Kneser–Ney トライグラムのベースラインに対してプロットする。

```
bits = -logp.mean(0) / math.log(2) / bytes_per_token  # One per position
kn_bits = -sum(lp_kn) / len(lp_kn) / math.log(2) / bytes_per_token
positions = np.arange(1, data.num_steps + 1)
```

```
d2l.plot(positions, [bits, [kn_bits] * data.num_steps],
         'tokens of context', 'bits per byte', legend=['RNN', 'KN trigram'],
         figsize=(4.5, 3))
print('bits per byte with 1, 2, 3, 4, 8, 32 tokens of context:',
      [round(float(bits[k - 1]), 2) for k in (1, 2, 3, 4, 8, 32)])
```

```
bits per byte with 1, 2, 3, 4, 8, 32 tokens of context: [2.3, 2.15, 2.13, 2.12, 2.12, 2.12]
```



この診断曲線から引き分けの理由がわかる。改善のほぼすべては 2 トークン目の文脈で得られており、損失はバイトあたり約 2.3 ビットから約 2.14 ビットへと約 0.15 下がる。3 トークン目がもたらす改善は 0.01〜0.02 程度にとどまり、位置 4 から 32 まで曲線は平坦である。順伝播において隠れ状態は全 32 トークンに依存**しうる**ものの、訓練されたモデルが実際に活用しているのは直近の 2〜3 トークンだけである。

なぜこれほど少ないのだろうか。1 つの説明は最適化にある。離れたトークンを利用するようセルに教えるはずの勾配が、逆方向へ伝わるステップごとに縮小してしまうというものであり、これは 8.5 節で測定する。もう 1 つの説明はデータにある。50,000 個の訓練トークンでは、2〜3 個以上前の位置にあるトークンから学習できる情報がそもそもほとんど含まれていないのかもしれない。上で出力した Kneser–Ney モデルの結果は 2 番目の説明を裏づけている。これらのモデルはカウントによって当てはめられるため消失しうる勾配が存在しないにもかかわらず、同じ箇所で飽和する。すなわち、トライグラムはバイグラムよりバイトあたり 0.15 ビット改善するのに対し、4-gram のトライグラムからの改善はわずか 0.01 にすぎない。ここでの実験だけでは、この 2 つの説明を切り分けることはできない。明確に検証するには、同じトークン列で訓練されつつ長いラグにわたって勾配が生き残るモデルが必要であり、第 12 章ではそのようなモデルを構築する。

8.4.5 テキスト生成

訓練が完了すると、この言語モデルは `predict` を通じて自己回帰的にテキストを生成する。まずユーザーのプレフィックスのトークン列上で隠れ状態をウォームアップし、続いて選択した各トークンを次の入力としてフィードバックしながら `num_tokens` 個の後続トークンを生成する（`temperature=0.0` のときは貪欲な argmax を用い、それ以外のときは温度スケーリングされたソフトマックスからサンプリングする。8.6 節を参照）。

```
@d2l.add_to_class(RNNLMScratch)
@torch.inference_mode()  # Generation needs no autograd graph
def predict(self, prefix, num_tokens, tok, device=None, temperature=0.0):
    outputs, state = tok.encode(prefix), None
    for i in range(len(outputs) - 1):  # Warm up on the prefix
        X = d2l.tensor([[outputs[i]]], device=device)
        _, state = self.rnn(self.embedding(X), state)
    for _ in range(num_tokens):  # Generate num_tokens continuation tokens
        X = d2l.tensor([[outputs[-1]]], device=device)
        rnn_outputs, state = self.rnn(self.embedding(X), state)
        logits = self.output_layer(rnn_outputs)[0, 0]
        if temperature == 0:  # Greedy: the highest-scoring token
            token = logits.argmax()
        else:  # Sample from the softmax of logits / temperature
            token = torch.distributions.Categorical(
                logits=logits / temperature).sample()
        outputs.append(int(token))
    return tok.decode(outputs)
```

model.predict を貪欲デコーディング（greedy decoding、temperature=0.0）でテストし、プロンプト 'The Time Traveller' から 50 トークンの続きを生成してみよう。

```
d2l.print_wrapped(repr(model.predict('The Time Traveller', 50,
                                     data.tokenizer, d2l.try_gpu())))
```

```
'The Time Traveller,\nthe\nfrom the\nfound,\ntheir\nfound,\ntheir\nfound,\nt
    heir\nfound,\ntheir\nfound,\ntheir\nfound,\ntheir\n'
```

貪欲デコーディングは決定論的であるため、隠れ状態が直近のフレーズに近い状態を再訪すると、そのループを際限なく繰り返してしまう。この失敗モードについては 8.6 節で詳しく調べ、対処する。

8.4.6 簡潔な実装

フレームワークが提供するリカレント層は、リカレントステップと時間ループを最適化された低水準カーネルに融合している。以下の RNN モジュールは RNNScratch のインターフェースおよび初期化と一致しており、RNNLM は RNNLMScratch を継承して組み込みの埋め込み層と線形層に差し替える。

nn.RNN は、入力項用とリカレント項用の 2 つのバイアスベクトルを保持する。その和が $\mathbf{b}_\mathrm{h}$ の役割を果たし、どちらもゼロで初期化される。

```
class RNN(d2l.Module):
    """The RNN model implemented with high-level APIs."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = nn.RNN(num_inputs, num_hiddens)
        for name, param in self.rnn.named_parameters():
            if name.startswith('weight'):
                nn.init.normal_(param, std=sigma)
```

```
            else:  # Both bias vectors start at zero
                nn.init.zeros_(param)

    def forward(self, inputs, H=None):
        return self.rnn(inputs, H)
```

nn.Embedding はデフォルトで各行を $\mathcal{N}(0,1)$ から抽出する。一方、nn.Linear の重みは明示的に設定する。

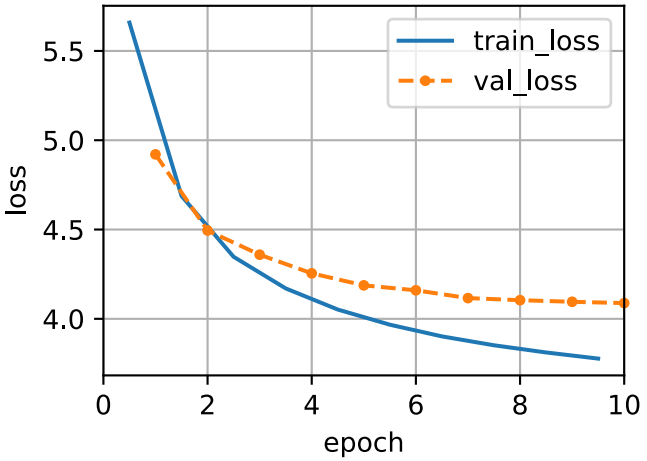
```
class RNNLM(d2l.RNNLMScratch):
    """The RNN-based language model implemented with high-level APIs."""
    def init_params(self):
        self.emb = nn.Embedding(self.vocab_size, self.rnn.num_inputs)
        self.linear = nn.Linear(self.rnn.num_hiddens, self.vocab_size)
        nn.init.normal_(self.linear.weight, std=self.sigma)
        nn.init.zeros_(self.linear.bias)

    def embedding(self, X):
        return self.emb(X.T)

    def output_layer(self, hiddens):
        return d2l.swapaxes(self.linear(hiddens), 0, 1)
```

この高水準 API による実装を検証するため、RNN と RNNLM を同じハイパーパラメータ（$d = 512$、$h = 128$、およびドロップアウト）でインスタンス化する。

```
rnn = RNN(num_inputs=512, num_hiddens=128)
model = RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
              emb_dropout=0.1)
```

ここで RNNLM を 10 エポックにわたり gradient_clip_val=1 で訓練する。融合されたリカレントカーネルが動作するため、ゼロからの実装に比べてごくわずかな実時間で完了する。

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



最後に、この簡潔なモデルの検証パープレキシティ ppl_concise を、'The Time Traveller' から貪欲に生成した 30 トークンの続きとともに報告する。

```
ppl_concise = math.exp(trainer.history['val_loss'][-1])
pred = model.predict('The Time Traveller', 30, data.tokenizer, d2l.try_gpu())
d2l.print_wrapped(f'perplexity {ppl_concise:.1f}, {pred!r}')
```

```
perplexity 59.6, "The Time Traveller.\n\n'I
    had\nfossibly\nthere\nfoundly\nfossibly\nthere\nf"
```

8.4.7 まとめ

リカレントニューラルネットワークは、すべてのタイムステップで共有される重みを用いて現在の入力と直前の状態から固定サイズの隠れ状態を更新し、履歴を非可逆な状態ベクトルに圧縮する代償として、1 ステップあたり一定のメモリ量と計算量を実現する。RNN 言語モデルは、埋め込みテーブル、tanh リカレントセル、および語彙サイズの出力射影（ベースラインのパラメータの 89% がここに存在する）を組み合わせ、教師強制によってトークンあたりの交差エントロピーを最小化するよう訓練される。大域的な勾配クリッピング（Equation 8.18）は、高い学習率のもとで勾配爆発のスパイクが訓練を破綻させるのを防ぐ。『The Time Machine』では、ドロップアウト付きの幅広な埋め込みにより、SGD で訓練した RNN はバイトあたり ≈ 2.1 ビットに達し、補間 Kneser–Ney トライグラムと引き分ける。訓練されたバニラ RNN が実質的に活用しているのは直近の 2〜3 トークンの文脈のみであり、その勾配がどこまで届くかは 8.5 節で測定する。

8.4.8 演習問題

1. **入力積の分離。** $\mathbf{X}_t \in \mathbb{R}^{b\times d}$ と $\mathbf{H}_{t-1} \in \mathbb{R}^{b\times h}$ を Equation 8.17 における入力と状態とする。
    1. 訓練時、ウィンドウ内の入力 $\mathbf{X}_1, \ldots, \mathbf{X}_T$ は最初のタイムステップを計算する前にすべて既知である。2 つの積 $\mathbf{X}_t\mathbf{W}_{\text{xh}}$ と $\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$ のうち、全 T ステップ分を 1 回の行列積でまとめて計算できるのはどちらか、またその各因子の形状は何か。タイムステップに関する逐次ループの内側に残さなければならない積はどちらか、またそれはなぜか。
    2. ループ内に残る 1 ステップあたりの計算量を、連結形式 $[\mathbf{X}_t, \mathbf{H}_{t-1}]$ の 1 ステップあたりの計算量と比較せよ。生成時には、ステップ t の入力はステップ $t-1$ で生成されたトークンとなる。このときも分離形式の利点は保たれるか。

2. **露出バイアス**。教師強制（8.4.1.2 節）ではすべての訓練ステップが真のプレフィックスに条件づけられるのに対し、生成時には各ステップがモデル自身の過去の出力に条件づけられる。この不一致を**露出バイアス**と呼ぶ (Ranzato et al. 2016)。
    1. 生成時に各トークンをモデルの次トークン分布からサンプリングするものとし、プレフィックスが正しい限りモデルは正しい次トークンに確率 $1 - \epsilon$ を割り当てるとする。生成された T 個のトークンに誤りが 1 つも含まれない確率はいくらか。また、$\epsilon = 0.01$、$T = 100$ のときの値を求めよ。
    2. **スケジュールドサンプリング** (Bengio et al. 2015) は、訓練中にモデルを自身の出力にさらす手法であり、各ステップである確率に従い、真のトークンの代わりに直前のステップにおけるモデルの予測からサンプリングしたトークンを入力とする。これは 8.4.1.2 節が依拠している教師強制のどの性質を手放すことになり、それによって訓練ウィンドウの計算はどのように変わるか。

3. **漸化式の内部でのウィンドウ表現**。ニューラル n-gram は直近 m 個の入力埋め込みからなる固定ウィンドウを読み取る。Equation 8.17 において ϕ を恒等写像、$\mathbf{b}_\text{h} = \mathbf{0}$ とし、状態に $h = md$ 個のユニットを持たせる。
    1. $\mathbf{W}_\text{xh}$ と $\mathbf{W}_\text{hh}$ を適切に選び、$\mathbf{H}_0 = \mathbf{0}$ から始めたとき、状態が連結 $\mathbf{H}_t = [\mathbf{X}_t, \mathbf{X}_{t-1}, \dots, \mathbf{X}_{t-m+1}]$（すべての $t \geq m$ に対して）となるようにせよ。
    2. $\mathbf{W}_\text{hh}^m = \mathbf{0}$ であることを示し、このことから m ステップより前の入力が $\mathbf{H}_t$ に影響しない理由を説明せよ。
    3. $d = 256$、$m = 20$ のとき、この構成には何個の状態ユニットが必要か。状態が 256 ユニットしかない RNN は、ウィンドウをそのまま保持する代わりに何をしなければならないか説明せよ。

4. [code] **積としての埋め込み参照**。$\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ を埋め込みテーブルとし、$\mathbf{x}$ に b トークンからなるミニバッチのトークン ID を格納するものとする。
    1. 入力を、一度は $\mathbf{E}$ の行のうち $\mathbf{x}$ が指定するものを選択する方法で、もう一度は $b \times |\mathcal{V}|$ の $\mathbf{x}$ のワンホット行列と $\mathbf{E}$ との積として計算し、両者が一致することを確認せよ。$|\mathcal{V}| \in \{10^3, 10^4, 10^5\}$ および $d = b = 256$ に対して両方の実行時間を計測し、それぞれの実行時間が $|\mathcal{V}|$ に対してどのように増加するか述べよ。
    2. 選択された行の全要素の和を、自動微分によって $\mathbf{E}$ に関して微分せよ。勾配のうち非ゼロとなるのはどの行か、またあるトークンがミニバッチ内に複数回出現するとき、その行には何が入るか。$|\mathcal{V}| = 10^4$ とし、一様ランダムに抽出した $b = 256$ 個の ID に対して、非ゼロの勾配を受け取る行の割合を見積もれ。

5. **任意のウィンドウを超えるパリティ**。●$x_1, x_2, \dots$ をビット列とし、ステップ t で求める出力を $x_1 + \cdots + x_t$ のパリティとする。
    1. 有限の m をどのように選んでも、直近 m 個の入力しか読み取らないモデルは、m より長い系列のすべてのステップで正しい出力を生成できないことを示せ。
    2. ϕ をしきい値活性化関数とする。任意の系列長に対して毎ステップの出力がパリティに一致するような、隠れユニット数 2 の Equation 8.17 形式の RNN を構成せよ。具体的な重みを与え、隠れユニット 1 個では不十分であることを示せ。
    3. 以降のすべてのステップで出力が正しくなるために、隠れ状態は $x_1, \dots, x_t$ について何ビットの情報を保持しなければならないか。これを 8.4.1.1 節における「非可逆な固定サイズの要約」としての状態の説明と関連づけよ。

6. [code] **勾配クリッピングのアブレーション**。`d2l.Trainer` は、`gradient_clip_val` が正のとき常に射影 Equation 8.18 を適用する。
    1. 最初のエポックの各更新におけるクリッピング前の勾配ノルムを記録し、プロットせ

よ。クリッピングはどのくらいの頻度で、どのタイミングで作動するか。
2. 本節のクリッピングなし実行の大半は `lr=4` で跳ね上がった。3 回連続でクリッピングなし実行が検証パープレキシティの急騰を起こさずに訓練できるまで、学習率を半分ずつ下げよ。その 10 エポック後の検証パープレキシティは、`lr=4` でクリッピングした実行と比べてどうか。このことから、クリッピングは何を可能にしているといえるか。
3. `RNNScratch` の tanh を ReLU に置き換え、クリッピングありとなしの両方で再訓練せよ。クリッピングは依然として必要か。tanh セルとの違いを隠れ状態の値域の観点から説明せよ。

7. [code] **利用可能な履歴**。すべての訓練ウィンドウはゼロ状態から始まるため、モデルは `num_steps` トークンを超える文脈を使うようには一切訓練されておらず、位置別の損失からはそれよりはるかに少ない文脈しか使っていないことがわかった。対照的に `predict` は、プレフィックス全体と生成されたすべてのトークンにわたって状態を引き継ぐ。訓練済みモデルを連続する 512 個の検証トークン上で次の 2 通りの方法で評価せよ。すなわち、訓練時と同様に 32 トークンごとに状態をゼロにリセットする方法と、全 512 トークンにわたって状態を引き継ぐ方法である。両者の平均交差エントロピーを比較せよ。状態の引き継ぎは性能を改善するか、悪化させるか、あるいは差を生まないか。また、その答えは訓練時のコンテキスト範囲を超えてモデルが到達する状態について何を示唆しているか。

8. [code] **Adam**。幅広の埋め込みを持つモデルを、SGD の代わりに 9.5 節の Adam オプティマイザで訓練せよ。具体的には、`RNNLMScratch` を継承し、学習率 0.003 の Adam を返すよう `configure_optimizers` をオーバーライドし、クリッピングは 1 のまま維持せよ。

   1. `emb_dropout=0.2` とし、上の比較と同様にトークン列 `val[32:]` をスコア付けし、そのバイトあたりビット数を SGD モデル、Kneser–Ney トライグラムおよび 4-gram、さらに $\lambda = 0.5$ における RNN とトライグラムの混合モデルと比較せよ。
   2. `emb_dropout=0.1` と 0.2 のそれぞれについて、各エポック後の検証損失を記録せよ。検証損失が最小になるのは何エポック目か、またその後はどうなるか。
   3. 訓練中に埋め込みの各行がどれだけ移動するかを再度測定せよ。Adam が SGD よりも大きく埋め込みを動かすのはなぜか。

9. [code] **語彙サイズとバイトあたりビット数**。●幅広の埋め込みを持つモデルを、`vocab_size` を 512、1,024、2,048、4,096 として `d2l.TimeMachine` 上で再訓練せよ。ウィンドウ数は語彙サイズに依存するため、割合で分割する。すなわち、各データセットを一度構築して `n = len(data.X)` を読み取り、次に `num_train=int(0.8 * n)` および `num_val=n - int(0.8 * n)` を指定して再構築せよ。（後から `data.num_train` を書き換えると、検証トークンを訓練ターゲットから除外する処理が飛ばされてしまう。）これにより、すべてのモデルがほぼ同じ範囲のテキストで訓練・検証される。トークンあたりのパープレキシティは直接比較できないため、バイトあたりビット数で比較せよ。

   1. バイトあたりビット数が最も低くなるのはどの語彙サイズか。
   2. 埋め込みテーブルと出力層のパラメータ数、およびトークン単位でのコーパス長は、語彙サイズに応じてどのように変化するか。測定した傾向を最もよく説明するのはこれらの効果のうちどれか。

10. [code] **書籍間の汎化**。●`d2l.TimeMachine` を継承し、『The War of the Worlds』（Project Gutenberg ebook 36）など別の H. G. Wells の小説のテキストを返すよう `_download` をオーバーライドせよ。その上でトークナイザとモデルを訓練せよ。

    1. 新しい小説のトークナイザで両方を符号化し、その小説自身のホールドアウトテキストと『The Time Machine』のそれぞれでモデルを評価せよ。両方についてパープレキシティとバイトあたりビット数を報告せよ。
    2. $s = \log_2(\text{ppl})$ をトークンあたりのサプライザル、b を評価対象テキストのトー

クンあたりバイト数とすると、バイトあたりビット数は s/b となる。『The Time Machine』（TM）におけるバイトあたりビット数と、新しい小説のホールドアウトテキスト（own）におけるバイトあたりビット数の比は、次のように分解できる。

$$\frac{s_{\text{TM}}}{s_{\text{own}}} \cdot \frac{b_{\text{own}}}{b_{\text{TM}}}. \tag{8.19}$$

第 1 の因子はモデルの予測を反映し、第 2 の因子はトークナイザの圧縮効率を反映する。両方の因子を測定せよ。1 からより大きく離れているのはどちらか。

8.5 通時的誤差逆伝播法

8.4.2 節における RNN 言語モデルの実験からは、2 つの際立った経験的知見が得られた。勾配クリッピング（gradient clipping）なしでは大半の訓練が不安定化した一方、クリッピングを行うと訓練は安定して収束したものの、学習されたネットワークは 32 トークンのコンテキストウィンドウのうち直近の数トークンしか利用していなかったのである。両方の現象は、まったく同じ数学的根源を共有している。リカレントネットワークはすべてのタイムステップで同じ重み行列を適用するため、系列を逆方向に伝播する勾配には、各ステップで状態ヤコビ行列（state Jacobian）が乗じられる。多数のステップを経るうちにこれらの因子が複利のように積み重なり、勾配は幾何級数的に増大または縮小する。

スカラーの漸化式 $h_t = w_{\text{h}} h_{t-1} + x_t$ を見るだけでも、その理由は明らかである。これを展開すると $h_t = \sum_{s \le t} w_{\text{h}}^{t-s} x_s$ となり、したがって $\partial h_t / \partial h_{t-k} = w_{\text{h}}^k$ となる。40 ステップにわたると、$w_{\text{h}} = 0.9$ では勾配が $0.9^{40} \approx 0.015$ まで縮小するのに対し、$w_{\text{h}} = 1.1$ では $1.1^{40} \approx 45$ まで膨張する。勾配爆発（exploding gradients）は最適化を即座に不安定化させるため、更新のクリッピングを余儀なくされる。一方、勾配消失（vanishing gradients）は目立たないがより厄介であり、遠く離れたトークンからの誤差信号を減衰させ、モデルが長期依存関係を学習できなくしてしまう。

RNN における勾配計算に、新しい微分アルゴリズムは必要ない。漸化式を時間方向に展開して重みを共有する深いフィードフォワードネットワークとし、標準的な誤差逆伝播法（4.3 節）を適用すればよい。この手順は**通時的誤差逆伝播法**（backpropagation through time、BPTT）として知られる (Werbos 1990)。以下では、逆方向の漸化式を行列形式で直接導出し、8.4.2 節で訓練した RNN の内部で勾配が実際にどこまで伝わるかを測定したうえで、打ち切り型通時的誤差逆伝播法（truncated backpropagation through time）を検討する。

8.5.1 勾配消失と勾配爆発

T ステップにわたって展開され、リカレントパラメータを共有し、系列全体の平均損失を $L = \frac{1}{T} \sum_{t=1}^{T} l(\mathbf{y}_t, \mathbf{o}_t)$ とする RNN を考えよう。誤差逆伝播法は、重み行列が現れるすべてのタイムステップからの局所的な勾配の寄与を足し合わせることで、重み共有（weight sharing）を扱う。逆伝播（backward pass）の構造を見通しよく確認するため、8.4 節の RNN を取り上げ、バイアスと tanh 非線形性を一時的に省略する（tanh は直後に復活させる）。b 個の系列からなるミニバッチと各ステップの入力 $\mathbf{X}_t \in \mathbb{R}^{b \times d}$ に対し、線形 RNN は次式を計算する。

$$\begin{aligned} \mathbf{H}_t &= \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}}, \\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{\text{hq}}, \end{aligned} \tag{8.20}$$

ここで、$\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$、$\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$、$\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$ である。Figure 8.10 は、このネットワーク

を展開した 3 つのタイムステップを表しており、順伝播（forward pass）を灰色で、逆伝播を赤色で示している。

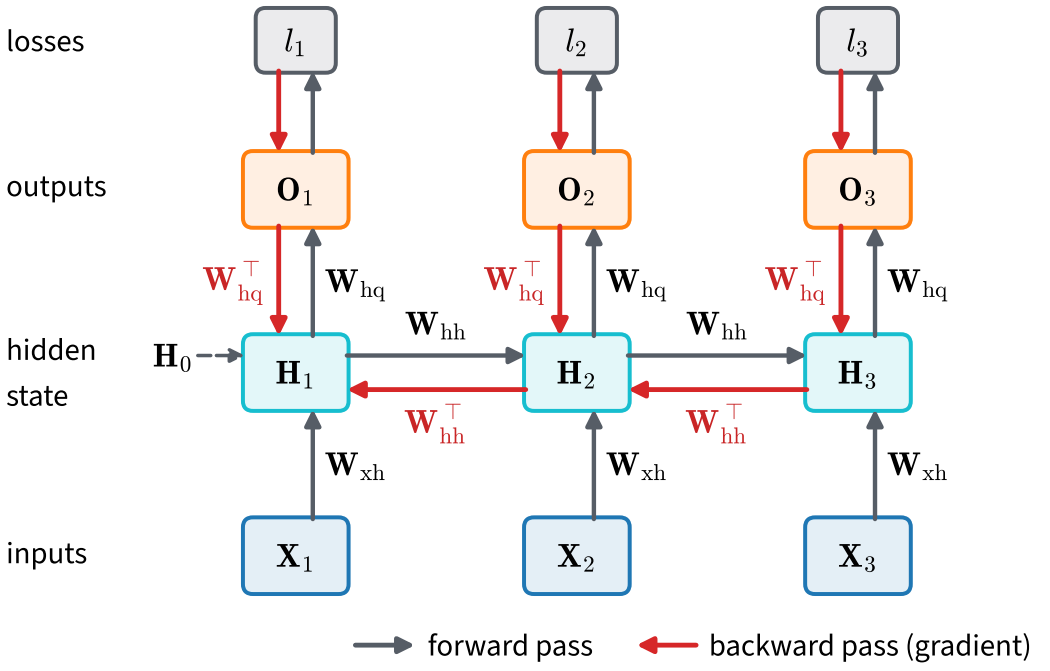


Figure 8.10: 3 つのタイムステップにわたって展開された線形 RNN の順伝播（灰色）と逆伝播（赤色）。最後のステップを除く各隠れ状態に到達する勾配は、2 本の赤い矢印の和となる。1 つは自身の出力から $\mathbf{W}_{\text{hq}}^\top$ を経る経路であり、もう 1 つは次の状態から $\mathbf{W}_{\text{hh}}^\top$ を経る経路である。

$\mathbf{G}_t = \partial L/\partial \mathbf{H}_t$ を、$\mathbf{H}_t$ に関する総損失の勾配とする。$\mathbf{H}_t$ はその時点の出力 $\mathbf{O}_t$ と次の隠れ状態 $\mathbf{H}_{t+1}$ の両方に入力され、さらに右からの行列積を通じた逆伝播では転置した重み行列が右から乗じられるため（4.3 節）、$\mathbf{G}_t$ は次の逆方向の漸化式に従う。

$$\begin{aligned}\mathbf{G}_t &= \frac{\partial L}{\partial \mathbf{O}_t}\mathbf{W}_{\text{hq}}^\top + \mathbf{G}_{t+1}\mathbf{W}_{\text{hh}}^\top,\\ \mathbf{G}_{T+1} &= \mathbf{0},\end{aligned} \tag{8.21}$$

これは Figure 8.10 において各隠れ状態に入ってくる 2 本の赤い矢印に対応する。Equation 8.21 を $\mathbf{G}_T$ から $\mathbf{G}_t$ まで逆方向に展開すると、次式が得られる。

$$\mathbf{G}_t = \sum_{j=t}^{T} \frac{\partial L}{\partial \mathbf{O}_j}\mathbf{W}_{\text{hq}}^\top \left(\mathbf{W}_{\text{hh}}^\top\right)^{j-t}. \tag{8.22}$$

したがって、未来のステップ j における損失は、$k = j - t$ ステップ前の状態へ、k^{th} 乗された $\mathbf{W}_{\text{hh}}^\top$ を通じて逆方向に伝わる（一般の非線形遷移 $\mathbf{H}_t = g(\mathbf{H}_{t-1}, \mathbf{X}_t, \mathbf{W}_{\text{hh}})$ の場合、この行列のべき乗は $j - t$ 個の中間状態ヤコビ行列の積 $\prod_{i=t+1}^{j} \partial \mathbf{H}_i/\partial \mathbf{H}_{i-1}$ に置き換わる）。1 回の逆方向ス

イープで $\mathbf{G}_1, \ldots, \mathbf{G}_T$ を計算してしまえば、パラメータ勾配はステップごとに 1 つの外積（outer product）を足し合わせるだけで求まる。

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_{\text{hh}}} &= \sum_{t=1}^{T} \mathbf{H}_{t-1}^\top \mathbf{G}_t,\\ \frac{\partial L}{\partial \mathbf{W}_{\text{xh}}} &= \sum_{t=1}^{T} \mathbf{X}_t^\top \mathbf{G}_t.\end{aligned} \tag{8.23}$$

1.3.5 節および 4.4 節で見たように、行列を k^{th} 乗すると、各固有ベクトルは λ^k 倍にスケールされる。ρ を最大の $|\lambda|$、すなわち $\mathbf{W}_{\text{hh}}$ の**スペクトル半径**（spectral radius）とすると、$(\mathbf{W}_{\text{hh}}^\top)^k$ のノルムは漸近的に ρ^k のオーダーでスケールする。$\rho < 1$ のとき遠いステップからの寄与は指数関数的に消失し、$\rho > 1$ のときは指数関数的に爆発する。ここで tanh 活性化関数を復活させると、各ステップの逆伝播勾配には、局所的な導関数 $1 - \mathbf{H}_t^2 \in (0, 1]$ が要素ごとに乗じられてから $\mathbf{W}_{\text{hh}}^\top$ が掛けられる。$1 - h^2 \leq 1$ であるため、tanh が勾配を増幅させることは決してなく、隠れユニットが ± 1 付近で飽和すると勾配消失へと強く引き寄せる。ただし、$\mathbf{W}_{\text{hh}}$ が大きく、かつ活性化値が 0 付近にとどまる場合には、依然として勾配爆発が起こりうる。

8.4.2 節で訓練した言語モデルの内部で実際に何が起きているかを測定してみよう。まず、32 トークンのウィンドウで `TimeMachine` データセットを読み込む。

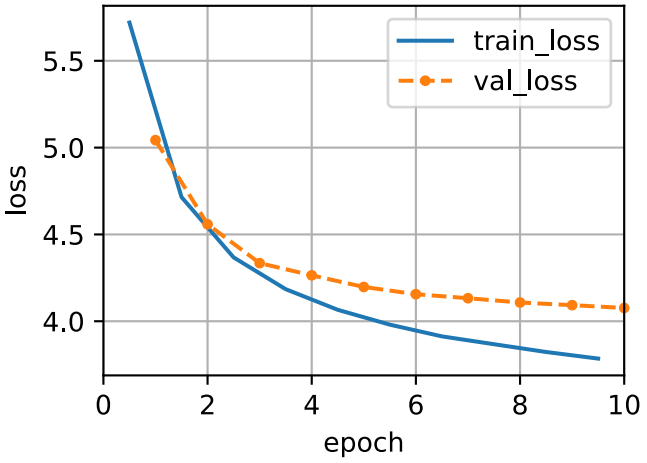
```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

次に、ゼロから実装したリカレント言語モデル `RNNLMScratch` を、埋め込み次元（embedding dimension）を $d = 512$、隠れ状態のサイズを $h = 128$ とし、ドロップアウトを加えてインスタンス化する。

```
rnn = d2l.RNNScratch(num_inputs=512, num_hiddens=128)
model = d2l.RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                         emb_dropout=0.1)
```

`model` を 10 エポックにわたり、学習率 `lr=4`、大域的勾配クリッピング `gradient_clip_val=1` の設定で訓練する。

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



より長いホライズンにわたる勾配フロー（gradient flow）を調べるため、検証データからランダムに 256 個のセグメント `seqs`（長さ $L = 64$、訓練ウィンドウの 2 倍）と、それぞれの直後のトークン `targets` をサンプリングする。

```
L = 64
val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val, 0])
starts = np.random.default_rng().choice(len(val) - L, 256, replace=False)
seqs = np.stack([val[s:s + L] for s in starts])  # (256, L) token ids
targets = val[starts + L]                        # The token after each stretch
```

ここで `model.rnn` を `seqs` にわたって順伝播させ、全 64 ステップ分の隠れ状態を `H` に収集するとともに、最終ステップの `logits`、および重み行列 `W_hh` と `W_hq` を取得する。

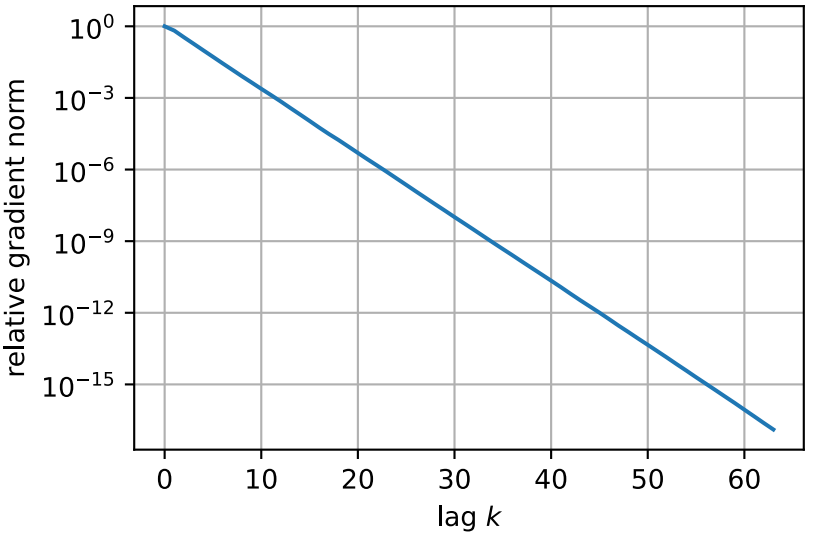
```
with torch.inference_mode():
    states, _ = model.rnn(model.embedding(
        d2l.tensor(seqs, device=d2l.try_gpu())))
    logits = model.output_layer(states[-1:])[:, 0]  # The last step only
H, logits = d2l.numpy(d2l.stack(states, 0)), d2l.numpy(logits)
W_hh, W_hq = d2l.numpy(model.rnn.W_hh), d2l.numpy(model.W_hq)
```

最終ステップの損失勾配 $\partial L_L/\partial \mathbf{O}_L = \mathrm{softmax}(\mathbf{O}_L) - \mathbf{Y}_L$ から出発し、$\mathbf{G}_{t-1} = (\mathbf{G}_t \odot (1 - \mathbf{H}_t^2))\mathbf{W}_{\mathrm{hh}}^\top$ によって時間を逆方向にたどり、相対勾配ノルム $\|\mathbf{G}_{L-k}\|/\|\mathbf{G}_L\|$ の幾何平均をラグ k の関数としてプロットする。

```
P = np.exp(logits - logits.max(1, keepdims=True))
P /= P.sum(1, keepdims=True)
P[np.arange(len(targets)), targets] -= 1  # dL/dO: softmax minus one-hot
G = (P @ W_hq.T).astype(np.float64)       # dL/dH at the last step
norms = []
for t in range(L - 1, -1, -1):            # Backward through time
    norms.append(np.linalg.norm(G, axis=1))
    G = (G * (1 - H[t] ** 2)) @ W_hh.T    # Through the tanh, then W_hh
rel = np.exp(np.log(np.array(norms) / norms[0]).mean(1))
```

```
print(f'spectral radius of W_hh: {np.abs(np.linalg.eigvals(W_hh)).max():.2f}; '
      f'mean tanh derivative: {(1 - H ** 2).mean():.2f}')
print('relative gradient norm at lags 1, 2, 4, 8, 16, 32:',
      [float(f'{rel[k]:.2g}') for k in (1, 2, 4, 8, 16, 32)])
d2l.plot(np.arange(L), rel, 'lag $k$', 'relative gradient norm',
         yscale='log', figsize=(4.5, 3))
```

```
spectral radius of W_hh: 1.82; mean tanh derivative: 0.27
relative gradient norm at lags 1, 2, 4, 8, 16, 32: [0.66, 0.35, 0.099, 0.0081, 5.7e-05, 3e-09]
```



訓練後の $\mathbf{W}_{\text{hh}}$ のスペクトル半径は 1.8 付近であり、純粋な線形漸化式であれば爆発する値にもかかわらず、実際の勾配は急速に消失する。およそ 8 ステップ以内に初期ノルムの 1% 未満へと落ち込み、32 トークンにわたると 10 億分の数程度にまで縮小するのである。逆方向の 1 ステップごとに保たれる勾配ノルムは半分強であり（ラグ 8 での出力値から $0.008^{1/8} \approx 0.55$ となる）、大まかな見積もり $1.8 \times 0.27 \approx 0.5$（スペクトル半径に平均の tanh 導関数を掛けた値）に近い。この減衰は、位置ごとの検証損失（validation loss）が 2〜3 トークンのコンテキストで頭打ちになるという 8.4.2 節の結果と整合するが、勾配消失がその原因であることの証明にはならない。ラグ 2 と 4 では勾配はまだ元の約 3 分の 1 および約 10 分の 1 の大きさを保っており、この程度の減衰であれば SGD は問題なく学習できるうえ、8.4.2 節の Kneser–Ney カウントも勾配を一切使わずに同じコンテキスト長（context length）で飽和しているからである。ただし、データにどのような情報が含まれていようとも、はるか前のトークンからの学習がこの減衰によって阻まれることは確かである。

8.5.1.1 勾配爆発と勾配消失への対策

勾配爆発と勾配消失には、根本的に異なる対策が求められる。勾配爆発はたった 1 回の過大な更新でパラメータを破壊しかねないため、大域的勾配クリッピング（8.4 節）が簡便な安全策となる。しかし、勾配の和 Equation 8.22 全体にスカラーを掛けても、遠いラグと近いラグの**相対的な**重みは決して変わらない。したがって勾配消失を克服するには、漸化式そのものを変更して記憶方向に沿った状態ヤコビ行列を単位行列付近に保つ必要がある。その方法には、学習されたゲートで制御される加法的な記憶経路を作る**ゲート付き**リカレントセル（LSTM や GRU など）を用いるアプローチと、状態の非線形性を取り除いて固有値を単位円付近に制約する**構造化線形漸化**

式（structured linear recurrences）および状態空間モデル（state space models）を用いるアプローチの 2 系統があり、どちらも第 12 章で詳しく扱う。

8.5.2 打ち切り型通時的誤差逆伝播法

T トークンからなるコーパス全体にわたって Equation 8.21 を評価するには、順伝播における全 T 個の隠れ状態をメモリに保持し、1 回のパラメータ更新のために全 T ステップを走査する必要がある。**打ち切り型通時的誤差逆伝播法**は、長いストリームを長さ T_0 の固定長セグメントへとスライスし、各セグメントの境界で誤差逆伝播法を打ち切ることで、1 回の更新あたりのメモリ量と計算量の両方に上限を設ける（Figure 8.11）。

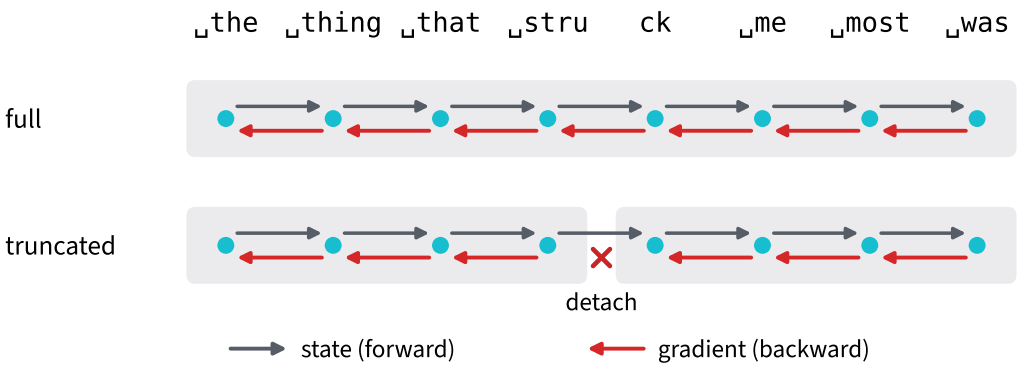


Figure 8.11: トークン系列に対する完全な通時的誤差逆伝播法と打ち切り型通時的誤差逆伝播法。どちらも同じ順伝播（灰色）を実行する。完全な誤差逆伝播法は系列全体に沿って勾配（赤色）を逆方向へ送る。打ち切りを行う場合は系列を 4 トークンのセグメントに分割し、各境界で状態を計算グラフから切り離す。その値は次のセグメントへと引き継がれるが、勾配が境界を越えて逆方向に流れることはない。

各ウィンドウで $\mathbf{H}_0 = \mathbf{0}$ にリセットする場合（8.4.2 節）でも、前のセグメントの最終状態を計算グラフから**切り離し**（detach）ながら順方向に引き継ぐ場合（Figure 8.11）でも、勾配がセグメントの境界を越えることは決してなく、高々 T_0 ステップという厳格な学習ホライズンが課される (Jaeger 2002)。しかし素の RNN では、32 トークンのウィンドウのずっと手前であるおよそ 8 ステップ後には勾配消失によって逆方向の信号がすでに途絶えてしまうため、ウィンドウを長くするだけではモデルの有効な記憶を延ばすことはできない。

8.5.3 まとめ

通時的誤差逆伝播法は、展開された RNN に通常の誤差逆伝播法を適用し、Equation 8.21 によって損失の感度を逆方向に伝播させるとともに、タイムステップ全体でパラメータ勾配を足し合わせる（Equation 8.23）。k ステップ逆方向に進む勾配には k 個の状態ヤコビ行列が乗じられ、線形 RNN ではこれが $(\mathbf{W}_{\text{hh}}^{\top})^k \sim \rho^k$ に帰着するため、勾配は $\rho > 1$ のとき爆発し、$\rho < 1$ のとき消失する。tanh を含めると各ステップで $1 - \mathbf{H}_t^2 \in (0, 1]$ が乗じられるため、訓練した RNN では $\rho \approx 1.8$ であるにもかかわらず、勾配はおよそ 8 ステップ以内に 1% 未満まで低下した。勾配クリッピングは急激な勾配爆発を抑え込み、打ち切り型 BPTT は 1 回の更新あたりのメモリ量と計算量を制限するが、長いホライズンにわたって勾配を保持するにはアーキテクチャ上の革新（第 12 章）が不可欠である。

8.5.4 演習問題

1. **手計算による逆伝播。** `RNNScratch`（8.4.2 節）の 1 ステップは次式を計算する。

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}).$$

   $\mathbf{G}_t = \partial L / \partial \mathbf{H}_t$ を、後続のステップおよびステップ t の出力から到達する勾配とする。

   1. このステップから $\partial L / \partial \mathbf{W}_{\text{xh}}$、$\partial L / \partial \mathbf{W}_{\text{hh}}$、$\partial L / \partial \mathbf{b}_{\text{h}}$、および $\partial L / \partial \mathbf{H}_{t-1}$ への寄与を、$\mathbf{G}_t$、$\mathbf{X}_t$、$\mathbf{H}_t$、$\mathbf{H}_{t-1}$、$\mathbf{W}_{\text{hh}}$ の関数として導出せよ。
   2. ステップ $t-1$ における出力からの寄与を $\partial L / \partial \mathbf{H}_{t-1}$ への寄与に加えると、$\mathbf{G}_{t-1}$ の漸化式が得られる。tanh とバイアスを取り除いたときにこれが Equation 8.21 に帰着することを示し、さらに tanh が各ステップで付け加える因子を答えよ。また、ウィンドウの T ステップにわたってパラメータ勾配がどのように累積するかを説明し、Equation 8.23 と比較せよ。

   CMU 11-785 の Homework 3 Part 1 を改変。

2. **べき乗法とアライメント。** $\mathbf{M} \in \mathbb{R}^{h \times h}$ を対称行列とし、その固有値 λ_i と固有ベクトル $\mathbf{v}_i$（$i = 1, \ldots, h$）が $|\lambda_i| \geq |\lambda_{i+1}|$ を満たすように並べられているとする。

   1. $\mathbf{M}^k$ が同じ固有ベクトルに対して固有値 λ_i^k を持つことを示せ。
   2. 標準正規分布に従う独立な成分を持つ確率ベクトル $\mathbf{x} \in \mathbb{R}^h$ に対し、$\mathbf{M}^k \mathbf{x}$ は $\mathbf{v}_1$ の方向へ k の増加とともに揃っていく（アライメントする）ことを示せ。これに必要な固有値の条件と、どのような意味でアライメントが成り立つかを述べよ。
   3. この結果は、RNN の勾配 Equation 8.22 に対して、ノルムの変化以外に何を意味するか。

3. [code] **コードによる逆伝播。**●`d2l.RNNLMScratch` を `d2l.TimeMachine` 上で 8.4.2 節と同様に訓練せよ。演習問題 1 で求めた 1 ステップの勾配と漸化式を用い、ロジットに関する損失の勾配（3.1.2.2 節）から出発して、`RNNScratch` セルのウィンドウ全体にわたる逆伝播をテンソル演算のみで実装せよ。それを 1 つの訓練ミニバッチ上で実行し、得られた `W_xh`、`W_hh`、`b_h` の勾配を自動微分によって計算された勾配と比較せよ。

   CMU 11-785 の Homework 3 Part 1 を改変。

4. [code] **非線形ヤコビ行列。**●ランダムな 100×100 の対称行列を生成してスペクトル半径 $\rho \in \{0.9, 1.0, 1.1\}$ にリスケールし、その k^{th} 乗のノルム $\|\mathbf{J}^k\|$ を k に対して片対数軸でプロットせよ。固有値から予測されるとおり、3 本の直線が得られるはずである。ここから対称かつ線形な場合を離れる。

   1. 対称行列を、各スペクトル半径にリスケールした一般の非対称行列 $\mathbf{W}_{\text{hh}}$ に置き換え、$\|\mathbf{J}^k\|$ を k に対してプロットせよ。$\|\mathbf{J}^k\|$ の $\rho = 0.9$ における最大値と、それが現れるラグを報告せよ。また、比 $\|\mathbf{J}^k\| / \rho^k$ がなぜ 1 を超えうるのか、そして k が大きくなっても有界にとどまるのはなぜか説明せよ（ランダム行列は確率 1 で対角化可能である）。
   2. 対称な $\mathbf{W}_{\text{hh}}$ に戻り、次の tanh RNN を実行せよ。

$$\mathbf{H}_t = \tanh(\mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{X}_t), \qquad \mathbf{H}_0 = \mathbf{0}, \tag{8.24}$$

      ただし 1 つの系列（$\mathbf{H}_t \in \mathbb{R}^{1 \times h}$）とし、入力 $\mathbf{X}_t$ の各成分は $\mathcal{N}(0, s^2)$ から独立に抽出するものとする。軌跡に沿ったヤコビ行列 $\partial \mathbf{H}_j / \partial \mathbf{H}_{j-1}$ の積のノルムを、$s = 0.1$ と $s = 1$ についてプロットせよ。非線形性と入力スケールによってノルムの増大はどのように変化するか。また、それはなぜか。
   3. $\mathbf{W}_{\text{hh}}$ の最大特異値が 1 未満であることは積が縮小するための十分条件であるのに対し、最大特異値が 1 より大きいことは積が増大するための必要条件ではあっても十分

条件ではない理由を説明せよ (Bengio et al. 1994; Pascanu et al. 2013)。この議論を、得られた tanh の曲線（$\rho = 0.9$ および $\rho = 1.1$）と関連づけよ。

5. [code] **ウィンドウ長と記憶ホライズン**。●8.5.1 節のモデルでは、最後の損失の勾配はおよそ 8 ステップ以内に元の大きさの 1% 未満まで低下し、これは 32 トークンの訓練ウィンドウの十分内側に収まっている。
   1. コードを実行する前に考えよ。このモデルを `num_steps=128` で再訓練した場合、このホライズンは移動すると予測されるか。Equation 8.22 における状態ヤコビ行列の積に基づいて論じよ。
   2. `num_steps` を 128 に設定して再訓練し、検証データの 256 トークンの区間上でラグに対する勾配を測定して、結果を自身の予測と比較せよ。
   3. 再訓練したモデルに対して、8.4.2 節の位置ごとの損失の測定を再現せよ。コンテキストの 3 番目のトークンを超えても損失は改善し続けるようになるか。

8.6 デコーディングと生成

本章ではここまで、次のトークンにどれだけ適切に確率を割り当てられるかという基準だけで言語モデルを評価してきた。しかし、テキストを**生成**するには、予測された分布から各ステップで具体的なトークンを 1 つ選択する必要もある。この選択は予想以上に重要である。というのも、選択規則が異なれば、まったく同じモデルからでも劇的に異なる文章が生成されるからである。8.4.2 節では、最も単純な規則がうまく機能しないことを見た。各ステップで最も確率の高い単一のトークンを貪欲に選び続けると、生成器はたちまち反復ループに陥ってしまう。では、次トークンの分布をどのようにすれば流暢なテキストへと変換できるのだろうか。

次トークン分布の系列を出力トークンへと変換するアルゴリズムは**デコーディング**と呼ばれる (Graves 2013; Holtzman et al. 2020)。デコーディング戦略は大きく 2 つの系統に分けられる。**最大化**（maximization）アルゴリズムはモデルの下で高い同時確率を持つ出力系列を探索するのに対し、**サンプリング**アルゴリズムは分布の信頼できない裾（tail）を変形または刈り込んだうえで各トークンを確率的に抽出する。以下では、この両系統を順に構築し比較していく。重要な点として、どのデコーディング規則も次トークン分布のみを介して言語モデルとやり取りするため、ここで構築するアルゴリズムは RNN、状態空間モデル、Transformer のいずれであっても、任意の自己回帰言語モデルにそのまま適用できる。

8.6.1 デコーディング問題

訓練済みの言語モデルは、次トークンの条件付き分布を与える。任意のプレフィックス $x_{<t} = x_1, \dots, x_{t-1}$ が与えられると $P(x_t \mid x_{<t})$ を出力し、確率の連鎖律（8.1 節）により、これらの条件付き確率から任意の完全な系列の同時確率が $P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$ と定まる。

デコーディングはその逆問題を突きつける。すなわち、これらの条件付き分布が与えられたとき、どの系列を出力すべきかという問題である。一見すると、**最も確率の高い**系列を見つけることが自明な目標に思えるが、厳密な最大化は計算量的にまったく実行不可能である。語彙 $\mathcal{V}$ とホライズン T に対して、可能な続きは $|\mathcal{V}|^T$ 通り存在する。本節で用いる 1,027 トークンの語彙と 50 トークンの続きでは $1027^{50} > 10^{150}$ 通りの候補となり、到底列挙できる数ではない。したがって、実用的なデコーダはいずれも探索木のごく小さなスライスのみを探索し、各ステップでどのプレフィックスを残すかを決定しなければならない。

仮に厳密な探索が可能であったとしても、確率が最大の単一の系列は、必ずしも求めているものとは**限らない**ことが多い。音声の文字起こしや技術文の翻訳のように、正解となる出力の幅が狭いタスクでは、高確率の系列を求めるアプローチがうまく機能する。対照的に、対話や創作執筆

のように多くの続きが同等に妥当である場合、よく較正されたモデルは多数の生じうる軌跡に確率を分散させるため、個々の続きが持つ確率質量はどうしても微小になる。モデルに最も確率の高い単一の経路を強制的に選ばせると、その自然な多様性が損なわれてしまうが、サンプリングは学習された分布から直接抽出することで多様性を保つ。

Figure 8.12 は、この対比に沿ってデコーディング戦略を整理したものである。最大化手法である貪欲デコーディング（greedy decoding）とビームサーチは、妥当な出力が限られる制約付きタスクにおける標準的な選択肢であり続ける一方、温度スケーリング（temperature scaling）や打ち切り（truncation）を組み合わせたサンプリング手法は、オープンエンドな生成におけるデフォルトとなっている。

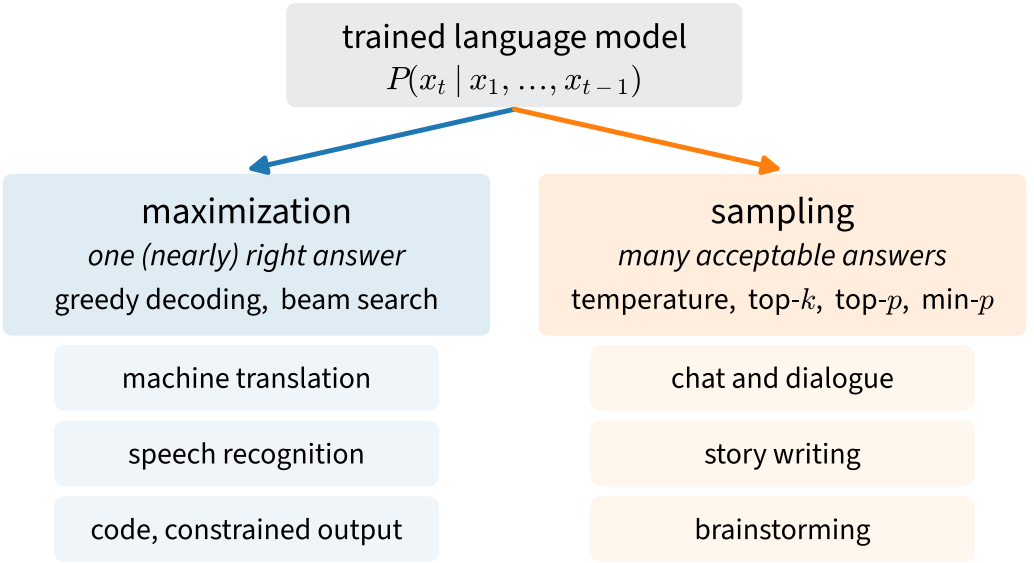


Figure 8.12: デコーディング規則の 2 つの系統。高確率の系列を探す手法は許容される出力が少ないタスクに適しており、サンプリングはオープンエンドな生成において多様性を保つ。

8.6.1.1 デコード対象のモデル

実験には、8.4.6 節で構築した簡潔な実装の RNN 言語モデルを用いる。このモデルは 512 次元の埋め込み、128 個の隠れユニット、埋め込みと隠れ状態の両方に対するドロップアウト、および『The Time Machine』の 1,027 個の BPE トークンに対する出力層を備えている。まず、`d2l.TimeMachine` から長さ 32 トークンの訓練ウィンドウと検証ウィンドウを読み込む。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

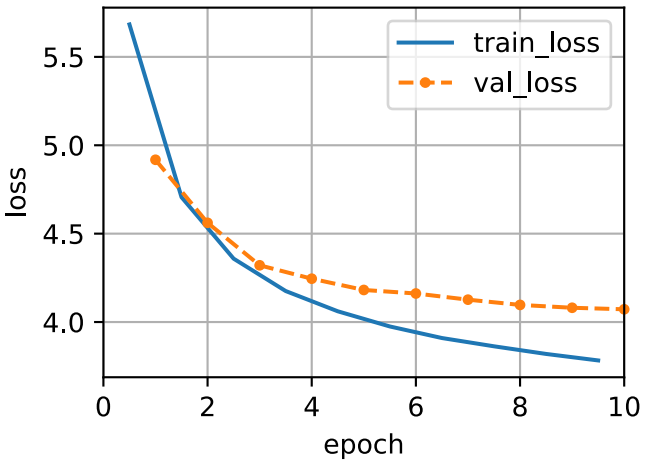
次に、簡潔な実装の再帰セル `d2l.RNN` と言語モデル `d2l.RNNLM` を、$d = 512$、$h = 128$、ドロップアウト、学習率 `lr=4` の設定で構築する。

```
rnn = d2l.RNN(num_inputs=512, num_hiddens=128)
model = d2l.RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
```

```
                    emb_dropout=0.1)
```

model を gradient_clip_val=1 の勾配クリッピング付きで 10 エポック訓練すると、『The Time Machine』における検証パープレキシティは約 60 となる。

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
print(f'validation perplexity {math.exp(trainer.history["val_loss"][-1]):.1f}')
```




```
validation perplexity 58.7
```

本節のすべてのデコーディングアルゴリズムは、モデルに依存しない単一のインターフェース step_fn(ids) を介して言語モデルにアクセスする。この関数は、それまでのプレフィックスのトークン ID を受け取り、次トークンのロジットからなる 1 次元 NumPy ベクトルを返す。通常の NumPy 配列を返すことで、4 つのフレームワークすべてに対して各デコーダを一度書くだけで済む。また、明示的な隠れ状態を省いているのは意図的な設計であり、リカレント型か Transformer かを問わず、あらゆる自己回帰モデルがこのシグネチャにそのまま適合する。

```
model.eval()  # Dropout off

def seq_logits(ids):  # Token ids in, NumPy logits at every position out
    with torch.inference_mode():
        logits = model(d2l.tensor([ids], device=d2l.try_gpu()))
    return d2l.numpy(logits)[0]

def step_fn(ids):  # Token ids in, NumPy logits for the next token out
    return seq_logits(ids)[-1]
```

そのトレードオフは効率性にある。1 回の呼び出しごとに、ロジットのベクトルを 1 つ取り出すためだけにプレフィックス**全体**にわたってネットワークを再評価してしまうのである。RNN ではこの余分な計算は回避可能であり、その計算コストについては本節の最後で改めて取り上げる。

8.6.2 貪欲デコーディング

考えられる限り最も単純なデコーダは、各ステップで条件付き確率が最も高い単一のトークンを確定させるものである。

$$x_t = \operatorname*{argmax}_{x \in \mathcal{V}} P(x \mid x_{<t}). \tag{8.25}$$

プレフィックスのスコアを計算し、argmax を選び、末尾に追加して繰り返す。貪欲デコーディングは生成トークンあたり 1 回のモデル評価しか必要とせず、完全に決定論的である。すなわち、同じプレフィックスからは常にまったく同じ続きが得られる。以下の補助関数 show は、トークン ID をデコードして引用符付きの文字列に変換し、改行を \n として表示するとともに、印字不可能なバイトを U+FFFD に置き換える。

```
def show(ids):  # Decoded text as a literal; control characters -> U+FFFD
    text = data.tokenizer.decode(ids)
    return repr(''.join(c if c.isprintable() or c == '\n' else '\ufffd'
                        for c in text))

def decode_greedy(text, num_tokens=50):
    ids = data.tokenizer.encode(text)
    for _ in range(num_tokens):
        ids.append(int(step_fn(ids).argmax()))
    return ids

for text in ('The Time Traveller', 'It seemed to me that'):
    d2l.print_wrapped(show(decode_greedy(text)))
```

```
"The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I had\nf"
"It seemed to me that\ntheir.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I"
```

8.4.2 節と同様に、どちらの続きもすぐに反復ループへと陥ってしまう。貪欲デコーディングには 2 つの構造的な弱点が内在している。1 つ目は**近視眼性**（myopia）である。局所的に最適な選択をつなぎ合わせても、大域的に最適な系列が得られるとは限らない。たとえば、モデルが最初のトークンに確率 $P(A) = 0.5$ と $P(B) = 0.4$ を割り当て、続く 2 ステップ目では A の後は不確実（最上位の続きの確率が 0.3）である一方、B の後は非常に確信度が高い（最上位の確率が 0.9）としよう。貪欲デコーディングは即座に A を確定させ、同時確率 $0.5 \times 0.3 = 0.15$ で終了してしまい、B から始まるはるかに確率の高い系列（同時確率 $0.4 \times 0.9 = 0.36$）を見逃してしまう（Figure 8.13）。

2 つ目の弱点は反復そのものである。貪欲デコーディングは各ステップで自身の決定論的な argmax を再帰構造にそのまま入力するため、次の隠れ状態は現在の状態の決定論的な関数となる。そのため、軌跡がいったん引き込みサイクル（attracting cycle）に落ち込むと、そこから抜け出させるランダム性が存在しない。しかも、これは本節の小さな RNN 特有の癖ではない。Holtzman et al. (2020) は、1,000 倍規模の大きい言語モデルにおいてもまったく同じ**ニューラルテキスト退化**（neural text degeneration）が生じることを実証し、オープンエンドなテキストに対する尤度最大化が一貫して反復的で味気ない、人間の文章とは明らかに異なる散文を生み出すことを示した。

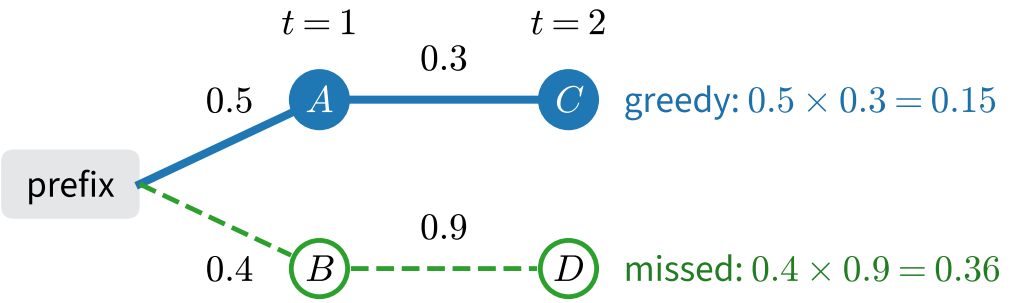


Figure 8.13: 2 ステップの例における貪欲デコーディングの近視眼性。貪欲デコーディングはより確率の高い最初のトークン A を選び、系列 AC（同時確率 $0.5 \times 0.3 = 0.15$）で終了する。一度も探索されない B を経由する経路は $0.4 \times 0.9 = 0.36$ に達する。

8.6.3 ビームサーチ

貪欲デコーディングが直ちに単一の経路を確定させ、全探索（exhaustive search）が可能なすべての経路を保持するのに対し、**ビームサーチ**は単一のハイパーパラメータである**ビーム幅**（beam width）w によって制御される中間的な手法をとる。ビームサーチは各ステップで、最もスコアの高い w 個のプレフィックスを残す。最も確率の高い w 個の最初のトークンから出発し、続く各ステップでは残っているすべてのプレフィックスを語彙全体にわたって展開し、$w \cdot |\mathcal{V}|$ 個の拡張候補すべてを累積対数確率でスコア付けしたうえで、上位 w 個の系列へと刈り込む。$w = 1$ とすれば貪欲デコーディングに一致し、候補が一切刈り込まれないほど w を大きくすれば全探索に一致する。

Figure 8.14 は、$w = 2$ とし、5 トークンの語彙 $\mathcal{V} = \{A, B, C, D, E\}$ 上で 3 ステップにわたって行うビームサーチを示している（共通のプロンプトは表記から省略している）。A と C が最も確率の高い上位 2 つの最初のトークンであるとしよう。ステップ 2 では、10 通りすべての拡張 $x' \in \mathcal{V}$ のスコアを次の式で計算する。

$$
\begin{aligned}
P(A, x') &= P(A)\, P(x' \mid A), \\
P(C, x') &= P(C)\, P(x' \mid C),
\end{aligned}
$$

そして上位 2 つ（たとえば AB と CE）を保持し、ステップ 3 でも同じ展開を繰り返して ABD と CED を残す。ビームサーチは $5^3 = 125$ 通りの系列すべてをスコア付けするのではなく、$5 + 10 + 10 = 25$ 通りのみを評価する。一般に、T トークンの生成にかかるコストは $O(w|\mathcal{V}|T)$ であり、貪欲デコーディングのわずか w 倍のコストで済み、全探索より指数関数的に少ない。

系列終了トークン（end-of-sequence token）を出力して系列を早期終了できる場合、異なる長さの候補を比較するには長さの補正が必要になる。各トークンの対数確率は非正であるため、生の累積対数確率は系列が長くなるにつれて単調に減少し、短い仮説が不当に有利になってしまう。標準的な対策は**長さ正規化**（length normalization）であり (Wu et al. 2016)、T トークンを長さ t_0 のプレフィックスに追加した候補を次の式でスコア付けする。

$$
\frac{1}{T^{\beta}} \sum_{t=t_0+1}^{t_0+T} \log P(x_t \mid x_{<t}), \tag{8.26}
$$

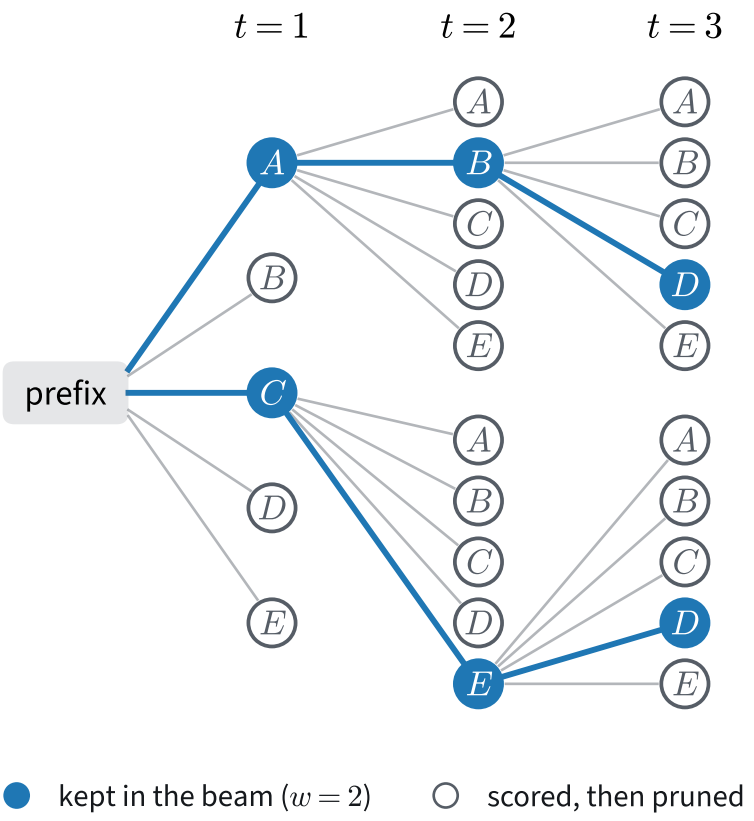


Figure 8.14: 5 トークンの語彙に対するビーム幅 2 のビームサーチ。各ステップで残っている各プレフィックスを語彙全体にわたって展開し、スコアが上位 2 つのプレフィックスのみを保持する。ここで残るのはまず A と C、次に AB と CE、続いて ABD と CED である。

ここで、指数 β（通常は 0.75 前後）は、正規化なしの対数確率（$\beta = 0$）とトークンあたりの平均対数確率（$\beta = 1$）の間を補間する。以下のコードでは、`alpha` という引数名を Wu et al. (2016) に倣って用いる。

以下の `beam_search` の実装では、完了した仮説（`eos_id` で終わるもの）をそのまま次のステップへ引き継ぎ、アクティブなビームのみを展開する。なお、ここでは 1 つ簡潔な最適化を施している。全体の上位 w 個の候補には、任意の単一プレフィックスからの拡張は高々 w 個しか含まれないため、各アクティブビームからは上位 w 個の次トークンだけを取り出せばよく、$|\mathcal{V}|$ 個の拡張すべてをソートする必要はない。

```
def beam_search(step_fn, prefix, num_tokens, beam_size=4,
                alpha=0.75, eos_id=None):
    """Length-normalized beam search (score = log P / len^alpha).
    Returns candidates sorted best-first; [0][1] is the argmax sequence."""
    def log_softmax(logits):
        logits = [float(l) for l in logits]
        m = max(logits)
        lse = m + math.log(sum(math.exp(l - m) for l in logits))
        return [l - lse for l in logits]
    score = lambda logp, ids: logp / max(1, len(ids) - len(prefix))**alpha
    beams = [(0.0, list(prefix), False)]  # (log-prob, sequence, finished)
    for _ in range(num_tokens):
        if all(done for _, _, done in beams):
            break
        candidates = [b for b in beams if b[2]]  # Finished pass through
        for logp, ids, _ in (b for b in beams if not b[2]):
            logprobs = log_softmax(step_fn(ids))
            best = sorted(range(len(logprobs)),
                          key=lambda i: -logprobs[i])[:beam_size]
            candidates += [(logp + logprobs[i], ids + [i], i == eos_id)
                           for i in best]
        beams = sorted(candidates, key=lambda b: score(b[0], b[1]),
                       reverse=True)[:beam_size]
    return sorted([(score(logp, ids), ids) for logp, ids, _ in beams],
                  reverse=True)
```

同じ 2 つのプロンプトを $w = 1$（貪欲デコーディングの再現）と $w = 4$ でデコードし、最上位仮説の長さ正規化スコア Equation 8.26 を出力してみよう。このコーパスには系列終了トークンがないため、すべての候補が長さ 30 に達し、定数の分母 $30^{0.75} \approx 12.8$ によって相対的なランキングは変化しない。

```
for text in ('The Time Traveller', 'It seemed to me that'):
    prefix = data.tokenizer.encode(text)
    for w in (1, 4):
        score, ids = beam_search(step_fn, prefix, 30, beam_size=w)[0]
        d2l.print_wrapped(f'w={w}, score {score:.2f}: {show(ids)}')
```

```
w=1, score -4.13: "The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I
    had\nfrom the\nsidea.\n"
w=4, score -3.43: "The Time Traveller.\n\n'Then I had\ndeliculture.\n\n'Then
    I had\ndeliculture.\n"
w=1, score -4.21: "It seemed to me that\ntheir.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nside"
```

```
w=4, score -3.60: "It seemed to me that it was a\npossibly.\n\n'As I saw
    a\npossibly.\n\n'As"
```

ビームを広げると、一般には貪欲デコーディングよりも確率の高い続きが見つかる。ただし、近似的な刈り込みによって後から有望になるはずだったプレフィックスが捨てられることもあるため、確実とは限らない。しかも、確率が高いからといって自動的に**より良い**文章になるとは限らない。オープンエンドな生成では、大規模言語モデル（large language model、LLM）におけるビームサーチは人間の文章よりも顕著に反復の多いテキストを生成する (Holtzman et al. 2020)。この対比については後ほど定量化する。対照的に、音声認識や機械翻訳のような制約の強いタスクでは、適度な幅（通常 4〜10）のビームサーチが依然として標準的なデコーダである (Sutskever et al. 2014; Wu et al. 2016)。

8.6.4 サンプリング

モデルが続きに関する妥当な分布を学習できているなら、最大化に代わる最も自然な選択肢は直接的な**祖先サンプリング**（ancestral sampling）である。すなわち、$x_t \sim P(x_t \mid x_{<t})$ を抽出し、末尾に追加して繰り返す。サンプリングは最大化が抑え込んでしまう豊かな多様性を取り戻すが、生のサンプリングはそのまま分布の裾の問題に直面する。ソフトマックスは語彙のすべての項目（ここでは約 1,000 トークン、現代の LLM では 10 万トークン以上）に非ゼロの確率を割り当てるため、何千もの不自然なトークンにわたる微小な確率が合計されると、各ステップで無視できない大きさの確率質量になってしまう。その結果、純粋なサンプリングではまったく不自然なトークンが頻繁に抽出され、それをコンテキストとして再入力することで残りの文章全体が脱線しかねない。これはまさに 8.1 節で議論したロールアウト誤差の蓄積そのものである。Holtzman et al. (2020) はこれを**信頼できない裾**（unreliable tail）と呼んだ。言語モデルの確率は分布の上位付近ではよく較正されているが、裾の奥深くではノイズが多いのである。実用的な対策はサンプリングの前に分布を変形または打ち切ることであり、広く使われている 4 つの制御手法がそれぞれ異なる方法でこれを実現する。

温度スケーリングは、ソフトマックスの前にすべてのロジット o_x を正のスカラー $\tau > 0$ で割る手法である。

$$P_\tau(x \mid x_{<t}) = \frac{\exp(o_x/\tau)}{\sum_{x' \in \mathcal{V}} \exp(o_{x'}/\tau)} \propto P(x \mid x_{<t})^{1/\tau}. \tag{8.27}$$

τ を 1 より小さくすると分布は鋭くなり（$\tau \to 0$ で貪欲デコーディングに収束する）、逆に τ を 1 より大きくすると一様分布に近づくよう平坦化される。しかし、温度は分布全体を一様に再スケーリングするため、大雑把な調整手段にすぎない。信頼できない裾を抑え込めるほどソフトマックスを冷やすと、もっともらしい上位トークン間の正当な多様性まで同時に潰れてしまうからである。

Top-k サンプリングは順位によって打ち切りを行い、最も確率の高い上位 k 個のトークンのみを残して再正規化し、サンプリングする (Fan et al. 2018; Radford et al. 2019)。一方、**Top-p**（または**ニュークリアス**）サンプリングは累積確率質量によって打ち切りを行い、確率の和が少なくとも p（一般には $p = 0.9$）に達する最小の上位トークン集合を保持する (Holtzman et al. 2020)。この違いが重要なのは、モデルのエントロピーがトークンごとに大きく変動するためである。次の単語がほぼ確実な場面では、固定された $k = 20$ は 19 個の不適切な候補を通過させてしまう一方、何十もの単語が同等に自然な場面では、$k = 20$ が本来の多様性を切り捨ててしまうおそれがある。Top-p は動的に適応し、鋭い分布では候補集合を縮小し、平坦な分布では拡大する。ただし、単一の最上位トークンだけで p を超える場合（たとえば確率が 0.95 であり、$p = 0.9$ の下で有力な 2 番手の候補を除外してしまう場合）には、依然として過度に貪欲な振る舞いをしうる。

Min-p サンプリングは、モデルの最上位確率に対する相対的な閾値を設定する手法であり (Nguyen et al. 2025)、最も確率の高い単一トークンの少なくとも $p_{\min}$ 倍（たとえば $p_{\min} = 0.05$）の確率を持つすべてのトークンを保持する。モデルの確信度が高いときにはカットオフが自動的に上昇して有力な候補のみを通し、モデルが不確実なときにはカットオフが下がってより幅広い探索を許容する。Nguyen et al. (2025) は、絶対的な質量閾値では平坦化された裾の大部分まで拾い込んでしまうような高い温度においても、この相対閾値であれば top-p よりはるかに穏やかに性能を保てることを示している。

8.6.4.1 統一サンプラ

温度、top-k、top-p、min-p はいずれもまったく同じ次トークン分布に対して作用するため、コンパクトな単一の関数ですべての組合せを扱える。以下の `sample_next` は、`temperature == 0` のときに貪欲な argmax を返し、それ以外のときは Equation 8.27 によって P_τ を構成してトークンを確率の降順にソートし、有効な各カットオフが許容する上位トークン数を計算して最も厳しい上限を採用し、再正規化したうえで NumPy の乱数生成器 `rng` を用いてトークンを 1 つ抽出する。

```
def sample_next(logits, temperature=1.0, k=None, p=None, min_p=None,
                rng=None):
    """Draw a token id from 1-D NumPy logits; temperature 0 is greedy.
    Top-k, top-p, and min-p truncate the temperature-scaled distribution."""
    logits = np.asarray(logits, dtype=np.float64)
    if temperature == 0:
        return int(logits.argmax())
    probs = np.exp((logits - logits.max()) / temperature)
    order = np.argsort(-probs)              # token ids, most probable first
    probs = probs[order] / probs.sum()
    keep = len(probs)
    if k is not None:                       # top-k: a fixed count
        keep = min(keep, k)
    if p is not None:                       # top-p: smallest head of mass p
        keep = min(keep, int(np.searchsorted(probs.cumsum(), p)) + 1)
    if min_p is not None:                   # min-p: a fraction of the top
        keep = min(keep, int((probs >= min_p * probs[0]).sum()))
    rng = np.random.default_rng() if rng is None else rng
    return int(order[rng.choice(keep, p=probs[:keep] / probs[:keep].sum())])
```

ラッパー関数 `generate` は、自己回帰デコーディング（autoregressive decoding）のループをまとめたものである。`step_fn` を呼び出し、`sample_next` で次のトークンを抽出し（戦略に関するキーワード引数はそのまま渡す）、末尾に追加し、`eos_id` が出力されたら早期終了する。『The Time Machine』のオープンエンドな文章には系列終了トークンがないため、本節の例では固定個数のトークンを生成する。

```
def generate(step_fn, prefix, num_tokens, eos_id=None, **strategy):
    """Extend prefix by up to num_tokens ids, each chosen by sample_next
    from step_fn(ids), the next-token logits; stop early at eos_id."""
    ids = list(prefix)
    for _ in range(num_tokens):
        ids.append(sample_next(step_fn(ids), **strategy))
        if eos_id is not None and ids[-1] == eos_id:
            break
    return ids
```

確率が $0.45, 0.25, 0.15, 0.10$ および 0.05 である 5 トークンのトイ分布を使うと、各カットオフの動作を直接確認できる。top-k（$k = 2$）はトークン 0 と 1 のみを保持し、top-p（$p = 0.8$）はトークン 0、1、2（累積質量が 0.85 に達する）を保持し、min-p（$p_{\min} = 0.5$）は少なくとも $0.5 \times 0.45 = 0.225$ の確率を持つトークン、すなわち再びトークン 0 と 1 を保持する。各規則の下で 1,000 個のサンプルを抽出すると、その台（support）が正確に一致することが確認できる。

```
logits = np.log([0.45, 0.25, 0.15, 0.10, 0.05])
print('greedy:', sample_next(logits, temperature=0))
for cutoff in (dict(k=2), dict(p=0.8), dict(min_p=0.5)):
    drawn = {sample_next(logits, **cutoff) for _ in range(1000)}
    print(cutoff, 'tokens drawn:', sorted(drawn))
```

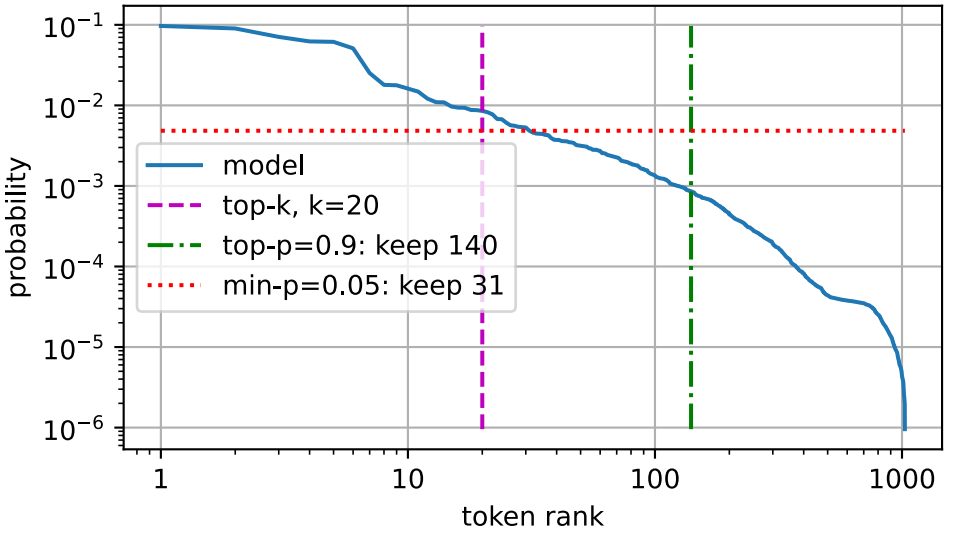
```
greedy: 0
{'k': 2} tokens drawn: [0, 1]
{'p': 0.8} tokens drawn: [0, 1, 2]
{'min_p': 0.5} tokens drawn: [0, 1]
```

予想どおり、どの規則も残った台のみから厳密にサンプリングしている。すなわち、トークン 0 と 1 が top-k と min-p で抽出され、トークン 0〜2 が top-p で抽出されている。

8.6.4.2 訓練済みモデルにおけるカットオフ

これら 3 つの規則が実際の言語モデル上でどのように振る舞うかを確認するため、`"The Time Traveller"` に続く RNN の次トークン分布をソートして両対数軸（log-log axes）上にプロットしてみよう。このように表示すると、各打ち切り規則は明快な幾何学的解釈を持つ。top-k は曲線の形状にかかわらず順位 k に固定の垂直線を引き、top-p は確率質量 p が蓄積される順位に垂直線を引き、min-p は最上位確率の $p_{\min}$ 倍の高さに**水平**線を引き、モデルの確信度に応じて上下する。垂直カットオフより右側、あるいは水平カットオフより下側にあるトークンはすべて刈り込まれる。

```
def next_probs(text):  # Next-token probabilities after text, sorted
    logits = step_fn(data.tokenizer.encode(text))
    probs = np.exp(logits - logits.max())
    return np.sort(probs / probs.sum())[::-1]

def cutoffs(probs, p=0.9, min_p=0.05):  # Tokens kept by top-p and min-p
    return (int(np.searchsorted(probs.cumsum(), p)) + 1,
            int((probs >= min_p * probs[0]).sum()))

probs = next_probs('The Time Traveller')
n_p, n_m = cutoffs(probs)
lo, hi = probs[-1], probs[0]
d2l.plot([np.arange(1, len(probs) + 1), [20, 20], [n_p, n_p],
          [1, len(probs)]], [probs, [lo, hi], [lo, hi], [0.05 * hi] * 2],
         xlabel='token rank', ylabel='probability', xscale='log',
         yscale='log', figsize=(5.5, 3),
         legend=['model', 'top-k, k=20', f'top-p=0.9: keep {n_p}',
                 f'min-p=0.05: keep {n_m}'])
```



両対数軸上では、ソートされた確率は順位 1 における 0.1 から 0.25 の間から順位 100 付近の約 10^{-3} まで着実に低下した後、残りの数百トークンにわたってさらに急激に落ち込んでいる。これが打ち切りによって除去される広大な低確率の裾である。モデルの確信度に対して各規則がどれだけ適応的に反応するかを観察するため、`"The Time Traveller"` をより短いプレフィックス `"The Time"` と比較する。後者は小説中の全 13 回の出現のうち 11 回で直後に `" Traveller"` が続くため、鋭く尖った分布をもたらす。

```
for text in ('The Time Traveller', 'The Time'):
    probs = next_probs(text)
    n_p, n_m = cutoffs(probs)
    print(f'{text!r}: top probability {probs[0]:.2f}; kept: top-k 20, '
          f'top-p {n_p}, min-p {n_m}')
```

```
'The Time Traveller': top probability 0.10; kept: top-k 20, top-p 140, min-p 31
'The Time': top probability 0.69; kept: top-k 20, top-p 6, min-p 2
```

top-k がコンテキストにかかわらず機械的に 20 トークンを保持するのに対し、2 つの適応的な規則はモデルの確信度が高い場面で急激に絞り込まれる。最上位トークンが支配的な `"The Time"` の後ではどちらも積極的に刈り込みを行い、とりわけ min-p はごく少数の候補しか残さない。しかし、`"The Time Traveller"` の後のようにモデルが不確実な場面では、両者の振る舞いは大きく分かれる。top-p は確率質量の 90% を集めるために裾の奥深くまで踏み込む必要があり、その大半が個々の確率 1% 未満である 100 個を優に超えるトークンを通過させてしまう。一方、min-p は最上位候補のみを基準とするため、最上位確率の 20 分の 1 以内に収まる高々数十個のトークンしか保持しない。

8.6.4.3 単一プレフィックス上での戦略の比較

まず $\tau = 0.5$、1.0、2.0 における純粋な温度スケーリングから始めて、各戦略が実際に生成する続きを確認してみよう。

```
prefix = data.tokenizer.encode('The Time Traveller')
for tau in (0.5, 1.0, 2.0):
    out = generate(step_fn, prefix, 50, temperature=tau)
    d2l.print_wrapped(f'tau={tau}: {show(out)}')
```

```
tau=0.5: "The Time Traveller pale of the\ndelight of\nthe lamp and\npresent
    bushes.\n\n'I had\npresently\nmetinessed\nof a time. I felt\nwhich\n"
tau=1.0: "The Time Traveller.'\n\n'From the powersing solack, and silent it
    would by vanished! I came gave me to me.\n\n'I took with me up of
    them?ly dancing ru"
tau=2.0: 'The Time Traveller went co tsprockated d⍰ir can⍰ stoelt-lice slow
    against\nich⍰ined to hished theirsentantworld in dararact disItvesj⍰
    perllow\nWild5ise?3 seemed'
```

$\tau = 0.5$ では、サンプラは高確率のトークン付近に保守的にとどまる。$\tau = 1.0$ では出力が目立って多様になる一方、時折崩れた単語が混じるようになる。そして $\tau = 2.0$ では、分布の平坦化によって裾のトークンが出力列を圧倒し、英語から逸脱するばかりか、有効な UTF-8 文字をなさないバイト列まで出力してしまう。次に、同じプロンプトに対して主要なデコーディング戦略をそれぞれの標準的な設定で直接比較してみよう。

```
strategies = {'greedy': dict(temperature=0), 'tau=1': dict(),
              'top-k=20': dict(k=20), 'top-p=0.9': dict(p=0.9),
              'min-p=0.05': dict(min_p=0.05)}
for name, strategy in strategies.items():
    out = generate(step_fn, prefix, 50, **strategy)
    d2l.print_wrapped(f'{name:>10}: {show(out)}')
```

```
    greedy: "The Time Traveller.\n\n'I had\nfrom the\nsidea.\n\n'I had\nfrom
    the\nsidea.\n\n'I had\nfrom the\nsidea.\n\n'I had\nf"
     tau=1: 'The Time Traveller, and sgat; so like a sttyj dayshindting,
    posserent and buth on that Gecting it leve against\nbut7 Sa crying\ndark
    weight at first'
  top-k=20: "The Time Traveller looked round in these gone. His animals
    had\nsoon.\n\n'I wood to see some\nthe Psychologist in\nmank, and I
    directive, and my hand. I was in"
 top-p=0.9: 'The Time Traveller raraments. Then I thought "As of trift his
    ownisterse of the Palace of Gallen Proburned this white.\n\n\'Chen my an
    exton.\n'
min-p=0.05: "The Time Traveller came to pates of\nbit\nmels. Then\n'Then.
    As\nbefore a bigroud of these chambuge steading\ninsmination. I"
```

当然ながら、1 編の中編小説で訓練された小さな単層 RNN では、いかなるデコーダを用いても完璧な文章は生成できず、規則ごとに 1 サンプルだけではノイズの多い証拠にすぎない。それでも定性的な対比は明白である。貪欲デコーディングは決定論的であり反復サイクルに陥るのに対し、どのサンプラも呼び出すたびにループのない新しいテキストを生成する。実務では、適度な温度と top-p または min-p による打ち切りを組み合わせることで流暢さと多様性のバランスがよくとれるため、これがオープンエンドな生成のデフォルトとなっている。

8.6.5 評価とコスト

8.6.5.1 生成テキストの品質

個々の生成文を目視で確認する以外に、デコーディング戦略を体系的に比較するにはどうすればよいだろうか。パープレキシティはこの問いに答えられない。パープレキシティはホールドアウトされた人間のテキストに対する**モデル**の評価指標であって特定のデコーディング規則を評価するものではなく、生成されたサンプルを自身のモデル尤度でスコア付けすれば、サンプリングがピークの尤度と引き換えに意図的に多様性を確保しているにもかかわらず、同語反復的に最大化手法が勝者となってしまうからである。

それでも、自動診断指標を使えば特定の失敗モードを的確に捉えられる。反復を定量化するため、ここでは **distinct-3** を用いる。これは生成された続きにおける全 3-gram に対するユニークな 3-gram の比率であり、3 トークンの連続列が一度も繰り返されないときに 1 となり、テキストがサイクルに陥ると 0 に近づく。以下では、検証データからランダムに選んだ長さ 6 トークンのプレフィックス 20 個に対し、貪欲デコーディング、ビームサーチ（$w = 4$）、および各サンプラの下で 100 トークンの続きを生成し、モデルの下でのトークンあたり平均対数確率と平均 distinct-3 の両方を報告する。

```
    def log_prob(ids, start):  # Mean log-probability of ids[start:] in one pass
        logits = seq_logits(ids[:-1])[start - 1:]
        lse = logits.max(1) + np.log(np.exp(logits - logits.max(1, keepdims=True)).sum(1))
        return float(np.mean(logits[np.arange(len(logits)), ids[start:]] - lse))

    def distinct(ids, n=3):  # Fraction of distinct n-grams in ids
        grams = [tuple(ids[i:i + n]) for i in range(len(ids) - n + 1)]
        return len(set(grams)) / len(grams)

    val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val])
    rows = np.random.default_rng().choice(len(val), 20, replace=False)
    prefixes = [[int(i) for i in val[r, :6]] for r in rows]
    decoders = {
        'greedy': lambda pre: generate(step_fn, pre, 100, temperature=0),
        'beam, w=4': lambda pre: beam_search(step_fn, pre, 100, beam_size=4)[0][1],
        'tau=1': lambda pre: generate(step_fn, pre, 100),
        'top-k=20': lambda pre: generate(step_fn, pre, 100, k=20),
        'top-p=0.9': lambda pre: generate(step_fn, pre, 100, p=0.9),
        'min-p=0.05': lambda pre: generate(step_fn, pre, 100, min_p=0.05)}
    for name, decode in decoders.items():
        outs = [decode(pre) for pre in prefixes]
        print(f'{name:>10}: log-prob per token '
              f'{np.mean([log_prob(o, 6) for o in outs]):.2f}, distinct-3 '
              f'{np.mean([distinct(o[6:]) for o in outs]):.2f}')
```

```
    greedy: log-prob per token -1.89, distinct-3 0.15
 beam, w=4: log-prob per token -1.46, distinct-3 0.19
     tau=1: log-prob per token -3.93, distinct-3 0.99
  top-k=20: log-prob per token -2.83, distinct-3 0.99
 top-p=0.9: log-prob per token -3.38, distinct-3 0.99
min-p=0.05: log-prob per token -2.78, distinct-3 0.98
```

これら 2 つの指標は、見事なまでに正反対の方向へ引っ張り合う。2 つの最大化戦略はいずれも任意のサンプラより顕著に高いトークンあたり対数確率を達成する（通常はビームサーチが貪欲デコーディングをわずかに上回る）が、3-gram の大半を繰り返すため distinct-3 のスコアは急落す

る。一方、4 つのサンプラはすべて 1 に近い distinct-3 を達成しており、Holtzman et al. (2020) が報告した尤度と多様性のトレードオフそのものをミニチュア規模で再現している。

サンプリング系統の内部では、4 つの規則すべてで多様性がほぼ飽和している一方、トークンあたり対数確率には明確な差が表れる。純粋な祖先サンプリング（$\tau = 1$）は平均対数確率が最も低く（トークンあたり約 -4 nat）、3 つの打ち切りサンプラはいずれも distinct-3 を損なうことなく尤度を大幅に引き上げる。先ほどのカットオフの保持数とも整合的に、最も広い台を残す top-p（0.9）が純粋なサンプリングに最も近く、より積極的に刈り込む top-k（20）と min-p（0.05）はトークンあたり対数確率を約 1 nat 押し上げており、裾の打ち切りが意図どおりに機能していることを直接示している。

distinct-3 のような自動指標は具体的な失敗モードを検出するうえで有用だが、一貫性、事実性、文体までは測れない。結局のところ、オープンエンドなテキスト生成は人間の読み手（あるいは較正された代理評価器）によって判断される一方、翻訳のように正解の参照文が存在する制約付きタスクは参照出力と照らし合わせて測定できる。要するに、パープレキシティは**モデル**の選択を導き、人間の選好（またはタスク固有のベンチマーク）が**デコーダ**の選択を導くのである。

8.6.5.2 生成のコスト

任意の自己回帰モデルをラップできるよう `step_fn` をステートレスに保ったため、新しいトークンを生成するたびにプレフィックス**全体**にわたって RNN を再実行することになり、T トークンの続きを生成するのに $O(T^2)$ 回のセル更新コストがかかる。また、自己回帰生成（autoregressive generation）は本質的に**逐次的**（sequential）でもある。すべての位置があらかじめ既知である教師強制（teacher forcing）による訓練とは異なり、トークン $t+1$ のスコアはトークン t が選ばれない限り計算できない。この逐次的な依存関係は本質的で避けられないものの、二次の再計算は RNN にとってはまったく不要である。隠れ状態がすでにプレフィックスを要約しているため、（`predict` が 8.4.2 節で行ったように）その状態をステップからステップへと引き継げば、系列長によらずトークンあたりちょうど 1 回のセル更新までコストを下げられる。再帰状態（recurrent state）を持たないモデルでは、過去の各トークンに対して計算した内容をキャッシュすることで同じ再計算を回避するが、その代わりプレフィックスの長さに比例して増大するメモリコストが生じる（第 10 章）。

生成トークンあたりの計算量とメモリが一定であることは、リカレント言語モデルを特徴づける効率上の利点である。しかし、まさにその固定サイズの状態がモデルのボトルネックにもなり、プレフィックスから記憶すべきあらゆる情報をわずか 128 個の浮動小数点数へと押し込めることになる（8.4.1.1 節）。次トークン分布が長距離の構造を捉えられていないモデルから、デコーディングの工夫だけで繊細な表現を引き出すことはできない。とはいえ、本節のデコーダはモデルを純粋なブラックボックスとして扱うため、ここで構築したすべてのアルゴリズムは、はるかに強力なモデルにもそのまま移行できる。

これをもって「**基礎**」パートは幕を閉じる。そしてその締めくくりは、まさにアーキテクチャ上の明確な境界線に位置している。本節の RNN は一定のコストで 1 トークンずつテキストを予測するが、実質的には直近 2〜3 トークンの履歴しか活用できておらず（8.4.2 節）、わずか数ステップで勾配が消失してしまうため（8.5 節）、それ以上多くの履歴を使えるよう学習できる見込みはほとんどない。ここからは 2 つの道が先へと続いている。1 つ目の道は、固定サイズの状態を保ちつつ再帰構造を再設計し、長いホライズンにわたって勾配が生き残るようにするアプローチである（第 12 章）。2 つ目の道は、固定サイズへの圧縮そのものを捨て去り、過去のすべてのトークンの表現を保持して、系列長に応じて増大するコストと引き換えにそれらへ直接注意を向けるアプローチ、すなわち現代の大規模言語モデルの中核をなす**アテンション**機構（第 10 章）である。根底にあるアーキテクチャがいずれであっても、本章で学んだサブワードトークナイザ、バイトあたりビット数（bits-per-byte）による評価、そしてデコーディングアルゴリズムはそのまま引き継がれる。

8.6.6 まとめ

デコーディングは、言語モデルの次トークン分布を具体的なトークン系列へと変換する処理である。最大化手法は高確率の出力を求めるものであり、貪欲デコーディングは各ステップで局所的に確定させるため近視眼性と反復ループの両方に悩まされる一方、ビームサーチは w 倍のコストで上位 w 個の仮説を追跡し（候補が早期終了しうる場合は長さ正規化を用いる）、出力空間の狭い制約付きタスクで優れた性能を発揮する。オープンエンドな生成では、サンプリングが自然な多様性を保ち、温度スケーリング（Equation 8.27）と top-k、top-p、または min-p による裾の打ち切りによって低確率の裾のトークンがロールアウトを破綻させるのを防ぐことで、distinct-3 の多様性では純粋なサンプリングに匹敵しつつ、はるかに確率の高いテキストを生成できる。最後に、パープレキシティが評価するのはデコーダではなくモデルであり、再帰的な隠れ状態を引き継ぐことで素朴なステートレスデコーディングにおける $O(T^2)$ の再計算を回避できる。

8.6.7 演習問題

1. **ビームサーチの性質。**
    1. ビーム幅 2、Figure 8.14 の 5 トークンの語彙、3 ステップを考える。ビームサーチが最も確率の高い 3 トークンの系列を返さないような条件付き確率を割り当て、両方の系列の確率を計算せよ。この構成例から、ビームサーチが最も確率の高い系列を見逃しうる仕組みについて何が言えるか。
    2. モデルがいかなるものであっても、T ステップにわたるビームサーチが全 $|\mathcal{V}|^T$ 通りの系列のうち最も確率の高いものを返すことを保証する最小のビーム幅はいくらか。
    3. $w \le |\mathcal{V}|$ とし、系列終了トークンは出現しないと仮定する。`step_fn` に対する呼び出しのうち、`beam_search` が T ステップにわたって行う回数を求めよ。また、全探索がすべてのプレフィックスを 1 回ずつ評価する場合の呼び出し回数も求めよ。$|\mathcal{V}| = 1{,}027$、$T = 50$、$w = 4$ のときに両方の回数を計算せよ。

2. [code] **長さ正規化。** `beam_search` を $w = 4$ として、`'The Time Traveller'` と `'It seemed to me that'` の後に最大 40 トークンまで実行せよ。改行文字の ID である `data.tokenizer.encode('\n')[0]` を `eos_id` として渡し、β（引数 `alpha`）を $\{0, 0.75, 1.5\}$ にわたって変化させよ。最上位となった候補の長さと合計対数確率を比較し、Equation 8.26 を用いてその傾向を説明せよ。

3. **温度の下での打ち切り。** `sample_next` は、すべてのカットオフを温度スケーリング後の分布 P_τ（Equation 8.27）に対して適用する。min-p の条件 $P_\tau(x \mid x_{<t}) \ge p_{\min} \max_{x'} P_\tau(x' \mid x_{<t})$ を、温度スケーリング前の確率に関する条件として書き表せ。5 トークンの分布 0.45、0.25、0.15、0.10、0.05 に対して、top-k ($k = 2$)、top-p ($p = 0.85$)、min-p ($p_{\min} = 0.5$) が $\tau = 2$ において保持する集合を計算せよ。3 つの規則のうち、あらゆる温度で同じ集合を保持するものはどれか。

4. [code] **Top-p と min-p の比較。**●$\tau = 1.5$ において、グリッド上の値にわたって p（top-p 用）と $p_{\min}$（min-p 用）を変化させよ。各設定について、任意に選んだ 5 つのプレフィックスから 50 トークンの続きを生成し、生成トークン全体にわたる 2 つの平均値、すなわちその規則が保持するトークン数と、温度スケーリング前のモデルの下で抽出されたトークンの対数確率を記録せよ。規則ごとに 1 本の曲線として、1 つ目の平均値を横軸、2 つ目の平均値を縦軸にとりプロットせよ。保持トークン数の平均が同じとき、より確率の高いトークンを抽出するのはどちらの規則か。得られた結果を Nguyen et al. (2025) の知見と比較せよ。

5. [code] **トークンあたりの定数コスト。**●`step_fn` は生成トークンごとにプレフィックス全体にわたってネットワークを再実行するため、T トークンの続きには $O(T^2)$ 回のセル更新コストがかかる（8.6.5.2 節）。

1. `step_fn` のステートフルな代替関数を作成せよ。ただし、`model.rnn` の隠れ状態を保持して最新のトークンのみを消費し、1 回の呼び出しにつき 1 回のセル更新だけを行うようにすること。両方のバージョンで `generate` が同じ貪欲な続きを生成することを検証せよ。
2. 50、200、800 トークンに対する `generate` の実行時間を両方のバージョンで計測せよ。その比から、それぞれの実行時間がトークン数に対してどのように増加するかを推定し、セル更新回数が予測する二次増加および線形増加からの乖離があればその理由を説明せよ。
3. 同じ置き換えがそのままでは `beam_search` に適用できないのはなぜか。また、どのような状態管理を行えば動作させられるか。

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.

Alsallakh, Bilal, Narine Kokhlikyan, Vivek Miglani, Jun Yuan, and Orion Reblitz-Richardson. 2020. “Mind the PAD – CNNs Can Develop Blind Spots.” **ArXiv:2010.02178**. https://arxiv.org/abs/2010.02178.

Arora, Sanjeev, Zhiyuan Li, and Kaifeng Lyu. 2019. “Theoretical Analysis of Auto Rate-Tuning by Batch Normalization.” **International Conference on Learning Representations**. https://arxiv.org/abs/1812.03981.

Arpit, Devansh, Stanisław Jastrzębski, Nicolas Ballas, et al. 2017. “A Closer Look at Memorization in Deep Networks.” **International Conference on Machine Learning**, 233–42.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. “Layer Normalization.” **ArXiv:1607.06450**. https://arxiv.org/abs/1607.06450.

Bachlechner, Thomas, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W. Cottrell, and Julian McAuley. 2021. “ReZero Is All You Need: Fast Convergence at Large Depth.” **Uncertainty in Artificial Intelligence**. https://arxiv.org/abs/2003.04887.

Balduzzi, David, Marcus Frean, Lennox Leary, J. P. Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. 2017. “The Shattered Gradients Problem: If ResNets Are the Answer, Then What Is the Question?” **Proceedings of the 34th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 70: 342–50.

Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. 2006. “SURF: Speeded up Robust Features.” **European Conference on Computer Vision**, 404–17. https://doi.org/10.1007/11744023_32.

Bayes, Thomas. 1763. “An Essay Towards Solving a Problem in the Doctrine of Chances.” **Philosophical Transactions of the Royal Society of London** 53: 370–418.

Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. “Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off.” **Proceedings of the National Academy of Sciences** 116 (32): 15849–54.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks.” **Advances in Neural Information Processing Systems** 28. https://arxiv.org/abs/1506.03099.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. “A Neural Probabilistic Language Model.” **Journal of Machine Learning Research** 3 (Feb): 1137–55. https://jmlr.org/papers/v3/bengio03a.html.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation.” **arXiv Preprint arXiv:1308.3432**.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. 1994. “Learning Long-Term Dependencies with Gradient Descent Is Difficult.” **IEEE Transactions on Neural Networks** 5 (2): 157–66. https://doi.org/10.1109/72.279181.

Beyer, Lucas, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. 2020. “Are We Done with ImageNet?” **ArXiv:2006.07159**. https://arxiv.org/abs/2006.07159.

Bishop, Chris M. 1995. “Training with Noise Is Equivalent to Tikhonov Regularization.” **Neural Computation** 7 (1): 108–16. https://doi.org/10.1162/neco.1995.7.1.108.

Black, Fischer, and Myron Scholes. 1973. “The Pricing of Options and Corporate Liabilities.” **Journal of Political Economy** 81: 637–54. https://doi.org/10.1086/260062.

Blum, Avrim, and Moritz Hardt. 2015. “The Ladder: A Reliable Leaderboard for Machine Learning Competitions.” **Proceedings of the 32nd International Conference on Machine Learning**, Proceedings of machine learning research, vol. 37: 1006–14.

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.

Brock, Andrew, Soham De, Samuel L. Smith, and Karen Simonyan. 2021. “High-Performance Large-Scale Image Recognition Without Normalization.” **International Conference on Machine Learning**. https://arxiv.org/abs/2102.06171.

Byrd, Jonathon, and Zachary C. Lipton. 2019. “What Is the Effect of Importance Weighting in Deep Learning?” **Proceedings of the 36th International Conference on Machine Learning**, Proceedings of machine learning research, vol. 97: 872–81.

Chen, Stanley F., and Joshua Goodman. 1999. “An Empirical Study of Smoothing Techniques for Language Modeling.” **Computer Speech & Language** 13 (4): 359–93. https://doi.org/10.1006/csla.1999.0128.

Chen, Tianqi, Mu Li, Yutian Li, et al. 2015. “MXNET: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems.” **ArXiv:1512.01274**. https://arxiv.org/abs/1512.01274.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. “Training Deep Nets with Sublinear Memory Cost.” **arXiv Preprint arXiv:1604.06174**.

Chollet, François. 2017. “Xception: Deep Learning with Depthwise Separable Convolutions.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/1610.02357.

Chu, Xiangxiang, Liang Li, and Bo Zhang. 2024. “Make RepVGG Greater Again: A Quantization-Aware Approach.” **AAAI Conference on Artificial Intelligence**. https://arxiv.org/abs/2212.01593.

Cover, Thomas M., and Peter E. Hart. 1967. “Nearest Neighbor Pattern Classification.” **IEEE Transactions on Information Theory** 13 (1): 21–27.

Cubuk, Ekin D., Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2020. “RandAugment: Practical Automated Data Augmentation with a Reduced Search Space.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops**. https://arxiv.org/abs/1909.13719.

Cybenko, George. 1989. “Approximation by Superpositions of a Sigmoidal Function.” **Mathematics of Control, Signals and Systems** 2 (4): 303–14. https://doi.org/10.1007/BF02551274.

Dalal, Navneet, and Bill Triggs. 2005. “Histograms of Oriented Gradients for Human Detection.” **2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)** 1: 886–93. https://doi.org/10.1109/cvpr.2005.177.

De Cock, Dean. 2011. “Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project.” **Journal of Statistics Education** 19 (3). https://doi.org/10.1080/10691898.2011.11889627.

De, Soham, and Samuel L. Smith. 2020. “Batch Normalization Biases Residual Blocks Towards the Identity Function in Deep Networks.” **Advances in Neural Information Processing Systems** 33. https://arxiv.org/abs/2002.10444.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. “Imagenet: A Large-Scale Hierarchical Image Database.” **2009 IEEE Conference on Computer Vision and Pattern Recognition**, 248–55. https://doi.org/10.1109/cvpr.2009.5206848.

Der Kiureghian, Armen, and Ove Ditlevsen. 2009. “Aleatory or Epistemic? Does It Matter?” **Structural Safety** 31 (2): 105–12. https://doi.org/10.1016/j.strusafe.2008.06.020.

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.

Ding, Xiaohan, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. “RepVGG: Making VGG-Style ConvNets Great Again.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/2101.03697.

Ding, Xiaohan, Xiangyu Zhang, Yizhuang Zhou, Jungong Han, Guiguang Ding, and Jian Sun. 2022. “Scaling up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/2203.06717.

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.

Dwork, Cynthia, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. “Preserving Statistical Validity in Adaptive Data Analysis.” **ACM Symposium on Theory of Computing**, 117–26.

Elman, Jeffrey L. 1990. “Finding Structure in Time.” **Cognitive Science** 14 (2): 179–211. https://doi.org/10.1016/0364-0213(90)90002-E.

Falcon, William, and The PyTorch Lightning team. 2019. **PyTorch Lightning**. https://doi.org/10.5281/zenodo.3828935.

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.

Field, David J. 1987. “Relations Between the Statistics of Natural Images and the Response Properties of Cortical Cells.” **JOSA A** 4 (12): 2379–94. https://doi.org/10.1364/josaa.4.002379.

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.

Fukushima, Kunihiko. 1980. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position.” **Biological Cybernetics** 36 (4): 193–202. https://doi.org/10.1007/BF00344251.

Gage, Philip. 1994. “A New Algorithm for Data Compression.” **The C Users Journal** 12 (2): 23–38.

Gal, Yarin, and Zoubin Ghahramani. 2016. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning.” **International Conference on Machine Learning**, 1050–59. https://arxiv.org/abs/1506.02142.

Gauss, Carl Friedrich. 1809. “Theoria Motus Corporum Coelestium.” In **Werke**. Königlich Preussische Akademie der Wissenschaften. https://doi.org/10.1007/978-3-642-92478-1.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 580–87. https://doi.org/10.1109/CVPR.2014.81.

Glorot, Xavier, and Yoshua Bengio. 2010. “Understanding the Difficulty of Training Deep Feedforward Neural Networks.” **Proceedings of the 13th International Conference on Artificial Intelligence and Statistics**, 249–56. https://proceedings.mlr.press/v9/glorot10a.html.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. 2011. “Deep Sparse Rectifier Neural Networks.” **Proceedings of the 14th International Conference on Artificial Intelligence and Statistics**, 315–23. https://proceedings.mlr.press/v15/glorot11a.html.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. **Deep Learning**. MIT Press.

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.

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.

Grinsztajn, Léo, Edouard Oyallon, and Gaël Varoquaux. 2022. “Why Do Tree-Based Models Still Outperform Deep Learning on Tabular Data?” **Advances in Neural Information Processing Systems** 35: 507–20. https://arxiv.org/abs/2207.08815.

Guo, Chuan, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. “On Calibration of Modern Neural Networks.” **International Conference on Machine Learning**, 1321–30. https://arxiv.org/abs/1706.04599.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. “Array Programming with NumPy.” **Nature** 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

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, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016b. “Identity Mappings in Deep Residual Networks.” **European Conference on Computer Vision**, 630–45. https://doi.org/10.1007/978-3-319-46493-0_38.

He, Tong, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2019. “Bag of Tricks for Image Classification with Convolutional Neural Networks.” **IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 558–67. https://arxiv.org/abs/1812.01187.

Hendrycks, Dan, and Kevin Gimpel. 2016. “Gaussian Error Linear Units (GELUs).” **ArXiv:1606.08415**. https://arxiv.org/abs/1606.08415.

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors.” **arXiv Preprint arXiv:1207.0580**.

Hoeffding, Wassily. 1963. “Probability Inequalities for Sums of Bounded Random Variables.” **Journal of the American Statistical Association** 58 (301): 13–30.

Hoerl, Arthur E, and Robert W Kennard. 1970. “Ridge Regression: Biased Estimation for Nonorthogonal Problems.” **Technometrics** 12 (1): 55–67.

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.

Howard, Andrew G., Menglong Zhu, Bo Chen, et al. 2017. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” **ArXiv:1704.04861**. https://arxiv.org/abs/1704.04861.

Hu, Jie, Li Shen, and Gang Sun. 2018. “Squeeze-and-Excitation Networks.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 7132–41. https://doi.org/10.1109/cvpr.2018.00745.

Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. “Densely Connected Convolutional Networks.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 4700–4708. https://doi.org/10.1109/cvpr.2017.243.

Huang, Gao, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. 2016. “Deep Networks with Stochastic Depth.” **European Conference on Computer Vision**. https://arxiv.org/abs/1603.09382.

Hubel, David H, and Torsten N Wiesel. 1959. “Receptive Fields of Single Neurones in the Cat’s Striate Cortex.” **Journal of Physiology** 148 (3): 574–91. https://doi.org/10.1113/jphysiol.1959.sp006308.

Hubel, David H, and Torsten N Wiesel. 1962. “Receptive Fields, Binocular Interaction and Functional Architecture in the Cat’s Visual Cortex.” **Journal of Physiology** 160 (1): 106–54. https://doi.org/10.1113/jphysiol.1962.sp006837.

Hubel, David H, and Torsten N Wiesel. 1968. “Receptive Fields and Functional Architecture of Monkey Striate Cortex.” **Journal of Physiology** 195 (1): 215–43. https://doi.org/10.1113/jphysiol.1968.sp008455.

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**.

Ioffe, Sergey, and Christian Szegedy. 2015. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” **International Conference on Machine Learning**, 448–56. https://arxiv.org/abs/1502.03167.

Izmailov, Pavel, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. 2018. “Averaging Weights Leads to Wider Optima and Better Generalization.” **Uncertainty in Artificial Intelligence**, 876–85. https://arxiv.org/abs/1803.05407.

Jaeger, Herbert. 2002. **Tutorial on Training Recurrent Neural Networks, Covering BPTT, RTRL, EKF and the “Echo State Network” Approach**. GMD-Forschungszentrum Informationstechnik Bonn.

Jelinek, Frederick, Robert L. Mercer, Lalit R. Bahl, and James K. Baker. 1977. “Perplexity—a Measure of the Difficulty of Speech Recognition Tasks.” **Journal of the Acoustical Society of America** 62 (S1): S63.

Kneser, Reinhard, and Hermann Ney. 1995. “Improved Backing-Off for M-Gram Language Modeling.” **International Conference on Acoustics, Speech, and Signal Processing** 1: 181–84. https://doi.org/10.1109/ICASSP.1995.479394.

Koh, Pang Wei, Shiori Sagawa, Henrik Marklund, et al. 2021. “WILDS: A Benchmark of in-the-Wild Distribution Shifts.” **International Conference on Machine Learning**, 5637–64. https://arxiv.org/abs/2012.07421.

Kohavi, Ron. 1995. “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.” **International Joint Conference on Artificial Intelligence** 14: 1137–45.

Kolmogorov, Andrey. 1933. **Grundbegriffe Der Wahrscheinlichkeitsrechnung**. Springer. https://doi.org/10.1007/978-3-642-49888-6.

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.

Kuzovkin, Ilya, Raul Vicente, Mathilde Petton, et al. 2018. “Activations of Deep Convolutional Neural Networks Are Aligned with Gamma Band Activity of Human Visual Cortex.” **Communications Biology** 1 (1): 1–12. https://doi.org/10.1038/s42003-018-0110-y.

LeCun, Yann, Bernhard Boser, John S Denker, et al. 1989. “Backpropagation Applied to Handwritten Zip Code Recognition.” **Neural Computation** 1 (4): 541–51. https://doi.org/10.1162/neco.1989.1.4.541.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. “Gradient-Based Learning Applied to Document Recognition.” **Proceedings of the IEEE** 86 (11): 2278–324. https://doi.org/10.1109/5.726791.

LeCun, Yann, LD Jackel, Leon Bottou, et al. 1995. “Comparison of Learning Algorithms for Handwritten Digit Recognition.” **International Conference on Artificial Neural Networks**, 53–60.

Legendre, Adrien Marie. 1805. **Nouvelles méthodes Pour La détermination Des Orbites Des Comètes**. Courcier.

Leshno, Moshe, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. 1993. “Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function.” **Neural Networks** 6 (6): 861–67.

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, Xiang, Shuo Chen, Xiaolin Hu, and Jian Yang. 2019. “Understanding the Disharmony Between Dropout and Batch Normalization by Variance Shift.” **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 2682–90. https://arxiv.org/abs/1801.05134.

Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. “Network in Network.” **ArXiv:1312.4400**. https://arxiv.org/abs/1312.4400.

Linnainmaa, Seppo. 1970. “The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors.” Master’s thesis, University of Helsinki.

Lipton, Zachary, Yu-Xiang Wang, and Alexander Smola. 2018. “Detecting and Correcting for Label Shift with Black Box Predictors.” **International Conference on Machine Learning**, 3122–30. https://arxiv.org/abs/1802.03916.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang. 2018. “DARTS: Differentiable Architecture Search.” **ArXiv:1806.09055**. https://arxiv.org/abs/1806.09055.

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.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2015. “Fully Convolutional Networks for Semantic Segmentation.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 3431–40. https://doi.org/10.1109/CVPR.2015.7298965.

Lopez-Paz, David, and Maxime Oquab. 2017. “Revisiting Classifier Two-Sample Tests.” **International Conference on Learning Representations**.

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.

Lowe, David G. 2004. “Distinctive Image Features from Scale-Invariant Keypoints.” **International Journal of Computer Vision** 60 (2): 91–110. https://doi.org/10.1023/b:visi.0000029664.99615.94.

Luo, Ping, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. 2018. “Towards Understanding Regularization in Batch Normalization.” **ArXiv:1809.00846**. https://arxiv.org/abs/1809.00846.

Luo, Wenjie, Yujia Li, Raquel Urtasun, and Richard Zemel. 2016. “Understanding the Effective Receptive Field in Deep Convolutional Neural Networks.” **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/1701.04128.

Markowitz, Harry. 1952. “Portfolio Selection.” **The Journal of Finance** 7 (1): 77–91.

McCulloch, Warren S, and Walter Pitts. 1943. “A Logical Calculus of the Ideas Immanent in Nervous Activity.” **Bulletin of Mathematical Biophysics** 5 (4): 115–33. https://doi.org/10.1016/s0092-8240(05)80006-0.

McKinney, Wes. 2010. “Data Structures for Statistical Computing in Python.” **Proceedings of the 9th Python in Science Conference**, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, et al. 2018. “Mixed Precision Training.” **International Conference on Learning Representations**.

Mikolov, Tomáš, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2011. “Extensions of Recurrent Neural Network Language Model.” **IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)**, 5528–31. https://doi.org/10.1109/ICASSP.2011.5947611.

Mishkin, Dmytro, and Jiri Matas. 2016. “All You Need Is a Good Init.” **International Conference on Learning Representations**. https://arxiv.org/abs/1511.06422.

Moon, Taesup, Alex Smola, Yi Chang, and Zhaohui Zheng. 2010. “Intervalrank: Isotonic Regression with Listwise and Pairwise Constraints.” **Proceedings of the 3rd ACM International Conference on Web Search and Data Mining**, 151–60. https://doi.org/10.1145/1718487.1718520.

Nair, Vinod, and Geoffrey E Hinton. 2010. “Rectified Linear Units Improve Restricted Boltzmann Machines.” **ICML**, 807–14. https://dl.acm.org/doi/10.5555/3104322.3104425.

Ney, Hermann, Ute Essen, and Reinhard Kneser. 1994. “On Structuring Probabilistic Dependences in Stochastic Language Modelling.” **Computer Speech & Language** 8 (1): 1–38. https://doi.org/10.1006/csla.1994.1001.

Nguyen, Minh Nhat, Andrew Baker, Clement Neo, Allen Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. 2025. "Turning up the Heat: Min-p Sampling for Creative and Coherent LLM Outputs." **International Conference on Learning Representations**. https://arxiv.org/abs/2407.01082.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." **International Conference on Machine Learning**, 1310–18.

Paszke, Adam, Sam Gross, Francisco Massa, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." **Advances in Neural Information Processing Systems** 32: 8026–37. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Peters, Jonas, Dominik Janzing, and Bernhard Schölkopf. 2017. **Elements of Causal Inference: Foundations and Learning Algorithms**. MIT Press. https://doi.org/10.7551/mitpress/11283.001.0001.

Pleiss, Geoff, Danlu Chen, Gao Huang, Tongcheng Li, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. "Memory-Efficient Implementation of Densenets." **ArXiv:1707.06990**. https://arxiv.org/abs/1707.06990.

Power, Alethea, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. 2022. "Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets." **ArXiv:2201.02177**. https://arxiv.org/abs/2201.02177.

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.

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.

Radosavovic, Ilija, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. 2019. "On Network Design Spaces for Visual Recognition." **Proceedings of the IEEE/CVF International Conference on Computer Vision**, 1882–90. https://doi.org/10.1109/iccv.2019.00052.

Radosavovic, Ilija, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. "Designing Network Design Spaces." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**, 10428–36. https://doi.org/10.1109/cvpr42600.2020.01044.

Rahimi, Ali, and Benjamin Recht. 2007. "Random Features for Large-Scale Kernel Machines." **Advances in Neural Information Processing Systems** 20.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. "Sequence Level Training with Recurrent Neural Networks." **International Conference on**

Learning Representations. https://arxiv.org/abs/1511.06732.

Recht, Benjamin, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. 2019. "Do ImageNet Classifiers Generalize to ImageNet?" **International Conference on Machine Learning**, 5389–400. https://arxiv.org/abs/1902.10811.

Riesenhuber, Maximilian, and Tomaso Poggio. 1999. "Hierarchical Models of Object Recognition in Cortex." **Nature Neuroscience** 2 (11): 1019–25. https://doi.org/10.1038/14819.

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.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams. 1986. "Learning Representations by Back-Propagating Errors." **Nature** 323 (6088): 533–36. https://doi.org/10.1038/323533a0.

Russakovsky, Olga, Jia Deng, Hao Su, et al. 2015. "ImageNet Large Scale Visual Recognition Challenge." **International Journal of Computer Vision** 115 (3): 211–52. https://doi.org/10.1007/s11263-015-0816-y.

Saerens, Marco, Patrice Latinne, and Christine Decaestecker. 2002. "Adjusting the Outputs of a Classifier to New a Priori Probabilities: A Simple Procedure." **Neural Computation** 14 (1): 21–41. https://doi.org/10.1162/089976602753284446.

Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/1801.04381.

Santurkar, Shibani, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. 2018. "How Does Batch Normalization Help Optimization?" **Advances in Neural Information Processing Systems**, 2483–93. https://doi.org/10.5555/3327345.3327508.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." **Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics**, 1715–25. https://doi.org/10.18653/v1/P16-1162.

Shannon, Claude Elwood. 1951. "Prediction and Entropy of Printed English." **The Bell System Technical Journal** 30 (1): 50–64. https://doi.org/10.1002/j.1538-7305.1951.tb01366.x.

Shimodaira, Hidetoshi. 2000. "Improving Predictive Inference Under Covariate Shift by Weighting the Log-Likelihood Function." **Journal of Statistical Planning and Inference** 90 (2): 227–44.

Shwartz-Ziv, Ravid, and Amitai Armon. 2022. "Tabular Data: Deep Learning Is Not All You Need." **Information Fusion** 81: 84–90. https://doi.org/10.1016/j.inffus.2021.11.011.

Simonyan, Karen, and Andrew Zisserman. 2015. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” **International Conference on Learning Representations**. https://arxiv.org/abs/1409.1556.

Sivic, Josef, and Andrew Zisserman. 2003. “Video Google: A Text Retrieval Approach to Object Matching in Videos.” **Proceedings of the IEEE International Conference on Computer Vision** 3: 1470–77. https://doi.org/10.1109/iccv.2003.1238663.

Soudry, Daniel, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. 2018. “The Implicit Bias of Gradient Descent on Separable Data.” **Journal of Machine Learning Research** 19 (70): 1–57.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” **Journal of Machine Learning Research** 15 (1): 1929–58. https://doi.org/10.5555/2627435.2670313.

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.

Szegedy, Christian, Wei Liu, Yangqing Jia, et al. 2015. “Going Deeper with Convolutions.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 1–9. https://doi.org/10.1109/cvpr.2015.7298594.

Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. “Rethinking the Inception Architecture for Computer Vision.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 2818–26. https://doi.org/10.1109/cvpr.2016.308.

Tan, Mingxing, and Quoc Le. 2019. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” **International Conference on Machine Learning**, 6105–14. https://proceedings.mlr.press/v97/tan19a.html.

Tan, Mingxing, and Quoc V. Le. 2021. “EfficientNetV2: Smaller Models and Faster Training.” **International Conference on Machine Learning**. https://arxiv.org/abs/2104.00298.

Telgarsky, Matus. 2016. “Benefits of Depth in Neural Networks.” **Conference on Learning Theory**, 1517–39.

Tibshirani, Robert. 1996. “Regression Shrinkage and Selection via the Lasso.” **Journal of the Royal Statistical Society: Series B** 58 (1): 267–88.

Tikhonov, A. N., and V. Y. Arsenin. 1977. **Solutions of Ill-Posed Problems**. W.H. Winston. https://doi.org/10.1137/1.9780898719741.

Touvron, Hugo, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. 2021. “Going Deeper with Image Transformers.” **IEEE/CVF International Conference on Computer Vision**, 32–42. https://arxiv.org/abs/2103.17239.

Tsoumakas, Grigorios, and Ioannis Katakis. 2007. "Multi-Label Classification: An Overview." **International Journal of Data Warehousing and Mining** 3 (3): 1–13. https://doi.org/10.4018/jdwm.2007070101.

Vapnik, V., and A. Chervonenkis. 1971. "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities." **Theory Probab. Appl.** 16 (2): 264–81. https://doi.org/10.1137/1116025.

Veit, Andreas, Michael J Wilber, and Serge Belongie. 2016. "Residual Networks Behave Like Ensembles of Relatively Shallow Networks." **Advances in Neural Information Processing Systems**. https://arxiv.org/abs/1605.06431.

Waibel, Alex, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. 1989. "Phoneme Recognition Using Time-Delay Neural Networks." **IEEE Transactions on Acoustics, Speech, and Signal Processing** 37 (3): 328–39. https://doi.org/10.1109/29.21701.

Wengert, Robert Edwin. 1964. "A Simple Automatic Derivative Evaluation Program." **Communications of the ACM** 7 (8): 463–64. https://doi.org/10.1145/355588.365726.

Werbos, Paul J. 1990. "Backpropagation Through Time: What It Does and How to Do It." **Proceedings of the IEEE** 78 (10): 1550–60. https://doi.org/10.1109/5.58337.

Wightman, Ross, Hugo Touvron, and Hervé Jégou. 2021. "ResNet Strikes Back: An Improved Training Procedure in Timm." **ArXiv:2110.00476**. https://arxiv.org/abs/2110.00476.

Wolpert, David H. 1996. "The Lack of a Priori Distinctions Between Learning Algorithms." **Neural Computation** 8 (7): 1341–90.

Woo, Sanghyun, Shoubhik Debnath, Ronghang Hu, et al. 2023. "ConvNeXt V2: Co-Designing and Scaling ConvNets with Masked Autoencoders." **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. https://arxiv.org/abs/2301.00808.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation." **ArXiv:1609.08144**. https://arxiv.org/abs/1609.08144.

Wu, Yuxin, and Kaiming He. 2018. "Group Normalization." **European Conference on Computer Vision**. https://arxiv.org/abs/1803.08494.

Xiao, Han, Kashif Rasul, and Roland Vollgraf. 2017. "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms." **ArXiv:1708.07747**. https://arxiv.org/abs/1708.07747.

Xie, Saining, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. "Aggregated Residual Transformations for Deep Neural Networks." **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 1492–500. https://doi.org/10.1109/cvpr.2017.634.

Yamaguchi, Kouichi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. 1990. “A Neural Network for Speaker-Independent Isolated Word Recognition.” **First International Conference on Spoken Language Processing**. https://doi.org/10.21437/icslp.1990-282.

Yu, Fisher, and Vladlen Koltun. 2016. “Multi-Scale Context Aggregation by Dilated Convolutions.” **International Conference on Learning Representations**. https://arxiv.org/abs/1511.07122.

Yun, Sangdoo, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. 2019. “CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features.” **Proceedings of the IEEE/CVF International Conference on Computer Vision**. https://arxiv.org/abs/1905.04899.

Zhang, Biao, and Rico Sennrich. 2019. “Root Mean Square Layer Normalization.” **Advances in Neural Information Processing Systems** 32.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. “Understanding Deep Learning (Still) Requires Rethinking Generalization.” **Communications of the ACM** 64 (3): 107–15. https://doi.org/10.1145/3446776.

Zhang, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. “mixup: Beyond Empirical Risk Minimization.” **International Conference on Learning Representations**. https://arxiv.org/abs/1710.09412.

Zhang, Hongyi, Yann N. Dauphin, and Tengyu Ma. 2019. “Fixup Initialization: Residual Learning Without Normalization.” **International Conference on Learning Representations**. https://arxiv.org/abs/1901.09321.

Zhang, Richard. 2019. “Making Convolutional Networks Shift-Invariant Again.” **International Conference on Machine Learning**. https://arxiv.org/abs/1904.11486.

Zhang, Wei, Jun Tanida, Kazuyoshi Itoh, and Yoshiki Ichioka. 1988. “Shift-Invariant Pattern Recognition Neural Network and Its Optical Architecture.” **Proceedings of Annual Conference of the Japan Society of Applied Physics**.

Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices.” **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**, 6848–56.

Zoph, Barret, and Quoc V Le. 2016. “Neural Architecture Search with Reinforcement Learning.” **ArXiv:1611.01578**. https://arxiv.org/abs/1611.01578.