

Dive into Deep Learning

ASTON ZHANG, ZACHARY C. LIPTON, MU LI, AND ALEXANDER J.
SMOLA

Contents

Preface	page xxv
Installation	xxxiv
Notation	xxxvii
1 Introduction	1
1.1 A Motivating Example	2
1.2 Key Components	4
1.3 Kinds of Machine Learning Problems	7
1.4 Roots	20
1.5 The Road to Deep Learning	22
1.6 Success Stories	25
1.7 The Essence of Deep Learning	27
1.8 Summary	29
1.9 Exercises	29
2 Preliminaries	30
2.1 Data Manipulation	30
2.1.1 Getting Started	30
2.1.2 Indexing and Slicing	33
2.1.3 Operations	34
2.1.4 Broadcasting	35
2.1.5 Saving Memory	36
2.1.6 Conversion to Other Python Objects	37
2.1.7 Summary	37
2.1.8 Exercises	38
2.2 Data Preprocessing	38
2.2.1 Reading the Dataset	38
2.2.2 Data Preparation	39
2.2.3 Conversion to the Tensor Format	40
2.2.4 Discussion	40
2.2.5 Exercises	40
2.3 Linear Algebra	41
2.3.1 Scalars	41

2.3.2	Vectors	42
2.3.3	Matrices	43
2.3.4	Tensors	44
2.3.5	Basic Properties of Tensor Arithmetic	45
2.3.6	Reduction	46
2.3.7	Non-Reduction Sum	47
2.3.8	Dot Products	48
2.3.9	Matrix–Vector Products	48
2.3.10	Matrix–Matrix Multiplication	49
2.3.11	Norms	50
2.3.12	Discussion	52
2.3.13	Exercises	53
2.4	Calculus	54
2.4.1	Derivatives and Differentiation	54
2.4.2	Visualization Utilities	56
2.4.3	Partial Derivatives and Gradients	58
2.4.4	Chain Rule	58
2.4.5	Discussion	59
2.4.6	Exercises	59
2.5	Automatic Differentiation	60
2.5.1	A Simple Function	60
2.5.2	Backward for Non-Scalar Variables	61
2.5.3	Detaching Computation	62
2.5.4	Gradients and Python Control Flow	63
2.5.5	Discussion	64
2.5.6	Exercises	64
2.6	Probability and Statistics	65
2.6.1	A Simple Example: Tossing Coins	66
2.6.2	A More Formal Treatment	68
2.6.3	Random Variables	69
2.6.4	Multiple Random Variables	70
2.6.5	An Example	73
2.6.6	Expectations	74
2.6.7	Discussion	76
2.6.8	Exercises	77
2.7	Documentation	78
2.7.1	Functions and Classes in a Module	78
2.7.2	Specific Functions and Classes	79
3	Linear Neural Networks for Regression	82
3.1	Linear Regression	82
3.1.1	Basics	83
3.1.2	Vectorization for Speed	88
3.1.3	The Normal Distribution and Squared Loss	88
3.1.4	Linear Regression as a Neural Network	90

3.1.5	Summary	91
3.1.6	Exercises	92
3.2	Object-Oriented Design for Implementation	93
3.2.1	Utilities	94
3.2.2	Models	96
3.2.3	Data	97
3.2.4	Training	97
3.2.5	Summary	98
3.2.6	Exercises	98
3.3	Synthetic Regression Data	99
3.3.1	Generating the Dataset	99
3.3.2	Reading the Dataset	100
3.3.3	Concise Implementation of the Data Loader	101
3.3.4	Summary	102
3.3.5	Exercises	102
3.4	Linear Regression Implementation from Scratch	103
3.4.1	Defining the Model	103
3.4.2	Defining the Loss Function	104
3.4.3	Defining the Optimization Algorithm	104
3.4.4	Training	105
3.4.5	Summary	107
3.4.6	Exercises	107
3.5	Concise Implementation of Linear Regression	108
3.5.1	Defining the Model	109
3.5.2	Defining the Loss Function	109
3.5.3	Defining the Optimization Algorithm	110
3.5.4	Training	110
3.5.5	Summary	111
3.5.6	Exercises	111
3.6	Generalization	112
3.6.1	Training Error and Generalization Error	113
3.6.2	Underfitting or Overfitting?	115
3.6.3	Model Selection	116
3.6.4	Summary	117
3.6.5	Exercises	117
3.7	Weight Decay	118
3.7.1	Norms and Weight Decay	119
3.7.2	High-Dimensional Linear Regression	120
3.7.3	Implementation from Scratch	121
3.7.4	Concise Implementation	122
3.7.5	Summary	124
3.7.6	Exercises	124
4	Linear Neural Networks for Classification	125
4.1	Softmax Regression	125

4.1.1	Classification	126
4.1.2	Loss Function	129
4.1.3	Information Theory Basics	130
4.1.4	Summary and Discussion	131
4.1.5	Exercises	132
4.2	The Image Classification Dataset	134
4.2.1	Loading the Dataset	134
4.2.2	Reading a Minibatch	135
4.2.3	Visualization	136
4.2.4	Summary	137
4.2.5	Exercises	137
4.3	The Base Classification Model	138
4.3.1	The Classifier Class	138
4.3.2	Accuracy	138
4.3.3	Summary	139
4.3.4	Exercises	139
4.4	Softmax Regression Implementation from Scratch	140
4.4.1	The Softmax	140
4.4.2	The Model	141
4.4.3	The Cross-Entropy Loss	141
4.4.4	Training	142
4.4.5	Prediction	143
4.4.6	Summary	143
4.4.7	Exercises	144
4.5	Concise Implementation of Softmax Regression	144
4.5.1	Defining the Model	145
4.5.2	Softmax Revisited	145
4.5.3	Training	146
4.5.4	Summary	146
4.5.5	Exercises	147
4.6	Generalization in Classification	147
4.6.1	The Test Set	148
4.6.2	Test Set Reuse	150
4.6.3	Statistical Learning Theory	151
4.6.4	Summary	153
4.6.5	Exercises	154
4.7	Environment and Distribution Shift	154
4.7.1	Types of Distribution Shift	155
4.7.2	Examples of Distribution Shift	157
4.7.3	Correction of Distribution Shift	159
4.7.4	A Taxonomy of Learning Problems	163
4.7.5	Fairness, Accountability, and Transparency in Machine Learning	164
4.7.6	Summary	165
4.7.7	Exercises	166

5	Multilayer Perceptrons	167
5.1	Multilayer Perceptrons	167
5.1.1	Hidden Layers	167
5.1.2	Activation Functions	171
5.1.3	Summary and Discussion	174
5.1.4	Exercises	175
5.2	Implementation of Multilayer Perceptrons	176
5.2.1	Implementation from Scratch	176
5.2.2	Concise Implementation	177
5.2.3	Summary	178
5.2.4	Exercises	179
5.3	Forward Propagation, Backward Propagation, and Computational Graphs	180
5.3.1	Forward Propagation	180
5.3.2	Computational Graph of Forward Propagation	181
5.3.3	Backpropagation	181
5.3.4	Training Neural Networks	183
5.3.5	Summary	183
5.3.6	Exercises	183
5.4	Numerical Stability and Initialization	184
5.4.1	Vanishing and Exploding Gradients	184
5.4.2	Parameter Initialization	187
5.4.3	Summary	188
5.4.4	Exercises	189
5.5	Generalization in Deep Learning	189
5.5.1	Revisiting Overfitting and Regularization	190
5.5.2	Inspiration from Nonparametrics	191
5.5.3	Early Stopping	192
5.5.4	Classical Regularization Methods for Deep Networks	193
5.5.5	Summary	193
5.5.6	Exercises	194
5.6	Dropout	194
5.6.1	Dropout in Practice	195
5.6.2	Implementation from Scratch	196
5.6.3	Concise Implementation	197
5.6.4	Summary	198
5.6.5	Exercises	198
5.7	Predicting House Prices on Kaggle	199
5.7.1	Downloading Data	199
5.7.2	Kaggle	200
5.7.3	Accessing and Reading the Dataset	201
5.7.4	Data Preprocessing	201
5.7.5	Error Measure	203
5.7.6	K -Fold Cross-Validation	204
5.7.7	Model Selection	204
5.7.8	Submitting Predictions on Kaggle	205

5.7.9	Summary and Discussion	206
5.7.10	Exercises	206
6	Builders' Guide	207
6.1	Layers and Modules	207
6.1.1	A Custom Module	209
6.1.2	The Sequential Module	211
6.1.3	Executing Code in the Forward Propagation Method	211
6.1.4	Summary	213
6.1.5	Exercises	213
6.2	Parameter Management	213
6.2.1	Parameter Access	214
6.2.2	Tied Parameters	215
6.2.3	Summary	216
6.2.4	Exercises	216
6.3	Parameter Initialization	216
6.3.1	Built-in Initialization	217
6.3.2	Summary	219
6.3.3	Exercises	219
6.4	Lazy Initialization	219
6.4.1	Summary	220
6.4.2	Exercises	221
6.5	Custom Layers	221
6.5.1	Layers without Parameters	221
6.5.2	Layers with Parameters	222
6.5.3	Summary	223
6.5.4	Exercises	223
6.6	File I/O	223
6.6.1	Loading and Saving Tensors	224
6.6.2	Loading and Saving Model Parameters	225
6.6.3	Summary	226
6.6.4	Exercises	226
6.7	GPUs	226
6.7.1	Computing Devices	227
6.7.2	Tensors and GPUs	228
6.7.3	Neural Networks and GPUs	230
6.7.4	Summary	231
6.7.5	Exercises	231
7	Convolutional Neural Networks	233
7.1	From Fully Connected Layers to Convolutions	234
7.1.1	Invariance	234
7.1.2	Constraining the MLP	235
7.1.3	Convolutions	237
7.1.4	Channels	238

7.1.5	Summary and Discussion	239
7.1.6	Exercises	239
7.2	Convolutions for Images	240
7.2.1	The Cross-Correlation Operation	240
7.2.2	Convolutional Layers	242
7.2.3	Object Edge Detection in Images	242
7.2.4	Learning a Kernel	244
7.2.5	Cross-Correlation and Convolution	245
7.2.6	Feature Map and Receptive Field	245
7.2.7	Summary	246
7.2.8	Exercises	247
7.3	Padding and Stride	247
7.3.1	Padding	248
7.3.2	Stride	250
7.3.3	Summary and Discussion	251
7.3.4	Exercises	251
7.4	Multiple Input and Multiple Output Channels	252
7.4.1	Multiple Input Channels	252
7.4.2	Multiple Output Channels	253
7.4.3	1×1 Convolutional Layer	255
7.4.4	Discussion	256
7.4.5	Exercises	256
7.5	Pooling	257
7.5.1	Maximum Pooling and Average Pooling	258
7.5.2	Padding and Stride	260
7.5.3	Multiple Channels	261
7.5.4	Summary	261
7.5.5	Exercises	262
7.6	Convolutional Neural Networks (LeNet)	262
7.6.1	LeNet	263
7.6.2	Training	265
7.6.3	Summary	266
7.6.4	Exercises	266
8	Modern Convolutional Neural Networks	268
8.1	Deep Convolutional Neural Networks (AlexNet)	269
8.1.1	Representation Learning	270
8.1.2	AlexNet	273
8.1.3	Training	276
8.1.4	Discussion	276
8.1.5	Exercises	277
8.2	Networks Using Blocks (VGG)	278
8.2.1	VGG Blocks	279
8.2.2	VGG Network	279
8.2.3	Training	281

8.2.4	Summary	282
8.2.5	Exercises	282
8.3	Network in Network (NiN)	283
8.3.1	NiN Blocks	283
8.3.2	NiN Model	284
8.3.3	Training	285
8.3.4	Summary	286
8.3.5	Exercises	286
8.4	Multi-Branch Networks (GoogLeNet)	287
8.4.1	Inception Blocks	287
8.4.2	GoogLeNet Model	288
8.4.3	Training	291
8.4.4	Discussion	291
8.4.5	Exercises	292
8.5	Batch Normalization	292
8.5.1	Training Deep Networks	293
8.5.2	Batch Normalization Layers	295
8.5.3	Implementation from Scratch	297
8.5.4	LeNet with Batch Normalization	298
8.5.5	Concise Implementation	299
8.5.6	Discussion	300
8.5.7	Exercises	301
8.6	Residual Networks (ResNet) and ResNeXt	302
8.6.1	Function Classes	302
8.6.2	Residual Blocks	304
8.6.3	ResNet Model	306
8.6.4	Training	308
8.6.5	ResNeXt	308
8.6.6	Summary and Discussion	310
8.6.7	Exercises	311
8.7	Densely Connected Networks (DenseNet)	312
8.7.1	From ResNet to DenseNet	312
8.7.2	Dense Blocks	313
8.7.3	Transition Layers	314
8.7.4	DenseNet Model	315
8.7.5	Training	315
8.7.6	Summary and Discussion	316
8.7.7	Exercises	316
8.8	Designing Convolution Network Architectures	317
8.8.1	The AnyNet Design Space	318
8.8.2	Distributions and Parameters of Design Spaces	320
8.8.3	RegNet	322
8.8.4	Training	323
8.8.5	Discussion	323
8.8.6	Exercises	324

9	Recurrent Neural Networks	325
9.1	Working with Sequences	327
9.1.1	Autoregressive Models	328
9.1.2	Sequence Models	330
9.1.3	Training	331
9.1.4	Prediction	333
9.1.5	Summary	335
9.1.6	Exercises	335
9.2	Converting Raw Text into Sequence Data	336
9.2.1	Reading the Dataset	336
9.2.2	Tokenization	337
9.2.3	Vocabulary	337
9.2.4	Putting It All Together	338
9.2.5	Exploratory Language Statistics	339
9.2.6	Summary	341
9.2.7	Exercises	342
9.3	Language Models	342
9.3.1	Learning Language Models	343
9.3.2	Perplexity	345
9.3.3	Partitioning Sequences	346
9.3.4	Summary and Discussion	347
9.3.5	Exercises	348
9.4	Recurrent Neural Networks	348
9.4.1	Neural Networks without Hidden States	349
9.4.2	Recurrent Neural Networks with Hidden States	349
9.4.3	RNN-Based Character-Level Language Models	351
9.4.4	Summary	352
9.4.5	Exercises	352
9.5	Recurrent Neural Network Implementation from Scratch	352
9.5.1	RNN Model	353
9.5.2	RNN-Based Language Model	354
9.5.3	Gradient Clipping	356
9.5.4	Training	357
9.5.5	Decoding	358
9.5.6	Summary	359
9.5.7	Exercises	359
9.6	Concise Implementation of Recurrent Neural Networks	360
9.6.1	Defining the Model	360
9.6.2	Training and Predicting	361
9.6.3	Summary	362
9.6.4	Exercises	362
9.7	Backpropagation Through Time	362
9.7.1	Analysis of Gradients in RNNs	362
9.7.2	Backpropagation Through Time in Detail	365
9.7.3	Summary	368

9.7.4	Exercises	368
10	Modern Recurrent Neural Networks	369
10.1	Long Short-Term Memory (LSTM)	370
10.1.1	Gated Memory Cell	370
10.1.2	Implementation from Scratch	373
10.1.3	Concise Implementation	375
10.1.4	Summary	376
10.1.5	Exercises	376
10.2	Gated Recurrent Units (GRU)	376
10.2.1	Reset Gate and Update Gate	377
10.2.2	Candidate Hidden State	378
10.2.3	Hidden State	378
10.2.4	Implementation from Scratch	379
10.2.5	Concise Implementation	380
10.2.6	Summary	381
10.2.7	Exercises	381
10.3	Deep Recurrent Neural Networks	382
10.3.1	Implementation from Scratch	383
10.3.2	Concise Implementation	384
10.3.3	Summary	385
10.3.4	Exercises	385
10.4	Bidirectional Recurrent Neural Networks	385
10.4.1	Implementation from Scratch	387
10.4.2	Concise Implementation	387
10.4.3	Summary	388
10.4.4	Exercises	388
10.5	Machine Translation and the Dataset	388
10.5.1	Downloading and Preprocessing the Dataset	389
10.5.2	Tokenization	390
10.5.3	Loading Sequences of Fixed Length	391
10.5.4	Reading the Dataset	392
10.5.5	Summary	393
10.5.6	Exercises	394
10.6	The Encoder–Decoder Architecture	394
10.6.1	Encoder	394
10.6.2	Decoder	395
10.6.3	Putting the Encoder and Decoder Together	395
10.6.4	Summary	396
10.6.5	Exercises	396
10.7	Sequence-to-Sequence Learning for Machine Translation	396
10.7.1	Teacher Forcing	397
10.7.2	Encoder	397
10.7.3	Decoder	399
10.7.4	Encoder–Decoder for Sequence-to-Sequence Learning	400

10.7.5	Loss Function with Masking	401
10.7.6	Training	401
10.7.7	Prediction	402
10.7.8	Evaluation of Predicted Sequences	403
10.7.9	Summary	404
10.7.10	Exercises	404
10.8	Beam Search	405
10.8.1	Greedy Search	405
10.8.2	Exhaustive Search	407
10.8.3	Beam Search	407
10.8.4	Summary	408
10.8.5	Exercises	408
11	Attention Mechanisms and Transformers	409
11.1	Queries, Keys, and Values	411
11.1.1	Visualization	413
11.1.2	Summary	414
11.1.3	Exercises	414
11.2	Attention Pooling by Similarity	415
11.2.1	Kernels and Data	415
11.2.2	Attention Pooling via Nadaraya–Watson Regression	417
11.2.3	Adapting Attention Pooling	418
11.2.4	Summary	419
11.2.5	Exercises	420
11.3	Attention Scoring Functions	420
11.3.1	Dot Product Attention	421
11.3.2	Convenience Functions	421
11.3.3	Scaled Dot Product Attention	423
11.3.4	Additive Attention	424
11.3.5	Summary	426
11.3.6	Exercises	426
11.4	The Bahdanau Attention Mechanism	427
11.4.1	Model	428
11.4.2	Defining the Decoder with Attention	428
11.4.3	Training	430
11.4.4	Summary	431
11.4.5	Exercises	432
11.5	Multi-Head Attention	432
11.5.1	Model	433
11.5.2	Implementation	433
11.5.3	Summary	435
11.5.4	Exercises	435
11.6	Self-Attention and Positional Encoding	435
11.6.1	Self-Attention	436
11.6.2	Comparing CNNs, RNNs, and Self-Attention	436

11.6.3	Positional Encoding	437
11.6.4	Summary	440
11.6.5	Exercises	440
11.7	The Transformer Architecture	440
11.7.1	Model	441
11.7.2	Positionwise Feed-Forward Networks	442
11.7.3	Residual Connection and Layer Normalization	443
11.7.4	Encoder	444
11.7.5	Decoder	445
11.7.6	Training	447
11.7.7	Summary	451
11.7.8	Exercises	451
11.8	Transformers for Vision	451
11.8.1	Model	452
11.8.2	Patch Embedding	453
11.8.3	Vision Transformer Encoder	453
11.8.4	Putting It All Together	454
11.8.5	Training	455
11.8.6	Summary and Discussion	455
11.8.7	Exercises	456
11.9	Large-Scale Pretraining with Transformers	456
11.9.1	Encoder-Only	457
11.9.2	Encoder-Decoder	459
11.9.3	Decoder-Only	461
11.9.4	Scalability	463
11.9.5	Large Language Models	465
11.9.6	Summary and Discussion	466
11.9.7	Exercises	467
12	Optimization Algorithms	468
12.1	Optimization and Deep Learning	468
12.1.1	Goal of Optimization	469
12.1.2	Optimization Challenges in Deep Learning	469
12.1.3	Summary	473
12.1.4	Exercises	473
12.2	Convexity	474
12.2.1	Definitions	474
12.2.2	Properties	476
12.2.3	Constraints	479
12.2.4	Summary	481
12.2.5	Exercises	482
12.3	Gradient Descent	482
12.3.1	One-Dimensional Gradient Descent	482
12.3.2	Multivariate Gradient Descent	486
12.3.3	Adaptive Methods	488

12.3.4	Summary	492
12.3.5	Exercises	492
12.4	Stochastic Gradient Descent	493
12.4.1	Stochastic Gradient Updates	493
12.4.2	Dynamic Learning Rate	495
12.4.3	Convergence Analysis for Convex Objectives	496
12.4.4	Stochastic Gradients and Finite Samples	498
12.4.5	Summary	499
12.4.6	Exercises	499
12.5	Minibatch Stochastic Gradient Descent	500
12.5.1	Vectorization and Caches	500
12.5.2	Minibatches	503
12.5.3	Reading the Dataset	504
12.5.4	Implementation from Scratch	504
12.5.5	Concise Implementation	507
12.5.6	Summary	509
12.5.7	Exercises	509
12.6	Momentum	510
12.6.1	Basics	510
12.6.2	Practical Experiments	514
12.6.3	Theoretical Analysis	516
12.6.4	Summary	518
12.6.5	Exercises	519
12.7	Adagrad	519
12.7.1	Sparse Features and Learning Rates	519
12.7.2	Preconditioning	520
12.7.3	The Algorithm	521
12.7.4	Implementation from Scratch	523
12.7.5	Concise Implementation	524
12.7.6	Summary	524
12.7.7	Exercises	525
12.8	RMSProp	525
12.8.1	The Algorithm	526
12.8.2	Implementation from Scratch	526
12.8.3	Concise Implementation	528
12.8.4	Summary	528
12.8.5	Exercises	529
12.9	Adadelta	529
12.9.1	The Algorithm	529
12.9.2	Implementation	530
12.9.3	Summary	531
12.9.4	Exercises	532
12.10	Adam	532
12.10.1	The Algorithm	532
12.10.2	Implementation	533

12.10.3	Yogi	534
12.10.4	Summary	535
12.10.5	Exercises	536
12.11	Learning Rate Scheduling	536
12.11.1	Toy Problem	537
12.11.2	Schedulers	539
12.11.3	Policies	540
12.11.4	Summary	545
12.11.5	Exercises	545
13	Computational Performance	547
13.1	Compilers and Interpreters	547
13.1.1	Symbolic Programming	548
13.1.2	Hybrid Programming	549
13.1.3	Hybridizing the Sequential Class	550
13.1.4	Summary	552
13.1.5	Exercises	552
13.2	Asynchronous Computation	552
13.2.1	Asynchrony via Backend	553
13.2.2	Barriers and Blockers	554
13.2.3	Improving Computation	555
13.2.4	Summary	555
13.2.5	Exercises	555
13.3	Automatic Parallelism	555
13.3.1	Parallel Computation on GPUs	556
13.3.2	Parallel Computation and Communication	557
13.3.3	Summary	558
13.3.4	Exercises	559
13.4	Hardware	559
13.4.1	Computers	560
13.4.2	Memory	561
13.4.3	Storage	562
13.4.4	CPUs	563
13.4.5	GPUs and other Accelerators	566
13.4.6	Networks and Buses	569
13.4.7	More Latency Numbers	570
13.4.8	Summary	571
13.4.9	Exercises	571
13.5	Training on Multiple GPUs	572
13.5.1	Splitting the Problem	573
13.5.2	Data Parallelism	574
13.5.3	A Toy Network	575
13.5.4	Data Synchronization	576
13.5.5	Distributing Data	577
13.5.6	Training	578

13.5.7	Summary	580
13.5.8	Exercises	580
13.6	Concise Implementation for Multiple GPUs	581
13.6.1	A Toy Network	581
13.6.2	Network Initialization	582
13.6.3	Training	582
13.6.4	Summary	583
13.6.5	Exercises	584
13.7	Parameter Servers	584
13.7.1	Data-Parallel Training	584
13.7.2	Ring Synchronization	586
13.7.3	Multi-Machine Training	588
13.7.4	Key-Value Stores	589
13.7.5	Summary	591
13.7.6	Exercises	591
14	Computer Vision	592
14.1	Image Augmentation	592
14.1.1	Common Image Augmentation Methods	593
14.1.2	Training with Image Augmentation	596
14.1.3	Summary	599
14.1.4	Exercises	599
14.2	Fine-Tuning	600
14.2.1	Steps	600
14.2.2	Hot Dog Recognition	601
14.2.3	Summary	605
14.2.4	Exercises	606
14.3	Object Detection and Bounding Boxes	606
14.3.1	Bounding Boxes	607
14.3.2	Summary	609
14.3.3	Exercises	609
14.4	Anchor Boxes	609
14.4.1	Generating Multiple Anchor Boxes	610
14.4.2	Intersection over Union (IoU)	612
14.4.3	Labeling Anchor Boxes in Training Data	613
14.4.4	Predicting Bounding Boxes with Non-Maximum Suppression	619
14.4.5	Summary	622
14.4.6	Exercises	623
14.5	Multiscale Object Detection	623
14.5.1	Multiscale Anchor Boxes	623
14.5.2	Multiscale Detection	625
14.5.3	Summary	626
14.5.4	Exercises	626
14.6	The Object Detection Dataset	627
14.6.1	Downloading the Dataset	627

14.6.2	Reading the Dataset	627
14.6.3	Demonstration	629
14.6.4	Summary	629
14.6.5	Exercises	630
14.7	Single Shot Multibox Detection	630
14.7.1	Model	630
14.7.2	Training	636
14.7.3	Prediction	638
14.7.4	Summary	639
14.7.5	Exercises	640
14.8	Region-based CNNs (R-CNNs)	642
14.8.1	R-CNNs	642
14.8.2	Fast R-CNN	643
14.8.3	Faster R-CNN	645
14.8.4	Mask R-CNN	646
14.8.5	Summary	647
14.8.6	Exercises	647
14.9	Semantic Segmentation and the Dataset	648
14.9.1	Image Segmentation and Instance Segmentation	648
14.9.2	The Pascal VOC2012 Semantic Segmentation Dataset	648
14.9.3	Summary	654
14.9.4	Exercises	654
14.10	Transposed Convolution	654
14.10.1	Basic Operation	654
14.10.2	Padding, Strides, and Multiple Channels	656
14.10.3	Connection to Matrix Transposition	657
14.10.4	Summary	659
14.10.5	Exercises	659
14.11	Fully Convolutional Networks	659
14.11.1	The Model	660
14.11.2	Initializing Transposed Convolutional Layers	662
14.11.3	Reading the Dataset	663
14.11.4	Training	664
14.11.5	Prediction	664
14.11.6	Summary	666
14.11.7	Exercises	666
14.12	Neural Style Transfer	666
14.12.1	Method	666
14.12.2	Reading the Content and Style Images	668
14.12.3	Preprocessing and Postprocessing	668
14.12.4	Extracting Features	669
14.12.5	Defining the Loss Function	670
14.12.6	Initializing the Synthesized Image	672
14.12.7	Training	673
14.12.8	Summary	674

14.12.9	Exercises	674
14.13	Image Classification (CIFAR-10) on Kaggle	674
14.13.1	Obtaining and Organizing the Dataset	675
14.13.2	Image Augmentation	678
14.13.3	Reading the Dataset	678
14.13.4	Defining the Model	679
14.13.5	Defining the Training Function	679
14.13.6	Training and Validating the Model	680
14.13.7	Classifying the Testing Set and Submitting Results on Kaggle	680
14.13.8	Summary	681
14.13.9	Exercises	682
14.14	Dog Breed Identification (ImageNet Dogs) on Kaggle	682
14.14.1	Obtaining and Organizing the Dataset	682
14.14.2	Image Augmentation	684
14.14.3	Reading the Dataset	685
14.14.4	Fine-Tuning a Pretrained Model	685
14.14.5	Defining the Training Function	686
14.14.6	Training and Validating the Model	687
14.14.7	Classifying the Testing Set and Submitting Results on Kaggle	688
14.14.8	Summary	688
14.14.9	Exercises	689
15	Natural Language Processing: Pretraining	690
15.1	Word Embedding (word2vec)	691
15.1.1	One-Hot Vectors Are a Bad Choice	691
15.1.2	Self-Supervised word2vec	691
15.1.3	The Skip-Gram Model	692
15.1.4	The Continuous Bag of Words (CBOW) Model	694
15.1.5	Summary	695
15.1.6	Exercises	695
15.2	Approximate Training	696
15.2.1	Negative Sampling	696
15.2.2	Hierarchical Softmax	698
15.2.3	Summary	699
15.2.4	Exercises	699
15.3	The Dataset for Pretraining Word Embeddings	699
15.3.1	Reading the Dataset	699
15.3.2	Subsampling	700
15.3.3	Extracting Center Words and Context Words	702
15.3.4	Negative Sampling	703
15.3.5	Loading Training Examples in Minibatches	704
15.3.6	Putting It All Together	705
15.3.7	Summary	706
15.3.8	Exercises	706
15.4	Pretraining word2vec	707

15.4.1	The Skip-Gram Model	707
15.4.2	Training	708
15.4.3	Applying Word Embeddings	711
15.4.4	Summary	711
15.4.5	Exercises	711
15.5	Word Embedding with Global Vectors (GloVe)	711
15.5.1	Skip-Gram with Global Corpus Statistics	712
15.5.2	The GloVe Model	713
15.5.3	Interpreting GloVe from the Ratio of Co-occurrence Probabilities	713
15.5.4	Summary	715
15.5.5	Exercises	715
15.6	Subword Embedding	715
15.6.1	The fastText Model	715
15.6.2	Byte Pair Encoding	716
15.6.3	Summary	719
15.6.4	Exercises	719
15.7	Word Similarity and Analogy	720
15.7.1	Loading Pretrained Word Vectors	720
15.7.2	Applying Pretrained Word Vectors	722
15.7.3	Summary	724
15.7.4	Exercises	724
15.8	Bidirectional Encoder Representations from Transformers (BERT)	724
15.8.1	From Context-Independent to Context-Sensitive	724
15.8.2	From Task-Specific to Task-Agnostic	725
15.8.3	BERT: Combining the Best of Both Worlds	725
15.8.4	Input Representation	726
15.8.5	Pretraining Tasks	728
15.8.6	Putting It All Together	731
15.8.7	Summary	732
15.8.8	Exercises	733
15.9	The Dataset for Pretraining BERT	733
15.9.1	Defining Helper Functions for Pretraining Tasks	734
15.9.2	Transforming Text into the Pretraining Dataset	736
15.9.3	Summary	738
15.9.4	Exercises	739
15.10	Pretraining BERT	739
15.10.1	Pretraining BERT	739
15.10.2	Representing Text with BERT	741
15.10.3	Summary	742
15.10.4	Exercises	743
16	Natural Language Processing: Applications	744
16.1	Sentiment Analysis and the Dataset	745
16.1.1	Reading the Dataset	745

16.1.2	Preprocessing the Dataset	746
16.1.3	Creating Data Iterators	747
16.1.4	Putting It All Together	747
16.1.5	Summary	748
16.1.6	Exercises	748
16.2	Sentiment Analysis: Using Recurrent Neural Networks	748
16.2.1	Representing Single Text with RNNs	749
16.2.2	Loading Pretrained Word Vectors	750
16.2.3	Training and Evaluating the Model	751
16.2.4	Summary	751
16.2.5	Exercises	752
16.3	Sentiment Analysis: Using Convolutional Neural Networks	752
16.3.1	One-Dimensional Convolutions	753
16.3.2	Max-Over-Time Pooling	754
16.3.3	The textCNN Model	755
16.3.4	Summary	758
16.3.5	Exercises	758
16.4	Natural Language Inference and the Dataset	759
16.4.1	Natural Language Inference	759
16.4.2	The Stanford Natural Language Inference (SNLI) Dataset	760
16.4.3	Summary	763
16.4.4	Exercises	763
16.5	Natural Language Inference: Using Attention	763
16.5.1	The Model	764
16.5.2	Training and Evaluating the Model	768
16.5.3	Summary	770
16.5.4	Exercises	770
16.6	Fine-Tuning BERT for Sequence-Level and Token-Level Applications	771
16.6.1	Single Text Classification	771
16.6.2	Text Pair Classification or Regression	772
16.6.3	Text Tagging	773
16.6.4	Question Answering	773
16.6.5	Summary	774
16.6.6	Exercises	774
16.7	Natural Language Inference: Fine-Tuning BERT	775
16.7.1	Loading Pretrained BERT	775
16.7.2	The Dataset for Fine-Tuning BERT	776
16.7.3	Fine-Tuning BERT	778
16.7.4	Summary	779
16.7.5	Exercises	779
17	Reinforcement Learning	781
17.1	Markov Decision Process (MDP)	782
17.1.1	Definition of an MDP	782
17.1.2	Return and Discount Factor	783

17.1.3	Discussion of the Markov Assumption	784
17.1.4	Summary	785
17.1.5	Exercises	785
17.2	Value Iteration	785
17.2.1	Stochastic Policy	785
17.2.2	Value Function	786
17.2.3	Action-Value Function	786
17.2.4	Optimal Stochastic Policy	787
17.2.5	Principle of Dynamic Programming	787
17.2.6	Value Iteration	788
17.2.7	Policy Evaluation	788
17.2.8	Implementation of Value Iteration	789
17.2.9	Summary	790
17.2.10	Exercises	791
17.3	Q-Learning	791
17.3.1	The Q-Learning Algorithm	791
17.3.2	An Optimization Problem Underlying Q-Learning	791
17.3.3	Exploration in Q-Learning	793
17.3.4	The “Self-correcting” Property of Q-Learning	793
17.3.5	Implementation of Q-Learning	794
17.3.6	Summary	795
17.3.7	Exercises	796
18	Gaussian Processes	797
18.1	Introduction to Gaussian Processes	798
18.1.1	Summary	807
18.1.2	Exercises	808
18.2	Gaussian Process Priors	809
18.2.1	Definition	809
18.2.2	A Simple Gaussian Process	810
18.2.3	From Weight Space to Function Space	811
18.2.4	The Radial Basis Function (RBF) Kernel	811
18.2.5	The Neural Network Kernel	813
18.2.6	Summary	814
18.2.7	Exercises	814
18.3	Gaussian Process Inference	815
18.3.1	Posterior Inference for Regression	815
18.3.2	Equations for Making Predictions and Learning Kernel Hyperparameters in GP Regression	817
18.3.3	Interpreting Equations for Learning and Predictions	817
18.3.4	Worked Example from Scratch	818
18.3.5	Making Life Easy with GPyTorch	822
18.3.6	Summary	825
18.3.7	Exercises	826

19	Hyperparameter Optimization	828
19.1	What Is Hyperparameter Optimization?	828
19.1.1	The Optimization Problem	829
19.1.2	Random Search	832
19.1.3	Summary	834
19.1.4	Exercises	835
19.2	Hyperparameter Optimization API	836
19.2.1	Searcher	836
19.2.2	Scheduler	837
19.2.3	Tuner	837
19.2.4	Bookkeeping the Performance of HPO Algorithms	838
19.2.5	Example: Optimizing the Hyperparameters of a Convolutional Neural Network	839
19.2.6	Comparing HPO Algorithms	841
19.2.7	Summary	842
19.2.8	Exercises	842
19.3	Asynchronous Random Search	843
19.3.1	Objective Function	844
19.3.2	Asynchronous Scheduler	845
19.3.3	Visualize the Asynchronous Optimization Process	851
19.3.4	Summary	852
19.3.5	Exercises	853
19.4	Multi-Fidelity Hyperparameter Optimization	853
19.4.1	Successive Halving	855
19.4.2	Summary	866
19.5	Asynchronous Successive Halving	867
19.5.1	Objective Function	869
19.5.2	Asynchronous Scheduler	870
19.5.3	Visualize the Optimization Process	879
19.5.4	Summary	879
20	Generative Adversarial Networks	880
20.1	Generative Adversarial Networks	880
20.1.1	Generate Some “Real” Data	882
20.1.2	Generator	883
20.1.3	Discriminator	883
20.1.4	Training	883
20.1.5	Summary	885
20.1.6	Exercises	885
20.2	Deep Convolutional Generative Adversarial Networks	886
20.2.1	The Pokemon Dataset	886
20.2.2	The Generator	887
20.2.3	Discriminator	889
20.2.4	Training	891
20.2.5	Summary	892

20.2.6	Exercises	892
21	Recommender Systems	893
21.1	Overview of Recommender Systems	893
21.1.1	Collaborative Filtering	894
21.1.2	Explicit Feedback and Implicit Feedback	895
21.1.3	Recommendation Tasks	895
21.1.4	Summary	895
21.1.5	Exercises	895
	Appendix A	Mathematics for Deep Learning
		897
	Appendix B	Tools for Deep Learning
		1035
	References	1089

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 we 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 blue skies academic discipline whose industrial significance was limited to a narrow set of real-world applications, including speech recognition and computer vision. Moreover, many of these applications required so much domain knowledge that they were often regarded as entirely separate areas for which machine learning was one small component. At that time, neural networks—the predecessors of the deep learning methods that we focus on in this book—were generally regarded as outmoded.

Yet in just few years, deep learning has taken the world by surprise, driving rapid progress in such diverse fields as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and biomedical informatics. Moreover, the success of deep learning in so many tasks of practical interest has even catalyzed developments in theoretical machine learning and statistics. With these advances in hand, we can now build cars that drive themselves with more autonomy than ever before (though less autonomy than some companies might have you believe), dialogue systems that debug code by asking clarifying questions, and software agents beating the best human players in the world at board games such as Go, a feat once thought to be decades away. Already, these tools exert ever-wider influence on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics, to climate modeling, to weather prediction, to biomedicine.

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 distilled, minimizing the onboarding time needed to bring new practitioners up to

date. Mature libraries should automate common tasks, and exemplar code should make it easy for practitioners to modify, apply, and extend common applications to suit their needs.

As an example, take dynamic web applications. Despite a large number of companies, such as Amazon, developing successful database-driven web applications in the 1990s, the potential of this technology to aid creative entrepreneurs was realized to a far greater degree only in the past ten years, owing in part to the development of powerful, well-documented frameworks.

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 mathematical form of a given model; (iii) the optimization algorithms for fitting the models to data; (iv) the statistical principles that tell us when we should expect our models to generalize to unseen data and practical methods for certifying that they have, in fact, generalized; and (v) the engineering techniques required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching 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.

When we started this book project, there were no resources that simultaneously (i) remained up to date; (ii) covered the breadth of modern machine learning practices with sufficient technical depth; and (iii) interleaved exposition of the quality one expects of a textbook with the clean runnable code that one expects of a hands-on tutorial. We found plenty of code examples illustrating 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, ResNet, 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 Distill¹, or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several deep learning textbooks have emerged—e.g., Goodfellow *et al.* (2016), which offers a comprehensive survey on the basics of 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 forum² 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. No workflows seemed suited to these demands, so we decided to assemble our own (Section B.6). We settled on GitHub to share the source and to facilitate community contributions; Jupyter notebooks for mixing code, equations and text; Sphinx as a rendering engine; and Discourse as a discussion platform. While our system is not perfect, these choices strike a compromise among the competing concerns. We believe that *Dive into Deep Learning* might be the first book published using such an integrated workflow.

Learning by Doing

Many textbooks present concepts in succession, covering each in exhaustive detail. For example, the excellent textbook of Bishop (2006), teaches each topic so thoroughly that getting to the chapter on linear regression requires a nontrivial amount of work. While experts love this book precisely for its thoroughness, for true beginners, this property limits its usefulness as an introductory text.

In this book, we 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 concepts.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter both introduces a reasonable number of new concepts and provides several self-contained working examples, using real datasets. This presented 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. By contrast, there is a big advantage to adhering to a policy of *one working example, one 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.

Throughout, we interleave the runnable code with background material as needed. In general, we err on the side of making tools available before explaining them fully (often filling in the background later). For instance, we might use *stochastic gradient descent* before explaining why it is useful or offering some intuition for 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 teaches deep learning concepts from scratch. Sometimes, we delve into fine details about models that would typically be hidden from users by modern deep learning frameworks. 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 often present two versions of the example: one where we implement everything from scratch, relying only on NumPy-like functionality and automatic differentiation, and a more prac-

tical example, where we write succinct code using the high-level APIs of deep learning frameworks. After explaining how some component works, we rely on the high-level API in subsequent tutorials.

Content and Structure

The book can be divided into roughly three parts, dealing with preliminaries, deep learning techniques, and advanced topics focused on real systems and applications (Fig. 1).

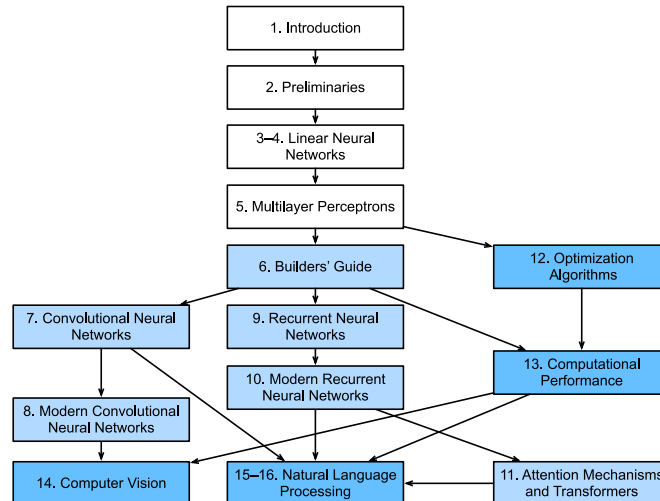


Fig. 1

Book structure.

- **Part 1: Basics and Preliminaries.** Chapter 1 is 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 elementary concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 5 cover the most fundamental concepts and techniques in deep learning, including regression and classification; linear models; multilayer perceptrons; and overfitting and regularization.
- **Part 2: Modern Deep Learning Techniques.** Chapter 6 describes the key computational components of deep learning systems and lays the groundwork for our subsequent implementations of more complex models. Next, Chapter 7 and Chapter 8 present convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Similarly, Chapter 9 and Chapter 10 introduce recurrent neural networks (RNNs), models that exploit sequential (e.g., temporal) structure in data and are commonly used for natural language processing and time series prediction. In Chapter 11, we describe a relatively new class of models, based on so-called *attention mechanisms*, that has displaced RNNs as the dominant architecture for most natural language processing tasks. These sections will bring you up to speed on the most powerful and general tools that are widely used by deep learning practitioners.

3



- **Part 3: Scalability, Efficiency, and Applications** (available online³). In Chapter 12, we discuss several common optimization algorithms used to train deep learning models. Next, in Chapter 13, we examine several key factors that influence the computational performance of deep learning code. Then, in Chapter 14, we illustrate major applications of deep learning in computer vision. Finally, in Chapter 15 and Chapter 16, we demonstrate how to pretrain language representation models and apply them to natural language processing tasks.

Code

Most sections of this book feature executable code. We believe that some intuitions are best developed via 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. However, deep learning practitioners today must often tread where no solid theory provides guidance. Despite our best attempts, formal explanations for the efficacy of various techniques are still lacking, for a variety of reasons: the mathematics to characterize these models can be so difficult; the explanation likely depends on properties of the data that currently lack clear definitions; and serious inquiry on these topics has only recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, each future edition of this book will provide insights that eclipse those presently available.

To avoid unnecessary repetition, we capture some of our most frequently imported and used functions and classes in the `d2l` package. Throughout, we mark blocks of code (such as functions, classes, or collection of import statements) with `#@save` to indicate that they will be accessed later via the `d2l` package. We offer a detailed overview of these classes and functions in Section B.8. The `d2l` package is lightweight and only requires the following dependencies:

```
#@save
import collections
import hashlib
import inspect
import math
import os
import random
import re
import shutil
import sys
import tarfile
import time
import zipfile
from collections import defaultdict
import pandas as pd
import requests
from IPython import display
from matplotlib import pyplot as plt
from matplotlib_inline import backend_inline

d2l = sys.modules[__name__]
```

Most of the code in this book is based on PyTorch, a popular open-source framework that has been enthusiastically embraced by the deep learning research community. All of the code in this book has passed tests under the latest stable version of PyTorch. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of PyTorch. We plan to keep the online version up to date. In case you encounter any problems, please consult *Installation* (page xxxiv) to update your code and runtime environment. Below lists dependencies in our PyTorch implementation.

```
#@save
import numpy as np
import torch
import torchvision
from PIL import Image
from scipy.spatial import distance_matrix
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
```

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 enter with some basics, including modest amounts of linear algebra, calculus, probability, and Python programming. Just in case you have forgotten anything, the online Appendix⁴ provides a refresher on most of the mathematics you will find in this book. Usually, we will prioritize intuition and ideas over mathematical rigor. If you would like to extend these foundations beyond the prerequisites to understand our book, we happily recommend some other terrific resources: *Linear Analysis* by Bollobás (1999) covers linear algebra and functional analysis in great depth. *All of Statistics* (Wasserman, 2013) provides a marvelous introduction to statistics. Joe Blitzstein's books⁵ and courses⁶ on probability and inference are pedagogical gems. And if you have not used Python before, you may want to peruse this Python tutorial⁷.

Notebooks, Website, GitHub, and Forum

All of our notebooks are available for download on the D2L.ai website⁸ and on GitHub⁹. Associated with this book, we have launched a discussion forum, located at discuss.d2l.ai¹⁰. Whenever you have questions on any section of the book, you can find a link to the associated discussion page at the end of each notebook.

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. This book was originally implemented with MXNet as the primary framework. We thank Anirudh Dagar and Yuan Tang for adapting a majority part of earlier MXNet code into PyTorch and TensorFlow implementations, respectively. Since July 2021, we have redesigned and reimplemented this book in PyTorch, MXNet, and TensorFlow, choosing PyTorch as the primary framework. We thank Anirudh Dagar for adapting a majority part of more recent PyTorch code into JAX implementations. We thank Gaosheng Wu, Liujun Hu, Ge Zhang, and Jiehang Xie from Baidu for adapting a majority part of more recent PyTorch code into PaddlePaddle implementations in the Chinese draft. We thank Shuai Zhang for integrating the LaTeX style from the press into the PDF building.

On GitHub, 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, sun-deepteki, topecongiro, tpd, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, Igov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcincio, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwds, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddaredygar, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, Igov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil, Peter Kulits, codypenta, joseppinilla, ahmaurya, karolszk, heytitle, Peter Goetz, rigtorp, Tiep Vu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, MatteoFerrara, hsneto, Katarzyna Biesialska, Gregory Bruss, Duy–Thanh Doan, paulaurel, graytowne, Duc Pham, sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, Cuong V. Nguyen, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysrael, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, Hoa Nguyen, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ghejc, BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran, djliden, Nikhil95, Oren Barkan, guoweis, haozhu233, pratikhack, Yue Ying, tayfununal, steinsag, charleybeller, Andrew Lumsdaine, Jiekui Zhang, Deepak Pathak, Florian Donhauser, Tim Gates, Adriaan Tijsseling, Ron

Medina, Gaurav Saha, Murat Semerci, Lei Mao, Levi McClenny, Joshua Broyde, jake221, jonbally, zyhazwraith, Brian Pulfer, Nick Tomasino, Lefan Zhang, Hongshen Yang, Vinney Cavallo, yuntai, Yuanxiang Zhu, amarazov, pasricha, Ben Greenawald, Shivam Upadhyay, Quanshangze Du, Biswajit Sahoo, Parthe Pandit, Ishan Kumar, HomunculusK, Lane Schwartz, varadgunjal, Jason Wiener, Armin Gholampoor, Shreshtha13, eigen-arnav, Hyeong-gyu Kim, EmilyOng, Bálint Mucsányi, Chase DuBois, Juntian Tao, Wenxiang Xu, Lifu Huang, filevich, quake2005, nils-werner, Yiming Li, Marsel Khisamutdinov, Francesco “Fuma” Fumagalli, Peilin Sun, Vincent Gurgul, qingfengtommy, Janmey Shukla, Mo Shan, Kaan Sancak, regob, AlexSauer, Gopalakrishna Ramachandra, Tobias Uelwer, Chao Wang, Tian Cao, Nicolas Corthorn, akash5474, kxxt, zxydi1992, Jacob Britton, Shuangchi He, zh-mou, krahets, Jie-Han Chen, Atishay Garg, Marcel Flygare, adtygan, Nik Vaessen, bolded, Louis Schlessinger, Balaji Varatharajan, atgetg, Kaixin Li, Victor Barbaros, Riccardo Musto, Elizabeth Ho, azimjonn, Guilherme Miotto, Alessandro Finamore, Joji Joseph, Anthony Biel, Zeming Zhao, shjustinbaek, gab-chen, nantekoto, Yutaro Nishiyama, Oren Amsalem, Tian-MaoMao, Amin Allahyar, Gijs van Tulder, Mikhail Berkov, iamorphen, Matthew Caseres, Andrew Walsh, pggPL, RohanKarthikeyan, Ryan Choi, and Likun Lei.

We thank Amazon Web Services, especially Wen-Ming Ye, George Karypis, Swami Sivasubramanian, Peter DeSantis, Adam Selipsky, 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. During the preparation of the book for publication, Cambridge University Press has offered excellent support. We thank our commissioning editor David Tranah for his help and professionalism.

Summary

Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, in such diverse fields as computer vision, natural language processing, and automatic speech recognition. To successfully apply deep learning, you must understand how to cast a problem, the basic 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.

Exercises

11



1. Register an account on the discussion forum of this book discuss.d2l.ai¹¹.
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.


12

Discussions¹².

Installation

In order to get up and running, we will need an environment for running Python, the Jupyter Notebook, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda


- 13  Your simplest option is to install Miniconda¹³. Note that the Python 3.x version is required. You can skip the following steps if your machine already has conda installed.

Visit the Miniconda website and determine the appropriate version for your system based on your Python 3.x version and machine architecture. Suppose that your Python version is 3.9 (our tested version). If you are using macOS, you would download the bash script whose name contains the strings “MacOSX”, navigate to the download location, and execute the installation as follows (taking Intel Macs as an example):

```
# The file name is subject to changes
sh Miniconda3-py39_4.12.0-MacOSX-x86_64.sh -b
```

A Linux user would download the file whose name contains the strings “Linux” and execute the following at the download location:

```
# The file name is subject to changes
sh Miniconda3-py39_4.12.0-Linux-x86_64.sh -b
```

- 14  A Windows user would download and install Miniconda by following its online instructions¹⁴. On Windows, you may search for cmd to open the Command Prompt (command-line interpreter) for running commands.

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

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

Then close and reopen your current shell. You should be able to create a new environment as follows:

```
conda create --name d2l python=3.9 -y
```

Now we can activate the d2l environment:

```
conda activate d2l
```

Installing the Deep Learning Framework and the d2l Package

15



Before installing any 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 are not relevant for our purposes). For example, if your computer has NVIDIA GPUs and has installed CUDA¹⁵, then you are all set. If your machine does not house any GPU, there is no need to worry just yet. Your CPU provides more than enough horsepower to get you through the first few chapters. Just remember that you will want to access GPUs before running larger models.

You can install PyTorch (the specified versions are tested at the time of writing) with either CPU or GPU support as follows:

```
pip install torch==2.0.0 torchvision==0.15.1
```

Our next step is to install the d2l package that we developed in order to encapsulate frequently used functions and classes found throughout this book:

```
pip install d2l==1.0.3
```

Downloading and Running the Code

16



Next, you will want to download the notebooks so that you can run each of the book's code blocks. Simply click on the "Notebooks" tab at the top of any HTML page on the D2L.ai website¹⁶ to download the code and then unzip it. Alternatively, you can fetch the notebooks from the command line as follows:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-1.0.3.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
cd pytorch
```

If you do not already have `unzip` installed, first run `sudo apt-get install unzip`. Now we can start the Jupyter Notebook server by running:

```
jupyter notebook
```

At this point, you can open `http://localhost:8888` (it may have already opened automatically) in your web browser. Then we can run the code for each section of the book. Whenever you open a new command line window, you will need to execute `conda activate d2l` to activate the runtime environment before running the D2L notebooks, or updating your packages (either the deep learning framework or the `d2l` package). To exit the environment, run `conda deactivate`.

Discussions¹⁷.

17



Notation

Throughout this book, we adhere to the following notational conventions. Note that some of these symbols are placeholders, while others refer to specific objects. As a general rule of thumb, the indefinite article “a” often indicates that the symbol is a placeholder and that similarly formatted symbols can denote other objects of the same type. For example, “ x : a scalar” means that lowercased letters generally represent scalar values, but “ \mathbb{Z} : the set of integers” refers specifically to the symbol \mathbb{Z} .

Numerical Objects

- x : a scalar
- \mathbf{x} : a vector
- \mathbf{X} : a matrix
- \mathbf{X} : a general tensor
- \mathbf{I} : the identity matrix (of some given dimension), i.e., a square matrix with 1 on all diagonal entries and 0 on all off-diagonals
- $x_i, [\mathbf{x}]_i$: the i^{th} 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{Z}^+ : the set of positive 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}
- $\mathcal{A} \setminus \mathcal{B}$: set subtraction of \mathcal{B} from \mathcal{A} (contains only those elements of \mathcal{A} that do not belong to \mathcal{B})

Functions and Operators

- $f(\cdot)$: a function
- $\log(\cdot)$: the natural logarithm (base e)
- $\log_2(\cdot)$: logarithm to base 2
- $\exp(\cdot)$: the exponential function
- $\mathbf{1}(\cdot)$: the indicator function; evaluates to 1 if the boolean argument is true, and 0 otherwise
- $\mathbf{1}_X(z)$: the set-membership indicator function; evaluates to 1 if the element z belongs to the set X and 0 otherwise
- $(\cdot)^\top$: transpose of a vector or a matrix
- \mathbf{X}^{-1} : inverse of matrix \mathbf{X}
- \odot : Hadamard (elementwise) product
- $[\cdot, \cdot]$: concatenation
- $\|\cdot\|_p$: ℓ_p norm
- $\|\cdot\|$: ℓ_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: inner (dot) product of vectors \mathbf{x} and \mathbf{y}
- \sum : summation over a collection of elements
- \prod : product over a collection of elements
- $\stackrel{\text{def}}{=}$: an equality asserted as a definition of the symbol on the left-hand side

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

- X : a random variable
- P : a probability distribution
- $X \sim P$: the random variable X follows distribution P
- $P(X = x)$: the probability assigned to the event where random variable X takes value x
- $P(X | Y)$: the conditional probability distribution of X given Y
- $p(\cdot)$: a probability density function (PDF) associated with distribution P
- $E[X]$: expectation of a random variable X
- $X \perp Y$: random variables X and Y are independent
- $X \perp Y | Z$: random variables X and Y are conditionally independent given Z
- σ_X : standard deviation of random variable X
- $\text{Var}(X)$: variance of random variable X , equal to σ_X^2
- $\text{Cov}(X, Y)$: covariance of random variables X and Y
- $\rho(X, Y)$: the Pearson correlation coefficient between X and Y , equals $\frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}$
- $H(X)$: entropy of random variable X
- $D_{\text{KL}}(P || Q)$: the KL-divergence (or relative entropy) from distribution Q to distribution P

Discussions¹⁸.



Until recently, nearly every computer program that you might have interacted with during an ordinary day was coded up as a rigid set of rules specifying precisely how it should behave. 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 might settle on the broad strokes of a working solution, for example: (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 a set of rules that map every conceivable circumstance to the corresponding action that our program should take.

To build the brains of our application, we might enumerate all the common events that our program should handle. For example, whenever a customer clicks to add an item to their shopping cart, our program should add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. We might then attempt to step through every possible corner case, testing the appropriateness of our rules and making any necessary modifications. What happens if a user initiates a purchase with an empty cart? 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 can write such programs and confidently launch them *before* ever seeing a real customer. Our ability to manually design automated systems 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 typically should not be worrying about machine learning.

Fortunately for the growing community of machine learning 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 factoid question, expressed in free-form text, and answers it correctly.
- Write a program that, given an image, identifies every person depicted in it and draws outlines around each.

- Write a program that presents users with products that they are likely to enjoy but unlikely, in the natural course of browsing, to encounter.

For these problems, even elite programmers would struggle to code up solutions from scratch. The reasons can vary. Sometimes the program that we are looking for follows a pattern that changes over time, so there is no fixed right answer! In such cases, any successful solution must adapt gracefully to a changing world. At other times, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations and following unknown principles. In the case of image recognition, the precise steps required to perform the task lie beyond our conscious understanding, even though our subconscious cognitive processes execute the task effortlessly.

Machine learning is the study of algorithms that can learn from experience. As a machine learning 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 follows 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, focusing 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 beginning 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 (app) 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 this story was fabricated 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* such as “Alexa”, “OK Google”, and “Hey 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} about 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 machine learning.



Fig. 1.1.1 Identify a wake word.

Here is 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 it. Armed with this ability, we can collect a huge *dataset* containing examples of audio snippets and associated labels, indicating which snippets contain the wake word. In the currently dominant approach to machine learning, we do not attempt to design a system *explicitly* to recognize wake words. Instead, we define a flexible program whose behavior is determined by a number of *parameters*. Then we use the dataset to determine the best possible parameter values, i.e., those that improve the performance of our program with respect to a chosen performance measure.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Once the parameters are fixed, we call the program a *model*. The set of all distinct programs (input–output mappings) that we can produce just by manipulating the parameters is called a *family* of 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 the model 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 the snippet contains the wake word.

If we choose the right family of models, 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 unlikely that our model will recognize “Alexa”, “Apricot”, or any other English word. In machine learning, the *learning* is the process by which we discover the right setting of the knobs for coercing the

desired behavior from our model. In other words, we *train* our model with data. As shown in Fig. 1.1.2, the training process usually looks like the following:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your data (e.g., audio snippets and corresponding {yes, no} labels).
3. Tweak the knobs to make the model perform better as assessed on those examples.
4. Repeat Steps 2 and 3 until the model is awesome.

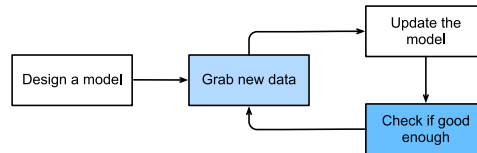


Fig. 1.1.2 A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, if presented 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*. That is to say, we can “program” a cat detector by providing our machine learning system with many examples of cats and dogs. 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. This barely scratches the surface of what machine learning can do. Deep learning, which we will explain in greater detail later, is just one among many popular methods for solving machine learning problems.

1.2 Key Components

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 based on known inputs given a dataset consisting of examples for which the labels are known, is called *supervised learning*. This is just one among many kinds of machine learning problems. Before we explore other varieties, we would like to shed more light on some core components that will follow us around, no matter what kind of machine learning problem we tackle:

1. The *data* that we can learn from.
2. A *model* of how to transform the data.
3. An *objective function* that quantifies how well (or badly) the model is doing.
4. An *algorithm* to adjust the model's parameters to optimize the objective function.

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 data *is*, but for now, we will focus on the key properties of the datasets that we will 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* (or *data point*, *data instance*, *sample*) typically consists of a set of attributes called *features* (sometimes called *covariates* or *inputs*), based on which the model must make its predictions. In supervised learning problems, our goal is to predict the value of a special attribute, called the *label* (or *target*), that is not part of the model's input.

If we were working with image data, each example might consist of an individual photograph (the features) and a number indicating the category to which the photograph belongs (the label). The photograph would be represented numerically as three grids of numerical values representing the brightness of red, green, and blue light at each pixel location. For example, a 200×200 pixel color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values.

Alternatively, we might work with electronic health record data and tackle the task of predicting the likelihood that a given patient will survive the next 30 days. Here, our features might consist of a collection of readily available attributes and frequently recorded measurements, including age, vital signs, comorbidities, current medications, and recent procedures. The label available for training would be a binary value indicating whether each patient in the historical data survived within the 30-day window.

In such cases, when every example is characterized by the same number of numerical features, we say that the inputs are fixed-length vectors and we call the (constant) length of the vectors the *dimensionality* of the data. As you might imagine, fixed-length inputs can be convenient, giving us one less complication 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 all to have the same resolution or shape. For images, we might consider cropping them 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 such as Amazon, IMDb, and 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 preconceived 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 might work in the small 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 out*. Moreover, poor predictive performance is not the only potential consequence. In sensitive applications of machine learning, like predictive policing, resume screening, and risk models used for lending, we must be especially alert to the consequences of garbage data. One commonly occurring failure mode concerns datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system that had never seen black skin before. Failure can also occur when the data does not only under-represent some groups but reflects societal prejudices. For example, if past hiring decisions are used to train a predictive model that will be used to screen resumes then machine learning models could inadvertently 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 models, we will also 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 updating our model, and some people might disagree on whether our proposal constituted an improvement or not.

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 for which higher is better, and turn it into a new function that is qualitatively identical but for which lower is better by flipping the sign. Because we choose lower to be better, these functions are sometimes called *loss functions*.

When trying to predict numerical values, the most common loss function is *squared error*, i.e., the square of the difference between the prediction and the ground truth target. For classification, the most common objective is to minimize error rate, i.e., the fraction of

examples on which our predictions disagree with the ground truth. Some objectives (e.g., squared error) are easy to optimize, while others (e.g., error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common instead to optimize a *surrogate objective*.

During optimization, we think of the loss as a function of the model's parameters, and treat the training dataset as a constant. We learn the best values of our model's parameters by minimizing the loss incurred on a 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 data. So we will typically want to split the available data into two partitions: the *training dataset* (or *training set*), for learning model parameters; and the *test dataset* (or *test set*), which is held out for evaluation. At the end of the day, we typically report how our models perform on both partitions. You could think of training performance as analogous to the scores that a student achieves on the practice exams used to prepare for some real final exam. Even if the results are encouraging, that does not guarantee success on the final exam. Over the course of studying, the student might begin to memorize the practice questions, appearing to master the topic but faltering when faced with previously unseen questions on the actual final exam. When a model performs well on the training set but fails to generalize to unseen data, we say that it is *overfitting* to the training data.

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. Popular optimization algorithms for deep learning are based on an approach called *gradient descent*. In brief, at each step, this method checks to see, for each parameter, how that training set loss would change if you perturbed that parameter by just a small amount. It would then update the parameter in the direction that lowers the loss.

1.3 Kinds of Machine Learning Problems

The wake word problem in our motivating example is just one among many that machine learning can tackle. To motivate the reader further and provide us with some common language that will follow us throughout the book, we now provide a broad overview of the landscape of machine learning problems.

1.3.1 Supervised Learning

Supervised learning describes tasks where we are given a dataset containing both features and labels and asked to produce a model that predicts the labels when given input features. Each feature-label pair is called an example. Sometimes, when the context is clear, we may use the term *examples* to refer to a collection of inputs, even when the corresponding

labels are unknown. The supervision comes into play because, for choosing the parameters, we (the supervisors) provide the model with a dataset consisting of labeled examples. In probabilistic terms, we typically are interested in estimating the conditional probability of a label given input features. While it is just one among several paradigms, supervised learning accounts for the majority of successful applications of machine learning in industry. Partly that is because many important tasks can be described crisply as estimating the probability of something unknown given a particular set of available data:

- Predict cancer vs. not cancer, given a computer tomography image.
- Predict the correct translation in French, given a sentence in English.
- Predict the price of a stock next month based on this month's financial reporting data.

While all supervised learning problems are captured by the simple description “predicting the labels given input features”, supervised learning itself can take diverse forms and require tons of modeling decisions, depending on (among other considerations) the type, size, and quantity of the inputs and outputs. For example, we use different models for processing sequences of arbitrary lengths and fixed-length vector representations. We will visit many of these problems in depth throughout this book.

Informally, the learning process looks something like the following. First, grab a big collection of examples for which the features are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that have 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.

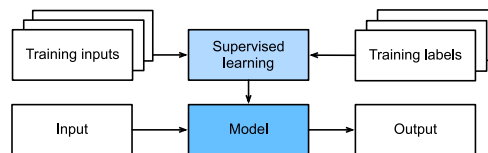


Fig. 1.3.1 Supervised learning.

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, in which 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: [600, 1, 1, 60]. However, if you live in Pittsburgh, it might look more like [3000, 4, 3, 10]. Fixed-length feature vectors like this are essential for most classic machine learning algorithms.

What makes a problem a regression is actually the form of the target. Say that you are in the market for a new home. You might want to estimate the fair market value of a house, given some features such as above. The data here might consist of historical home listings and the labels might be the observed sales prices. When labels take on arbitrary numerical values (even within some interval), we call this a *regression* problem. The goal is to produce a model whose predictions closely approximate the actual label values.

Lots of practical problems are easily described as 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 prize¹⁹. 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 is likely to be regression. For example:



- How many hours will this surgery take?
- How much rainfall will this town have in the next six hours?

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 3 hours removing gunk from your sewage pipes. Then they sent you a bill of 350 dollars. Now imagine that your friend hired the same contractor for 2 hours and received a bill of 250 dollars. 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 dollars per hour plus 50 dollars to show up at your house. If you followed that much, then you already understand the high-level idea behind *linear* regression.

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes this is not possible, e.g., if some of the variation arises from factors beyond 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 minimizing the squared error loss function. As we will see later, this loss corresponds to the assumption that our data were corrupted by Gaussian noise.

Classification

While regression models are great for addressing *how many?* questions, lots of problems do not fit comfortably in this template. Consider, for example, a bank that wants to develop a

check scanning feature for its mobile app. Ideally, the customer would simply snap a photo of a check and the app would automatically recognize the text from the image. Assuming that we had some ability to segment out image patches corresponding to each handwritten character, then the primary remaining task would be to determine which character among some known set is depicted in each image patch. These kinds of *which one?* problems are called *classification* and require a different set of tools from those used for regression, although many techniques will carry over.

In *classification*, we want our model to look at features, e.g., the pixel values in an image, and then predict to which *category* (sometimes called a *class*) among some discrete set of options, an example belongs. For handwritten digits, we might have ten 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 could consist of images of animals and our labels might be the classes {cat, dog}. Whereas in regression we sought a regressor to output a numerical value, in classification we seek a classifier, whose output is the predicted class assignment.

For reasons that we will get into as the book gets more technical, it can be difficult to optimize a model that can only output a *firm* categorical assignment, e.g., either “cat” or “dog”. In these cases, it is usually much easier to express our model in the language of probabilities. Given features of an example, our model assigns a probability to each possible class. Returning to our animal classification example where the classes are {cat, dog}, a classifier might see an image and output the probability that the image is a cat as 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 a notion of uncertainty. It is not the only one available and we will discuss others in chapters dealing with more advanced topics.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include handwritten character recognition {0, 1, 2, ... 9, a, b, c, ...}. While we attacked regression problems by trying to minimize the squared error loss function, the common loss function for classification problems is called *cross-entropy*, whose name will be demystified when we introduce information theory in later chapters.

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

Now, assume that you built a classifier and trained it to predict whether a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs that the probability that Fig. 1.3.2 shows a death cap is 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. Thus, in order to make a decision about whether to eat the mushroom, we need to compute the expected detriment associated with each action which depends both on the likely outcomes and the benefits or harms associated with each. In this case, the detriment incurred by eating the mushroom



Fig. 1.3.2 Death cap - do not eat!

might be $0.2 \times \infty + 0.8 \times 0 = \infty$, whereas the loss of discarding it is $0.2 \times 0 + 0.8 \times 1 = 0.8$. Our caution was justified: as any mycologist would tell us, the mushroom in Fig. 1.3.2 is actually a death cap.

Classification can get much more complicated than just binary or multiclass classification. For instance, there are some variants of classification addressing hierarchically structured classes. In such cases not all errors are equal—if we must err, we might prefer to misclassify to a related class rather than a distant class. Usually, this is referred to as *hierarchical classification*. For inspiration, you might think of Linnaeus²⁰, who organized fauna in a hierarchy.

20



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 with a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattlesnakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could have fatal consequences.

Tagging

Some classification problems 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*, a popular German fairy tale featuring four animals (Fig. 1.3.3).

As you can see, the photo features a cat, a rooster, a dog, and a donkey, with some trees in the background. If we anticipate encountering such images, multiclass classification might not be the right problem formulation. Instead, we might want to give the model the option of saying the image depicts a cat, a dog, a donkey, *and* a rooster.



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

The problem of learning to predict classes that are not mutually exclusive is called *multi-label classification*. Auto-tagging problems are typically best described in terms of multi-label classification. Think of the tags people might apply to posts on a technical blog, e.g., “machine learning”, “technology”, “gadgets”, “programming languages”, “Linux”, “cloud computing”, “AWS”. A typical article might have 5–10 tags applied. Typically, tags will exhibit some correlation structure. Posts about “cloud computing” are likely to mention “AWS” and posts about “machine learning” are likely to mention “GPUs”.

Sometimes such tagging problems draw on enormous label sets. The National Library of Medicine employs many professional annotators who associate each article to be indexed in PubMed with a set of tags drawn from the Medical Subject Headings (MeSH) ontology, a collection of roughly 28,000 tags. Correctly tagging articles is important because it allows researchers to conduct exhaustive reviews of the literature. This is a time-consuming process and typically there is a one-year lag between archiving and tagging. Machine learning can provide provisional tags until each article has a proper manual review. Indeed, for several years, the BioASQ organization has hosted competitions²¹ for this task.

21



Search

22



In the field of information retrieval, we often impose ranks on sets 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, among a set of relevant results, should be shown most prominently to a particular user. One way of doing this might be to first assign a score to every element in the set and then to retrieve the top-rated elements. PageRank²², the original secret sauce behind the Google search engine, was an early example of such a scoring system. Weirdly, the scoring provided by PageRank did not depend on the actual query. Instead, they relied on a simple relevance filter to identify the set of relevant candidates and then used PageRank to prioritize the more authoritative pages. 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 items relevant 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 science fiction 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, and 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, or Goodreads). In other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction or maybe just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score, such as an expected star rating or the probability that a given user will purchase a particular item.

Given such a model, for any given user, we could retrieve the set of objects with the largest scores, which could then be recommended to the user. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. Fig. 1.3.4 displays the deep learning books recommended by Amazon based on personalization algorithms tuned to capture Aston's preferences.

Despite their tremendous economic value, recommender 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. For example, on a five-point scale, you might notice that items receive many one- and five-star ratings but that there are conspicuously few three-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

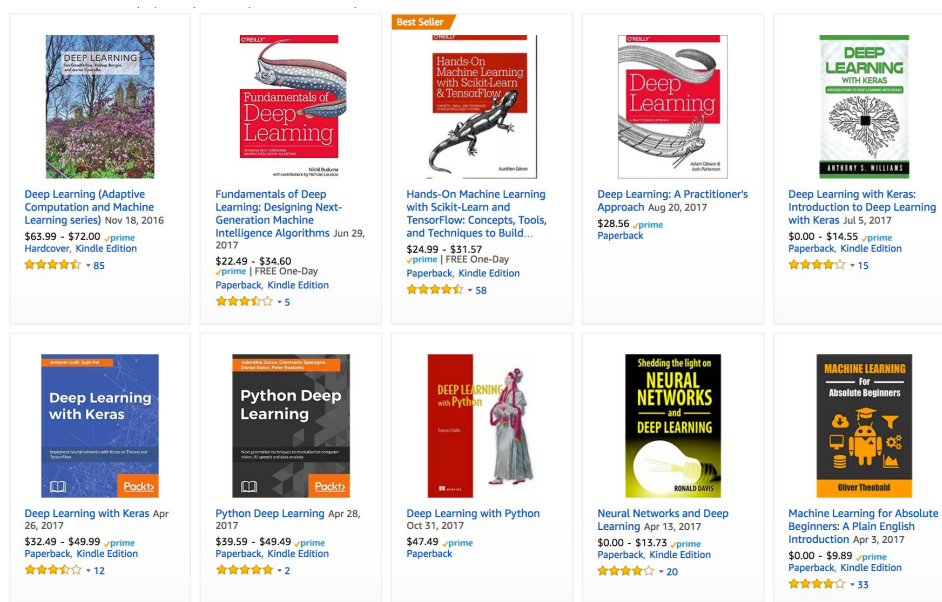


Fig. 1.3.4 Deep learning books recommended by Amazon.

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. For example, we considered predicting house prices given a fixed set of features: square footage, number of bedrooms, number of bathrooms, and the transit time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each among a fixed number of classes and predicting star ratings associated with purchases based on the user ID and product ID alone. In these cases, once our model is trained, after each test example is fed into our model, it is immediately forgotten. We assumed that successive observations were independent and thus there was no need to hold on to this context.

But how should we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames. The same goes for language. For example, one popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translations in another language.

Such problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts whenever their risk of dying in the next 24 hours exceeds some threshold. Here, we would not throw away everything that we know about

the patient history every hour, because we might not want to make predictions based only on the most recent measurements.

Questions like these are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model either to ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence-to-sequence learning* considers problems where both inputs and outputs consist of variable-length sequences. Examples include machine translation and speech-to-text transcription. While it is impossible to consider all types of sequence transformations, the following special cases are worth mentioning.

Tagging and Parsing. This involves annotating a text sequence with attributes. Here, the inputs and outputs are *aligned*, i.e., they are of the same number and occur in a corresponding order. For instance, in *part-of-speech (PoS) tagging*, we annotate every word in a sentence with the corresponding part of speech, i.e., “noun” or “direct object”. Alternatively, we might want to know which groups of contiguous words refer to named entities, like *people*, *places*, or *organizations*. In the cartoonishly simple example below, we might just want to indicate whether or not any word in the sentence is part of a named entity (tagged as “Ent”).

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

Automatic Speech Recognition. With speech recognition, the input sequence is an audio recording of a speaker (Fig. 1.3.5), and the output is a 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 may correspond to a single spoken word. These are sequence-to-sequence learning problems, where the output is much shorter than the input. While humans are remarkably good at recognizing speech, even from low-quality audio, getting computers to perform the same feat is a formidable challenge.

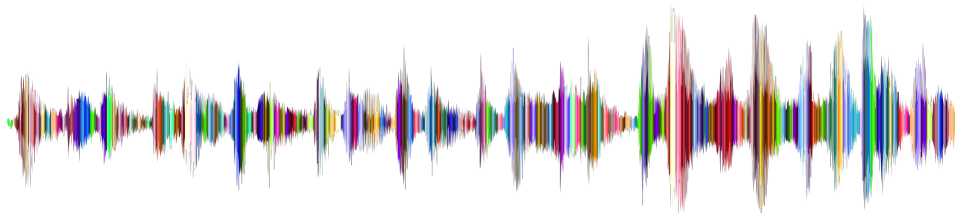


Fig. 1.3.5 -D-e-e-p- L-ea-r-ni-ng- in an audio recording.

Text to Speech. This is the inverse of automatic speech recognition. Here, the input is text and the output is an audio file. In this case, the output is much longer than the input.

Machine Translation. Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order, in machine translation, unaligned data poses a new challenge. Here the input and output sequences can have different lengths, and the corre-

sponding regions of the respective sequences may appear in a different order. 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:	Have you already looked at this excellent textbook?
Wrong alignment:	Have you yourself already this excellent textbook 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. Such topics are active areas of research.

1.3.2 Unsupervised and Self-Supervised Learning

The previous examples focused on supervised learning, where we feed the model a giant dataset containing both the features and corresponding label values. You could think of the supervised learner as having an extremely specialized job and an extremely dictatorial boss. The boss stands over the learner's shoulder and tells them exactly what to do in every situation until they learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, pleasing such a boss is pretty easy. You just recognize the pattern as quickly as possible and imitate the boss's actions.

Considering the opposite situation, 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 had 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 vague. We call this class of problems *unsupervised learning*, and the type and number of questions we can ask is limited only by our creativity. We will address unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the following 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, and mountain peaks? Likewise, given a collection of users' browsing activities, 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 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*. If the dependence is linear, it is called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space such that symbolic properties can be well matched? This can be used to describe entities and their relations, such as "Rome" – "Italy" + "France" = "Paris".

- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, and salaries, can we discover how they are related simply based on empirical data? The fields concerned with *causality* and *probabilistic graphical models* tackle such questions.
- Another important and exciting recent development in unsupervised learning is the advent of *deep generative models*. These models estimate the density of the data, either explicitly or *implicitly*. Once trained, we can use a generative model either to score examples according to how likely they are, or to sample synthetic examples from the learned distribution. Early deep learning breakthroughs in generative modeling came with the invention of *variational autoencoders* (Kingma and Welling, 2014, Rezende *et al.*, 2014) and continued with the development of *generative adversarial networks* (Goodfellow *et al.*, 2014). More recent advances include normalizing flows (Dinh *et al.*, 2014, Dinh *et al.*, 2017) and diffusion models (Ho *et al.*, 2020, Sohl-Dickstein *et al.*, 2015, Song and Ermon, 2019, Song *et al.*, 2021).

A further development in unsupervised learning has been the rise of *self-supervised learning*, techniques that leverage some aspect of the unlabeled data to provide supervision. For text, we can train models to “fill in the blanks” by predicting randomly masked words using their surrounding words (contexts) in big corpora without any labeling effort (Devlin *et al.*, 2018)! For images, we may train models to tell the relative position between two cropped regions of the same image (Doersch *et al.*, 2015), to predict an occluded part of an image based on the remaining portions of the image, or to predict whether two examples are perturbed versions of the same underlying image. Self-supervised models often learn representations that are subsequently leveraged by fine-tuning the resulting models on some downstream task of interest.

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 each 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 the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For example, supervised learning assumes the simple interaction pattern depicted in Fig. 1.3.6.

This simplicity of offline learning has its charms. The upside is that we can worry about pattern recognition in isolation, with no concern about complications arising from interactions with a dynamic environment. But this problem formulation is limiting. If you grew up reading Asimov’s Robot novels, then you probably picture artificially intelligent agents capable not only of making predictions, but also of taking actions in the world. We want to think about intelligent *agents*, not just predictive models. This means that we need to think about choosing *actions*, not just making predictions. In contrast to mere predictions, actions actually impact the environment. If we want to train an intelligent agent, we must

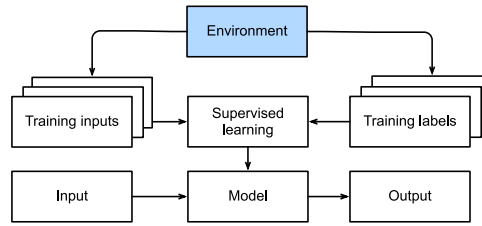


Fig. 1.3.6 Collecting data for supervised learning from an environment.

account for the way its actions might impact the future observations of the agent, and so offline learning is inappropriate.

Considering the interaction with an environment opens a whole set of new modeling questions. The following are just a few examples.

- Does the environment remember what we did previously?
- Does the environment want to help us, e.g., a user reading text into a speech recognizer?
- Does the environment want to beat us, e.g., spammers adapting their emails to evade spam filters?
- Does the environment have shifting dynamics? For example, would future data always resemble the past or would the patterns change over time, either naturally or in response to our automated tools?

These questions raise the problem of *distribution shift*, where training and test data are different. An example of this, that many of us may have met, is when taking exams written by a lecturer, while the homework was composed by their teaching assistants. Next, we briefly describe reinforcement learning, a rich framework for posing learning problems in which an agent interacts 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*. This might include applications to robotics, to dialogue systems, and even to developing artificial intelligence (AI) for video games. *Deep reinforcement learning*, which applies deep learning to reinforcement learning problems, has surged in popularity. The breakthrough deep Q-network, that beat humans at Atari games using only the visual input (Mnih *et al.*, 2015), and the AlphaGo program, which dethroned the world champion at the board game Go (Silver *et al.*, 2016), 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 time steps. At each time step, the agent receives some *observation* from the environment and must choose an *action* that is subsequently transmitted back to the environment via some mechanism (sometimes called an *actuator*), when, after each loop, the agent receives a reward from the environment. This process is

illustrated in Fig. 1.3.7. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of a reinforcement learning agent is governed by a *policy*. In brief, a *policy* is just a function that maps from observations of the environment to actions. The goal of reinforcement learning is to produce good policies.

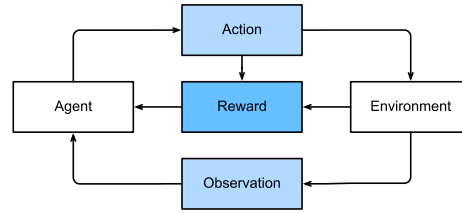


Fig. 1.3.7 The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the reinforcement learning framework. For example, supervised learning can be recast as reinforcement learning. Say we had a classification problem. We could create a reinforcement learning 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 learning problem.

Further, reinforcement learning can also address many problems that supervised learning cannot. For example, in supervised learning, we always expect that the training input comes associated with the correct label. But in reinforcement learning, we do not assume that, for each observation the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider the game of chess. The only real reward signal comes at the end of the game when we either win, earning a reward of, say, 1, or when we lose, receiving a reward of, say, -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 number of well-chosen actions over the previous year. Getting promoted in the future requires figuring out which actions along the way led to the earlier promotions.

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 your cleaning robot found itself trapped in one of many identical closets in your house. Rescuing the robot involves inferring its precise location which might require considering earlier observations prior to it 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-term reward in exchange for knowledge.

The general reinforcement learning problem has a very general setting. Actions affect sub-

sequent observations. Rewards are only observed when they correspond to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may be asking too much. 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 reinforcement learning problem a *Markov decision process*. When the state does not depend on the previous actions, we call it a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, we have the classic *multi-armed bandit problem*.

1.4 Roots

We have just reviewed a small subset of problems that machine learning can address. For a diverse set of machine learning problems, deep learning provides powerful tools for their solution. Although many deep learning methods are recent inventions, the core ideas behind learning from data have been studied for centuries. In fact, humans have held the desire to analyze data and to predict future outcomes for ages, and it is this desire that is at the root of much of natural science and mathematics. Two examples are the Bernoulli distribution, named after Jacob Bernoulli (1655–1705)²³, and the Gaussian distribution discovered by Carl Friedrich Gauss (1777–1855)²⁴. Gauss invented, for instance, the least mean squares algorithm, which is still used today for a multitude of problems from insurance calculations to medical diagnostics. Such tools enhanced the 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)²⁵ illustrates averaging the length of 16 adult men’s feet to estimate the typical foot length in the population (Fig. 1.4.1).

As a group of individuals exited a church, 16 adult men were asked to line up in a row and have their feet measured. The sum of these measurements was then divided by 16 to obtain an estimate for what now is called one foot. This “algorithm” was later improved to deal with misshapen feet; The two men with the shortest and longest feet were sent away, averaging only over the remainder. This is among the earliest examples of a trimmed mean estimate.

Statistics really took off with the availability and collection of data. One of its pioneers, Ronald Fisher (1890–1962)²⁶, contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as linear discriminant analysis) and concepts (such as the Fisher information matrix) still hold a prominent place in the foundations of modern statistics. Even his data resources had a lasting impact. The Iris dataset that Fisher released in 1936 is still sometimes used to demonstrate machine learning algorithms. Fisher was also a proponent of eugenics, which should remind us that the morally



Fig. 1.4.1 Estimating the length of a foot.

dubious use of data science has as long and enduring a history as its productive use in industry and the natural sciences.

Other influences for machine learning came from the information theory of Claude Shannon (1916–2001)²⁷ and the theory of computation proposed by Alan Turing (1912–1954)²⁸. Turing posed the question “can machines think?” in his famous paper *Computing Machinery and Intelligence* (Turing, 1950). Describing what is now known as the Turing test, he proposed that a machine can be considered *intelligent* if it is difficult for a human evaluator to distinguish between the replies from a machine and those of a human, based purely on textual interactions.

Further influences came from neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. Many scholars have asked whether one could explain and possibly reverse engineer this capacity. One of the first biologically inspired algorithms was formulated by Donald Hebb (1904–1985)²⁹. In his groundbreaking book *The Organization of Behavior* (Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. These ideas inspired later work, such as Rosenblatt’s perceptron learning algorithm, and 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 mainly due to two reasons. First, training a network is computationally very expensive. While random-access memory 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 1936 was still a popular tool for testing the efficacy of algorithms. The MNIST dataset 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 in many applications. Moreover, 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 availability of massive amounts of data, thanks to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of low-cost, high-quality sensors, inexpensive data storage (Kryder’s law), and cheap computation (Moore’s law). In particular, the landscape of computation in deep learning was revolutionized by advances in GPUs that were originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible were within reach. This is best illustrated in `tab_intro_decade`.

:Dataset vs. computer memory and computational power

Table 1.5.1: `label:tab_intro_decade`

Decade	Dataset	Memory	Floating point calculations per second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (house prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

Note that random-access memory has not kept pace with the growth in data. At the same time, increases in computational power have outpaced the growth in datasets. This means that statistical models need to become more memory efficient, and so they are free to spend more computer cycles optimizing parameters, thanks to the increased compute budget. Consequently, the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep neural networks. This is also one of the reasons why many of the mainstays of deep learning, such as multilayer perceptrons (McCulloch and Pitts, 1943), convolutional neural networks (LeCun *et al.*, 1998), long short-term memory (Hochreiter and Schmidhuber, 1997), and Q-Learning (Watkins and Dayan, 1992), were essentially “rediscovered” in the past decade, after lying 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 of ideas below barely scratches the surface of what has 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 overfitting. Here, noise is injected (Bishop, 1995) throughout the neural network during training.
- *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. Researchers found an elegant solution by using what can only be viewed as a *learnable pointer structure* (Bahdanau *et al.*, 2014). Rather than having to remember an entire text sequence, 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 sequences, since the model no longer needed to remember the entire sequence before commencing the generation of a new one.
- Built solely on attention mechanisms, the *Transformer* architecture (Vaswani *et al.*, 2017)

has demonstrated superior *scaling* behavior: it performs better with an increase in dataset size, model size, and amount of training compute (Kaplan *et al.*, 2020). This architecture has demonstrated compelling success in a wide range of areas, such as natural language processing (Brown *et al.*, 2020, Devlin *et al.*, 2018), computer vision (Dosovitskiy *et al.*, 2021, Liu *et al.*, 2021), speech recognition (Gulati *et al.*, 2020), reinforcement learning (Chen *et al.*, 2021), and graph neural networks (Dwivedi and Bresson, 2020). For example, a single Transformer pretrained on modalities as diverse as text, images, joint torques, and button presses can play Atari, caption images, chat, and control a robot (Reed *et al.*, 2022).

30



- Modeling probabilities of text sequences, *language models* can predict text given other text. Scaling up the data, model, and compute has unlocked a growing number of capabilities of language models to perform desired tasks via human-like text generation based on input text (Anil *et al.*, 2023, Brown *et al.*, 2020, Chowdhery *et al.*, 2022, Hoffmann *et al.*, 2022, OpenAI, 2023, Rae *et al.*, 2021, Touvron *et al.*, 2023a, Touvron *et al.*, 2023b). For instance, aligning language models with human intent (Ouyang *et al.*, 2022), OpenAI's ChatGPT³⁰ allows users to interact with it in a conversational way to solve problems, such as code debugging and creative writing.
- Multi-stage designs, e.g., via the memory networks (Sukhbaatar *et al.*, 2015) and the neural programmer-interpreter (Reed and De Freitas, 2015) permitted statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep neural network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, just as a processor can modify memory for a computation.
- A key development in *deep generative modeling* was the invention of *generative adversarial networks* (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 generative adversarial networks 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, density estimation was opened up to a wide variety of techniques. Examples of galloping zebras (Zhu *et al.*, 2017) and of fake celebrity faces (Karras *et al.*, 2017) are each testimony to this progress. Even amateur doodlers can produce photorealistic images just based on sketches describing the layout of a scene (Park *et al.*, 2019).
- Furthermore, while the diffusion process gradually adds random noise to data samples, *diffusion models* (Ho *et al.*, 2020, Sohl-Dickstein *et al.*, 2015) learn the denoising process to gradually construct data samples from random noise, reversing the diffusion process. They have started to replace generative adversarial networks in more recent deep generative models, such as in DALL-E 2 (Ramesh *et al.*, 2022) and Imagen (Saharia *et al.*, 2022) for creative art and image generation based on text descriptions.
- In many cases, a single GPU is insufficient for processing the large amounts of data

available for training. Over the past decade the ability to build parallel and distributed training algorithms has improved significantly. One of the key challenges in designing scalable algorithms is that the workhorse of deep learning optimization, stochastic gradient descent, relies on relatively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1,024 GPUs with a minibatch size of, say, 32 images per batch amounts to an aggregate minibatch of about 32,000 images. Work, first by Li (2017) and subsequently by You *et al.* (2017) and Jia *et al.* (2018) pushed the size up to 64,000 observations, reducing training time for the ResNet-50 model on the ImageNet dataset to less than 7 minutes. By comparison, training times were initially of the order of days.



- The ability to parallelize computation has also contributed to progress in *reinforcement learning*. This has led to significant progress in computers achieving superhuman performance on tasks like Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo) where environment simulators are available. See, e.g., Silver *et al.* (2016) for a description of such achievements in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) tuples are available. Simulation provides such an avenue.
- Deep learning frameworks have played a crucial role in disseminating ideas. The first generation of open-source frameworks for neural network modeling consisted of Caffe³¹, Torch³², and Theano³³. Many seminal papers were written using these tools. These have now been superseded by TensorFlow³⁴ (often used via its high-level API Keras³⁵), CNTK³⁶, Caffe 2³⁷, and Apache MXNet³⁸. The third generation of frameworks consists of so-called *imperative* tools for deep learning, a trend that was arguably ignited by Chainer³⁹, which used a syntax similar to Python NumPy to describe models. This idea was adopted by both PyTorch⁴⁰, the Gluon API⁴¹ of MXNet, and JAX⁴².

The division of labor between system researchers building better tools and statistical modelers building better neural 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 Ph.D. students at Carnegie Mellon University in 2014. By now, this task can be accomplished with under 10 lines of code, putting it firmly within the reach of any programmer.

1.6 Success Stories



Artificial intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail sorting systems using optical character recognition have been deployed since the 1990s. This is, after all, the source of the famous MNIST dataset of handwritten digits. The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud auto-

matically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, and MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization, and ranking on the Internet. In other words, machine learning is pervasive, albeit often hidden from sight.

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously and that are directly related to consumers. Many of such advances are attributed to deep learning.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, and Google's assistant, are able to respond to spoken requests with a reasonable degree of accuracy. This includes menial jobs, like turning on light switches, and more complex tasks, such as arranging 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 their ability to recognize speech accurately. The accuracy of such systems has gradually increased to the point of achieving parity with humans for certain applications (Xiong *et al.*, 2018).
- Object recognition has likewise come a long way. Identifying the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark researchers from NEC Labs and University of Illinois at Urbana-Champaign achieved a top-five error rate of 28% (Lin *et al.*, 2010). By 2017, this error rate was reduced to 2.25% (Hu *et al.*, 2018). Similarly, stunning results have been achieved for identifying birdsong and for diagnosing skin cancer.
- Prowess in games used to provide a measuring stick for human ability. Starting from TD-Gammon, a program for playing backgammon using temporal difference reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Compared with backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Garry Kasparov using massive parallelism, special-purpose hardware and efficient search through the game tree (Campbell *et al.*, 2002). Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, using deep learning combined with Monte Carlo tree sampling (Silver *et al.*, 2016). The challenge in Poker was that the state space is large and only partially observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown and Sandholm, 2017).
- Another indication of progress in AI is the advent of self-driving vehicles. While full autonomy is not yet within reach, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable 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 visual aspect of these problems. The rest is heavily tuned by engineers.

This barely scratches the surface of significant applications of machine learning. 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, which is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, questions about a coming AI apocalypse and the plausibility of a *singularity* have been raised in non-technical articles. The fear is that somehow machine learning systems will become sentient and make decisions, independently of their programmers, that directly impact the lives of humans. To some extent, AI already affects the livelihood of humans in direct ways: creditworthiness is assessed automatically, autopilots mostly navigate vehicles, decisions about whether to grant bail use statistical data as input. More frivolously, we can ask Alexa to switch on the coffee machine.

Fortunately, we are far from a sentient AI system that could deliberately manipulate its human creators. First, AI systems are engineered, trained, and deployed in a specific, goal-oriented manner. While their behavior might give the illusion of general intelligence, it is a combination of rules, heuristics and statistical models that underlie the design. Second, at present, there are simply no tools for *artificial general intelligence* 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 routine tasks, currently fulfilled by humans, can and will be automated. Farm robots will likely reduce the costs for organic farmers but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences for large swaths of society, since menial jobs provide much employment in many countries. 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 as a much more pressing concern than the potential of malevolent superintelligence for destroying humanity.

1.7 The Essence of Deep Learning

Thus far, we have talked in broad terms about machine learning. Deep learning is the subset of machine learning concerned with models based on many-layered neural networks. It is *deep* in precisely the sense that its models learn many *layers* of transformations. While this might sound narrow, deep learning has given rise to a dizzying array of models, techniques, problem formulations, and applications. Many intuitions have been developed to explain the benefits of depth. Arguably, all machine learning has many layers of computation, the first consisting of feature processing steps. What differentiates deep learning is that the operations learned at each of the many layers of representations are learned jointly from data.

The problems that we have discussed so far, such as 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 those where deep learning excels and traditional methods falter. It turns out that these many-layered models are capable of addressing low-level perceptual data in a way that previous tools could not. Arguably the most significant commonality in deep learning methods is *end-to-end training*. That is, rather than assembling a system based on components that are individually tuned, one builds the system and then tunes their performance jointly. For instance, in computer vision scientists used to separate the process of *feature engineering* from the process of building machine learning models. The Canny edge detector (Canny, 1987) and Lowe’s SIFT feature extractor (Lowe, 2004) reigned supreme for over a decade as algorithms for mapping images into feature vectors. In bygone days, the crucial part of applying machine learning to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to shallow models. Unfortunately, there is only so much that humans can accomplish by ingenuity in comparison with a consistent evaluation over millions of choices carried out automatically by an algorithm. When deep learning took over, these feature extractors were replaced by automatically tuned filters that yielded superior accuracy.

Thus, 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. Moreover, 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, thereby offering a unified set of tools for tackling diverse problems.

Beyond end-to-end training, we are experiencing a transition from parametric statistical descriptions to fully nonparametric models. When data is scarce, one needs to rely on simplifying assumptions about reality in order to obtain useful models. When data is abundant, these can be replaced by nonparametric models that better fit the data. To some extent, this mirrors the progress that physics experienced in the middle of the previous century with the availability of computers. Rather than solving by hand parametric approximations of how electrons behave, one can now resort to numerical simulations of the associated partial differential equations. This has led to much more accurate models, albeit often at the expense of interpretation.

Another difference from previous work is the acceptance of suboptimal solutions, dealing with nonconvex nonlinear optimization problems, and the willingness to try things before proving them. This new-found empiricism in dealing with statistical problems, combined with a rapid influx of talent has led to rapid progress in the development of practical algorithms, albeit in many cases at the expense of modifying and re-inventing tools that existed for decades.

In the end, the deep learning community prides itself on sharing tools across academic and corporate boundaries, releasing many excellent libraries, statistical models, and trained networks as open source. It is in this spirit that the notebooks forming this book are freely available for distribution and use. We have worked hard to lower the barriers of access for

anyone wishing to learn about deep learning and we hope that our readers will benefit from this.

1.8 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, and optimization. Often, it is used as a means of implementing AI solutions. As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. Considered as multi-level representation learning through learning many layers of transformations, deep learning replaces not only the shallow models at the end of traditional machine learning pipelines, but also the labor-intensive process of feature engineering. 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. Furthermore, the availability of efficient deep learning frameworks has made design and implementation of whole system optimization significantly easier, and this is a key component in obtaining high performance.

1.9 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? What data might you need to learn the desired behavior?
2. Which problems that you encounter have many examples for their solution, yet no specific way for automating them? These may be prime candidates for using deep learning.
3. Describe the relationships between algorithms, data, and computation. How do characteristics of the data and the current available computational resources influence the appropriateness of various algorithms?
4. Name some settings where end-to-end training is not currently the default approach but where it might be useful.

Discussions⁴³.



To prepare for your dive into deep learning, you will need a few survival skills: (i) techniques for storing and manipulating data; (ii) libraries for ingesting and preprocessing data from a variety of sources; (iii) knowledge of the basic linear algebraic operations that we apply to high-dimensional data elements; (iv) just enough calculus to determine which direction to adjust each parameter in order to decrease the loss function; (v) the ability to automatically compute derivatives so that you can forget much of the calculus you just learned; (vi) some basic fluency in probability, our primary language for reasoning under uncertainty; and (vii) some aptitude for finding answers in the official documentation when you get stuck.

In short, this chapter provides a rapid introduction to the basics that you will need to follow *most* of the technical content in this book.

2.1 Data Manipulation

In order to get anything done, we need some way to store and manipulate data. Generally, there are two important things we need to do with data: (i) acquire them; and (ii) process them once they are inside the computer. There is no point in acquiring data without some way to store it, so to start, let's get our hands dirty with n -dimensional arrays, which we also call *tensors*. If you already know the NumPy scientific computing package, this will be a breeze. For all modern deep learning frameworks, the *tensor class* (ndarray in MXNet, Tensor in PyTorch and TensorFlow) resembles NumPy's ndarray, with a few killer features added. First, the tensor class supports automatic differentiation. Second, it leverages GPUs to accelerate numerical computation, whereas NumPy only runs on CPUs. These properties make neural networks both easy to code and fast to run.

2.1.1 Getting Started

To start, we import the PyTorch library. Note that the package name is `torch`.

```
import torch
```

A tensor represents a (possibly multidimensional) array of numerical values. In the one-dimensional case, i.e., when only one axis is needed for the data, a tensor is called a *vector*.

With two axes, a tensor is called a *matrix*. With $k > 2$ axes, we drop the specialized names and just refer to the object as a k^{th} -order tensor.

PyTorch provides a variety of functions for creating new tensors prepopulated with values. For example, by invoking `arange(n)`, we can create a vector of evenly spaced values, starting at 0 (included) and ending at `n` (not included). By default, the interval size is 1. Unless otherwise specified, new tensors are stored in main memory and designated for CPU-based computation.

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

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

Each of these values is called an *element* of the tensor. The tensor `x` contains 12 elements. We can inspect the total number of elements in a tensor via its `numel` method.

```
x.numel()
```

```
12
```

We can access a tensor's *shape* (the length along each axis) by inspecting its `shape` attribute. Because we are dealing with a vector here, the shape contains just a single element and is identical to the size.

```
x.shape
```

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

We can change the shape of a tensor without altering its size or values, by invoking `reshape`. For example, we can transform our vector `x` whose shape is (12,) to a matrix `X` with shape (3, 4). This new tensor retains all elements but reconfigures them into a matrix. Notice that the elements of our vector are laid out one row at a time and thus `x[3] == X[0, 3]`.

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

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

Note that specifying every shape component to `reshape` is redundant. Because we already know our tensor's size, we can work out one component of the shape given the rest. For example, given a tensor of size n and target shape (h, w) , we know that $w = n/h$. To

automatically infer one component of the shape, we can place a `-1` for the shape component that should be inferred automatically. In our case, instead of calling `x.reshape(3, 4)`, we could have equivalently called `x.reshape(-1, 4)` or `x.reshape(3, -1)`.

Practitioners often need to work with tensors initialized to contain all 0s or 1s. We can construct a tensor with all elements set to 0 and a shape of (2, 3, 4) via the `zeros` function.

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

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

Similarly, we can create a tensor with all 1s by invoking `ones`.

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

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

We often wish to sample each element randomly (and independently) from a given probability distribution. For example, the parameters of neural networks are often initialized randomly. The following snippet creates a tensor with elements drawn from a standard Gaussian (normal) distribution with mean 0 and standard deviation 1.

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

```
tensor([[ 0.1351, -0.9099, -0.2028,  2.1937],
        [-0.3200, -0.7545,  0.8086, -1.8730],
        [ 0.3929,  0.4931,  0.9114, -0.7072]])
```

Finally, we can construct tensors by supplying the exact values for each element by supplying (possibly nested) Python list(s) containing numerical literals. Here, we construct a matrix with a list of lists, where the outermost list corresponds to axis 0, and the inner list corresponds to axis 1.


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

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

2.1.2 Indexing and Slicing

As with Python lists, we can access tensor elements by indexing (starting with 0). To access an element based on its position relative to the end of the list, we can use negative indexing. Finally, we can access whole ranges of indices via slicing (e.g., `X[start:stop]`), where the returned value includes the first index (*start*) *but not the last* (*stop*). Finally, when only one index (or slice) is specified for a k^{th} -order tensor, it is applied along axis 0. Thus, in the following code, `[-1]` selects the last row and `[1:3]` selects the second and third rows.

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

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

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

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

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

If we want to assign multiple elements the same value, we apply the indexing on the left-hand side of the assignment operation. For instance, `[:2, :]` accesses the first and second rows, where `:` takes all the elements along axis 1 (column). While we discussed indexing for matrices, this also works for vectors and for tensors of more than two dimensions.

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

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

2.1.3 Operations

Now that we know how to construct tensors and how to read from and write to their elements, we can begin to manipulate them with various mathematical operations. Among the most useful of these are the *elementwise* operations. These apply a standard scalar operation to each element of a tensor. For functions that take two tensors as inputs, elementwise operations apply some standard binary operator on each pair of corresponding elements. We can create an elementwise function from any function that maps from a scalar to a scalar.

In mathematical notation, we denote such *unary* scalar operators (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$. This just means that the function maps from any real number onto some other real number. Most standard operators, including unary ones like e^x , can be applied elementwise.

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

```
tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
        162754.7969, 162754.7969, 162754.7969, 2980.9580, 8103.0840,
        22026.4648, 59874.1406])
```

Likewise, we denote *binary* scalar operators, which map pairs of real numbers to a (single) real number via 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 i^{th} 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 for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**) have all been *lifted* to elementwise operations for identically-shaped tensors of arbitrary shape.

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

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

In addition to elementwise computations, we can also perform linear algebraic operations, such as dot products and matrix multiplications. We will elaborate on these in Section 2.3.

We can also *concatenate* multiple tensors, stacking them end-to-end to form a larger one. 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) instead of columns (axis 1). We can see that the first output's axis-0 length (6) is the sum of the two input tensors' axis-0 lengths (3 + 3); while the second output's axis-1 length (8) is the sum of the two input tensors' axis-1 lengths (4 + 4).

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

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

Sometimes, we want to construct a binary tensor via *logical statements*. Take $X == Y$ as an example. For each position i, j , if $X[i, j]$ and $Y[i, j]$ are equal, then the corresponding entry in the result takes value 1, otherwise it takes value 0.

```
X == Y
```

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

Summing all the elements in the tensor yields a tensor with only one element.

```
X.sum()
```

```
tensor(66.)
```

2.1.4 Broadcasting

By now, you know how to perform elementwise binary operations on two tensors of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise binary operations by invoking the *broadcasting mechanism*. Broadcasting works according to the following two-step procedure: (i) expand one or both arrays by copying elements along axes with length 1 so that after this transformation, the two tensors have the same shape; (ii) perform an elementwise operation on the resulting arrays.

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

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

Since a and b are 3×1 and 1×2 matrices, respectively, their shapes do not match up. Broadcasting produces a larger 3×2 matrix by replicating matrix a along the columns and matrix b along the rows before adding them elementwise.

```
a + b
```

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

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 dereference the tensor that Y used to point to and instead point Y at the newly allocated memory. We can demonstrate this issue with Python's `id()` function, which gives us the exact address of the referenced object in memory. Note that after we run $Y = Y + X$, `id(Y)` points to a different location. That is because Python first evaluates $Y + X$, allocating new memory for the result and then points Y to this new location in memory.

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

```
False
```

This might be undesirable for two reasons. First, we do not want to run around allocating memory unnecessarily all the time. In machine learning, we often have hundreds of megabytes of parameters and update all of them multiple times per second. Whenever possible, we want to perform these updates *in place*. Second, we might point at the same parameters from multiple variables. If we do not update in place, we must be careful to update all of these references, lest we spring a memory leak or inadvertently refer to stale parameters.

Fortunately, performing in-place operations is easy. We can assign the result of an operation to a previously allocated array Y by using slice notation: $Y[:] = \text{<expression>}$. To illustrate this concept, we overwrite the values of tensor Z , after initializing it, using `zeros_like`, to have the same shape as Y .

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

```
id(Z): 140381179266448
id(Z): 140381179266448
```

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 (ndarray), or vice versa, is easy. The torch tensor and NumPy array will share their underlying memory, and changing one through an in-place operation will also change the other.

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

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

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

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

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

2.1.7 Summary

The tensor class is the main interface for storing and manipulating data in deep learning libraries. Tensors provide a variety of functionalities including construction routines; indexing and slicing; basic mathematics operations; broadcasting; memory-efficient assignment; and conversion to and from other Python objects.

2.1.8 Exercises

1. Run the code in this section. Change the conditional statement `X == Y` 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?

Discussions⁴⁴.

44



2.2 Data Preprocessing

So far, we have been working with synthetic data that arrived in ready-made tensors. However, to apply deep learning in the wild we must extract messy data stored in arbitrary formats, and preprocess it to suit our needs. Fortunately, the *pandas* library⁴⁵ can do much of the heavy lifting. This section, while no substitute for a proper *pandas* tutorial⁴⁶, will give you a crash course on some of the most common routines.

45



46



2.2.1 Reading the Dataset

Comma-separated values (CSV) files are ubiquitous for the storing of tabular (spreadsheet-like) data. In them, each line corresponds to one record and consists of several (comma-separated) fields, e.g., “Albert Einstein, March 14 1879, Ulm, Federal polytechnic school, field of gravitational physics”. To demonstrate how to load CSV files with *pandas*, we create a CSV file below `./data/house_tiny.csv`. This file represents a dataset of homes, where each row corresponds to a distinct home and the columns correspond to the number of rooms (NumRooms), the roof type (RoofType), and the price (Price).

```
import os

os.makedirs(os.path.join('.', 'data'), exist_ok=True)
data_file = os.path.join('.', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,RoofType,Price\n'
            'NA,NA,127500\n'
            '2,NA,106000\n'
            '4,Slate,178100\n'
            'NA,NA,140000')
```

Now let's import *pandas* and load the dataset with `read_csv`.

```
import pandas as pd

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

	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000

2.2.2 Data Preparation

In supervised learning, we train models to predict a designated *target* value, given some set of *input* values. Our first step in processing the dataset is to separate out columns corresponding to input versus target values. We can select columns either by name or via integer-location based indexing (`iloc`).

You might have noticed that pandas replaced all CSV entries with value NA with a special NaN (*not a number*) value. This can also happen whenever an entry is empty, e.g., “3,,270000”. These are called *missing values* and they are the “bed bugs” of data science, a persistent menace that you will confront throughout your career. Depending upon the context, missing values might be handled either via *imputation* or *deletion*. Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

Here are some common imputation heuristics. For categorical input fields, we can treat NaN as a category. Since the RoofType column takes values Slate and NaN, pandas can convert this column into two columns RoofType_Slate and RoofType_nan. A row whose roof type is Slate will set values of RoofType_Slate and RoofType_nan to 1 and 0, respectively. The converse holds for a row with a missing RoofType value.

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

For missing numerical values, one common heuristic is to replace the NaN entries with the mean value of the corresponding column.

```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

	NumRooms	RoofType_Slate	RoofType_nan
0	3.0	False	True
1	2.0	False	True

(continues on next page)

(continued from previous page)

2	4.0	True	False
3	3.0	False	True

2.2.3 Conversion to the Tensor Format

Now that all the entries in `inputs` and `targets` are numerical, we can load them into a tensor (recall Section 2.1).

```
import torch
```

```
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

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

2.2.4 Discussion

You now know how to partition data columns, impute missing variables, and load pandas data into tensors. In Section 5.7, you will pick up some more data processing skills. While this crash course kept things simple, data processing can get hairy. For example, rather than arriving in a single CSV file, our dataset might be spread across multiple files extracted from a relational database. For instance, in an e-commerce application, customer addresses might live in one table and purchase data in another. Moreover, practitioners face myriad data types beyond categorical and numeric, for example, text strings, images, audio data, and point clouds. Oftentimes, advanced tools and efficient algorithms are required in order to prevent data processing from becoming the biggest bottleneck in the machine learning pipeline. These problems will arise when we get to computer vision and natural language processing. Finally, we must pay attention to data quality. Real-world datasets are often plagued by outliers, faulty measurements from sensors, and recording errors, which must be addressed before feeding the data into any model. Data visualization tools such as `seaborn`⁴⁷, `Bokeh`⁴⁸, or `matplotlib`⁴⁹ can help you to manually inspect the data and develop intuitions about the type of problems you may need to address.

2.2.5 Exercises

1. Try loading datasets, e.g., `Abalone` from the UCI Machine Learning Repository⁵⁰ and inspect their properties. What fraction of them has missing values? What fraction of the variables is numerical, categorical, or text?
2. Try indexing and selecting data columns by name rather than by column number. The pandas documentation on indexing⁵¹ has further details on how to do this.

3. How large a dataset do you think you could load this way? What might be the limitations? Hint: consider the time to read the data, representation, processing, and memory footprint. Try this out on your laptop. What happens if you try it out on a server?
4. How would you deal with data that has a very large number of categories? What if the category labels are all unique? Should you include the latter?
5. What alternatives to pandas can you think of? How about loading NumPy tensors from a file⁵²? Check out Pillow⁵³, the Python Imaging Library.

52

Discussions⁵⁴.

53



2.3 Linear Algebra

54



By now, we can load datasets into tensors and manipulate these tensors with basic mathematical operations. To start building sophisticated models, we will also need a few tools from linear algebra. This section offers a gentle introduction to the most essential concepts, starting from scalar arithmetic and ramping up to matrix multiplication.

```
import torch
```

2.3.1 Scalars

Most everyday mathematics consists of manipulating numbers one at a time. Formally, we call these values *scalars*. For example, the temperature in Palo Alto is a balmy 72 degrees Fahrenheit. If you wanted to convert the temperature to Celsius you would evaluate the expression $c = \frac{5}{9}(f - 32)$, setting f to 72. In this equation, the values 5, 9, and 32 are constant scalars. The variables c and f in general represent unknown scalars.

We denote scalars by ordinary lower-cased letters (e.g., x , y , and z) and the space of all (continuous) *real-valued* scalars by \mathbb{R} . For expedience, we will skip past rigorous definitions of *spaces*: just remember that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol \in (pronounced “in”) denotes membership in a set. For example, $x, y \in \{0, 1\}$ indicates that x and y are variables that can only take values 0 or 1.

Scalars are implemented as tensors that contain only one element. Below, we assign two scalars and perform the familiar addition, multiplication, division, and exponentiation operations.

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

2.3.2 Vectors

For current purposes, you can think of a vector as a fixed-length array of scalars. As with their code counterparts, we call these scalars the *elements* of the vector (synonyms include *entries* and *components*). When vectors represent examples from real-world datasets, their values hold some real-world significance. For example, if we were training a model to predict the risk of a loan defaulting, we might associate each applicant with a vector whose components correspond to quantities like their income, length of employment, or number of previous defaults. If we were studying the risk of heart attack, each vector might represent a patient and its components might correspond to their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. We denote vectors by bold lowercase letters, (e.g., \mathbf{x} , \mathbf{y} , and \mathbf{z}).

Vectors are implemented as 1st-order tensors. In general, such tensors can have arbitrary lengths, subject to memory limitations. Caution: in Python, as in most programming languages, vector indices start at 0, also known as *zero-based indexing*, whereas in linear algebra subscripts begin at 1 (one-based indexing).

```
x = torch.arange(3)
x
```

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

We can refer to an element of a vector by using a subscript. For example, x_2 denotes the second element of \mathbf{x} . Since x_2 is a scalar, we do not bold it. By default, we visualize vectors by stacking their elements vertically.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad (2.3.1)$$

Here x_1, \dots, x_n are elements of the vector. Later on, we will distinguish between such *column vectors* and *row vectors* whose elements are stacked horizontally. Recall that we access a tensor's elements via indexing.

```
x[2]
```

```
tensor(2)
```

To indicate that a vector contains n elements, we write $\mathbf{x} \in \mathbb{R}^n$. Formally, we call n the *dimensionality* of the vector. In code, this corresponds to the tensor's length, accessible via Python's built-in `len` function.

```
len(x)
```

```
3
```

We can also access the length via the shape attribute. The shape is a tuple that indicates a tensor's length along each axis. Tensors with just one axis have shapes with just one element.

```
x.shape
```

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

Oftentimes, the word “dimension” gets overloaded to mean both the number of axes and the length along a particular axis. To avoid this confusion, we use *order* to refer to the number of axes and *dimensionality* exclusively to refer to the number of components.

2.3.3 Matrices

Just as scalars are 0th-order tensors and vectors are 1st-order tensors, matrices are 2nd-order tensors. We denote matrices by bold capital letters (e.g., \mathbf{X} , \mathbf{Y} , and \mathbf{Z}), and represent them in code by tensors with two axes. The expression $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates that a matrix \mathbf{A} contains $m \times n$ real-valued scalars, arranged as m rows and n columns. When $m = n$, we say that a matrix is *square*. Visually, we can illustrate any matrix as a table. To refer to an individual element, we subscript both the row and column indices, e.g., a_{ij} is the value that belongs to \mathbf{A} 's i^{th} row and j^{th} column:

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

In code, we represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ by a 2nd-order tensor with shape (m, n) . We can convert any appropriately sized $m \times n$ tensor into an $m \times n$ matrix by passing the desired shape to reshape:

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

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

Sometimes we want to flip the axes. When we exchange a matrix's rows and columns, the result is called its *transpose*. Formally, we signify a matrix \mathbf{A} 's transpose by \mathbf{A}^\top and if

$\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for all i and j . Thus, the transpose of an $m \times n$ matrix is an $n \times m$ matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \quad (2.3.3)$$

In code, we can access any matrix's transpose as follows:

```
A.T
```

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

Symmetric matrices are the subset of square matrices that are equal to their own transposes:

$\mathbf{A} = \mathbf{A}^\top$. The following matrix is symmetric:

```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

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

Matrices are useful for representing datasets. Typically, rows correspond to individual records and columns correspond to distinct attributes.

2.3.4 Tensors

While you can go far in your machine learning journey with only scalars, vectors, and matrices, eventually you may need to work with higher-order tensors. Tensors give us a generic way of describing extensions to n^{th} -order arrays. We call software objects of the *tensor class* “tensors” precisely because they too can have arbitrary numbers of axes. While it may be confusing to use the word *tensor* for both the mathematical object and its realization in code, our meaning should usually be clear from context. We denote general tensors by capital letters with a special font face (e.g., \mathbf{X} , \mathbf{Y} , and \mathbf{Z}) and their indexing mechanism (e.g., x_{ijk} and $[\mathbf{X}]_{1,2i-1,3}$) follows naturally from that of matrices.

Tensors will become more important when we start working with images. Each image arrives as a 3rd-order tensor with axes corresponding to the height, width, and *channel*. At each spatial location, the intensities of each color (red, green, and blue) are stacked along the channel. Furthermore, a collection of images is represented in code by a 4th-order tensor, where distinct images are indexed along the first axis. Higher-order tensors are constructed, as were vectors and matrices, by growing the number of shape components.

```
torch.arange(24).reshape(2, 3, 4)
```

```
tensor([[[ 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 higher-order tensors all have some handy properties. For example, elementwise operations produce outputs that have the same shape as their operands.

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone() # Assign a copy of A to B by allocating new memory
A, A + B
```

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

The elementwise product of two matrices is called their *Hadamard product* (denoted \odot). We can spell out the entries of the Hadamard product of two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \quad (2.3.4)$$

```
A * B
```

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

Adding or multiplying a scalar and a tensor produces a result with the same shape as the original tensor. Here, each element of the tensor is added to (or multiplied by) the scalar.

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

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

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

2.3.6 Reduction

Often, we wish to calculate the sum of a tensor's elements. To express the sum of the elements in a vector \mathbf{x} of length n , we write $\sum_{i=1}^n x_i$. There is a simple function for it:

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

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

To express sums over the elements of tensors of arbitrary shape, we simply sum over all its axes. 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()
```

```
(torch.Size([2, 3]), tensor(15.))
```

By default, invoking the sum function *reduces* a tensor along all of its axes, eventually producing a scalar. Our libraries also allow us to specify the axes along which the tensor should be reduced. To sum over all elements along the rows (axis 0), we specify `axis=0` in `sum`. Since the input matrix reduces along axis 0 to generate the output vector, this axis is missing from the shape of the output.

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

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

Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns.

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

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

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]) == A.sum() # Same as A.sum()
```

```
tensor(True)
```

A related quantity is the *mean*, also called the *average*. We calculate the mean by dividing the sum by the total number of elements. Because computing the mean is so common, it gets a dedicated library function that works analogously to `sum`.

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

```
(tensor(2.5000), tensor(2.5000))
```

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

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

```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

2.3.7 Non-Reduction Sum

Sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean. This matters when we want to use the broadcast mechanism.

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

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

For instance, since `sum_A` keeps its two axes after summing each row, we can divide `A` by `sum_A` with broadcasting to create a matrix where each row sums up to 1.

```
A / sum_A
```

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

If we want to calculate the cumulative sum of elements of \mathbf{A} along some axis, say $\text{axis}=0$ (row by row), we can call the `cumsum` function. By design, this function does not reduce the input tensor along any axis.

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

```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

2.3.8 Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra would not deserve its own section. Fortunately, this is where things get more interesting. One of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (also known as *inner product*, $\langle \mathbf{x}, \mathbf{y} \rangle$) 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 = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

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

Equivalently, we can calculate the dot product of two vectors by performing an elementwise multiplication followed by a sum:

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

```
tensor(3.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector $\mathbf{x} \in \mathbb{R}^n$, and a set of weights, denoted by $\mathbf{w} \in \mathbb{R}^n$, the weighted sum of the values in \mathbf{x} according to the weights \mathbf{w} could be expressed as the dot product $\mathbf{x}^\top \mathbf{w}$. When the weights are nonnegative and sum to 1, i.e., $(\sum_{i=1}^n w_i = 1)$, the dot product expresses a *weighted average*. After normalizing two vectors to have unit length, the dot products express the cosine of the angle between them. Later in this section, we will formally introduce this notion of *length*.

2.3.9 Matrix–Vector Products

Now that we know how to calculate dot products, we can begin to understand the *product* between an $m \times n$ matrix \mathbf{A} and an n -dimensional vector \mathbf{x} . To start off, we visualize our

matrix in terms of its row vectors

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

where each $\mathbf{a}_i^\top \in \mathbb{R}^n$ is a row vector representing the i^{th} row of the matrix \mathbf{A} .

The matrix–vector product $\mathbf{A}\mathbf{x}$ is simply a column vector of length m , whose i^{th} element is the dot product $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \quad (2.3.6)$$

We can think of multiplication with 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 are remarkably useful. For example, we can represent rotations as multiplications by certain square matrices. Matrix–vector products also describe the key calculation involved in computing the outputs of each layer in a neural network given the outputs from the previous layer.

To express a matrix–vector product in code, we use the `mv` function. Note that the column dimension of \mathbf{A} (its length along axis 1) must be the same as the dimension of \mathbf{x} (its length). Python has a convenience operator `@` that can execute both matrix–vector and matrix–matrix products (depending on its arguments). Thus we can write `A@x`.

```
A.shape, x.shape, torch.mv(A, x), A@x
```

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

2.3.10 Matrix–Matrix Multiplication

Once 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 $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \quad (2.3.7)$$

Let $\mathbf{a}_i^\top \in \mathbb{R}^k$ denote the row vector representing the i^{th} row of the matrix \mathbf{A} and let $\mathbf{b}_j \in \mathbb{R}^k$

denote the column vector from the j^{th} column of the matrix \mathbf{B} :

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

To form the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$, we simply compute each element c_{ij} as the dot product between the i^{th} row of \mathbf{A} and the j^{th} column of \mathbf{B} , i.e., $\mathbf{a}_i^\top \mathbf{b}_j$:

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

We can think of the matrix–matrix multiplication \mathbf{AB} as performing m matrix–vector products or $m \times n$ dot products and stitching the results together to form an $n \times m$ matrix. In the following snippet, we perform matrix multiplication on \mathbf{A} and \mathbf{B} . Here, \mathbf{A} is a matrix with two rows and three columns, and \mathbf{B} is a matrix with three rows and four columns. After multiplication, we obtain a matrix with two rows and four columns.

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

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

The term *matrix–matrix multiplication* is often simplified to *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.11 Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* it is. For instance, the ℓ_2 norm measures the (Euclidean) length of a vector. Here, we are employing a notion of *size* that concerns the magnitude of a vector’s components (not its dimensionality).

A norm is a function $\|\cdot\|$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector \mathbf{x} , if we scale (all elements of) the vector by a scalar $\alpha \in \mathbb{R}$, its norm scales accordingly:

$$\|\alpha \mathbf{x}\| = |\alpha| \|\mathbf{x}\|. \quad (2.3.10)$$

2. For any vectors \mathbf{x} and \mathbf{y} : norms satisfy the triangle inequality:

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|. \quad (2.3.11)$$

3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:

$$\|\mathbf{x}\| > 0 \text{ for all } \mathbf{x} \neq \mathbf{0}. \quad (2.3.12)$$

Many functions are valid norms and different norms encode different notions of size. The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of a right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called the ℓ_2 norm and expressed as

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}. \quad (2.3.13)$$

The method norm calculates the ℓ_2 norm.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

The ℓ_1 norm is also common and the associated measure is called the Manhattan distance. By definition, the ℓ_1 norm sums the absolute values of a vector's elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|. \quad (2.3.14)$$

Compared to the ℓ_2 norm, it is less sensitive to outliers. To compute the ℓ_1 norm, we compose the absolute value with the sum operation.

```
torch.abs(u).sum()
```

```
tensor(7.)
```

Both the ℓ_2 and ℓ_1 norms are special cases of the more general ℓ_p norms:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}. \quad (2.3.15)$$

In the case of matrices, matters are more complicated. After all, matrices can be viewed both as collections of individual entries *and* as objects that operate on vectors and transform them into other vectors. For instance, we can ask by how much longer the matrix–vector product $\mathbf{X}\mathbf{v}$ could be relative to \mathbf{v} . This line of thought leads to what is called the *spectral*

norm. For now, we introduce the *Frobenius norm*, which is much easier to compute and defined as the square root of the sum of the squares of a matrix's elements:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}. \quad (2.3.16)$$

The Frobenius norm behaves as if it were an ℓ_2 norm of a matrix-shaped vector. Invoking the following function will calculate the Frobenius norm of a matrix.

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

While we do not want to get too far ahead of ourselves, we already can plant some intuition about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *maximize* the revenue associated with a recommender model; *minimize* the distance between predictions and the ground truth observations; *minimize* the distance between representations of photos of the same person while *maximizing* the distance between representations of photos of different people. These distances, which constitute the objectives of deep learning algorithms, are often expressed as norms.

2.3.12 Discussion

In this section, we have reviewed all the linear algebra that you will need to understand a significant chunk of modern deep learning. There is a lot more to linear algebra, though, and much of it 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 more inclined to learn more mathematics once you have gotten your hands dirty applying machine learning to real datasets. So while we reserve the right to introduce more mathematics later on, we wrap up this section here.

If you are eager to learn more linear algebra, there are many excellent books and online resources. For a more advanced crash course, consider checking out Strang (1993), Kolter (2008), and Petersen and Pedersen (2008).

To recap:

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as sum and mean, respectively.

- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the ℓ_1 and ℓ_2 norms, and common matrix norms include the *spectral* and *Frobenius* norms.

2.3.13 Exercises

1. Prove that the transpose of the transpose of a matrix is the matrix itself: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Given two matrices \mathbf{A} and \mathbf{B} , show that sum and transposition commute: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Given any square matrix \mathbf{A} , is $\mathbf{A} + \mathbf{A}^\top$ always symmetric? Can you prove the result by using only the results of the previous two exercises?
4. We defined the tensor \mathbf{X} of shape (2, 3, 4) in this section. What is the output of `len(X)`? Write your answer without implementing any code, then check your answer using code.
5. For a tensor \mathbf{X} of arbitrary shape, does `len(X)` always correspond to the length of a certain axis of \mathbf{X} ? What is that axis?
6. Run `A / A.sum(axis=1)` and see what happens. Can you analyze the results?
7. When traveling between two points in downtown 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 of shape (2, 3, 4). What are the shapes of the summation outputs along axes 0, 1, and 2?
9. Feed a tensor with three or more axes to the `linalg.norm` function and observe its output. What does this function compute for tensors of arbitrary shape?
10. Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$ and $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{14}}$, initialized with Gaussian random variables. You want to compute the product \mathbf{ABC} . Is there any difference in memory footprint and speed, depending on whether you compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. Why?
11. Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$ and $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{16}}$. Is there any difference in speed depending on whether you compute \mathbf{AB} or \mathbf{AC}^\top ? Why? What changes if you initialize $\mathbf{C} = \mathbf{B}^\top$ without cloning memory? Why?
12. Consider three matrices, say $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{100 \times 200}$. Construct a tensor with three axes by

stacking $[A, B, C]$. What is the dimensionality? Slice out the second coordinate of the third axis to recover B . Check that your answer is correct.



Discussions⁵⁵.

2.4 Calculus

For a long time, how to calculate the area of a circle remained a mystery. Then, in Ancient Greece, the mathematician Archimedes came up with the clever idea to inscribe a series of polygons with increasing numbers of vertices on the inside of a circle (Fig. 2.4.1). For a polygon with n vertices, we obtain n triangles. The height of each triangle approaches the radius r as we partition the circle more finely. At the same time, its base approaches $2\pi r/n$, since the ratio between arc and secant approaches 1 for a large number of vertices. Thus, the area of the polygon approaches $n \cdot r \cdot \frac{1}{2} (2\pi r/n) = \pi r^2$.

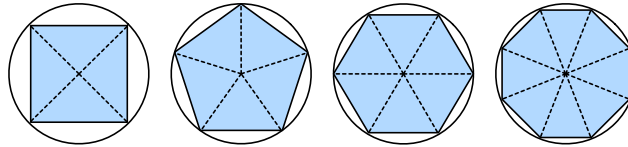


Fig. 2.4.1 Finding the area of a circle as a limit procedure.

This limiting procedure is at the root of both *differential calculus* and *integral calculus*. The former can tell us how to increase or decrease a function's value by manipulating its arguments. This comes in handy for the *optimization problems* that we face in deep learning, where we repeatedly update our parameters in order to decrease the loss function. Optimization addresses how to fit our models to training data, and calculus is its key prerequisite. However, do not forget that our ultimate goal is to perform well on *previously unseen* data. That problem is called *generalization* and will be a key focus of other chapters.

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
from d2l import torch as d2l
```

2.4.1 Derivatives and Differentiation

Put simply, a *derivative* is the rate of change in a function with respect to changes in its arguments. Derivatives can tell us how rapidly a loss function would increase or decrease were we to *increase* or *decrease* each parameter by an infinitesimally small amount. Formally, for functions $f : \mathbb{R} \rightarrow \mathbb{R}$, that map from scalars to scalars, the *derivative* of f at a point x is defined as

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}. \quad (2.4.1)$$

This term on the right hand side is called a *limit* and it tells us what happens to the value of an expression as a specified variable approaches a particular value. This limit tells us what the ratio between a perturbation h and the change in the function value $f(x+h) - f(x)$ converges to as we shrink its size to zero.

When $f'(x)$ exists, f is said to be *differentiable* at x ; and when $f'(x)$ exists for all x on a set, e.g., the interval $[a, b]$, we say that f is differentiable on this set. Not all functions are differentiable, including many that we wish to optimize, such as accuracy and the area under the receiving operating characteristic (AUC). However, because computing the derivative of the loss is a crucial step in nearly all algorithms for training deep neural networks, we often optimize a differentiable *surrogate* instead.

We can interpret the derivative $f'(x)$ as the *instantaneous* rate of change of $f(x)$ with respect to x . Let's develop some intuition with an example. Define $u = f(x) = 3x^2 - 4x$.

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

Setting $x = 1$, we see that $\frac{f(x+h)-f(x)}{h}$ approaches 2 as h approaches 0. While this experiment lacks the rigor of a mathematical proof, we can quickly see that indeed $f'(1) = 2$.

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

```
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
```

There are several equivalent notational conventions for derivatives. Given $y = f(x)$, the following expressions are equivalent:

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

where the symbols $\frac{d}{dx}$ and D are *differentiation operators*. Below, we present the derivatives of some common functions:

$$\begin{aligned} \frac{d}{dx}C &= 0 && \text{for any constant } C \\ \frac{d}{dx}x^n &= nx^{n-1} && \text{for } n \neq 0 \\ \frac{d}{dx}e^x &= e^x \\ \frac{d}{dx}\ln x &= x^{-1}. \end{aligned} \quad (2.4.3)$$

Functions composed from differentiable functions are often themselves differentiable. The following rules come in handy for working with compositions of any differentiable functions f and g , and constant C .

$$\begin{aligned}
 \frac{d}{dx}[Cf(x)] &= C \frac{d}{dx}f(x) && \text{Constant multiple rule} \\
 \frac{d}{dx}[f(x) + g(x)] &= \frac{d}{dx}f(x) + \frac{d}{dx}g(x) && \text{Sum rule} \\
 \frac{d}{dx}[f(x)g(x)] &= f(x) \frac{d}{dx}g(x) + g(x) \frac{d}{dx}f(x) && \text{Product rule} \\
 \frac{d}{dx} \frac{f(x)}{g(x)} &= \frac{g(x) \frac{d}{dx}f(x) - f(x) \frac{d}{dx}g(x)}{g^2(x)} && \text{Quotient rule}
 \end{aligned} \tag{2.4.4}$$

Using this, we can apply the rules to find the derivative of $3x^2 - 4x$ via

$$\frac{d}{dx}[3x^2 - 4x] = 3 \frac{d}{dx}x^2 - 4 \frac{d}{dx}x = 6x - 4. \tag{2.4.5}$$

Plugging in $x = 1$ shows that, indeed, the derivative equals 2 at this location. Note that derivatives tell us the *slope* of a function at a particular location.

2.4.2 Visualization Utilities

We can visualize the slopes of functions using the `matplotlib` library. We need to define a few functions. As its name indicates, `use_svg_display` tells `matplotlib` to output graphics in SVG format for crisper images. The comment `#@save` is a special modifier that allows us to save any function, class, or other code block to the `d2l` package so that we can invoke it later without repeating the code, e.g., via `d2l.use_svg_display()`.

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

Conveniently, we can set figure sizes with `set_figsize`. Since the import statement from `matplotlib` `import pyplot as plt` was marked via `#@save` in the `d2l` package, we can call `d2l.plt`.

```
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 `set_axes` function can associate axes with properties, including labels, ranges, and scales.

```
#@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)
```

(continues on next page)

(continued from previous page)

```

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, we can define a plot function to overlay multiple curves. Much of the code here is just ensuring that the sizes and shapes of inputs match.

```

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

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

    if has_one_axis(X): X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)

    set_figsize(figsize)
    if axes is None:
        axes = d2l.plt.gca()
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        axes.plot(x,y,fmt) if len(x) else axes.plot(y,fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)

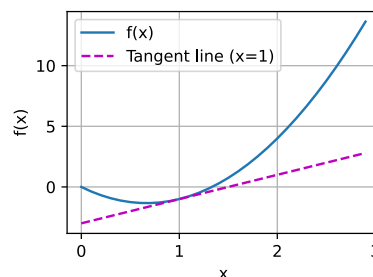
```

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)'])

```



2.4.3 Partial Derivatives and Gradients

Thus far, we have been differentiating functions of just one variable. In deep learning, we also need to work with functions of *many* variables. We briefly introduce notions of the derivative that apply to such *multivariate* functions.

Let $y = f(x_1, x_2, \dots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its i^{th} parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \quad (2.4.6)$$

To calculate $\frac{\partial y}{\partial x_i}$, we can treat $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ as constants and calculate the derivative of y with respect to x_i . The following notational conventions for partial derivatives are all common and all mean the same thing:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_{x_i} f = \partial_i f = f_{x_i} = f_i = D_i f = D_{x_i} f. \quad (2.4.7)$$

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

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = [\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \dots, \partial_{x_n} f(\mathbf{x})]^T. \quad (2.4.8)$$

When there is no ambiguity, $\nabla_{\mathbf{x}} f(\mathbf{x})$ is typically replaced by $\nabla f(\mathbf{x})$. The following rules come in handy for differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$ we have $\nabla_{\mathbf{x}} \mathbf{A} \mathbf{x} = \mathbf{A}^T$ and $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A} = \mathbf{A}$.
- For square matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ we have that $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^T) \mathbf{x}$ and in particular $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix \mathbf{X} , we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$.

2.4.4 Chain Rule

In deep learning, the gradients of concern are often difficult to calculate because we are working with deeply nested functions (of functions (of functions...)). Fortunately, the *chain rule* takes care of this. Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}. \quad (2.4.9)$$

Turning back to multivariate functions, suppose that $y = f(\mathbf{u})$ has variables u_1, u_2, \dots, u_m , where each $u_i = g_i(\mathbf{x})$ has variables x_1, x_2, \dots, x_n , i.e., $\mathbf{u} = g(\mathbf{x})$. Then the chain rule states that

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1} \frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2} \frac{\partial u_2}{\partial x_i} + \dots + \frac{\partial y}{\partial u_m} \frac{\partial u_m}{\partial x_i} \text{ and so } \nabla_{\mathbf{x}} y = \mathbf{A} \nabla_{\mathbf{u}} y, \quad (2.4.10)$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$ is a *matrix* that contains the derivative of vector \mathbf{u} with respect to vector \mathbf{x} . Thus, evaluating the gradient requires computing a vector–matrix product. This is one of the key reasons why linear algebra is such an integral building block in building deep learning systems.

2.4.5 Discussion

While we have just scratched the surface of a deep topic, a number of concepts already come into focus: first, the composition rules for differentiation can be applied routinely, enabling us to compute gradients *automatically*. This task requires no creativity and thus we can focus our cognitive powers elsewhere. Second, computing the derivatives of vector-valued functions requires us to multiply matrices as we trace the dependency graph of variables from output to input. In particular, this graph is traversed in a *forward* direction when we evaluate a function and in a *backwards* direction when we compute gradients. Later chapters will formally introduce backpropagation, a computational procedure for applying the chain rule.

From the viewpoint of optimization, gradients allow us to determine how to move the parameters of a model in order to lower the loss, and each step of the optimization algorithms used throughout this book will require calculating the gradient.

2.4.6 Exercises

1. So far we took the rules for derivatives for granted. Using the definition and limits prove the properties for (i) $f(x) = c$, (ii) $f(x) = x^n$, (iii) $f(x) = e^x$ and (iv) $f(x) = \log x$.
2. In the same vein, prove the product, sum, and quotient rule from first principles.
3. Prove that the constant multiple rule follows as a special case of the product rule.
4. Calculate the derivative of $f(x) = x^x$.
5. What does it mean that $f'(x) = 0$ for some x ? Give an example of a function f and a location x for which this might hold.
6. Plot the function $y = f(x) = x^3 - \frac{1}{x}$ and plot its tangent line at $x = 1$.
7. Find the gradient of the function $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
8. What is the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|_2$? What happens for $\mathbf{x} = \mathbf{0}$?
9. 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 $z = z(a, b)$?
10. Given a function $f(x)$ that is invertible, compute the derivative of its inverse $f^{-1}(x)$. Here we have that $f^{-1}(f(x)) = x$ and conversely $f(f^{-1}(y)) = y$. Hint: use these properties in your derivation.

56

Discussions⁵⁶.

2.5 Automatic Differentiation

Recall from Section 2.4 that calculating derivatives is the crucial step in all the optimization algorithms that we will use to train deep networks. While the calculations are straightforward, working them out by hand can be tedious and error-prone, and these issues only grow as our models become more complex.

Fortunately all modern deep learning frameworks take this work off our plates by offering *automatic differentiation* (often shortened to *autograd*). As we pass data through each successive function, the framework builds a *computational graph* that tracks how each value depends on others. To calculate derivatives, automatic differentiation works backwards through this graph applying the chain rule. The computational algorithm for applying the chain rule in this fashion is called *backpropagation*.

While autograd libraries have become a hot concern over the past decade, they have a long history. In fact the earliest references to autograd date back over half of a century (Wengert, 1964). The core ideas behind modern backpropagation date to a PhD thesis from 1980 (Speelpenning, 1980) and were further developed in the late 1980s (Griewank, 1989). While backpropagation has become the default method for computing gradients, it is not the only option. For instance, the Julia programming language employs forward propagation (Revels *et al.*, 2016). Before exploring methods, let's first master the autograd package.

```
import torch
```

2.5.1 A Simple Function

Let's assume 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, we assign \mathbf{x} an initial value.

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

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

Before we calculate the gradient of y with respect to \mathbf{x} , we need a place to store it. In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory. Note that the gradient of a scalar-valued function with respect to a vector \mathbf{x} is vector-valued with the same shape as \mathbf{x} .