Dive into Deep Learning

Release 0.14.3

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Aug 11, 2020

Contents

Preface **1**

Installation **9**

Notation **13**

1 Introduction **17**
1.1 A Motivating Example . . . 18
1.2 The Key Components: Data, Models, and Algorithms . . . 20
1.3 Kinds of Machine Learning . . . 23
1.4 Roots . . . 35
1.5 The Road to Deep Learning . . . 37
1.6 Success Stories . . . 39

2 Preliminaries **43**
2.1 Data Manipulation . . . 43
2.1.1 Getting Started . . . 44
2.1.2 Operations . . . 46
2.1.3 Broadcasting Mechanism . . . 48
2.1.4 Indexing and Slicing . . . 48
2.1.5 Saving Memory . . . 49
2.1.6 Conversion to Other Python Objects . . . 50
2.2 Data Preprocessing . . . 51
2.2.1 Reading the Dataset . . . 51
2.2.2 Handling Missing Data . . . 52
2.2.3 Conversion to the Tensor Format . . . 53
2.3 Linear Algebra . . . 54
2.3.1 Scalars . . . 54
2.3.2 Vectors . . . 54
2.3.3 Matrices . . . 56
2.3.4 Tensors . . . 57
2.3.5 Basic Properties of Tensor Arithmetic . . . 58
2.3.6 Reduction . . . 59

2.4.3 Gradients . . . 71
2.4.4 Chain Rule . . . 71
2.5 Automatic Differentiation . . . 72
2.5.1 A Simple Example . . . 72
2.5.2 Backward for Non-Scalar Variables . . . 74
2.5.3 Detaching Computation . . . 74
2.5.4 Computing the Gradient of Python Control Flow . . . 75
2.6 Probability . . . 76
2.6.1 Basic Probability Theory . . . 77
2.6.2 Dealing with Multiple Random Variables . . . 81
2.6.3 Expectation and Variance . . . 84
2.7 Documentation . . . 85
2.7.1 Finding All the Functions and Classes in a Module . . . 85
2.7.2 Finding the Usage of Specific Functions and Classes . . . 85

3 Linear Neural Networks **89**
3.1 Linear Regression . . . 89
3.1.1 Basic Elements of Linear Regression . . . 89
3.1.2 Vectorization for Speed . . . 93
3.1.3 The Normal Distribution and Squared Loss . . . 95
3.1.4 From Linear Regression to Deep Networks . . . 96
3.2 Linear Regression Implementation from Scratch . . . 99
3.2.1 Generating the Dataset . . . 99
3.2.2 Reading the Dataset . . . 100
3.2.3 Initializing Model Parameters . . . 101
3.2.4 Defining the Model . . . 102
3.2.5 Defining the Loss Function . . . 102
3.2.6 Defining the Optimization Algorithm . . . 102
3.2.7 Training . . . 103
3.3 Concise Implementation of Linear Regression . . . 105
3.3.1 Generating the Dataset . . . 105
3.3.2 Reading the Dataset . . . 105
3.3.3 Defining the Model . . . 106
3.3.4 Initializing Model Parameters . . . 107
3.3.5 Defining the Loss Function . . . 107
3.3.6 Defining the Optimization Algorithm . . . 107
3.3.7 Training . . . 108
3.4 Softmax Regression . . . 109
3.4.1 Classification Problem . . . 110
3.4.2 Network Architecture . . . 110
3.4.3 Softmax Operation . . . 111
3.4.4 Vectorization for Minibatches . . . 112
3.4.5 Loss Function . . . 112

3.6.2 Defining the Softmax Operation . . . 120
3.6.3 Defining the Model . . . 121
3.6.4 Defining the Loss Function . . . 121
3.6.5 Classification Accuracy . . . 122
3.6.6 Training . . . 123
3.6.7 Prediction . . . 125
3.7 Concise Implementation of Softmax Regression . . . 126
3.7.1 Initializing Model Parameters . . . 127
3.7.2 Softmax Implementation Revisited . . . 127
3.7.3 Optimization Algorithm . . . 128
3.7.4 Training . . . 128

4 Multilayer Perceptrons **131**
4.1 Multilayer Perceptrons . . . 131
4.1.1 Hidden Layers . . . 131
4.1.2 Activation Functions . . . 134
4.2 Implementation of Multilayer Perceptrons from Scratch . . . 139
4.2.1 Initializing Model Parameters . . . 139
4.2.2 Activation Function . . . 140
4.2.3 Model . . . 140
4.2.4 Loss Function . . . 140
4.2.5 Training . . . 141
4.3 Concise Implementation of Multilayer Perceptrons . . . 142
4.3.1 Model . . . 142
4.4 Model Selection, Underfitting, and Overfitting . . . 144
4.4.1 Training Error and Generalization Error . . . 144
4.4.2 Model Selection . . . 147
4.4.3 Underfitting or Overfitting? . . . 148
4.4.4 Polynomial Regression . . . 149
4.5 Weight Decay . . . 154
4.5.1 Norms and Weight Decay . . . 154
4.5.2 High-Dimensional Linear Regression . . . 156
4.5.3 Implementation from Scratch . . . 156
4.5.4 Concise Implementation . . . 158
4.6 Dropout . . . 161
4.6.1 Overfitting Revisited . . . 161
4.6.2 Robustness through Perturbations . . . 162
4.6.3 Dropout in Practice . . . 163
4.6.4 Implementation from Scratch . . . 163
4.6.5 Concise Implementation . . . 166
4.7 Forward Propagation, Backward Propagation, and Computational Graphs . . . 167
4.7.1 Forward Propagation . . . 168
4.7.2 Computational Graph of Forward Propagation . . . 168

4.9.3 Correction of Distribution Shift . . . 181
4.9.4 A Taxonomy of Learning Problems . . . 184
4.9.5 Fairness, Accountability, and Transparency in Machine Learning . . . 186
4.10 Predicting House Prices on Kaggle . . . 187
4.10.1 Downloading and Caching Datasets . . . 187
4.10.2 Kaggle . . . 189
4.10.3 Accessing and Reading the Dataset . . . 190
4.10.4 Data Preprocessing . . . 191
4.10.5 Training . . . 192
4.10.6 K-Fold Cross-Validation . . . 193
4.10.7 Model Selection . . . 194
4.10.8 Submitting Predictions on Kaggle . . . 195

5 Deep Learning Computation 199
5.1 Layers and Blocks . . . 199
5.1.1 A Custom Block . . . 201
5.1.2 The Sequential Block . . . 203
5.1.3 Executing Code in the Forward Propagation Function . . . 204
5.1.4 Compilation . . . 205
5.2 Parameter Management . . . 206
5.2.1 Parameter Access . . . 207
5.2.2 Parameter Initialization . . . 210
5.2.3 Tied Parameters . . . 212
5.3 Deferred Initialization . . . 213
5.3.1 Instantiating a Network . . . 214
5.4 Custom Layers . . . 216
5.4.1 Layers without Parameters . . . 216
5.4.2 Layers with Parameters . . . 217
5.5 File I/O . . . 218
5.5.1 Loading and Saving Tensors . . . 218
5.5.2 Loading and Saving Model Parameters . . . 219
5.6 GPUs . . . 221
5.6.1 Computing Devices . . . 222
5.6.2 Tensors and GPUs . . . 223
5.6.3 Neural Networks and GPUs . . . 225

6 Convolutional Neural Networks 227
6.1 From Fully-Connected Layers to Convolutions . . . 228
6.1.1 Invariance . . . 228
6.1.2 Constraining the MLP . . . 229
6.1.3 Convolutions . . . 231
6.1.4 “Where’s Waldo” Revisited . . . 231
6.2 Convolutions for Images . . . 233

6.3.2 Stride . . . 241
6.4 Multiple Input and Multiple Output Channels . . . 243
6.4.1 Multiple Input Channels . . . 243
6.4.2 Multiple Output Channels . . . 244
6.4.3 1×1 Convolutional Layer . . . 245
6.5 Pooling . . . 247
6.5.1 Maximum Pooling and Average Pooling . . . 247
6.5.2 Padding and Stride . . . 249
6.5.3 Multiple Channels . . . 250
6.6 Convolutional Neural Networks (LeNet) . . . 251
6.6.1 LeNet . . . 252
6.6.2 Training . . . 254

7 Modern Convolutional Neural Networks 257
7.1 Deep Convolutional Neural Networks (AlexNet) . . . 257
7.1.1 Learning Representations . . . 258
7.1.2 AlexNet . . . 261
7.1.3 Reading the Dataset . . . 264
7.1.4 Training . . . 264
7.2 Networks Using Blocks (VGG) . . . 265
7.2.1 VGG Blocks . . . 266
7.2.2 VGG Network . . . 266
7.2.3 Training . . . 268
7.3 Network in Network (NiN) . . . 270
7.3.1 NiN Blocks . . . 270
7.3.2 NiN Model . . . 272
7.3.3 Training . . . 273
7.4 Networks with Parallel Concatenations (GoogLeNet) . . . 274
7.4.1 Inception Blocks . . . 274
7.4.2 GoogLeNet Model . . . 275
7.4.3 Training . . . 278
7.5 Batch Normalization . . . 279
7.5.1 Training Deep Networks . . . 279
7.5.2 Batch Normalization Layers . . . 281
7.5.3 Implementation from Scratch . . . 282
7.5.4 Applying Batch Normalization in LeNet . . . 283
7.5.5 Concise Implementation . . . 285
7.5.6 Controversy . . . 286
7.6 Residual Networks (ResNet) . . . 287
7.6.1 Function Classes . . . 287
7.6.2 Residual Blocks . . . 288
7.6.3 ResNet Model . . . 291
7.6.4 Training . . . 293

8.1 Sequence Models . . . 299
8.1.1 Statistical Tools . . . 300
8.1.2 A Toy Example . . . 303
8.1.3 Predictions . . . 304
8.2 Text Preprocessing . . . 307
8.2.1 Reading the Dataset . . . 307
8.2.2 Tokenization . . . 308
8.2.3 Vocabulary . . . 308
8.2.4 Putting All Things Together . . . 310
8.3 Language Models and the Dataset . . . 310
8.3.1 Estimating a Language Model . . . 311
8.3.2 Markov Models and n-grams . . . 312
8.3.3 Natural Language Statistics . . . 312
8.3.4 Training Data Preparation . . . 315
8.4 Recurrent Neural Networks . . . 318
8.4.1 Recurrent Networks Without Hidden States . . . 319
8.4.2 Recurrent Networks with Hidden States . . . 320
8.4.3 Steps in a Language Model . . . 321
8.4.4 Perplexity . . . 322
8.5 Implementation of Recurrent Neural Networks from Scratch . . . 323
8.5.1 One-hot Encoding . . . 324
8.5.2 Initializing the Model Parameters . . . 324
8.5.3 RNN Model . . . 325
8.5.4 Prediction . . . 326
8.5.5 Gradient Clipping . . . 326
8.5.6 Training . . . 327
8.6 Concise Implementation of Recurrent Neural Networks . . . 331
8.6.1 Defining the Model . . . 331
8.6.2 Training and Predicting . . . 332
8.7 Backpropagation Through Time . . . 334
8.7.1 A Simplified Recurrent Network . . . 334
8.7.2 The Computational Graph . . . 336
8.7.3 BPTT in Detail . . . 337

9 Modern Recurrent Neural Networks 339
9.1 Gated Recurrent Units (GRU) . . . 339
9.1.1 Gating the Hidden State . . . 340
9.1.2 Implementation from Scratch . . . 342
9.1.3 Concise Implementation . . . 345
9.2 Long Short Term Memory (LSTM) . . . 346
9.2.1 Gated Memory Cells . . . 347
9.2.2 Implementation from Scratch . . . 350
9.2.3 Concise Implementation . . . 352

9.5.1 Reading and Preprocessing the Dataset . . . 363
9.5.2 Tokenization . . . 364
9.5.3 Vocabulary . . . 365
9.5.4 Loading the Dataset . . . 365
9.5.5 Putting All Things Together . . . 366
9.6 Encoder-Decoder Architecture . . . 367
9.6.1 Encoder . . . 367
9.6.2 Decoder . . . 367
9.6.3 Model . . . 368
9.7 Sequence to Sequence . . . 369
9.7.1 Encoder . . . 370
9.7.2 Decoder . . . 371
9.7.3 The Loss Function . . . 372
9.7.4 Training . . . 373
9.7.5 Predicting . . . 375
9.8 Beam Search . . . 376
9.8.1 Greedy Search . . . 376
9.8.2 Exhaustive Search . . . 378
9.8.3 Beam Search . . . 378

10 Attention Mechanisms **381**
10.1 Attention Mechanisms . . . 381
10.1.1 Dot Product Attention . . . 384
10.1.2 MLP Attention . . . 385
10.2 Sequence to Sequence with Attention Mechanisms . . . 386
10.2.1 Decoder . . . 387
10.2.2 Training . . . 389
10.3 Transformer . . . 390
10.3.1 Multi-Head Attention . . . 392
10.3.2 Position-wise Feed-Forward Networks . . . 394
10.3.3 Add and Norm . . . 395
10.3.4 Positional Encoding . . . 396
10.3.5 Encoder . . . 398
10.3.6 Decoder . . . 399
10.3.7 Training . . . 401

11 Optimization Algorithms **405**
11.1 Optimization and Deep Learning . . . 405
11.1.1 Optimization and Estimation . . . 406
11.1.2 Optimization Challenges in Deep Learning . . . 407
11.2 Convexity . . . 411
11.2.1 Basics . . . 411
11.2.2 Properties . . . 414

11.4.3 Convergence Analysis for Convex Objectives . . . 433
11.4.4 Stochastic Gradients and Finite Samples . . . 434
11.5 Minibatch Stochastic Gradient Descent . . . 436
11.5.1 Vectorization and Caches . . . 436
11.5.2 Minibatches . . . 438
11.5.3 Reading the Dataset . . . 439
11.5.4 Implementation from Scratch . . . 439
11.5.5 Concise Implementation . . . 443
11.6 Momentum . . . 445
11.6.1 Basics . . . 445
11.6.2 Practical Experiments . . . 449
11.6.3 Theoretical Analysis . . . 452
11.7 Adagrad . . . 454
11.7.1 Sparse Features and Learning Rates . . . 455
11.7.2 Preconditioning . . . 455
11.7.3 The Algorithm . . . 457
11.7.4 Implementation from Scratch . . . 458
11.7.5 Concise Implementation . . . 459
11.8 RMSProp . . . 461
11.8.1 The Algorithm . . . 461
11.8.2 Implementation from Scratch . . . 462
11.8.3 Concise Implementation . . . 464
11.9 Adadelta . . . 465
11.9.1 The Algorithm . . . 465
11.9.2 Implementation . . . 465
11.10 Adam . . . 467
11.10.1 The Algorithm . . . 468
11.10.2 Implementation . . . 469
11.10.3 Yogi . . . 470
11.11 Learning Rate Scheduling . . . 472
11.11.1 Toy Problem . . . 472
11.11.2 Schedulers . . . 474
11.11.3 Policies . . . 476

12 Computational Performance **483**
12.1 Compilers and Interpreters . . . 483
12.1.1 Symbolic Programming . . . 484
12.1.2 Hybrid Programming . . . 485
12.1.3 HybridSequential . . . 486
12.2 Asynchronous Computation . . . 490
12.2.1 Asynchrony via Backend . . . 490
12.2.2 Barriers and Blockers . . . 492
12.2.3 Improving Computation . . . 493

12.4.4 CPUs . . . 504
12.4.5 GPUs and other Accelerators . . . 507
12.4.6 Networks and Buses . . . 510
12.4.7 More Latency Numbers . . . 511
12.5 Training on Multiple GPUs . . . 513
12.5.1 Splitting the Problem . . . 514
12.5.2 Data Parallelism . . . 515
12.5.3 A Toy Network . . . 517
12.5.4 Data Synchronization . . . 518
12.5.5 Distributing Data . . . 519
12.5.6 Training . . . 520
12.5.7 Experiment . . . 521
12.6 Concise Implementation for Multiple GPUs . . . 522
12.6.1 A Toy Network . . . 523
12.6.2 Parameter Initialization and Logistics . . . 523
12.6.3 Training . . . 525
12.6.4 Experiments . . . 526
12.7 Parameter Servers . . . 527
12.7.1 Data Parallel Training . . . 528
12.7.2 Ring Synchronization . . . 530
12.7.3 Multi-Machine Training . . . 533
12.7.4 (key,value) Stores . . . 535

13 Computer Vision **537**
13.1 Image Augmentation . . . 537
13.1.1 Common Image Augmentation Method . . . 538
13.1.2 Using an Image Augmentation Training Model . . . 542
13.2 Fine-Tuning . . . 546
13.2.1 Hot Dog Recognition . . . 547
13.3 Object Detection and Bounding Boxes . . . 552
13.3.1 Bounding Box . . . 553
13.4 Anchor Boxes . . . 554
13.4.1 Generating Multiple Anchor Boxes . . . 555
13.4.2 Intersection over Union . . . 557
13.4.3 Labeling Training Set Anchor Boxes . . . 557
13.4.4 Bounding Boxes for Prediction . . . 561
13.5 Multiscale Object Detection . . . 564
13.6 The Object Detection Dataset . . . 567
13.6.1 Downloading the Dataset . . . 567
13.6.2 Reading the Dataset . . . 568
13.6.3 Demonstration . . . 569
13.7 Single Shot Multibox Detection (SSD) . . . 570
13.7.1 Model . . . 570

13.9.1 Image Segmentation and Instance Segmentation . . . 587
13.9.2 The Pascal VOC2012 Semantic Segmentation Dataset . . . 588
13.10 Transposed Convolution . . . 593
13.10.1 Basic 2D Transposed Convolution . . . 593
13.10.2 Padding, Strides, and Channels . . . 594
13.10.3 Analogy to Matrix Transposition . . . 595
13.11 Fully Convolutional Networks (FCN) . . . 596
13.11.1 Constructing a Model . . . 597
13.11.2 Initializing the Transposed Convolution Layer . . . 599
13.11.3 Reading the Dataset . . . 600
13.11.4 Training . . . 600
13.11.5 Prediction . . . 601
13.12 Neural Style Transfer . . . 603
13.12.1 Technique . . . 604
13.12.2 Reading the Content and Style Images . . . 605
13.12.3 Preprocessing and Postprocessing . . . 606
13.12.4 Extracting Features . . . 606
13.12.5 Defining the Loss Function . . . 607
13.12.6 Creating and Initializing the Composite Image . . . 609
13.12.7 Training . . . 610
13.13 Image Classification (CIFAR-10) on Kaggle . . . 613
13.13.1 Obtaining and Organizing the Dataset . . . 614
13.13.2 Image Augmentation . . . 616
13.13.3 Reading the Dataset . . . 617
13.13.4 Defining the Model . . . 618
13.13.5 Defining the Training Functions . . . 619
13.13.6 Training and Validating the Model . . . 620
13.13.7 Classifying the Testing Set and Submitting Results on Kaggle . . . 620
13.14 Dog Breed Identification (ImageNet Dogs) on Kaggle . . . 622
13.14.1 Obtaining and Organizing the Dataset . . . 623
13.14.2 Image Augmentation . . . 624
13.14.3 Reading the Dataset . . . 625
13.14.4 Defining the Model . . . 625
13.14.5 Defining the Training Functions . . . 626
13.14.6 Training and Validating the Model . . . 627
13.14.7 Classifying the Testing Set and Submitting Results on Kaggle . . . 627

14 Natural Language Processing: Pretraining **631**
14.1 Word Embedding (word2vec) . . . 632
14.1.1 Why Not Use One-hot Vectors? . . . 632
14.1.2 The Skip-Gram Model . . . 632
14.1.3 The Continuous Bag of Words (CBOW) Model . . . 634
14.2 Approximate Training . . . 636

14.4.1 The Skip-Gram Model . . . 647
14.4.2 Training . . . 648
14.4.3 Applying the Word Embedding Model . . . 650
14.5 Word Embedding with Global Vectors (GloVe) . . . 651
14.5.1 The GloVe Model . . . 652
14.5.2 Understanding GloVe from Conditional Probability Ratios . . . 653
14.6 Subword Embedding . . . 654
14.6.1 fastText . . . 654
14.6.2 Byte Pair Encoding . . . 655
14.7 Finding Synonyms and Analogies . . . 658
14.7.1 Using Pretrained Word Vectors . . . 659
14.7.2 Applying Pretrained Word Vectors . . . 660
14.8 Bidirectional Encoder Representations from Transformers (BERT) . . . 663
14.8.1 From Context-Independent to Context-Sensitive . . . 663
14.8.2 From Task-Specific to Task-Agnostic . . . 663
14.8.3 BERT: Combining the Best of Both Worlds . . . 664
14.8.4 Input Representation . . . 665
14.8.5 Pretraining Tasks . . . 667
14.8.6 Putting All Things Together . . . 670
14.9 The Dataset for Pretraining BERT . . . 671
14.9.1 Defining Helper Functions for Pretraining Tasks . . . 672
14.9.2 Transforming Text into the Pretraining Dataset . . . 674
14.10 Pretraining BERT . . . 677
14.10.1 Pretraining BERT . . . 677
14.10.2 Representing Text with BERT . . . 679

15 Natural Language Processing: Applications 683
15.1 Sentiment Analysis and the Dataset . . . 684
15.1.1 The Sentiment Analysis Dataset . . . 684
15.1.2 Putting All Things Together . . . 687
15.2 Sentiment Analysis: Using Recurrent Neural Networks . . . 687
15.2.1 Using a Recurrent Neural Network Model . . . 688
15.3 Sentiment Analysis: Using Convolutional Neural Networks . . . 691
15.3.1 One-Dimensional Convolutional Layer . . . 692
15.3.2 Max-Over-Time Pooling Layer . . . 694
15.3.3 The TextCNN Model . . . 695
15.4 Natural Language Inference and the Dataset . . . 698
15.4.1 Natural Language Inference . . . 698
15.4.2 The Stanford Natural Language Inference (SNLI) Dataset . . . 699
15.5 Natural Language Inference: Using Attention . . . 703
15.5.1 The Model . . . 703
15.5.2 Training and Evaluating the Model . . . 707
15.6 Fine-Tuning BERT for Sequence-Level and Token-Level Applications . . . 710

16 Recommender Systems **721**
16.1 Overview of Recommender Systems . . . 721
16.1.1 Collaborative Filtering . . . 722
16.1.2 Explicit Feedback and Implicit Feedback . . . 723
16.1.3 Recommendation Tasks . . . 723
16.2 The MovieLens Dataset . . . 724
16.2.1 Getting the Data . . . 724
16.2.2 Statistics of the Dataset . . . 725
16.2.3 Splitting the dataset . . . 726
16.2.4 Loading the data . . . 727
16.3 Matrix Factorization . . . 728
16.3.1 The Matrix Factorization Model . . . 729
16.3.2 Model Implementation . . . 730
16.3.3 Evaluation Measures . . . 730
16.3.4 Training and Evaluating the Model . . . 731
16.4 AutoRec: Rating Prediction with Autoencoders . . . 733
16.4.1 Model . . . 733
16.4.2 Implementing the Model . . . 734
16.4.3 Reimplementing the Evaluator . . . 734
16.4.4 Training and Evaluating the Model . . . 735
16.5 Personalized Ranking for Recommender Systems . . . 736
16.5.1 Bayesian Personalized Ranking Loss and its Implementation . . . 737
16.5.2 Hinge Loss and its Implementation . . . 738
16.6 Neural Collaborative Filtering for Personalized Ranking . . . 739
16.6.1 The NeuMF model . . . 740
16.6.2 Model Implementation . . . 741
16.6.3 Customized Dataset with Negative Sampling . . . 742
16.6.4 Evaluator . . . 742
16.6.5 Training and Evaluating the Model . . . 744
16.7 Sequence-Aware Recommender Systems . . . 746
16.7.1 Model Architectures . . . 746
16.7.2 Model Implementation . . . 748
16.7.3 Sequential Dataset with Negative Sampling . . . 749
16.7.4 Load the MovieLens 100K dataset . . . 750
16.7.5 Train the Model . . . 751
16.8 Feature-Rich Recommender Systems . . . 752
16.8.1 An Online Advertising Dataset . . . 753
16.8.2 Dataset Wrapper . . . 753
16.9 Factorization Machines . . . 755
16.9.1 2-Way Factorization Machines . . . 755
16.9.2 An Efficient Optimization Criterion . . . 756
16.9.3 Model Implementation . . . 756
16.9.4 Load the Advertising Dataset . . . 757

17.1.1 Generate some "real" data . . . 765
17.1.2 Generator . . . 766
17.1.3 Discriminator . . . 766
17.1.4 Training . . . 766
17.2 Deep Convolutional Generative Adversarial Networks . . . 769
17.2.1 The Pokemon Dataset . . . 769
17.2.2 The Generator . . . 770
17.2.3 Discriminator . . . 772
17.2.4 Training . . . 773

18 Appendix: Mathematics for Deep Learning 777
18.1 Geometry and Linear Algebraic Operations . . . 778
18.1.1 Geometry of Vectors . . . 778
18.1.2 Dot Products and Angles . . . 780
18.1.3 Hyperplanes . . . 782
18.1.4 Geometry of Linear Transformations . . . 785
18.1.5 Linear Dependence . . . 787
18.1.6 Rank . . . 787
18.1.7 Invertibility . . . 788
18.1.8 Determinant . . . 789
18.1.9 Tensors and Common Linear Algebra Operations . . . 790
18.2 Eigendecompositions . . . 794
18.2.1 Finding Eigenvalues . . . 794
18.2.2 Decomposing Matrices . . . 795
18.2.3 Operations on Eigendecompositions . . . 795
18.2.4 Eigendecompositions of Symmetric Matrices . . . 796
18.2.5 Gershgorin Circle Theorem . . . 796
18.2.6 A Useful Application: The Growth of Iterated Maps . . . 797
18.2.7 Conclusions . . . 802
18.3 Single Variable Calculus . . . 803
18.3.1 Differential Calculus . . . 803
18.3.2 Rules of Calculus . . . 806
18.4 Multivariable Calculus . . . 813
18.4.1 Higher-Dimensional Differentiation . . . 814
18.4.2 Geometry of Gradients and Gradient Descent . . . 815
18.4.3 A Note on Mathematical Optimization . . . 816
18.4.4 Multivariate Chain Rule . . . 817
18.4.5 The Backpropagation Algorithm . . . 819
18.4.6 Hessians . . . 822
18.4.7 A Little Matrix Calculus . . . 824
18.5 Integral Calculus . . . 829
18.5.1 Geometric Interpretation . . . 829
18.5.2 The Fundamental Theorem of Calculus . . . 831

18.7.2 Numerical Optimization and the Negative Log-Likelihood . . . 857
18.7.3 Maximum Likelihood for Continuous Variables . . . 859
18.8 Distributions . . . 861
18.8.1 Bernoulli . . . 861
18.8.2 Discrete Uniform . . . 863
18.8.3 Continuous Uniform . . . 864
18.8.4 Binomial . . . 866
18.8.5 Poisson . . . 868
18.8.6 Gaussian . . . 871
18.8.7 Exponential Family . . . 874
18.9 Naive Bayes . . . 875
18.9.1 Optical Character Recognition . . . 876
18.9.2 The Probabilistic Model for Classification . . . 877
18.9.3 The Naive Bayes Classifier . . . 877
18.9.4 Training . . . 878
18.10 Statistics . . . 882
18.10.1 Evaluating and Comparing Estimators . . . 882
18.10.2 Conducting Hypothesis Tests . . . 886
18.10.3 Constructing Confidence Intervals . . . 890
18.11 Information Theory . . . 893
18.11.1 Information . . . 893
18.11.2 Entropy . . . 895
18.11.3 Mutual Information . . . 897
18.11.4 Kullback–Leibler Divergence . . . 901
18.11.5 Cross Entropy . . . 903

19 Appendix: Tools for Deep Learning 907
19.1 Using Jupyter . . . 907
19.1.1 Editing and Running the Code Locally . . . 907
19.1.2 Advanced Options . . . 911
19.2 Using Amazon SageMaker . . . 912
19.2.1 Registering and Logging In . . . 912
19.2.2 Creating a SageMaker Instance . . . 913
19.2.3 Running and Stopping an Instance . . . 914
19.2.4 Updating Notebooks . . . 915
19.3 Using AWS EC2 Instances . . . 916
19.3.1 Creating and Running an EC2 Instance . . . 916
19.3.2 Installing CUDA . . . 921
19.3.3 Installing MXNet and Downloading the D2L Notebooks . . . 922
19.3.4 Running Jupyter . . . 923
19.3.5 Closing Unused Instances . . . 924
19.4 Using Google Colab . . . 924
19.5 Selecting Servers and GPUs . . . 925

Bibliography 955

Python Module Index 965

Index 967

Preface

Just a few years ago, there were no legions of deep learning scientists developing intelligent products and services at major companies and startups. When the youngest among us (the authors) entered the field, machine learning did not command headlines in daily newspapers. Our parents had no idea what machine learning was, let alone why we might prefer it to a career in medicine or law. Machine learning was a forward-looking academic discipline with a narrow set of real-world applications. And those applications, e.g., speech recognition and computer vision, required so much domain knowledge that they were often regarded as separate areas entirely for which machine learning was one small component. Neural networks then, the antecedents of the deep learning models that we focus on in this book, were regarded as outmoded tools.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and statistical modeling. With these advances in hand, we can now build cars that drive themselves with more autonomy than ever before (and less autonomy than some companies might have you believe), smart reply systems that automatically draft the most mundane emails, helping people dig out from oppressively large inboxes, and software agents that dominate the world's best humans at board games like Go, a feat once thought to be decades away. Already, these tools exert ever-wider impacts on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics to biology.

About This Book

This book represents our attempt to make deep learning approachable, teaching you the *concepts*, the *context*, and the *code*.

One Medium Combining Code, Math, and HTML

For any computing technology to reach its full impact, it must be well-understood, well-documented, and supported by mature, well-maintained tools. The key ideas should be clearly

Testing the potential of deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding (i) the motivations for casting a problem in a particular way; (ii) the mathematics of a given modeling approach; (iii) the optimization algorithms for fitting the models to data; and (iv) and the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching both the critical thinking skills required to formulate problems, the mathematics to solve them, and the software tools to implement those solutions all in one place presents formidable challenges. Our goal in this book is to present a unified resource to bring would-be practitioners up to speed.

At the time we started this book project, there were no resources that simultaneously (i) were up to date; (ii) covered the full breadth of modern machine learning with substantial technical depth; and (iii) interleaved exposition of the quality one expects from an engaging textbook with the clean runnable code that one expects to find in hands-on tutorials. We found plenty of code examples for how to use a given deep learning framework (e.g., how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g., code snippets for LeNet, AlexNet, ResNets, etc) scattered across various blog posts and GitHub repositories. However, these examples typically focused on *how* to implement a given approach, but left out the discussion of *why* certain algorithmic decisions are made. While some interactive resources have popped up sporadically to address a particular topic, e.g., the engaging blog posts published on the website Distill[3], or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several textbooks have emerged, most notably (Goodfellow et al., 2016), which offers a comprehensive survey of the concepts behind deep learning, these resources do not marry the descriptions to realizations of the concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.

We set out to create a resource that could (i) be freely available for everyone; (ii) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (iii) include runnable code, showing readers *how* to solve problems in practice; (iv) allow for rapid updates, both by us and also by the community at large; and (v) be complemented by a forum[4] for interactive discussion of technical details and to answer questions.

These goals were often in conflict. Equations, theorems, and citations are best managed and laid out in LaTeX. Code is best described in Python. And webpages are native in HTML and JavaScript. Furthermore, we want the content to be accessible both as executable code, as a physical book, as a downloadable PDF, and on the Internet as a website. At present there exist no tools and no workflow perfectly suited to these demands, so we had to assemble our own. We describe our approach in detail in Section 19.6. We settled on GitHub to share the source and to allow for edits, Jupyter notebooks for mixing code, equations and text, Sphinx as a rendering engine to generate multiple outputs, and Discourse for the forum. While our system is not yet perfect, these choices provide a good compromise among the competing concerns. We believe that this might be the first book published using such an integrated workflow.

[3] http://distill.pub

Learning by Doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook (Bishop, 2006), teaches each topic so thoroughly, that getting to the chapter on linear regression requires a non-trivial amount of work. While experts love this book precisely for its thoroughness, for beginners, this property limits its usefulness as an introductory text.

In this book, we will teach most concepts *just in time*. In other words, you will learn concepts at the very moment that they are needed to accomplish some practical end. While we take some time at the outset to teach fundamental preliminaries, like linear algebra and probability, we want you to taste the satisfaction of training your first model before worrying about more esoteric probability distributions.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter introduces both a reasonable number of new concepts and provides single self-contained working examples—using real datasets. This presents an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. On the other hand, there is a big advantage to adhering to a policy of *1 working example, 1 notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook and start modifying it.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some curatorial decisions.

This book will teach deep learning concepts from scratch. Sometimes, we want to delve into fine details about the models that would typically be hidden from the user by deep learning frameworks' advanced abstractions. This comes up especially in the basic tutorials, where we want you to understand everything that happens in a given layer or optimizer. In these cases, we will often present two versions of the example: one where we implement everything from scratch, relying only on the NumPy interface and automatic differentiation, and another, more practical example, where we write succinct code using Gluon. Once we have taught you how some component works, we can just use the Gluon version in subsequent tutorials.

Content and Structure

The book can be roughly divided into three parts, which are presented by different colors in Fig. 1:

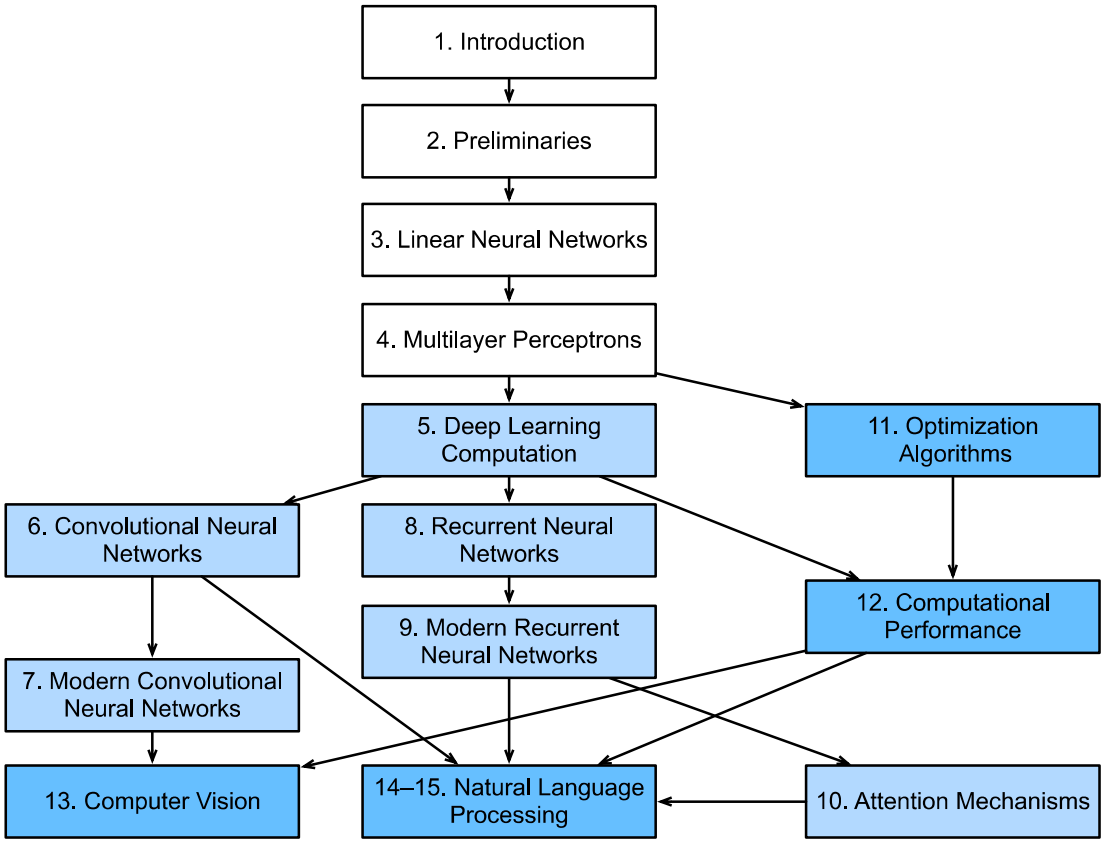


Fig. 1: Book structure

- The first part covers basics and preliminaries. Chapter 1 offers an introduction to deep learning. Then, in Chapter 2, we quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to store and manipulate data, and how to apply various numerical operations based on basic concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 4 cover the most basic concepts and techniques of deep learning, such as linear regression, multilayer perceptrons and regularization.
- The next five chapters focus on modern deep learning techniques. Chapter 5 describes the various key components of deep learning calculations and lays the groundwork for us to subsequently implement more complex models. Next, in Chapter 6 and Chapter 7, we introduce convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Subsequently, in Chapter 8 and Chapter 9, we introduce

chapter, Chapter 12 examines several key factors that influence the computational performance of your deep learning code. In Chapter 13, we illustrate major applications of deep learning in computer vision. In Chapter 14 and Chapter 15, we show how to pretrain language representation models and apply them to natural language processing tasks.

Code

Most sections of this book feature executable code because of our belief in the importance of an interactive learning experience in deep learning. At present, certain intuitions can only be developed through trial and error, tweaking the code in small ways and observing the results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. Unfortunately, at present, such elegant theories elude us. Despite our best attempts, formal explanations for various techniques are still lacking, both because the mathematics to characterize these models can be so difficult and also because serious inquiry on these topics has only just recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, future editions of this book will be able to provide insights in places the present edition cannot.

At times, to avoid unnecessary repetition, we encapsulate the frequently-imported and referred-to functions, classes, etc. in this book in the `d2l` package. For any block such as a function, a class, or multiple imports to be saved in the package, we will mark it with `#@save`. We offer a detailed overview of these functions and classes in Section 19.7. The `d2l` package is light-weight and only requires the following packages and modules as dependencies:

```
#@save
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
import os
import pandas as pd
import random
import re
import shutil
import sys
import tarfile
import time
import requests
import zipfile
import hashlib
d2l = sys.modules[__name__]
```

Most of the code in this book is based on Apache MXNet. MXNet is an open-source framework for deep learning and the preferred choice of AWS (Amazon Web Services), as well as many colleges and companies. All of the code in this book has passed tests under the newest MXNet version

```
#@save
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
```

Target Audience

This book is for students (undergraduate or graduate), engineers, and researchers, who seek a solid grasp of the practical techniques of deep learning. Because we explain every concept from scratch, no previous background in deep learning or machine learning is required. Fully explaining the methods of deep learning requires some mathematics and programming, but we will only assume that you come in with some basics, including (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, in the Appendix, we provide a refresher on most of the mathematics covered in this book. Most of the time, we will prioritize intuition and ideas over mathematical rigor. There are many terrific books which can lead the interested reader further. For instance, Linear Analysis by Bela Bollobas (Bollobas, 1999) covers linear algebra and functional analysis in great depth. All of Statistics (Wasserman, 2013) is a terrific guide to statistics. And if you have not used Python before, you may want to peruse this Python tutorial[5].

Forum

Associated with this book, we have launched a discussion forum, located at discuss.d2l.ai[6]. When you have questions on any section of the book, you can find the associated discussion page link at the end of each chapter.

Acknowledgments

We are indebted to the hundreds of contributors for both the English and the Chinese drafts. They helped improve the content and offered valuable feedback. Specifically, we thank every contributor of this English draft for making it better for everyone. Their GitHub IDs or names are (in no particular order): alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther

sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, cuongvng, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, nguyenhoa93, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, yoderj, mathresearch.

We thank Amazon Web Services, especially Swami Sivasubramanian, Raju Gulabani, Charlie Bell, and Andrew Jassy for their generous support in writing this book. Without the available time, resources, discussions with colleagues, and continuous encouragement this book would not have happened.

Summary

- Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, including computer vision, natural language processing, automatic speech recognition.
- To successfully apply deep learning, you must understand how to cast a problem, the mathematics of modeling, the algorithms for fitting your models to data, and the engineering techniques to implement it all.
- This book presents a comprehensive resource, including prose, figures, mathematics, and code, all in one place.
- To answer questions related to this book, visit our forum at https://discuss.d2l.ai/.
- All notebooks are available for download on GitHub.

Exercises

1. Register an account on the discussion forum of this book discuss.d2l.ai[7].
2. Install Python on your computer.
3. Follow the links at the bottom of the section to the forum, where you will be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.

Discussions[8]

Installation

In order to get you up and running for hands-on learning experience, we need to set you up with an environment for running Python, Jupyter notebooks, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda

The simplest way to get going will be to install Miniconda[9]. The Python 3.x version is required. You can skip the following steps if conda has already been installed. Download the corresponding Miniconda sh file from the website and then execute the installation from the command line using `sh <FILENAME> -b`. For macOS users:

```
# The file name is subject to changes
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

For Linux users:

```
# The file name is subject to changes
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Next, initialize the shell so we can run `conda` directly.

```
~/miniconda3/bin/conda init
```

Now close and re-open your current shell. You should be able to create a new environment as following:

```
conda create --name d2l -y
```

[9] https://conda.io/en/latest/miniconda.html

Downloading the D2L Notebooks

Next, we need to download the code of this book. You can click the "All Notebooks" tab on the top of any HTML page to download and unzip the code. Alternatively, if you have `unzip` (otherwise run `sudo apt install unzip`) available:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Now we will want to activate the `d2l` environment and install `pip`. Enter `y` for the queries that follow this command.

```
conda activate d2l
conda install python=3.7 pip -y
```

Installing the Framework and the `d2l` Package

Before installing the deep learning framework, please first check whether or not you have proper GPUs on your machine (the GPUs that power the display on a standard laptop do not count for our purposes). If you are installing on a GPU server, proceed to *GPU Support* (page 11) for instructions to install a GPU-supported version.

Otherwise, you can install the CPU version. That will be more than enough horsepower to get you through the first few chapters but you will want to access GPUs before running larger models.

```
pip install mxnet==1.6.0
```

We also install the `d2l` package that encapsulates frequently used functions and classes in this book.

```
# -U: Upgrade all packages to the newest available version
pip install -U d2l
```

Once they are installed, we now open the Jupyter notebook by running:

```
jupyter notebook
```

At this point, you can open http://localhost:8888 (it usually opens automatically) in your Web browser. Then we can run the code for each section of the book. Please always execute `conda activate d2l` to activate the runtime environment before running the code of the book or updating

GPU Support

By default, the deep learning framework is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA[10], then you should install a GPU-enabled version. If you have installed the CPU-only version, you may need to remove it first by running:

```
pip uninstall mxnet
```

Then we need to find the CUDA version you installed. You may check it through `nvcc --version` or `cat /usr/local/cuda/version.txt`. Assume that you have installed CUDA 10.1, then you can install with the following command:

```
# For Windows users
pip install mxnet-cu101==1.6.0b20190926

# For Linux and macOS users
pip install mxnet-cu101==1.6.0
```

You may change the last digits according to your CUDA version, e.g., `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0.

Exercises

1. Download the code for the book and install the runtime environment.

Discussions[11]

Notation

The notation used throughout this book is summarized below.

Numbers

- x: A scalar
- $\mathbf{x}$: A vector
- $\mathbf{X}$: A matrix
- X: A tensor
- $\mathbf{I}$: An identity matrix
- x_i, $[\mathbf{x}]_i$: The ith element of vector $\mathbf{x}$
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: The element of matrix $\mathbf{X}$ at row i and column j

Set Theory

- $\mathcal{X}$: A set
- $\mathbb{Z}$: The set of integers
- $\mathbb{R}$: The set of real numbers
- $\mathbb{R}^n$: The set of n-dimensional vectors of real numbers
- $\mathbb{R}^{a \times b}$: The set of matrices of real numbers with a rows and b columns
- $|\mathcal{X}|$: Cardinality (number of elements) of set $\mathcal{X}$
- $\mathcal{A} \cup \mathcal{B}$: Union of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A} \cap \mathcal{B}$: Intersection of sets $\mathcal{A}$ and $\mathcal{B}$

Functions and Operators

- $f(\cdot)$: A function
- $\log(\cdot)$: The natural logarithm
- $\exp(\cdot)$: The exponential function
- $\mathbf{1}_{\mathcal{X}}$: The indicator function
- $(\cdot)^\top$: Transpose of a vector or a matrix
- $\mathbf{X}^{-1}$: Inverse of matrix $\mathbf{X}$
- $\odot$: Hadamard (elementwise) product
- $[\cdot, \cdot]$: Concatenation
- $|\mathcal{X}|$: Cardinality of set $\mathcal{X}$
- $\|\cdot\|_p$: L_p norm
- $\|\cdot\|$: L_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: Dot product of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\sum$: Series addition
- $\prod$: Series multiplication
- $\stackrel{\text{def}}{=}$: Definition

Calculus

- $\frac{dy}{dx}$: Derivative of y with respect to x
- $\frac{\partial y}{\partial x}$: Partial derivative of y with respect to x
- $\nabla_{\mathbf{x}} y$: Gradient of y with respect to $\mathbf{x}$
- $\int_a^b f(x)\, dx$: Definite integral of f from a to b with respect to x
- $\int f(x)\, dx$: Indefinite integral of f with respect to x

Probability and Information Theory

- $P(\cdot)$: Probability distribution

- $X \perp Y \mid Z$: Random variables X and Y are conditionally independent given random variable Z
- $\mathrm{Var}(X)$: Variance of random variable X
- σ_X: Standard deviation of random variable X
- $\mathrm{Cov}(X, Y)$: Covariance of random variables X and Y
- $\rho(X, Y)$: Correlation of random variables X and Y
- $H(X)$: Entropy of random variable X
- $D_{\mathrm{KL}}(P \| Q)$: KL-divergence of distributions P and Q

Complexity

- $\mathcal{O}$: Big O notation

Discussions[12]

1 | Introduction

Until recently, nearly every computer program that we interact with daily was coded by software developers from first principles. Say that we wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder the problem, we would come up with the broad strokes of a working solution that might probably look something like this: (i) users interact with the application through an interface running in a web browser or mobile application; (ii) our application interacts with a commercial-grade database engine to keep track of each user's state and maintain records of historical transactions; and (iii) at the heart of our application, the *business logic* (you might say, the *brains*) of our application spells out in methodical detail the appropriate action that our program should take in every conceivable circumstance.

To build the *brains* of our application, we'd have to step through every possible corner case that we anticipate encountering, devising appropriate rules. Each time a customer clicks to add an item to their shopping cart, we add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. While few developers ever get it completely right the first time (it might take some test runs to work out the kinks), for the most part, we could write such a program from first principles and confidently launch it *before ever seeing a real customer*. Our ability to design automated systems from first principles that drive functioning products and systems, often in novel situations, is a remarkable cognitive feat. And when you are able to devise solutions that work 100% of the time, *you should not be using machine learning*.

Fortunately for the growing community of machine learning (ML) scientists, many tasks that we would like to automate do not bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling one of the following problems:

- Write a program that predicts tomorrow's weather given geographic information, satellite images, and a trailing window of past weather.
- Write a program that takes in a question, expressed in free-form text, and answers it correctly.
- Write a program that given an image can identify all the people it contains, drawing outlines around each.

effortlessly). ML is the study of powerful techniques that can *learn* from *experience*. As an ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, its performance improves. Contrast this with our deterministic e-commerce platform, which performs according to the same business logic, no matter how much experience accrues, until the developers themselves *learn* and decide that it is time to update the software. In this book, we will teach you the fundamentals of machine learning, and focus in particular on deep learning, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, and genomics.

1.1 A Motivating Example

Before we could begin writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Using an iPhone, Alex called out "Hey Siri", awakening the phone's voice recognition system. Then Mu commanded "directions to Blue Bottle coffee shop". The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While we fabricated this story for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smart phone can engage several machine learning models.

Imagine just writing a program to respond to a *wake word* like "Alexa", "Okay, Google" or "Siri". Try coding it up in a room by yourself with nothing but a computer and a code editor, as illustrated in Fig. 1.1.1. How would you write such a program from first principles? Think about it... the problem is hard. Every second, the microphone will collect roughly 44,000 samples. Each sample is a measurement of the amplitude of the sound wave. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, no}` on whether the snippet contains the wake word? If you are stuck, do not worry. We do not know how to write such a program from scratch either. That is why we use ML.

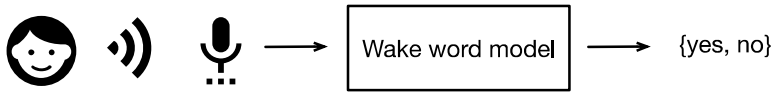


Fig. 1.1.1: Identify an awake word.

Here's the trick. Often, even when we do not know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you do not know *how to program a computer* to recognize the word "Alexa", you yourself *are able* to recognize the word "Alexa". Armed with this ability, we can collect a huge *dataset* containing examples of audio and label those that *do* and that *do not* contain the wake word.

models. And the *meta-program* that uses our dataset to choose the parameters is called a *learning algorithm*.

Before we can go ahead and engage the learning algorithm, we have to define the problem precisely, pinning down the exact nature of the inputs and outputs, and choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, and it generates a selection among `{yes, no}` as *output*. If all goes according to plan the model's guesses will typically be correct as to whether (or not) the snippet contains the wake word.

If we choose the right family of models, then there should exist one setting of the knobs such that the model fires `yes` every time it hears the word "Alexa". Because the exact choice of the wake word is arbitrary, we will probably need a model family sufficiently rich that, via another setting of the knobs, it could fire `yes` only upon hearing the word "Apricot". We expect that the same model family should be suitable for *"Alexa" recognition* and *"Apricot" recognition* because they seem, intuitively, to be similar tasks. However, we might need a different family of models entirely if we want to deal with fundamentally different inputs or outputs, say if we wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set all of the knobs randomly, it is not likely that our model will recognize "Alexa", "Apricot", or any other English word. In deep learning, the *learning* is the process by which we discover the right setting of the knobs coercing the desired behavior from our model.

As shown in Fig. 1.1.2, the training process usually looks like this:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your labeled data (e.g., audio snippets and corresponding `{yes, no}` labels)
3. Tweak the knobs so the model sucks less with respect to those examples
4. Repeat until the model is awesome.

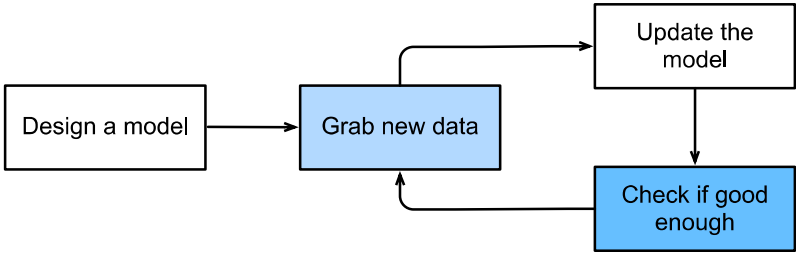
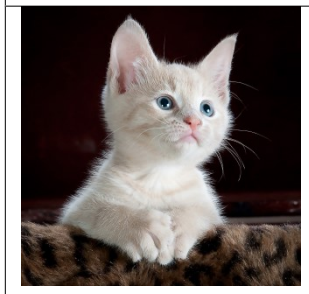
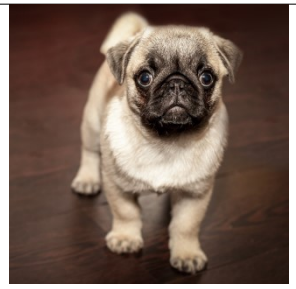
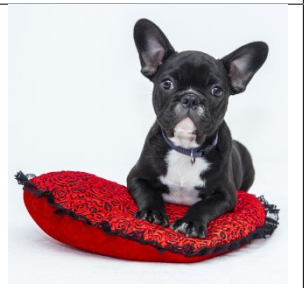


Fig. 1.1.2: A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, *if we present it with a large labeled dataset*. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. We can "program" a cat detector by providing our machine learning system with many examples of

cat	cat	dog	dog
			

This way the detector will eventually learn to emit a very large positive number if it is a cat, a very large negative number if it is a dog, and something closer to zero if it is not sure, and this barely scratches the surface of what ML can do.

Deep learning is just one among many popular methods for solving machine learning problems. Thus far, we have only talked about machine learning broadly and not deep learning. To see why deep learning is important, we should pause for a moment to highlight a couple of crucial points.

First, the problems that we have discussed thus far—learning from the raw audio signal, the raw pixel values of images, or mapping between sentences of arbitrary lengths and their counterparts in foreign languages—are problems where deep learning excels and where traditional ML methods faltered. Deep models are *deep* in precisely the sense that they learn many *layers* of computation. It turns out that these many-layered (or hierarchical) models are capable of addressing low-level perceptual data in a way that previous tools could not. In bygone days, the crucial part of applying ML to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to *shallow* models. One key advantage of deep learning is that it replaces not only the *shallow* models at the end of traditional learning pipelines, but also the labor-intensive process of feature engineering. Second, by replacing much of the *domain-specific preprocessing*, deep learning has eliminated many of the boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, and other application areas, offering a unified set of tools for tackling diverse problems.

1.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels, and we gave a hand-wavy sense of how we might *train* a model to approximate a mapping from snippets to classifications. This sort of problem, where we try to predict a designated unknown *label* given known *inputs*, given a dataset consisting of examples, for which the labels are known is called *supervised learning*, and it is just one among many *kinds* of machine learning problems. In the next section, we will take a deep dive into the different ML problems. First, we'd like to shed more light on some core components that will follow us around, no matter what kind of ML

1.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering what precisely constitutes data, but for now, we will err on the practical side and focus on the key properties to be concerned with. Generally, we are concerned with a collection of *examples*. In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* typically consists of a collection of numerical attributes called *features*. In the supervised learning problems above, a special feature is designated as the prediction *target*, (sometimes called the *label* or *dependent variable*). The given features from which the model must make its predictions can then simply be called the *features*, (or often, the *inputs*, *covariates*, or *independent variables*).

If we were working with image data, each individual photograph might constitute an *example*, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values, corresponding to the brightness of the red, green, and blue channels for each spatial location. In a more traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age, vital signs, diagnoses, etc.

When every example is characterized by the same number of numerical values, we say that the data consists of *fixed-length* vectors and we describe the (constant) length of the vectors as the *dimensionality* of the data. As you might imagine, fixed-length can be a convenient property. If we wanted to train a model to recognize cancer in microscopy images, fixed-length inputs mean we have one less thing to worry about.

However, not all data can easily be represented as fixed-length vectors. While we might expect microscope images to come from standard equipment, we cannot expect images mined from the Internet to all show up with the same resolution or shape. For images, we might consider cropping them all to a standard size, but that strategy only gets us so far. We risk losing information in the cropped out portions. Moreover, text data resists fixed-length representations even more stubbornly. Consider the customer reviews left on e-commerce sites like Amazon, IMDB, or TripAdvisor. Some are short: "it stinks!". Others ramble for pages. One major advantage of deep learning over traditional methods is the comparative grace with which modern models can handle *varying-length* data.

Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models and rely less heavily on pre-conceived assumptions. The regime change from (comparatively) small to big data is a major contributor to the success of modern deep learning. To drive the point home, many of the most exciting models in deep learning do not work without large datasets. Some others work in the low-data regime, but are no better than traditional approaches.

Finally, it is not enough to have lots of data and to process it cleverly. We need the *right* data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is captured well by the cliché: *garbage in, garbage*

capture and automate historical injustices. Note that this can all happen without the data scientist actively conspiring, or even being aware.

1.2.2 Models

Most machine learning involves *transforming* the data in some sense. We might want to build a system that ingests photos and predicts *smiley-ness*. Alternatively, we might want to ingest a set of sensor readings and predict how *normal* vs *anomalous* the readings are. By *model*, we denote the computational machinery for ingesting data of one type, and spitting out predictions of a possibly different type. In particular, we are interested in statistical models that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems the problems that we focus on in this book stretch the limits of classical methods. Deep learning is differentiated from classical approaches principally by the set of powerful models that it focuses on. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep neural networks, we will discuss some more traditional methods.

1.2.3 Objective functions

Earlier, we introduced machine learning as "learning from experience". By *learning* here, we mean *improving* at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose to update our model, and some people might disagree on whether the proposed update constituted an improvement or a decline.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these objective functions. By convention, we usually define objective functions so that *lower* is *better*. This is merely a convention. You can take any function f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y-\hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to optimize. Others (like error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common to optimize a *surrogate objective*.

Typically, the loss function is defined with respect to the model's parameters and depends upon the dataset. The best values of our model's parameters are learned by minimizing the loss incurred on a *training set* consisting of some number of *examples* collected for training. However, doing well on the training data does not guarantee that we will do well on (unseen) test data. So we will typically

eralize to unseen data, we say that it is *overfitting*. In real-life terms, this is like flunking the real exam despite doing well on practice exams.

1.2.4 Optimization algorithms

Once we have got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. The most popular optimization algorithms for neural networks follow an approach called gradient descent. In short, at each step, they check to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. They then update the parameter in the direction that reduces the loss.

1.3 Kinds of Machine Learning

In the following sections, we discuss a few *kinds* of machine learning problems in greater detail. We begin with a list of *objectives*, i.e., a list of things that we would like machine learning to do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, including types of data, models, training techniques, etc. The list below is just a sampling of the problems ML can tackle to motivate the reader and provide us with some common language for when we talk about more problems throughout the book.

1.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given *inputs*. The targets, which we often call *labels*, are generally denoted by y. The input data, also called the *features* or covariates, are typically denoted $\mathbf{x}$. Each (input, target) pair is called an *example* or *instance*. Sometimes, when the context is clear, we may use the term examples, to refer to a collection of inputs, even when the corresponding targets are unknown. We denote any particular instance with a subscript, typically i, for instance $(\mathbf{x}_i, y_i)$. A dataset is a collection of n instances $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Our goal is to produce a model f_θ that maps any input $\mathbf{x}_i$ to a prediction $f_\theta(\mathbf{x}_i)$.

To ground this description in a concrete example, if we were working in healthcare, then we might want to predict whether or not a patient would have a heart attack. This observation, *heart attack* or *no heart attack*, would be our label y. The input data $\mathbf{x}$ might be vital signs such as heart rate, diastolic and systolic blood pressure, etc.

The supervision comes into play because for choosing the parameters θ, we (the supervisors) provide the model with a dataset consisting of *labeled examples* $(\mathbf{x}_i, y_i)$, where each example $\mathbf{x}_i$ is matched with the correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$.

Even with the simple description "predict targets from inputs" supervised learning can take a great many forms and require a great many modeling decisions, depending on (among other considerations) the type, size, and the number of inputs and outputs. For example, we use different models to process sequences (like strings of text or time series data) and for processing fixed-length vector representations. We will visit many of these problems in depth throughout the first 9 parts of this book.

Informally, the learning process looks something like this: Grab a big collection of examples for which the covariates are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that has already been collected (e.g., did a patient die within the following year?) and other times we might need to employ human annotators to label the data, (e.g., assigning images to categories).

Together, these inputs and corresponding labels comprise the training set. We feed the training dataset into a supervised learning algorithm, a function that takes as input a dataset and outputs another function, *the learned model*. Finally, we can feed previously unseen inputs to the learned model, using its outputs as predictions of the corresponding label. The full process is drawn in Fig. 1.3.1.

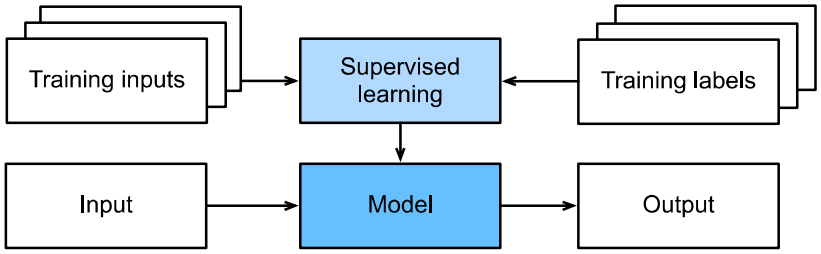


Fig. 1.3.1: Supervised learning.

Regression

Perhaps the simplest supervised learning task to wrap your head around is *regression*. Consider, for example, a set of data harvested from a database of home sales. We might construct a table, where each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes (walking) to the center of town. In this dataset, each *example* would be a specific house, and the corresponding *feature vector* would be one row in the table.

If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[100, 0, .5, 60]$. However, if you live in Pittsburgh, it might look more like $[3000, 4, 3, 10]$. Feature vectors like this are essential for most classic machine learning algorithms. We will continue to denote the feature vector corresponding to any example i as $\mathbf{x}_i$ and we can compactly refer to the full table containing all of the feature vectors as X.

We denote any individual target y_i (corresponding to example $\mathbf{x}_i$) and the set of all targets $\mathbf{y}$ (corresponding to all examples X). When our targets take on arbitrary values in some range, we call this a regression problem. Our goal is to produce a model whose predictions closely approximate the actual target values. We denote the predicted target for any instance $\hat{y}_i$. Do not worry if the notation is bogging you down. We will unpack it more thoroughly in the subsequent chapters.

Lots of practical problems are well-described regression problems. Predicting the rating that a user will assign to a movie can be thought of as a regression problem and if you designed a great algorithm to accomplish this feat in 2009, you might have won the 1-million-dollar Netflix prize[13]. Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *How much?* or *How many?* problem should suggest regression.

- "How many hours will this surgery take?": *regression*
- "How many dogs are in this photo?": *regression*.

However, if you can easily pose your problem as "Is this a _ ?", then it is likely, classification, a different kind of supervised problem that we will cover next. Even if you have never worked with machine learning before, you have probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor spent $x_1 = 3$ hours removing gunk from your sewage pipes. Then he sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that he received a bill of $y_2 = \$250$. If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there is some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data examples, you could already identify the contractor's pricing structure: $100 per hour plus $50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression (and you just implicitly designed a linear model with a bias term).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that is not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we will try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we will focus on one of two very common losses, the L1 loss where

$$l(y, y') = \sum_i |y_i - y_i'| \tag{1.3.1}$$

and the least mean squares loss, or L_2 loss where

$$l(y, y') = \sum_i (y_i - y_i')^2. \tag{1.3.2}$$

As we will see later, the L_2 loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the L_1 loss corresponds to an assumption of noise from a Laplace distri-

Classification

While regression models are great for addressing *how many?* questions, lots of problems do not bend comfortably to this template. For example, a bank wants to add check scanning to its mobile app. This would involve the customer snapping a photo of a check with their smart phone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it addresses is called *classification*. It is treated with a different set of algorithms than those used for regression (although many techniques will carry over).

In classification, we want our model to look at a feature vector, e.g., the pixel values in an image, and then predict which category (formally called *classes*), among some (discrete) set of options, an example belongs. For hand-written digits, we might have 10 classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call binary classification. For example, our dataset X could consist of images of animals and our *labels* Y might be the classes $\{\text{cat}, \text{dog}\}$. While in regression, we sought a *regressor* to output a real value $\hat{y}$, in classification, we seek a *classifier*, whose output $\hat{y}$ is the predicted class assignment.

For reasons that we will get into as the book gets more technical, it can be hard to optimize a model that can only output a hard categorical assignment, e.g., either *cat* or *dog*. In these cases, it is usually much easier to instead express our model in the language of probabilities. Given an example x, our model assigns a probability $\hat{y}_k$ to each label k. Because these are probabilities, they need to be positive numbers and add up to 1 and thus we only need $K - 1$ numbers to assign probabilities of K categories. This is easy to see for binary classification. If there is a 0.6 (60%) probability that an unfair coin comes up heads, then there is a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat $P(y = \text{cat} \mid x) = 0.9$. We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class conveys one notion of uncertainty. It is not the only notion of uncertainty and we will discuss others in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L_1 or L_2 loss functions, the common loss function for classification problems is called cross-entropy.

Note that the most likely class is not necessarily the one that you are going to use for your decision. Assume that you find this beautiful mushroom in your backyard as shown in Fig. 1.3.2.

Fig. 1.3.2: Death cap—do not eat!

Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs $P(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% sure that our mushroom *is not* a death cap. Still, you would have to be a fool to eat it. That is because the certain benefit of a delicious dinner is not worth a 20% risk of dying from it. In other words, the effect of the *uncertain risk* outweighs the benefit by far. We can look at this more formally. Basically, we need to compute the expected risk that we incur, i.e., we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action}|x) = E_{y\sim p(y|x)}[\text{loss}(\text{action}, y)]. \tag{1.3.3}$$

Hence, the loss L incurred by eating the mushroom is $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$, whereas the cost of discarding it is $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Our caution was justified: as any mycologist would tell us, the above mushroom actually *is* a death cap. Classification can get much more complicated than just binary, multiclass, or even multi-label classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal—if we must err, we would prefer to misclassify to a related class rather than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus[14], who organized the animals in a hierarchy.

In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle for a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattle snakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could be deadly.

[14] https://en.wikipedia.org/wiki/Carl_Linnaeus

Tagging

Some classification problems do not fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image of the Town Musicians of Bremen.

Fig. 1.3.3: A cat, a rooster, a dog and a donkey

As you can see, there is a cat in the picture, and a rooster, a dog, a donkey, and a bird, with some trees in the background. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat *and* a dog *and* a donkey *and* a rooster *and* a bird.

The problem of learning to predict classes that are *not mutually exclusive* is called multi-label classification. Auto-tagging problems are typically best described as multi-label classification prob-

each article that gets indexed in PubMed to associate it with the relevant terms from MeSH, a collection of roughly 28k tags. This is a time-consuming process and the annotators typically have a one year lag between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has hosted a competition[15] to do precisely this.

Search and ranking

Sometimes we do not just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for example, the goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results is *most relevant* for a particular user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning `A B C D E` and `C A B E D`. Even if the result set is the same, the ordering within the set matters.

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. PageRank[16], the original secret sauce behind the Google search engine was an early example of such a scoring system but it was peculiar in that it did not depend on the actual query. Here they relied on a simple relevance filter to identify the set of relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

Recommender systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a SciFi fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, or news recommendation.

In some cases, customers provide explicit feedback communicating how much they liked a particular product (e.g., the product ratings and reviews on Amazon, IMDB, GoodReads, etc.). In some other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score y_{ij}, such as an estimated rating or the probability of purchase, given a user u_i and product p_j.

Given such a model, then for any given user, we could retrieve the set of objects with the largest scores y_{ij}, which could then be recommended to the customer. Production systems are consid-

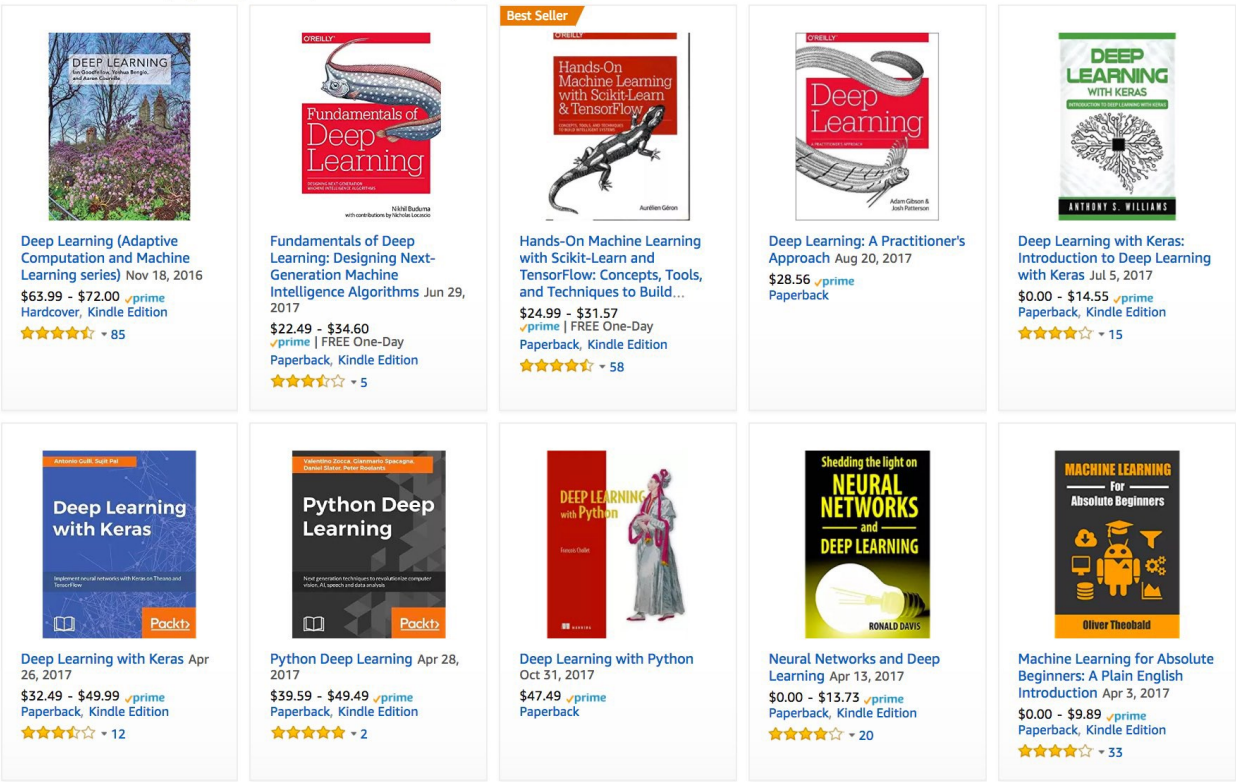


Fig. 1.3.4: Deep learning books recommended by Amazon.

Despite their tremendous economic value, recommendation systems naively built on top of predictive models suffer some serious conceptual flaws. To start, we only observe *censored feedback*. Users preferentially rate movies that they feel strongly about: you might notice that items receive many 5 and 1 star ratings but that there are conspicuously few 3-star ratings. Moreover, current purchase habits are often a result of the recommendation algorithm currently in place, but learning algorithms do not always take this detail into account. Thus it is possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) and in turn is recommended even more frequently. Many of these problems about how to deal with censoring, incentives, and feedback loops, are important open research questions.

Sequence Learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it

Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from the spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of special cases are worth mentioning:

Tagging and Parsing. This involves annotating a text sequence with attributes. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities.

```
Tom has dinner in Washington with Sally.
Ent  -    -    -     Ent      -    Ent
```

Automatic Speech Recognition. With speech recognition, the input sequence x is an audio recording of a speaker (shown in Fig. 1.3.5), and the output y is the textual transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e., there is no 1:1 correspondence between audio and text, since thousands of samples correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.

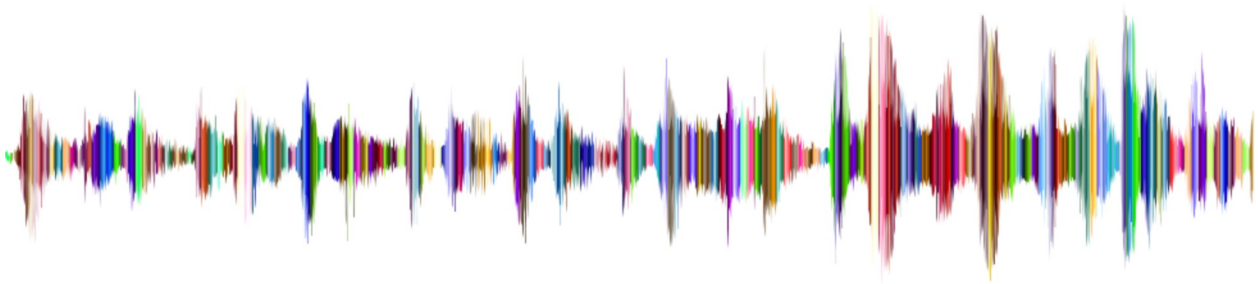


Fig. 1.3.5: `-D-e-e-p- L-ea-r-ni-ng-`

Consider the following illustrative example of the peculiar tendency of Germans to place the verbs at the end of sentences.

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial looked-at?
```

Many related problems pop up in other learning tasks. For instance, determining the order in which a user reads a Webpage is a two-dimensional layout analysis problem. Dialogue problems exhibit all kinds of additional complications, where determining what to say next requires taking into account real-world knowledge and the prior state of the conversation across long temporal distances. This is an active area of research.

1.3.2 Unsupervised learning

All the examples so far were related to *Supervised Learning*, i.e., situations where we feed the model a giant dataset containing both the features and corresponding target values. You could think of the supervised learner as having an extremely specialized job and an extremely anal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it is easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you'd better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is. We call this class of problems *unsupervised learning*, and the type and number of questions we could ask is limited only by our creativity. We will address a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the questions you might ask:

- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, can we group them into users with similar behavior? This problem is typically known as *clustering*.
- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are quite well described by velocity, diameter, and mass of the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as *subspace estimation* problems. If the dependence is linear, it is called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e., the space

- Another important and exciting recent development in unsupervised learning is the advent of *generative adversarial networks* (GANs). These give us a procedural way to synthesize data, even complicated structured data like images and audio. The underlying statistical mechanisms are tests to check whether real and fake data are the same. We will devote a few notebooks to them.

1.3.3 Interacting with an Environment

So far, we have not discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data upfront, then set our pattern recognition machines in motion without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For supervised learning, the process looks like Fig. 1.3.6.

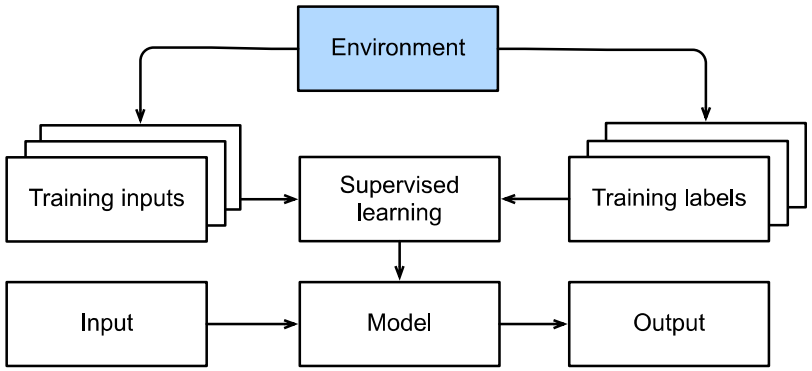


Fig. 1.3.6: Collect data for supervised learning from an environment.

This simplicity of offline learning has its charms. The upside is we can worry about pattern recognition in isolation, without any distraction from these other problems. But the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot Series, then you might imagine artificially intelligent bots capable not only of making predictions, but of taking actions in the world. We want to think about intelligent *agents*, not just predictive *models*. That means we need to think about choosing *actions*, not just making *predictions*. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

Considering the interaction with an environment opens a whole set of new modeling questions. Does the environment:

This last question raises the problem of *distribution shift*, (when training and test data are different). It is a problem that most of us have experienced when taking exams written by a lecturer, while the homeworks were composed by his TAs. We will briefly describe reinforcement learning and adversarial learning, two settings that explicitly consider interaction with an environment.

1.3.4 Reinforcement learning

If you are interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you are probably going to wind up focusing on *reinforcement learning* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games. *Deep reinforcement learning* (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input[17], and the AlphaGo program that dethroned the world champion at the board game Go[18] are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of *timesteps*. At each timestep t, the agent receives some observation o_t from the environment and must choose an action a_t that is subsequently transmitted back to the environment via some mechanism (sometimes called an actuator). Finally, the agent receives a reward r_t from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL agent is governed by a *policy*. In short, a *policy* is just a function that maps from observations (of the environment) to actions. The goal of reinforcement learning is to produce a good policy.

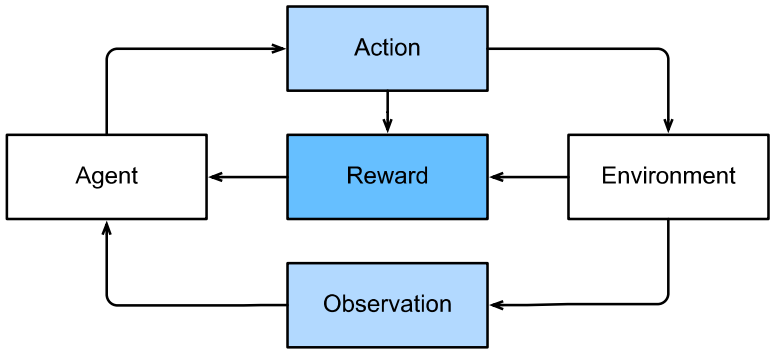


Fig. 1.3.7: The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL agent with one *action* corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised problem.

That being said, RL can also address many problems that supervised learning cannot. For exam-

assign a reward of -1. So reinforcement learners must deal with the *credit assignment problem*: determining which actions to credit or blame for an outcome. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best currently-known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-run reward in exchange for knowledge.

MDPs, bandits, and friends

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. Moreover, not every practical problem exhibits all this complexity. As a result, researchers have studied a number of *special cases* of reinforcement learning problems.

When the environment is fully observed, we call the RL problem a *Markov Decision Process* (MDP). When the state does not depend on the previous actions, we call the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, this problem is the classic *multi-armed bandit problem*.

1.4 Roots

Although many deep learning methods are recent inventions, humans have held the desire to analyze data and to predict future outcomes for centuries. In fact, much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655-1705)[19], and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777-1855)[20]. He invented, for instance, the least mean squares algorithm, which is still used today for countless problems from insurance calculations to medical diagnostics. These tools gave rise to an experimental approach in the natural sciences—for instance, Ohm's law relating current and voltage in a resistor is perfectly described by a linear model.

Even in the middle ages, mathematicians had a keen intuition of estimates. For instance, the geometry book of Jacob Köbel (1460-1533)[21] illustrates averaging the length of 16 adult men's feet

Fig. 1.4.1: Estimating the length of a foot

Fig. 1.4.1 illustrates how this estimator works. The 16 adult men were asked to line up in a row, when leaving church. Their aggregate length was then divided by 16 to obtain an estimate for what now amounts to 1 foot. This "algorithm" was later improved to deal with misshapen feet—the 2 men with the shortest and longest feet respectively were sent away, averaging only over the remainder. This is one of the earliest examples of the trimmed mean estimate.

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890-1962)[22], contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as Linear Discriminant Analysis) and formula (such as the Fisher Information Matrix) are still in frequent use today (even the Iris dataset that he released in 1936 is still used sometimes to illustrate machine learning algorithms). Fisher was also a proponent of eugenics, which should remind us that the morally dubious use of data science has as long and enduring a history as its productive use in industry and the natural sciences.

A second influence for machine learning came from Information Theory (Claude Shannon, 1916-2001)[23] and the Theory of computation via Alan Turing (1912-1954)[24]. Turing posed the question "can machines think?" in his famous paper Computing machinery and intelligence[25] (Mind, October 1950). In what he described as the Turing test, a machine can be considered intelligent if it

verse engineer this capacity. One of the oldest algorithms inspired in this fashion was formulated by Donald Hebb (1904-1985)[26]. In his groundbreaking book The Organization of Behavior (Hebb & Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. It is the prototype of Rosenblatt's perceptron learning algorithm and it laid the foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior and diminish undesirable behavior to obtain good settings of the parameters in a neural network.

Biological inspiration is what gave *neural networks* their name. For over a century (dating back to the models of Alexander Bain, 1873 and James Sherrington, 1890), researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time, the interpretation of biology has become less literal but the name stuck. At its heart, lie a few key principles that can be found in most networks today:

- The alternation of linear and nonlinear processing units, often referred to as *layers*.
- The use of the chain rule (also known as *backpropagation*) for adjusting parameters in the entire network at once.

After initial rapid progress, research in neural networks languished from around 1995 until 2005. This was due to a number of reasons. Training a network is computationally very expensive. While RAM was plentiful at the end of the past century, computational power was scarce. Second, datasets were relatively small. In fact, Fisher's Iris dataset from 1932 was a popular tool for testing the efficacy of algorithms. MNIST with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as Kernel Methods, Decision Trees and Graphical Models proved empirically superior. Unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

1.5 The Road to Deep Learning

Much of this changed with the ready availability of large amounts of data, due to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of cheap, high-quality sensors, cheap data storage (Kryder's law), and cheap computation (Moore's law), in particular in the form of GPUs, originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible became relevant (and vice versa). This is best illustrated in Table 1.5.1.

Table 1.5.1: Dataset vs. computer memory and computational power

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)

in computational power has outpaced that of the data available. This means that statistical models needed to become more memory efficient (this is typically achieved by adding nonlinearities) while simultaneously being able to spend more time on optimizing these parameters, due to an increased compute budget. Consequently, the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep networks. This is also one of the reasons why many of the mainstays of deep learning, such as multilayer perceptrons (McCulloch & Pitts, 1943), convolutional neural networks (LeCun et al., 1998), Long Short-Term Memory (Hochreiter & Schmidhuber, 1997), and Q-Learning (Watkins & Dayan, 1992), were essentially "rediscovered" in the past decade, after laying comparatively dormant for considerable time.

The recent progress in statistical models, applications, and algorithms, has sometimes been likened to the Cambrian Explosion: a moment of rapid progress in the evolution of species. Indeed, the state of the art is not just a mere consequence of available resources, applied to decades old algorithms. Note that the list below barely scratches the surface of the ideas that have helped researchers achieve tremendous progress over the past decade.

- Novel methods for capacity control, such as Dropout (Srivastava et al., 2014) have helped to mitigate the danger of overfitting. This was achieved by applying noise injection (Bishop, 1995) throughout the network, replacing weights by random variables for training purposes.
- Attention mechanisms solved a second problem that had plagued statistics for over a century: how to increase the memory and complexity of a system without increasing the number of learnable parameters. (Bahdanau et al., 2014) found an elegant solution by using what can only be viewed as a learnable pointer structure. Rather than having to remember an entire sentence, e.g., for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to the intermediate state of the translation process. This allowed for significantly increased accuracy for long sentences, since the model no longer needed to remember the entire sentence before commencing the generation of a new sentence.
- Multi-stage designs, e.g., via the Memory Networks (MemNets) (Sukhbaatar et al., 2015) and the Neural Programmer-Interpreter (Reed & DeFreitas, 2015) allowed statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, similar to how a processor can modify memory for a computation.
- Another key development was the invention of GANs (Goodfellow et al., 2014). Traditionally, statistical methods for density estimation and generative models focused on finding proper probability distributions and (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by the lack of flexibility inherent in the statistical models. The crucial innovation in GANs was to replace the sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that the discriminator (effectively a two-sample test) cannot distinguish fake from real data. Through the ability to use arbitrary algorithms to generate data, it opened up density estimation to

atively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1024 GPUs with a minibatch size of, say 32 images per batch amounts to an aggregate minibatch of 32k images. Recent work, first by Li (Li, 2017), and subsequently by (You et al., 2017) and (Jia et al., 2018) pushed the size up to 64k observations, reducing training time for ResNet50 on ImageNet to less than 7 minutes. For comparison—initially training times were measured in the order of days.

- The ability to parallelize computation has also contributed quite crucially to progress in reinforcement learning, at least whenever simulation is an option. This has led to significant progress in computers achieving superhuman performance in Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo). See e.g., (Silver et al., 2016) for a description of how to achieve this in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) triples are available, i.e., whenever it is possible to try out lots of things to learn how they relate to each other. Simulation provides such an avenue.

- Deep Learning frameworks have played a crucial role in disseminating ideas. The first generation of frameworks allowing for easy modeling encompassed Caffe[27], Torch[28], and Theano[29]. Many seminal papers were written using these tools. By now, they have been superseded by TensorFlow[30], often used via its high level API Keras[31], CNTK[32], Caffe 2[33], and Apache MxNet[34]. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer[35], which used a syntax similar to Python NumPy to describe models. This idea was adopted by both PyTorch[36], the Gluon API[37] of MXNet, and Jax[38]. It is the latter group that this course uses to teach deep learning.

The division of labor between systems researchers building better tools and statistical modelers building better networks has greatly simplified things. For instance, training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new machine learning PhD students at Carnegie Mellon University in 2014. By now, this task can be accomplished with less than 10 lines of code, putting it firmly into the grasp of programmers.

1.6 Success Stories

Artificial Intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail is sorted using optical character recognition. These systems have been deployed since the 90s (this is, after all, the source of the famous MNIST and USPS sets of handwritten digits). The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud automatically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization and ranking on the Internet. In other words, artificial intelligence and machine learning are pervasive, albeit often hidden from sight.

[27] https://github.com/BVLC/caffe

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, or Google's assistant are able to answer spoken questions with a reasonable degree of accuracy. This includes menial tasks such as turning on light switches (a boon to the disabled) up to making barber's appointments and offering phone support dialog. This is likely the most noticeable sign that AI is affecting our lives.
- A key ingredient in digital assistants is the ability to recognize speech accurately. Gradually the accuracy of such systems has increased to the point where they reach human parity (Xiong et al., 2018) for certain applications.
- Object recognition likewise has come a long way. Estimating the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark (Lin et al., 2010) achieved a top-5 error rate of 28%. By 2017, (Hu et al., 2018) reduced this error rate to 2.25%. Similarly, stunning results have been achieved for identifying birds, or diagnosing skin cancer.
- Games used to be a bastion of human intelligence. Starting from TDGammon [23], a program for playing Backgammon using temporal difference (TD) reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Unlike Backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Garry Kasparov, Campbell et al. (Campbell et al., 2002), using massive parallelism, special purpose hardware and efficient search through the game tree. Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, (Silver et al., 2016) using Deep Learning combined with Monte Carlo tree sampling. The challenge in Poker was that the state space is large and it is not fully observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017). This illustrates the impressive progress in games and the fact that advanced algorithms played a crucial part in them.
- Another indication of progress in AI is the advent of self-driving cars and trucks. While full autonomy is not quite within reach yet, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable at least partial autonomy. What makes full autonomy so challenging is that proper driving requires the ability to perceive, to reason and to incorporate rules into a system. At present, deep learning is used primarily in the computer vision aspect of these problems. The rest is heavily tuned by engineers.

Again, the above list barely scratches the surface of where machine learning has impacted practical applications. For instance, robotics, logistics, computational biology, particle physics, and astronomy owe some of their most impressive recent advances at least in parts to machine learning. ML is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, the question of the AI apocalypse, or the AI singularity has been raised in nontechnical articles on AI. The fear is that somehow machine learning systems will become sentient

bination of rules, heuristics and statistical models that underlie the design. Second, at present tools for *artificial general intelligence* simply do not exist that are able to improve themselves, reason about themselves, and that are able to modify, extend and improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. It is likely that many menial tasks fulfilled by truck drivers and shop assistants can and will be automated. Farm robots will likely reduce the cost for organic farming but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences on large swaths of society (truck drivers and shop assistants are some of the most common jobs in many states). Furthermore, statistical models, when applied without care can lead to racial, gender or age bias and raise reasonable concerns about procedural fairness if automated to drive consequential decisions. It is important to ensure that these algorithms are used with care. With what we know today, this strikes us a much more pressing concern than the potential of malevolent superintelligence to destroy humanity.

Summary

- Machine learning studies how computer systems can leverage *experience* (often data) to improve performance at specific tasks. It combines ideas from statistics, data mining, artificial intelligence, and optimization. Often, it is used as a means of implementing artificially-intelligent solutions.
- As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. This is often accomplished by a progression of learned transformations.
- Much of the recent progress in deep learning has been triggered by an abundance of data arising from cheap sensors and Internet-scale applications, and by significant progress in computation, mostly through GPUs.
- Whole system optimization is a key component in obtaining good performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

Exercises

1. Which parts of code that you are currently writing could be "learned", i.e., improved by learning and automatically determining design choices that are made in your code? Does your code include heuristic design choices?
2. Which problems that you encounter have many examples for how to solve them, yet no spe-

Discussions[39]

2 | Preliminaries

To get started with deep learning, we will need to develop a few basic skills. All machine learning is concerned with extracting information from data. So we will begin by learning the practical skills for storing, manipulating, and preprocessing data.

Moreover, machine learning typically requires working with large datasets, which we can think of as tables, where the rows correspond to examples and the columns correspond to attributes. Linear algebra gives us a powerful set of techniques for working with tabular data. We will not go too far into the weeds but rather focus on the basic of matrix operations and their implementation.

Additionally, deep learning is all about optimization. We have a model with some parameters and we want to find those that fit our data *the best*. Determining which way to move each parameter at each step of an algorithm requires a little bit of calculus, which will be briefly introduced. Fortunately, the `autograd` package automatically computes differentiation for us, and we will cover it next.

Next, machine learning is concerned with making predictions: what is the likely value of some unknown attribute, given the information that we observe? To reason rigorously under uncertainty we will need to invoke the language of probability.

In the end, the official documentation provides plenty of descriptions and examples that are beyond this book. To conclude the chapter, we will show you how to look up documentation for the needed information.

This book has kept the mathematical content to the minimum necessary to get a proper understanding of deep learning. However, it does not mean that this book is mathematics free. Thus, this chapter provides a rapid introduction to basic and frequently-used mathematics to allow anyone to understand at least *most* of the mathematical content of the book. If you wish to understand *all* of the mathematical content, further reviewing the online appendix on mathematics[40] should be sufficient.

2.1 Data Manipulation

In order to get anything done, we need some way to store and manipulate data. Generally, there

(ndarray in MXNet, Tensor in both PyTorch and TensorFlow) is similar to NumPy's ndarray with a few killer features. First, GPU is well-supported to accelerate the computation whereas NumPy only supports CPU computation. Second, the tensor class supports automatic differentiation. These properties make the tensor class suitable for deep learning. Throughout the book, when we say tensors, we are referring to instances of the tensor class unless otherwise stated.

2.1.1 Getting Started

In this section, we aim to get you up and running, equipping you with the basic math and numerical computing tools that you will build on as you progress through the book. Do not worry if you struggle to grok some of the mathematical concepts or library functions. The following sections will revisit this material in the context of practical examples and it will sink. On the other hand, if you already have some background and want to go deeper into the mathematical content, just skip this section.

To start, we import the np (numpy) and npx (numpy_extension) modules from MXNet. Here, the np module includes functions supported by NumPy, while the npx module contains a set of extensions developed to empower deep learning within a NumPy-like environment. When using tensors, we almost always invoke the set_np function: this is for compatibility of tensor processing by other components of MXNet.

```
from mxnet import np, npx
npx.set_np()
```

A tensor represents a (possibly multi-dimensional) array of numerical values. With one axis, a tensor corresponds (in math) to a *vector*. With two axes, a tensor corresponds to a *matrix*. Tensors with more than two axes do not have special mathematical names.

To start, we can use arange to create a row vector x containing the first 12 integers starting with 0, though they are created as floats by default. Each of the values in a tensor is called an *element* of the tensor. For instance, there are 12 elements in the tensor x. Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation.

```
x = np.arange(12)
x
```

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

We can access a tensor's *shape* (the length along each axis) by inspecting its shape property.

```
x.shape
```

```
12
```

To change the shape of a tensor without altering either the number of elements or their values, we can invoke the `reshape` function. For example, we can transform our tensor, `x`, from a row vector with shape (12,) to a matrix with shape (3, 4). This new tensor contains the exact same values, but views them as a matrix organized as 3 rows and 4 columns. To reiterate, although the shape has changed, the elements have not. Note that the size is unaltered by reshaping.

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

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

Reshaping by manually specifying every dimension is unnecessary. If our target shape is a matrix with shape (height, width), then after we know the width, the height is given implicitly. Why should we have to perform the division ourselves? In the example above, to get a matrix with 3 rows, we specified both that it should have 3 rows and 4 columns. Fortunately, tensors can automatically work out one dimension given the rest. We invoke this capability by placing `-1` for the dimension that we would like tensors to automatically infer. In our case, instead of calling `x.reshape(3, 4)`, we could have equivalently called `x.reshape(-1, 4)` or `x.reshape(3, -1)`.

Typically, we will want our matrices initialized either with zeros, ones, some other constants, or numbers randomly sampled from a specific distribution. We can create a tensor representing a tensor with all elements set to 0 and a shape of (2, 3, 4) as follows:

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

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

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

Similarly, we can create tensors with each element set to 1 as follows:

```
np.ones((2, 3, 4))
```

```
array([[[1., 1., 1., 1.],
```

we will typically initialize their values randomly. The following snippet creates a tensor with shape (3, 4). Each of its elements is randomly sampled from a standard Gaussian (normal) distribution with a mean of 0 and a standard deviation of 1.

```
np.random.normal(0, 1, size=(3, 4))
```

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434405 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

We can also specify the exact values for each element in the desired tensor by supplying a Python list (or list of lists) containing the numerical values. Here, the outermost list corresponds to axis 0, and the inner list to axis 1.

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
array([[2., 1., 4., 3.],
       [1., 2., 3., 4.],
       [4., 3., 2., 1.]])
```

2.1.2 Operations

This book is not about software engineering. Our interests are not limited to simply reading and writing data from/to arrays. We want to perform mathematical operations on those arrays. Some of the simplest and most useful operations are the *elementwise* operations. These apply a standard scalar operation to each element of an array. For functions that take two arrays as inputs, elementwise operations apply some standard binary operator on each pair of corresponding elements from the two arrays. We can create an elementwise function from any function that maps from a scalar to a scalar.

In mathematical notation, we would denote such a *unary* scalar operator (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$. This just means that the function is mapping from any real number ($\mathbb{R}$) onto another. Likewise, we denote a *binary* scalar operator (taking two real inputs, and yielding one output) by the signature $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Given any two vectors $\mathbf{u}$ and $\mathbf{v}$ *of the same shape*, and a binary operator f, we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i, where c_i, u_i, and v_i are the ith elements of vectors $\mathbf{c}, \mathbf{u}$, and $\mathbf{v}$. Here, we produced the vector-valued $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ by *lifting* the scalar function to an elementwise vector operation.

The common standard arithmetic operators (`+`, `-`, `*`, `/`, and `**`) have all been *lifted* to elementwise operations for any identically-shaped tensors of arbitrary shape. We can call elementwise operations on any two tensors of the same shape. In the following example, we use commas to formulate a 5-element tuple, where each element is the result of an elementwise operation.

(continued from previous page)

```
 array([ 2.,  4.,  8., 16.]),
 array([0.5, 1. , 2. , 4. ]),
 array([ 1.,  4., 16., 64.]))
```

Many more operations can be applied elementwise, including unary operators like exponentiation.

```
np.exp(x)
```

```
array([2.7182817e+00, 7.3890562e+00, 5.4598148e+01, 2.9809580e+03])
```

In addition to elementwise computations, we can also perform linear algebra operations, including vector dot products and matrix multiplication. We will explain the crucial bits of linear algebra (with no assumed prior knowledge) in Section 2.3.

We can also *concatenate* multiple tensors together, stacking them end-to-end to form a larger tensor. We just need to provide a list of tensors and tell the system along which axis to concatenate. The example below shows what happens when we concatenate two matrices along rows (axis 0, the first element of the shape) vs. columns (axis 1, the second element of the shape). We can see that the first output tensor's axis-0 length (6) is the sum of the two input tensors' axis-0 lengths ($3+3$); while the second output tensor's axis-1 length (8) is the sum of the two input tensors' axis-1 lengths ($4+4$).

```
X = np.arange(12).reshape(3, 4)
Y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([X, Y], axis=0), np.concatenate([X, Y], axis=1)
```

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

Sometimes, we want to construct a binary tensor via *logical statements*. Take `X == Y` as an example. For each position, if `X` and `Y` are equal at that position, the corresponding entry in the new tensor takes a value of 1, meaning that the logical statement `X == Y` is true at that position; otherwise that position takes 0.

```
X.sum()
```

```
array(66.)
```

2.1.3 Broadcasting Mechanism

In the above section, we saw how to perform elementwise operations on two tensors of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise operations by invoking the *broadcasting mechanism*. This mechanism works in the following way: First, expand one or both arrays by copying elements appropriately so that after this transformation, the two tensors have the same shape. Second, carry out the elementwise operations on the resulting arrays.

In most cases, we broadcast along an axis where an array initially only has length 1, such as in the following example:

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

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

Since a and b are 3×1 and 1×2 matrices respectively, their shapes do not match up if we want to add them. We *broadcast* the entries of both matrices into a larger 3×2 matrix as follows: for matrix a it replicates the columns and for matrix b it replicates the rows before adding up both elementwise.

```
a + b
```

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

2.1.4 Indexing and Slicing

Just as in any other Python array, elements in a tensor can be accessed by index. As in any Python

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

Beyond reading, we can also write elements of a matrix by specifying indices.

```
X[1, 2] = 9
X
```

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

If we want to assign multiple elements the same value, we simply index all of them and then assign them the value. For instance, `[0:2, :]` accesses the first and second rows, where `:` takes all the elements along axis 1 (column). While we discussed indexing for matrices, this obviously also works for vectors and for tensors of more than 2 dimensions.

```
X[0:2, :] = 12
X
```

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

2.1.5 Saving Memory

Running operations can cause new memory to be allocated to host results. For example, if we write `Y = X + Y`, we will dereference the tensor that `Y` used to point to and instead point `Y` at the newly allocated memory. In the following example, we demonstrate this with Python's `id()` function, which gives us the exact address of the referenced object in memory. After running `Y = Y + X`, we will find that `id(Y)` points to a different location. That is because Python first evaluates `Y + X`, allocating new memory for the result and then makes `Y` point to this new location in memory.

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

```
False
```

concept, we first create a new matrix `Z` with the same shape as another `Y`, using `zeros_like` to allocate a block of 0 entries.

```
Z = np.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140228072655200
id(Z): 140228072655200
```

If the value of `X` is not reused in subsequent computations, we can also use `X[:] = X + Y` or `X += Y` to reduce the memory overhead of the operation.

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

```
True
```

2.1.6 Conversion to Other Python Objects

Converting to a NumPy tensor, or vice versa, is easy. The converted result does not share memory. This minor inconvenience is actually quite important: when you perform operations on the CPU or on GPUs, you do not want to halt computation, waiting to see whether the NumPy package of Python might want to be doing something else with the same chunk of memory.

```
A = X.asnumpy()
B = np.array(A)
type(A), type(B)
```

```
(numpy.ndarray, mxnet.numpy.ndarray)
```

To convert a size-1 tensor to a Python scalar, we can invoke the `item` function or Python's built-in functions.

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

```
(array([3.5]), 3.5, 3.5, 3)
```

Summary

- The main interface to store and manipulate data for deep learning is the tensor (n-dimensional array). It provides a variety of functionalities including basic mathematics operations, broadcasting, indexing, slicing, memory saving, and conversion to other Python objects.

Exercises

1. Run the code in this section. Change the conditional statement `X == Y` in this section to `X < Y` or `X > Y`, and then see what kind of tensor you can get.
2. Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

Discussions[41]

2.2 Data Preprocessing

So far we have introduced a variety of techniques for manipulating data that are already stored in tensors. To apply deep learning to solving real-world problems, we often begin with preprocessing raw data, rather than those nicely prepared data in the tensor format. Among popular data analytic tools in Python, the `pandas` package is commonly used. Like many other extension packages in the vast ecosystem of Python, `pandas` can work together with tensors. So, we will briefly walk through steps for preprocessing raw data with `pandas` and converting them into the tensor format. We will cover more data preprocessing techniques in later chapters.

2.2.1 Reading the Dataset

As an example, we begin by creating an artificial dataset that is stored in a csv (comma-separated values) file `../data/house_tiny.csv`. Data stored in other formats may be processed in similar ways. The following `mkdir_if_not_exist` function ensures that the directory `../data` exists. Note that the comment `#@save` is a special mark where the following function, class, or statements are saved in the `d2l` package so later they can be directly invoked (e.g., `d2l.mkdir_if_not_exist(path)`) without being redefined.

```
import os

def mkdir_if_not_exist(path):  #@save
    """Make a directory if it does not exist."""
    if not isinstance(path, str):
```

```
data_file = '../data/house_tiny.csv'
mkdir_if_not_exist('../data')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # Column names
    f.write('NA,Pave,127500\n')  # Each row represents a data example
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

To load the raw dataset from the created csv file, we import the `pandas` package and invoke the `read_csv` function. This dataset has four rows and three columns, where each row describes the number of rooms ("NumRooms"), the alley type ("Alley"), and the price ("Price") of a house.

```
# If pandas is not installed, just uncomment the following line:
# !pip install pandas
import pandas as pd

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

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2 Handling Missing Data

Note that "NaN" entries are missing values. To handle missing data, typical methods include *imputation* and *deletion*, where imputation replaces missing values with substituted ones, while deletion ignores missing values. Here we will consider imputation.

By integer-location based indexing (`iloc`), we split `data` into `inputs` and `outputs`, where the former takes the first two columns while the latter only keeps the last column. For numerical values in `inputs` that are missing, we replace the "NaN" entries with the mean value of the same column.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
```

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

2.2.3 Conversion to the Tensor Format

Now that all the entries in `inputs` and `outputs` are numerical, they can be converted to the tensor format. Once data are in this format, they can be further manipulated with those tensor functionalities that we have introduced in Section 2.1.

```
from mxnet import np

X, y = np.array(inputs.values), np.array(outputs.values)
X, y
```

```
(array([[3., 1., 0.],
        [2., 0., 1.],
        [4., 0., 1.],
        [3., 0., 1.]], dtype=float64),
 array([127500, 106000, 178100, 140000], dtype=int64))
```

Summary

- Like many other extension packages in the vast ecosystem of Python, `pandas` can work together with tensors.
- Imputation and deletion can be used to handle missing data.

Exercises

Create a raw dataset with more rows and columns.

1. Delete the column with the most missing values.
2. Convert the preprocessed dataset to the tensor format.

2.3 Linear Algebra

Now that you can store and manipulate data, let us briefly review the subset of basic linear algebra that you will need to understand and implement most of models covered in this book. Below, we introduce the basic mathematical objects, arithmetic, and operations in linear algebra, expressing each of them through mathematical notation and the corresponding implementation in code.

2.3.1 Scalars

If you never studied linear algebra or machine learning, then your past experience with math probably consisted of thinking about one number at a time. And, if you ever balanced a checkbook or even paid for dinner at a restaurant then you already know how to do basic things like adding and multiplying pairs of numbers. For example, the temperature in Palo Alto is 52 degrees Fahrenheit. Formally, we call values consisting of just one numerical quantity *scalars*. If you wanted to convert this value to Celsius (the metric system's more sensible temperature scale), you would evaluate the expression $c = \frac{5}{9}(f - 32)$, setting f to 52. In this equation, each of the terms—5, 9, and 32—are scalar values. The placeholders c and f are called *variables* and they represent unknown scalar values.

In this book, we adopt the mathematical notation where scalar variables are denoted by ordinary lower-cased letters (e.g., x, y, and z). We denote the space of all (continuous) *real-valued* scalars by $\mathbb{R}$. For expedience, we will punt on rigorous definitions of what precisely *space* is, but just remember for now that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ can be pronounced "in" and simply denotes membership in a set. Analogously, we could write $x, y \in \{0, 1\}$ to state that x and y are numbers whose value can only be 0 or 1.

A scalar is represented by a tensor with just one element. In the next snippet, we instantiate two scalars and perform some familiar arithmetic operations with them, namely addition, multiplication, division, and exponentiation.

```
from mxnet import np, npx
npx.set_np()

x = np.array(3.0)
y = np.array(2.0)

x + y, x * y, x / y, x ** y
```

```
(array(5.), array(6.), array(1.5), array(9.))
```

2.3.2 Vectors

minutes of exercise per day, etc. In math notation, we will usually denote vectors as bold-faced, lower-cased letters (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).

We work with vectors via one-dimensional tensors. In general tensors can have arbitrary lengths, subject to the memory limits of your machine.

```
x = np.arange(4)
x
```

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

We can refer to any element of a vector by using a subscript. For example, we can refer to the ith element of $\mathbf{x}$ by x_i. Note that the element x_i is a scalar, so we do not bold-face the font when referring to it. Extensive literature considers column vectors to be the default orientation of vectors, so does this book. In math, a vector $\mathbf{x}$ can be written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.3.1}$$

where $x_1, \ldots, x_n$ are elements of the vector. In code, we access any element by indexing into the tensor.

```
x[3]
```

```
array(3.)
```

Length, Dimensionality, and Shape

Let us revisit some concepts from Section 2.1. A vector is just an array of numbers. And just as every array has a length, so does every vector. In math notation, if we want to say that a vector $\mathbf{x}$ consists of n real-valued scalars, we can express this as $\mathbf{x} \in \mathbb{R}^n$. The length of a vector is commonly called the *dimension* of the vector.

As with an ordinary Python array, we can access the length of a tensor by calling Python's built-in `len()` function.

```
len(x)
```

```
4
```

```
(4,)
```

Note that the word "dimension" tends to get overloaded in these contexts and this tends to confuse people. To clarify, we use the dimensionality of a *vector* or an *axis* to refer to its length, i.e., the number of elements of a vector or an axis. However, we use the dimensionality of a tensor to refer to the number of axes that a tensor has. In this sense, the dimensionality of some axis of a tensor will be the length of that axis.

2.3.3 Matrices

Just as vectors generalize scalars from order zero to order one, matrices generalize vectors from order one to order two. Matrices, which we will typically denote with bold-faced, capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), are represented in code as tensors with two axes.

In math notation, we use $\mathbf{A} \in \mathbb{R}^{m \times n}$ to express that the matrix $\mathbf{A}$ consists of m rows and n columns of real-valued scalars. Visually, we can illustrate any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a table, where each element a_{ij} belongs to the ith row and j^{th} column:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.3.2}$$

For any $\mathbf{A} \in \mathbb{R}^{m \times n}$, the shape of $\mathbf{A}$ is (m, n) or $m \times n$. Specifically, when a matrix has the same number of rows and columns, its shape becomes a square; thus, it is called a *square matrix*.

We can create an $m \times n$ matrix by specifying a shape with two components m and n when calling any of our favorite functions for instantiating a tensor.

```
A = np.arange(20).reshape(5, 4)
A
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

We can access the scalar element a_{ij} of a matrix $\mathbf{A}$ in :eqref:`eq_matrix_def` by specifying the indices for the row (i) and column (j), such as $[\mathbf{A}]_{ij}$. When the scalar elements of a matrix $\mathbf{A}$, such as in :eqref:`eq_matrix_def`, are not given, we may simply use the lower-case letter of the matrix $\mathbf{A}$ with the index subscript, a_{ij}, to refer to $[\mathbf{A}]_{ij}$. To keep notation simple, commas are inserted to separate indices only when necessary, such as $a_{2,3j}$ and $[\mathbf{A}]_{2i-1,3}$.

Now we access a matrix's transpose in code.

```
A.T
```

```
array([[ 0.,  4.,  8., 12., 16.],
       [ 1.,  5.,  9., 13., 17.],
       [ 2.,  6., 10., 14., 18.],
       [ 3.,  7., 11., 15., 19.]])
```

As a special type of the square matrix, a *symmetric matrix* $\mathbf{A}$ is equal to its transpose: $\mathbf{A} = \mathbf{A}^\top$. Here we define a symmetric matrix B.

```
B = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

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

Now we compare B with its transpose.

```
B == B.T
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Matrices are useful data structures: they allow us to organize data that have different modalities of variation. For example, rows in our matrix might correspond to different houses (data examples), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 2.2. Thus, although the default orientation of a single vector is a column vector, in a matrix that represents a tabular dataset, it is more conventional to treat each data example as a row vector in the matrix. And, as we will see in later chapters, this convention will enable common deep learning practices. For example, along the outermost axis of a tensor, we can access or enumerate minibatches of data examples, or just data examples if no minibatch exists.

2.3.4 Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes. Tensors ("tensors" in this subsection refer to algebraic objects) give us a

```
X = np.arange(24).reshape(2, 3, 4)
X
```

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

       [[12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

2.3.5 Basic Properties of Tensor Arithmetic

Scalars, vectors, matrices, and tensors ("tensors" in this subsection refer to algebraic objects) of an arbitrary number of axes have some nice properties that often come in handy. For example, you might have noticed from the definition of an elementwise operation that any elementwise unary operation does not change the shape of its operand. Similarly, given any two tensors with the same shape, the result of any binary elementwise operation will be a tensor of that same shape. For example, adding two matrices of the same shape performs elementwise addition over these two matrices.

```
A = np.arange(20).reshape(5, 4)
B = A.copy()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]]),
 array([[ 0.,  2.,  4.,  6.],
        [ 8., 10., 12., 14.],
        [16., 18., 20., 22.],
        [24., 26., 28., 30.],
        [32., 34., 36., 38.]]))
```

Specifically, elementwise multiplication of two matrices is called their *Hadamard product* (math notation $\odot$). Consider matrix $\mathbf{B} \in \mathbb{R}^{m \times n}$ whose element of row i and column j is b_{ij}. The Hadamard product of matrices $\mathbf{A}$ (defined in :eqref:`eq_matrix_def`) and $\mathbf{B}$

$$\begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \end{bmatrix}$$

```
array([[  0.,   1.,   4.,   9.],
       [ 16.,  25.,  36.,  49.],
       [ 64.,  81., 100., 121.],
       [144., 169., 196., 225.],
       [256., 289., 324., 361.]])
```

Multiplying or adding a tensor by a scalar also does not change the shape of the tensor, where each element of the operand tensor will be added or multiplied by the scalar.

```
a = 2
X = np.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(array([[[ 2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9.],
         [10., 11., 12., 13.]],

        [[14., 15., 16., 17.],
         [18., 19., 20., 21.],
         [22., 23., 24., 25.]]]),
 (2, 3, 4))
```

2.3.6 Reduction

One useful operation that we can perform with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the $\sum$ symbol. To express the sum of the elements in a vector $\mathbf{x}$ of length d, we write $\sum_{i=1}^{d} x_i$. In code, we can just call the function for calculating the sum.

```
x = np.arange(4)
x, x.sum()
```

```
(array([0., 1., 2., 3.]), array(6.))
```

We can express sums over the elements of tensors of arbitrary shape. For example, the sum of the elements of an $m \times n$ matrix $\mathbf{A}$ could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum()
```

```
((5, 4), array(190.))
```

```
(array([40., 45., 50., 55.]), (4,))
```

Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns. Thus, the dimension of axis 1 of the input is lost in the output shape.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(array([ 6., 22., 38., 54., 70.]), (5,))
```

Reducing a matrix along both rows and columns via summation is equivalent to summing up all the elements of the matrix.

```
A.sum(axis=[0, 1])  # Same as `A.sum()`
```

```
array(190.)
```

A related quantity is the *mean*, which is also called the *average*. We calculate the mean by dividing the sum by the total number of elements. In code, we could just call the function for calculating the mean on tensors of arbitrary shape.

```
A.mean(), A.sum() / A.size
```

```
(array(9.5), array(9.5))
```

Likewise, the function for calculating the mean can also reduce a tensor along the specified axes.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(array([ 8.,  9., 10., 11.]), array([ 8.,  9., 10., 11.]))
```

Non-Reduction Sum

However, sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean.

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

```
A / sum_A
```

```
array([[0.        , 0.16666667, 0.33333334, 0.5       ],
       [0.18181819, 0.22727273, 0.27272728, 0.3181818 ],
       [0.21052632, 0.23684211, 0.2631579 , 0.28947368],
       [0.22222222, 0.24074075, 0.25925925, 0.2777778 ],
       [0.22857143, 0.24285714, 0.25714287, 0.27142859]])
```

If we want to calculate the cumulative sum of elements of `A` along some axis, say `axis=0` (row by row), we can call the `cumsum` function. This function will not reduce the input tensor along any axis.

```
A.cumsum(axis=0)
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  6.,  8., 10.],
       [12., 15., 18., 21.],
       [24., 28., 32., 36.],
       [40., 45., 50., 55.]])
```

2.3.7 Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra probably would not deserve its own section. However, one of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (or $\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = np.ones(4)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2., 3.]), array([1., 1., 1., 1.]), array(6.))
```

Note that we can express the dot product of two vectors equivalently by performing an elementwise multiplication and then a sum:

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

```
array(6.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted

2.3.8 Matrix-Vector Products

Now that we know how to calculate dot products, we can begin to understand *matrix-vector products*. Recall the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vector $\mathbf{x} \in \mathbb{R}^n$ defined and visualized in (2.3.2) and (2.3.1) respectively. Let us start off by visualizing the matrix $\mathbf{A}$ in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{2.3.5}$$

where each $\mathbf{a}_i^\top \in \mathbb{R}^n$ is a row vector representing the ith row of the matrix $\mathbf{A}$. The matrix-vector product $\mathbf{Ax}$ is simply a column vector of length m, whose i^{th} element is the dot product $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{2.3.6}$$

We can think of multiplication by a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a transformation that projects vectors from $\mathbb{R}^n$ to $\mathbb{R}^m$. These transformations turn out to be remarkably useful. For example, we can represent rotations as multiplications by a square matrix. As we will see in subsequent chapters, we can also use matrix-vector products to describe the most intensive calculations required when computing each layer in a neural network given the values of the previous layer.

Expressing matrix-vector products in code with tensors, we use the same `dot` function as for dot products. When we call `np.dot(A, x)` with a matrix `A` and a vector `x`, the matrix-vector product is performed. Note that the column dimension of `A` (its length along axis 1) must be the same as the dimension of `x` (its length).

```
A.shape, x.shape, np.dot(A, x)
```

```
((5, 4), (4,), array([ 14.,  38.,  62.,  86., 110.]))
```

2.3.9 Matrix-Matrix Multiplication

If you have gotten the hang of dot products and matrix-vector products, then *matrix-matrix multiplication* should be straightforward.

Say that we have two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ \end{bmatrix} \qquad \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ \end{bmatrix}$$

easiest to think of $\mathbf{A}$ in terms of its row vectors and $\mathbf{B}$ in terms of its column vectors:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{2.3.8}$$

Then the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$ is produced as we simply compute each element c_{ij} as the dot product $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

We can think of the matrix-matrix multiplication $\mathbf{AB}$ as simply performing m matrix-vector products and stitching the results together to form an $n \times m$ matrix. In the following snippet, we perform matrix multiplication on `A` and `B`. Here, `A` is a matrix with 5 rows and 4 columns, and `B` is a matrix with 4 rows and 3 columns. After multiplication, we obtain a matrix with 5 rows and 3 columns.

```
B = np.ones(shape=(4, 3))
np.dot(A, B)
```

```
array([[ 6.,  6.,  6.],
       [22., 22., 22.],
       [38., 38., 38.],
       [54., 54., 54.],
       [70., 70., 70.]])
```

Matrix-matrix multiplication can be simply called *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.10 Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* a vector is. The notion of *size* under consideration here concerns not dimensionality but rather the magnitude of the components.

In linear algebra, a vector norm is a function f that maps a vector to a scalar, satisfying a handful of properties. Given any vector $\mathbf{x}$, the first property says that if we scale all the elements of a vector by a constant factor α, its norm also scales by the *absolute value* of the same constant factor:

That makes sense, as in most contexts the smallest *size* for anything is 0. The final property requires that the smallest norm is achieved and only achieved by a vector consisting of all zeros.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

You might notice that norms sound a lot like measures of distance. And if you remember Euclidean distances (think Pythagoras' theorem) from grade school, then the concepts of non-negativity and the triangle inequality might ring a bell. In fact, the Euclidean distance is a norm: specifically it is the L_2 norm. Suppose that the elements in the n-dimensional vector $\mathbf{x}$ are $x_1, \ldots, x_n$. The L_2 *norm* of $\mathbf{x}$ is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{2.3.14}$$

where the subscript 2 is often omitted in L_2 norms, i.e., $\|\mathbf{x}\|$ is equivalent to $\|\mathbf{x}\|_2$. In code, we can calculate the L_2 norm of a vector as follows.

```
u = np.array([3, -4])
np.linalg.norm(u)
```

```
array(5.)
```

In deep learning, we work more often with the squared L_2 norm. You will also frequently encounter the L_1 *norm*, which is expressed as the sum of the absolute values of the vector elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|. \tag{2.3.15}$$

As compared with the L_2 norm, it is less influenced by outliers. To calculate the L_1 norm, we compose the absolute value function with a sum over the elements.

```
np.abs(u).sum()
```

```
array(7.)
```

Both the L_2 norm and the L_1 norm are special cases of the more general L_p *norm*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{2.3.16}$$

Analogous to L_2 norms of vectors, the *Frobenius norm* of a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ is the square root of the sum of the squares of the matrix elements:

```
np.linalg.norm(np.ones((4, 9)))
```

```
array(6.)
```

Norms and Objectives

While we do not want to get too far ahead of ourselves, we can plant some intuition already about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *minimize* the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, the objectives, perhaps the most important components of deep learning algorithms (besides the data), are expressed as norms.

2.3.11 More on Linear Algebra

In just this section, we have taught you all the linear algebra that you will need to understand a remarkable chunk of modern deep learning. There is a lot more to linear algebra and a lot of that mathematics is useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you will be much more inclined to learn more mathematics once you have gotten your hands dirty deploying useful machine learning models on real datasets. So while we reserve the right to introduce more mathematics much later on, we will wrap up this section here.

If you are eager to learn more about linear algebra, you may refer to either the online appendix on linear algebraic operations[43] or other excellent resources (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

Summary

- Scalars, vectors, matrices, and tensors are basic mathematical objects in linear algebra.
- Vectors generalize scalars, and matrices generalize vectors.
- Scalars, vectors, matrices, and tensors have zero, one, two, and an arbitrary number of axes, respectively.
- A tensor can be reduced along the specified axes by `sum` and `mean`.

Exercises

1. Prove that the transpose of a matrix $\mathbf{A}$'s transpose is $\mathbf{A}$: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, show that the sum of transposes is equal to the transpose of a sum: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Given any square matrix $\mathbf{A}$, is $\mathbf{A} + \mathbf{A}^\top$ always symmetric? Why?
4. We defined the tensor `X` of shape (2, 3, 4) in this section. What is the output of `len(X)`?
5. For a tensor `X` of arbitrary shape, does `len(X)` always correspond to the length of a certain axis of `X`? What is that axis?
6. Run `A / A.sum(axis=1)` and see what happens. Can you analyze the reason?
7. When traveling between two points in Manhattan, what is the distance that you need to cover in terms of the coordinates, i.e., in terms of avenues and streets? Can you travel diagonally?
8. Consider a tensor with shape (2, 3, 4). What are the shapes of the summation outputs along axis 0, 1, and 2?
9. Feed a tensor with 3 or more axes to the `linalg.norm` function and observe its output. What does this function compute for tensors of arbitrary shape?

Discussions[44]

2.4 Calculus

Finding the area of a polygon had remained mysterious until at least 2,500 years ago, when ancient Greeks divided a polygon into triangles and summed their areas. To find the area of curved shapes, such as a circle, ancient Greeks inscribed polygons in such shapes. As shown in Fig. 2.4.1, an inscribed polygon with more sides of equal length better approximates the circle. This process is also known as the *method of exhaustion*.

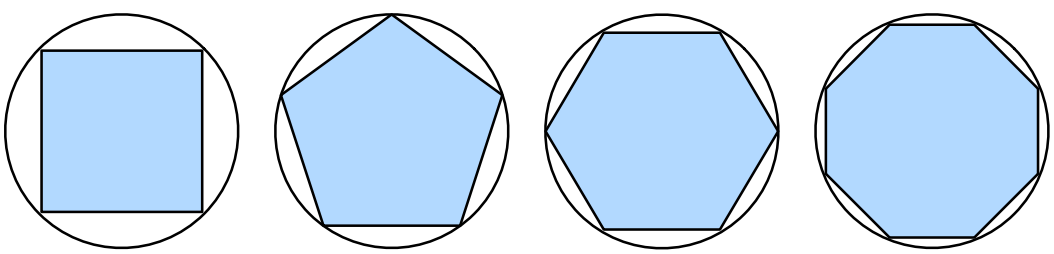


Fig. 2.4.1: Find the area of a circle with the method of exhaustion.

In fact, the method of exhaustion is where *integral calculus* (will be described in Section 18.5) orig

that answers the question "how *bad* is our model?" This question is more subtle than it appears. Ultimately, what we really care about is producing a model that performs well on data that we have never seen before. But we can only fit the model to data that we can actually see. Thus we can decompose the task of fitting models into two key concerns: i) *optimization*: the process of fitting our models to observed data; ii) *generalization*: the mathematical principles and practitioners' wisdom that guide as to how to produce models whose validity extends beyond the exact set of data examples used to train them.

To help you understand optimization problems and methods in later chapters, here we give a very brief primer on differential calculus that is commonly used in deep learning.

2.4.1 Derivatives and Differentiation

We begin by addressing the calculation of derivatives, a crucial step in nearly all deep learning optimization algorithms. In deep learning, we typically choose loss functions that are differentiable with respect to our model's parameters. Put simply, this means that for each parameter, we can determine how rapidly the loss would increase or decrease, were we to *increase* or *decrease* that parameter by an infinitesimally small amount.

Suppose that we have a function $f : \mathbb{R} \to \mathbb{R}$, whose input and output are both scalars. The *derivative* of f is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{2.4.1}$$

if this limit exists. If $f'(a)$ exists, f is said to be *differentiable* at a. If f is differentiable at every number of an interval, then this function is differentiable on this interval. We can interpret the derivative $f'(x)$ in (2.4.1) as the *instantaneous* rate of change of $f(x)$ with respect to x. The so-called instantaneous rate of change is based on the variation h in x, which approaches 0.

To illustrate derivatives, let us experiment with an example. Define $u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

def f(x):
    return 3 * x ** 2 - 4 * x
```

By setting $x = 1$ and letting h approach 0, the numerical result of $\frac{f(x+h)-f(x)}{h}$ in (2.4.1) approaches 2. Though this experiment is not a mathematical proof, we will see later that the derivative u' is 2 when $x = 1$.

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Let us familiarize ourselves with a few equivalent notations for derivatives. Given $y = f(x)$, where x and y are the independent variable and the dependent variable of the function f, respectively. The following expressions are equivalent:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

where symbols $\frac{d}{dx}$ and D are *differentiation operators* that indicate operation of *differentiation*. We can use the following rules to differentiate common functions:

- $DC = 0$ (C is a constant),
- $Dx^n = nx^{n-1}$ (the *power rule*, n is any real number),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

To differentiate a function that is formed from a few simpler functions such as the above common functions, the following rules can be handy for us. Suppose that functions f and g are both differentiable and C is a constant, we have the *constant multiple rule*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

the *sum rule*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

the *product rule*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

and the *quotient rule*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Now we can apply a few of the above rules to find $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$. Thus, by setting $x = 1$, we have $u' = 2$: this is supported by our earlier experiment in this section where

```python
def use_svg_display():  #@save
    """Use the svg format to display a plot in Jupyter."""
    display.set_matplotlib_formats('svg')
```

We define the `set_figsize` function to specify the figure sizes. Note that here we directly use `d2l.plt` since the import statement `from matplotlib import pyplot as plt` has been marked for being saved in the `d2l` package in the preface.

```python
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

The following `set_axes` function sets properties of axes of figures produced by `matplotlib`.

```python
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

With these three functions for figure configurations, we define the `plot` function to plot multiple curves succinctly since we will need to visualize many curves throughout the book.

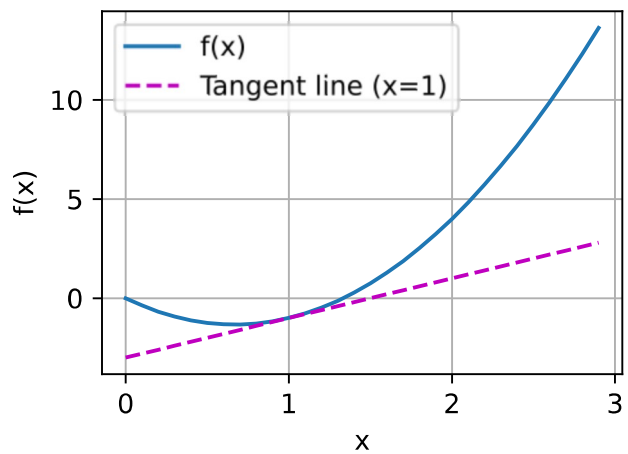
```python
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if `X` (tensor or list) has 1 axis
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))
```

(continued from previous page)

```
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Now we can plot the function $u = f(x)$ and its tangent line $y = 2x - 3$ at $x = 1$, where the coefficient 2 is the slope of the tangent line.

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```



2.4.2 Partial Derivatives

So far we have dealt with the differentiation of functions of just one variable. In deep learning, functions often depend on *many* variables. Thus, we need to extend the ideas of differentiation to these *multivariate* functions.

Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its ith parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

To calculate $\frac{\partial y}{\partial x_i}$, we can simply treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the deriva

2.4.3 Gradients

We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain the *gradient* vector of the function. Suppose that the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ and the output is a scalar. The gradient of the function $f(\mathbf{x})$ with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{2.4.9}$$

where $\nabla_{\mathbf{x}} f(\mathbf{x})$ is often replaced by $\nabla f(\mathbf{x})$ when there is no ambiguity.

Let $\mathbf{x}$ be an n-dimensional vector, the following rules are often used when differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$. As we will see later, gradients are useful for designing optimization algorithms in deep learning.

2.4.4 Chain Rule

However, such gradients can be hard to find. This is because multivariate functions in deep learning are often *composite*, so we may not apply any of the aforementioned rules to differentiate these functions. Fortunately, the *chain rule* enables us to differentiate composite functions.

Let us first consider functions of a single variable. Suppose that functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

Now let us turn our attention to a more general scenario where functions have an arbitrary number of variables. Suppose that the differentiable function y has variables $u_1, u_2, \ldots, u_m$, where each differentiable function u_i has variables $x_1, x_2, \ldots, x_n$. Note that y is a function of $x_1, x_2, \ldots, x_n$. Then the chain rule gives

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

for any $i = 1, 2, \ldots, n$.

- A gradient is a vector whose components are the partial derivatives of a multivariate function with respect to all its variables.
- The chain rule enables us to differentiate composite functions.

Exercises

1. Plot the function $y = f(x) = x^3 - \frac{1}{x}$ and its tangent line when $x = 1$.
2. Find the gradient of the function $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
3. What is the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|_2$?
4. Can you write out the chain rule for the case where $u = f(x, y, z)$ and $x = x(a, b)$, $y = y(a, b)$, and $z = z(a, b)$?

Discussions[45]

2.5 Automatic Differentiation

As we have explained in Section 2.4, differentiation is a crucial step in nearly all deep learning optimization algorithms. While the calculations for taking these derivatives are straightforward, requiring only some basic calculus, for complex models, working out the updates by hand can be a pain (and often error-prone).

Deep learning frameworks expedite this work by automatically calculating derivatives, i.e., *automatic differentiation*. In practice, based on our designed model the system builds a *computational graph*, tracking which data combined through which operations to produce the output. Automatic differentiation enables the system to subsequently backpropagate gradients. Here, *backpropagate* simply means to trace through the computational graph, filling in the partial derivatives with respect to each parameter.

```
from mxnet import autograd, np, npx
npx.set_np()
```

2.5.1 A Simple Example

As a toy example, say that we are interested in differentiating the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, let us create the variable x and assign it an initial value.

```
x = np.arange(4.0)
x
```

could quickly run out of memory. Note that a gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is itself vector-valued and has the same shape as $\mathbf{x}$.

```
# We allocate memory for a tensor's gradient by invoking `attach_grad`
x.attach_grad()
# After we calculate a gradient taken with respect to `x`, we will be able to
# access it via the `grad` attribute, whose values are initialized with 0s
x.grad
```

```
array([0., 0., 0., 0.])
```

Now let us calculate y.

```
# Place our code inside an `autograd.record` scope to build the computational
# graph
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

Since `x` is a vector of length 4, an inner product of `x` and `x` is performed, yielding the scalar output that we assign to `y`. Next, we can automatically calculate the gradient of `y` with respect to each component of `x` by calling the function for backpropagation and printing the gradient.

```
y.backward()
x.grad
```

```
array([ 0.,  4.,  8., 12.])
```

The gradient of the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. Let us quickly verify that our desired gradient was calculated correctly.

```
x.grad == 4 * x
```

```
array([ True,  True,  True,  True])
```

Now let us calculate another function of `x`.

```
with autograd.record():
    y = x.sum()
y.backward()
```

2.5.2 Backward for Non-Scalar Variables

Technically, when y is not a scalar, the most natural interpretation of the differentiation of a vector y with respect to a vector x is a matrix. For higher-order and higher-dimensional y and x, the differentiation result could be a high-order tensor.

However, while these more exotic objects do show up in advanced machine learning (including in deep learning), more often when we are calling backward on a vector, we are trying to calculate the derivatives of the loss functions for each constituent of a *batch* of training examples. Here, our intent is not to calculate the differentiation matrix but rather the sum of the partial derivatives computed individually for each example in the batch.

```
# When we invoke `backward` on a vector-valued variable `y` (function of `x`),
# a new scalar variable is created by summing the elements in `y`. Then the
# gradient of that scalar variable with respect to `x` is computed
with autograd.record():
    y = x * x  # `y` is a vector
y.backward()
x.grad  # Equals to y = sum(x * x)
```

```
array([0., 2., 4., 6.])
```

2.5.3 Detaching Computation

Sometimes, we wish to move some calculations outside of the recorded computational graph. For example, say that y was calculated as a function of x, and that subsequently z was calculated as a function of both y and x. Now, imagine that we wanted to calculate the gradient of z with respect to x, but wanted for some reason to treat y as a constant, and only take into account the role that x played after y was calculated.

Here, we can detach y to return a new variable u that has the same value as y but discards any information about how y was computed in the computational graph. In other words, the gradient will not flow backwards through u to x. Thus, the following backpropagation function computes the partial derivative of `z = u * x` with respect to x while treating u as a constant, instead of the partial derivative of `z = x * x * x` with respect to x.

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u
```

```
array([ True,  True,  True,  True])
```

2.5.4 Computing the Gradient of Python Control Flow

One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable. In the following snippet, note that the number of iterations of the `while` loop and the evaluation of the `if` statement both depend on the value of the input `a`.

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

Let us compute the gradient.

```
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

We can now analyze the `f` function defined above. Note that it is piecewise linear in its input `a`. In other words, for any `a` there exists some constant scalar `k` such that `f(a) = k * a`, where the value of `k` depends on the input `a`. Consequently `d / a` allows us to verify that the gradient is correct.

```
a.grad == d / a
```

```
array(True)
```

Summary

- Deep learning frameworks can automate the calculation of derivatives. To use it, we first attach gradients to those variables with respect to which we desire partial derivatives. We

Exercises

1. Why is the second derivative much more expensive to compute than the first derivative?
2. After running the function for backpropagation, immediately run it again and see what happens.
3. In the control flow example where we calculate the derivative of `d` with respect to `a`, what would happen if we changed the variable `a` to a random vector or matrix. At this point, the result of the calculation `f(a)` is no longer a scalar. What happens to the result? How do we analyze this?
4. Redesign an example of finding the gradient of the control flow. Run and analyze the result.
5. Let $f(x) = \sin(x)$. Plot $f(x)$ and $\frac{df(x)}{dx}$, where the latter is computed without exploiting that $f'(x) = \cos(x)$.

Discussions[46]

2.6 Probability

In some form or another, machine learning is all about making predictions. We might want to predict the *probability* of a patient suffering a heart attack in the next year, given their clinical history. In anomaly detection, we might want to assess how *likely* a set of readings from an airplane's jet engine would be, were it operating normally. In reinforcement learning, we want an agent to act intelligently in an environment. This means we need to think about the probability of getting a high reward under each of the available actions. And when we build recommender systems we also need to think about probability. For example, say *hypothetically* that we worked for a large online bookseller. We might want to estimate the probability that a particular user would buy a particular book. For this we need to use the language of probability. Entire courses, majors, theses, careers, and even departments, are devoted to probability. So naturally, our goal in this section is not to teach the whole subject. Instead we hope to get you off the ground, to teach you just enough that you can start building your first deep learning models, and to give you enough of a flavor for the subject that you can begin to explore it on your own if you wish.

We have already invoked probabilities in previous sections without articulating what precisely they are or giving a concrete example. Let us get more serious now by considering the first case: distinguishing cats and dogs based on photographs. This might sound simple but it is actually a formidable challenge. To start with, the difficulty of the problem may depend on the resolution of the image.

[46] https://discuss.d2l.ai/t/34

Fig. 2.6.1: Images of varying resolutions (10×10, 20×20, 40×40, 80×80, and 160×160 pixels).

As shown in Fig. 2.6.1, while it is easy for humans to recognize cats and dogs at the resolution of 160×160 pixels, it becomes challenging at 40×40 pixels and next to impossible at 10×10 pixels. In other words, our ability to tell cats and dogs apart at a large distance (and thus low resolution) might approach uninformed guessing. Probability gives us a formal way of reasoning about our level of certainty. If we are completely sure that the image depicts a cat, we say that the *probability* that the corresponding label y is "cat", denoted $P(y = \text{“cat”})$ equals 1. If we had no evidence to suggest that $y =$ "cat" or that $y =$ "dog", then we might say that the two possibilities were equally *likely* expressing this as $P(y = \text{“cat”}) = P(y = \text{“dog”}) = 0.5$. If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability $0.5 < P(y = \text{“cat”}) < 1$.

Now consider the second case: given some weather monitoring data, we want to predict the probability that it will rain in Taipei tomorrow. If it is summertime, the rain might come with probability 0.5.

In both cases, we have some value of interest. And in both cases we are uncertain about the outcome. But there is a key difference between the two cases. In this first case, the image is in fact either a dog or a cat, and we just do not know which. In the second case, the outcome may actually be a random event, if you believe in such things (and most physicists do). So probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in a broad set of contexts.

2.6.1 Basic Probability Theory

One natural approach for each value is to take the individual count for that value and to divide it by the total number of tosses. This gives us an *estimate* of the probability of a given *event*. The *law of large numbers* tell us that as the number of tosses grows this estimate will draw closer and closer to the true underlying probability. Before going into the details of what is going here, let us try it out.

To start, let us import the necessary packages.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()
```

Next, we will want to be able to cast the die. In statistics we call this process of drawing examples from probability distributions *sampling*. The distribution that assigns probabilities to a number of discrete choices is called the *multinomial distribution*. We will give a more formal definition of *distribution* later, but at a high level, think of it as just an assignment of probabilities to events.

To draw a single sample, we simply pass in a vector of probabilities. The output is another vector of the same length: its value at index i is the number of times the sampling outcome corresponds to i.

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)
```

```
array([0, 0, 0, 1, 0, 0], dtype=int64)
```

If you run the sampler a bunch of times, you will find that you get out random values each time. As with estimating the fairness of a die, we often want to generate many samples from the same distribution. It would be unbearably slow to do this with a Python `for` loop, so the function we are using supports drawing multiple samples at once, returning an array of independent samples in any shape we might desire.

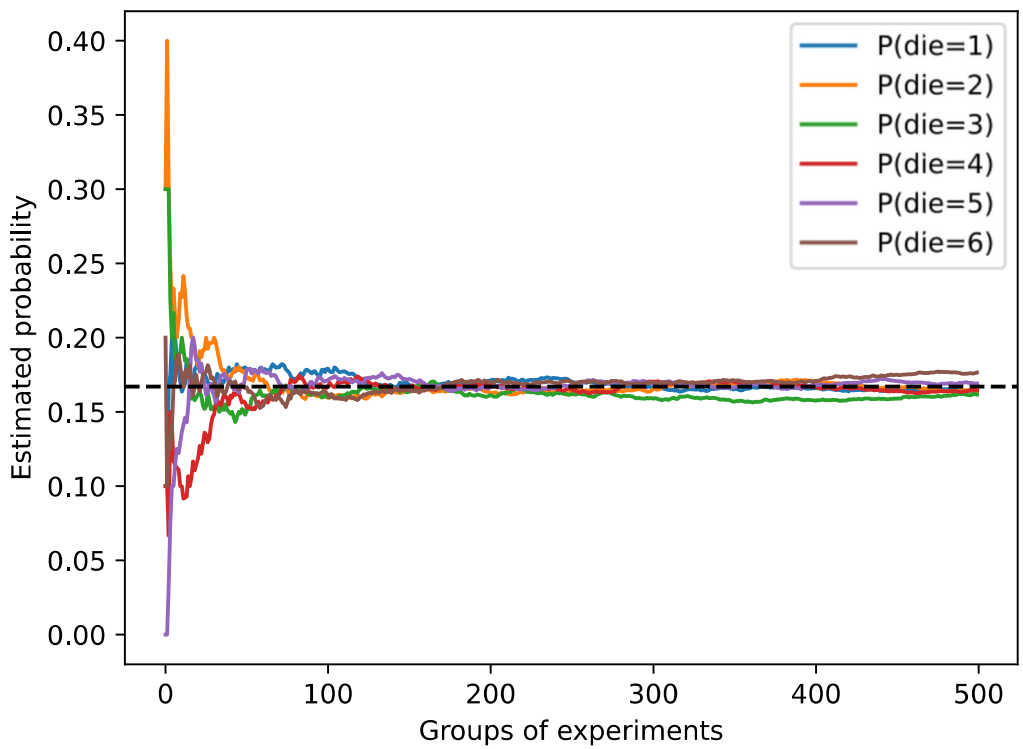
```
np.random.multinomial(10, fair_probs)
```

```
array([1, 1, 5, 1, 1, 1], dtype=int64)
```

Now that we know how to sample rolls of a die, we can simulate 1000 rolls. We can then go through and count, after each of the 1000 rolls, how many times each number was rolled. Specifically, we calculate the relative frequency as the estimate of the true probability.

```
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000
```

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].asnumpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



Each solid curve corresponds to one of the six values of the die and gives our estimated probability that the die turns up that value as assessed after each group of experiments. The dashed black line gives the true underlying probability. As we get more data by conducting more experiments, the 6 solid curves converge towards the true probability.

Axioms of Probability Theory

of an event $\mathcal{A}$ in the given sample space $\mathcal{S}$, denoted as $P(\mathcal{A})$, satisfies the following properties:

- For any event $\mathcal{A}$, its probability is never negative, i.e., $P(\mathcal{A}) \geq 0$;
- Probability of the entire sample space is 1, i.e., $P(\mathcal{S}) = 1$;
- For any countable sequence of events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ that are *mutually exclusive* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for all $i \neq j$), the probability that any happens is equal to the sum of their individual probabilities, i.e., $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

These are also the axioms of probability theory, proposed by Kolmogorov in 1933. Thanks to this axiom system, we can avoid any philosophical dispute on randomness; instead, we can reason rigorously with a mathematical language. For instance, by letting event $\mathcal{A}_1$ be the entire sample space and $\mathcal{A}_i = \emptyset$ for all $i > 1$, we can prove that $P(\emptyset) = 0$, i.e., the probability of an impossible event is 0.

Random Variables

In our random experiment of casting a die, we introduced the notion of a *random variable*. A random variable can be pretty much any quantity and is not deterministic. It could take one value among a set of possibilities in a random experiment. Consider a random variable X whose value is in the sample space $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ of rolling a die. We can denote the event "seeing a 5" as $\{X = 5\}$ or $X = 5$, and its probability as $P(\{X = 5\})$ or $P(X = 5)$. By $P(X = a)$, we make a distinction between the random variable X and the values (e.g., a) that X can take. However, such pedantry results in a cumbersome notation. For a compact notation, on one hand, we can just denote $P(X)$ as the *distribution* over the random variable X: the distribution tells us the probability that X takes any value. On the other hand, we can simply write $P(a)$ to denote the probability that a random variable takes the value a. Since an event in probability theory is a set of outcomes from the sample space, we can specify a range of values for a random variable to take. For example, $P(1 \leq X \leq 3)$ denotes the probability of the event $\{1 \leq X \leq 3\}$, which means $\{X = 1, 2, \text{or}, 3\}$. Equivalently, $P(1 \leq X \leq 3)$ represents the probability that the random variable X can take a value from $\{1, 2, 3\}$.

Note that there is a subtle difference between *discrete* random variables, like the sides of a die, and *continuous* ones, like the weight and the height of a person. There is little point in asking whether two people have exactly the same height. If we take precise enough measurements you will find that no two people on the planet have the exact same height. In fact, if we take a fine enough measurement, you will not have the same height when you wake up and when you go to sleep. So there is no purpose in asking about the probability that someone is 1.801392782910287192101967405274862 02 meters tall. Given the world population of humans the probability is virtually 0. It makes more sense in this case to ask whether someone's height falls into a given interval, say between 1.79 and 1.81 meters. In these cases we quantify the likelihood that we see a value as a *density*. The height of exactly 1.80 meters has no probability, but nonzero density. In the interval between any two different heights we have nonzero probability. In the rest of this section, we consider probability in discrete space. For probability over continuous random

2.6.2 Dealing with Multiple Random Variables

Very often, we will want to consider more than one random variable at a time. For instance, we may want to model the relationship between diseases and symptoms. Given a disease and a symptom, say "flu" and "cough", either may or may not occur in a patient with some probability. While we hope that the probability of both would be close to zero, we may want to estimate these probabilities and their relationships to each other so that we may apply our inferences to effect better medical care.

As a more complicated example, images contain millions of pixels, thus millions of random variables. And in many cases images will come with a label, identifying objects in the image. We can also think of the label as a random variable. We can even think of all the metadata as random variables such as location, time, aperture, focal length, ISO, focus distance, and camera type. All of these are random variables that occur jointly. When we deal with multiple random variables, there are several quantities of interest.

Joint Probability

The first is called the *joint probability* $P(A = a, B = b)$. Given any values a and b, the joint probability lets us answer, what is the probability that $A = a$ and $B = b$ simultaneously? Note that for any values a and b, $P(A = a, B = b) \leq P(A = a)$. This has to be the case, since for $A = a$ and $B = b$ to happen, $A = a$ has to happen *and* $B = b$ also has to happen (and vice versa). Thus, $A = a$ and $B = b$ cannot be more likely than $A = a$ or $B = b$ individually.

Conditional Probability

This brings us to an interesting ratio: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. We call this ratio a *conditional probability* and denote it by $P(B = b \mid A = a)$: it is the probability of $B = b$, provided that $A = a$ has occurred.

Bayes' theorem

Using the definition of conditional probabilities, we can derive one of the most useful and celebrated equations in statistics: *Bayes' theorem*. It goes as follows. By construction, we have the *multiplication rule* that $P(A, B) = P(B \mid A)P(A)$. By symmetry, this also holds for $P(A, B) = P(A \mid B)P(B)$. Assume that $P(B) > 0$. Solving for one of the conditional variables we get

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.1}$$

Note that here we use the more compact notation where $P(A, B)$ is a *joint distribution* and $P(A \mid B)$ is a *conditional distribution*. Such distributions can be evaluated for particular values $A = a, B = b$.

Marginalization

Bayes' theorem is very useful if we want to infer one thing from the other, say cause and effect, but we only know the properties in the reverse direction, as we will see later in this section. One important operation that we need, to make this work, is *marginalization*. It is the operation of determining $P(B)$ from $P(A, B)$. We can see that the probability of B amounts to accounting for all possible choices of A and aggregating the joint probabilities over all of them:

$$P(B) = \sum_A P(A, B), \tag{2.6.2}$$

which is also known as the *sum rule*. The probability or distribution as a result of marginalization is called a *marginal probability* or a *marginal distribution*.

Independence

Another useful property to check for is *dependence* vs. *independence*. Two random variables A and B being independent means that the occurrence of one event of A does not reveal any information about the occurrence of an event of B. In this case $P(B \mid A) = P(B)$. Statisticians typically express this as $A \perp B$. From Bayes' theorem, it follows immediately that also $P(A \mid B) = P(A)$. In all the other cases we call A and B dependent. For instance, two successive rolls of a die are independent. In contrast, the position of a light switch and the brightness in the room are not (they are not perfectly deterministic, though, since we could always have a broken light bulb, power failure, or a broken switch).

Since $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ is equivalent to $P(A, B) = P(A)P(B)$, two random variables are independent if and only if their joint distribution is the product of their individual distributions. Likewise, two random variables A and B are *conditionally independent* given another random variable C if and only if $P(A, B \mid C) = P(A \mid C)P(B \mid C)$. This is expressed as $A \perp B \mid C$.

Application

Let us put our skills to the test. Assume that a doctor administers an AIDS test to a patient. This test is fairly accurate and it fails only with 1% probability if the patient is healthy but reporting him as diseased. Moreover, it never fails to detect HIV if the patient actually has it. We use D_1 to indicate the diagnosis (1 if positive and 0 if negative) and H to denote the HIV status (1 if positive and 0 if negative). Table 2.6.1 lists such conditional probabilities.

Table 2.6.1: Conditional probability of $P(D_1 \mid H)$.

Conditional probability	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

apply marginalization and the multiplication rule to determine

$$\begin{aligned}
&P(D_1 = 1) \\
=&P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\
=&P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\
=&0.011485.
\end{aligned} \tag{2.6.3}$$

Thus, we get

$$\begin{aligned}
&P(H = 1 \mid D_1 = 1) \\
=&\frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}. \\
=&0.1306
\end{aligned} \tag{2.6.4}$$

In other words, there is only a 13.06% chance that the patient actually has AIDS, despite using a very accurate test. As we can see, probability can be counterintuitive.

What should a patient do upon receiving such terrifying news? Likely, the patient would ask the physician to administer another test to get clarity. The second test has different characteristics and it is not as good as the first one, as shown in Table 2.6.2.

Table 2.6.2: Conditional probability of $P(D_2 \mid H)$.

Conditional probability	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Unfortunately, the second test comes back positive, too. Let us work out the requisite probabilities to invoke Bayes' theorem by assuming the conditional independence:

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1 \mid H = 0) \\
=&P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0) \\
=&0.0003,
\end{aligned} \tag{2.6.5}$$

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1 \mid H = 1) \\
=&P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1) \\
=&0.98.
\end{aligned} \tag{2.6.6}$$

Now we can apply marginalization and the multiplication rule:

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1) \\
=&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\
=&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1) \\
=&0.00176955.
\end{aligned} \tag{2.6.7}$$

In the end, the probability of the patient having AIDS given both positive tests is

2.6.3 Expectation and Variance

To summarize key characteristics of probability distributions, we need some measures. The *expectation* (or average) of the random variable X is denoted as

$$E[X] = \sum_x xP(X = x). \tag{2.6.9}$$

When the input of a function $f(x)$ is a random variable drawn from the distribution P with different values x, the expectation of $f(x)$ is computed as

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x). \tag{2.6.10}$$

In many cases we want to measure by how much the random variable X deviates from its expectation. This can be quantified by the variance

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.11}$$

Its square root is called the *standard deviation*. The variance of a function of a random variable measures by how much the function deviates from the expectation of the function, as different values x of the random variable are sampled from its distribution:

$$\text{Var}[f(x)] = E\left[(f(x) - E[f(x)])^2\right]. \tag{2.6.12}$$

Summary

- We can sample from probability distributions.
- We can analyze multiple random variables using joint distribution, conditional distribution, Bayes' theorem, marginalization, and independence assumptions.
- Expectation and variance offer useful measures to summarize key characteristics of probability distributions.

Exercises

1. We conducted $m = 500$ groups of experiments where each group draws $n = 10$ samples. Vary m and n. Observe and analyze the experimental results.
2. Given two events with probability $P(\mathcal{A})$ and $P(\mathcal{B})$, compute upper and lower bounds on $P(\mathcal{A} \cup \mathcal{B})$ and $P(\mathcal{A} \cap \mathcal{B})$. (Hint: display the situation using a Venn Diagram[47].)
3. Assume that we have a sequence of random variables, say A, B, and C, where B only de-

2.7 Documentation

Due to constraints on the length of this book, we cannot possibly introduce every single MXNet function and class (and you probably would not want us to). The API documentation and additional tutorials and examples provide plenty of documentation beyond the book. In this section we provide you with some guidance to exploring the MXNet API.

2.7.1 Finding All the Functions and Classes in a Module

In order to know which functions and classes can be called in a module, we invoke the `dir` function. For instance, we can query all properties in the module for generating random numbers:

```
from mxnet import np
print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '_
↪_package__', '__spec__', '_mx_nd_np', 'absolute_import', 'choice', 'multinomial', 'normal',
↪ 'rand', 'randint', 'shuffle', 'uniform']
```

Generally, we can ignore functions that start and end with __ (special objects in Python) or functions that start with a single _(usually internal functions). Based on the remaining function or attribute names, we might hazard a guess that this module offers various methods for generating random numbers, including sampling from the uniform distribution (`uniform`), normal distribution (`normal`), and multinomial distribution (`multinomial`).

2.7.2 Finding the Usage of Specific Functions and Classes

For more specific instructions on how to use a given function or class, we can invoke the `help` function. As an example, let us explore the usage instructions for tensors' `ones` function.

```
help(np.ones)
```

```
Help on function ones in module mxnet.numpy:

ones(shape, dtype=<class 'numpy.float32'>, order='C', ctx=None)
    Return a new array of given shape and type, filled with ones.
    This function currently only supports storing multi-dimensional data
    in row-major (C-style).

    Parameters
```

```
    How to store multi-dimensional data in memory, currently only row-major
    (C-style) is supported.
ctx : Context, optional
    An optional device context (default is the current default context).

Returns
-------
out : ndarray
    Array of ones with the given shape, dtype, and ctx.

Examples
--------
>>> np.ones(5)
array([1., 1., 1., 1., 1.])

>>> np.ones((5,), dtype=int)
array([1, 1, 1, 1, 1], dtype=int64)

>>> np.ones((2, 1))
array([[1.],
       [1.]])

>>> s = (2,2)
>>> np.ones(s)
array([[1., 1.],
       [1., 1.]])
```

From the documentation, we can see that the `ones` function creates a new tensor with the specified shape and sets all the elements to the value of `1`. Whenever possible, you should run a quick test to confirm your interpretation:

```
np.ones(4)
```

```
array([1., 1., 1., 1.])
```

In the Jupyter notebook, we can use `?` to display the document in another window. For example, `list?` will create content that is almost identical to `help(list)`, displaying it in a new browser window. In addition, if we use two question marks, such as `list??`, the Python code implementing the function will also be displayed.

Summary

Exercises

1. Look up the documentation for any function or class in the deep learning framework. Can you also find the documentation on the official website of the framework?

Discussions[50]