# Contents

**Acknowledgments** **vii**

**Notation** **ix**

**1 Introduction** **1**
1.1 Who Should Read This Book? . . . . . 8
1.2 Historical Trends in Deep Learning . . . . . 11

**I Applied Math and Machine Learning Basics** **26**

**2 Linear Algebra** **28**
2.1 Scalars, Vectors, Matrices and Tensors . . . . . 28
2.2 Multiplying Matrices and Vectors . . . . . 30
2.3 Identity and Inverse Matrices . . . . . 32
2.4 Linear Dependence and Span . . . . . 33
2.5 Norms . . . . . 35
2.6 Special Kinds of Matrices and Vectors . . . . . 36
2.7 Eigendecomposition . . . . . 38
2.8 Singular Value Decomposition . . . . . 40
2.9 The Moore-Penrose Pseudoinverse . . . . . 41
2.10 The Trace Operator . . . . . 42
2.11 Determinant . . . . . 43
2.12 Example: Principal Components Analysis . . . . . 43

**3 Probability and Information Theory** **48**
3.1 Why Probability? . . . . . 48
3.2 Random Variables . . . . . 51
3.3 Probability Distributions . . . . . 51
3.4 Marginal Probability . . . . . 53
3.5 Conditional Probability . . . . . 53

3.6 The Chain Rule of Conditional Probabilities . . . . . 54
3.7 Independence and Conditional Independence . . . . . 54
3.8 Expectation, Variance and Covariance . . . . . 55
3.9 Information Theory . . . . . 56
3.10 Common Probability Distributions . . . . . 59
3.11 Useful Properties of Common Functions . . . . . 65
3.12 Bayes' Rule . . . . . 67
3.13 Technical Details of Continuous Variables . . . . . 67
3.14 Structured Probabilistic Models . . . . . 68
3.15 Example: Naive Bayes . . . . . 71

**4 Numerical Computation 77**
4.1 Overflow and Underflow . . . . . 77
4.2 Poor Conditioning . . . . . 78
4.3 Gradient-Based Optimization . . . . . 79
4.4 Constrained Optimization . . . . . 88
4.5 Example: Linear Least Squares . . . . . 90

**5 Machine Learning Basics 92**
5.1 Learning Algorithms . . . . . 92
5.2 Example: Linear Regression . . . . . 100
5.3 Generalization, Capacity, Overfitting and Underfitting . . . . . 103
5.4 Hyperparameters and Validation Sets . . . . . 113
5.5 Estimators, Bias and Variance . . . . . 115
5.6 Maximum Likelihood Estimation . . . . . 124
5.7 Bayesian Statistics . . . . . 127
5.8 Supervised Learning Algorithms . . . . . 134
5.9 Unsupervised Learning Algorithms . . . . . 139
5.10 Weakly Supervised Learning . . . . . 142
5.11 Building a Machine Learning Algorithm . . . . . 143
5.12 The Curse of Dimensionality and Statistical Limitations of Local Generalization . . . . . 145

**II Deep Networks: Modern Practices 156**

**6 Feedforward Deep Networks 158**
6.1 MLPs from the 1980's . . . . . 159
6.2 Estimating Conditional Statistics . . . . . 163
6.3 Parametrizing a Learned Predictor . . . . . 163
6.4 Flow Graphs and Back-Propagation . . . . . 175

6.5 Back-propagation through Random Operations and Graphical Models . . . 188
6.6 Universal Approximation Properties and Depth . . . 192
6.7 Feature / Representation Learning . . . 195
6.8 Piecewise Linear Hidden Units . . . 197
6.9 Historical Notes . . . 199

**7 Regularization of Deep or Distributed Models 201**
7.1 Regularization from a Bayesian Perspective . . . 203
7.2 Classical Regularization: Parameter Norm Penalty . . . 204
7.3 Classical Regularization as Constrained Optimization . . . 212
7.4 Regularization and Under-Constrained Problems . . . 213
7.5 Dataset Augmentation . . . 214
7.6 Classical Regularization as Noise Robustness . . . 216
7.7 Early Stopping as a Form of Regularization . . . 220
7.8 Parameter Tying and Parameter Sharing . . . 227
7.9 Sparse Representations . . . 228
7.10 Bagging and Other Ensemble Methods . . . 230
7.11 Dropout . . . 232
7.12 Multi-Task Learning . . . 235
7.13 Adversarial Training . . . 237

**8 Optimization for Training Deep Models 239**
8.1 Optimization for Model Training . . . 240
8.2 Challenges in Neural Network Optimization . . . 245
8.3 Optimization Algorithms I: Basic Algorithms . . . 258
8.4 Optimization Algorithms II: Adaptive Learning Rates . . . 264
8.5 Optimization Algorithms III: Approximate Second-Order Methods 270
8.6 Optimization Algorithms IV: Natural Gradient Methods . . . 278
8.7 Optimization Strategies and Meta-Algorithms . . . 281

**9 Convolutional Networks 294**
9.1 The Convolution Operation . . . 295
9.2 Motivation . . . 298
9.3 Pooling . . . 304
9.4 Convolution and Pooling as an Infinitely Strong Prior . . . 306
9.5 Variants of the Basic Convolution Function . . . 308
9.6 Structured Outputs . . . 314
9.7 Data Types . . . 315
9.8 Efficient Convolution Algorithms . . . 317

9.9 Random or Unsupervised Features . . . . . 318
9.10 The Neuroscientific Basis for Convolutional Networks . . . . . 319
9.11 Convolutional Networks and the History of Deep Learning . . . . 326

**10 Sequence Modeling: Recurrent and Recursive Nets 327**
10.1 Unfolding Flow Graphs and Sharing Parameters . . . . . 328
10.2 Recurrent Neural Networks . . . . . 330
10.3 Bidirectional RNNs . . . . . 344
10.4 Encoder-Decoder Sequence-to-Sequence Architectures . . . . . 344
10.5 Deep Recurrent Networks . . . . . 346
10.6 Recursive Neural Networks . . . . . 348
10.7 The Challenge of Long-Term Dependencies . . . . . 349

**11 Practical methodology 367**
11.1 Default Baseline Models . . . . . 369
11.2 Selecting Hyperparameters . . . . . 370
11.3 Debugging Strategies . . . . . 379

**12 Applications 384**
12.1 Large Scale Deep Learning . . . . . 384
12.2 Computer Vision . . . . . 392
12.3 Speech Recognition . . . . . 397
12.4 Natural Language Processing and Neural Language Models . . . 401
12.5 Structured Outputs . . . . . 417
12.6 Other Applications . . . . . 419

**III Deep Learning Research 428**

**13 Linear Factor Models and Autoencoders 431**
13.1 Regularized Autoencoders . . . . . 432
13.2 Representational Power, Layer Size and Depth . . . . . 437
13.3 Reconstruction Distribution . . . . . 437
13.4 Linear Factor Models . . . . . 439
13.5 Probabilistic PCA and Factor Analysis . . . . . 440
13.6 Reconstruction Error as Log-Likelihood . . . . . 443
13.7 Sparse Representations . . . . . 445
13.8 Denoising Autoencoders . . . . . 449
13.9 Contractive Autoencoders . . . . . 454

**14 Representation Learning 457**
14.1 Greedy Layerwise Unsupervised Pre-Training . . . . . 458

14.2 Transfer Learning and Domain Adaptation . . . . . 465
14.3 Semi-Supervised Learning . . . . . 472
14.4 Semi-Supervised Learning and Disentangling Underlying Causal Factors . . . . . 473
14.5 Assumption of Underlying Factors and Distributed Representation 475
14.6 Exponential Gain in Representational Efficiency from Distributed Representations . . . . . 479
14.7 Exponential Gain in Representational Efficiency from Depth . . . 481
14.8 Priors regarding the Underlying Factors . . . . . 483

**15 The Manifold Perspective on Representation Learning 487**
15.1 Manifold Interpretation of PCA and Linear Auto-Encoders . . . . 495
15.2 Manifold Interpretation of Sparse Coding . . . . . 498
15.3 The Entropy Bias from Maximum Likelihood . . . . . 498
15.4 Manifold Learning via Regularized Auto-Encoders . . . . . 499
15.5 Tangent Distance, Tangent-Prop, and Manifold Tangent Classifier 500

**16 Structured Probabilistic Models for Deep Learning 503**
16.1 The Challenge of Unstructured Modeling . . . . . 504
16.2 Using Graphs to Describe Model Structure . . . . . 508
16.3 Advantages of Structured Modeling . . . . . 523
16.4 Learning about Dependencies . . . . . 523
16.5 Inference and Approximate Inference over Latent Variables . . . . 524
16.6 The Deep Learning Approach to Structured Probabilistic Models 526

**17 Monte Carlo Methods 529**
17.1 Markov Chain Monte Carlo Methods . . . . . 529
17.2 The Difficulty of Mixing between Well-Separated Modes . . . . . 531

**18 Confronting the Partition Function 533**
18.1 The Log-Likelihood Gradient of Energy-Based Models . . . . . 534
18.2 Stochastic Maximum Likelihood and Contrastive Divergence . . . 536
18.3 Pseudolikelihood . . . . . 543
18.4 Score Matching and Ratio Matching . . . . . 545
18.5 Denoising Score Matching . . . . . 547
18.6 Noise-Contrastive Estimation . . . . . 547
18.7 Estimating the Partition Function . . . . . 549

**19 Approximate inference 557**
19.1 Inference as Optimization . . . . . 559
19.2 Expectation Maximization . . . . . 560

19.3 MAP Inference: Sparse Coding as a Probabilistic Model . . . . . 561
19.4 Sequence Modeling with Graphical Models . . . . . 562
19.5 Combining Neural Networks and Search . . . . . 571
19.6 Variational Inference and Learning . . . . . 576
19.7 Stochastic Inference . . . . . 580
19.8 Learned Approximate Inference . . . . . 580

**20 Deep Generative Models 582**
20.1 Boltzmann Machines . . . . . 582
20.2 Restricted Boltzmann Machines . . . . . 585
20.3 Training Restricted Boltzmann Machines . . . . . 588
20.4 Deep Belief Networks . . . . . 592
20.5 Deep Boltzmann Machines . . . . . 595
20.6 Boltzmann Machines for Real-Valued Data . . . . . 606
20.7 Convolutional Boltzmann Machines . . . . . 609
20.8 Other Boltzmann Machines . . . . . 610
20.9 Directed Generative Nets . . . . . 610
20.10 Auto-Regressive Networks . . . . . 613
20.11 A Generative View of Autoencoders . . . . . 617
20.12 Generative Stochastic Networks . . . . . 622
20.13 Methodological Notes . . . . . 624

**Bibliography 628**

**Index 676**

# Acknowledgments

This book would not have been possible without the contributions of many people.

We would like to thank those who commented on our proposal for the book and helped plan its contents and organization: Hugo Larochelle, Guillaume Alain, Kyunghyun Cho, Çağlar Gülçehre, Razvan Pascanu, David Krueger and Thomas Rohée.

We would like to thank the people who offered feedback on the content of the book itself. Some offered feedback on many chapters: Martín Abadi, Julian Serban, Laurent Dinh, Guillaume Alain, Kelvin Xu, Meire Fortunato, Ilya Sutskever, Vincent Vanhoucke, David Warde-Farley, Augustus Q. Odena, David Sussillo, Matko Bošnjak, Stephan Dreseitl, Jurgen Van Gael, Dustin Webb, Johannes Roith, Ion Androutsopoulos, Karl Pichotta, Pawel Chilinski, Halis Sak, Frédéric Francis, Jonathan Hunt and Grigory Sapunov.

We would also like to thank those who provided us with useful feedback on individual chapters:

- Chapter 1, Introduction: Johannes Roith, Eric Morris, Samira Ebrahimi, Ozan Çağlayan and Sebastien Bratieres.
- Chapter 2, Linear Algebra: Pierre Luc Carrier, Li Yao, Thomas Rohée, Colby Toland, Amjad Almahairi, Sergey Oreshkov, István Petrás, Dennis Prangle Alessandro Vitale Nikola Banić and Eric Fosler-Lussier.
- Chapter 3, Probability and Information Theory: Rasmus Antti, Stephan Gouws, Vincent Dumoulin, Artem Oboturov, Li Yao, John Philip Anderson, Rui Fa, Kai Arulkumaran, and Miao Fan.
- Chapter 4, Numerical Computation: Tran Lam An.
- Chapter 5, Machine Learning Basics: Dzmitry Bahdanau and Zheng Sun.
- Chapter 6, Feedforward Deep Networks: David Krueger.
- Chapter 7, Regularization of Deep or Distributed Models: Wei Xue.

- Chapter 8, Optimization for Training Deep Models: James Martens and Marcel Ackermann.
- Chapter 9, Convolutional Networks: Mehdi Mirza, Çağlar Gülçehre and Martín Arjovsky.
- Chapter 10, Sequence Modeling: Recurrent and Recursive Nets: Mihaela Rosca, Razvan Pascanu, Dmitriy Serdyuk and Dongyu Shi.
- Chapter 18, Confronting the Partition Function: Sam Bowman and Ozan Çağlayan.
- Chapter 20, Deep Generative Models: Fady Medhat
- Bibliography, Leslie N. Smith.

We also want to thank those who allowed us to reproduce images, figures or data from their publications: David Warde-Farley, Matthew D. Zeiler, Rob Fergus, Chris Olah, Jason Yosinski, Nicolas Chapados and James Bergstra. We indicate their contributions in the figure captions throughout the text.

Finally, we would like to thank Google for allowing Ian Goodfellow to work on the book as his 20% project while at Google. In particular, we would like to thank Ian's former manager, Greg Corrado, and his subsequent manager, Samy Bengio, for their support of this effort.

# Notation

This section provides a concise reference describing the notation used throughout this book. If you are unfamiliar with any of these mathematical concepts, this notation reference may seem intimidating. However, do not despair, we describe most of these ideas in chapters 1-3.

**Numbers and Arrays**

| | |
|---|---|
| $a$ | A scalar (integer or real) value with the name "a" |
| $\boldsymbol{a}$ | A vector with the name "a" |
| $\boldsymbol{A}$ | A matrix with the name "A" |
| $\mathsf{A}$ | A tensor with the name "A" |
| $\boldsymbol{I}_n$ | Identity matrix with $n$ rows and $n$ columns |
| $\boldsymbol{I}$ | Identity matrix with dimensionality implied by context |
| $\boldsymbol{e}_i$ | Standard basis vector $[0, \ldots, 0, 1, 0, \ldots, 0]$ with a 1 at position $i$. |
| $\text{diag}(\boldsymbol{a})$ | A square, diagonal matrix with entries given by $\boldsymbol{a}$ |
| $\mathrm{a}$ | A scalar random variable with the name "a" |
| $\mathbf{a}$ | A vector-valued random variable with the name "a" |
| $\mathbf{A}$ | A matrix-valued random variable with the name "A" |

## Sets and Graphs

| | |
|---|---|
| $\mathbb{A}$ | A set with the name "A" |
| $\mathbb{R}$ | The set of real numbers |
| $\{0, 1\}$ | The set containing 0 and 1 |
| $\{0, 1, \ldots, n\}$ | The set of all integers between 0 and $n$ |
| $[a, b]$ | The real interval including $a$ and $b$ |
| $(a, b]$ | The real interval excluding $a$ but including $b$ |
| $\mathbb{A} \backslash \mathbb{B}$ | Set subtraction, i.e., the elements of $\mathbb{A}$ that are not in $\mathbb{B}$ |
| $\mathcal{G}$ | A graph with the name "G" |
| $Pa_{\mathcal{G}}(\mathrm{x}_i)$ | The parents of $\mathrm{x}_i$ in $\mathcal{G}$. |

## Indexing

| | |
|---|---|
| $a_i$ | Element $i$ of vector $\boldsymbol{a}$, with indexing starting at 1 |
| $a_{-i}$ | All elements of vector $a$ except for element $i$ |
| $A_{i,j}$ | Element $i, j$ of matrix $\boldsymbol{A}$ |
| $\boldsymbol{A}_{i,:}$ | Row $i$ of matrix $\boldsymbol{A}$ |
| $\boldsymbol{A}_{:,i}$ | Column $i$ of matrix $\boldsymbol{A}$ |
| $\mathbf{A}_{i,j,k}$ | Element $(i, j, k)$ of a 3-D tensor $\mathbf{A}$ |
| $\mathbf{A}_{:,:,i}$ | 2-D slice of a 3-D tensor |
| $\mathrm{a}_i$ | Element $i$ of the random vector $\mathbf{a}$ |

## Linear Algebra Operations

| | |
|---|---|
| $\boldsymbol{A}^\top$ | Transpose of matrix $\boldsymbol{A}$ |
| $\boldsymbol{A}^+$ | Moore-Penrose pseudoinverse of $\boldsymbol{A}$ |
| $\boldsymbol{A} \odot \boldsymbol{B}$ | Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$ |

## Calculus

| | |
|---|---|
| $\frac{dy}{dx}$ | Derivative of $y$ with respect to $x$ |
| $\frac{\partial y}{\partial x}$ | Partial derivative of $y$ with respect to $x$ |
| $\nabla \ y$ | Gradient of $y$ with respect to $x$ |
| $\nabla \ \ y$ | Matrix derivatives of $y$ with respect to $x$ |
| $\frac{\partial f}{\partial \boldsymbol{x}}$ | Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ |
| $\nabla^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$ | The Hessian matrix of $f$ at input point $\boldsymbol{x}$ |
| $f(\boldsymbol{x})d\boldsymbol{x}$ | Definite integral over the entire domain of $\boldsymbol{x}$ |
| $\int_{\mathbb{S}} f(\boldsymbol{x})d\boldsymbol{x}$ | Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$ |

## Probability and Information Theory

| | |
|---|---|
| $\mathrm{a} \perp \mathrm{b}$ | The random variables a and b are independent. |
| $\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$ | They are are conditionally independent given c. |
| $\mathbb{E}_{x \sim P}[f(x)]$ or $\mathbb{E} f(x)$ | Expectation of $f(x)$ with respect to $P(\mathrm{x})$ |
| $\mathrm{Var}(f(x))$ | Variance of $f(x)$ under $P(\mathrm{x})$ |
| $\mathrm{Cov}(f(x), g(x))$ | Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x}, \mathrm{y})$ |
| $H(\mathrm{x})$ | Shannon entropy of the random variable x |
| $D_{\mathrm{KL}}(P \| Q)$ | Kullback-Leibler divergence of P and Q |

### Functions

| | |
|---|---|
| $f \circ g$ | Composition of the functions $f$ and $g$ |
| $f(\boldsymbol{x}; \boldsymbol{\theta})$ | A function of $\boldsymbol{x}$ parameterized by $\boldsymbol{\theta}$ |
| $\log x$ | Natural logarithm of $x$ |
| $\sigma(x)$ | Logistic sigmoid, $1/(1 + \exp(-x))$ |
| $\zeta(x)$ | Softplus, $\log(1 + \exp(x))$ |
| $\|\|\boldsymbol{x}\|\|_p$ | $L^p$ norm of $\boldsymbol{x}$ |
| $x^+$ | Positive part of $x$, i.e., $\max(0, x)$ |
| $\mathbf{1}_{condition}$ | is 1 if the condition is true, 0 otherwise. |

Sometimes we write $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathbf{X})$, when $f$ is a function of a scalar rather than a vector, matrix, or tensor. In this case, we mean to apply $f$ to the array element-wise. For example, if $\mathbf{C} = \sigma(\mathbf{X})$, then $C_{i,j,k} = \sigma(X_{i,j,k})$ for all valid values of $i$, $j$ and $k$.

### Datasets and distributions

| | |
|---|---|
| $\mathbb{X}$ | A set of training examples |
| $\boldsymbol{x}^{(i)}$ | The $i$-th example (input) from a dataset |
| $y^{(i)}$ or $\boldsymbol{y}^{(i)}$ | The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning |
| $\boldsymbol{X}$ | The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$ |

# Chapter 1

# Introduction

Inventors have long dreamed of creating machines that think. Ancient Greek myths tell of intelligent objects, such as animated statues of human beings and tables that arrive full of food and drink when called.

When programmable computers were first conceived, people wondered whether they might become intelligent, over a hundred years before one was built (Lovelace, 1842). Today, *artificial intelligence (AI)* is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and to support basic scientific research.

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

This book is about a solution to these more intuitive problems. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs. The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI *deep learning*.

Many of the early successes of AI took place in relatively sterile and formal environments and did not require computers to have much knowledge about the

world. For example, IBM's Deep Blue chess-playing system defeated world champion Garry Kasparov in 1997 (Hsu, 2002). Chess is of course a very simple world, containing only sixty-four locations and thirty-two pieces that can move in only rigidly circumscribed ways. Devising a successful chess strategy is a tremendous accomplishment, but the challenge is not due to the difficulty of describing the relevant concepts to the computer. Chess can be completely described by a very brief list of completely formal rules, easily provided ahead of time by the programmer.

Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the best human chess player, but are only recently matching some of the abilities of average human beings to recognize objects or speech. A person's everyday life requires an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer.

Several artificial intelligence projects have sought to hard-code knowledge about the world in formal languages. A computer can reason about statements in these formal languages automatically using logical inference rules. This is known as the *knowledge base* approach to artificial intelligence. None of these projects has lead to a major success. One of the most famous such projects is Cyc (Lenat and Guha, 1989). Cyc is an inference engine and a database of statements in a language called CycL. These statements are entered by a staff of human supervisors. It is an unwieldy process. People struggle to devise formal rules with enough complexity to accurately describe the world. For example, Cyc failed to understand a story about a person named Fred shaving in the morning (Linde, 1992). Its inference engine detected an inconsistency in the story: it knew that people do not have electrical parts, but because Fred was holding an electric razor, it believed the entity "FredWhileShaving" contained electrical parts. It therefore asked whether Fred was still a person while he was shaving.

The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as *machine learning*. The introduction of machine learning allowed computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. A simple machine learning algorithm called *logistic regression* can determine whether to recommend cesarean delivery (Mor-Yosef *et al.*, 1990). A simple machine learning algorithm called *naive Bayes* can separate legitimate e-mail from spam e-mail.

The performance of these simple machine learning algorithms depends heavily on the *representation* of the data they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information, such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a *feature*. Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence the way that the features are defined in any way. If logistic regression was given a 3-D MRI image of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions. Individual voxels[1] in an MRI scan have negligible correlation with any complications that might occur during delivery.

This dependence on representations is a general phenomenon that appears throughout computer science and even daily life. In computer science, operations such as searching a collection of data can proceed exponentially faster if the collection is structured and indexed intelligently. People can easily perform arithmetic on Arabic numerals, but find arithmetic on Roman numerals much more time consuming. It is not surprising that the choice of representation has an enormous effect on the performance of machine learning algorithms. For a simple visual example, see Fig. 1.1.

Many artificial intelligence tasks can be solved by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. For example, a useful feature for speaker identification from sound is the pitch. The pitch can be formally specified—it is the lowest frequency major peak of the spectrogram. It is useful for speaker identification because it is determined by the size of the vocal tract, and therefore gives a strong clue as to whether the speaker is a man, woman, or child.

However, for many tasks, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as *representation learning*. Learned representations of-

[1] A voxel is the value at a single point in a 3-D scan, much as a pixel as the value at a single point in an image.

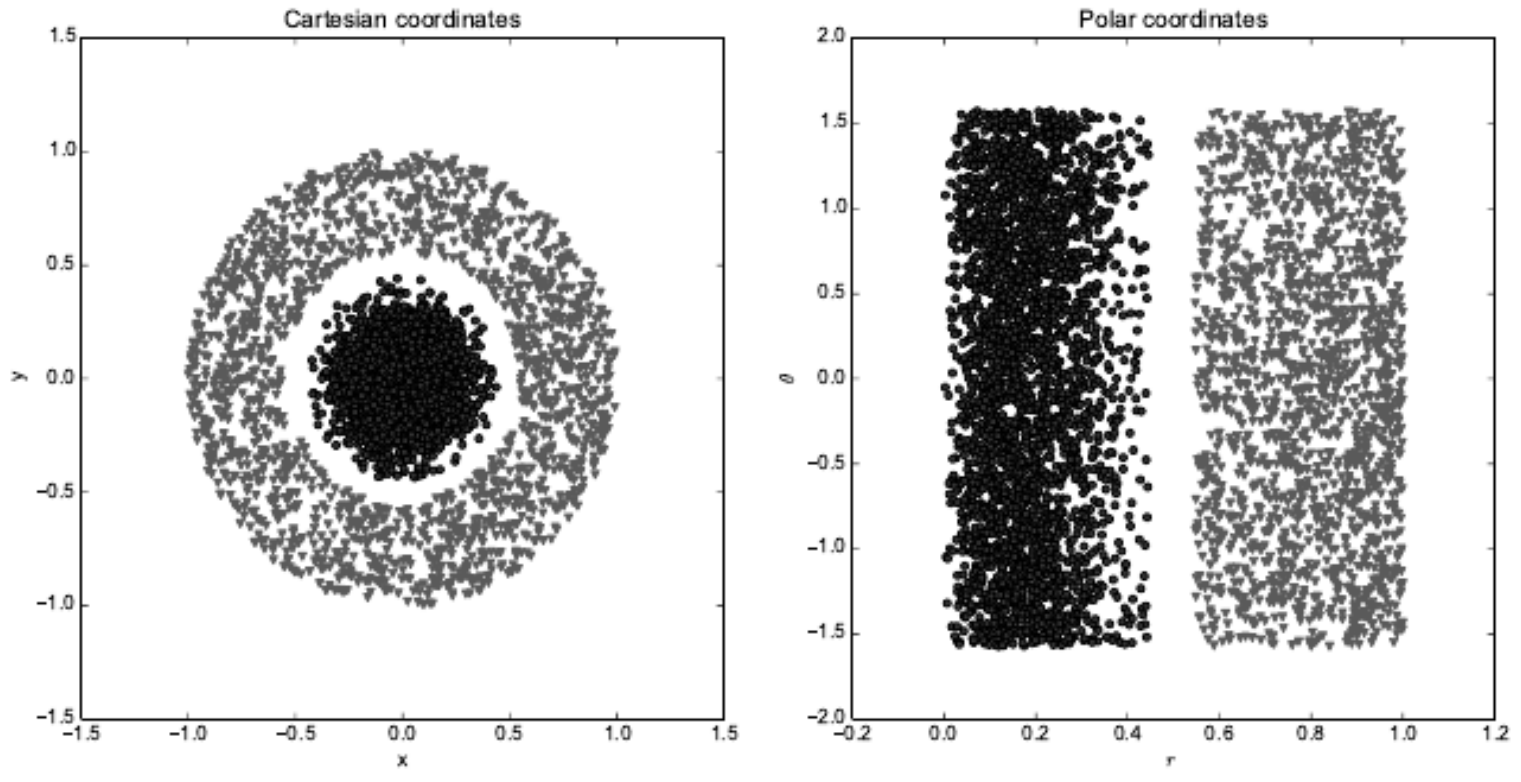


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data with polar coordinates and the task becomes simple to solve with a vertical line. (Figure credit: David Warde-Farley)

ten result in much better performance than can be obtained with hand-designed representations. They also allow AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

The quintessential example of a representation learning algorithm is the *autoencoder*. An autoencoder is the combination of an *encoder* function that converts the input data into a different representation, and a *decoder* function that converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties.

When designing features or algorithms for learning features, our goal is usually to separate the *factors of variation* that explain the observed data. In this context, we use the word "factors" simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not quantities that are directly observed but they may exist either as unobserved objects or forces in the physical world that affect observable quantities, or they are constructs in the human mind that provide useful simplifying explanations

or inferred causes of the observed data. They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data. When analyzing a speech recording, the factors of variation include the speaker's age, their sex, their accent and the words that they are speaking. When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.

A major source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data we are able to observe. The individual pixels in an image of a red car might be very close to black at night. The shape of the car's silhouette depends on the viewing angle. Most applications require us to *disentangle* the factors of variation and discard the ones that we do not care about.

Of course, it can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation, such as a speaker's accent, can only be identified using sophisticated, nearly human-level understanding of the data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

*Deep learning* solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning allows the computer to build complex concepts out of simpler concepts. Fig. 1.2 shows how a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.

The quintessential example of a deep learning model is the feedforward deep network or *multilayer perceptron* (MLP). A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions. We can think of each application of a different mathematical function as providing a new representation of the input.

The idea of learning the right representation for the data provides one perspective on deep learning. Another perspective on deep learning is that it allows the computer to learn a multi-step computer program. Each layer of the representation can be thought of as the state of the computer's memory after executing another set of instructions in parallel. Networks with greater depth can execute more instructions in sequence. Being able to execute instructions sequentially offers great power because later instructions can refer back to the results of earlier instructions. According to this view of deep learning, not all of the information in a layer's representation of the input necessarily encodes factors of variation that explain the input. The representation is also used to store state information

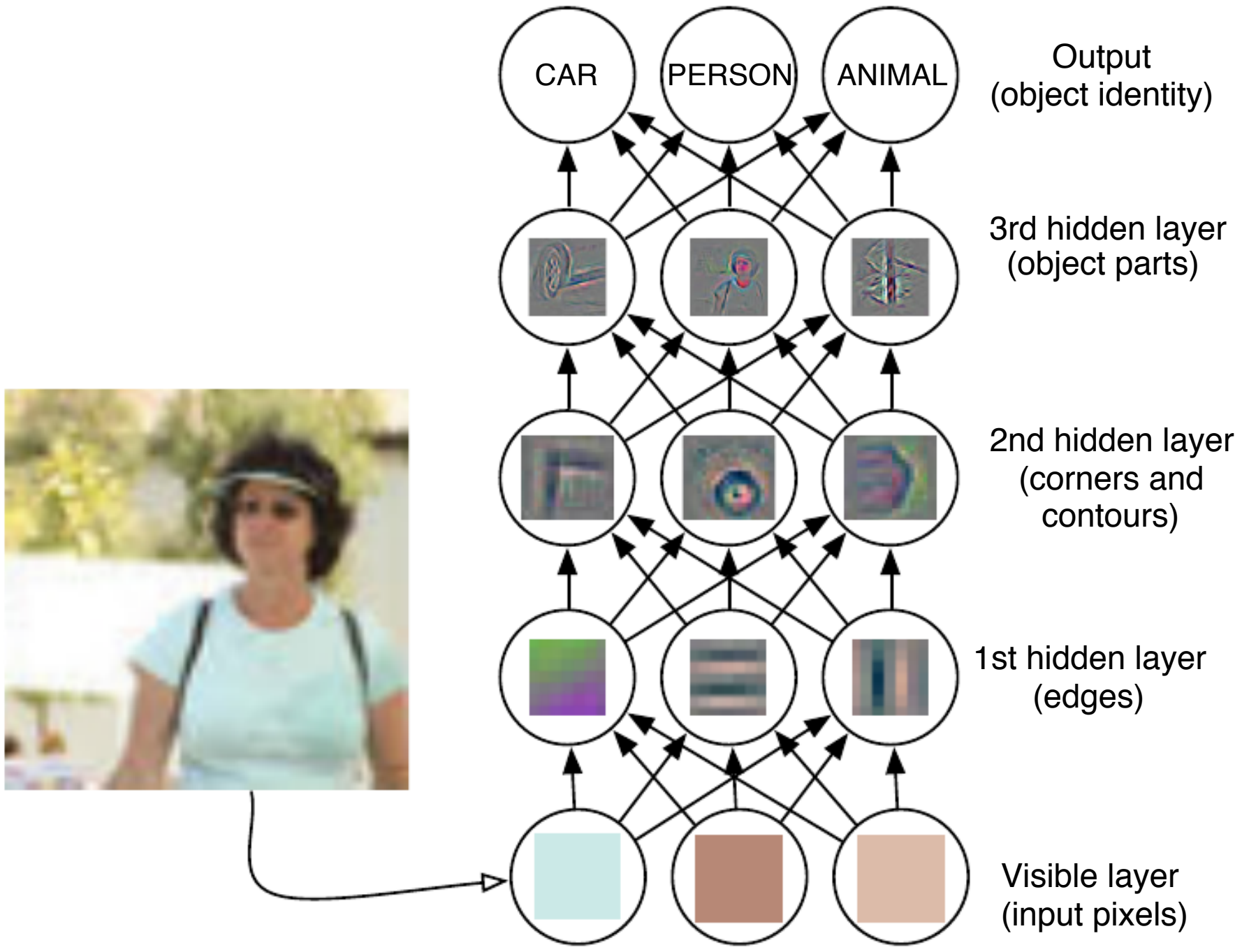


Figure 1.2: Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating this mapping seems insurmountable if tackled directly. Deep learning resolves this difficulty by breaking the desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. The input is presented at the *visible layer*, so named because it contains the variables that we are able to observe. Then a series of *hidden layers* extracts increasingly abstract features from the image. These layers are called "hidden" because their values are not given in the data; instead the model must determine which concepts are useful for explaining the relationships in the observed data. The images here are visualizations of the kind of feature represented by each hidden unit. Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels. Given the first hidden layer's description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges. Given the second hidden layer's description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners. Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image. Images reproduced with permission from Zeiler and Fergus (2014).

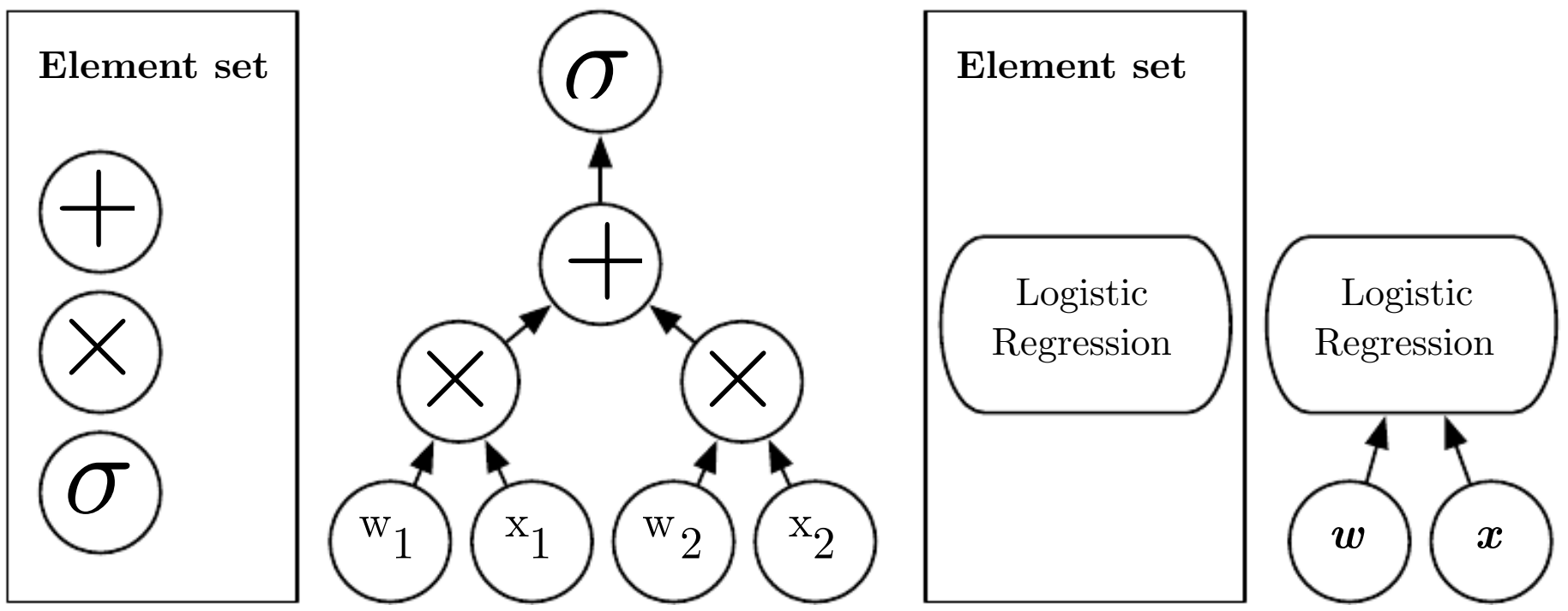


Figure 1.3: Illustration of computational flow graphs mapping an input to an output where each node performs an operation. Depth is the length of the longest path from input to output but depends on the definition of what constitutes a possible computational step. The computation depicted in these graphs is the output of a logistic regression model, $\sigma(\boldsymbol{w}^T\boldsymbol{x})$, where $\sigma$ is the logistic sigmoid function. If we use addition, multiplication and logistic sigmoids as the elements of our computer language, then this model has depth three. If we view logistic regression as an element itself, then this model has depth one.

that helps to execute a program that can make sense of the input. This state information could be analogous to a counter or pointer in a traditional computer program. It has nothing to do with the content of the input specifically, but it helps the model to organize its processing.

There are two main ways of measuring the depth of a model.

The first view is based on the number of sequential instructions that must be executed to evaluate the architecture. We can think of this as the length of the longest path through a flow chart that describes how to compute each of the model's outputs given its inputs. Just as two equivalent computer programs will have different lengths depending on which language the program is written in, the same function may be drawn as a flow chart with different depths depending on which functions we allow to be used as individual steps in the flow chart. Fig. 1.3 illustrates how this choice of language can give two different measurements for the same architecture.

Another approach, used by deep probabilistic models, illustrates not the depth of the computational graph but the depth of the graph describing how concepts are related to each other. In this case, the depth of the flow-chart of the computations needed to compute the representation of each concept may be much deeper than the graph of the concepts themselves. This is because the system's understanding of the simpler concepts can be refined given information about the more complex concepts. For example, an AI system observing an image of a face with one eye in

shadow may initially only see one eye. After detecting that a face is present, it can then infer that a second eye is probably present as well. In this case, the graph of concepts only includes two layers—a layer for eyes and a layer for faces—but the graph of computations includes $2n$ layers if we refine our estimate of each concept given the other $n$ times.

Because it is not always clear which of these two views—the depth of the computational graph, or the depth of the probabilistic modeling graph—is most relevant, and because different people choose different sets of smallest elements from which to construct their graphs, there is no single correct value for the depth of an architecture, just as there is no single correct value for length of a computer program. Nor is there a consensus about how much depth a model requires to qualify as "deep." However, deep learning can safely be regarded as the study of models that either involve a greater amount of composition of learned functions or learned concepts than traditional machine learning does.

To summarize, deep learning, the subject of this book, is an approach to AI. Specifically, it is a type of machine learning, a technique that allows computer systems to improve with experience and data. According to the authors of this book, machine learning is the only viable approach to building AI systems that can operate in complicated, real-world environments. Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts and representations, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. Fig. 1.4 illustrates the relationship between these different AI disciplines. Fig. 1.5 gives a high-level schematic of how each works.

## 1.1 Who Should Read This Book?

This book can be useful for a variety of readers, but we wrote it with two main target audiences in mind. One of these target audiences is university students (undergraduate or graduate) learning about machine learning, including those who are beginning a career in deep learning and artificial intelligence research. The other target audience is software engineers who do not have a machine learning or statistics background, but want to rapidly acquire one and begin using deep learning in their product or platform. Software engineers working in a wide variety of industries are likely to find deep learning to be useful, as it has already proven successful in many areas including computer vision, speech and audio processing, natural language processing, robotics, bioinformatics and chemistry, video games, search engines, online advertising and finance.

This book has been organized into three parts in order to best accommodate

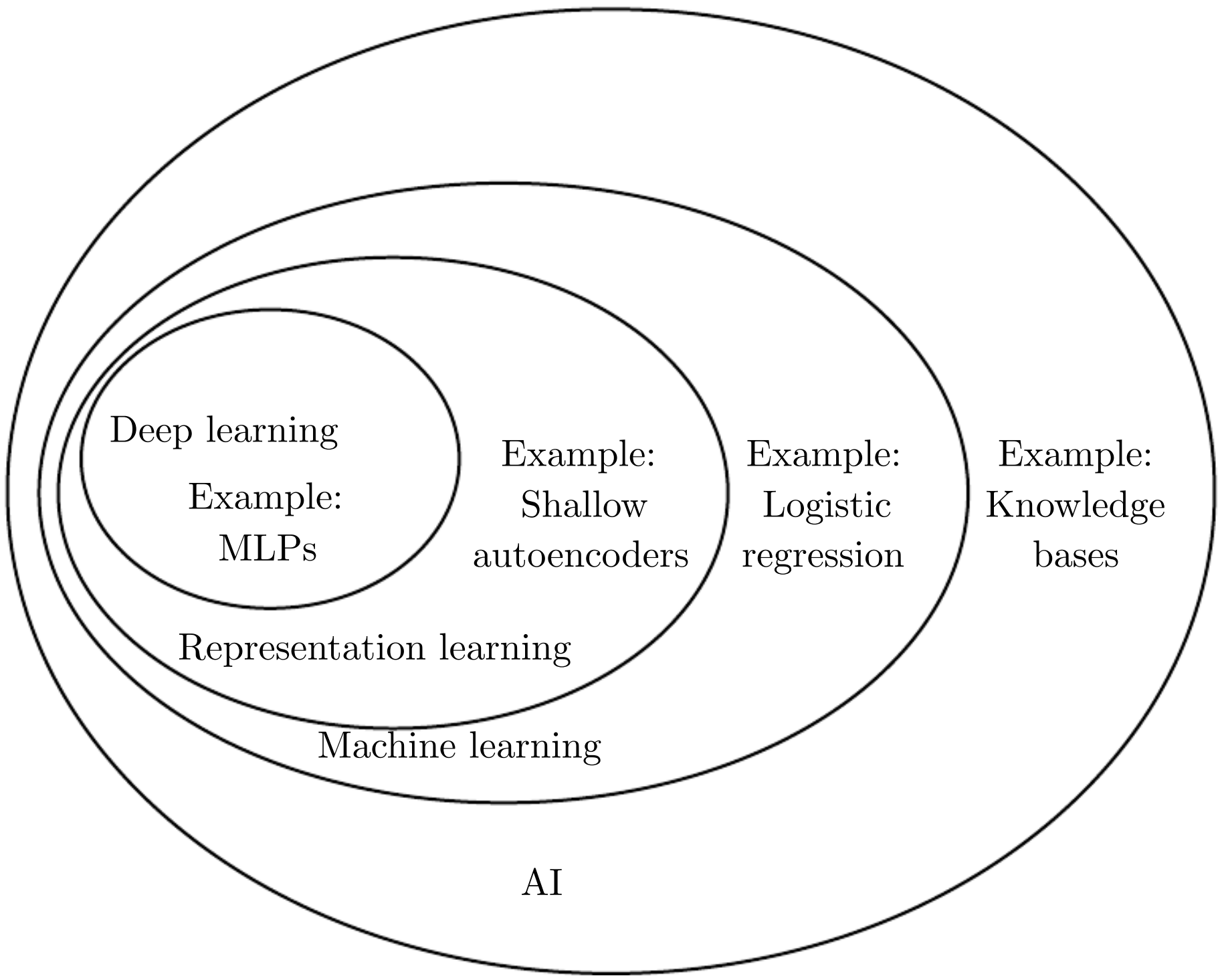


Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

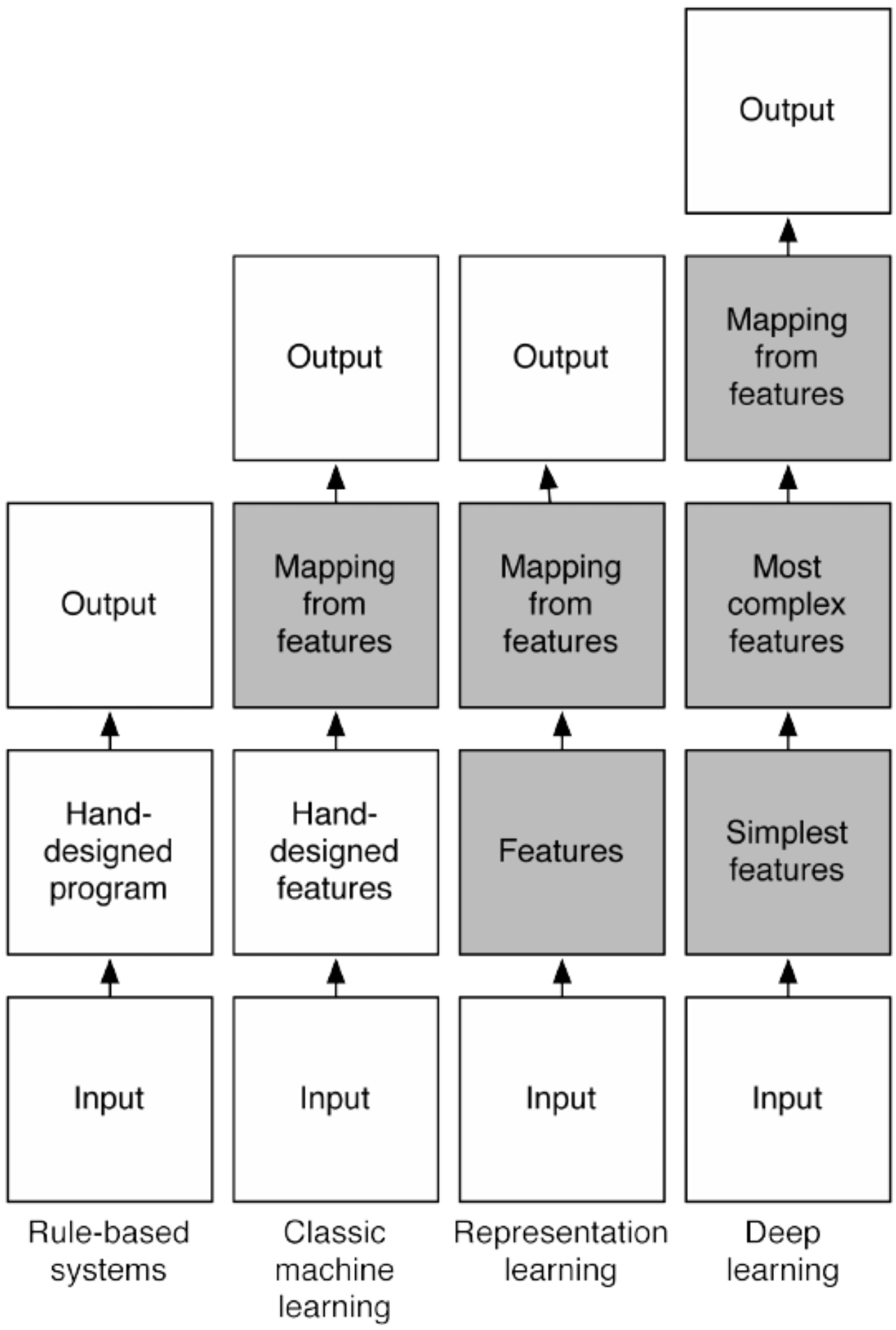


Figure 1.5: Flow-charts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.

a variety of readers. Part 1 introduces basic mathematical tools and machine learning concepts. Part 2 describes the most established deep learning algorithms that are essentially solved technologies. Part 3 describes more speculative ideas that are widely believed to be important for future research in deep learning.

Readers should feel free to skip parts that are not relevant given their interests or background. Readers familiar with linear algebra, probability, and fundamental machine learning concepts can skip part 1, for example, while readers who just want to implement a working system need not read beyond part 2.

We do assume that all readers come from a computer science background. We assume familiarity with programming, a basic understanding of computational performance issues, complexity theory, introductory level calculus and some of the terminology of graph theory.

## 1.2 Historical Trends in Deep Learning

It is easiest to understand deep learning with some historical context. Rather than providing a detailed history of deep learning, we identify a few key trends:

Deep learning has had a long and rich history, but has gone by many names reflecting different philosophical viewpoints, and has waxed and waned in popularity.

Deep learning has become more useful as the amount of available training data has increased.

Deep learning models have grown in size over time as computer hardware and software infrastructure for deep learning has improved.

Deep learning has solved increasingly complicated applications with increasing accuracy over time.

### 1.2.1 The Many Names and Changing Fortunes of Neural Networks

We expect that many readers of this book have heard of deep learning as an exciting new technology, and are surprised to see a mention of "history" in a book about an emerging field. In fact, deep learning has a long and rich history. Deep learning only *appears* to be new, because it was relatively unpopular for several years preceding its current popularity, and because it has gone through many different names. While the term "deep learning" is relatively new, the field dates back to the 1950s. The field has been rebranded many times, reflecting the influence of different researchers and different perspectives.

A comprehensive history of deep learning is beyond the scope of this pedagogical textbook. However, some basic context is useful for understanding deep learning. Broadly speaking, there have been three waves of development of deep learning: deep learning known as *cybernetics* in the 1940s-1960s, deep learning known as *connectionism* in the 1980s-1990s, and the current resurgence under the name deep learning beginning in 2006. See Figure 1.6 for a basic timeline.

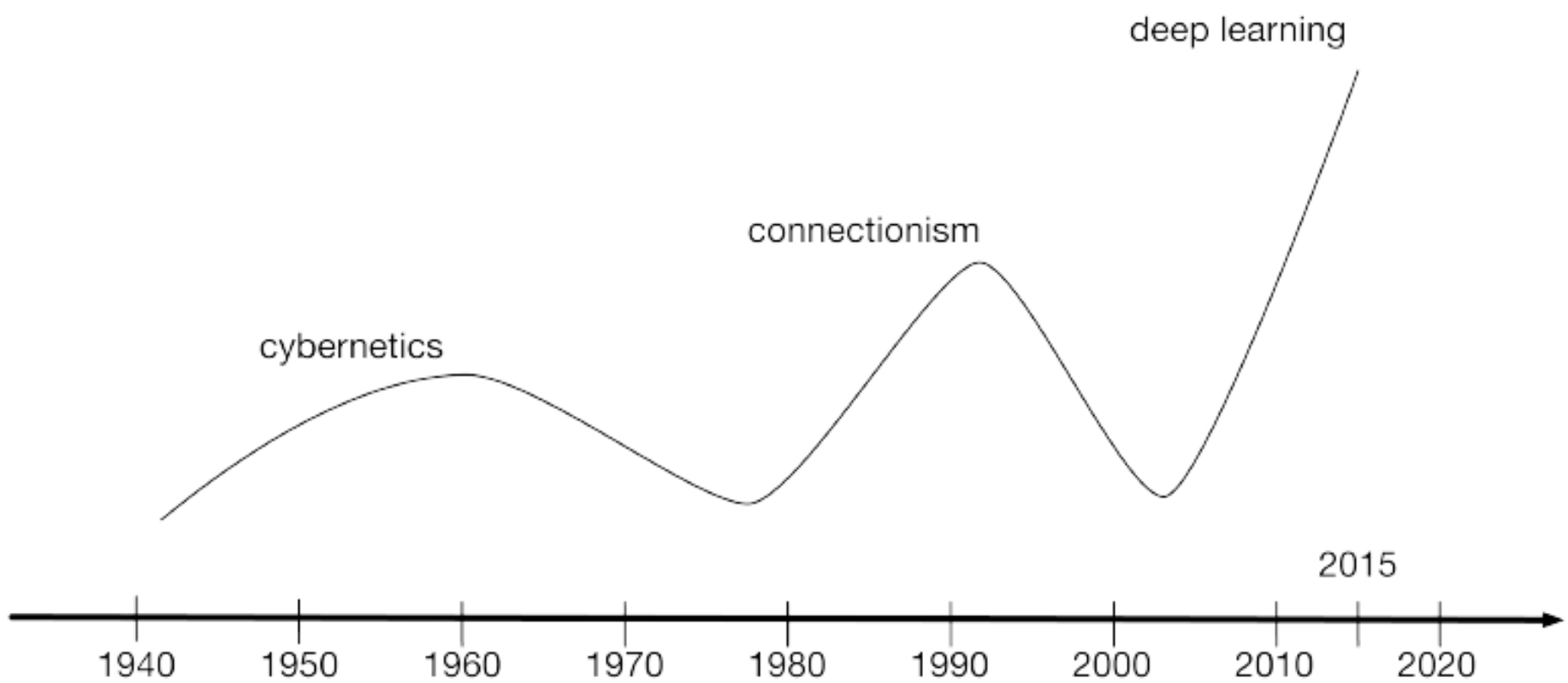


Figure 1.6: The three historical waves of artificial neural nets research, starting with cybernetics in the 1940-1960's, with the perceptron (Rosenblatt, 1958) to train a single neuron, then the connectionist approach of the 1980-1995 period, with back-propagation (Rumelhart *et al.*, 1986a) to train a neural network with one or two hidden layers, and the current wave, deep learning, started around 2006 (Hinton *et al.*, 2006; Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a), which allows us to train very deep networks.

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, i.e. models of how learning happens or could happen in the brain. As a result, one of the names that deep learning has gone by is *artificial neural networks* (ANNs). The corresponding perspective on deep learning models is that they are engineered systems inspired by the biological brain (whether the human brain or the brain of another animal). The neural perspective on deep learning is motivated by two main ideas. One idea is that the brain provides a proof by example that intelligent behavior is possible, and a conceptually straightforward path to building intelligence is to reverse engineer the computational principles behind the brain and duplicate its functionality. Another perspective is that it would be deeply interesting to understand the brain and the principles that underlie human intelligence, so machine learning models that shed light on these basic scientific questions are useful apart from their ability to solve engineering applications.

The modern term "deep learning" goes beyond the neuroscientific perspective on the current breed of machine learning models. It appeals to a more general principle of learning *multiple levels of composition*, which can be applied in machine learning frameworks that are not necessarily neurally inspired.

The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. These models were designed to take a set of $n$ input values $x_1, \ldots, x$ and associate them with an output $y$. These models would learn a set of weights $w_1, \ldots, w$ and compute their output $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + \quad + x \; w$ . This first wave of neural networks research was known as *cybernetics* (see Fig. 1.6).

The McCulloch-Pitts Neuron (McCulloch and Pitts, 1943) was an early model of brain function. This linear model could recognize two different categories of inputs by testing whether $f(\boldsymbol{x}, \boldsymbol{w})$ is positive or negative. Of course, for the model to correspond to the desired definition of the categories, the weights needed to be set correctly. These weights could be set by the human operator. In the 1950s, the perceptron (Rosenblatt, 1958, 1962) became the first model that could learn the weights defining the categories given examples of inputs from each category. The Adaptive Linear Element (ADALINE), which dates from about the same time, simply returned the value of $f(\boldsymbol{x})$ itself to predict a real number (Widrow and Hoff, 1960), and could also learn to predict these numbers from data.

These simple learning algorithms greatly affected the modern landscape of machine learning. The training algorithm used to adapt the weights of the ADALINE was a special case of an algorithm called *stochastic gradient descent*. Slightly modified versions of the stochastic gradient descent algorithm remain the dominant training algorithms for deep learning models today.

Models based on the $f(\boldsymbol{x}, \boldsymbol{w})$ used by the perceptron and ADALINE are called *linear models*. These models remain some of the most widely used machine learning models, though in many cases they are *trained* in different ways than the original models were trained.

Linear models have many limitations. Most famously, they cannot learn the XOR function, where $f([0,1], \boldsymbol{w}) = 1$ and $f([1,0], \boldsymbol{w}) = 1$ but $f([1,1], \boldsymbol{w}) = 0$ and $f([0,0], \boldsymbol{w}) = 0$. Critics who observed these flaws in linear models caused a backlash against biologically inspired learning in general (Minsky and Papert, 1969). This is the first dip in the popularity of neural networks in our broad timeline (Fig. 1.6).

Today, neuroscience is regarded as an important source of inspiration for deep learning researchers, but it is no longer the predominant guide for the field.

The main reason for the diminished role of neuroscience in deep learning research today is that we simply do not have enough information about the brain to use it as a guide. To obtain a deep understanding of the actual algorithms

used by the brain, we would need to be able to monitor the activity of (at the very least) thousands of interconnected neurons simultaneously. Because we are not able to do this, we are far from understanding even some of the most simple and well-studied parts of the brain (Olshausen and Field, 2005).

Neuroscience has given us a reason to hope that a single deep learning algorithm can solve many different tasks. Neuroscientists have found that ferrets can learn to "see" with the auditory processing region of their brain if their brains are rewired to send visual signals to that area (Von Melchner *et al.*, 2000). This suggests that much of the mammalian brain might use a single algorithm to solve most of the different tasks that the brain solves. Before this hypothesis, machine learning research was more fragmented, with different communities of researchers studying natural language processing, vision, motion planning and speech recognition. Today, these application communities are still separate, but it is common for deep learning research groups to study many or even all of these application areas simultaneously.

We are able to draw some rough guidelines from neuroscience. The basic idea of having many computational units that become intelligent only via their interactions with each other is inspired by the brain. The Neocognitron (Fukushima, 1980) introduced a powerful model architecture for processing images that was inspired by the structure of the mammalian visual system and later became the basis for the modern convolutional network (LeCun *et al.*, 1998b), as we will see in Chapter 9.10. Most neural networks today are based on a model neuron called the *rectified linear unit*. These units were developed from a variety of viewpoints, with (Nair and Hinton, 2010b) and Glorot *et al.* (2011a) citing neuroscience as an influence, and Jarrett *et al.* (2009a) citing more engineering-oriented influences. While neuroscience is an important source of inspiration, it need not be taken as a rigid guide. We know that actual neurons compute very different functions than modern rectified linear units, but greater neural realism has not yet found a machine learning value or interpretation. Also, while neuroscience has successfully inspired several neural network *architectures*, we do not yet know enough about biological learning for neuroscience to offer much guidance for the *learning algorithms* we use to train these architectures.

Media accounts often emphasize the similarity of deep learning to the brain. While it is true that deep learning researchers are more likely to cite the brain as an influence than researchers working in other machine learning fields such as kernel machines or Bayesian statistics, one should not view deep learning as an attempt to simulate the brain. Modern deep learning draws inspiration from many fields, especially applied math fundamentals like linear algebra, probability, information theory, and numerical optimization. While some deep learning researchers cite neuroscience as an important influence, others are not concerned

with neuroscience at all.

It is worth noting that the effort to understand how the brain works on an algorithmic level is alive and well. This endeavor is primarily known as "computational neuroscience" and is a separate field of study from deep learning. It is common for researchers to move back and forth between both fields. The field of deep learning is primarily concerned with how to build computer systems that are able to successfully solve tasks requiring intelligence, while the field of computational neuroscience is primarily concerned with building more accurate models of how the brain actually works.

In the 1980s, the second wave of neural network research emerged in great part via a movement called *connectionism* or *parallel distributed processing* (Rumelhart *et al.*, 1986d). Connectionism arose in the context of cognitive science. Cognitive science is an interdisciplinary approach to understanding the mind, combining multiple different levels of analysis. During the early 1980s, most cognitive scientists studied models of symbolic reasoning. Despite their popularity, symbolic models were difficult to explain in terms of how the brain could actually implement them using neurons. The connectionists began to study models of cognition that could actually be grounded in neural implementations, reviving many ideas dating back to the work of psychologist Donald Hebb in the 1940s (Hebb, 1949).

The central idea in connectionism is that a large number of simple computational units can achieve intelligent behavior when networked together. This insight applies equally to neurons in biological nervous systems and to hidden units in computational models.

Several key concepts arose during the connectionism movement of the 1980s that remain central to today's deep learning.

One of these concepts is that of *distributed representation*. This is the idea that each input to a system should be represented by many features, and each feature should be involved in the representation of many possible inputs. For example, suppose we have a vision system that can recognize cars, trucks, and birds and these objects can each be red, green, or blue. One way of representing these inputs would be to have a separate neuron or hidden unit that activates for each of the nine possible combinations: red truck, red car, red bird, green truck, and so on. This requires nine different neurons, and each neuron must independently learn the concept of color and object identity. One way to improve on this situation is to use a distributed representation, with three neurons describing the color and three neurons describing the object identity. This requires only six neurons total instead of nine, and the neuron describing redness is able to learn about redness from images of cars, trucks and birds, not only from images of one specific category of objects. The concept of distributed representation is central to this book, and will be described in greater detail in Chapter 16.

Another major accomplishment of the connectionist movement was the successful use of back-propagation to train deep neural networks with internal representations and the popularization of the back-propagation algorithm (Rumelhart *et al.*, 1986a; LeCun, 1987). This algorithm has waxed and waned in popularity but as of this writing is currently the dominant approach to training deep models.

The second wave of neural networks research lasted until the mid-1990s. At that point, the popularity of neural networks declined again. This was in part due to a negative reaction to the failure of neural networks (and AI research in general) to fulfill excessive promises made by a variety of people seeking investment in neural network-based ventures, but also due to improvements in other fields of machine learning: kernel machines (Boser *et al.*, 1992; Cortes and Vapnik, 1995; Schölkopf *et al.*, 1999) and graphical models (Jordan, 1998).

Kernel machines enjoy many nice theoretical guarantees. In particular, training a kernel machine is a *convex optimization problem* (this will be explained in more detail in Chapter 4) which means that the training process can be guaranteed to find the optimal model efficiently. This made kernel machines very amenable to software implementations that "just work" without much need for the human operator to understand the underlying ideas. Soon, most machine learning applications consisted of manually designing good features to provide to a kernel machine for each different application area.

During this time, neural networks continued to obtain impressive performance on some tasks (LeCun *et al.*, 1998c; Bengio *et al.*, 2001a). The Canadian Institute for Advanced Research (CIFAR) helped to keep neural networks research alive via its Neural Computation and Adaptive Perception research initiative. This program united machine research groups led by Geoffrey Hinton at University of Toronto, Yoshua Bengio at University of Montreal, and Yann LeCun at New York University. It had a multi-disciplinary nature that also included neuroscientists and experts in human and computer vision.

At this point in time, deep networks were generally believed to be very difficult to train. We now know that algorithms that have existed since the 1980s work quite well, but this was not apparent circa 2006. The issue is perhaps simply that these algorithms were too computationally costly to allow much experimentation with the hardware available at the time.

The third wave of neural networks research began with a breakthrough in 2006. Geoffrey Hinton showed that a kind of neural network called a deep belief network could be efficiently trained using a strategy called greedy layer-wise pretraining (Hinton *et al.*, 2006), which will be described in more detail in Chapter 16.1. The other CIFAR-affiliated research groups quickly showed that the same strategy could be used to train many other kinds of deep networks (Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a) and systematically helped to improve gen-

eralization on test examples. This wave of neural networks research popularized the use of the term *deep learning* to emphasize that researchers were now able to train deeper neural networks than had been possible before, and to emphasize the theoretical importance of depth (Bengio and LeCun, 2007a; Delalleau and Bengio, 2011; Pascanu *et al.*, 2014a; Montufar *et al.*, 2014). Deep neural networks displaced kernel machines with manually designed features for several important application areas during this time—in part because the time and memory cost of training a kernel machine is quadratic in the size of the dataset, and datasets grew to be large enough for this cost to outweigh the benefits of convex optimization. This third wave of popularity of neural networks continues to the time of this writing, though the focus of deep learning research has changed dramatically within the time of this wave. The third wave began with a focus on new unsupervised learning techniques and the ability of deep models to generalize well from small datasets, but today there is more interest in much older supervised learning algorithms and the ability of deep models to leverage large labeled datasets.

### 1.2.2 Increasing Dataset Sizes

One may wonder why deep learning has only recently become recognized as a crucial technology if it has existed since the 1950s. Deep learning has been successfully used in commercial applications since the 1990s, but was often regarded as being more of an art than a technology and something that only an expert could use, until recently. It is true that some skill is required to get good performance from a deep learning algorithm. Fortunately, the amount of skill required reduces as the amount of training data increases. The learning algorithms reaching human performance on complex tasks today are nearly identical to the learning algorithms that struggled to solve toy problems in the 1980s, though the models we train with these algorithms have undergone changes that simplify the training of very deep architectures. The most important new development is that today we can provide these algorithms with the resources they need to succeed. Fig. 1.7 shows how the size of benchmark datasets has increased remarkably over time. This trend is driven by the increasing digitization of society. As more and more of our activities take place on computers, more and more of what we do is recorded. As our computers are increasingly networked together, it becomes easier to centralize these records and curate them into a dataset appropriate for machine learning applications. The age of "Big Data" has made machine learning much easier because the key burden of statistical estimation—generalizing well to new data after observing only a small amount of data—has been considerably lightened. As of 2015, a rough rule of thumb is that a supervised deep learning algorithm will generally achieve acceptable performance with around 5,000 labeled examples per category, and will match or exceed human performance when

trained with a dataset containing at least 10 million labeled examples. Working successfully with datasets smaller than this is an important research area, focusing in particular on how we can take advantage of large quantities of unlabeled examples, with unsupervised or semi-supervised learning.

### 1.2.3 Increasing Model Sizes

Another key reason that neural networks are wildly successful today after enjoying comparatively little success since the 1980s is that we have the computational resources to run much larger models today. One of the main insights of connectionism is that animals become intelligent when many of their neurons work together. An individual neuron or small collection of neurons is not particularly useful.

Biological neurons are not especially densely connected. As seen in Fig. 1.9, our machine learning models have had a number of connections per neuron that was within an order of magnitude of even mammalian brains for decades.

In terms of the total number of neurons, neural networks have been astonishingly small until quite recently, as shown in Fig. 1.10. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. This growth is driven by faster computers with larger memory and by the availability of larger datasets. Larger networks are able to achieve higher accuracy on more complex tasks. This trend looks set to continue for decades. Unless new technologies allow faster scaling, artificial neural networks will not have the same number of neurons as the human brain until at least the 2050s. Biological neurons may represent more complicated functions than current artificial neurons, so biological neural networks may be even larger than this plot portrays.

In retrospect, it is not particularly surprising that neural networks with fewer neurons than a leech were unable to solve sophisticated artificial intelligence problems. Even today's networks, which we consider quite large from a computational systems point of view, are smaller than the nervous system of even relatively primitive vertebrate animals like frogs.

The increase in model size over time, due to the availability of faster CPUs, the advent of general purpose GPUs, faster network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

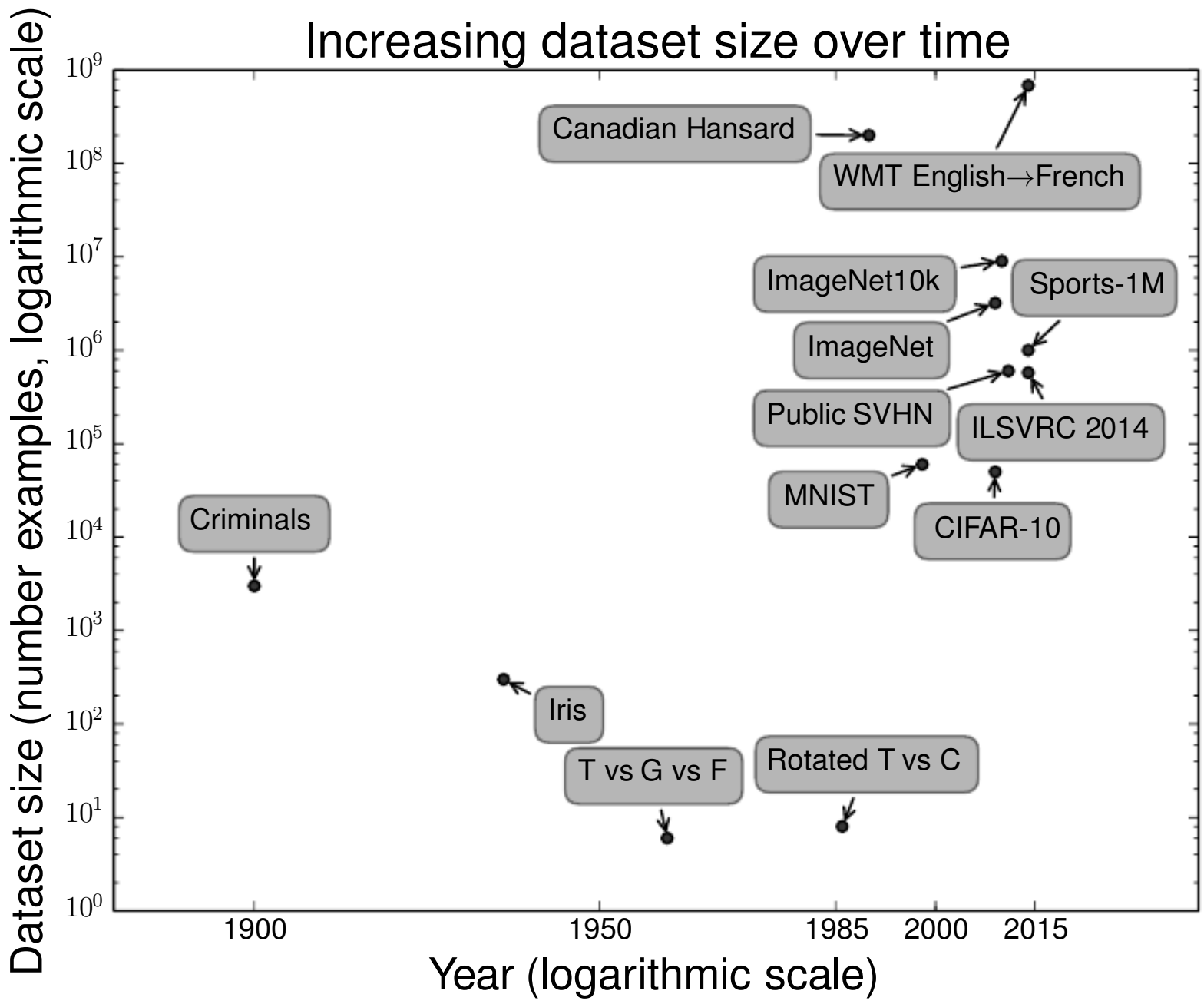


Figure 1.7: Dataset sizes have increased greatly over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through 1980s, the pioneers of biologically-inspired machine learning often worked with small, synthetic datasets, such as low-resolution bitmaps of letters, that were designed to incur low computational cost and demonstrate that neural networks were able to learn specific kinds of functions (Widrow and Hoff, 1960; Rumelhart *et al.*, 1986b). In the 1980s and 1990s, machine learning became more statistical in nature and began to leverage larger datasets containing tens of thousands of examples such as the MNIST dataset (show in Fig. 1.8) of scans of handwritten numbers (LeCun *et al.*, 1998c). In the first decade of the 2000s, more sophisticated datasets of this same size, such as the CIFAR-10 dataset (Krizhevsky and Hinton, 2009) continued to be produced. Toward the end of that decade and throughout the first half of the 2010s, significantly larger datasets, containing hundreds of thousands to tens of millions of examples, completely changed what was possible with deep learning. These datasets included the public Street View House Numbers dataset(Netzer *et al.*, 2011), various versions of the ImageNet dataset (Deng *et al.*, 2009, 2010a; Russakovsky *et al.*, 2014a), and the Sports-1M dataset (Karpathy *et al.*, 2014). At the top of the graph, we see that datasets of translated sentences, such as IBM's dataset constructed from the Canadian Hansard (Brown *et al.*, 1990) and the WMT 2014 dataset (Schwenk, 2014) are typically far ahead of other dataset sizes.

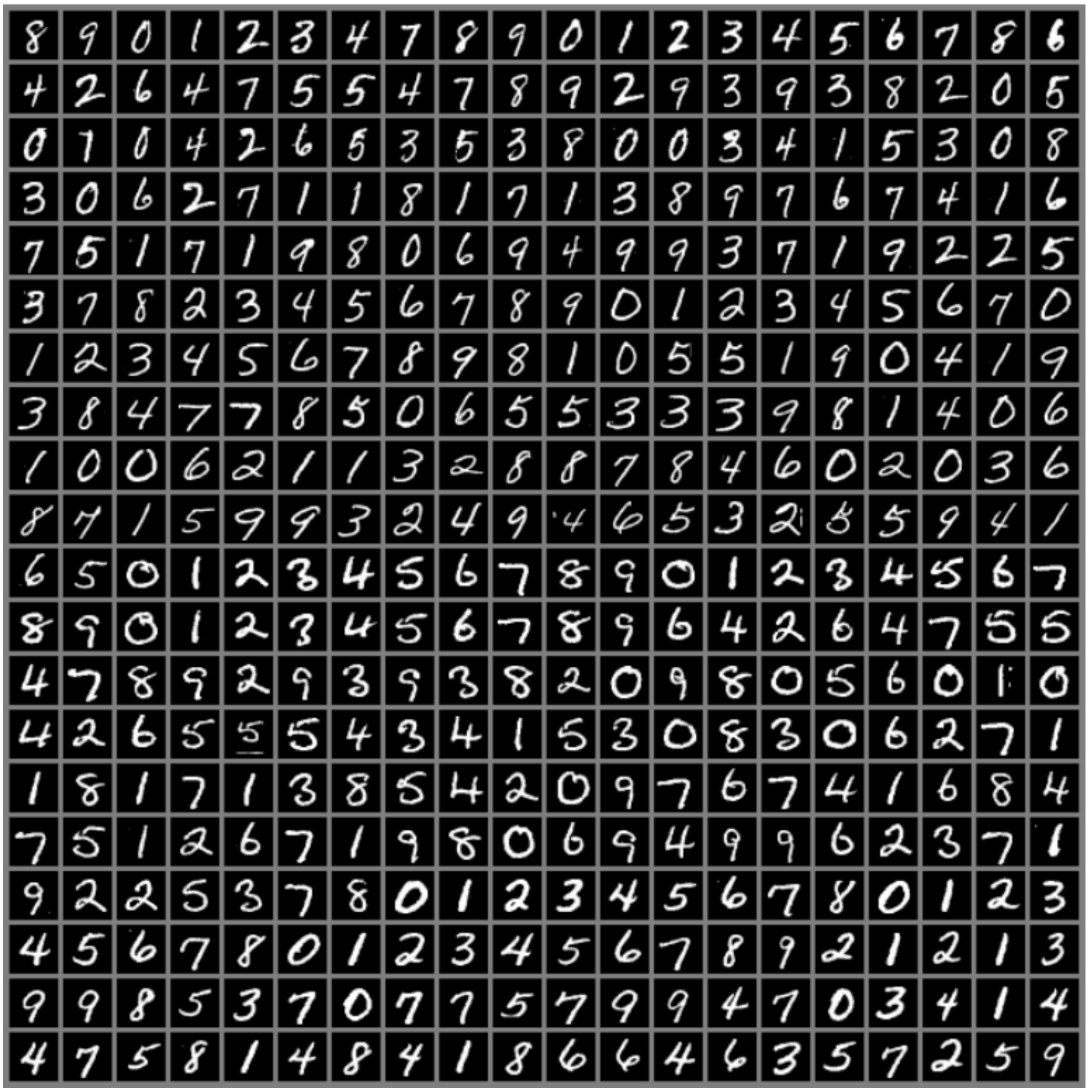

Figure 1.8: Example inputs from the MNIST dataset. The "NIST" stands for National Institute of Standards and Technology, the agency that originally collected this data. The "M" stands for "modified," since the data has been preprocessed for easier use with machine learning algorithms. The MNIST dataset consists of scans of handwritten digits and associated labels describing which digit 0-9 is contained in each image. This simple classification problem is one of the simplest and most widely used tests in deep learning research. It remains popular despite being quite easy for modern techniques to solve. Geoffrey Hinton has described it as "the *drosophila* of machine learning," meaning that it allows machine learning researchers to study their algorithms in controlled laboratory conditions, much as biologists often study fruit flies.

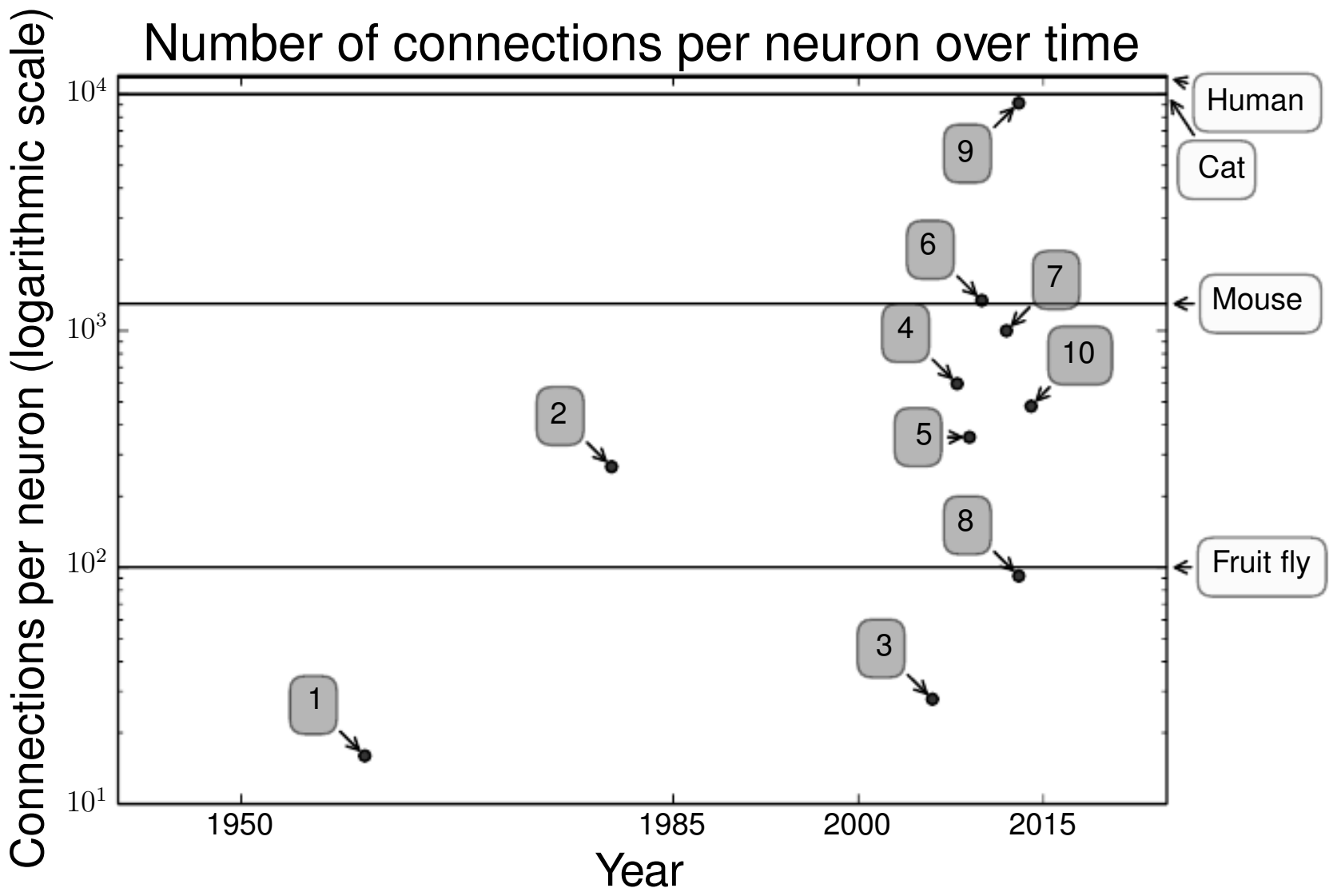


Figure 1.9: Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, and it is quite common for other neural networks to have as many connections per neuron as smaller mammals like mice. Even the human brain does not have an exorbitant amount of connections per neuron. The sparse connectivity of biological neural networks means that our artificial networks are able to match the performance of biological neural networks despite limited hardware. Modern neural networks are much smaller than the brains of any vertebrate animal, but we typically train each network to perform just one task, while an animal's brain has different areas devoted to different tasks. Biological neural network sizes from Wikipedia (2015).

1. Adaptive Linear Element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
4. Deep Boltzmann machines (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009b)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012a)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

### 1.2.4 Increasing Accuracy, Application Complexity and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition or prediction. Moreover, deep learning has consistently been applied with success to broader and broader sets of applications.

The earliest deep models were used to recognize individual objects in tightly cropped, extremely small images (Rumelhart *et al.*, 1986a). Since then there has been a gradual increase in the size of images neural networks could process. Modern object recognition networks process rich high-resolution photographs and do not have a requirement that the photo be cropped near the object to be recognized (Krizhevsky *et al.*, 2012b). Similarly, the earliest networks could only recognize two kinds of objects (or in some cases, the absence or presence of a single kind of object), while these modern networks typically recognize at least 1,000 different categories of objects. The largest contest in object recognition is the ImageNet Large-Scale Visual Recognition Competition held each year. A dramatic moment in the meteoric rise of deep learning came when a convolutional network won this challenge for the first time and by a wide margin, bringing down the state-of-the-art error rate from 26.1% to 15.3% (Krizhevsky *et al.*, 2012b). Since then, these competitions are consistently won by deep convolutional nets, and as of this writing, advances in deep learning had brought the latest error rate in this contest down to 6.5% as shown in Fig. 1.11, using even deeper networks (Szegedy *et al.*, 2014a). Outside the framework of the contest, this error rate has now dropped to below 5% (Ioffe and Szegedy, 2015; Wu *et al.*, 2015).

Deep learning has also had a dramatic impact on speech recognition. After improving throughout the 1990s, the error rates for speech recognition stagnated starting in about 2000. The introduction of deep learning (Dahl *et al.*, 2010; Deng *et al.*, 2010b; Seide *et al.*, 2011; Hinton *et al.*, 2012a) to speech recognition resulted in a sudden drop of error rates by up to half! We will explore this history in more detail in Sec. 12.3.

Deep networks have also had spectacular successes for pedestrian detection and image segmentation (Sermanet *et al.*, 2013; Farabet *et al.*, 2013a; Couprie *et al.*, 2013) and yielded superhuman performance in traffic sign classification (Ciresan *et al.*, 2012).

At the same time that the scale and accuracy of deep networks has increased, so has the complexity of the tasks that they can solve. Goodfellow *et al.* (2014d) showed that neural networks could learn to output an entire sequence of characters transcribed from an image, rather than just identifying a single object. Previously, it was widely believed that this kind of learning required labeling of the individual elements of the sequence (Gülçehre and Bengio, 2013). Since this time, a neural network designed to model sequences, the Long Short-Term Memory or LSTM

(Hochreiter and Schmidhuber, 1997), has enjoyed an explosion in popularity. LSTMs and related models are now used to model relationships between *sequences* and other *sequences* rather than just fixed inputs. This sequence-to-sequence learning seems to be on the cusp of revolutionizing another application: machine translation (Sutskever *et al.*, 2014a; Bahdanau *et al.*, 2014).

This trend of increasing complexity has been pushed to its logical conclusion with the introduction of the Neural Turing Machine (Graves *et al.*, 2014a), a neural network that can learn entire programs. This neural network has been shown to be able to learn how to sort lists of numbers given examples of scrambled and sorted sequences. This self-programming technology is in its infancy, but in the future could in principle be applied to nearly any task.

Many of these applications of deep learning are highly profitable, given enough data to apply deep learning to. Deep learning is now used by many top technology companies including Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA and NEC.

Deep learning has also made contributions back to other sciences. Modern convolutional networks for object recognition provide a model of visual processing that neuroscientists can study (DiCarlo, 2013). Deep learning also provides useful tools for processing massive amounts of data and making useful predictions in scientific fields. It has been successfully used to predict how molecules will interact in order to help pharmaceutical companies design new drugs (Dahl *et al.*, 2014), to search for subatomic particles (Baldi *et al.*, 2014), and to automatically parse microscope images used to construct a 3-D map of the human brain (Knowles-Barley *et al.*, 2014). We expect deep learning to appear in more and more scientific fields in the future.

In summary, deep learning is an approach to machine learning that has drawn heavily on our knowledge of the human brain, statistics and applied math as it developed over the past several decades. In recent years, it has seen tremendous growth in its popularity and usefulness, due in large part to more powerful computers, larger datasets and techniques to train deeper networks. The years ahead are full of challenges and opportunities to improve deep learning even further and bring it to new frontiers.

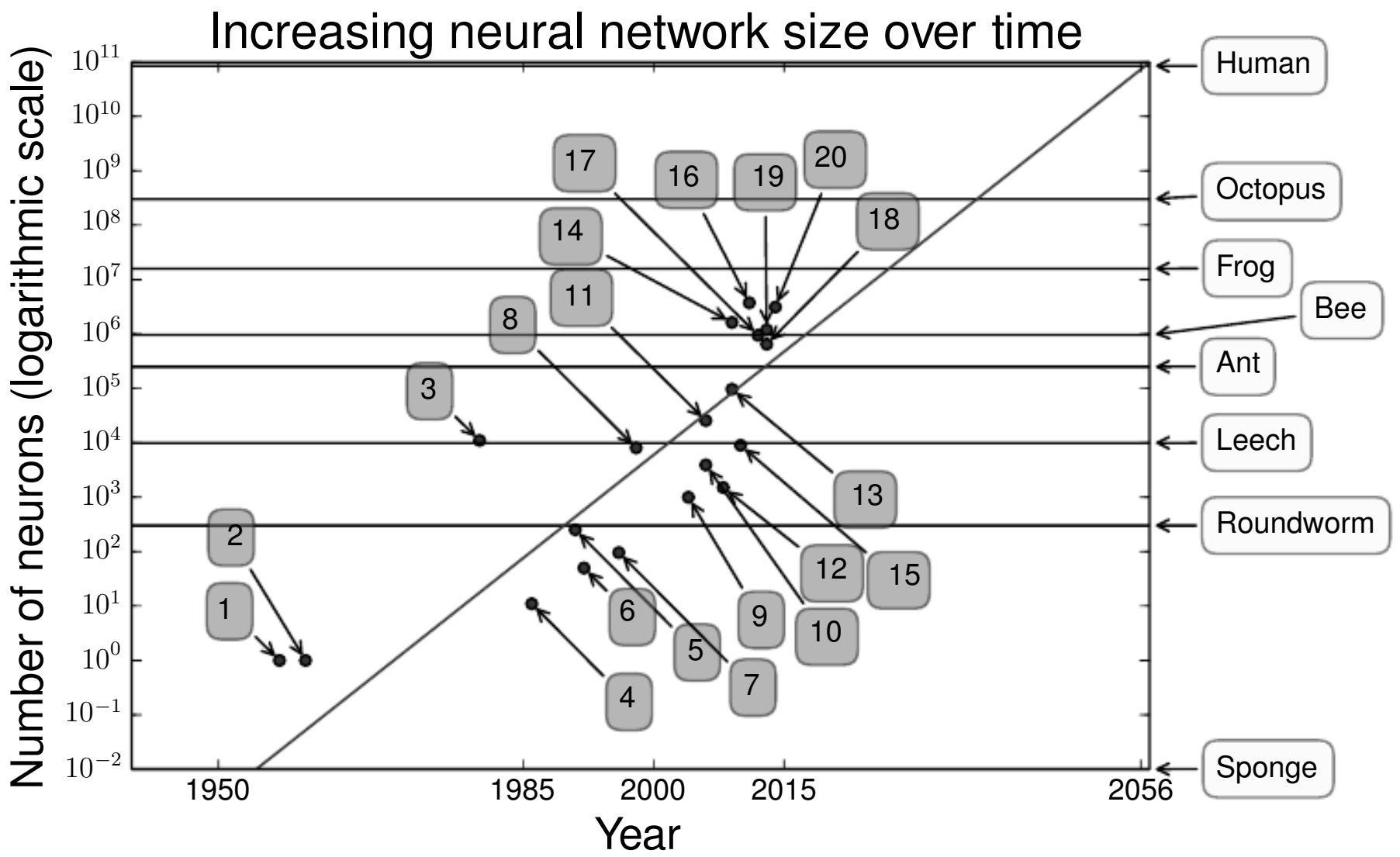


Figure 1.10: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive Linear Element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early backpropagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machines (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009b)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012a)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

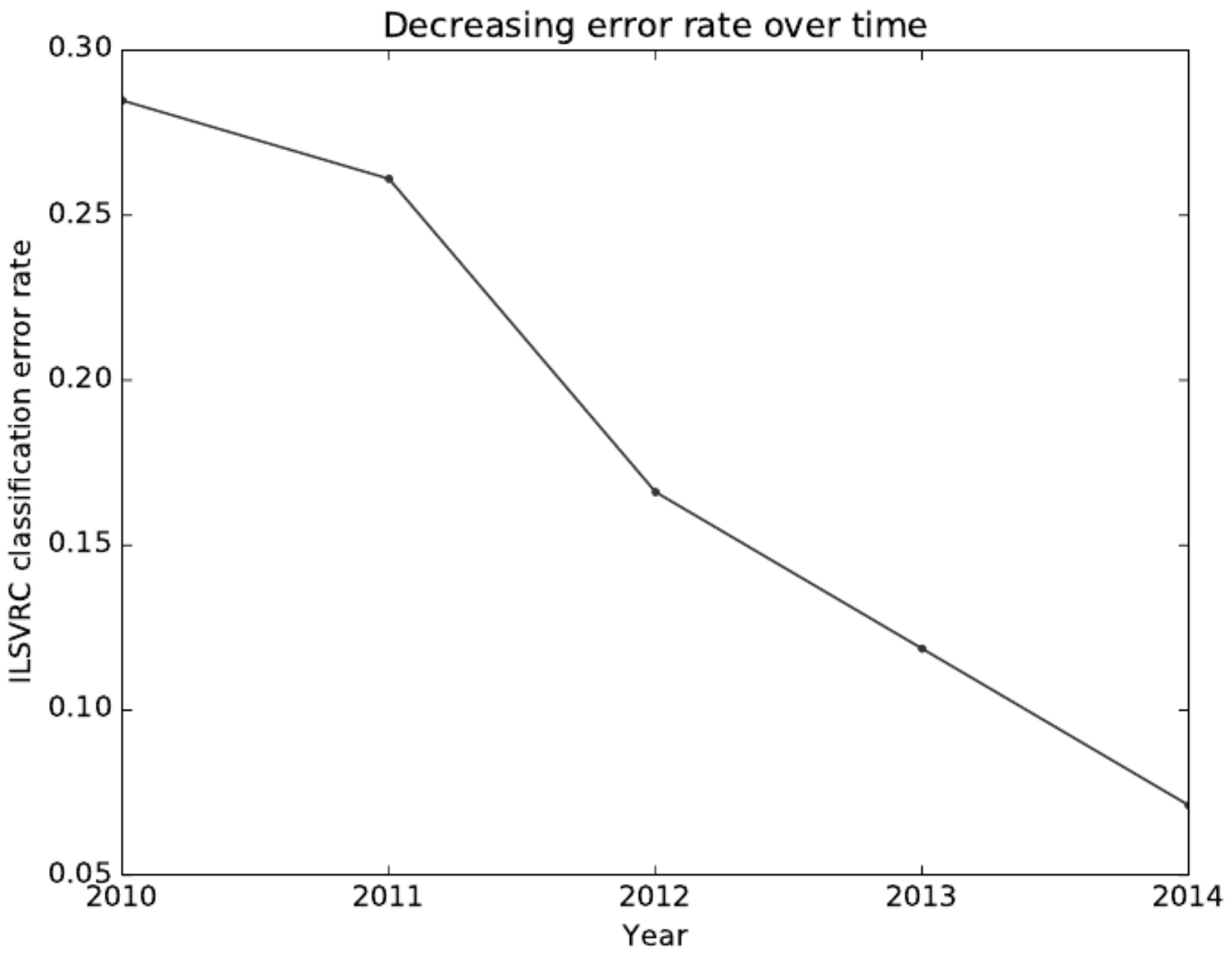


Figure 1.11: Since deep networks reached the scale necessary to compete in the ImageNet Large Scale Visual Recognition, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky *et al.* (2014b).

# Part I

# Applied Math and Machine Learning Basics

This part of the book introduces the basic mathematical concepts needed to understand deep learning. We begin with general ideas from applied math, that allow us to define functions of many variables, find the highest and lowest points on these functions and quantify degrees of belief.

Next, we describe the fundamental goals of machine learning. We describe how to accomplish these goals by specifying a model that represents certain beliefs, designing a cost function that measures how well those beliefs correspond with reality and using a training algorithm to minimize that cost function.

This elementary framework is the basis for a broad variety of machine learning algorithms, including approaches to machine learning that are not deep. In the subsequent parts of the book, we develop deep learning algorithms within this framework.

# Chapter 2

# Linear Algebra

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. However, because linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms. We therefore begin the technical content of the book with a focused presentation of the key linear algebra ideas that are most important in deep learning.

If you are already familiar with linear algebra, feel free to skip this chapter. If you have previous experience with these concepts but need a detailed reference sheet to review key formulas, we recommend *The Matrix Cookbook* (Petersen and Pedersen, 2006). If you have no exposure at all to linear algebra, this chapter will teach you enough to read this book, but we highly recommend that you also consult another resource focused exclusively on teaching linear algebra, such as (Shilov, 1977). This chapter will completely omit many important linear algebra topics that are not essential for understanding deep learning.

## 2.1 Scalars, Vectors, Matrices and Tensors

The study of linear algebra involves several types of mathematical objects:

- *Scalars*: A scalar is just a single number, in contrast to most of the other objects studied in linear algebra, which are usually arrays of multiple numbers. We write scalars in italics. We usually give scalars lower-case variable names. When we introduce them, we specify what kind of number they are. For example, we might say "Let $s \in \mathbb{R}$ be the slope of the line," while defining a real-valued scalar, or "Let $n \in \mathbb{N}$ be the number of units," while defining a natural number scalar.

- *Vectors*: A vector is an array of numbers. The numbers have an order to them, and we can identify each individual number by its index in that ordering. Typically we give vectors lower case names written in bold typeface, such as $\boldsymbol{x}$. The elements of the vector are identified by writing its name in italic typeface, with a subscript. The first element of $\boldsymbol{x}$ is $x_1$, the second element is $x_2$ and so on. We also need to say what kind of numbers are stored in the vector. If each element is in $\mathbb{R}$, and the vector has $n$ elements, then the vector lies in the set formed by taking the Cartesian product of $\mathbb{R}$ $n$ times, denoted as $\mathbb{R}^n$. When we need to explicitly identify the elements of a vector, we write them as a column enclosed in square brackets:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

  We can think of vectors as identifying points in space, with each element giving the coordinate along a different axis.

  Sometimes we need to index a set of elements of a vector. In this case, we define a set containing the indices and write the set as a subscript. For example, to access $x_1$, $x_3$ and $x_6$, we define the set $S = \{1, 3, 6\}$ and write $\boldsymbol{x}_S$. We use the $-$ sign to index the complement of a set. For example $\boldsymbol{x}_{-1}$ is the vector containing all elements of $\boldsymbol{x}$ except for $x_1$, and $\boldsymbol{x}_{-S}$ is the vector containing all of the elements of $\boldsymbol{x}$ except for $x_1$, $x_3$ and $x_6$.

- *Matrices*: A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one. We usually give matrices upper-case variable names with bold typeface, such as $\boldsymbol{A}$. If a real-valued matrix $\boldsymbol{A}$ has a height of $m$ and a width of $n$, then we say that $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. We usually identify the elements of a matrix using its name in italic but not bold font, and the indices are listed with separating commas. For example, $A_{1,1}$ is the upper left entry of $\boldsymbol{A}$ and $A_{m,n}$ is the bottom right entry. We can identify all of the numbers with vertical coordinate $i$ by writing a ":" for the horizontal coordinate. For example, $\boldsymbol{A}_{i,:}$ denotes the horizontal cross section of $\boldsymbol{A}$ with vertical coordinate $i$. This is known as the $i$-th *row* of $\boldsymbol{A}$. Likewise, $\boldsymbol{A}_{:,i}$ is the $i$-th *column* of $\boldsymbol{A}$. When we need to explicitly identify the elements of a matrix, we write them as an array enclosed in square brackets:

$$\begin{matrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{matrix}.$$

$$\begin{bmatrix} 29 & \\ & \end{bmatrix}$$

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \Rightarrow \boldsymbol{A}^\top = \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \end{bmatrix}$$

Figure 2.1: The transpose of the matrix can be thought of as a mirror image across the main diagonal.

Sometimes we may need to index matrix-valued expressions that are not just a single letter. In this case, we use subscripts after the expression, but do not convert anything to lower case. For example, $f(\boldsymbol{A})_{i,j}$ gives element $(i, j)$ of the matrix computed by applying the function $f$ to $\boldsymbol{A}$.

- *Tensors*: In some cases we will need an array with more than two axes. In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a *tensor*. We denote a tensor named "A" with this typeface: $\mathbf{A}$. We identify the element of $\mathbf{A}$ at coordinates $(i, j, k)$ by writing $A_{i,j,k}$.

One important operation on matrices is the *transpose*. The transpose of a matrix is the mirror image of the matrix across a diagonal line, called the *main diagonal*, running down and to the right, starting from its upper left corner. See Fig. 2.1 for a graphical depiction of this operation. We denote the transpose of a matrix $\boldsymbol{A}$ as $\boldsymbol{A}^\top$, and it is defined such that

$$(\boldsymbol{A}^\top)_{i,j} = A_{j,i}\,.$$

Vectors can be thought of as matrices that contain only one column. The transpose of a vector is therefore a matrix with only one row. Sometimes we define a vector by writing out its elements in the text inline as a row matrix, then using the transpose operator to turn it into a standard column vector, e.g. $\boldsymbol{x} = [x_1, x_2, x_3]^\top$.

We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ where $C_{i,j} = A_{i,j} + B_{i,j}$.

We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix: $\boldsymbol{D} = a \cdot \boldsymbol{B} + c$ where $D_{i,j} = a \cdot B_{i,j} + c$.

## 2.2 Multiplying Matrices and Vectors

One of the most important operations involving matrices is multiplication of two matrices. The *matrix product* of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is a third matrix $\boldsymbol{C}$. In order

for this product to be defined, $\boldsymbol{A}$ must have the same number of columns as $\boldsymbol{B}$ has rows. If $\boldsymbol{A}$ is of shape $m \times n$ and $\boldsymbol{B}$ is of shape $n \times p$, then $\boldsymbol{C}$ is of shape $m \times p$. We can write the matrix product just by placing two or more matrices together, e.g.

$$\boldsymbol{C} = \boldsymbol{AB}.$$

The product operation is defined by

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}.$$

Note that the standard product of two matrices is *not* just a matrix containing the product of the individual elements. Such an operation exists and is called the *element-wise product* or *Hadamard product*, and is denoted in this book[1] as $\boldsymbol{A} \odot \boldsymbol{B}$.

The *dot product* between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same dimensionality is the matrix product $\boldsymbol{x}^\top \boldsymbol{y}$. We can think of the matrix product $\boldsymbol{C} = \boldsymbol{AB}$ as computing $c_{i,j}$ as the dot product between row $i$ of $\boldsymbol{A}$ and column $j$ of $\boldsymbol{B}$.

Matrix product operations have many useful properties that make mathematical analysis of matrices more convenient. For example, matrix multiplication is distributive:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AC}.$$

It is also associative:

$$\boldsymbol{A}(\boldsymbol{BC}) = (\boldsymbol{AB})\boldsymbol{C}.$$

Matrix multiplication is *not* commutative, unlike scalar multiplication.

The transpose of a matrix product also has a simple form:

$$(\boldsymbol{AB})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top.$$

Since the focus of this textbook is not linear algebra, we do not attempt to develop a comprehensive list of useful properties of the matrix product here, but the reader should be aware that many more exist.

We now know enough linear algebra notation to write down a system of linear equations:

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a known matrix, $\boldsymbol{b} \in \mathbb{R}^m$ is a known vector, and $\boldsymbol{x} \in \mathbb{R}^n$ is a vector of unknown variables we would like to solve for. Each element $x_i$ of $\boldsymbol{x}$ is one of these unknowns to solve for. Each row of $\boldsymbol{A}$ and each element of $\boldsymbol{b}$ provide another constraint. We can rewrite equation 2.1 as:

$$\boldsymbol{A}_{1,:}\boldsymbol{x} = b_1$$

[1] The element-wise product is used relatively rarely, so the notation for it is not as standardized as the other operations described in this chapter.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2.2: *Example identity matrix*: This is $\boldsymbol{I}_3$.

$$\boldsymbol{A}_{2,:}\boldsymbol{x} = b_2$$

$$\dots$$

$$\boldsymbol{A}_{m,:}\boldsymbol{x} = b_m$$

or, even more explicitly, as:

$$a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2$$

$$\dots$$

$$a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m.$$

Matrix-vector product notations provides a more compact representation for equations of this form.

## 2.3 Identity and Inverse Matrices

Linear algebra offers a powerful tool called *matrix inversion* that allows us to solve equation 2.1 for many values of $\boldsymbol{A}$.

To describe matrix inversion, we first need to define the concept of an *identity matrix*. An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix. We denote the $n$-dimensional identity matrix as $\boldsymbol{I}_n$. Formally,

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x}.$$

The structure of the identity matrix is simple: all of the entries along the main diagonal are 1, while all of the other entries are zero. See Fig. 2.2 for an example.

The *matrix inverse* of $\boldsymbol{A}$ is denoted as $\boldsymbol{A}^{-1}$, and it is defined as the matrix such that

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n.$$

We can now solve equation 2.1 by the following steps:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$
$$\boldsymbol{I}_n\,\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$
$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}.$$

Of course, this depends on it being possible to find $\boldsymbol{A}^{-1}$. We discuss the conditions for the existence of $\boldsymbol{A}^{-1}$ in the following section.

When $\boldsymbol{A}^{-1}$ exists, several different algorithms exist for finding it in closed form. In theory, the same inverse matrix can then be used to solve the equation many times for different values of $\boldsymbol{b}$. However, $\boldsymbol{A}^{-1}$ is primarily useful as a theoretical tool, and should not actually be used in practice for most software applications. Because $\boldsymbol{A}^{-1}$ can only be represented with limited precision on a digital computer, algorithms that make use of the value of $\boldsymbol{b}$ can usually obtain more accurate estimates of $\boldsymbol{x}$.

## 2.4 Linear Dependence and Span

In order for $\boldsymbol{A}^{-1}$ to exist, equation 2.1 must have exactly one solution for every value of $\boldsymbol{b}$. However, it is also possible for the system of equations to have no solutions or infinitely many solutions for some values of $\boldsymbol{b}$. It is not possible to have more than one but less than infinitely many solutions for a particular $\boldsymbol{b}$; if both $\boldsymbol{x}$ and $\boldsymbol{y}$ are solutions then

$$\boldsymbol{z} = \alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y}$$

is also a solution for any real $\alpha$.

To analyze how many solutions the equation has, we can think of the columns of $\boldsymbol{A}$ as specifying different directions we can travel from the *origin* (the point specified by the vector of all zeros), and determine how many ways there are of reaching $\boldsymbol{b}$. In this view, each element of $\boldsymbol{x}$ specifies how far we should travel in each of these directions, i.e. $x_i$ specifies how far to move in the direction of column $i$:

$$\boldsymbol{A}\boldsymbol{x} = \sum_i x_i\,\boldsymbol{A}_{:,i}\,.$$

In general, this kind of operation is called a *linear combination*. Formally, a linear combination of some set of vectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ is given by multiplying each vector $\boldsymbol{v}^{(i)}$ by a corresponding scalar coefficient and adding the results:

$$\sum_i c_i\,\boldsymbol{v}^{(i)}\,.$$

The *span* of a set of vectors is the set of all points obtainable by linear combination of the original vectors.

Determining whether $\boldsymbol{Ax} = \boldsymbol{b}$ has a solution thus amounts to testing whether $\boldsymbol{b}$ is in the span of the columns of $\boldsymbol{A}$. This particular span is known as the *column space* or the *range* of $\boldsymbol{A}$.

In order for the system $\boldsymbol{Ax} = \boldsymbol{b}$ to have a solution for all values of $\boldsymbol{b} \in \mathbb{R}^m$, we therefore require that the column space of $\boldsymbol{A}$ be all of $\mathbb{R}^m$. If any point in $\mathbb{R}^m$ is excluded from the column space, that point is a potential value of $\boldsymbol{b}$ that has no solution. This implies immediately that $\boldsymbol{A}$ must have at least $m$ columns, i.e., $n \geq m$. Otherwise, the dimensionality of the column space must be less than $m$. For example, consider a $3 \times 2$ matrix. The target $\boldsymbol{b}$ is 3-D, but $\boldsymbol{x}$ is only 2-D, so modifying the value of $\boldsymbol{x}$ at best allows us to trace out a 2-D plane within $\mathbb{R}^3$. The equation has a solution if and only if $\boldsymbol{b}$ lies on that plane.

Having $n \geq m$ is only a necessary condition for every point to have a solution. It is not a sufficient condition, because it is possible for some of the columns to be redundant. Consider a $2 \times 2$ matrix where both of the columns are equal to each other. This has the same column space as a $2 \times 1$ matrix containing only one copy of the replicated column. In other words, the column space is still just a line, and fails to encompass all of $\mathbb{R}^2$, even though there are two columns.

Formally, this kind of redundancy is known as *linear dependence*. A set of vectors is *linearly independent* if no vector in the set is a linear combination of the other vectors. If we add a vector to a set that is a linear combination of the other vectors in the set, the new vector does not add any points to the set's span. This means that for the column space of the matrix to encompass all of $\mathbb{R}^m$, the matrix must contain at least one set of $m$ linearly independent columns. This condition is both necessary and sufficient for equation 2.1 to have a solution for every value of $\boldsymbol{b}$. Note that the requirement is for a set to have exactly $m$ linear independent columns, not at least $m$. No set of $m$-dimensional vectors can have more than $m$ mutually linearly independent columns, but a matrix with more than $m$ columns may have more than one such set.

In order for the matrix to have an inverse, we additionally need to ensure that equation 2.1 has *at most* one solution for each value of $\boldsymbol{b}$. To do so, we need to ensure that the matrix has at most $m$ columns. Otherwise there is more than one way of parametrizing each solution.

Together, this means that the matrix must be *square*, that is, we require that $m = n$ and that all of the columns must be linearly independent. A square matrix with linearly dependent columns is known as *singular*.

If $\boldsymbol{A}$ is not square or is square but singular, it can still be possible to solve the equation. However, we can not use the method of matrix inversion to find the solution.

So far we have discussed matrix inverses as being multiplied on the left. It is

also possible to define an inverse that is multiplied on the right:

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}.$$

For square matrices, the left inverse and right inverse are equal.

## 2.5 Norms

Sometimes we need to measure the size of a vector. In machine learning, we usually measure the size of vectors using an $L^p$*norm*:

$$||\boldsymbol{x}||_p = \left( \sum_i |x_i|^p \right)^{\frac{1}{p}}$$

for $p \in \mathbb{R}, p \geq 1$.

Norms, including the $L^p$ norm, are functions mapping vectors to non-negative values, satisfying these properties that make them behave like distances between points:

- $f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ (the *triangle inequality*)
- $\forall \alpha \in \mathbb{R}, f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

The $L^2$ norm, with $p = 2$, is known as the *Euclidean norm*. It is simply the Euclidean distance from the origin to the point identified by $\boldsymbol{x}$. This is probably the most common norm used in machine learning. It is also common to measure the size of a vector using the squared $L^2$ norm, which can be calculated simply as $\boldsymbol{x}^\top \boldsymbol{x}$.

The squared $L^2$ norm is more convenient to work with mathematically and computationally than the $L^2$ norm itself. For example, the derivatives of the squared $L^2$ norm with respect to each element of $\boldsymbol{x}$ each depend only on the corresponding element of $\boldsymbol{x}$, while all of the derivatives of the $L^2$ norm depend on the entire vector. In many contexts, the squared $L^2$ norm may be undesirable because it increases very slowly near the origin. In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases, we turn to a function that grows at the same rate in all locations, but retains mathematical simplicity: the $L^1$ norm. The $L^1$ norm may be simplified to

$$||\boldsymbol{x}||_1 = \sum_i |x_i|.$$

The $L^1$ norm is commonly used in machine learning when the difference between zero and nonzero elements is very important. Every time an element of $\boldsymbol{x}$ moves away from 0 by $\epsilon$, the $L^1$ norm increases by $\epsilon$.

We sometimes measure the size of the vector by counting its number of nonzero elements (and when we use the $L^1$ norm, we often use it as a proxy for this function). Some authors refer to this function as the "$l_0$ norm," but this is incorrect terminology, because scaling the vector by $\alpha$ does not change the number of nonzero entries.

One other norm that commonly arises in machine learning is the $l_\infty$ norm, also known as *the max norm.* This norm simplifies to

$$||\boldsymbol{x}||_\infty = \max_i |x_i|,$$

e.g., the absolute value of the element with the largest magnitude in the vector.

Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the otherwise obscure *Frobenius norm*

$$||A||_F = \sqrt{\sum_{i,j} a_{i,j}^2},$$

which is analogous to the $L^2$ norm of a vector.

The dot product of two vectors can be rewritten in terms of norms. Specifically,

$$\boldsymbol{x}^\top \boldsymbol{y} = ||\boldsymbol{x}||_2 ||\boldsymbol{y}||_2 \cos\theta$$

where $\theta$ is the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$.

## 2.6 Special Kinds of Matrices and Vectors

Some special kinds of matrices and vectors are particularly useful.

*Diagonal* matrices only have non-zero entries along the main diagonal. Formally, a matrix $\boldsymbol{D}$ is diagonal if and only if $d_{i,j} = 0$ for all $i \neq j$. We've already seen one example of a diagonal matrix: the identity matrix, where all of the diagonal entries are 1. In this book[2], we write $\mathrm{diag}(\boldsymbol{v})$ to denote a square diagonal matrix whose diagonal entries are given by the entries of the vector $\boldsymbol{v}$. Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient. To compute $\mathrm{diag}(\boldsymbol{v})\boldsymbol{x}$, we only need to scale each element $x_i$ by $v_i$. In other words, $\mathrm{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \odot \boldsymbol{x}$. Inverting a square diagonal matrix is also efficient. The inverse exists only if every diagonal entry is nonzero,

[2] There is not a standardized notation for constructing a diagonal matrix from a vector.

and in that case, $\mathrm{diag}(\boldsymbol{v})^{-1} = \mathrm{diag}([1/v_1, \ldots, 1/v_n]^\top)$. In many cases, we may derive some very general machine learning algorithm in terms of arbitrary matrices, but obtain a less expensive (and less descriptive) algorithm by restricting some matrices to be diagonal.

Note that not all diagonal matrices need be square. It is possible to construct a rectangular diagonal matrix. Non-square diagonal matrices do not have inverses but it is still possible to multiply by them cheaply. For a non-square diagonal matrix $\boldsymbol{D}$, the product $\boldsymbol{Dx}$ will involving scaling each element of $\boldsymbol{x}$, and either concatenating some zeros to the result if $\boldsymbol{D}$ is taller than it is wide, or discarding some of the last elements of the vector if $\boldsymbol{D}$ is wider than it is tall.

A *symmetric* matrix is any matrix that is equal to its own transpose:

$$\boldsymbol{A} = \boldsymbol{A}^\top.$$

Symmetric matrices often arise when the entries are generated by some function of two arguments that does not depend on the order of the arguments. For example, if $\boldsymbol{A}$ is a matrix of distance measurements, with $a_{i,j}$ giving the distance from point $i$ to point $j$, then $a_{i,j} = a_{j,i}$ because distance functions are symmetric.

A *unit vector* is a vector with *unit norm*:

$$||\boldsymbol{x}||_2 = 1.$$

A vector $\boldsymbol{x}$ and a vector $\boldsymbol{y}$ are *orthogonal* to each other if $\boldsymbol{x}^\top\boldsymbol{y} = 0$. If both vectors have nonzero norm, this means that they are at 90 degree angles to each other. In $\mathbb{R}^n$, at most $n$ vectors may be mutually orthogonal with nonzero norm. If the vectors are not only orthogonal but also have unit norm, we call them *orthonormal*.

An *orthogonal matrix* is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

$$\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{I}.$$

This implies that

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\top,$$

so orthogonal matrices are of interest because their inverse is very cheap to compute. Pay careful attention to the definition of orthogonal matrices. Counterintuitively, their rows are not merely orthogonal but fully orthonormal. There is no special term for a matrix whose rows or columns are orthogonal but not orthonormal.

## 2.7 Eigendecomposition

Many mathematical objects can be understood better by breaking them into constituent parts, or finding some properties of them that are universal, not caused by the way we choose to represent them.

For example, integers can be decomposed into prime factors. The way we represent the number 12 will change depending on whether we write it in base ten or in binary, but it will always be true that $12 = 2 \times 2 \times 3$. From this representation we can conclude useful properties, such as that 12 is not divisible by 5, or that any integer multiple of 12 will be divisible by 3.

Much as we can discover something about the true nature of an integer by decomposing it into prime factors, we can also decompose matrices in ways that show us information about their functional properties that is not obvious from the representation of the matrix as an array of elements.

One of the most widely used kinds of matrix decomposition is called *eigendecomposition*, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

An *eigenvector* of a square matrix $\boldsymbol{A}$ is a non-zero vector $v$ such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}.$$

The scalar $\lambda$ is known as the *eigenvalue* corresponding to this eigenvector. (One can also find a *left eigenvector* such that $\boldsymbol{v}^\top \boldsymbol{A} = \lambda \boldsymbol{v}$, but we are usually concerned with right eigenvectors).

Note that if $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, then so is any rescaled vector $s\boldsymbol{v}$ for $s \in \mathbb{R}, s \neq 0$. Moreover, $s\boldsymbol{v}$ still has the same eigenvalue. For this reason, we usually only look for unit eigenvectors.

We can represent the matrix $\boldsymbol{A}$ using an *eigendecomposition*, with eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ and corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$ by concatenating the eigenvectors into a matrix $\boldsymbol{V} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}]$, (i.e. one column per eigenvector) and concatenating the eigenvalues into a vector $\boldsymbol{\lambda}$. Then the matrix

$$\boldsymbol{A} = \boldsymbol{V}\operatorname{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$$

has the desired eigenvalues and eigenvectors. If we make $\boldsymbol{V}$ an orthogonal matrix, then we can think of $\boldsymbol{A}$ as scaling space by $\lambda_i$ in direction $\boldsymbol{v}^{(i)}$. See Fig. 2.3 for an example.

We have seen that *constructing* matrices with specific eigenvalues and eigenvectors allows us to stretch space in desired directions. However, we often want to *decompose* matrices into their eigenvalues and eigenvectors. Doing so can help us to analyze certain properties of the matrix, much as decomposing an integer into its prime factors can help us understand the behavior of that integer.

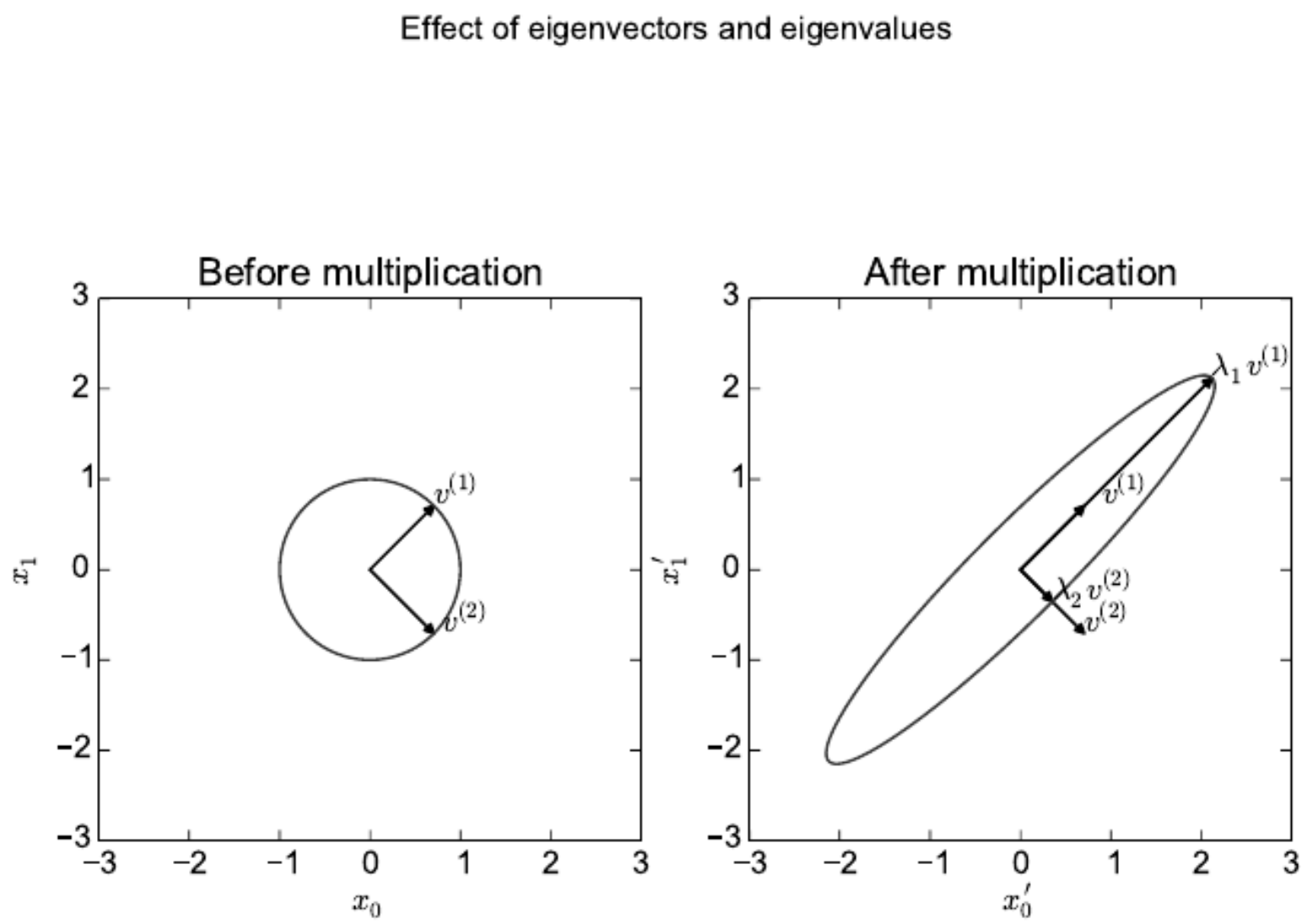


Figure 2.3: An example of the effect of eigenvectors and eigenvalues. Here, we have a matrix $\boldsymbol{A}$ with two orthonormal eigenvectors, $\boldsymbol{v}^{(1)}$ with eigenvalue $\lambda_1$ and $\boldsymbol{v}^{(2)}$ with eigenvalue $\lambda_2$. *Left)* We plot the set of all unit vectors $\boldsymbol{u} \in \mathbb{R}^2$ as a unit circle. *Right)* We plot the set of all points $\boldsymbol{Au}$. By observing the way that $\boldsymbol{A}$ distorts the unit circle, we can see that it scales space in direction $\boldsymbol{v}^{(i)}$ by $\lambda_i$.

Not every matrix can be decomposed into eigenvalues and eigenvectors. In some cases, the decomposition exists, but may involve complex rather than real numbers. Fortunately, in this book, we usually need to decompose only a specific class of matrices that have a simple decomposition. Specifically, every real symmetric matrix can be decomposed into an expression using only real-valued eigenvectors and eigenvalues:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top,$$

where $\boldsymbol{Q}$ is an orthogonal matrix composed of eigenvectors of $\boldsymbol{A}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix. The eigenvalue $\Lambda_{i,i}$ is associated with the eigenvector in column $i$ of $\boldsymbol{Q}$, denoted as $\boldsymbol{Q}_{:,i}$.

While any real symmetric matrix $\boldsymbol{A}$ is guaranteed to have an eigendecomposition, the eigendecomposition is not unique. If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue, and we could equivalently choose a $\boldsymbol{Q}$ using those eigenvectors instead. By convention, we usually sort the entries of $\boldsymbol{\Lambda}$ in descending order. Under this convention, the eigendecomposition is unique only if all of the eigenvalues are unique.

The eigendecomposition of a matrix tells us many useful facts about the matrix. The matrix is singular if and only if any of the eigenvalues are 0. The eigendecomposition can also be used to optimize quadratic expressions of the form $f(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x}$ subject to $||\boldsymbol{x}||_2 = 1$. Whenever $\boldsymbol{x}$ is equal to an eigenvector of $\boldsymbol{A}$, $f$ takes on the value of the corresponding eigenvalue. The maximum value of $f$ within the constraint region is the maximum eigenvalue and its minimum value within the constraint region is the minimum eigenvalue.

A matrix whose eigenvalues are all positive is called *positive definite*. A matrix whose eigenvalues are all positive or zero-valued is called *positive semidefinite*. Likewise, if all eigenvalues are negative, the matrix is *negative definite*, and if all eigenvalues are negative or zero-valued, it is *negative semidefinite*. Positive semidefinite matrices are interesting because they guarantee that $\forall \boldsymbol{x},\ \boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x} \geq 0$. Positive definite matrices additionally guarantee that $\boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$.

## 2.8 Singular Value Decomposition

In Sec. 2.7, we saw how to decompose a matrix into eigenvectors and eigenvalues. The *singular value decomposition* (SVD) provides another way to factorize a matrix, into *singular vectors* and *singular values*. The SVD allows us to discover some of the same kind of information as the eigendecomposition. However, the SVD is more generally applicable. Every real matrix has a singular value decomposition, but the same is not true of the eigenvalue decomposition. For example,

if a matrix is not square, the eigendecomposition is not defined, and we must use a singular value decomposition instead.

Recall that the eigendecomposition involves analyzing a matrix $\boldsymbol{A}$ to discover a matrix $\boldsymbol{V}$ of eigenvectors and a vector of eigenvalues $\boldsymbol{\lambda}$ such that we can rewrite $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{V}\operatorname{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}.$$

The singular value decomposition is similar, except this time we will write $\boldsymbol{A}$ as a product of three matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^{\top}.$$

Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix. Then $\boldsymbol{U}$ is defined to be an $m \times m$ matrix, $\boldsymbol{D}$ to be an $m \times n$ matrix, and $\boldsymbol{V}$ to be an $n \times n$ matrix.

Each of these matrices is defined to have a special structure. The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are both defined to be orthogonal matrices. The matrix $\boldsymbol{D}$ is defined to be a diagonal matrix. Note that $\boldsymbol{D}$ is not necessarily square.

The elements along the diagonal of $\boldsymbol{D}$ are known as the *singular values* of the matrix $\boldsymbol{A}$. The columns of $\boldsymbol{U}$ are known as the *left-singular vectors*. The columns of $\boldsymbol{V}$ are known as as the *right-singular vectors*.

We can actually interpret the singular value decomposition of $\boldsymbol{A}$ in terms of the eigendecomposition of functions of $\boldsymbol{A}$. The left-singular vectors of $\boldsymbol{A}$ are the eigenvectors of $\boldsymbol{A}\boldsymbol{A}^{\top}$. The right-singular vectors of $\boldsymbol{A}$ are the eigenvectors of $\boldsymbol{A}^{\top}\boldsymbol{A}$. The non-zero singular values of $\boldsymbol{A}$ are the square roots of the eigenvalues of $\boldsymbol{A}^{\top}\boldsymbol{A}$. The same is true for $\boldsymbol{A}\boldsymbol{A}^{\top}$.

Perhaps the most useful feature of the SVD is that we can use it to partially generalize matrix inversion to non-square matrices, as we will see in the next section.

## 2.9 The Moore-Penrose Pseudoinverse

Matrix inversion is not defined for matrices that are not square. Suppose we want to make a left-inverse $\boldsymbol{B}$ of a matrix $\boldsymbol{A}$, so that we can solve a linear equation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$$

by left-multiplying each side to obtain

$$\boldsymbol{x} = \boldsymbol{B}\boldsymbol{y}.$$

Depending on the structure of the problem, it may not be possible to design a unique mapping from $\boldsymbol{A}$ to $\boldsymbol{B}$.

If $\boldsymbol{A}$ is taller than it is wide, then it is possible for this equation to have no solution. If $\boldsymbol{A}$ is wider than it is tall, then there could be multiple possible solutions.

The *Moore-Penrose Pseudoinverse* allows us to make some headway in these cases. The pseudoinverse of $\boldsymbol{A}$ is defined as a matrix

$$\boldsymbol{A}^{+} = \lim_{\alpha \searrow 0} (\boldsymbol{A}^{\top}\boldsymbol{A} + \alpha \boldsymbol{I})^{-1} \boldsymbol{A}^{\top}.$$

Practical algorithms for computing the pseudoinverse are not based on this definition, but rather the formula

$$\boldsymbol{A}^{+} = \boldsymbol{V}\boldsymbol{D}^{+}\boldsymbol{U}^{\top},$$

where $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ are the singular value decomposition of $\boldsymbol{A}$, and the pseudoinverse $\boldsymbol{D}^{+}$ of a diagonal matrix $\boldsymbol{D}$ is obtained by taking the reciprocal of its non-zero elements then taking the transpose of the resulting matrix.

When $\boldsymbol{A}$ has more columns than rows, then solving a linear equation using the pseudoinverse provides one of the many possible solutions. Specifically, it provides the solution $\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{y}$ with minimal Euclidean norm $||\boldsymbol{x}||_2$ among all possible solutions.

When $\boldsymbol{A}$ has more rows than columns, it is possible for there to be no solution. In this case, using the pseudoinverse gives us the $\boldsymbol{x}$ for which $\boldsymbol{A}\boldsymbol{x}$ is as close as possible to $\boldsymbol{y}$ in terms of Euclidean norm $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}||_2$.

## 2.10 The Trace Operator

The trace operator gives the sum of all of the diagonal entries of a matrix:

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_{i} a_{i,i}.$$

The trace operator is useful for a variety of reasons. Some operations that are difficult to specify without resorting to summation notation can be specified using matrix products and the trace operator. For example, the trace operator provides an alternative way of writing the Frobenius norm of a matrix:

$$||\boldsymbol{A}||_F = \sqrt{\mathrm{Tr}(\boldsymbol{A}^{\top}\boldsymbol{A})}.$$

The trace operator also has many useful properties that make it easy to manipulate expressions involving the trace operator. For example, the trace operator is invariant to the transpose operator:

$$\mathrm{Tr}(\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}^{\top}).$$

The trace of a square matrix composed of many factors is also invariant to moving the last factor into the first position:

$$\mathrm{Tr}(\boldsymbol{ABC}) = \mathrm{Tr}(\boldsymbol{CAB}) = \mathrm{Tr}(\boldsymbol{BCA})$$

or more generally,

$$\mathrm{Tr}(\prod_{i=1}^{n} \boldsymbol{F}^{(i)}) = \mathrm{Tr}(\boldsymbol{F}^{(n)} \prod_{i=1}^{n-1} \boldsymbol{F}^{(i)}).$$

Another useful fact to keep in mind is that a scalar is its own trace, i.e. $a = Tr(a)$. This can be useful when wishing to manipulate inner products. Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two column vectors in $\mathbb{R}^n$

$$\boldsymbol{a}^\top \boldsymbol{b} = \mathrm{Tr}(\boldsymbol{a}^\top \boldsymbol{b}) = \mathrm{Tr}(\boldsymbol{b}\boldsymbol{a}^\top).$$

## 2.11 Determinant

The determinant of a square matrix, denoted $\det(\boldsymbol{A})$, is a function mapping matrices to real scalars. The determinant is equal to the product of all the matrix's eigenvalues. The absolute value of the determinant can be thought of as a measure of how much multiplication by the matrix expands or contracts space. If the determinant is 0, then space is contracted completely along at least one dimension, causing it to lose all of its volume. If the determinant is 1, then the transformation is volume-preserving.

## 2.12 Example: Principal Components Analysis

One simple machine learning algorithm, *principal components analysis (PCA)* can be derived using only knowledge of basic linear algebra.

Suppose we have a collection of $m$ points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ in $\mathbb{R}^n$. Suppose we would like to apply lossy compression to these points, i.e. we would like to find a way of storing the points that requires less memory but may lose some precision. We would like to lose as little precision as possible.

One way we can encode these points is to represent a lower-dimensional version of them. For each point $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ we will find a corresponding code vector $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$. If $l$ is smaller than $n$, it will take less memory to store the code points than the original data. We will want to find some encoding function that produces the code for an input, $f(\boldsymbol{x}) = \boldsymbol{c}$ and a decoding function that produces the reconstructed input given its code, i.e., $\boldsymbol{x} \approx g(f(\boldsymbol{x}))$.

PCA is defined by our choice of the decoding function. Specifically, to make the decoder very simple, we choose to use matrix multiplication to map the code

back into $\mathbb{R}^n$. Let $g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, where $\boldsymbol{D} \in \mathbb{R}^{n \times l}$ is the matrix defining the decoding.

Computing the optimal code for this decoder could be a difficult problem. To keep the encoding problem easy, PCA constrains the columns of $\boldsymbol{D}$ to be orthogonal to each other. (Note that $\boldsymbol{D}$ is still not technically "an orthogonal matrix" unless $l = n$)

With the problem as described so far, many solutions are possible, because we can increase the scale of $\boldsymbol{D}_{:,i}$ if we decrease $c_i$ proportionally for all points. To give the problem a unique solution, we constrain all of the columns of $\boldsymbol{D}$ to have unit norm.

In order to turn this basic idea into an algorithm we can implement, the first thing we need to do is figure out how to generate the optimal code point $\boldsymbol{c}^*$ for each input point $\boldsymbol{x}$. One way to do this is to minimize the distance between the input point $\boldsymbol{x}$ and its reconstruction, $g(\boldsymbol{c}^*)$. We can measure this distance using a norm. In the principal components algorithm, we use the $L^2$ norm:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - g(\boldsymbol{c})||_2.$$

We can switch to the squared $L^2$ norm instead of the $L^2$ norm itself, because both are minimized by the same value of $\boldsymbol{c}$. This is because the $L^2$ norm is non-negative and the squaring operation is monotonically increasing for non-negative arguments.

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - g(\boldsymbol{c})||_2^2$$

The function being minimized simplifies to

$$(\boldsymbol{x} - g(\boldsymbol{c}))^\top (\boldsymbol{x} - g(\boldsymbol{c}))$$

(by the definition of the $L^2$ norm)

$$= \boldsymbol{x}^\top \boldsymbol{x} - \boldsymbol{x}^\top g(\boldsymbol{c}) - g(\boldsymbol{c})^\top \boldsymbol{x} + g(\boldsymbol{c})^\top g(\boldsymbol{c})$$

(by the distributive property)

$$= \boldsymbol{x}^\top \boldsymbol{x} - 2\boldsymbol{x}^\top g(\boldsymbol{c}) + g(\boldsymbol{c})^\top g(\boldsymbol{c})$$

(because a scalar is equal to the transpose of itself).

We can now change the function being minimized again, to omit the first term, since this term does not depend on $\boldsymbol{c}$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top g(\boldsymbol{c}) + g(\boldsymbol{c})^\top g(\boldsymbol{c}).$$

To make further progress, we must substitute in the definition of $g(\boldsymbol{c})$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{D}^\top \boldsymbol{D}\boldsymbol{c}$$

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{I}_l \boldsymbol{c}$$

(by the orthogonality and unit norm constraints on $\boldsymbol{D}$)

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}$$

We can solve this optimization problem using vector calculus (see section 4.3 if you do not know how to do this):

$$\nabla_{\boldsymbol{c}}(-2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}) = \boldsymbol{0}$$

$$-2\boldsymbol{D}^\top \boldsymbol{x} + 2\boldsymbol{c} = \boldsymbol{0}$$

$$\boldsymbol{c} = \boldsymbol{D}^\top \boldsymbol{x}.$$

This is good news: we can optimally encode $\boldsymbol{x}$ just using a matrix-vector operation. To encode a vector, we apply the encoder function

$$f(\boldsymbol{x}) = \boldsymbol{D}^\top \boldsymbol{x}.$$

Using a further matrix multiplication, we can also define the PCA reconstruction operation:

$$r(\boldsymbol{x}) = g\left(f\left(\boldsymbol{x}\right)\right) = \boldsymbol{D}\boldsymbol{D}^\top \boldsymbol{x}. \tag{2.2}$$

Next, we need to choose the encoding matrix $\boldsymbol{D}$. To do so, we revisit the idea of minimizing the $L^2$ distance between inputs and reconstructions. However, since we will use the same matrix $\boldsymbol{D}$ to decode all of the points, we can no longer consider the points in isolation. Instead, we must minimize the Frobenius norm of the matrix of errors computed over all dimensions and all points:

$$\boldsymbol{D}^* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} \left(x_j^{(i)} - r(\boldsymbol{x}^{(i)})_j\right)^2} \text{ subject to } \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l \tag{2.3}$$

To derive the algorithm for finding $\boldsymbol{D}^*$, we will start by considering the case where $l = 1$. In this case, $\boldsymbol{D}$ is just a single vector, $\boldsymbol{d}$. Substituting Eq. 2.2 into Eq. 2.3 and simplifying $\boldsymbol{D}$ into $\boldsymbol{d}$, the problem reduces to

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^{(i)}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1.$$

The above formulation is the most direct way of performing the substitution, but is not the most stylistically pleasing way to write the equation. It places the scalar value $\boldsymbol{d}^\top \boldsymbol{x}^{(i)}$ on the right of the vector $\boldsymbol{d}$. It is more conventional to write scalar coefficients on the left of vector they operate on. We therefore usually write such a formula as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{d}^\top \boldsymbol{x}^{(i)} \boldsymbol{d}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1,$$

or, exploiting the fact that a scalar is its own tranpose, as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i)\top} \boldsymbol{d}\boldsymbol{d}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1.$$

The reader should aim to become familiar with such cosmetic rearrangements.

At this point, it can be helpful to rewrite the problem in terms of a single design matrix of examples, rather than as a sum over separate example vectors. This will allow us to use more compact notation. Let $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ be the matrix defined by stacking all of the vectors describing the points, such that $\boldsymbol{X}_{i,:} = \boldsymbol{x}^{(i)\top}$. We can now rewrite the problem as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} ||\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top||_F^2 \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1.$$

Disregarding the constraint for the moment, we can simplify the Frobenius norm portion as follows:

$$\arg\min_{\boldsymbol{d}} ||\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top||_F^2$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}\left(\left(\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\right)^\top \left(\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\right)\right)$$

(by the alternate definition of the Frobenius norm)

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X} - \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}) - \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)$$

$$= \arg\min_{\boldsymbol{d}} -\operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)$$

(because terms not involving $\boldsymbol{d}$ do not affect the arg min)

$$= \arg\min_{\boldsymbol{d}} -2\operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top)$$

(because we can cycle the order of the matrices inside a trace)

$$= \arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{d}\boldsymbol{d}^\top)$$

(using the same property again)

At this point, we re-introduce the constraint:

$$\arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(due to the constraint)

$$= \arg\min_{\boldsymbol{d}} -\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

This optimization problem may be solved using eigendecomposition. Specifically, the optimal $\boldsymbol{d}$ is given by the eigenvector of $\boldsymbol{X}^\top \boldsymbol{X}$ corresponding to the largest eigenvalue.

In the general case, where $l > 1$, $\boldsymbol{D}$ is given by the $l$ eigenvectors corresponding to the largest eigenvalues. This may be shown using proof by induction. We recommend writing this proof as an exercise.

# Chapter 3

# Probability and Information Theory

In this chapter, we describe probability theory. Probability theory is a mathematical framework for representing uncertain statements. It provides a means of quantifying uncertainty and axioms for deriving new uncertain statements. In artificial intelligence applications, we use probability theory in two major ways. First, the laws of probability tell us how AI systems should reason, so we design our algorithms to compute or approximate various expressions derived using probability theory. Second, we can use probability and statistics to theoretically analyze the behavior of proposed AI systems.

Probability theory is a fundamental tool of many disciplines of science and engineering. We provide this chapter to ensure that readers whose background is primarily in software engineering with limited exposure to probability theory can understand the material in this book. If you are already familiar with probability theory, feel free to skip this chapter. If you have absolutely no prior experience with probability, this chapter should be sufficient to successfully carry out deep learning research projects, but we do suggest that you consult an additional resource, such as (Jaynes, 2003).

## 3.1 Why Probability?

Many branches of computer science deal mostly with entities that are entirely deterministic and certain. A programmer can usually safely assume that a CPU will execute each machine instruction flawlessly. Errors in hardware do occur, but are rare enough that most software applications do not need to be designed to account for them. Given that many computer scientists and software engineers work in a relatively clean and certain environment, it can be surprising that

machine learning makes heavy use of probability theory.

This is because machine learning must always deal with uncertain quantities, and sometimes may also need to deal with stochastic (non-deterministic) quantities. Uncertainty and stochasticity can arise from many sources. Researchers have made compelling arguments for quantifying uncertainty using probability since at least the 1980s. Many of the arguments presented here are summarized from or inspired by Pearl (1988).

Nearly all activities require some ability to reason in the presence of uncertainty. In fact, beyond mathematical statements that are true by definition, it is difficult to think of any proposition that is absolutely true or any event that is absolutely guaranteed to occur.

There are three possible sources of uncertainty:

1. Inherent stochasticity in the system being modeled. For example, most interpretations of quantum mechanics describe the dynamics of subatomic particles as being probabilistic. We can also create theoretical scenarios that we postulate to have random dynamics, such as a hypothetical card game where we assume that the cards are truly shuffled into a random order.

2. Incomplete observability. Even deterministic systems can appear stochastic when we cannot observe all of the variables that drive the behavior of the system. For example, in the Monty Hall problem, a game show contestant is asked to choose between three doors and wins a prize held behind the chosen door. Two doors lead to a goat while a third leads to a car. The outcome given the contestant's choice is deterministic, but from the contestant's point of view, the outcome is uncertain.

3. Incomplete modeling. When we use a model that must discard some of the information we have observed, the discarded information results in uncertainty in the model's predictions. For example, suppose we build a robot that can exactly observe the location of every object around it. If the robot discretizes space when predicting the future location of these objects, then the discretization makes the robot immediately become uncertain about the precise position of objects: each object could be anywhere within the discrete cell that it was observed to occupy.

In many cases, it is more practical to use a simple but uncertain rule rather than a complex but certain one, even if the true rule is deterministic and our modeling system has the fidelity to accommodate a complex rule. For example, the simple rule "Most birds fly" is cheap to develop and is broadly useful, while a rule of the form, "Birds fly, except for very young birds that have not yet learned to fly, sick or injured birds that have lost the ability to fly, flightless species of birds

including the cassowary, ostrich and kiwi. . . " is expensive to develop, maintain and communicate, and after all of this effort is still very brittle and prone to failure.

Given that we need a means of representing and reasoning about uncertainty, it is not immediately obvious that probability theory can provide all of the tools we want for artificial intelligence applications. Probability theory was originally developed to analyze the frequencies of events. It is easy to see how probability theory can be used to study events like drawing a certain hand of cards in a game of poker. These kinds of events are often repeatable. When we say that an outcome has a probability $p$ of occurring, it means that if we repeated the experiment (e.g., draw a hand of cards) infinitely many times, then proportion $p$ of the repetitions would result in that outcome. This kind of reasoning does not seem immediately applicable to propositions that are not repeatable. If a doctor analyzes a patient and says that the patient has a 40% chance of having the flu, this means something very different—we can not make infinitely many replicas of the patient, nor is there any reason to believe that different replicas of the patient would present with the same symptoms yet have varying underlying conditions. In the case of the doctor diagnosing the patient, we use probability to represent a *degree of belief*, with 1 indicating absolute certainty and 0 indicating absolute uncertainty. The former kind of probability, related directly to the rates at which events occur, is known as *frequentist probability*, while the latter, related to qualitative levels of certainty, is known as *Bayesian probability*.

It turns out that if we list several properties that we expect common sense reasoning about uncertainty to have, then the only way to satisfy those properties is to treat Bayesian probabilities as behaving exactly the same as frequentist probabilities. For example, if we want to compute the probability that a player will win a poker game given that she has a certain set of cards, we use exactly the same formulas as when we compute the probability that a patient has a disease given that she has certain symptoms. For more details about why a small set of common sense assumptions implies that the same axioms must control both kinds of probability, see Ramsey (1926).

Probability can be seen as the extension of logic to deal with uncertainty. Logic provides a set of formal rules for determining what propositions are implied to be true or false given the assumption that some other set of propositions is true or false. Probability theory provides a set of formal rules for determining the likelihood of a proposition being true given the likelihood of other propositions.

## 3.2 Random Variables

A *random variable* is a variable that can take on different values randomly. We typically denote the random variable itself with a lower case letter in plain typeface, and the values it can take on with lower case script letters. For example, $x_1$ and $x_2$ are both possible values that the random variable $\mathrm{x}$ can take on. For vector-valued variables, we would write the random variable as $\mathbf{x}$ and one of its values as $\boldsymbol{x}$. On its own, a random variable is just a description of the states that are possible; it must be coupled with a probability distribution that specifies how likely each of these states are.

Random variables may be discrete or continuous. A discrete random variable is one that has a finite or countably infinite number of states. Note that these states are not necessarily the integers; they can also just be named states that are not considered to have any numerical value. A continuous random variable is associated with a real value.

## 3.3 Probability Distributions

A *probability distribution* is a description of how likely a random variable or set of random variables is to take on each of its possible states. The way we describe probability distributions depends on whether the variables are discrete or continuous.

### 3.3.1 Discrete Variables and Probability Mass Functions

A probability distribution over discrete variables may be described using a *probability mass function* (PMF). We typically denote probability mass functions with a capital $P$. Often we associate each random variable with a different probability mass function and the reader must infer which probability mass function to use based on the identity of the random variable, rather than the name of the function; $P(\mathrm{x})$ is usually not the same as $P(\mathrm{y})$.

The probability mass function maps from a state of a random variable to the probability of that random variable taking on that state. $P(x)$ denotes the probability that $\mathrm{x} = x$, with a probability of 1 indicating that $\mathrm{x} = x$ is certain and a probability of 0 indicating that $\mathrm{x} = x$ is impossible. Sometimes to disambiguate which PMF to use, we write the name of the random variable explicitly: $P(\mathrm{x} = x)$. Sometimes we define a variable first, then use $\sim$ notation to specify which distribution it follows later: $x \sim P(\mathrm{x})$.

Probability mass functions can act on many variables at the same time. Such a probability distribution over many variables is known as a *joint probability*

*distribution.* $P(\mathrm{x} = x, \mathrm{y} = y)$ denotes the probability that $\mathrm{x} = x$ and $\mathrm{y} = y$ simultaneously. We may also write $P(x, y)$ for brevity.

To be a probability mass function on a set of random variables x, a function $P$ must meet the following properties:

- The domain of $P$ must be the set of all possible states of x.
- $\forall x \in \mathrm{x}, 0 \leq P(x) \leq 1$. An impossible event has probability 0 and no state can be less probable than that. Likewise, an event that is guaranteed to happen has probability 1, and no state can have a greater chance of occurring.
- $\sum_{x \in \mathrm{x}} P(x) = 1$. ($P$ must guarantee that *some* state occurs.)

For example, consider a single discrete random variable x with $k$ different states. We can place a *uniform distribution* on x—that is, make each of its states equally likely—by setting its probability mass function to

$$P(\mathrm{x} = x_i) = \frac{1}{k}$$

for all $i$. We can see that this fits the requirements for a probability mass function. The value $\frac{1}{k}$ is positive because $k$ is a positive integer. We also see that $\sum_i P(\mathrm{x} = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1$, so the distribution is properly normalized.

### 3.3.2 Continuous Variables and Probability Density Functions

When working with continuous random variables, we describe probability distributions using a *probability density function (PDF)* rather than a probability mass function. To be a probability density function, a function $p$ must satisfy the following properties:

- The domain of $p$ must be the set of all possible states of x.
- $\forall x \in \mathrm{x}, p(x) \geq 0$. Note that we do not require $p(x) \leq 1$.
- $\int p(x)dx = 1$.

A probability density function $p(x)$ does not give the probability of a specific state directly, instead the probability of landing inside an infinitesimal region with volume $\delta x$ is given by $p(x)\delta x$.

We can integrate the density function to find the actual probability mass of a set of points. Specifically, the probability that $x$ lies in some set $S$ is given by the integral of $p(x)$ over that set. In the univariate example, the probability that $x$ lies in the interval $[a, b]$ is given by $_{[a,b]}\, p(x)dx$.



$\int$

For an example of a probability density function corresponding to a specific probability density over a continuous random variable, consider a uniform distribution on an interval of the real numbers. We can do this with a function $u(x; a, b)$, where $a$ and $b$ are the endpoints of the interval, with $b > a$. (The ";" notation means "parametrized by"; we consider $x$ to be the argument of the function, while $a$ and $b$ are parameters that define the function) To ensure that there is no probability mass outside the interval, we say $u(x; a, b) = 0$ for all $x \notin [a, b]$. Within $[a, b]$, $u(x; a, b) = \frac{1}{b-a}$. We can see that this is nonnegative everywhere. Additionally, it integrates to 1. We often denote that $x$ follows the uniform distribution on $[a, b]$ by writing $x \sim U(a, b)$.

## 3.4 Marginal Probability

Sometimes we know the probability distribution over a set of variables and we want to know the probability distribution over just a subset of them. The probability distribution over the subset is known as the *marginal probability* distribution.

For example, suppose we have discrete random variables x and y, and we know $P(\mathrm{x}, \mathrm{y})$. We can find $P(\mathrm{x})$ with the *sum rule*:

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y).$$

The name "marginal probability" comes from the process of computing marginal probabilities on paper. When the values of $P(\mathrm{x}, \mathrm{y})$ are written in a grid with different values of $x$ in rows and different values of $y$ in columns, it is natural to sum across a row of the grid, then write $P(x)$ in the margin of the paper just to the right of the row.

For continuous variables, we need to use integration instead of summation:

$$p(x) = \int p(x, y) dy.$$

## 3.5 Conditional Probability

In many cases, we are interested in the probability of some event, given that some other event has happened. This is called a *conditional probability*. We denote the conditional probability that $\mathrm{y} = y$ given $\mathrm{x} = x$ as $P(\mathrm{y} = y \mid \mathrm{x} = x)$. This conditional probability can be computed with the formula

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = P(\mathrm{y} = y, \mathrm{x} = x)/P(\mathrm{x} = x).$$

Note that this is only defined when $P(\mathrm{x} = x) > 0$. We cannot compute the conditional probability conditioned on an event that never happens.

It is important not to confuse conditional probability with computing what would happen if some action were undertaken. The conditional probability that a person is from Germany given that they speak German is quite high, but if a randomly selected person is taught to speak German, their country of origin does not change. Computing the consequences of an action is called making an *intervention query*. Intervention queries are the domain of *causal modeling*, which we do not explore in this book.

## 3.6 The Chain Rule of Conditional Probabilities

Any joint probability distribution over many random variables may be decomposed into conditional distributions over only one variable:

$$P(\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)})\Pi_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(i-1)}). \tag{3.1}$$

This observation is known as the *chain rule* or *product rule* of probability. It follows immediately from the definition of conditional probability. For example, applying the definition twice, we get

$$\begin{aligned} P(\mathrm{a}, \mathrm{b}, \mathrm{c}) &= P(\mathrm{a} \mid \mathrm{b}, \mathrm{c})P(\mathrm{b}, \mathrm{c}) \\ P(\mathrm{b}, \mathrm{c}) &= P(\mathrm{b} \mid \mathrm{c})P(\mathrm{c}) \\ P(\mathrm{a}, \mathrm{b}, \mathrm{c}) &= P(\mathrm{a} \mid \mathrm{b}, \mathrm{c})P(\mathrm{b} \mid \mathrm{c})P(\mathrm{c}). \end{aligned}$$

Note how every statement about probabilities remains true if we add conditions (stuff on the right-hand side of the vertical bar) consistently on all the "P"'s in the statement. We can use this to derive the same thing differently:

$$\begin{aligned} P(\mathrm{a}, \mathrm{b} \mid \mathrm{c}) &= P(\mathrm{a} \mid \mathrm{b}, \mathrm{c})P(\mathrm{b} \mid \mathrm{c}) \\ P(\mathrm{a}, \mathrm{b}, \mathrm{c}) &= P(\mathrm{a}, \mathrm{b} \mid \mathrm{c})P(\mathrm{c}) = P(\mathrm{a} \mid \mathrm{b}, \mathrm{c})P(\mathrm{b} \mid \mathrm{c})P(\mathrm{c}). \end{aligned}$$

## 3.7 Independence and Conditional Independence

Two random variables x and y are *independent* if their probability distribution can be expressed as a product of two factors, one involving only x and one involving only y:

$$\forall x \in \mathrm{x}, y \in \mathrm{y},\ p(\mathrm{x} = x, \mathrm{y} = y) = p(\mathrm{x} = x)p(\mathrm{y} = y).$$

Two random variables x and y are *conditionally independent* given a random variable z if the conditional probability distribution over x and y factorizes in this way for every value of z:

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z},\ p(\mathrm{x} = x, \mathrm{y} = y \mid \mathrm{z} = z) = p(\mathrm{x} = x \mid \mathrm{z} = z)p(\mathrm{y} = y \mid \mathrm{z} = z).$$

We can denote independence and conditional independence with compact notation: $\mathrm{x} \perp \mathrm{y}$ means that x and y are independent, while $\mathrm{x} \perp \mathrm{y} \mid \mathrm{z}$ means that x and y are conditionally independent given z.

## 3.8 Expectation, Variance and Covariance

The *expectation* or *expected value* of some function $f(x)$ with respect to a probability distribution $P(\mathrm{x})$ is the average or mean value that $f$ takes on when $x$ is drawn from $P$. For discrete variables this can be computed with a summation:

$$\mathbb{E}_{x \sim P}[f(x)] = \sum_x P(x)f(x),$$

while for continuous variables, it is computed with an integral:

$$\mathbb{E}_{x \sim P}[f(x)] = \int p(x)f(x)dx.$$

When the identity of the distribution is clear from the context, we may simply write the name of the random variable that the expectation is over, e.g. $\mathbb{E}_{\mathrm{x}}[f(x)]$. If it is clear which random variable the expectation is over, we may omit the subscript entirely, e.g. $\mathbb{E}[f(x)]$. By default, we can assume that $\mathbb{E}[\cdot]$ averages over the values of all the random variables inside the brackets. Likewise, when there is no ambiguity, we may omit the square brackets.

Expectations are linear, for example, $\mathbb{E}[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}[f(x)] + \beta \mathbb{E}[g(x)]$, when $\alpha$ and $\beta$ are fixed (not random and not depending on $x$).

The *variance* gives a measure of how much the different values of a function are spread apart:

$$\mathrm{Var}(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right].$$

When the variance is low, the values of $f(x)$ cluster near their expected value. The square root of the variance is known as the *standard deviation.*

The *covariance* gives some sense of how much two values are linearly related to each other, as well as the scale of these variables:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])\left(g(y) - \mathbb{E}\left[g(y)\right]\right)\right].$$

High absolute values of the covariance mean that the values change a lot and are both far from their respective means at the same time. If the sign of the covariance is positive, then the values tend to change in the same direction, while if it is negative, they tend to change in opposite directions. Other measures such as *correlation* normalize the contribution of each variable in order to measure only how much the variables are related, rather than also being affected by the scale of the separate variables.

The notions of covariance and dependence are conceptually related, but are in fact distinct concepts. Two random variables that have non-zero covariance are dependent. However, they may have zero covariance without being independent. For example, suppose we first generate $x$, then generate $s \in \{-1, 1\}$ with each state having probability 0.5, then generate $y$ as $s(x - \mathbb{E}[x])$. Clearly, $x$ and $y$ are not independent, because $y$ only has two possible values given $x$. However, $\mathrm{Cov}(x, y) = 0$.

The *covariance matrix* of a random vector $\boldsymbol{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that

$$\mathrm{Cov}(\mathbf{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j).$$

Note that the diagonal elements give $\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \mathrm{Var}(\mathrm{x}_i)$.

## 3.9 Information Theory

Information theory is a branch of applied mathematics that revolves around quantifying how much information is present in a signal. It was originally invented to study sending messages from discrete alphabets over a noisy channel, such as communication via radio transmission. In this context, information theory tells how to design optimal codes and calculate the expected length of messages sampled from specific probability distributions using various encoding schemes. In the context of machine learning, we can also apply information theory to continuous variables where some of these message length interpretations do not apply. This field is fundamental to many areas of electrical engineering and computer science. In this textbook, we mostly use a few key ideas from information theory to characterize probability distributions or quantify similarity between probability distributions. For more detail on information theory, see (Cover and Thomas, 2006; MacKay, 2003).

The basic intuition behind information theory is that learning that an unlikely event has occurred is more informative than learning that a likely event has occurred. A message saying "the sun rose this morning" is so uninformative as to be unnecessary to send, but a message saying "there was a solar eclipse this morning" is very informative.

We would like to quantify information in a way that formalizes this intuition. Specifically,

- Likely events should have low information content, and in the extreme case, events that are guaranteed to happen should have no information content whatsoever.
- Less likely events should have higher information content.
- Independent events should have additive information. For example, finding out that a tossed coin has come up as heads twice should convey twice as much information as finding out that a tossed coin has come up as heads once.

In order to satisfy all three of these properties, we define the *self-information* of an event $\mathrm{x} = x$ to be

$$I(x) = -\log P(x). \tag{3.2}$$

In this book, we always use log to mean the natural logarithm, with base $e$. Our definition of $I(x)$ is therefore written in units of *nats*. One nat is the amount of information gained by observing an event of probability $\frac{1}{e}$. Other texts use base-2 logarithms and units called *bits* or *shannons*; information measured in bits is just a rescaling of information measured in nats.

When x is continuous, we use the same definition of information by analogy, but some of the properties from the discrete case are lost. For example, an event with unit density still has zero information, despite not being an event that is guaranteed to occur.

Self-information deals only with a single outcome. We can quantify the amount of uncertainty in an entire probability distribution using the *Shannon entropy*[1]:

$$H(\mathrm{x}) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)]. \tag{3.3}$$

also denoted $H(P)$. In other words, the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. It actually gives a lower bound on the number of bits (if the logarithm is base 2, otherwise the units are different) needed in average to encode symbols drawn from a distribution $P$. Distributions that are nearly deterministic (where the outcome is nearly certain) have low entropy; distributions that are closer to uniform have high entropy. See Fig. 3.1 for a demonstration. When x is continous, the Shannon entropy is known as the *differential entropy*.

[1]Shannon entropy is named for Claude Shannon, the father of information theory (Shannon, 1948, 1949). For an interesting biographical account of Shannon and some of his contemporaries, see *Fortune's Formula* by William Poundstone (Poundstone, 2005).

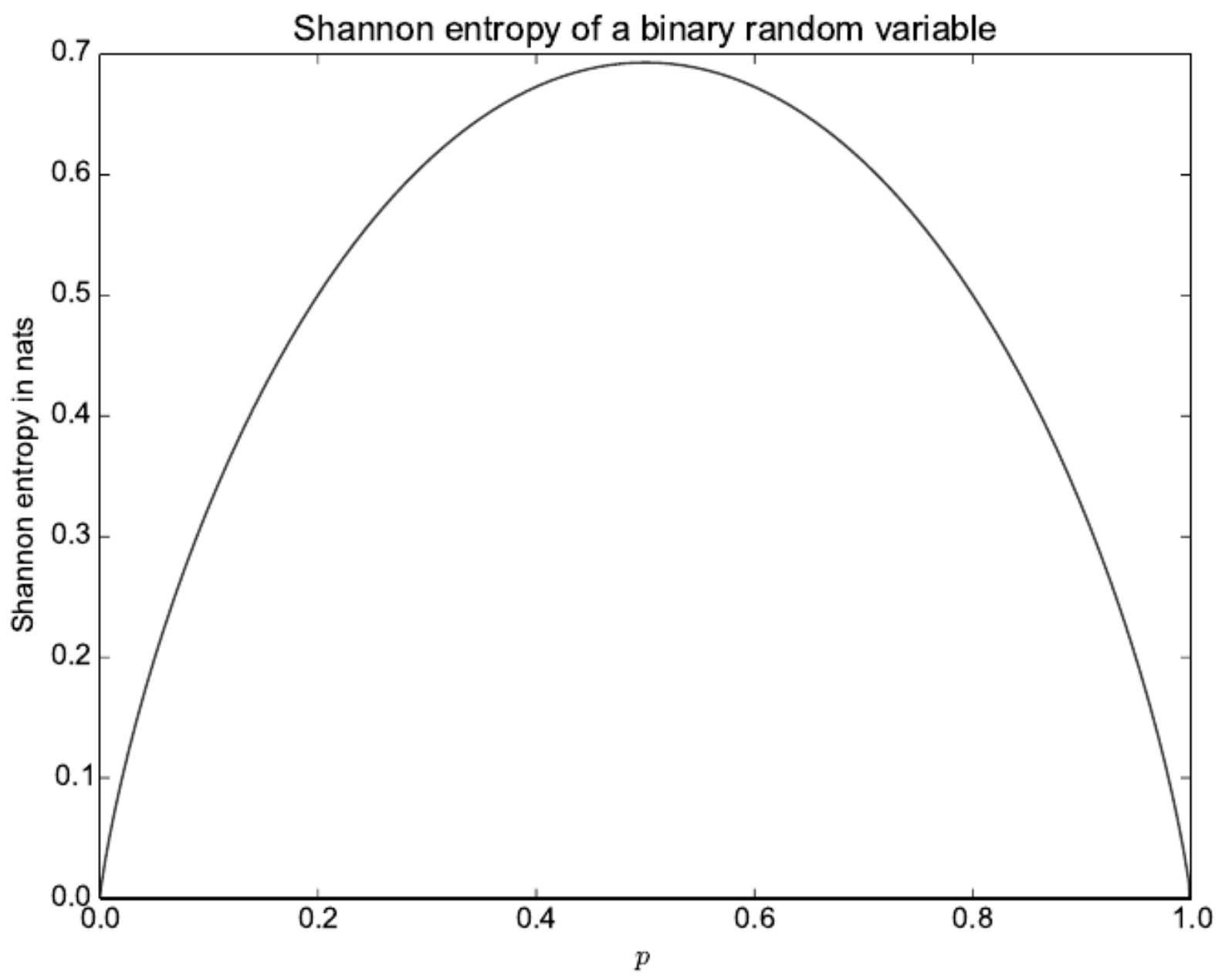


Figure 3.1: This plot shows how distributions that are closer to deterministic have low Shannon entropy while distributions that are close to uniform have high Shannon entropy. On the horizontal axis, we plot $p$, the probability of a binary random variable being equal to 1. When $p$ is near 0, the distribution is nearly deterministic, because the random variable is nearly always 0. When $p$ is near 1, the distribution is nearly deterministic, because the random variable is nearly always 1. When $p = 0.5$, the entropy is maximal, because the distribution is uniform over the two outcomes.

If we have two separate probability distributions $P(\mathrm{x})$ and $Q(\mathrm{x})$ over the same random variable x, we can measure how different these two distributions are using the *Kullback-Leibler (KL) divergence*:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{x\sim P}\left[\log P(x) - \log Q(x)\right]. \tag{3.4}$$

In the case of discrete variables, it is the extra amount of information (measured in bits if we use the base 2 logarithm, but in machine learning we usually use nats and the natural logarithm) needed to send a message containing symbols drawn from probability distribution $P$, when we use a code that was designed to minimize the length of messages drawn from probability distribution $Q$.

The KL divergence has many useful properties, most notably that it is non-negative. The KL divergence is 0 if and only if $P$ and $Q$ are the same distribution in the case of discrete variables, or equal "almost everywhere" in the case of continuous variables (see section 3.13 for details). Because the KL divergence is non-negative and measures the difference between two distributions, it is often conceptualized as measuring some sort of distance between these distributions. However, it is not a true distance measure because it is not symmetric, i.e. $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ for some $P$ and $Q$.

A quantity that is closely related to the KL divergence is the cross entropy $H(P,Q) = H(P) + D_{\mathrm{KL}}(P\|Q)$, which is similar to the KL divergence but lacking the term on the left:

$$H(P,Q) = \mathbb{E}_{x\sim P}\log Q(x).$$

When computing many of these quantities, it is common to encounter expressions of the form $0\log 0$. By convention, in the context of information theory, we treat these expressions as $\lim_{x\to 0} x\log x = 0$.

## 3.10 Common Probability Distributions

Several simple probability distributions are useful in many contexts in machine learning.

### 3.10.1 Bernoulli Distribution

The *Bernoulli* distribution is a distribution over a single binary random variable. It is controlled by a single parameter $\phi \in [0,1]$, which gives the probability of the random variable being equal to 1. It has the following properties:

$$P(\mathrm{x} = 1) = \phi$$

$$P(\mathrm{x} = 0) = 1 - \phi$$
$$P(\mathrm{x} = x) = \phi^x (1 - \phi)^{1-x}$$
$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$$
$$\mathrm{Var}_{\mathrm{x}}(\mathrm{x}) = \phi(1 - \phi)$$
$$H(\mathrm{x}) = (\phi - 1)\log(1 - \phi) - \phi \log \phi.$$

### 3.10.2 Multinoulli Distribution

The *multinoulli* or *categorical* distribution is a distribution over a single discrete variable with $k$ different states, where $k$ is finite[2]. The multinoulli distribution is parametrized by a vector $\boldsymbol{p} \in [0, 1]^{k-1}$, where $p_i$ gives the probability of the $i$-th state. The final, $k$-th state's probability is given by $1 - \mathbf{1}^\top \boldsymbol{p}$. Note that we must constrain $\mathbf{1}^\top \boldsymbol{p} \leq 1$. Multinoulli distributions are often used to refer to distributions over categories of objects, so we do not usually assume that state 1 has numerical value 1, etc. For this reason, we do not usually need to compute the expectation or variance of multinoulli-distributed random variables.

The Bernoulli and multinoulli distributions are sufficient to describe any distribution over their domain. This is because they model discrete variables for which it is feasible to simply enumerate all of the states. When dealing with continuous variables, there are uncountably many states, so any distribution described by a small number of parameters must impose strict limits on the distribution.

### 3.10.3 Gaussian Distribution

The most commonly used distribution over real numbers is the *normal distribution*, also known as the *Gaussian distribution*:

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

See Fig. 3.2 for a schematic.

The two parameters $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$ control the normal distribution. $\mu$ gives the coordinate of the central peak. This is also the mean of the distribution, i.e. $\mathbb{E}[\mathrm{x}] = \mu$. The standard deviation of the distribution is given by $\sigma$, i.e. $\mathrm{Var}(\mathrm{x}) = \sigma^2$.

[2] "Multinoulli" is a recently coined term. The multinoulli distribution is a special case of the *multinomial* distribution. A multinomial distribution is the distribution over vectors in $\{0, \ldots, n\}^k$ representing how many times each of the $k$ categories is visited when $n$ samples are drawn from a multinoulli distribution. Many texts use the term "multinomial" to refer to multinoulli distributions without clarifying that they refer only to the $n = 1$ case.

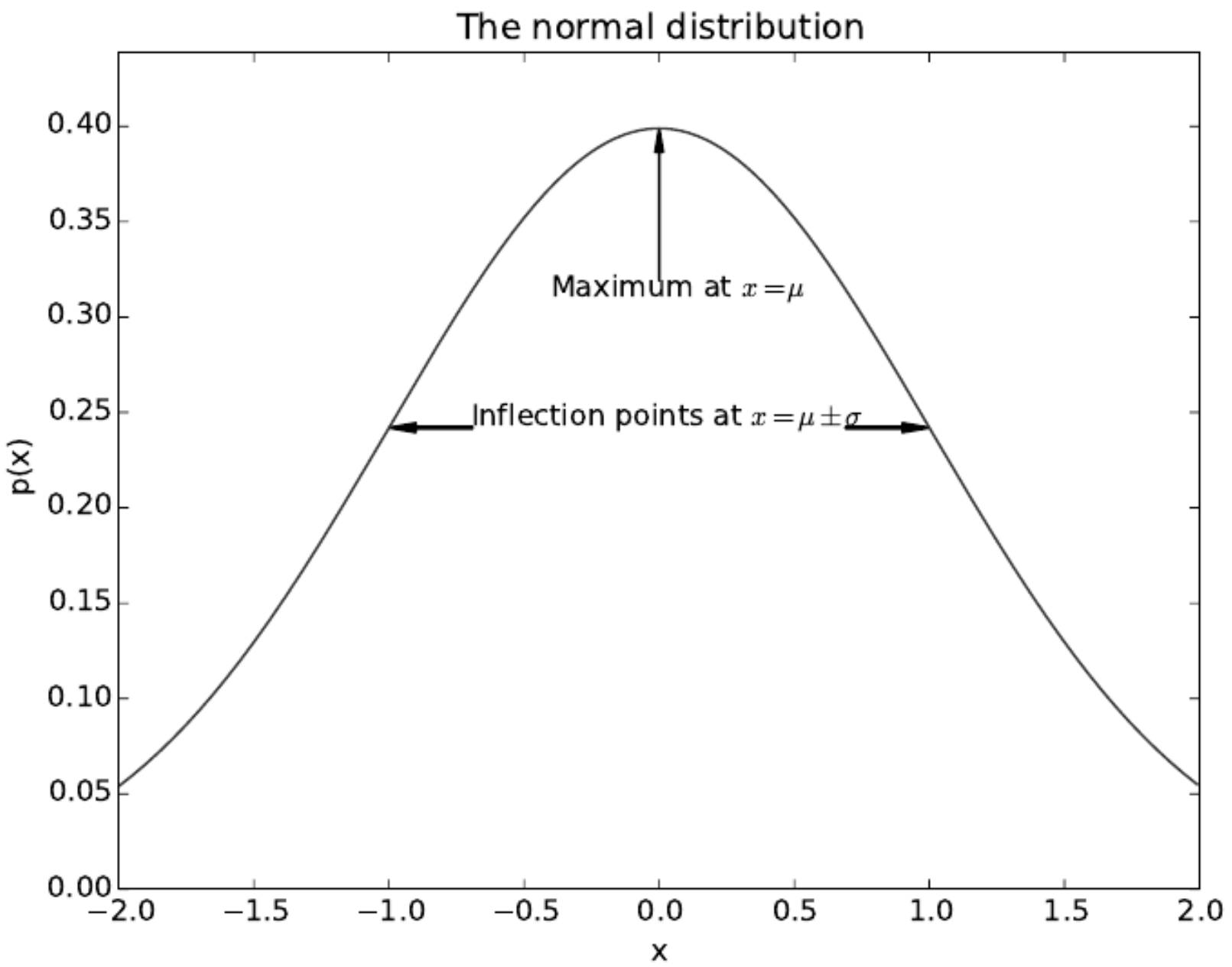


Figure 3.2: *The normal distribution*: The normal distribution $\mathcal{N}(x \mid \mu, \sigma^2)$ exhibits a classic "bell curve" shape, with the $x$ coordinate of its central peak given by $\mu$, and the width of its peak controlled by $\sigma$. In this example, we depict the *standard normal distribution*, with $\mu = 0$ and $\sigma = 1$.

Note that when we evaluate the PDF, we need to square and invert $\sigma$. When we need to frequently evaluate the PDF with different parameter values, a more efficient way of parametrizing the distribution is to use a parameter $\beta \in \mathbb{R}^+$ to control the *precision* or inverse variance of the distribution:

$$\mathcal{N}(x \mid \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x-\mu)^2\right).$$

Normal distributions are a sensible choice for many applications. In the absence of prior knowledge about what form a distribution over the real numbers should take, the normal distribution is a good default choice for two major reasons.

First, many distributions we wish to model are truly close to being normal distributions. The *central limit theorem* shows that the sum of many independent random variables is approximately normally distributed. This means that in practice, many complicated systems can be modeled successfully as normally distributed noise, even if the system can be decomposed into parts with more structured behavior.

Second, the normal distribution in some sense makes the fewest assumptions of any distribution over the reals, so choosing to use it inserts the least amount of prior knowledge into a model. Out of all distributions with the same variance, the normal distribution has the highest entropy. It is not possible to place a uniform distribution on all of $\mathbb{R}$. The closest we can come to doing so is to use a normal distribution with high variance.

The normal distribution generalizes to $\mathbb{R}^n$, in which case it is known as the *multivariate normal distribution*. It may be parametrized with a positive definite symmetric matrix $\boldsymbol{\Sigma}$:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sqrt{\frac{1}{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

The parameter $\boldsymbol{\mu}$ still gives the mean of the distribution, though now it is vector-valued. The parameter $\boldsymbol{\Sigma}$ gives the covariance matrix of the distribution. As in the univariate case, the covariance is not necessarily the most computationally efficient way to parametrize the distribution, since we need to invert $\boldsymbol{\Sigma}$ to evaluate the PDF. We can instead use a *precision matrix* $\boldsymbol{\beta}$:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\beta}(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

### 3.10.4 Exponential and Laplace Distributions

In the context of deep learning, we often want to have a probability distribution with a sharp point at $x = 0$. To accomplish this, we can use the *exponential distribution*:

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x).$$

The exponential distribution uses the indicator function $\mathbf{1}_{x \geq 0}$ to assign probability zero to all negative values of $x$.

A closely related probability distribution that allows us to place a sharp peak of probability mass at an arbitrary point $\mu$ is the *Laplace distribution*

$$p(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The Laplace distribution is sometimes also called the *double exponential distribution* because it can be written as the sum of two exponential distributions, with one flipped and both shifted to reposition the mode to $x = \mu$. The term "Laplace distribution" is preferred because "double exponential distribution" also has other meanings.

### 3.10.5 Dirac Distribution

In some cases, we wish to specify that all of the mass in a probability distribution clusters around a single point. This can be accomplished by defining a PDF using the Dirac delta function, $\delta(x)$:

$$p(x) = \delta(x - \mu).$$

The Dirac delta function is defined such that it is zero-valued everywhere but 0, yet integrates to 1. By defining $p(x)$ to be $\delta$ shifted by $-\mu$ we obtain an infinitely narrow and infinitely high peak of probability mass where $x = \mu$.

A common use of the Dirac delta distribution is as a component of the so-called *empirical distribution*,

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i) \tag{3.5}$$

which puts probability mass $\frac{1}{n}$ on each of the $n$ points $x_1, \ldots x_n$ forming a given data set or collection of samples. The Dirac delta distribution is only necessary to define the empirical distribution over continuous variables. For discrete variables, the situation is simpler: an empirical distribution can be conceptualized as a multinoulli distribution, with a probability associated to each possible input value that is simply equal to the *empirical frequency* of that value in the training set.

We can view the empirical distribution formed from a dataset of training examples as specifying the distribution that we sample from when we train a model on this dataset. Another important perspective on the empirical distribution is that it is the probability density that maximizes the likelihood of the training data (see Section 5.6). Many machine learning algorithms can be configured to have arbitrarily high capacity. If given enough capacity, these algorithms will simply learn the empirical distribution. This is a bad outcome because the model does not generalize at all and assigns infinitesimal probability to any point in space that did not occur in the training set. A central problem in machine learning is studying how to limit the capacity of a model in a way that prevents it from simply learning the empirical distribution while also allowing it to learn complicated functions.

The empirical distribution is a particular form of *mixture*, discussed next.

### 3.10.6 Mixtures of Distributions

It is also common to define probability distributions by composing other simpler probability distributions. One common way of combining distributions is to construct a *mixture distribution*. A mixture distribution is made up of several component distributions. On each trial, the choice of which component distribution generates the sample is determined by sampling a component identity from a multinoulli distribution:

$$P(\mathrm{x}) = \sum_i P(\mathrm{c} = i) P(\mathrm{x} \mid \mathrm{c} = i)$$

where $P(\mathrm{c})$ is the multinoulli distribution over component identities.

The mixture model is one simple strategy for combining probability distributions to create a richer distribution. In chapter 13, we explore the art of building complex probability distributions from simple ones in more detail.

The mixture model allows us to briefly glimpse a concept that will be of paramount importance later—the *latent variable*. A latent variable is a random variable that we cannot observe directly. Latent variables may be related to x through their joint distribution $P(\mathrm{x}, \mathrm{c}) = P(\mathrm{x} \mid \mathrm{c}) P(\mathrm{c})$. The distribution $P(\mathrm{c})$ over the latent variable and the distribution $P(\mathrm{x} \mid \mathrm{c})$ related the latent variables to the visible variables determines the shape of the distribution $P(\mathrm{x})$ even though it is possible to describe $P(\mathrm{x})$ without reference to the latent variable. Latent variables are discussed further in Section 13.4.

A very powerful and common type of mixture model is the *Gaussian mixture* model, in which the components $P(\mathrm{x} \mid \mathrm{c} = i)$ are Gaussians, each with its mean $\boldsymbol{\mu}_i$ and covariance $\boldsymbol{\Sigma}_i$. Some mixtures can have more constraints, for example, the covariances could be shared across components, i.e., $\boldsymbol{\Sigma}_i = \boldsymbol{\Sigma}_j = \boldsymbol{\Sigma}$, or the covariance

matrices could be constrained to be diagonal or simply equal to a scalar times the identity. A Gaussian mixture model is a *universal approximator* of densities, in the sense that any smooth density can be approximated to a particular precision by a Gaussian mixture model with enough components. Gaussian mixture models have been used in many settings, and are particularly well known for their use as acoustic models in speech recognition (Bahl *et al.*, 1987).

## 3.11 Useful Properties of Common Functions

Certain functions arise often while working with probability distributions, especially the probability distributions used in deep learning models.

One of these functions is the *logistic sigmoid*:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

The logistic sigmoid is commonly used to produce the $\phi$ parameter of a Bernoulli distribution because its range is $(0, 1)$, which lies within the valid range of values for the $\phi$ parameter. See Fig. 3.3 for a graph of the sigmoid function.

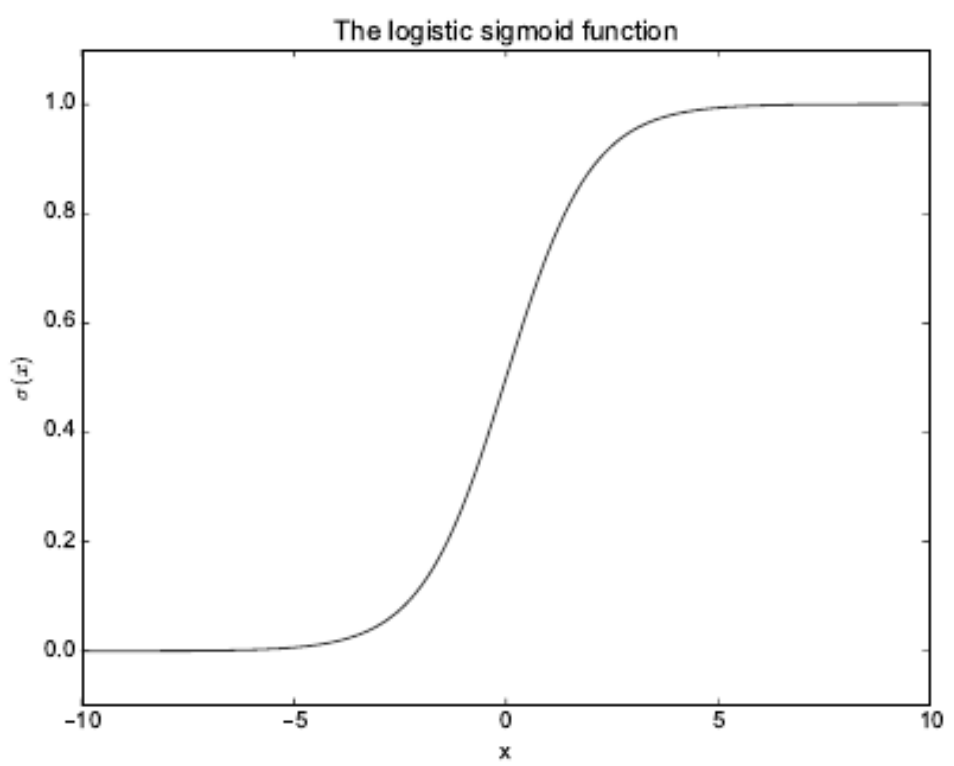


Figure 3.3: The logistic sigmoid function.

Another commonly encountered function is the *softplus* function (Dugas *et al.*, 2001):

$$\zeta(x) = \log\left(1 + \exp(x)\right).$$

The softplus function can be useful for producing the $\beta$ or $\sigma$ parameter of a normal distribution because its range is $\mathbb{R}^+$. It also arises commonly when manipulating

expressions involving sigmoids. The name of the softplus function comes from the fact that it is a smoothed or "softened" version of

$$x^+ = \max(0, x).$$

See Fig. 3.4 for a graph of the softplus function.

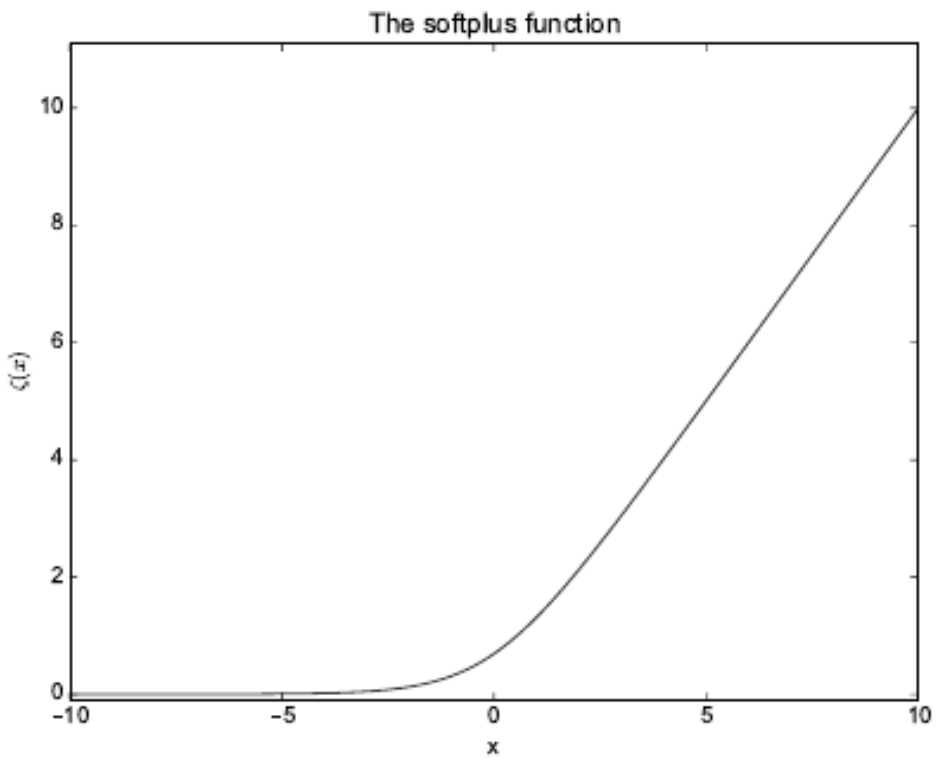


Figure 3.4: The softplus function.

The following properties are all useful enough that you may wish to memorize them:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$1 - \sigma(x) = \sigma(-x)$$

$$\log \sigma(x) = -\zeta(-x)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x)$$

$$\forall x \in (0, 1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1 - x}\right)$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log\left(\exp(x) - 1\right)$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy$$

$$\zeta(x) - \zeta(-x) = x$$

The function $\sigma^{-1}(x)$ is called the *logit* in statistics, but this term is more rarely used in machine learning. The final property provides extra justification for the name "softplus", since $x^+ - x^- = x$.

## 3.12 Bayes' Rule

We often find ourselves in a situation where we know $P(\mathrm{y} \mid \mathrm{x})$ and need to know $P(\mathrm{x} \mid \mathrm{y})$. Fortunately, if we also know $P(\mathrm{x})$, we can compute the desired quantity using *Bayes' rule*:

$$P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})}.$$

Note that while $P(\mathrm{y})$ appears in the formula, it is usually feasible to compute $P(\mathrm{y}) = \sum_x P(\mathrm{y} \mid x)P(x)$, so we do not need to begin with knowledge of $P(\mathrm{y})$.

Bayes' rule is straightforward to derive from the definition of conditional probability, but it is useful to know the name of this formula since many texts refer to it by name. It is named after the Reverend Thomas Bayes, who first discovered a special case of the formula. The general version presented here was independently discovered by Pierre-Simon Laplace.

## 3.13 Technical Details of Continuous Variables

A proper formal understanding of continuous random variables and probability density functions requires developing probability theory in terms of a branch of mathematics known as *measure theory*. Measure theory is beyond the scope of this textbook, but we can briefly sketch some of the issues that measure theory is employed to resolve.

In section 3.3.2, we saw that the probability of a continuous vector-valued $\mathbf{x}$ lying in some set $S$ is given by the integral of $p(\boldsymbol{x})$ over the set $S$. Some choices of set $S$ can produce paradoxes. For example, it is possible to construct two sets $S_1$ and $S_2$ such that $P(S_1) + P(S_2) > 1$ but $S_1 \cap S_2 = \emptyset$. These sets are generally constructed making very heavy use of the infinite precision of real numbers, for example by making fractal-shaped sets or sets that are defined by transforming the set of rational numbers[3]. One of the key contributions of measure theory is to provide a characterization of the set of sets that we can compute the probability of without encountering paradoxes. In this book, we only integrate over sets with relatively simple descriptions, so this aspect of measure theory never becomes a relevant concern.

[3]The Banach-Tarski theorem provides a fun example of such sets.

For our purposes, measure theory is more useful for describing theorems that apply to most points in $\mathbb{R}^n$ but do not apply to some corner cases. Measure theory provides a rigorous way of describing that a set of points is negligibly small. Such a set is said to have "*measure zero*". We do not formally define this concept in this textbook. However, it is useful to understand the intuition that a set of measure zero occupies no volume in the space we are measuring. For example, within $\mathbb{R}^2$, a line has measure zero, while a filled polygon has positive measure. Likewise, an individual point has measure zero. Any union of countably many sets that each have measure zero also has measure zero (so the set of all the rational numbers has measure zero, for instance).

Another useful term from measure theory is "*almost everywhere*". A property that holds almost everywhere holds throughout all of space except for on a set of measure zero. Because the exceptions occupy a negligible amount of space, they can be safely ignored for many applications. Some important results in probability theory hold for all discrete values but only hold "almost everywhere" for continuous values.

One other detail we must be aware of relates to handling random variables that are deterministic functions of one another. Suppose we have two random variables, $\boldsymbol{x}$ and $\boldsymbol{y}$, such that $\boldsymbol{y} = g(\boldsymbol{x})$. You might think that $p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))$. This is actually not the case.

Suppose $\boldsymbol{y} = \frac{\boldsymbol{x}}{2}$ and $\boldsymbol{x} \sim U(0,1)$. If we use the rule $p_y(\boldsymbol{y}) = p_x(2\boldsymbol{y})$ then $p_y$ will be 0 everywhere except the interval $[0, \frac{1}{2}]$, and it will be 1 on this interval. This means

$$\int p_y(\boldsymbol{y}) d\boldsymbol{y} = \frac{1}{2}$$

which violates the definition of a probability distribution.

This common mistake is wrong because it fails to account for the distortion of space introduced by the function $g(\boldsymbol{x})$. Recall that the probability of $\boldsymbol{x}$ lying in an infinitesimally small region with volume $\delta\boldsymbol{x}$ is given by $p(\boldsymbol{x})\delta\boldsymbol{x}$. Since $g$ can expand or contract space, the infinitesimal volume surrounding $\boldsymbol{x}$ in $\boldsymbol{x}$ space may have different volume in $\boldsymbol{y}$ space. To correct the problem, we need to preserve the property

$$|p_y(g(\boldsymbol{x}))dy| = |p_x(\boldsymbol{x})dx|.$$

Solving from this, we obtain

$$p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))|\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}}|$$

or equivalently

$$p_x(\boldsymbol{x}) = p_y(g(\boldsymbol{x}))|\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}}|. \tag{3.6}$$

In higher dimensions, the absolute value of the derivative generalizes to the determinant of the *Jacobian matrix* — the matrix with $J_{i,j} = \frac{\partial x_i}{\partial y_j}$.

## 3.14 Structured Probabilistic Models

Machine learning algorithms often involve probability distributions over a very large number of random variables. Often, these probability distributions involve direct interactions between relatively few variables. Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together. For example, suppose we have three random variables, a, b, and c. Suppose that a influences the value of b and b influences the value of c, but that a and c are independent given b. We can represent the probability distribution over all three variables as a product of probability distributions over two variables:

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{b}).$$

These factorizations can greatly reduce the number of parameters needed to describe the distribution. Each factor uses a number of parameters that is exponential in the number of variables in the factor. This means that we can greatly reduce the cost of representing a distribution if we are able to find a factorization into distributions over fewer variables.

We can describe these kinds of factorizations using graphs. Here we use the word "graph" in the sense of graph theory, i.e. a set of vertices that may be connected to each other with edges. When we represent the factorization of a probability distribution with a graph, we call it a *structured probabilistic model* or *graphical model*.

There are two main kinds of structured probabilistic models: directed and undirected. Both kinds of graphical models use a graph in which each node in the graph corresponds to a random variable, and an edge connecting two random variables means that the probability distribution is able to represent direct interactions between those two random variables.

*Directed* models use graphs with directed edges, and they represent factorizations into conditional probability distributions, as in the example above. Specifically, a directed model contains one factor for every random variable $\mathrm{x}_i$ in the distribution, and that factor consists of the conditional distribution over $\mathrm{x}_i$ given the parents of $\mathrm{x}_i$:

$$p(\mathbf{x}) = \prod_i p\left(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i)\right).$$

See Fig. 3.5 for an example of a directed graph and the factorization of probability distributions it represents.

*Undirected* models use graphs with undirected edges, and they represent factorizations into a set of functions; unlike in the directed case, these functions are usually not probability distributions of any kind. Any set of nodes that are all connected to each other in $\mathcal{G}$ is called a clique. Each clique $\mathcal{C}^{(i)}$ in an undirected model is associated with a factor $\phi^{(i)}(\mathcal{C}^{(i)})$. These factors are just functions, not probability distributions. The output of each factor must be non-negative, but there is no constraint that the factor must sum or integrate to 1 like a probability distribution.

The probability of a configuration of random variables is *proportional* to the product of all of these factors—assignments that result in larger factor values are more likely. Of course, there is no guarantee that this product will sum to 1. We therefore divide by a normalizing constant $Z$, defined to be the sum or integral over all states of the product of the $\phi$ functions, in order to obtain a normalized probability distribution:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)} \left( \mathcal{C}^{(i)} \right).$$

See Fig. 3.6 for an example of an undirected graph and the factorization of probability distributions it represents.

Keep in mind that these graphical representations of factorizations are a language for describing probability distributions. They are not mutually exclusive families of probability distributions. Being directed or undirected is not a property of a probability distribution; it is a property of a particular *description* of a probability distribution, but any probability distribution may be described in both ways.

Throughout part I and part II of this book, we will use structured probabilistic models merely as a language to describe which direct probabilistic relationships different machine learning algorithms choose to represent. No further understanding of structured probabilistic models is needed until the discussion of research topics, in part III, where we will explore structured probabilistic models in much greater detail.

## 3.15 Example: Naive Bayes

We now know enough probability theory that we can perform some simple applications with a probabilistic model. In this example, we will show how to infer the probability that a patient has the flu using a simple probabilistic model. For now,

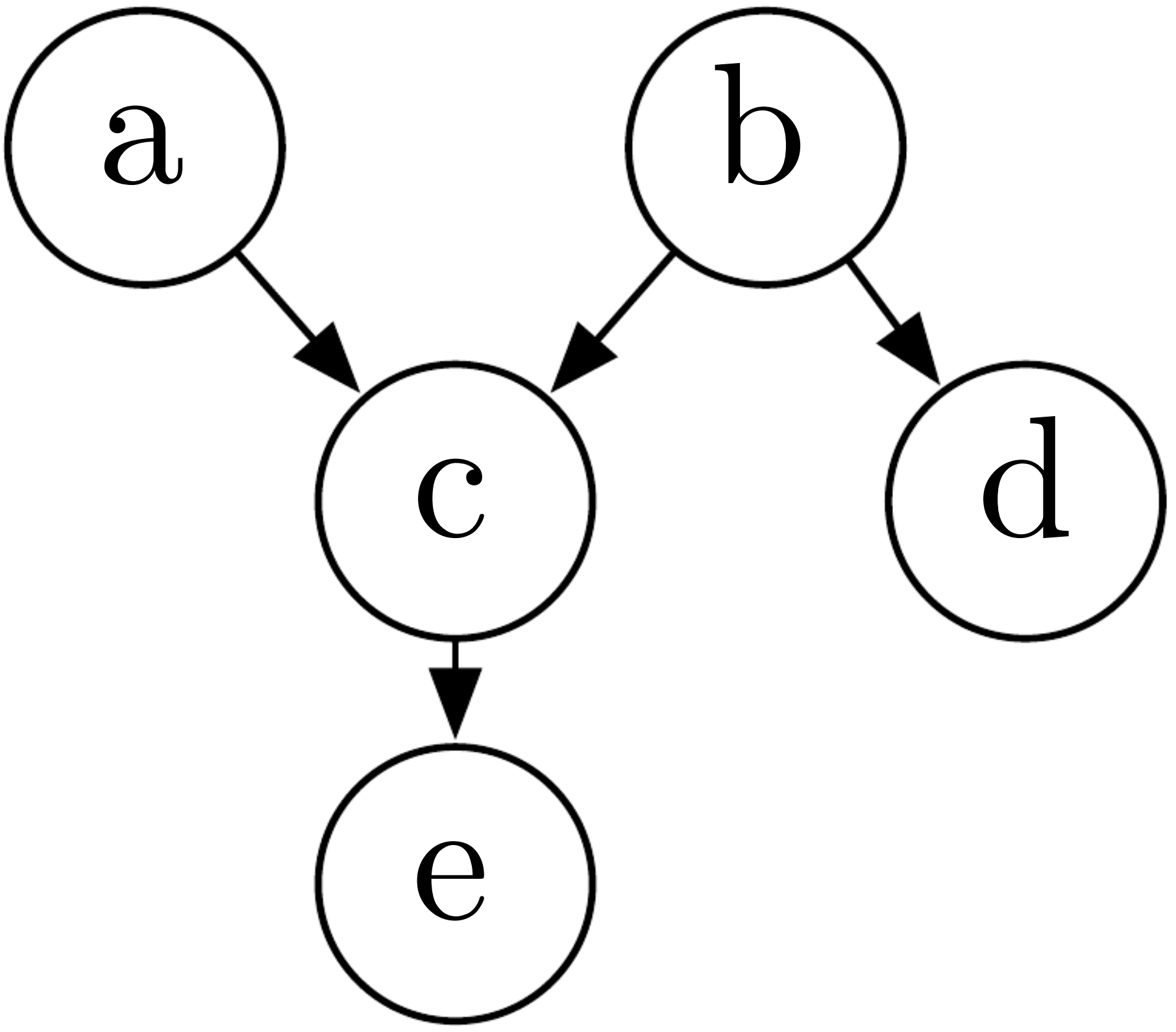


Figure 3.5: A directed graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as $p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c})$. This graph allows us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

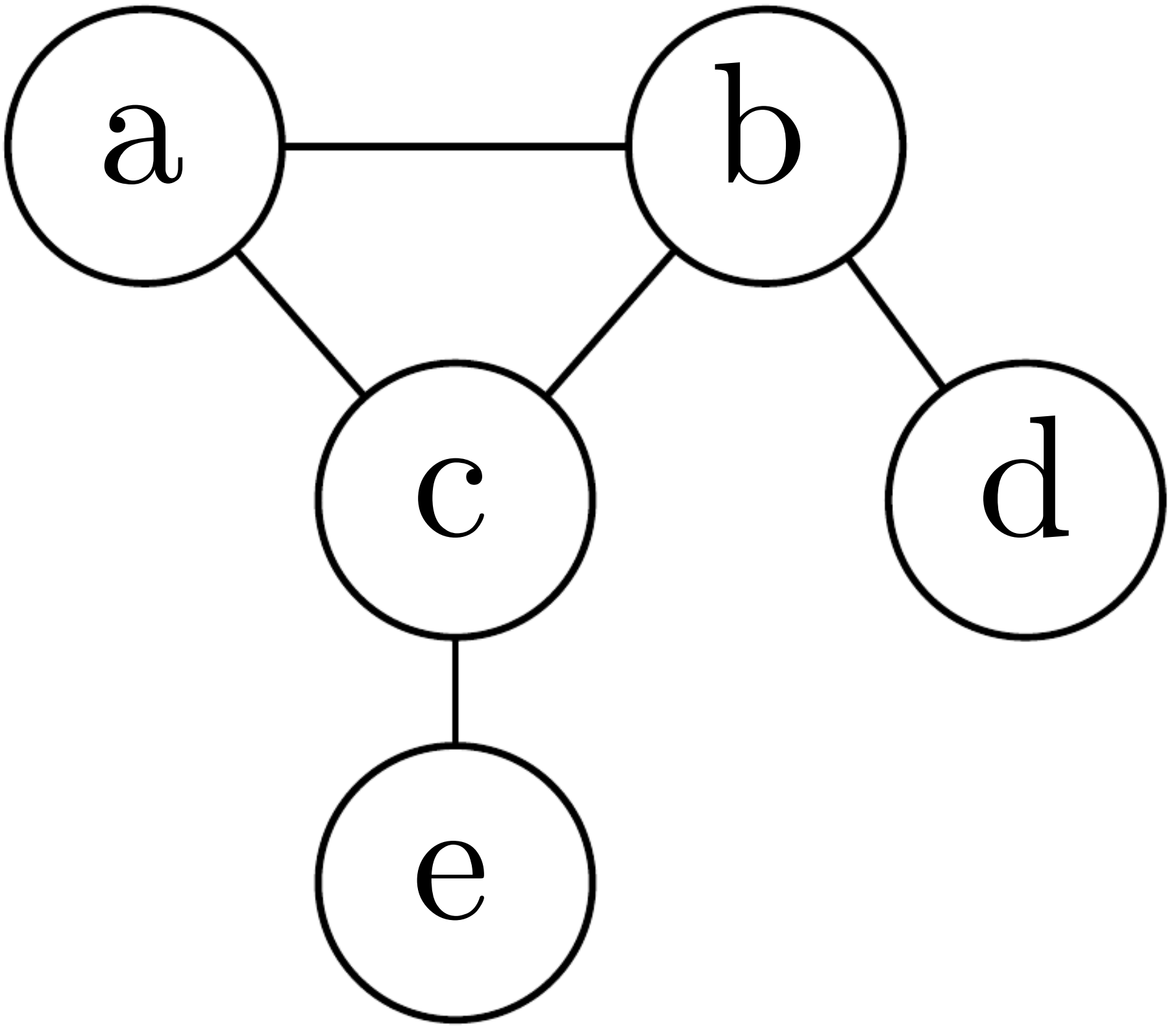


Figure 3.6: An undirected graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as $p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = \frac{1}{Z}\phi^{(1)}(\mathrm{a}, \mathrm{b}, \mathrm{c})\phi^{(2)}(\mathrm{b}, \mathrm{d})\phi^{(3)}(\mathrm{c}, \mathrm{e})$. This graph allows us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

we will assume that we just know the correct model somehow. Later chapters will cover the concepts needed to learn the model from data.

The *Naive Bayes* model is a simple probabilistic model that is often used to recognize patterns. The model consists of one random variable $\mathrm{c}$ representing a category and a set of random variables $\mathbb{F} = \{\mathrm{f}^{(1)}, \ldots, \mathrm{f}^{(n)}\}$ representing features of objects in each category. In this example, we'll use Naive Bayes to diagnose patients as having the flu or not. The random variable $\mathrm{c}$ can thus have two values: $c_0$ representing the category of patients who do not have the flu, and $c_1$ representing the category of patients who do. Suppose $\mathrm{f}^{(1)}$ is the random variable representing whether the patient has a sore throat, with $f_0^{(1)}$ representing no sore throat, and $f_1^{(1)}$ representing a sore throat. Suppose $\mathrm{f}^{(2)} \in \mathbb{R}$ is the patient's temperature in degrees Celsius.

When using the Naive Bayes model, we assume that all of the features are independent from each other given the category:

$$P(\mathrm{c}, \mathrm{f}^{(1)}, \ldots, \mathrm{f}^{(n)}) = P(\mathrm{c}) \prod_i P(\mathrm{f}^{(i)} \mid \mathrm{c}).$$

See Fig. 3.7 for a directed graphical model that expresses these conditional independence assumptions. These assumptions are very strong and unlikely to be true in naturally occuring situations, hence the name "naive". Surprisingly, Naive Bayes often produces good predictions in practice (even though the assumptions do not hold precisely), and is a good baseline model to start with when tackling a new problem.

Beyond these conditional independence assumptions, the Naive Bayes framework does not specify anything about the probability distribution. The specific choice of distributions is left up to the designer. In our flu example, let's make $P(\mathrm{c})$ a Bernoulli distribution, with $P(\mathrm{c} = c_1) = \phi^{(\mathrm{c})}$. We can also make $P(\mathrm{f}^{(1)} \mid \mathrm{c})$ a Bernoulli distribution, with

$$P(\mathrm{f}^{(1)} = f_1^{(1)} \mid \mathrm{c} = c) = \phi_c^{\mathrm{f}}.$$

In other words, the Bernoulli parameter changes depending on the value of $\mathrm{c}$. Finally, we need to choose the distribution over $\mathrm{f}^{(2)}$. Since $\mathrm{f}^{(2)}$ is real-valued, a normal distribution is a good choice. Because $\mathrm{f}^{(2)}$ is a temperature, there are hard limits to the values it can take on—it cannot go below $0K$, for example. Fortunately, these values are so far from the values measured in human patients that we can safely ignore these hard limits. Values outside the hard limits will receive extremely low probability under the normal distribution so long as the mean and variance are set correctly. As with $\mathrm{f}^{(1)}$, we need to use different parameters for different values of $\mathrm{c}$, to represent that patients with the flu have different

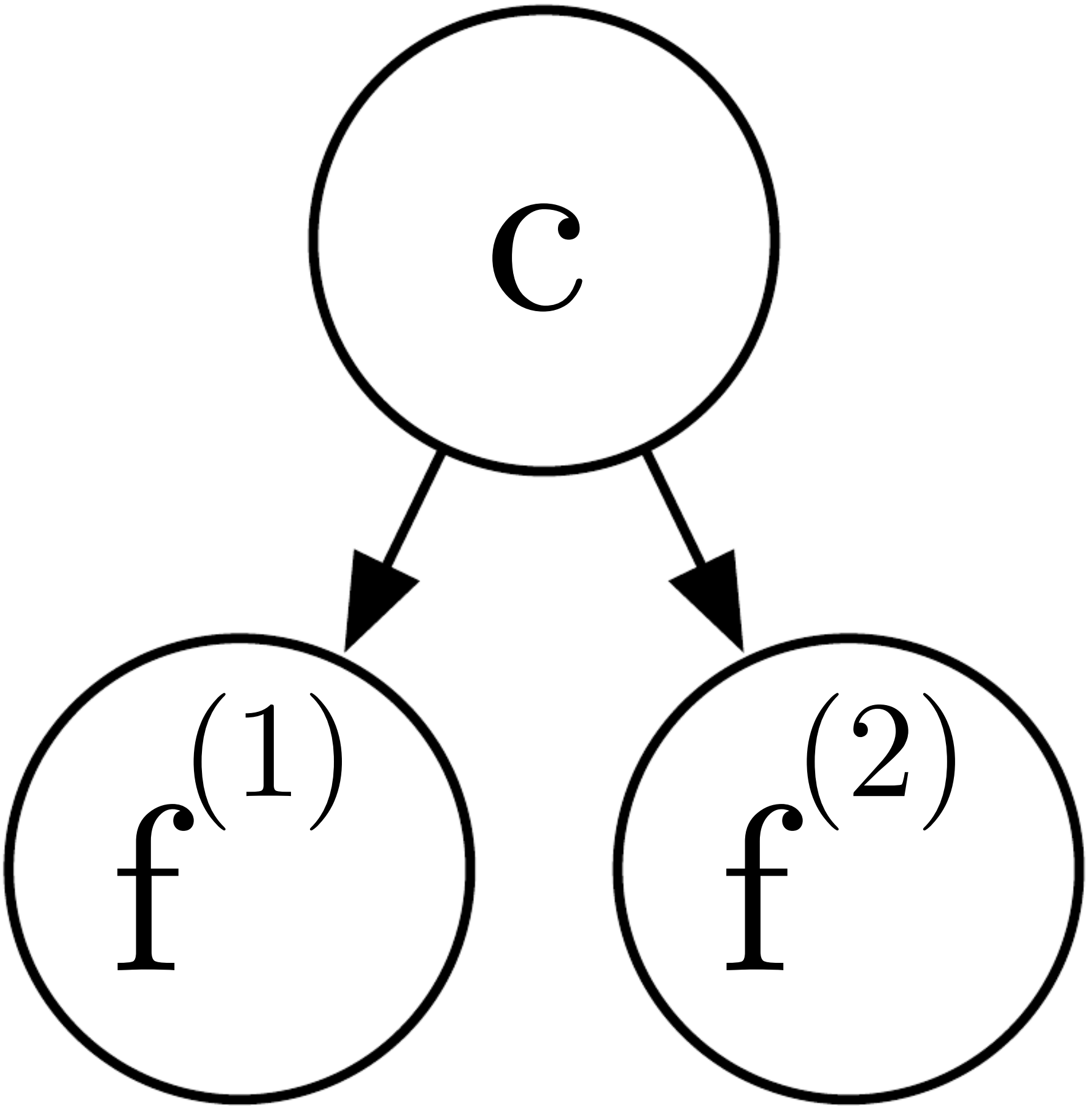


Figure 3.7: A directed graphical model depicting the conditional independence assumptions used by the Naive Bayes model.

temperatures than patients without it:

$$\mathrm{f}^{(2)} \sim N(\mathrm{f}^{(2)} \mid \mu_c, \sigma_c^2).$$

Now we are ready to determine how likely a patient is to have the flu. To do this, we want to compute $P(\mathrm{c} \mid \mathbb{F})$, but we know $P(\mathrm{c})$ and $P(\mathbb{F} \mid \mathrm{c})$. This suggests that we should use Bayes' rule to determine the desired distribution. The word "Bayes" in the name "Naive Bayes" comes from this frequent use of Bayes' rule in conjunction with the model. We begin by applying Bayes' rule:

$$P(\mathrm{c} \mid \mathbb{F}) = \frac{P(\mathrm{c})P(\mathbb{F} \mid \mathrm{c})}{P(\mathbb{F})}. \tag{3.7}$$

We do not know $P(\mathbb{F})$. Fortunately, it is easy to compute:

$$P(\mathbb{F}) = \sum_{c \in \mathrm{c}} P(\mathrm{c} = c, \mathbb{F}) \text{ (by the sum rule)}$$

$$= \sum_{c \in \mathrm{c}} P(\mathrm{c} = c)P(\mathbb{F} \mid \mathrm{c} = c) \text{ (by the chain rule)}.$$

Substituting this result back into equation 3.7, we obtain

$$P(\mathrm{c} \mid \mathbb{F}) = \frac{P(\mathrm{c})P(\mathbb{F} \mid \mathrm{c})}{\sum_{c \in \mathrm{c}} P(\mathrm{c} = c)P(\mathbb{F} \mid \mathrm{c} = c)}$$

$$= \frac{P(\mathrm{c})\Pi_i P(\mathrm{f}^{(i)} \mid \mathrm{c})}{\sum_{c \in \mathrm{c}} P(\mathrm{c} = c)\Pi_i P(\mathrm{f}^{(i)} \mid \mathrm{c} = c)}$$

by the Naive Bayes assumptions. This is as far as we can simplify the expression for a general Naive Bayes model.

We can simplify the expression further by substituting in the definitions of the particular probability distributions we have defined for our flu diagnosis example:

$$P(\mathrm{c} = c \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) = \frac{g(c)}{\sum_{c' \in \mathrm{c}} g(c')}$$

where

$$g(c) = P(\mathrm{c} = c)P(\mathrm{f}^{(1)} = f^{(1)} \mid \mathrm{c} = c)P(\mathrm{f}^{(2)} = f^{(2)} \mid \mathrm{c} = c).$$

Since c only has two possible values in our example, we can simplify this to:

$$P(\mathrm{c} = 1 \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) = \frac{g(1)}{g(0) + g(1)}$$

$$= \frac{1}{1 + \frac{g(0)}{g(1)}}$$

$$
\begin{aligned}
&= \frac{1}{1 + \exp\left(\log g(0) - \log g(1)\right)} \\
&= \sigma\left(\log g(1) - \log g(0)\right). \qquad (3.8)
\end{aligned}
$$

To go further, let's simplify $\log g(i)$:

$$
\log g(i) = \log \quad \phi^{(\mathrm{c})i}(1-\phi^{(\mathrm{c})})^{1-i}\phi_1^{(\mathrm{f})f_1}(1-\phi_1^{(\mathrm{f})})^{1-f_1}\sqrt{\frac{1}{2\pi\sigma_i^2}}\exp\left(-\frac{1}{2\sigma_i^2}(f_2-\mu_i)^2\right)\Bigg)
$$

$$
= i\log\phi^{(\mathrm{c})} + (1-i)\log\Big(1-\phi^{(\mathrm{c})}\Big) + f_1\log\phi_i^{(\mathrm{f})} + (1-f_1)\log(1-\phi_i^{(\mathrm{f})}) + \frac{1}{2}\log\frac{1}{2\pi\sigma_i^2} - \frac{1}{2\sigma_i^2}(f_2-\mu_i)^2.
$$

Substituting this back into equation 3.8, we obtain

$$
P(\mathrm{c} = c \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) =
$$

$$
\begin{aligned}
\sigma\Bigg(&\log\phi^{(\mathrm{c})} - \log(1-\phi^{(\mathrm{c})}) + f_1\log\phi_1^{(\mathrm{f})} + (1-f_1)\log(1-\phi_1^{(\mathrm{f})}) \\
&\qquad - f_1\log\phi_0^{(\mathrm{f})} + (1-f_1)\log(1-\phi_0^{(\mathrm{f})}) \\
&- \frac{1}{2}\log 2\pi\sigma_1^2 + \frac{1}{2}\log 2\pi\sigma_0^2 - \frac{1}{2\sigma_1^2}(f_2-\mu_1)^2 + \frac{1}{2\sigma_0^2}(f_2-\mu_0)^2\Bigg).
\end{aligned}
$$

From this formula, we can read off various intuitive properties of the Naive Bayes classifier's behavior on this example problem, regarding the *inference* that can be drawn from a trained model. The probability of the patient having the flu grows like a sigmoidal curve. We move farther to the left as $f_2$, the patient's temperature, moves farther away from $\mu_1$, the average temperature of a flu patient.

# Chapter 4

# Numerical Computation

Machine learning algorithms usually require a high amount of numerical computation. This typically refers to algorithms that solve mathematical problems by methods that iteratively update estimates of the solution, rather than analytically deriving a formula providing a symbolic expression for the correct solution. Common operations include solving systems of linear equations and finding the value of an argument that minimizes a function. Even just evaluating a mathematical function on a digital computer can be difficult when the function involves real numbers, which cannot be represented precisely using a finite amount of memory.

## 4.1 Overflow and Underflow

The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns. This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer. In many cases, this is just rounding error. Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in theory to fail in practice if they are not designed to minimize the accumulation of rounding error.

One form of rounding error that is particularly devastating is *underflow*. Underflow occurs when numbers near zero are rounded to zero. Many functions behave qualitatively differently when their argument is zero rather than a small positive number. For example, we usually want to avoid division by zero (some software environments will raise exceptions when this occurs, otherwise will return a result with a placeholder not-a-number value) or taking the logarithm of zero (this is usually treated as $-\infty$, which then becomes not-a-number if it is used for further arithmetic).

Another highly damaging form of numerical error is *overflow*. Overflow occurs when numbers with large magnitude are approximated as $\infty$ or $-\infty$. Further arithmetic will usually change this infinite values into not-a-number values.

For an example of the need to design software implementations to deal with overflow and underflow, consider the softmax function, typically used to predict the probabilities associated with a multinoulli distribution:

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_j^n \exp(x_j)}.$$

Consider what happens when all of the $x_i$ are equal to some constant $c$. Analytically, we can see that all of the outputs should be equal to $\frac{1}{n}$. Numerically, this may not occur when $c$ has large magnitude. If $c$ is very negative, then $\exp(c)$ will underflow. This means the denominator of the softmax will become 0, so the final result is undefined. When $c$ is very large and positive, $\exp(c)$ will overflow, again resulting in the expression as a whole being undefined. Both of these difficulties can be resolved by instead evaluating $\text{softmax}(\boldsymbol{z})$ where $\boldsymbol{z} = \boldsymbol{x} - \max_i x_i$. Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector. Subtracting $\max_i x_i$ results in the largest argument to exp being 0, which rules out the possibility of overflow. Likewise, at least one term in the denominator has a value of 1, which rules out the possibility of underflow in the denominator leading to a division by zero.

There is still one small problem. Underflow in the numerator can still cause the expression as a whole to evaluate to zero. This means that if we implement $\log \text{softmax}(\boldsymbol{x})$ by first running the softmax subroutine then passing the result to the log function, we could erroneously obtain $-\infty$. Instead, we must implement a separate function that calculates $\log \text{softmax}$ in a numerically stable way. The $\log \text{softmax}$ function can be stabilized using the same trick as we used to stabilize the softmax function.

For the most part, we do not explicitly detail all of the numerical considerations involved in implementing the various algorithms described in this book. Implementors should keep numerical issues in mind when developing implementations. Many numerical issues can be avoided by using Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012), a software package that automatically detects and stabilizes many common numerically unstable expressions that arise in the context of deep learning.

## 4.2 Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are per-

turbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ has an eigenvalue decomposition, its *condition number* is

$$\max_{i,j} |\frac{\lambda_i}{\lambda_j}|,$$

i.e. the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input.

Note that this is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion. Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself. With iterative algorithms such as solving a linear system (or the worked-out example of linear least square by gradient descent, Section 4.5) ill-conditioning (in that case of the linear system matrix) yields very slow convergence of the iterative algorithm, i.e., more iterations are needed to achieve some given degree of approximation to the final solution.

## 4.3 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. We usually phrase most optimization problems in terms of minimizing $f(\boldsymbol{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\boldsymbol{x})$.

The function we want to minimize or maximize is called the *objective function* or *criterion*. When we are minimizing it, we may also call it the *cost function*, *loss function*, or *error function*. In this book, we use these terms interchangeably, though some machine learning publications assign special meaning to some of these terms.

We often denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $x^* = \arg\min f(\boldsymbol{x})$.

We assume the reader is already familiar with calculus, but provide a brief review of how calculus concepts relate to optimization here.

Suppose we have a function $y = f(x)$, where both $x$ and $y$ are real numbers. The *derivative* of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point $x$. In other words, it specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

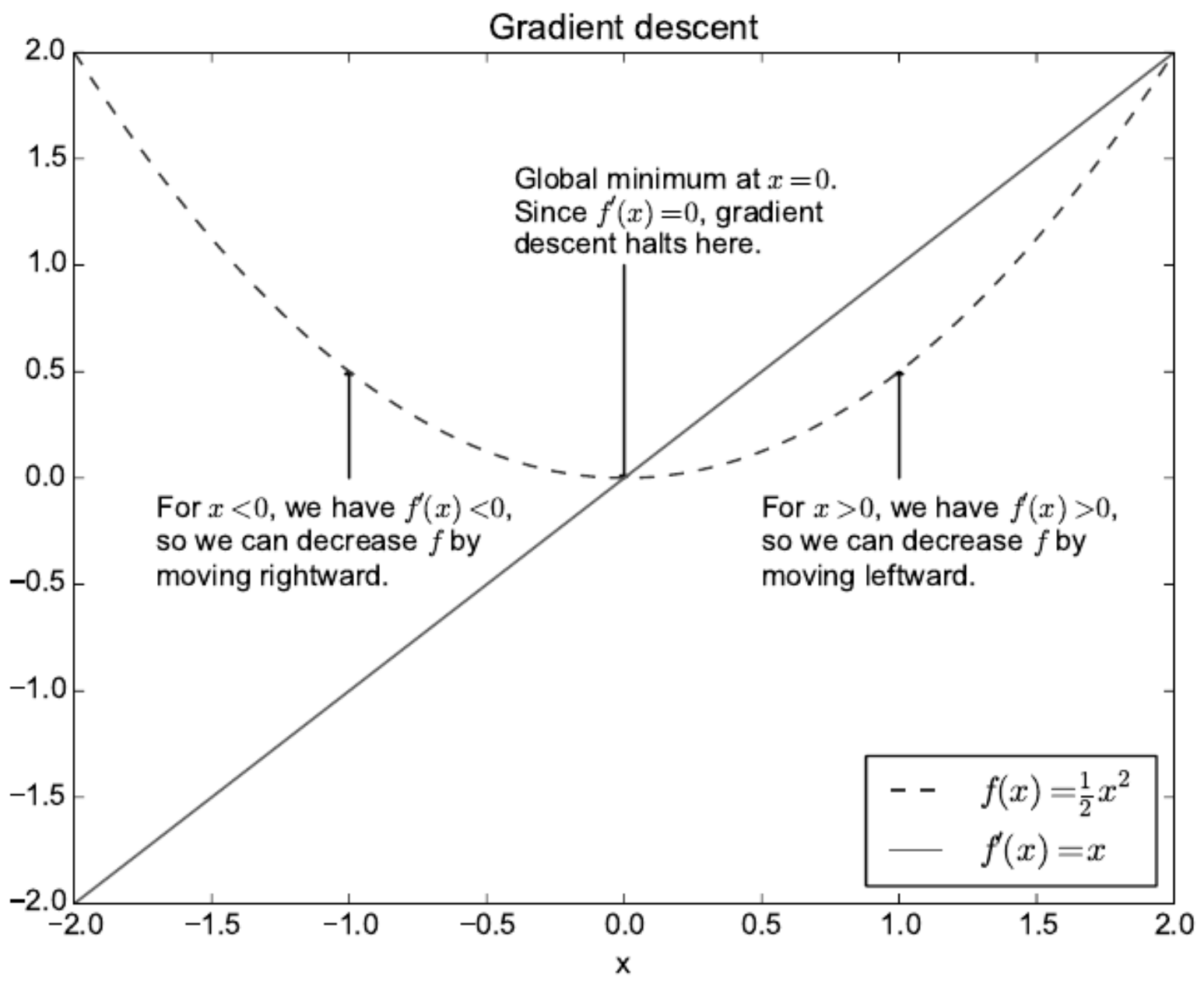


Figure 4.1: An illustration of how the derivatives of a function can be used to follow the function downhill to a minimum. This technique is called *gradient descent*.

The derivative is therefore useful for minimizing a function because it tells us how to change $x$ in order to make a small improvement in $y$. For example, we know that $f(x - \epsilon \operatorname{sign}(f'(x)))$ is less than $f(x)$ for small enough $\epsilon$. We can thus reduce $f(x)$ by moving $x$ in small steps with opposite sign of the the derivative. This technique is called gradient descent (Cauchy, 1847a). See Fig. 4.1 for an example of this technique.

When $f'(x) = 0$, the derivative provides no information about which direction to move. Points where $f'(x) = 0$ are known as *critical points* or *stationary points*. A *local minimum* is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A *local maximum* is a point where $f(x)$ is higher than at all neighboring points, so it is not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as *saddle points*. See Fig. 4.2 for examples of each type of critical point.

A point that obtains the absolute lowest value of $f(x)$ is a *global minimum*. It

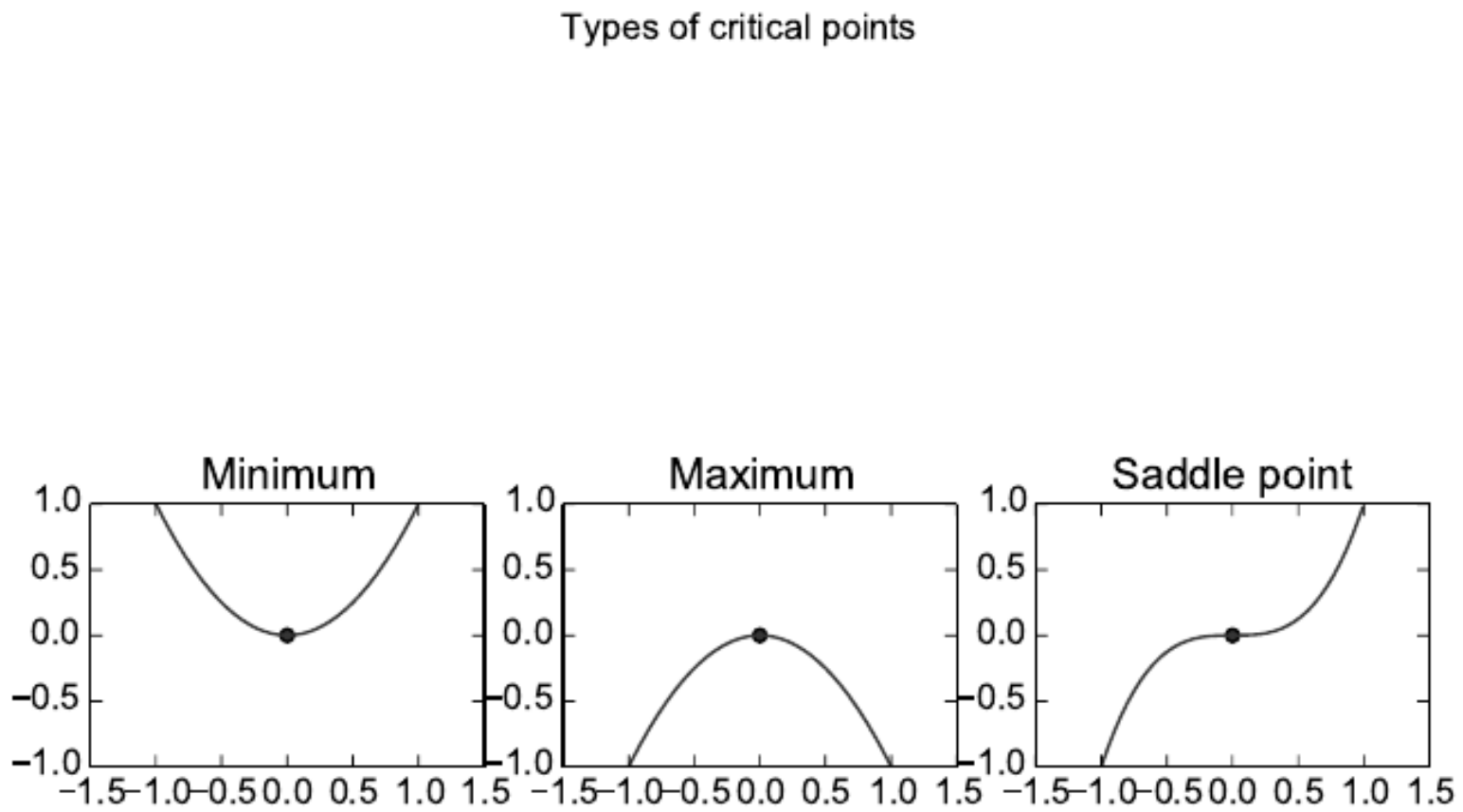


Figure 4.2: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors that are both higher and lower than the point itself. The situation in higher dimension is qualitatively different, especially for saddle points: see Figures 4.4 and 4.5.

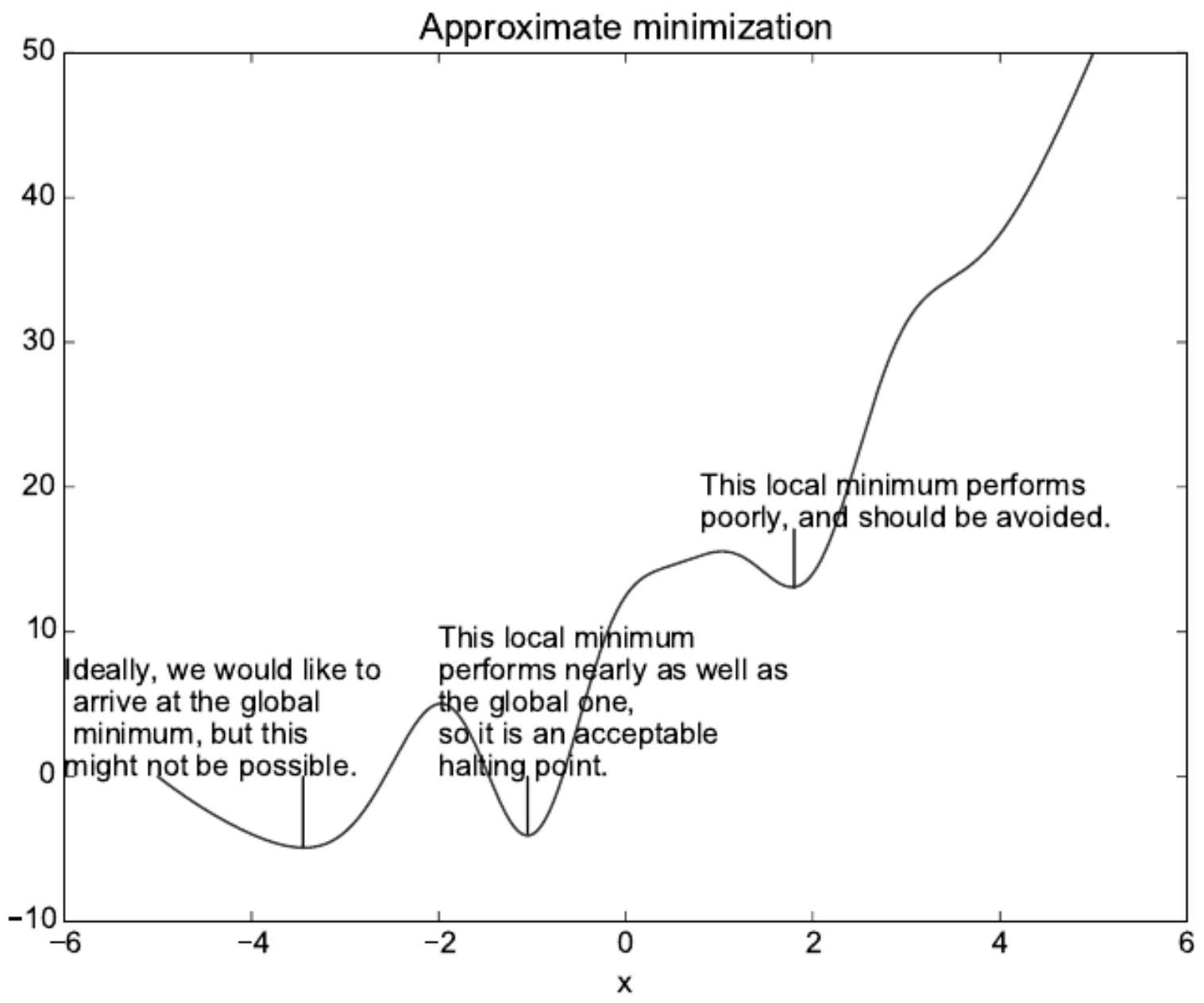


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

is possible for there to be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of deep learning, we optimize functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions. All of this makes optimization very difficult, especially when the input to the function is multidimensional. We therefore usually settle for finding a value of $f$ that is very low, but not necessarily minimal in any formal sense. See Fig. 4.3 for an example.

We often minimize functions that have multiple inputs: $f : \mathbb{R}^n \to \mathbb{R}$. Note that for the concept of "minimization" to make sense, there must still be only one output.

For these functions, we must make use of the concept of *partial derivatives*. The partial derivative $\frac{\partial}{\partial x} f(\boldsymbol{x})$ measures how $f$ changes as only the variable $x_i$

increases at point $\boldsymbol{x}$. The *gradient* generalizes the notion of derivative to the case where the derivative is with respect to a vector: $f$ is the vector containing all of the partial derivatives, denoted $\nabla \ f(\boldsymbol{x})$. Element $i$ of the gradient is the partial derivative of $f$ with respect to $x_i$. In multiple dimensions, critical points are points where every element of the gradient is equal to zero.

The *directional derivative* in direction $\boldsymbol{u}$ (a unit vector) is the slope of the function $f$ in direction $u$. In other words, the derivative of the function $f(\boldsymbol{x}+\alpha\boldsymbol{u})$ with respect to $\alpha$, evaluated at $\alpha = 0$. Using the chain rule, we can see that this is $\boldsymbol{u}^\top \nabla \ f(\boldsymbol{x})$.

To minimize $f$, we would like to find the direction in which $f$ decreases the fastest. We can do this using the directional derivative:

$$\begin{aligned} & \min_{, \quad =1} \quad \boldsymbol{u}^\top \nabla \ f(\boldsymbol{x}) \\ = & \min_{, \quad =1} \quad ||\boldsymbol{u}||_2 ||\nabla \ f(\boldsymbol{x})||_2 \cos\theta \end{aligned}$$

where $\theta$ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $||\boldsymbol{u}||_2 = 1$ and ignoring factors that don't depend on $\boldsymbol{u}$, this simplifies to min $\cos\theta$. This is minimized when $\boldsymbol{u}$ points in the opposite direction as the gradient. In other words, the gradient points directly uphill, and the negative gradient points directly downhill. We can decrease $f$ by moving in the direction of the negative gradient. This is known as the *method of steepest descent* or *gradient descent.*

Steepest descent proposes a new point

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla \ f(\boldsymbol{x})$$

where $\epsilon$ is the size of the step. We can choose $\epsilon$ in several different ways. A popular approach is to set $\epsilon$ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(\boldsymbol{x} - \epsilon \nabla \ f(\boldsymbol{x}))$ for several values of $\epsilon$ and choose the one that results in the smallest objective function value. This last strategy is called a *line search.*

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm, and just jump directly to the critical point by solving the equation $\nabla \ f(\boldsymbol{x}) = 0$ for $\boldsymbol{x}$.

Sometimes we need to find all of the partial derivatives of all of the elements of a vector-valued function. The matrix containing all such partial derivatives is known as a *Jacobian matrix.* Specifically, if we have a function $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n\times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x} f(\boldsymbol{x})_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a *second derivative.* For example, for a function $f : \mathbb{R}^n \to \mathbb{R}$, the derivative with respect to $x_i$ of the derivative of $f$ with respect to $x_j$ is denoted as $\frac{\partial}{\partial x \ \partial x} f$.

In a single dimension, we can denote $\frac{d}{dx}f$ by $f''(x)$.

The second derivative tells us how the first derivative will change as we vary the input. This means it can be useful for determining whether a critical point is a local maximum, a local minimum, or saddle point. Recall that on a critical point, $f'(x) = 0$. When $f''(x) > 0$, this means that $f'(x)$ increases as we move to the right, and $f'(x)$ decreases as we move to the left. This means $f'(x - \epsilon) < 0$ and $f'(x + \epsilon) > 0$ for small enough $\epsilon$. In other words, as we move right, the slope begins to point uphill to the right, and as we move left, the slope begins to point uphill to the left. Thus, when $f'(x) = 0$ and $f''(x) > 0$, we can conclude that $x$ is a local minimum. Similarly, when $f'(x) = 0$ and $f''(x) < 0$, we can conclude that $x$ is a local maximum. This is known as the *second derivative test*. Unfortunately, when $f''(x) = 0$, the test is inconclusive. In this case $x$ may be a saddle point, or a part of a flat region.

In multiple dimensions, we need to examine all of the second derivatives of the function. These derivatives can be collected together into a matrix called the *Hessian matrix*. The Hessian matrix $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}).$$

Equivalently, the Hessian is the Jacobian of the gradient.

Anywhere that the second partial derivatives are continuous, the differential operators are commutative, i.e. their order can be swapped:

$$\frac{\partial^2}{\partial x_i \, \partial x_j} f(\boldsymbol{x}) = \frac{\partial^2}{\partial x_j \, \partial x_i} f(\boldsymbol{x}).$$

This implies that $H_{i,j} = H_{j,i}$, so the Hessian matrix is symmetric at such points. Most of the functions we encounter in the context of deep learning have a symmetric Hessian almost everywhere. Because the Hessian matrix is real and symmetric, we can decompose it into a set of real eigenvalues and an orthogonal basis of eigenvectors.

Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions. At a critical point, where $\nabla f(\boldsymbol{x}) = 0$, we can examine the eigenvalues of the Hessian to determine whether the critical point is a local maximum, local minimum, or saddle point. When the Hessian is positive definite[1], the point is a local minimum. This can be seen by observing that the directional second derivative in any direction must be positive, and making reference to the univariate second derivative test. Likewise, when the Hessian is negative definite[2], the point is a local maximum. In multiple dimensions, it is

all its eigenvalues are positive

all its eigenvalues are negative

actually possible to find positive evidence of saddle points in some cases. When at least one eigenvalue is positive and at least one eigenvalue is negative, we know that $\boldsymbol{x}$ is a local maximum on one cross section of $f$ but a local minimum on another cross section. See Fig. 4.4 for an example. Finally, the multidimensional second derivative test can be inconclusive, just like the univariate version. The test is inconclusive whenever all of the non-zero eigenvalues have the same sign, but at least one eigenvalue is zero. This is because the univariate second derivative test is inconclusive in the cross section corresponding to the zero eigenvalue.

The Hessian can also be useful for understanding the performance of gradient descent. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer. See Fig. 4.5 for an example.

This issue can be resolved by using information from the Hessian matrix to guide the search. The simplest method for doing so is known as *Newton's method*. Newton's method is based on using a second-order *Taylor series expansion* to approximate $f(\boldsymbol{x})$ near some point $\boldsymbol{x}_0$, ignoring derivatives of higher order:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + (\boldsymbol{x} - \boldsymbol{x}_0)^\top \nabla \quad f(\boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^\top H(f)(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0).$$

If we then solve for the critical point of this function, we obtain:

$$\boldsymbol{x}^* = \boldsymbol{x}_0 - H(f)(\boldsymbol{x}_0)^{-1} \nabla \quad f(\boldsymbol{x}_0).$$

When the function can be locally approximated as quadratic, iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. As discussed in Section 8.2.4, Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent can in principle escape a saddle point, although it may take a lot of time if the negative eigenvalues are very small in magnitude, producing a kind of plateau around the saddle point.

Optimization algorithms such as gradient descent that use only the gradient are called *first-order optimization algorithms*. Optimization algorithms such as Newton's method that also use the Hessian matrix are called *second-order optimization algorithms* (Nocedal and Wright, 2006).

The optimization algorithms employed in most contexts in this book are applicable to a wide variety of functions, but come with almost no guarantees. This

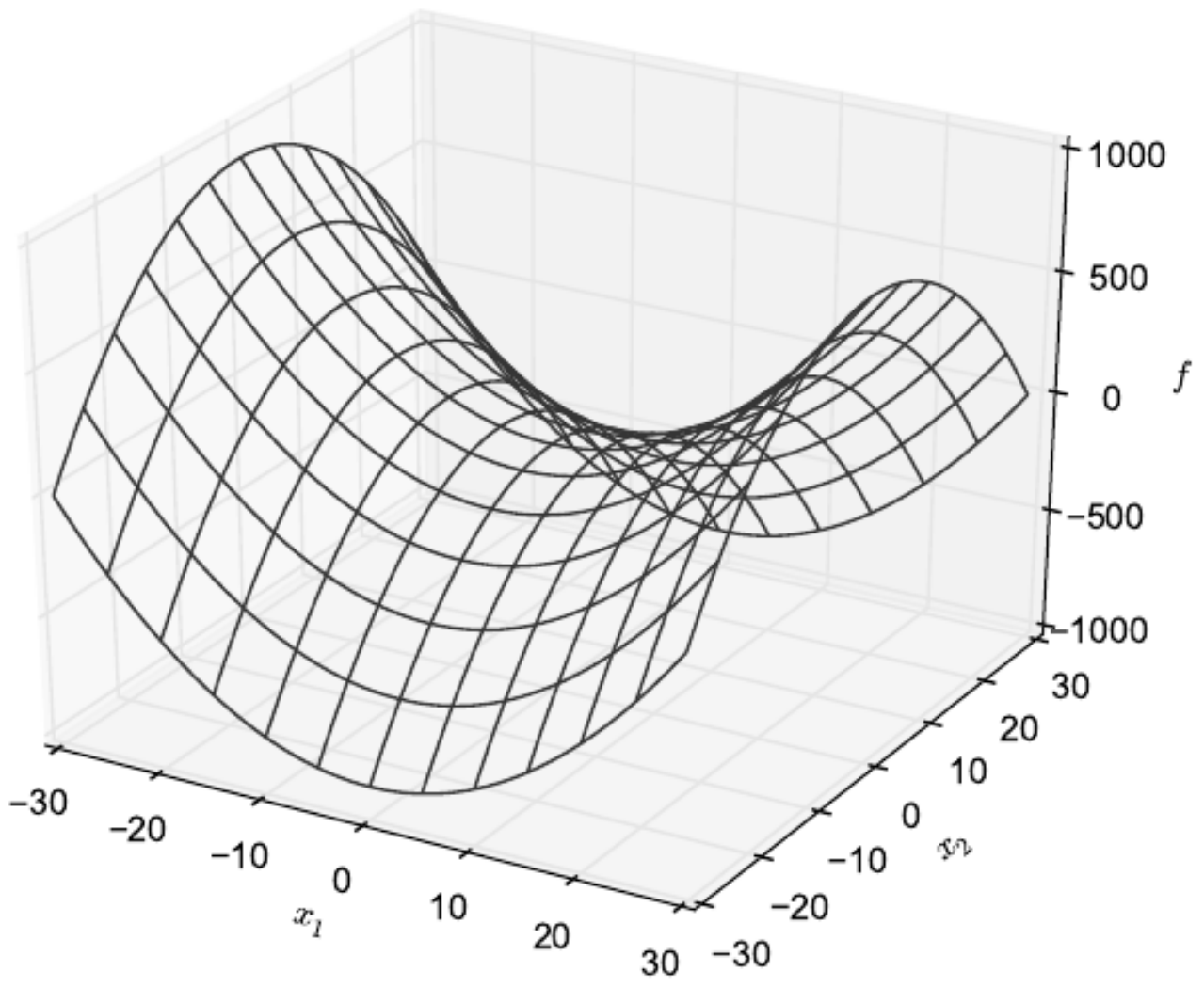


Figure 4.4: A saddle point containing both positive and negative curvature. The function in this example is $f(\boldsymbol{x}) = x \quad - x \quad$. Along the axis corresponding to $x$ , the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to $x$ , the function curves downward. This direction is an eigenvector of the Hessian with negative eigenvalue. The name "saddle point" derives from the saddle-like shape of this function. This is the quintessential example of a function with a saddle point. Note that in more than one dimension, it is not necessary to have an eigenvalue of 0 in order to get a saddle point: it is only necessary to have both positive and negative eigenvalues.

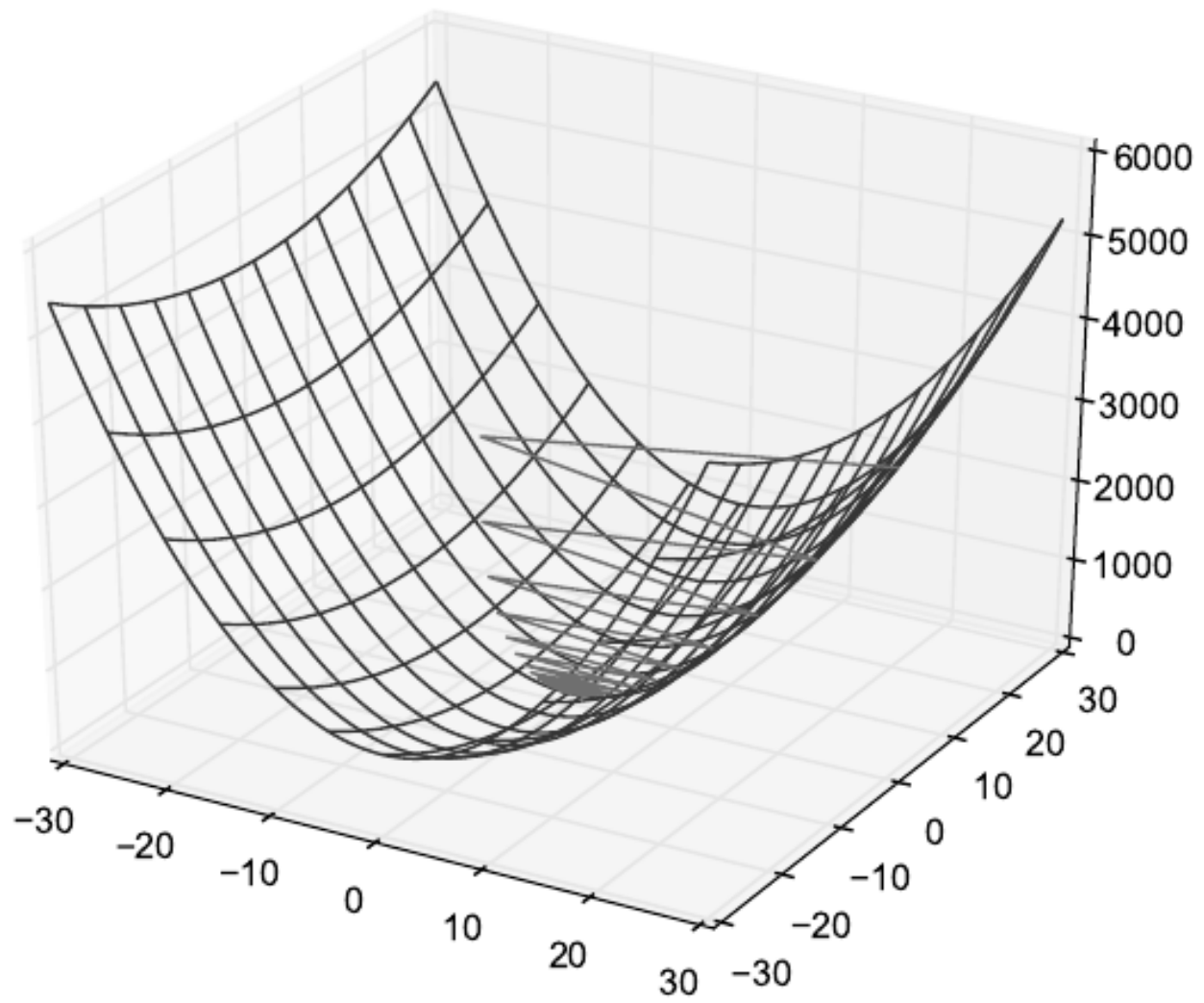


Figure 4.5: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent on a quadratic function whose Hessian matrix has condition number 5 (curvature is 5 times larger in one direction than in some other direction). The lines above the mesh indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature. Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context.

is because the family of functions used in deep learning is quite complicated. In many other fields, the dominant approach to optimization is to design optimization algorithms for a limited family of functions. Perhaps the most successful field of specialized optimization is *convex optimization*. Convex optimization algorithms are able to provide many more guarantees, but are applicable only to functions for which the Hessian is positive definite everywhere. Such functions are well-behaved because they lack saddle points and all of their local minima are necessarily global minima. However, most problems in deep learning are difficult to express in terms of convex optimization. Convex optimization is used only as a subroutine of some deep learning algorithms. Ideas from the analysis of convex optimization algorithms can be useful for proving the convergence of deep learning algorithms. However, in general, the importance of convex optimization is greatly diminished in the context of deep learning. For more information about convex optimization, see Boyd and Vandenberghe (2004) or Rockafellar (1997).

## 4.4 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(\boldsymbol{x})$ over all possible values of $\boldsymbol{x}$. Instead we may wish to find the maximal or minimal value of $f(\boldsymbol{x})$ for values of $\boldsymbol{x}$ in some set $\mathbb{S}$. This is known as *constrained optimization*. Points $\boldsymbol{x}$ that lie within the set $\mathbb{S}$ are called *feasible* points in constrained optimization terminology.

We often wish to find a solution that is small in some sense. A common approach in such situations is to impose a norm constraint, such as $||\boldsymbol{x}|| \leq 1$.

One simple approach to constrained optimization is simply to modify gradient descent taking the constraint into account. If we use a small constant step size $\epsilon$, we can make gradient descent steps, then project the result back into $S$. If we use a line search (see previous section), we can search only over step sizes $\epsilon$ that yield new $\boldsymbol{x}$ points that are feasible, or we can project each point on the line back into the constraint region. When possible, this method can be made more efficient by projecting the gradient into the tangent space of the feasible region before taking the step or beginning the line search (Rosen, 1960).

A more sophisticated approach is to design a different, unconstrained optimization problem whose solution can be converted into a solution to the original, constrained optimization problem. For example, if we want to minimize $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^2$ with $\boldsymbol{x}$ constrained to have exactly unit $L^2$ norm, we can instead minimize $g(\theta) = f([\cos\theta, \sin\theta]^T)$ with respect to $\theta$, then return $[\cos\theta, \sin\theta]$ as the solution to the original problem. This approach requires creativity; the transformation between optimization problems must be designed specifically for each case we encounter.

The *Karush–Kuhn–Tucker (KKT) approach* [3] provides a very general solution to constrained optimization. With the KKT approach, we introduce a new function called the *generalized Lagrangian* or *generalized Lagrange function.*

To define the Lagrangian, we first need to describe $S$ in terms of equations and inequalities. We want a description of $S$ in terms of $m$ functions $g$ and $n$ functions $h_j$ so that $S = \{\boldsymbol{x} \mid \forall i, g_i(\boldsymbol{x}) = 0 \text{ and } \forall j, h_j(\boldsymbol{x}) \leq 0\}$. The equations involving $g_i$ are called the *equality constraints* and the inequalities involving $h_j$ are called *inequality constraints.*

We introduce new variables $\lambda_i$ and $\alpha_j$ for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x}).$$

We can now solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. Observe that, so long as at least one feasible point exists and $f(\boldsymbol{x})$ is not permitted to have value $\infty$, then

$$\min \max_{,} \max_{\geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}).$$

has the same optimal objective function value and set of optimal points $\boldsymbol{x}$ as

$$\min_{\in S} f(\boldsymbol{x}).$$

This follows because any time the constraints are satisfied,

$$\max_{,} \max_{\geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}),$$

while any time a constraint is violated,

$$\max_{,} \max_{\geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty.$$

These properties guarantee that no infeasible point will ever be optimal, and that the optimum within the feasible points is unchanged.

To perform constrained maximization, we can construct the generalized Lagrange function of $-f(\boldsymbol{x})$, which leads to this optimization problem:

$$\min \max_{,} \max_{\geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x}).$$

The KKT approach generalizes the method of which only allows equality constraints

We may also convert this to a problem with maximization in the outer loop:

$$\max \min_{, \geq 0} f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) - \sum_j \alpha_j h_j(\boldsymbol{x}).$$

Note that the sign of the term for the equality constraints does not matter; we may define it with addition or subtraction as we wish, because the optimization is free to choose any sign for each $\lambda_i$.

The inequality constraints are particularly interesting. We say that a constraint $h_i(\boldsymbol{x})$ is *active* if $h_i(\boldsymbol{x}^*) = 0$. If a constraint is not active, then the solution to the problem is the same whether or not that constraint exists. Because an inactive $h_i$ has negative value, then the solution to $\min \max \max_{, \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ will have $\alpha_i = 0$. We can thus observe that at the solution, $\boldsymbol{\alpha}\boldsymbol{h}(\boldsymbol{x}) = \ .$ In other words, for all $i$, we know that at least one of the constraints $\alpha_i \geq 0$ and $h_i(\boldsymbol{x}) \leq 0$ must be active at the solution. To gain some intuition for this idea, we can say that either the solution is on the boundary imposed by the inequality and we must use its KKT multiplier to influence the solution to $\boldsymbol{x}$, or the inequality has no influence on the solution and we represent this by zeroing out its KKT multiplier.

The properties that the gradient of the generalized Lagrangian is zero, all constraints on both $\boldsymbol{x}$ and the KKT multipliers are satisfied, and $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) =$ are called the *Karush-Kuhn-Tucker* (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951). Together, these properties describe the optimal points of constrained optimization problems.

In the case where there are no inequality constraints, the KKT approach simplifies to the method of Lagrange multipliers. For more information about the KKT approach, see Nocedal and Wright (2006).

## 4.5 Example: Linear Least Squares

Suppose we want to find the value of $\boldsymbol{x}$ that minimizes

$$f(\boldsymbol{x}) = \frac{1}{2}||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2.$$

There are specialized linear algebra algorithms that can solve this problem efficiently. However, we can also explore how to solve it using gradient-based optimization as a simple example of how these techniques work.

First, we need to obtain the gradient:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}.$$

We can then follow this gradient downhill, taking small steps. See Algorithm 4.1 for details.

An algorithm to minimize $f(\boldsymbol{x}) = \frac{1}{2}||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2$ with respect to $\boldsymbol{x}$ using gradient descent.

Set $\epsilon$, the step size, and $\delta$, the tolerance, to small, positive numbers.

$||\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b}||_2 > \delta$

$\boldsymbol{x} \leftarrow \boldsymbol{x} - \epsilon \left(\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b}\right)$

One can also solve this problem using Newton's method. In this case, because the true function is quadratic, the quadratic approximation employed by Newton's method is exact, and the algorithm converges to the global minimum in a single step.

Now suppose we wish to minimize the same function, but subject to the constraint $\boldsymbol{x}^\top \boldsymbol{x} \leq 1$. To do so, we introduce the Lagrangian

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda \left(\boldsymbol{x}^\top \boldsymbol{x} - 1\right).$$

We can now solve the problem

$$\min \max_{\lambda, \lambda \geq 0} L(\boldsymbol{x}, \lambda).$$

The solution to the unconstrained least squares problem is given by $\boldsymbol{x} = \boldsymbol{A}^+ \boldsymbol{b}$. If this point is feasible, then it is the solution to the constrained problem. Otherwise, we must find a solution where the constraint is active. By differentiating the Lagrangian with respect to $\boldsymbol{x}$, we obtain the equation

$$\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b} + 2\lambda \boldsymbol{x} = 0.$$

This tells us that the solution will take the form

$$\boldsymbol{x} = (\boldsymbol{A}^\top \boldsymbol{A} + 2\lambda \boldsymbol{I})^{-1} \boldsymbol{A}^\top \boldsymbol{b}.$$

The magnitude of $\lambda$ must be chosen such that the result obeys the constraint. We can find this value by performing gradient ascent on $\lambda$. To do so, observe

$$\frac{\partial}{\partial \lambda} L(\boldsymbol{x}, \lambda) = \boldsymbol{x}^\top \boldsymbol{x} - 1.$$

When the norm of $\boldsymbol{x}$ exceeds 1, this derivative is positive, so to ascend the gradient and increase the Lagrangian with respect to $\lambda$, we increase $\lambda$. This will in turn shrink the optimal $\boldsymbol{x}$. The process continues until $\boldsymbol{x}$ has the correct norm and the derivative on $\lambda$ is 0.

# Chapter 5

# Machine Learning Basics

Deep learning is a specific kind of machine learning. In order to understand deep learning well, one must have a solid understanding of the basic principles of machine learning. This chapter provides a brief course in the most important general principles that will be applied throughout the rest of the book. Novice readers or those that want a wider perspective are encouraged to consider machine learning textbooks with a more comprehensive coverage of the fundamentals, such as Murphy (2012) or Bishop (2006). If you are already familiar with machine learning basics, feel free to skip ahead to Section 5.12. That section covers some perspectives on traditional machine learning techniques that have strongly influenced the development of deep learning algorithms.

## 5.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning? A popular definition of learning in the context of computer programs is "A computer program is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$" (Mitchell, 1997). One can imagine a very wide variety of experiences $E$, tasks $T$, and performance measures $P$, and we do not make any attempt in this book to provide a formal definition of what may be used for each of these entities. Instead, the following sections provide intuitive descriptions and examples of the different kinds of tasks, performance measures and experiences that can be used to construct machine learning algorithms.

### 5.1.1 The Task, $T$

Machine learning is mostly interesting because of the tasks we can accomplish with it. From an engineering point of view, machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because understanding it allows us to understand the principles that underlie intelligent behavior, and intelligent behavior is defined as being able to accomplish certain tasks.

In this relatively formal definition of the word "task," the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.

Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks include the following:

- *Classification*: In this type of task, the computer program is asked to specify which of $k$ categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$ which may then be applied to any input. Here the output of $f(\boldsymbol{x})$ can be interpreted as an estimate of the category that $\boldsymbol{x}$ belongs to. There are other variants of the classification task, for example, where $f$ outputs a probability distribution over classes. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. For example, the Willow Garage PR2 robot is able to act as a waiter that can recognize different kinds of drinks and deliver them to people on command (Goodfellow *et al.*, 2010). Modern object recognition is best accomplished with deep learning (Krizhevsky *et al.*, 2012a; Ioffe and Szegedy, 2015). Object recognition is the same basic technology that allows computers to recognize faces (Taigman *et al.*, 2014), which can be used to automatically tag people in photo collections and allow computers to interact more naturally with their users.

- *Classification with missing inputs* : Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. In order to solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output. When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set* of functions. Each function corresponds

to classifying $\boldsymbol{x}$ with a different subset of its inputs missing. This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive. One way to efficiently define such a large set of functions is to learn a probability distribution over all of the relevant variables, then solve the classification task by marginalizing out the missing variables. With $n$ input variables, we can now obtain all $2^n$ different classification functions needed for each possible set of missing inputs, but we only need to learn a single function describing the joint probability distribution. See Goodfellow *et al.* (2013b) for an example of a deep probabilistic model applied to such a task in this way. Many of the other tasks described in this section can also be generalized to work with missing inputs; classification with missing inputs is just one example of what machine learning can do.

- *Regression* : In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premia), or the prediction of future prices of securities. These kinds of predictions are also used for algorithmic trading.

- *Transcription* : In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters (e.g. in ASCII or Unicode format). Google Street View uses deep learning to process address numbers in this way Goodfellow *et al.* (2014d). Another example is speech recognition, where the computer program is provided an audio waveform and emits a sequence of characters or word ID codes describing the words that were spoken in the audio recording. Deep learning is a crucial component of modern speech recognition systems used at major companies including Microsoft, IBM and Google (Hinton *et al.*, 2012b).

- *Translation*: In a translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. This is commonly applied to natural languages, such as to translate from English to French. Deep

learning has recently begun to have an important impact on this kind of task (Sutskever *et al.*, 2014a; Bahdanau *et al.*, 2014).

- *Structured output* tasks involve any task where the output is a vector containing important relationships between the different elements. This is a broad category, and includes the transcription and translation tasks described above, but also many other tasks. One example is parsing—mapping a natural language sentence into a tree that describes its grammatical structure and the relative role of its constituents. See Collobert (2011) for an example of deep learning applied to a parsing task. Another example is pixel-wise segmentation of images, where the computer program assigns every pixel in an image to a specific category. For example, deep learning can be used to annotate the locations of roads in aerial photographs (Mnih and Hinton, 2010). The output need not have its form mirror the structure of the input as closely as in these annotation-style tasks. For example, in an image captioning, the computer program observes an image and outputs a natural language sentence describing the image (Kiros *et al.*, 2014a,b; Mao *et al.*, 2015; Vinyals *et al.*, 2015; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015a). These tasks are called structured output tasks because the program must output several values that are all tightly inter-related. For example, the words produced by an image captioning program must form a valid sentence.

- *Anomaly detection*: In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modeling your purchasing habits, a credit card company can detect misuse of your cards. If a thief steals your credit card or credit card information, the thief's purchases will often come from a different probability distribution over purchase types than your own. The credit card company can prevent fraud by placing a hold on an account as soon as that card has been used for an uncharacteristic purchase.

- *Synthesis and sampling*: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. This can be useful for media applications where it can be expensive or boring for an artist to generate large volumes of content by hand. For example, video games can automatically generate textures for large objects or landscapes, rather than requiring an artist to manually label each pixel (Luo *et al.*, 2013). In some cases, we want the sampling or synthesis procedure to generate some specific kind of output given the input. For example, in a speech synthesis task, we provide a written sentence and ask the program

to emit an audio waveform containing a spoken version of that sentence. This is a kind of structured output task, but with the added qualification that there is no single correct output for each input, and we explicitly desire a large amount of variation in the output, in order for the output to seem more natural and realistic.

- *Imputation of missing values*: In this type of task, the machine learning algorithm is given a new example $\boldsymbol{x} \in \mathbb{R}^n$, but with some entries $x_i$ of $\boldsymbol{x}$ missing. The algorithm must provide a prediction of the values of the missing entries.

- *Denoising*: In this type of task, the machine learning algorithm is given in input a *corrupted example* $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ obtained by an unknown corruption process from a *clean example* $\boldsymbol{x} \in \mathbb{R}^n$. The learner must predict the clean example $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, or more generally predict the conditional probability distribution $P(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$.

- *Density or probability function estimation*: In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{\text{model}}(\boldsymbol{x})$ can be interpreted as a probability density function (if $\mathbf{x}$ is continuous) or a probability function (if $\mathbf{x}$ is discrete) on the space that the examples were drawn from. To do such a task well (we will specify exactly what that means when we discuss performance measures $P$), the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. Most of the tasks described above require that the learning algorithm has at least implicitly captured the structure of the probability distribution. Density estimation allows us to explicitly capture that distribution. In principle, we can then perform computations on that distribution in order to solve the other tasks as well. For example, if we have performed density estimation to obtain a probability distribution $p(\boldsymbol{x})$, we can use that distribution to solve the missing value imputation task. If a value $x_i$ is missing and the other values $\boldsymbol{x}_{-i}$ are given, then we know the distribution over it is given by $p(x_i \mid \boldsymbol{x}_{-i})$ . In practice, density estimation does not always allow us to solve all of these related tasks, because in many cases the required operations on $p(\boldsymbol{x})$ are computationally intractable.

Of course, many other tasks and types of tasks are possible. The types of tasks we listed here are only intended to provide examples of what machine learning can do, not to define a rigid taxonomy of tasks.

### 5.1.2 The Performance Measure, $P$

In order to evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure $P$ is specific to the task $T$ being carried out by the system.

For tasks such as classification, classification with missing inputs, and transcription, we often measure the *accuracy* of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the *error rate*, the proportion of examples for which the model produces an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not. For tasks such as density estimation, we can measure the probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a *test set* of data that is separate from the data used for training the machine learning system.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

In some cases, this is because it is difficult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more fine-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. For example, this arises frequently in the context of density estimation. Many of the best probabilistic models represent probability distributions only implicitly. Computing the actual probability value assigned to a specific point in space is intractable. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

### 5.1.3 The Experience, $E$

Machine learning algorithms can be broadly categorized as *unsupervised* or *supervised* by what kind of experience they are allowed to have during the learning

process.

Most of the learning algorithms in this book can be understood as being allowed to experience an entire *dataset*. A dataset is a collection of many objects called *examples*, with each example containing many *features* that have been objectively measured. Sometimes we will also call examples *data points*.

One of the oldest datasets studied by statisticians and machine learning researchers is the Iris dataset (Fisher, 1936). It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each of the parts of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

*Unsupervised learning algorithms* experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like dividing the dataset into clusters of similar examples.

*Supervised learning algorithms* experience a dataset containing features, but each example is also associated with a *label* or *target*. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector $\mathbf{x}$, and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, e.g. estimating $p(\mathbf{y} \mid \mathbf{x})$. The term *supervised learning* originates from the view of the target $\mathbf{y}$ being provided by an instructor or teacher that shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Unsupervised learning and supervised learning are not formally defined terms. The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \Pi_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1}).$$

This decomposition means that we can solve the ostensibly unsupervised problem of modeling $p(\mathbf{x})$ by splitting it into $n$ supervised learning problems. Alternatively,

we can solve the supervised learning problem of learning $p(y \mid \mathbf{x})$ by using traditional unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$ and inferring

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}.$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help to roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

Some machine learning algorithms do not just experience a fixed dataset. For example, *reinforcement learning* algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. Such algorithms are beyond the scope of this book. Please see Sutton and Barto (1998); Bertsekas and Tsitsiklis (1996) for a deeper treatment of reinforcement learning.

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples. Each example is a collection of observations called *features* collected from a different time or place. If we wish to make a system for recognizing objects from photographs, we might use a machine learning algorithm where each example is a photograph, and the features within the example are the brightness values of each of the pixels within the photograph. If we wish to perform speech recognition, we might collect a dataset where each example is a recording of a person saying a word or sentence, and each of the features is the amplitude of the sound wave at a particular moment in time.

One common way of describing a dataset is with a *design matrix*. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\boldsymbol{X} \in \mathbb{R}^{150 \times 4}$, where $X_{i,1}$ is the sepal length of plant $i$, $X_{i,2}$ is the sepal width of plant $i$, etc. We will describe most of the learning algorithms in this book in terms of how they operate on design matrix datasets.

Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible. For example, if you have a collection of photographs with different widths and heights, then different photographs will contain different numbers of pixels, so not all of the photographs may be described with the same length of vector. Different sections of this book describe how to handle different types of heterogeneous data. In cases like these, rather than describing the dataset

as a matrix with $m$ rows, we will describe it as a set containing $m$ elements, e.g. $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$. This notation does not imply that any two example vectors $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(j)}$ have the same size.

In the case of supervised learning, the example contains a label or target as well as a collection of features. For example, if we want to use a learning algorithm to perform object recognition from photographs, we need to specify which object appears in each of the photos. We might do this with a numeric code, with 0 signifying a person, 1 signifying a car, 2 signifying a cat, etc. Often when working with a dataset containing a design matrix of feature observations $\boldsymbol{X}$, we also provide a vector of labels $\boldsymbol{y}$, with $y_i$ providing the label for example $i$.

Of course, sometimes the label may be more than just a single number. For example, if we want to train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Just as there is no formal definition of supervised and unsupervised learning, there is no rigid taxonomy of datasets or experiences. The structures described here cover most cases, but it is always possible to design new ones for new applications.

## 5.2 Example: Linear Regression

In the previous section, we saw that a machine learning algorithm is an algorithm that is capable of improving a computer program's performance at some task via experience. Now it is time to define some specific machine learning algorithms.

Let's begin with an example of a simple machine learning algorithm: *linear regression*. In this section, we will only describe what the linear regression algorithm does. We wait until later sections of this chapter to justify the algorithm and show more formally that it actually works.

As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\boldsymbol{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the value that our model predicts $y$ should take on. We define the output to be

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of *parameters*.

Parameters are values that control the behavior of the system. In this case, $w_i$ is the coefficient that we multiply by feature $x_i$ before summing up the contributions from all the features. We can think of $\boldsymbol{w}$ as a set of *weights* that determine how each feature affects the prediction. If a feature $x_i$ receives a positive weight $w_i$, then increasing the value of that feature increases the value of our prediction

$\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction.

We thus have a definition of our task $T$: to predict $y$ from $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. Next we need a definition of our performance measure, $P$.

Let's suppose that we have a design matrix of $m$ example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of $y$ for each of these examples. Because this dataset will only be used for evaluation, we call it the *test set*. Let's refer to the design matrix of inputs as $\boldsymbol{X}^{(\text{test})}$ and the vector of regression targets as $\boldsymbol{y}^{(\text{test})}$.

One way of measuring the performance of the model is to compute the *mean squared error* of the model on the test set. If $\hat{\boldsymbol{y}}^{(\text{test})}$ is the predictions of the model on the test set, then the mean squared error is given by

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i (\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})})_i^2.$$

Intuitively, one can see that this error measure decreases to 0 when $\hat{\boldsymbol{y}}^{(\text{test})} = \boldsymbol{y}^{(\text{test})}$. We can also see that

$$\text{MSE}_{\text{test}} = \frac{1}{m} ||\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})}||_2^2,$$

so the error increases whenever the Euclidean distance between the predictions and the targets increases.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights $\boldsymbol{w}$ in a way that reduces $\text{MSE}_{\text{test}}$ when the algorithm is allowed to gain experience by observing a training set $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$. One intuitive way of doing this (which we will justify later) is just to minimize the mean squared error on the training set, $\text{MSE}_{\text{train}}$.

To minimize $\text{MSE}_{\text{train}}$, we can simply solve for where its gradient is $\mathbf{0}$:

$$\nabla \text{MSE}_{\text{train}} = 0$$

$$\Rightarrow \nabla \frac{1}{m} ||\hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0$$

$$\Rightarrow \frac{1}{m} \nabla ||\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0$$

$$\Rightarrow \nabla (\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})})^\top (\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}) = 0$$

$$\Rightarrow \nabla (\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})}) = 0$$

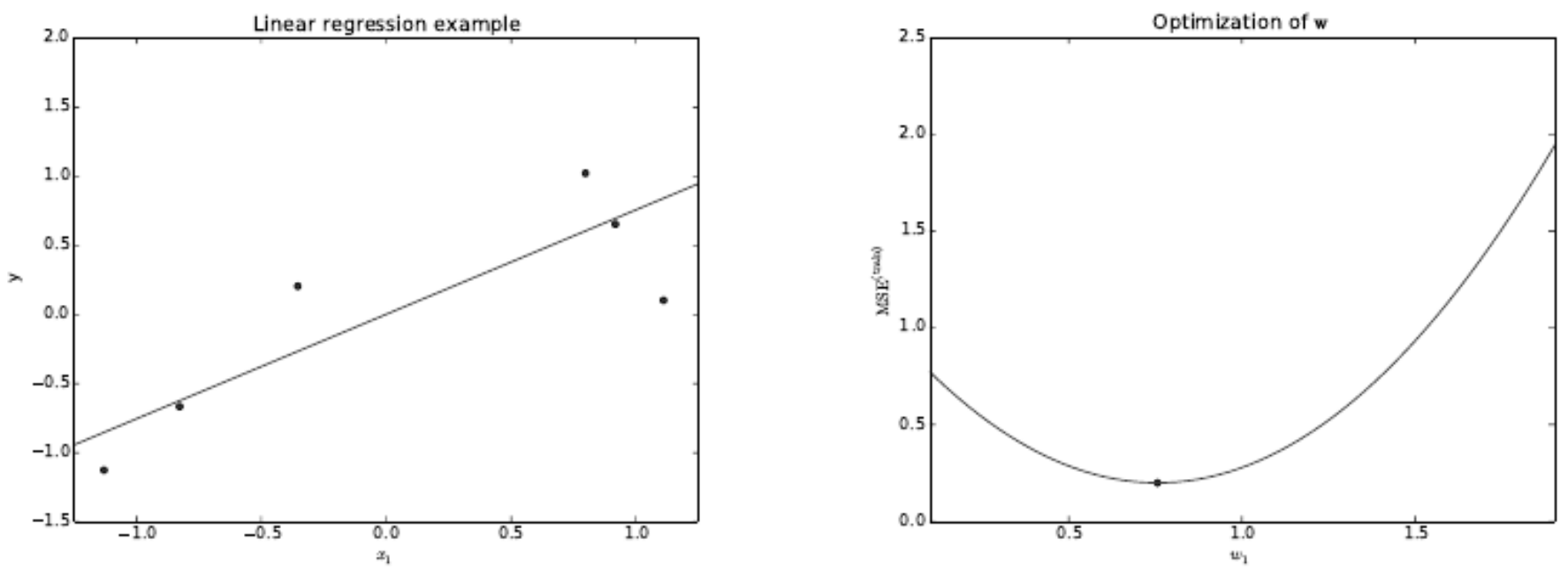


Figure 5.1: Consider this example linear regression problem, with a training set consisting of 5 data points, each containing one feature. This means that the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, $w$ . *(Left)* Observe that linear regression learns to set $w$ such that the line $y = w\,x$ comes as close as possible to passing through all the training points. *(Right)* The plotted point indicates the value of $w$ found by the normal equations, which we can see minimizes the mean squared error on the training set.

$$\Rightarrow 2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - 2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} = 0$$
$$\Rightarrow \boldsymbol{w} = (\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})})^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} \tag{5.1}$$

The system of equations defined by Eq. 5.1 is known as the *normal equations*. Solving these equations constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see Fig. 5.1.

It's worth noting that the term *linear regression* is often used to refer to a slightly more sophisticated model with one additional parameter—an intercept term $b$. In this model

$$\hat{y} = \boldsymbol{w}^\top\boldsymbol{x} + b$$

so the mapping from parameters to predictions is still a linear function but the mapping from features to predictions is now an affine function. This extension to affine functions means that the plot of the model's predictions still looks like a line, but it need not pass through the origin. We will frequently use the term "linear" when referring to affine functions throughout this book.

Linear regression is of course an extremely simple and limited learning algorithm, but it provides an example of how a learning algorithm can work. In the subsequent sections we will describe some of the basic principles underlying learning algorithm design and demonstrate how these principles can be used to build more complicated learning algorithms.

## 5.3 Generalization, Capacity, Overfitting and Underfitting

The central challenge in machine learning is that we must perform well on *new, previously unseen* inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called *generalization*.

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the *training error*, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the *generalization error* to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a *test set* of examples that were collected separate from the training set.

In our linear regression example, we trained the model by minimizing the training error,

$$\frac{1}{m^{(\text{train})}}||\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2,$$

but we actually care about the test error, $\frac{1}{m}||\boldsymbol{X}^{(\text{test})}\boldsymbol{w} - \boldsymbol{y}^{(\text{test})}||_2^2$.

How can we affect performance on the test set when we only get to observe the training set? The field of *statistical learning theory* provides some answers. If the training and the test set are collected arbitrarily, there is indeed little we can do. If we are allowed to make some assumptions about how the training and test set are collected, then we can make some progress.

We typically make a set of assumptions known collectively as the *i.i.d. assumptions*. These assumptions are that the examples in each dataset are *independent* from each other, and that the train set and test set are *identically distributed*, drawn from the same probability distribution as each other. We call that shared underlying distribution the *data generating distribution*, or *data generating process* (which is particularly relevant if the examples are not independent). This probabilistic framework allows us to mathematically study the relationship between training error and test error.

One immediate connection we can observe between the training and test error is that for a randomly selected model, the two have the same expected value. Suppose we have a probability distribution $p(\boldsymbol{x}, y)$ and we sample from it repeatedly to generate the train set and the test set. For some fixed value $\boldsymbol{w}$, then the expected training set error under this sampling process is exactly the same

as the expected test set error under this sampling process. The only difference between the two conditions is the name we assign to the dataset we sample. From this observation, we can see that it is natural for there to be some relationship between training and test error under these assumptions.

Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: *underfitting* and *overfitting*. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its *capacity*. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit, i.e., memorize properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its *hypothesis space*, the set of functions that the learning algorithm is allowed to choose as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

A polynomial of degree one gives us the linear regression model with which we are already familiar, with prediction

$$\hat{y} = b + wx.$$

By introducing $x^2$ as another feature provided to the linear regression model, we can learn a model that is quadratic as a function of $x$:

$$\hat{y} = b + w_1 x + w_2 x^2 .$$

Note that this is still a linear function of the parameters, so we can still use the normal equations to train the model in closed form. We can continue to add more

powers of $x$ as additional features, for example to obtain a polynomial of degree 9:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i.$$

Machine learning algorithms will generally perform best when their capacity is appropriate in regard to the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Model with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may overfit.

Fig. 5.2 shows this principle in action. We compare a linear, quadratic and degree-9 predictor attempting to fit a problem where the true underlying function is quadratic. The linear function is unable to capture the curvature in the true underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass exactly through the training points, because we have more parameters than training examples. We have little chance of choosing a solution that generalizes well when so many wildly different solutions exist. In this example, the quadratic model is perfectly matched the true structure of the task so it generalizes well to new data.

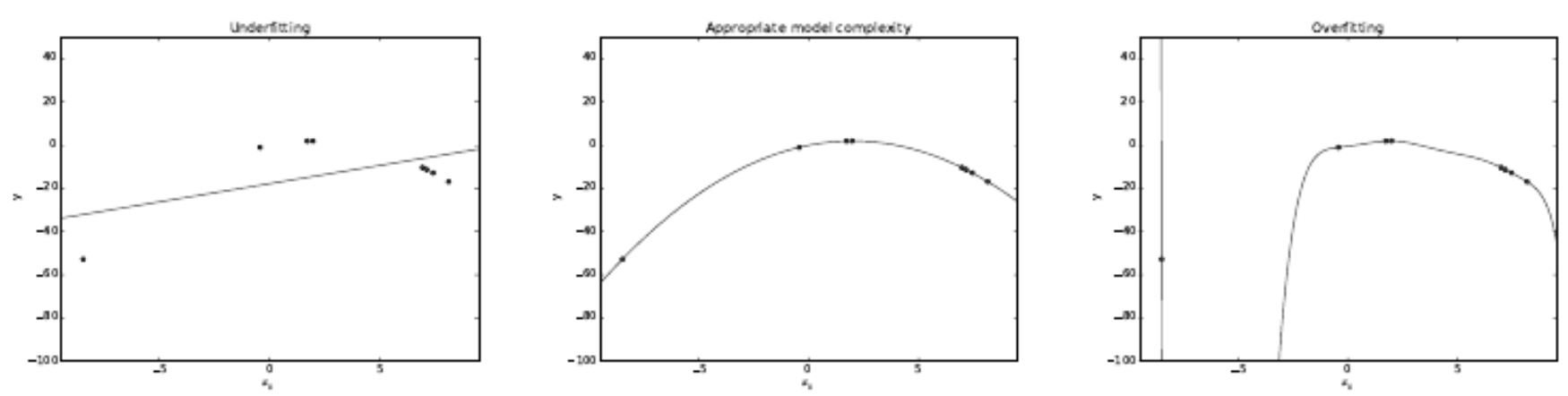


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling $x$ values and choosing $y$ deterministically by evaluating a quadratic function. (*Left*) A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (*Center*) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. (*Right*) A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudo-inverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

Here we have only described changing a model's capacity by changing the number of input features it has (and simultaneously adding new parameters associated with those features). There are many other ways of controlling the capacity of a machine learning algorithm, which we will explore in the sections ahead.

Our modern ideas about improving the generalization of machine learning models are refinements of thought dating back to philosophers at least as early as Ptolemy. Many early scholars invoke a principle of parsimony that is now most widely known as *Occam's razor* (c. 1287-1347). This principle states that among competing hypotheses that explain known observations equally well, one should choose the "simplest" one. This idea was formalized and made more precise in the 20th century by the founders of statistical learning theory (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). Statistical learning theory provides various means of quantifying model capacity and showing that the discrepancy between training error and generalization error is bounded by a quantity that grows with the ratio of capacity to number of training examples (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). These bounds provide intellectual justification that machine learning algorithms can work, but they are rarely used in practice when working with deep learning algorithms. This is in part because the bounds are often quite loose and in part because it can be quite difficult to determine the capacity of deep learning algorithms.

We must remember that while simpler functions are more likely to generalize (to have a small gap between training and test error) we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming your error measure has a minimum value). Typically, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in Figure 5.3.

To reach the most extreme case of arbitrarily high capacity, we introduce the concept of *non-parametric* models. So far, we have seen only parametric models, such as linear regression. Parametric models learn a function described by a parameter vector whose size is finite and fixed before any data is observed. Non-parametric models have no such limitation.

Sometimes, non-parametric models are just theoretical abstractions (such an algorithm that searches over all possible probability distributions) that cannot be implemented in practice. However, we can also design practical non-parametric models by making their complexity a function of the training set size. One example of such an algorithm is *nearest neighbor regression.* Unlikely linear regression, which has a fixed-length vector of weights, the nearest neighbor regression model simply stores the $\boldsymbol{X}$ and $\boldsymbol{y}$ from the training set. When asked to classify a test

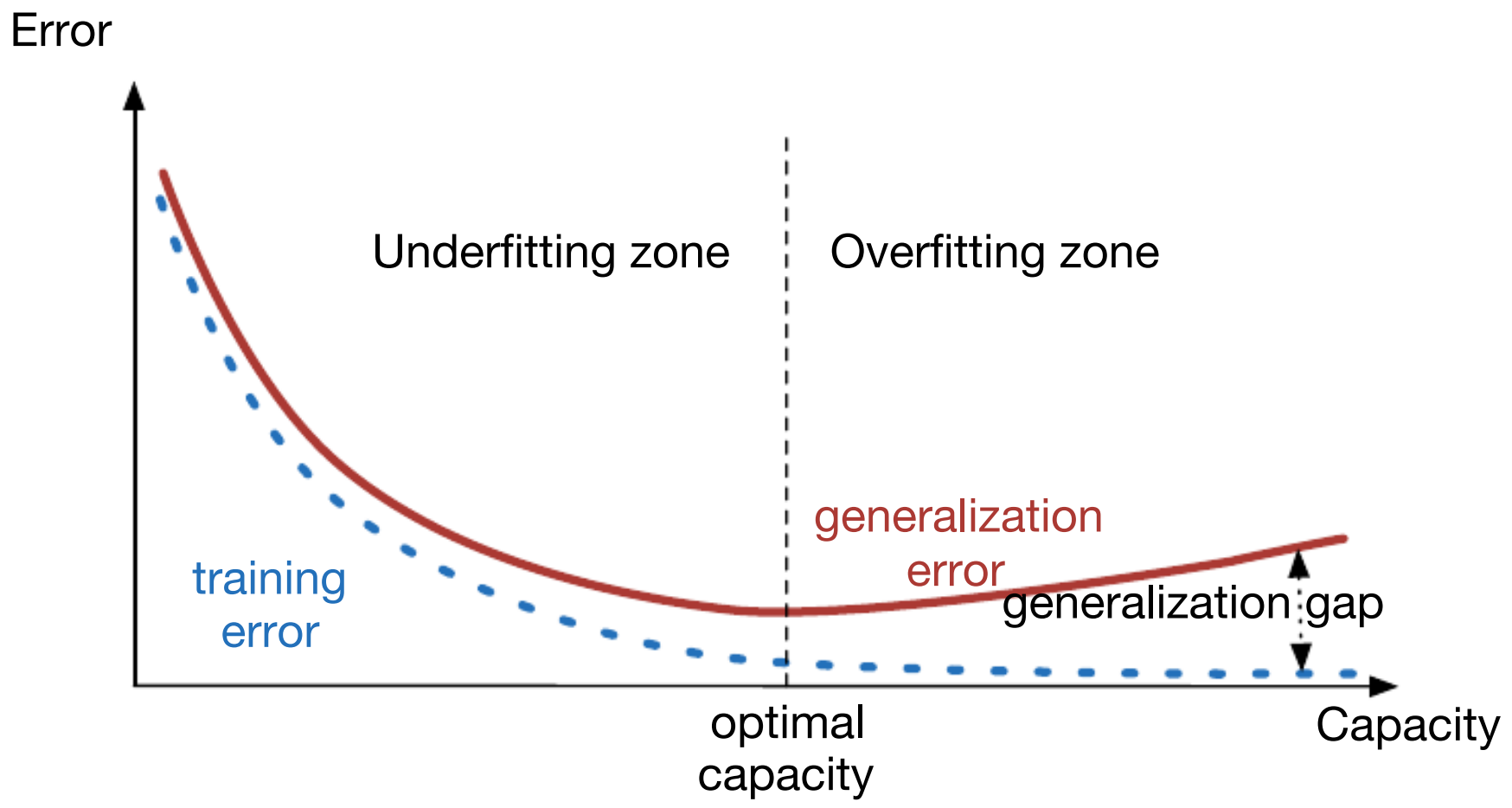


Figure 5.3: Typical relationship between capacity (horizontal axis) and both training (bottom curve, dotted) and generalization (or test) error (top curve, bold). At the left end of the graph, training error and generalization error are both high. This is the *underfitting regime*. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the *overfitting regime*, where capacity is too large, above the *optimal capacity*.

point $\boldsymbol{x}$, the model looks up the nearest entry in the training set and returns the associated regression target. In other words, $\hat{y} = y_i$ where $i = \arg\min ||\boldsymbol{X}_{i,:} - \boldsymbol{x}||_2^2$. This learning algorithm is able to achieve the minimum possible training error (which might be greater than zero, if two identical inputs are associated with different outputs) on any regression dataset.

Finally, we can also create a non-parametric learning algorithm by wrapping a parametric learning algorithm inside another algorithm that increases the number of parameters as needed. For example, we could imagine an outer loop of learning that changes the degree of the polynomial learned by linear regression on top of a polynomial expansion of the input.

The ideal model is an oracle that simply knows the true probability distribution that generates the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. In the case of regression, the mapping from $\boldsymbol{x}$ to $y$ may be inherently stochastic, or $\boldsymbol{x}$ may contain insufficient information to perfectly predict $y$, resulting in a probability distribution over possible values for $y$. The error incurred by an oracle making predictions from the true distribution $p(\boldsymbol{x}, y)$ is called the *Bayes error*.

Training and generalization vary as the size of the training set varies. Expected generalization error can never decrease as the number of training examples decreases. For non-parameteric models, more data yields better generalization until the best possible error is achieved. Any fixed parametric model with less than optimal capacity will asymptote to a higher error value. See Fig. 5.4 for an illustration. Note that it is possible for the model to have optimal capacity and yet still have a large gap between training and generalization error. In this situation, we may be able to reduce this gap by gathering more training examples.

It's worth mentioning that capacity is not just determined by which model we use. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the *representational capacity* of the model. In many cases, finding the best function within this family is a very difficult optimization problem. In practice, the learning algorithm does not actually find the best function, just one that significantly reduces the training error. These additional restrictions mean that the model's *effective capacity* may be less than its *representational capacity*.

### 5.3.1 The No Free Lunch Theorem

Although learning theory, sketched above, suggests that it is possible to generalize, one should consider a serious caveat, discussed here. Generally speaking, inductive reasoning, or inferring general rules from a limited set of examples, is not logically valid. To logically infer a rule describing every member of a set, one must have information about every member of that set. One may wonder then how the

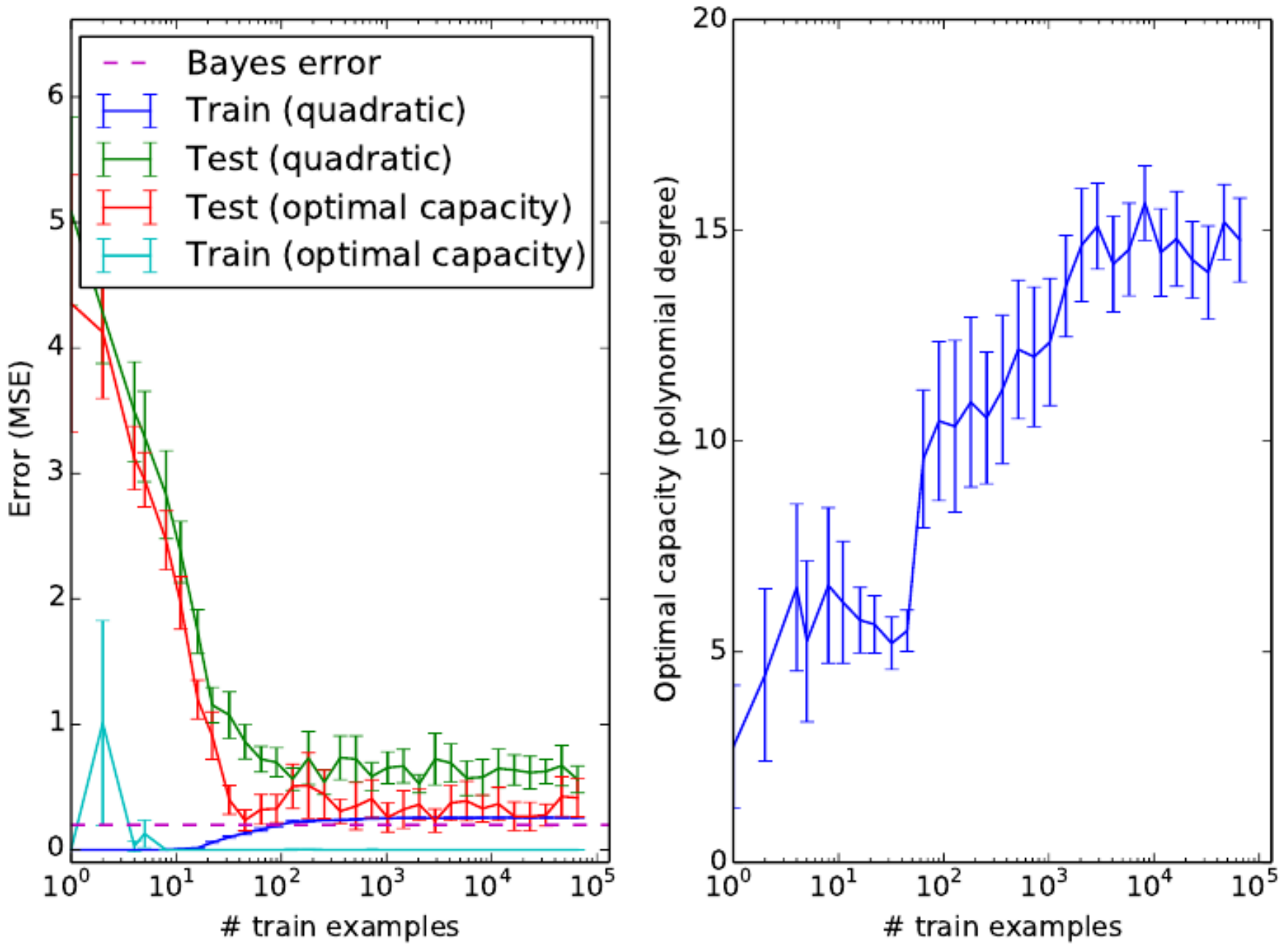


Figure 5.4: The effect of the training dataset size on the train and test error of the model, as well as on the optimal model capacity. We used a synthetic regression problem, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95% confidence intervals. Left) The mean squared error on the train and test set for two different models. One is a quadratic model, while the other has its degree chosen to minimize the test error. Both are fit in closed form, using the Moore-Penrose pseudoinverse to solve the normal equations. For the quadratic model, the training error increases as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error decreases, because fewer incorrect hypotheses are consistent with the training data. However, the quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. Ultimately test error of the optimal capacity model asymptotes to the Bayes error. The training error of the optimal capacity model can dip below the Bayes error, due to the ability of the training algorithm to memorize specific instances of this noise, but as the training size increases to infinity the training error of any model must rise to at least the Bayes error. Right) As the training set size increases, the optimal capacity increases. We observe this effect here by plotting the degree of the optimal polynomial regressor. The optimal capacity plateaus after reaching a level of sufficient complexity to solve the task.

claims that machine learning can generalize well are logically valid.

In part, machine learning avoids this problem by offering only probabilistic rules, rather than the entirely certain rules used in purely logical reasoning. Machine learning promises to find rules that are *probably* correct about *most* members of the set they concern.

Unfortunately, even this does not resolve the entire problem. The *no free lunch theorem* for machine learning (Wolpert, 1996) states that, averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other. The most sophisticated algorithm we can conceive of has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class.

Fortunately, these results hold only when we average over *all* possible data generating distributions. If we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of machine learning algorithms perform well on data drawn from the kinds of data generating distributions we care about.

### 5.3.2 Regularization

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.

So far, the only method of modifying a learning algorithm we have discussed is to increase or decrease the model's capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose. We gave the specific example of increasing or decreasing the degree of a polynomial for a regression problem. The view we have described so far is oversimplified.

The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. The learning algorithm we have studied so far, linear regression, has a hypothesis space consisting of the set of linear functions of its input. These linear functions can be very useful for problems where the relationship between inputs and outputs truly is close to linear. They are less useful for problems that behave in a very non-linear fashion. For example, linear regression

would not perform very well if we tried to use it to predict $\sin(x)$ from $x$. We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution in its hypothesis space to another. This means that both functions are eligible, but one is preferred. The unpreferred solution may only be chosen if it fits the training data significantly better than the preferred solution.

For example, we can modify the training criterion for linear regression to include *weight decay*. To perform linear regression with weight decay, we minimize not only the mean squared error on the training set, but instead a criterion $J(\boldsymbol{w})$ that expresses a preference for the weights to have smaller squared $L^2$ norm. Specifically,

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w},$$

where $\lambda$ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger $\lambda$ forces the weights to become smaller. Minimizing $J(\boldsymbol{w})$ results in a choice of weights that make a tradeoff between fitting the training data and being small. This gives us solutions that have a smaller slope, or put weight on fewer of the features. As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of $\lambda$. See Fig. 5.5 for the results.

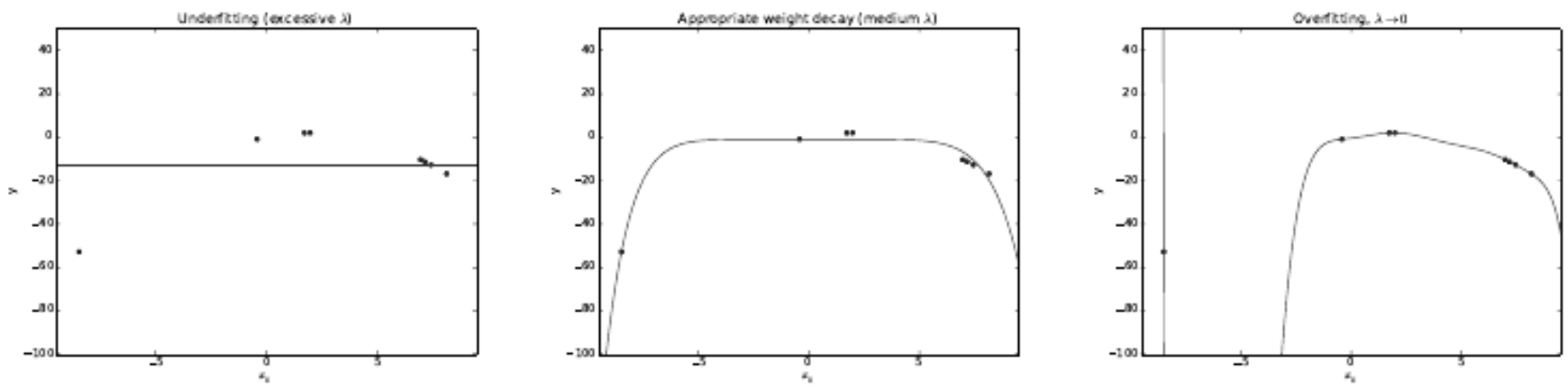


Figure 5.5: We fit a high-degree polynomial regression model to our example training set from Fig. 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (*Left*) With very large $\lambda$, we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. (*Center*) With a medium value of $\lambda$, the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients. (*Right*) With weight decay approaching zero (i.e., using the Moore-Penrose pseudo-inverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in Fig. 5.2.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

In our weight decay example, we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize. There are many other ways of expressing preferences for different solutions, both implicitly and explicitly. Together, these different approaches are known as *regularization*. **Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.** Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

The no free lunch theorem has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well-suited to the particular task we want to solve. The philosophy of deep learning in general and this book in particular is that a very wide range of tasks (such as all of the intellectual tasks that people can do) may all be solved effectively using very general-purpose forms of regularization.

## 5.4 Hyperparameters and Validation Sets

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called *hyperparameters*. The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm).

In the polynomial regression example we saw in Fig. 5.2, there is a single hyperparameter: the degree of the polynomial, which acts as a *capacity* hyperparameter. The $\lambda$ value used to control the strength of weight decay is another example of a hyperparameter.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize. More frequently, we do not learn the hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting (refer to Figure 5.3). For example, we can always fit the training set better with a higher degree polynomial and a weight decay setting of $\lambda = 0$..

To solve this problem, we need a *validation set* of examples that the training algorithm does not observe.

Earlier we discussed how a held-out test set, composed of examples coming from the same distribution as the training set, can be used to estimate the generalization error of a learner, after the learning process has completed. It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set.

For this reason, we always construct the validation set from the *training* data. Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly. The subset of data used to learn the parameters is still typically called the training set, even though this may be confused with the larger pool of data used for the entire training process. The subset of data used to guide the selection of hyperparameters is called the validation set. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the test set error, though typically by a smaller amount than the training error. Typically, one uses about 80% of the data for training and 20% for validation.

In practice, when the same test set has been used repeatedly to evaluate performance of different algorithms over many years, and especially if we consider

all the attempts from the scientific community at beating the reported state-of-the-art performance on that test set, we end up having optimistic evaluations with the test set as well. Benchmarks can thus become stale and then do not reflect the true field performance of a trained system. Thankfully, the community tends to move on to new (and usually more ambitious and larger) benchmark datasets.

### 5.4.1 Cross-Validation

One issue with the idea of splitting the dataset into train/test or train/validation/test subsets is that only a small fraction of examples are used to evaluate generalization. The consequence is that there is a lot of statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task.

With large datasets with hundreds of thousands of examples or more, this is not a serious issue, but when the dataset is too small, there are alternative procedures, which allow one to use all of the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training / testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the $k$-fold cross-validation procedure, in which a partition of the dataset is formed by splitting it in $k$ non-overlapping subsets. Then $k$ train/test splits can be obtained by keeping each time the i-th subset as a test set and the rest as a training set. The average test error across all these $k$ training/testing experiments can then be reported. One problem is that there exists no unbiased estimators of the variance of such average error estimators (Bengio and Grandvalet, 2004), but approximations are typically used.

If model selection or hyperparameter optimization is required, things get more computationally expensive: one can recurse the k-fold cross-validation idea, inside the training set. So we can have an outer loop that estimates test error and provides a "training set" for a hyperparameter-free learner, calling it $k$ times to "train". That hyperparameter-free learner can then split its received training set by k-fold cross-validation into internal training/validation subsets (for example, splitting into $k-1$ subsets is convenient, to reuse the same test blocks as the outer loop), call a hyperparameter-specific learner for each choice of hyperparameter value on each of the training partition of this inner loop, and compute the validation error by averaging across the $k-1$ validation sets the errors made by the $k-1$ hyperparameter-specific learners trained on each of the internal training subsets.

## 5.5 Estimators, Bias and Variance

The field of statistics gives us many tools that can be used to achieve the machine learning goal of solving a task not only on the training set but also to generalize. Foundational concepts such as parameter estimation, bias and variance are useful to formally characterize notions of generalization, underfitting and overfitting.

### 5.5.1 Point Estimation

Point estimation is the attempt to provide the single "best" prediction of some quantity of interest. In general the quantity of interest can be a single parameter or a vector of parameters in some parametric model, such as the weights in our linear regression example in Section 5.2, but it can also be a whole function.

In order to distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter $\theta$ by $\hat{\theta}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of $m$ *independent and identically distributed* (i.i.d.) data points. A **point estimator** is any function of the data:

$$\hat{\theta}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{5.2}$$

In other words, any statistic[1] is a point estimate. Notice that no mention is made of any correspondence between the estimator and the parameter being estimated. There is also no constraint that the range of $g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ should correspond to that of the true parameter.

This definition of a point estimator is very general and allows the designer of an estimator great flexibility. What distinguishes "just any" function of the data from most of the estimators that are in common usage is their properties. For now, we take the frequentist perspective on statistics. That is, we assume that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a function of the data. Since the data is drawn from a random process, any function of the data is random. Therefore $\hat{\boldsymbol{\theta}}$ is a random variable.

Point estimation can also refer to the estimation of the relationship between input and target variables. We refer to these types of point estimates as function estimators.

**Function Estimation** As we mentioned above, sometimes we are interested in performing function estimation (or function approximation). Here we are trying to predict a variable (or vector) $\boldsymbol{y}$ given an input vector $\boldsymbol{x}$ (also called the covariates). We consider that there is a function $f(\boldsymbol{x})$ that describes the relationship

[1] A statistic is a function of the data, typically of the whole training set, such as the mean.

between $\boldsymbol{y}$ and $\boldsymbol{x}$. For example, we may assume that $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ stands for the part of $\boldsymbol{y}$ that is not predictable from $\boldsymbol{x}$.

In function estimation, we are interested in approximating $f$ with a model or estimate $\hat{f}$. Note that we are really not adding anything new here to our notion of a point estimator, the function estimator $\hat{f}$ is simply a point estimator in function space.

The linear regression example we discussed above in Section. 5.2 and the polynomial regression example discussed in Section. 5.3 are both examples of function estimation where we estimate a model $\hat{f}$ of the relationship between an input $\boldsymbol{x}$ and target $\boldsymbol{y}$.

In the following we will review the most commonly studied properties of point estimators and discuss what they tell us about these estimators.

As $\hat{\boldsymbol{\theta}}$ and $\hat{f}$ are random variables (or vectors, or functions), they are distributed according to some probability distribution. We refer to this distribution as the *sampling distribution*. When we discuss properties of the estimator, we are really describing properties of the sampling distribution.

### 5.5.2 Bias

The bias of an estimator is defined as:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta} \tag{5.3}$$

where the expectation is over the data (seen as samples from a random variable) and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ according to the data generating distribution. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be *unbiased* if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, i.e., if $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be *asymptotically unbiased* if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, i.e., if $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

**Example: Bernoulli Distribution** Consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Bernoulli distribution, $x^{(i)} \in \{0, 1\}$, where $i \in [1, m]$. The Bernoulli *p.m.f.* (probability mass function, or probability function) is given by $P(x^{(i)}; \theta) = \theta^{x} \quad (1-\theta)^{(1-x \quad )}$.

We are interested in knowing if the estimator $\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ is biased.

$$
\begin{aligned}
\text{bias}(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \sum_{x=0}^{1} \left(x^{(i)}\theta^{x}(1-\theta)^{(1-x)}\right) - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} (\theta) - \theta \\
&= \theta - \theta = 0
\end{aligned}
$$

Since $\text{bias}(\hat{\theta}) = 0$, we say that our estimator $\hat{\theta}$ is unbiased.

**Example: Gaussian Distribution Estimator of the Mean** Now, consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Gaussian (Normal) distribution ($x^{(i)} \sim \text{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$). The Gaussian *p.d.f.* (probability density function) is given by $p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{1}{2}\frac{(x-\mu)}{\sigma}\right)$.

A common estimator of the Gaussian mean parameter is known as the *sample mean*:

$$
\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{5.4}
$$

To determine the bias of the sample mean, we are again interested in calculating its expectation:

$$
\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right]\right) - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mu\right) - \mu \\
&= \mu - \mu = 0
\end{aligned}
$$

Thus we find that the sample mean is an unbiased estimator of Gaussian mean parameter.

**Example: Gaussian Distribution Estimators of the Variance** Sticking with the Gaussian family of distributions. We consider two different estimators of the variance parameter $\sigma^2$. We are interested in knowing if either estimator is biased.

The first estimator of $\sigma^2$ we consider is known as the *sample variance*:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2, \tag{5.5}$$

where $\hat{\mu}_m$ is the sample mean, defined above. More formally, we are interested in computing

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2$$

We now simplify the term $\mathbb{E}[\hat{\sigma}_m^2]$

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2 - 2x^{(i)}\hat{\mu}_m + \hat{\mu}_m^2\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[(x^{(i)})^2\right] - 2\mathbb{E}\left[x^{(i)}\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right] + \mathbb{E}\left[\left(\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right)\left(\frac{1}{m}\sum_{k=1}^{m}x^{(k)}\right)\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(1-\frac{2}{m}\right)\mathbb{E}\left[(x^{(i)})^2\right] - \frac{2}{m}\sum_{j\neq i}\mathbb{E}\left[x^{(i)}x^{(j)}\right] + \frac{1}{m^2}\sum_{j=1}^{m}\mathbb{E}\left[(x^{(j)})^2\right]\right. \\
&\quad \left. + \frac{1}{m^2}\sum_{j=1}^{m}\sum_{k\neq j}\mathbb{E}\left[x^{(j)}x^{(k)}\right]\right) \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(\frac{m-2}{m}\right)(\mu^2+\sigma^2) - \frac{2(m-1)}{m}(\mu^2) + \frac{1}{m}(\mu^2+\sigma^2) + \frac{(m-1)}{m}(\mu^2)\right) \\
&= \frac{m-1}{m}\sigma^2
\end{aligned}$$

So the bias of $\hat{\sigma}_m^2$ is $-\sigma^2/m$. Therefore, the sample variance is a biased estimator.

We now consider a modified estimator of the variance sometimes called the *unbiased sample variance*:

$$\tilde{\sigma}^2_m = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2 \tag{5.6}$$

As the name suggests this estimator is unbiased, that is, we find that $\mathbb{E}[\tilde{\sigma}^2_m] = \sigma^2$:

$$\begin{aligned}\mathbb{E}[\tilde{\sigma}^2_m] &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\ &= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}^2_m] \\ &= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \\ &= \sigma^2.\end{aligned}$$

We have two estimators: one is biased and the other is not. While unbiased estimators are clearly desirable, they are not always the "best" estimators. As we will see we often use biased estimators that possess other important properties.

### 5.5.3 Variance and Standard Error

Another property of the estimator that we might want to consider is how much we expect it to vary as a function of the data sample. Just as we computed the expectation of the estimator to determine its bias, we can compute its *variance*.

$$\mathrm{Var}(\hat{\theta}) = \mathbb{E}[\hat{\theta}^2] - \mathbb{E}[\hat{\theta}]^2 \tag{5.7}$$

The variance of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data generating process. Just as we might like an estimator to exhibit low bias we would also like it to have relatively low variance.

When we compute the mean of a finite number of samples, there is uncertainty around it, in the sense that we could have obtained other samples from the same distribution and their mean would have been different. The expected degree of variation in any estimator is a source of error that we want to quantify. For this purpose, one defines the *standard error* (se) of an estimator $\hat{\theta}$ as

$$\mathrm{SE}(\hat{\theta}) = \sqrt{\mathrm{Var}[\hat{\theta}]}. \tag{5.8}$$

In particular, the standard error of the mean, sometimes simply called the standard error, is the standard deviation of the mean, when we consider the mean as

a random variable:

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}]} = \frac{\sigma}{\sqrt{m}} \tag{5.9}$$

where $\sigma^2$ is the true variance of the samples $x^i$. It is often estimated by replacing $\sigma$ in the above formula by the square root of the sample variance (or of the unbiased sample variance). The standard error of the mean is very useful in machine learning experiments to obtain an estimator of uncertainty around the average error made by some learning algorithm on a particular dataset. Taking advantage of the central limit theorem, which tells us that the mean will be approximately distributed with a normal distribution, we can use the standard error to compute the probability that the true expectation falls in any chosen interval (usually around the mean). For example, a 95% confidence interval around the mean is $\hat{\mu}_m$ is $(\hat{\mu}_m - 1.96\mathrm{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\mathrm{SE}(\hat{\mu}_m))$, under the normal distribution with mean $\hat{\mu}_m$ and variance $\mathrm{SE}(\hat{\mu}_m)^2$.

**Example: Bernoulli Distribution** Let's once again consider a set samples $(\{x^{(1)}, \ldots, x^{(m)}\})$ drawn independently and identically from a Bernoulli distribution (recall $P(x^{(i)}; \theta) = \theta^x \ (1-\theta)^{(1-x\ )}$). This time we are interested in computing the variance of the estimator $\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$.

$$\begin{aligned}
\mathrm{Var}\left(\hat{\theta}_m\right) &= \mathrm{Var}\ \ \frac{1}{m}\sum_{i=1}^{m} x^{(i)}\Bigg) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \mathrm{Var}\left(x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \\
&= \frac{1}{m^2} m\theta(1-\theta) \\
&= \frac{1}{m}\theta(1-\theta)
\end{aligned}$$

Note that the variance of the estimator decreases as a function of $m$, the number of examples in the dataset. This is a common property of popular estimators that we will return to when we discuss consistency (see Sec. 5.5.5).

**Example: Gaussian Distribution Estimators of the Variance** We again consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ independently and identically distributed according to a Gaussian distribution ($x^{(i)} \sim \mathrm{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$).

We now consider the variance of the two estimators of the variance: the sample variance,

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2, \tag{5.10}$$

and the unbiased sample variance,

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2. \tag{5.11}$$

In order to determine the variance of these estimators we will take advantage of a known relationship between the sample variance and the Chi Squared distribution, specifically, that $\frac{m-1}{\sigma}\hat{\sigma}^2$ happens to be $\chi^2$ distributed. We can then use this together with the fact that the variance of a $\chi^2$ random variable with $m-1$ degrees of freedom is $2(m-1)$.

$$\begin{aligned}
\mathrm{Var}\left(\frac{m-1}{\sigma^2}\tilde{\sigma}^2\right) &= 2(m-1) \\
\frac{(m-1)^2}{\sigma^4}\mathrm{Var}\left(\tilde{\sigma}^2\right) &= 2(m-1) \\
\mathrm{Var}\left(\tilde{\sigma}^2\right) &= \frac{2\sigma^4}{(m-1)}
\end{aligned}$$

By noticing that $\hat{\sigma}^2 = \frac{m-1}{m}\tilde{\sigma}^2$, and using $\tilde{\sigma}^2$'s relationship to the $\chi^2$ distribution, it is straightforward to show that $\mathrm{Var}\left(\hat{\sigma}^2\right) = \frac{2(m-1)\sigma}{m}$.

To derive this last relation, we used the fact that $\mathrm{Var}\left(\tilde{\sigma}^2\right) = \left(\frac{m}{m-1}\right)^2 \mathrm{Var}\left(\hat{\sigma}^2\right)$, that is $\mathrm{Var}\left(\tilde{\sigma}^2\right) > \mathrm{Var}\left(\hat{\sigma}^2\right)$. So while the bias of $\tilde{\sigma}^2$ is smaller than the bias of $\hat{\sigma}^2$, the variance of $\tilde{\sigma}^2$ is greater.

### 5.5.4 Trading off Bias and Variance and the Mean Squared Error

Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the true value that any particular sampling of the data is likely to cause.

What happens when we are given a choice between two estimators, one with more bias and one with more variance? How do we choose between them? For example, let's imagine that we are interested in approximating the function shown in Fig. 5.2 and we are only offered the choice between a model with large bias and one that suffers from large variance. How do we choose between them?

In machine learning, perhaps the most common and empirically successful way to negotiate this kind of trade-off, in general is by cross-validation. Alternatively, we can also compare the *mean squared error* (MSE) of the estimates:

$$\begin{aligned}\mathrm{MSE} &= \mathbb{E}[(\hat{\theta}_n - \theta)^2] \\ &= \mathrm{Bias}(\hat{\theta}_n)^2 + \mathrm{Var}(\hat{\theta}_n)\end{aligned} \tag{5.12}$$

The MSE measures the overall expected deviation—in a squared error sense—between the estimator and the true value of the parameter $\theta$. As is clear from Eq. 5.12, evaluating the MSE incorporates both the bias and the variance. Desirable estimators are those with small MSE and these are estimators that manage to keep both their bias and variance somewhat in check.

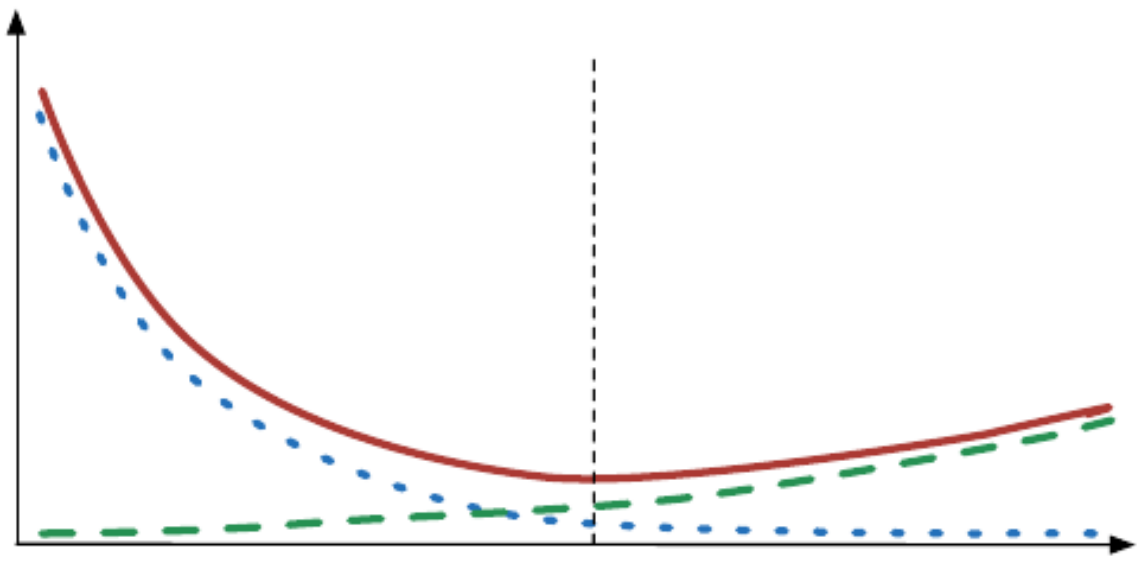

Figure 5.6: As capacity increases ($x$-axis), bias (dotted) tends to decrease and variance (dashed) tends to increase, yielding another U-shaped curve for generalization error (bold curve). If we vary capacity along one axis, there is an optimal capacity, with underfitting when the capacity is below this optimum and overfitting when it is above. This relationship is similar to the relationship between capacity, underfitting, and overfitting, discussed in Section 5.3 and Figure 5.3.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. In the case where generalization error is measured by the MSE (where bias and variance are meaningful components of generalization error), increasing capacity tends to increase variance and decrease bias. This is illustrated in Figure 5.6, where we see again the U-shaped curve of generalization error as a function of of capacity.

**Example: Gaussian Distribution Estimators of the Variance** In the last section we saw that when we compared the sample variance, $\hat{\sigma}^2$, and the unbiased sample variance, $\tilde{\sigma}^2$, we see that while $\hat{\sigma}^2$ has higher bias, $\tilde{\sigma}^2$ has higher variance.

The mean squared error offers a way of balancing the tradeoff between bias and variance and suggest which estimator we might prefer. For $\hat{\sigma}^2$, the mean squared error is given by:

$$\begin{aligned}
\mathrm{MSE}(\hat{\sigma}_m^2) &= \mathrm{Bias}(\hat{\sigma}_m^2)^2 + \mathrm{Var}(\hat{\sigma}_m^2) && (5.13)\\
&= \left(\frac{-\sigma^2}{m}\right)^2 + \frac{2(m-1)\sigma^4}{m^2} && (5.14)\\
&= \left(\frac{1+2(m-1)}{m^2}\right)\sigma^4 && (5.15)\\
&= \left(\frac{2m-1}{m^2}\right)\sigma^4 && (5.16)
\end{aligned}$$

The mean squared error of the unbiased alternative is given by:

$$\begin{aligned}
\mathrm{MSE}(\tilde{\sigma}_m^2) &= \mathrm{Bias}(\tilde{\sigma}_m^2)^2 + \mathrm{Var}(\tilde{\sigma}_m^2) && (5.17)\\
&= 0 + \frac{2\sigma^4}{(m-1)} && (5.18)\\
&= \frac{2}{(m-1)}\sigma^4. && (5.19)
\end{aligned}$$

Comparing the two, we see that the MSE of the unbiased sample variance, $\tilde{\sigma}_m^2$ , is actually higher than the MSE of the (biased) sample variance, $\hat{\sigma}_m^2$ . This implies that despite incurring bias in the estimator $\hat{\sigma}_m^2$, the resulting reduction in variance more than makes up for the difference, at least in a mean squared sense.

### 5.5.5 Consistency

As we have already discussed, sometimes we may wish to choose an estimator that is biased. For example, in order to minimize the variance of the estimator. However we might still wish that, as the number of data points in our dataset increases, our point estimates converge to the true value of the parameter. More formally, we would like that $\lim_{n\to\infty} \hat{\theta}_n \xrightarrow{p} \theta$.[2] This condition is known as *consistency*[3] and ensures that the bias induced by the estimator is assured to diminish as the number of data examples grows.

Asymptotic unbiasedness is not equivalent to consistency. For example, consider estimating the mean parameter $\mu$ of a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with a dataset consisting of $n$ samples: $\{x_1, \ldots, x_n\}$. We could use the first sample

---

The symbol means that the convergence is in probability, i.e. for any 0, ( ) 0 as .

This is sometime referred to as weak consistency, with strong consistency referring to the convergence of to .

$x_1$ of the dataset as an *unbiased* estimator: $\hat{\theta} = x_1$, In that case, $\mathbb{E}(\hat{\theta}_n) = \theta$ so the estimator is unbiased no matter how many data points are seen. This, of course, implies that the estimate is asymptotically unbiased. However, this is not a consistent estimator as it is *not* the case that $\hat{\theta}_n \to \theta$ as $n \to \infty$.

## 5.6 Maximum Likelihood Estimation

Previously, we have seen some definitions of common estimators and analyzed their properties. But where did these estimators come from? Rather than guessing that some function might make a good estimator and then analyzing its bias and variance, we would like to have some principle from which we can derive specific functions that are good estimators for different models.

The most common such principle is the maximum likelihood principle.

Consider a set of $m$ examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn independently from the true but unknown data generating distribution $p_{\text{data}}(\mathbf{x})$.

Let $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$. In other words, $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$ maps any configuration $\boldsymbol{x}$ to a real number estimating the true probability $p_{\text{data}}(\boldsymbol{x})$.

The maximum likelihood estimator for $\boldsymbol{\theta}$ is then defined as

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{5.20}$$

$$= \arg\max \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.21}$$

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. To attain a more convenient but equivalent optimization problem, we observe that the logarithm of the $\arg\max$ is the $\arg\max$ of logarithm:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.22}$$

Because the argmax does not change when we rescale the cost function, we can divide by $m$ to obtain a version of the criterion that is expressed as an expectation:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max \mathbb{E}_{\sim \hat{p}} \quad \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{5.23}$$

One way to interpret maximium likelihood estimation is to view at as minimizing the dissimilarity between the empirical distribution defined by the training

set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by

$$D_{\mathrm{KL}}\left(\hat{p}_{\mathrm{data}} \| p_{\mathrm{model}}\right) = \mathbb{E}_{\sim \hat{p}} \left[\log \hat{p}_{\mathrm{data}}(\boldsymbol{x}) - \log p_{\mathrm{model}}(\boldsymbol{x})\right].$$

Note that the term on the left is a function only of the data generating process, not the model. This means when we train the model to minimize the KL divergence, we need only minimize

$$-\mathbb{E}_{\sim \hat{p}} \left[\log \hat{p}_{\mathrm{model}}(\boldsymbol{x})\right]$$

which is of course the same as the maximization in Eq. 5.23. Note that this also corresponds exactly to minimizing the cross entropy between the distributions.

We can thus see maximum likelihood as an attempt to make the model distribution match the empirical distribution $\hat{p}_{\mathrm{data}}$. Ideally, we would like to match the true data generating distribution $p_{\mathrm{data}}$, but we have no direct access to this distribution.

While the optimal $\boldsymbol{\theta}$ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence, the values of the objective functions are different. In software, we often phrase both as minimizing a cost function. Maximimum likelihood thus becomes minimization of the negative log-likelihood (NLL), or equivalently, minimization of the cross entropy. The perspective of maximum likelihood as minimum KL divergence becomes helpful in this case because the KL divergence has a known minimum value of zero. The negative log-likelihood can actually become negative when $\boldsymbol{x}$ is real-valued.

### 5.6.1 Conditional Log-Likelihood and Mean Squared Error

The maximum likelihood estimator can readily be generalized to the case where our goal is not to estimate a probability function but rather a *conditional probability*, e.g., $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$, to predict $\mathbf{y}$ given $\mathbf{x}$. This is actually the most common situation where we do supervised learning (Section 5.8), i.e., the examples are pairs $(\boldsymbol{x}, \boldsymbol{y})$. If $\boldsymbol{X}$ represents all our inputs and $\boldsymbol{Y}$ all our observed targets, then the conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}). \tag{5.24}$$

If the examples are assumed to be i.i.d., then this can be decomposed into

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.25}$$

**Example: Linear Regression** Let us consider as an example the special case of linear regression, introduced earlier in Section 5.2. In that case, the conditional density of $\mathbf{y}$, given $\mathbf{x} = \boldsymbol{x}$, is a Gaussian with mean $\mu(\boldsymbol{x})$ that is a learned function of $\boldsymbol{x}$, with unconditional variance $\sigma^2$. Since the examples are assumed to be i.i.d., the conditional log-likelihood (Eq. 5.24) becomes

$$\begin{aligned}\log P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}) &= \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\ &= \sum_{i=1}^{m} \frac{-1}{2\sigma^2} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2 - m \log \sigma - \frac{m}{2} \log(2\pi)\end{aligned}$$

where $\hat{\boldsymbol{y}}^{(i)} = \mu(\boldsymbol{x}^{(i)})$ is the output of the linear regression on the $i$-th input $\boldsymbol{x}^{(i)}$ and $m$ is the dimension of the $\boldsymbol{y}$ vectors. Comparing the above with the mean squared error (Section 5.2) we immediately see that if $\sigma$ is fixed, maximizing the above is equivalent (up to an additive and a multiplicative constant that do not change the value of the optimal parameter) to minimizing the training set mean squared error, i.e.,

$$MSE_{\text{train}} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2.$$

Note that the MSE is an average rather than a sum, which is more practical from a numerical point of view (so you can compare MSEs of sets of different sizes more easily). In practice, researchers reporting log-likelihoods and conditional log-likelihoods also tend to report the per-example average log-likelihood, for the very same reason. The exponential of the average log-likelihood is also called the *perplexity* and is used in language modeling applications.

Whereas in the case of linear regression we have $\mu(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x}$, the above equally applies to other forms of regression, e.g., with a neural network predicting with $\mu(\boldsymbol{x})$ the expected value of $\mathbf{y}$ given $\boldsymbol{x}$.

### 5.6.2 Properties of Maximum Likelihood

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \to \infty$, in terms of its rate of convergence as $m$ increases.

The maximum likelihood estimator has the property of consistency (see Sec. 5.5.5 above), i.e., as more training are considered, the estimator converges to the best one in some sense. There are other inductive principles besides the maximum likelihood estimator, many of which share the property of being consistent estimators. However, there is the question of how many training examples one needs to achieve a particular generalization error, or equivalently what estimation error

one gets for a given number of training examples, also called *efficiency*. This is typically studied in the *parametric case* (like in linear regression) where our goal is to estimate the value of a parameter (and assuming it is possible to identify the true parameter), not the value of a function. A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter values, where the expectation is over $m$ training samples from the data generating distribution. That parametric mean squared error decreases as $m$ increases, and for $m$ large, the Cramér-Rao lower bound (Rao, 1945; Cramér, 1946) shows that no consistent estimator has a lower mean squared error than the maximum likelihood estimator.

For these reasons (consistency and efficiency), the maximum likelihood induction principle is often considered the preferred one in machine learning, modulo slight adjustments such as described in the next Section, to better deal with the non-asymptotic case where the number of examples is small enough to yield overfitting behavior.

## 5.7 Bayesian Statistics

So far we have discussed approaches based on estimating a single value of $\boldsymbol{\theta}$, then making all predictions thereafter based on that one estimate. Another approach is to consider all possible values of $\boldsymbol{\theta}$ when making a prediction. *Bayesian statistics* provides a natural and theoretically elegant way to carry out this approach.

Historically, statistics has become divided between two communities. One of these communities is known as *frequentist statistics* or *orthodox statistics*. The other is known as *Bayesian statistics*. The difference is mainly one of world view but can have important practical implications.

As discussed in Sec. 5.5.1, the frequentist perspective is that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a random variable on account of it being a function of the data (which are seen as random).

The Bayesian perspective on statistics is quite different and, in some sense, more intuitive. The Bayesian uses probability to reflect degrees of certainty of states of knowledge. The data is directly observed and so is not random. On the other hand, the true parameter $\boldsymbol{\theta}$ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent our knowledge of $\boldsymbol{\theta}$ using the *prior probability distribution*, $p(\boldsymbol{\theta})$ (sometimes referred to as simply 'the prior'). Generally, the prior distribution is quite broad (i.e. with high entropy) to reflect a high degree of uncertainty in the value of $\boldsymbol{\theta}$ before observing any data. For example, we might assume *a priori* that $\boldsymbol{\theta}$ lies in some finite range or volume, with a uniform distribution. Many priors instead reflect a preference for "simpler"

solutions (such as smaller magnitude coefficients, or a function that is closer to being constant).

Now consider that we have a set of data samples $\{x^{(1)}, \ldots, x^{(m)}\}$. We can recover the effect of data on our belief about $\boldsymbol{\theta}$ by combining the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta})$ with the prior via Bayes' rule:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})} \tag{5.26}$$

If the data is at all informative about the value of $\boldsymbol{\theta}$, the *posterior distribution* $p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})$ will have less entropy (will be more 'peaky') than the prior $p(\boldsymbol{\theta})$.

Relative to maximum likelihood estimation, Bayesian estimation offers two important differences. First, unlike the maximum likelihood point estimate of $\boldsymbol{\theta}$, the Bayesian makes decision with respect to a full distribution over $\boldsymbol{\theta}$. For example, after observing $m$ examples, the predicted distribution over the next data sample, $x^{(m+1)}$, is given by

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})\, d\boldsymbol{\theta} \tag{5.27}$$

Here each value of $\boldsymbol{\theta}$ with positive probability density contributes to the prediction of the next example, with the contribution weighted by the posterior density itself. After having observed $\{x^{(1)}, \ldots, x^{(m)}\}$, if we are still quite uncertain about the value of $\boldsymbol{\theta}$, then this uncertainty is incorporated directly into any predictions we might make.

In Sec. 5.5, we discussed how the frequentist statistics addresses the uncertainty in a given point estimator of $\boldsymbol{\theta}$ by evaluating its variance. The variance of the estimator is an assessment of how the estimate might change will alternative samplings of the observed (or training) data. The Bayesian answer to the question of how to deal with the uncertainty in the estimator is to simply integrate over it, which tends to protect well against overfitting.

The second important difference between the Bayesian approach to estimation and the Maximum Likelihood approach is due to the contribution of the Bayesian prior distribution. The prior has an influence by shifting probability mass density towards regions of the parameter space that are preferred *a priori*. In practice, the prior often expresses a preference for models that are simpler or more smooth. One important effect of the prior is to actually reduce the uncertainty (or entropy) in the posterior density over $\boldsymbol{\theta}$.

We have already noted that combining the prior, $p(\boldsymbol{\theta})$, with the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta})$ results in a distribution that is less entropic (more peaky) than the prior. This is just the result of a basic property of probability distributions: Entropy(product of two densities) $\leq$ Entropy(either density). This implies

that the posterior density on $\boldsymbol{\theta}$ is also less entropic than the data likelihood alone (when viewed and normalized as a density over $\boldsymbol{\theta}$). The hypothesis space with the Bayesian approach is, to some extent, more constrained than that with an ML approach. Thus we expect a contribution of the prior to be a further reduction in overfitting as compared to ML estimation.

**Example: Linear Regression** Here we consider the Bayesian estimation approach to learning the linear regression parameters. In linear regression, we learn a linear mapping from an input vector $\boldsymbol{x} \in \mathbb{R}^n$ to predict the value of a scalar $y \in \mathbb{R}$. The prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}.$$

Given a set of $m$ training samples $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of $y$ over the entire training set as:

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}.$$

Expressed as a Gaussian conditional distribution on $\boldsymbol{y}^{(\text{train})}$, we have

$$\begin{aligned} p(\boldsymbol{y}^{(\text{train})} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{w}) &= \mathcal{N}(\boldsymbol{y}^{(\text{train})}; \boldsymbol{X}^{(\text{train})\top} \boldsymbol{w}, \boldsymbol{I}) \\ &\propto \exp\left(-\frac{1}{2}(\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w})^\top (\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w})\right), \end{aligned}$$

where we will follow the standard MSE formulation in assuming that the Gaussian variance on $y$ is one. In what follows, to reduce the notational burden, we refer to $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$ as simply $(\boldsymbol{X}, \boldsymbol{y})$.

To determine the posterior distribution over the model parameter vector $\boldsymbol{w}$, we first need to specify a prior distribution. The prior should reflect our naive belief about the value of these parameters. While it is sometimes difficult or unnatural to express our prior beliefs in terms of the parameters of the model, in practice we typically assume a fairly broad distribution expressing a high degree of uncertainty about $\boldsymbol{\theta}$ in our prior belief.

For real-valued parameters it is common to use a Gaussian as a prior distribution:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \Lambda_0) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1} (\boldsymbol{w} - \boldsymbol{\mu}_0)\right)$$

where $\boldsymbol{\mu}_0$ and $\Lambda_0$ are the prior distribution mean vector and covariance matrix (inverse of covariance matrix) respectively.[4]

---

Unless there is a reason to assume a particular covariance structure, we typically assume a diagonal covariance matrix $\Lambda = \text{diag}(\quad)$.

With the prior thus specified, we can now proceed in determining the *posterior* distribution over the model parameters.

$$
\begin{aligned}
p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) &\propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \\
&\propto \exp\left(-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^{\top}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})\right) \exp\left(-\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu}_0)^{\top}\Lambda_0^{-1}(\boldsymbol{w}-\boldsymbol{\mu}_0)\right) \\
&\propto \exp\left(-\frac{1}{2}\left(-2\boldsymbol{y}^{\top}\boldsymbol{X}\boldsymbol{w}+\boldsymbol{w}^{\top}\boldsymbol{X}^{\top}\boldsymbol{X}\boldsymbol{w}+\boldsymbol{w}^{\top}\Lambda_0^{-1}\boldsymbol{w}-2\boldsymbol{\mu}_0^{\top}\Lambda_0^{-1}\boldsymbol{w}\right)\right)
\end{aligned}
$$

We now make the substitutions $\Lambda_m = \left(\boldsymbol{X}^{\top}\boldsymbol{X}+\Lambda_0^{-1}\right)^{-1}$ and $\boldsymbol{\mu}_m = \Lambda_m\left(\boldsymbol{X}^{\top}\boldsymbol{y}+\Lambda_0^{-1}\boldsymbol{\mu}_0\right)$ into the derivation of the posterior (and complete the square) to get:

$$
p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu}_m)^{\top}\Lambda_m^{-1}(\boldsymbol{w}-\boldsymbol{\mu}_m)+\frac{1}{2}\boldsymbol{\mu}_m^{\top}\Lambda_m^{-1}\boldsymbol{\mu}_m\right) \tag{5.28}
$$

$$
\propto \exp\left(-\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu}_m)^{\top}\Lambda_m^{-1}(\boldsymbol{w}-\boldsymbol{\mu}_m)\right). \tag{5.29}
$$

In the above, we have dropped all terms that do not include the parameter vector $\boldsymbol{w}$. In Eq. 5.29, we recognize that the posterior distribution has the form of a Gaussian distribution with mean vector $\boldsymbol{\mu}_m$ and covariance matrix $\Lambda_m$. It is interesting to note that this justifies our dropping all terms unrelated to $\boldsymbol{w}$, since we know that the posterior distribution must be normalized and, as a Gaussian, we know what that normalization constant must be (where $n$ is the dimension of the input):

$$
p(\boldsymbol{w} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})}) = \frac{1}{\sqrt{(2\pi)^n\,|\Lambda_m|}} \exp\left(-\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu}_m)^{\top}\Lambda_m^{-1}(\boldsymbol{w}-\boldsymbol{\mu}_m)\right). \tag{5.30}
$$

### 5.7.1 Maximum (MAP) Estimation

While, in principle, we can use the full Bayesian posterior distribution over the parameter $\boldsymbol{\theta}$ as our estimate of this parameter, it is still often desirable to have a single point estimate (for example, most operations involving the Bayesian posterior for most interesting models are intractable and must be heavily approximated). Rather than simply returning to the maximum likelihood estimate, we can still gain some of the benefit of the Bayesian approach by allowing the prior to influence the choice of the point estimate. One rational way to do this is to choose the *maximum a posteriori* (MAP) point estimate. The MAP estimate chooses the point of maximal posterior probability (or maximal probability density in the

more common case of continuous $\boldsymbol{\theta}$).

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \tag{5.31}$$

We recognize, above on the right hand side, $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$, i.e. the standard log-likelihood term and $\log p(\boldsymbol{\theta})$ corresponding to the prior distribution.

As discussed above the advantage brought by introducing the influence of the prior on the MAP estimate is to leverage information other than that contained in the training data. This additional information helps to reduce the variance in the MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

**Example: Regularized Linear Regression** We discussed above the Bayesian approach to linear regression. Given a set of $m$ training samples of input output pairs: $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of $y$ over the entire training set as:

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}.$$

where prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$.

Recall from Sec. 5.6.1 that the maximum likelihood estimate for the model parameters is given by:

$$\hat{\boldsymbol{w}}_{\text{ML}} = (\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})})^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.32}$$

For the sake of comparison to the maximum likelihood solution, we will make the simplifying assumption that the prior covariance matrix is scalar: $\Lambda_0 = \lambda_0 \boldsymbol{I}$. As mentioned previously, in practice, this is a very common form of prior distribution. We will also assume that $\boldsymbol{\mu}_0 = 0$. This is also a very common assumption in practice and corresponds to acknowledging that *a priori*, we do not know if the features of $\boldsymbol{x}$ have a positive or negative correlation with $y$. Adding these assumptions, the MAP estimate of the model parameters (corresponding to the mean of the Gaussian posterior density, in Eq. 5.29) becomes:

$$\hat{\boldsymbol{w}}_{\text{MAP}} = \Lambda_m \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.33}$$

where $\boldsymbol{\mu}_0$ and $\Lambda_0$ are the prior mean and covariance respectively and $\Lambda_m$ is the posterior covariance and is given by:

$$\Lambda_m = \left( \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} + \lambda_0^{-1} \boldsymbol{I} \right)^{-1} \tag{5.34}$$

Comparing Eqs. 5.32 and 5.33, we see that the MAP estimate amounts to a weighted combination of the prior maximum probability value, $\boldsymbol{\mu}_0$, and the ML

estimate. As the variance of the prior distribution tends to infinity, the MAP estimate reduces to the ML estimate. As the variance of the prior tends to zero, the MAP estimate tends to zero (actually it tends to $\boldsymbol{\mu}_0$ which here is assumed to be zero).

We can make the model capacity tradeoff between the ML estimate and the MAP estimate more explicit by analyzing the bias and variance of these estimates.

It is relatively easy to show that the ML estimate is unbiased, i.e. that $\mathbb{E}[\hat{\boldsymbol{w}}_{\mathrm{ML}}] = \boldsymbol{w}$ and that it has a variance given by:

$$\mathrm{Var}(\boldsymbol{w}_{\mathrm{ML}}) = (\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})})^{-1} \tag{5.35}$$

In order to derive the bias of the MAP estimate, we need to calculate the expectation:

$$\begin{aligned}
E[\hat{\boldsymbol{w}}_{\mathrm{MAP}}] &= \mathbb{E}[\Lambda_m \boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{y}^{(\mathrm{train})}] \\
&= E\left[\Lambda_m \boldsymbol{X}^{(\mathrm{train})\top}\left(\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} + \epsilon\right)\right] \\
&= \Lambda_m\left(\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w}\right) + \Lambda_m \boldsymbol{X}^{(\mathrm{train})\top} E[\epsilon] \\
&= \left(\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})} + \lambda_0^{-1}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w},
\end{aligned} \tag{5.36}$$

We see that while the expected value of the ML estimate is the true parameter value $\boldsymbol{w}$ (i.e. the parameters that we assume generated the data); the expected value of the MAP estimate is a weighted average of $\boldsymbol{w}$ and the prior mean $\boldsymbol{\mu}$. We compute the bias as:

$$\begin{aligned}
\mathrm{Bias}(\hat{\boldsymbol{w}}_{\mathrm{MAP}}) &= E[\hat{\boldsymbol{w}}_{\mathrm{MAP}}] - \boldsymbol{w} \\
&= -\left(\lambda_0 \boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})} + \boldsymbol{I}\right)^{-1}\boldsymbol{w}.
\end{aligned}$$

Since the bias is not zero, we can conclude that the MAP estimate is biased, and as expected we can see that as the variance of the prior $\lambda_0 \to \infty$, the bias tends to zero. As the variance of the prior $\lambda_0 \to \mathbf{0}$, the bias tends to $\boldsymbol{w}$.

In order to compute the variance, we use the identity $\mathrm{Var}(\hat{\theta}) = \mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2$.

So before computing the variance we need to compute $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$:

$$\begin{aligned}
\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] &= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\hat{\boldsymbol{y}}^{(\text{train})}\boldsymbol{y}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\
&= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\Big(\boldsymbol{X}^{(\text{train})}\boldsymbol{w}+\boldsymbol{\epsilon}\Big)\Big(\boldsymbol{X}^{(\text{train})}\boldsymbol{w}+\boldsymbol{\epsilon}\Big)^{\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\
&= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\
&\qquad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\top}\right]\boldsymbol{X}^{(\text{train})}\Lambda_m \\
&= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\
&\qquad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\
&= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m
\end{aligned}$$

With $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$ thus computed, the variance of the MAP estimate of our linear regression model is given by:

$$\begin{aligned}
\text{Var}(\hat{\boldsymbol{w}}_{\text{MAP}}) &= \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] - \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\right]\mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}^{\top}\right] \\
&= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m - E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} \\
&= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\
&= \Big(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\Big)^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} \\
&\qquad \times \Big(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\Big)^{-1}
\end{aligned} \tag{5.37}$$

It is perhaps difficult to compare Eqs. 5.35 and 5.37. But if we assume that $\boldsymbol{w}$ is one-dimensional (along with $\boldsymbol{x}$), it becomes a bit easier to see that, as long as $\lambda_0$ is bounded, then $\text{Var}(\hat{w}_{\text{ML}}) = \frac{1}{x} > \text{Var}(\hat{w}_{\text{MAP}}) = \frac{\lambda x}{(1+\lambda x)}$.

From the above analysis we can see that the role of the prior in the MAP estimate is to trade increased bias for a reduction in variance. The goal, of course, is to try to avoid overfitting. The incurred bias is a consequence of the reduction in model capacity caused by limiting the space of hypotheses to those with significant probability density under the prior.

Many regularized estimation strategies, such as maximum likelihood learning regularized with weight decay, can be interpreted as making the MAP approximation to Bayesian inference. This view applies when the regularization consists of adding an extra term to the objective function that corresponds to $\log p(\boldsymbol{\theta})$. Not all such regularizer terms correspond to MAP Bayesian inference. For example, some regularizer terms may not be the logarithm of a probability distribution. Other regularization terms depend on the data, which of course a prior probability distribution is not allowed to do.

## 5.8 Supervised Learning Algorithms

Recall from Section 5.1.3 that supervised learning algorithms are roughly speaking, learning algorithms that learn to associate some input with some output, given a training set of examples of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. In many cases the outputs $\boldsymbol{y}$ may be difficult to collect automatically and must be provided by a human "supervisor," but the term still applies even when the training set targets were collected automatically.

### 5.8.1 Probabilistic Supervised Learning

Most supervised learning algorithms in this book are based on estimating a probability distribution $p(y \mid \boldsymbol{x})$. We can do this simply by using maximum conditional likelihood estimation (Sec. 5.6.1, also just called maximum likelihood for short) to find the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

We have already seen that linear regression corresponds to the family $p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y \mid \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I})$. We can generalize linear regression to the classification scenario by defining a different family of probability distributions. If we have two classes, class 0 and class 1, then we need only specify the probability of one of these classes. The probability of class 1 determines the probability of class 0, because these two values must add up to 1.

The normal distribution over real-valued numbers that we used for linear regression is parameterized in terms of a mean. Any value we supply for this mean is valid. A distribution over a binary variable is slightly more complicated, because its mean must always be between 0 and 1. One way to solve this problem is to use the logistic sigmoid function to squash the output of the linear function into the interval (0, 1) and interpret that value as a probability:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}).$$

This approach is known as *logistic regression* (a somewhat strange name since we use the model for classification rather than regression).

In the case of linear regression, we were able to find the optimal weights by solving the normal equations. Logistic regression is somewhat more difficult. There is no closed-form solution for its optimal weights. Instead, we must search for them by maximizing the log-likelihood. We can do this by minimizing the negative log-likelihood (NLL) using gradient descent.

This same strategy can be applied to essentially any supervised learning problem, by writing down a parametric family of probability of conditional distributions over the right kind of input and output variables.

### 5.8.2 Support Vector Machines

One of the most influential approaches to supervised learning is the support vector machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995). This model is similar to logistic regression in that it is driven by a linear function $\boldsymbol{w}^\top \boldsymbol{x} + b$. Unlike logistic regression, the support vector machine does not provide probabilities, but only outputs a class identity.

One key innovation associated with support vector machines is the *kernel trick*. The kernel trick consists of observing that many machine learning algorithms can be written exclusively in terms of dot products between examples. For example, it can be shown that the linear function used by the support vector machine can be re-written as

$$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)}$$

where $\boldsymbol{x}^{(i)}$ is a training example and $\boldsymbol{\alpha}$ is a vector of coefficients. Rewriting the learning algorithm this way allows us to replace $\boldsymbol{x}$ by the output of a given feature function $\phi(\boldsymbol{x})$ and the dot product with a function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$ called a *kernel*.

We can then make predictions using the function

$$f(x) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{5.38}$$

This function is linear in the space that $\phi$ maps to, but non-linear as a function of $\boldsymbol{x}$.

The kernel trick is powerful for two reasons. First, it allows us to learn models that are non-linear as a function of $\boldsymbol{x}$ using convex optimization techniques that are guaranteed to converge efficiently. This is only possible because we consider $\phi$ fixed and only optimize $\boldsymbol{\alpha}$, i.e., the optimization algorithm can view the decision function as being linear in a different space. Second, the kernel function $k$ need not be implemented in terms of explicitly applying the $\phi$ mapping and then applying the dot product. The dot product in $\phi$ space might be equivalent to a non-linear but computationally less expensive operation in $\boldsymbol{x}$ space. For example, we could design an *infinite-dimensional* feature mapping $\phi(\boldsymbol{x})$ over the non-negative integers. Suppose that this mapping returns a vector containing $\boldsymbol{x}$ ones followed by infinitely many zeros. Explicitly constructing this mapping, or taking the dot product between two such vectors, costs infinite time and memory. But we can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ that is exactly equivalent to to this infinite-dimensional dot product. The most commonly used kernel is the *Gaussian kernel*

$$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 I) \tag{5.39}$$

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel corresponds to the dot product $k(\boldsymbol{u}, \boldsymbol{v}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x})$ on an infinite-dimensional feature space $\phi$ and also has an interpretation as a similarity function, acting like a kind of template matching.

Support vector machines are not the only algorithm that can be enhanced using the kernel trick. Many linear models can be enhanced in this way. This category of algorithms is known as *kernel machines* or *kernel methods*.

A major drawback to kernel machines is that the cost of learning the $\boldsymbol{\alpha}$ coefficients is quadratic in the number of training examples. A related problem is that the cost of evaluating the decision function is linear in the number of training examples, because the $i$-th example contributes a term $\alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ to the decision function. Support vector machines are able to mitigate this by learning an $\boldsymbol{\alpha}$ vector that contains mostly zeros. Classifying a new example then requires evaluating the kernel function only for the training examples that have non-zero $\alpha_i$. These training examples are known as *support vectors*. Another major drawback of common kernel machines (such as those using the Gaussian kernel) is more statistical and regards their difficulty in generalizing to complex variations far from the training examples, as discussed in Section 5.12.

The analysis of the statistical limitations of support vector machines with general purpose kernels like the Gaussian kernels actually motivated the rebirth of neural networks through deep learning. Support vector machines and other kernel machines have often been viewed as a competitor to deep learning (though some deep networks can in fact be interpreted as support vector machines with learned kernels). The current deep learning renaissance began when deep networks were shown to outperform support vector machines on the MNIST benchmark dataset (Hinton *et al.*, 2006). One of the main reasons for the current popularity of deep learning relative to support vector machines is the fact that the cost of training kernel machines usually scales quadratically with the number of examples in the training set. For a deep network of fixed size, the memory cost of training is constant with respect to training set size (except for the memory needed to store the examples themselves) and the runtime of a single pass through the training set is linear in training set size. These asymptotic results meant that kernelized SVMs dominated while datasets were small, but deep models currently dominate now that datasets are large.

### 5.8.3 Other Simple Supervised Learning Algorithms

We have already briefly encountered another non-probabilistic supervised learning algorithm, nearest neighbor regression. More generally, $k$-nearest neighbors is a family of techniques that can be used for classification or regression. As a non-parametric learning algorithm, there are no parameters. In fact, there is not even

really a training stage or learning process. Instead, at test time, when we want to produce an output $y$ for a new test input $\boldsymbol{x}$, we find the $k$ nearest neighbors to $\boldsymbol{x}$ in the training data $\boldsymbol{X}$. We then return the average of the corresponding $y$ values in the training set. This works for esentially any kind of supervised learning where we can define an average over $y$ values. In the case of classification, we can average over one-hot code vectors $\boldsymbol{c}$ with $c_y = 1$ and $c_i = 0$ for all other values of $i$. We can then interpret the average over these one-hot codes as giving a probability distribution over classes. As a non-parametric learning algorithm, $k$-nearest neighbors has unlimited capacity and will eventually converge to the Bayes error given a large enough training set if $k$ is properly reduced as the number of examples is increased. However, it may perform very badly on small, finite training sets. One weakness of $k$-nearest neighbors is that it cannot learn that one feature is more discriminative than another. For example, imagine we have a regression task with $\boldsymbol{x} \in \mathbb{R}^{100}$ drawn from an isotropic Gaussian distribution, but only a single variable $x_1$ is relevant to the output. Suppose further that this feature simply encodes the output directly, i.e. that $y = x_1$ in all cases. Nearest neighbor regression will not be able to detect this simple pattern. The nearest neighbor of most points $\boldsymbol{x}$ will be determined by the large number of features $x_2$ through $x_{100}$, not by the lone feature $x_1$. Thus the output on small training sets will essentially be random.

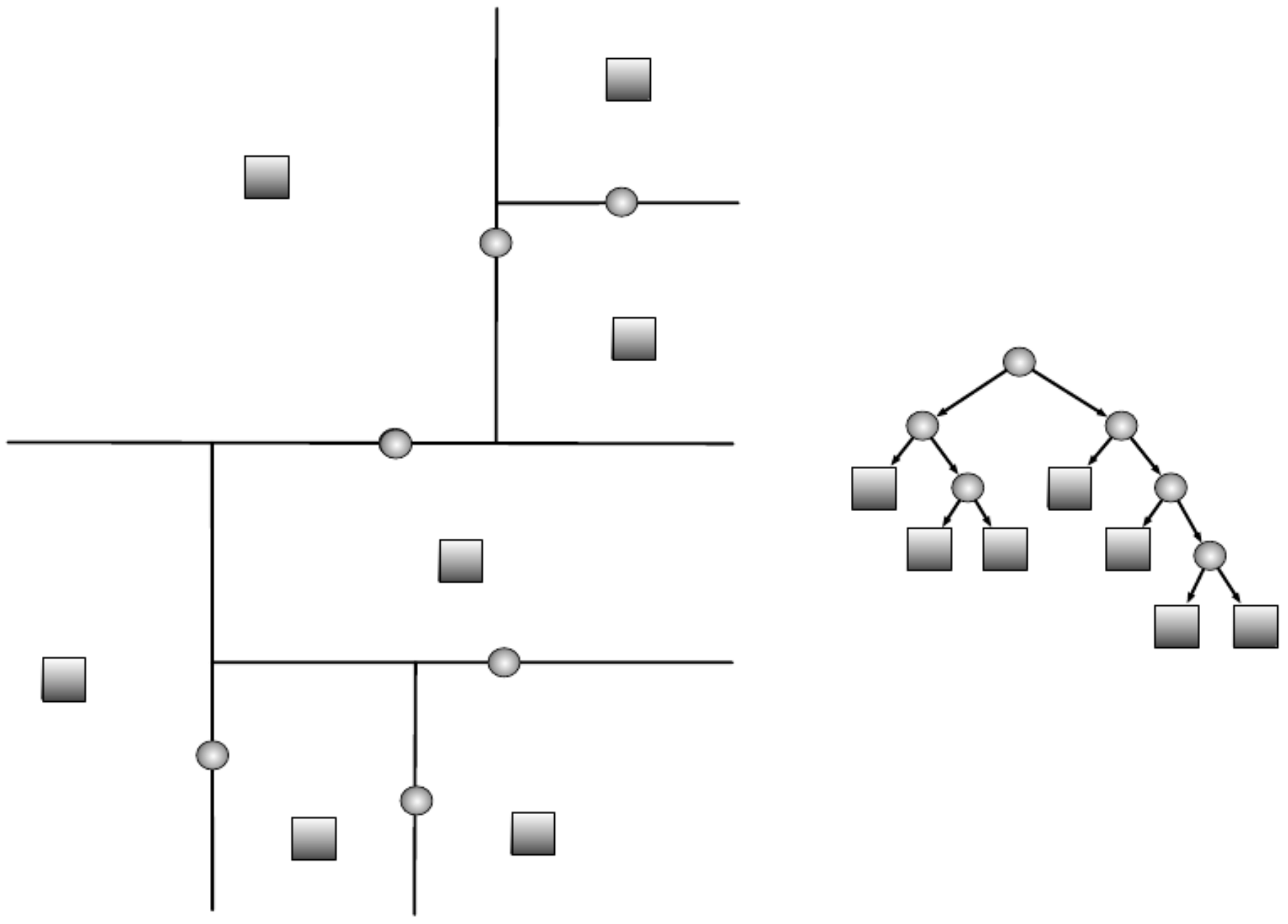

Figure 5.7: Decision tree (right) and how it cuts the input space into regions, with a constant output in each region (left). Each node of the tree (circle or square) is associated with a region (the entire space for the root node, with the empty string identifier). Internal nodes (circles) split their region in two, via an axis-aligned cut (occasionally, some decision trees use more complicated cuts than shown in this example). Leaf nodes (squares) are associated with a model output, typically set to the average target output for the training examples that fall in the corresponding region. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by appending a bit to its parent identifier (0=choose left or top, 1=choose right or bottom). Note that the result is a piecewise-constant function, and note how the number of regions (pieces) cannot be greater than the number of examples, hence it is not possible to learn a function that has more local maxima than the number of training examples.

Another type of learning algorithm that also breaks the input space into regions and has separate parameters for each region is the *decision tree* (Breiman *et al.*, 1984) and its many variants. As shown in Fig. 5.7, each node of the decision tree is associated with a region in the input space, and internal nodes break that region into one sub-region for each child of the node (typically using an axis-aligned cut). Space is thus sub-divided into non-overlapping regions, with a one-to-one correspondence between leaf nodes and input regions. Each leaf node usally maps every point in its input region to the same output. Decision trees

are usually trained with specialized algorithms that are beyond the scope of this book. The learning algorithm can be considered non-parametric if it is allowed to learn a tree of arbitrary size, though decision trees are usually regularized with size constraints that turn them into parametric models in practice. Note that decision trees as they are typically used, with axis-aligned splits and constant outputs within each node, struggle to solve some problems that are easy even for logistic regression. For example, if we have a two-class problem and the positive class occurs wherever $x_2 > x_1$, the decision boundary is not axis-aligned. The decision tree will thus need to approximate the decision boundary with many nodes, implementing a step function that constantly walks back and forth across the true decision function with axis-aligned steps.

As we have seen, nearest neighbor predictors and decision trees have many limitations. Nonetheless, they are useful learning algorithms when computational resources are constrained. We can also build intuition for more sophisticated learning algorithms by thinking about the similarities and differences between sophisticated algorithms and $k$-NN or decision tree baselines.

See Murphy (2012); Bishop (2006); Hastie *et al.* (2001) or other machine learning textbooks for more material on traditional supervised learning algorithms.

## 5.9 Unsupervised Learning Algorithms

Recall from Section 5.1.3 that unsupervised algorithms are those that experience only "features" but not a supervision signal. The distinction between supervised and unsupervised algorithms is not formally and rigidly defined because there is no objective test for distinguishing whether a value is a feature or a target provided by a supervisor. Informally, unsupervised learning refers to most attempts to extract information from a distribution that do not require human labor to annotate examples. The term is usually associated with density estimation, learning to draw samples from a distribution, learning to denoise data from some distribution, finding a manifold that the data lies near, or clustering the data into groups of related examples.

**Learning a representation of data** A classic unsupervised learning task is to find the 'best' representation of the data. By 'best' we can mean different things, but generally speaking we are looking for a representation that preserves as much information about $x$ as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than $x$ itself.

There are multiple ways of defining a *simpler* representation, some of the most common include lower dimensional representations, sparse representations and independent representations. Low-dimensional representations attempt to

compress as much information about $x$ as possible in a smaller representation. Sparse representations generally embed the dataset into a high-dimensional representation[5] where the number of non-zero entries is small. This results in an overall structure of the representation that tends to distribute data along the axes of the representation space. Independent representations attempt to *disentangle* the sources of variation underlying the data distribution such that the dimensions of the representation are statistically independent.

Of course these three criteria are certainly not mutually exclusive. Low-dimensional representations often yield elements that have fewer or weaker dependencies than the original high-dimensional data. This is because one way to reduce the size of a representation is to find and remove redundancies. Identifying and removing more redundancy allows the dimensionality reduction algorithm to achieve more compression while discarding less information.

The notion of representation is one of the central themes of deep learning and therefore one of the central themes in this book. Chapter 16 discusses some of the qualities we would like in our learned representations, along with specific representation learning algorithms more powerful than the simple one presented next, Principal Components Analysis.

### 5.9.1 Principal Components Analysis

In the remainder of this section we will consider one of the most widely used unsupervised learning methods: Principle Components Analysis (PCA). PCA is an orthogonal, linear transformation of the data that projects it into a representation where the elements are uncorrelated (shown in Figure 5.8).

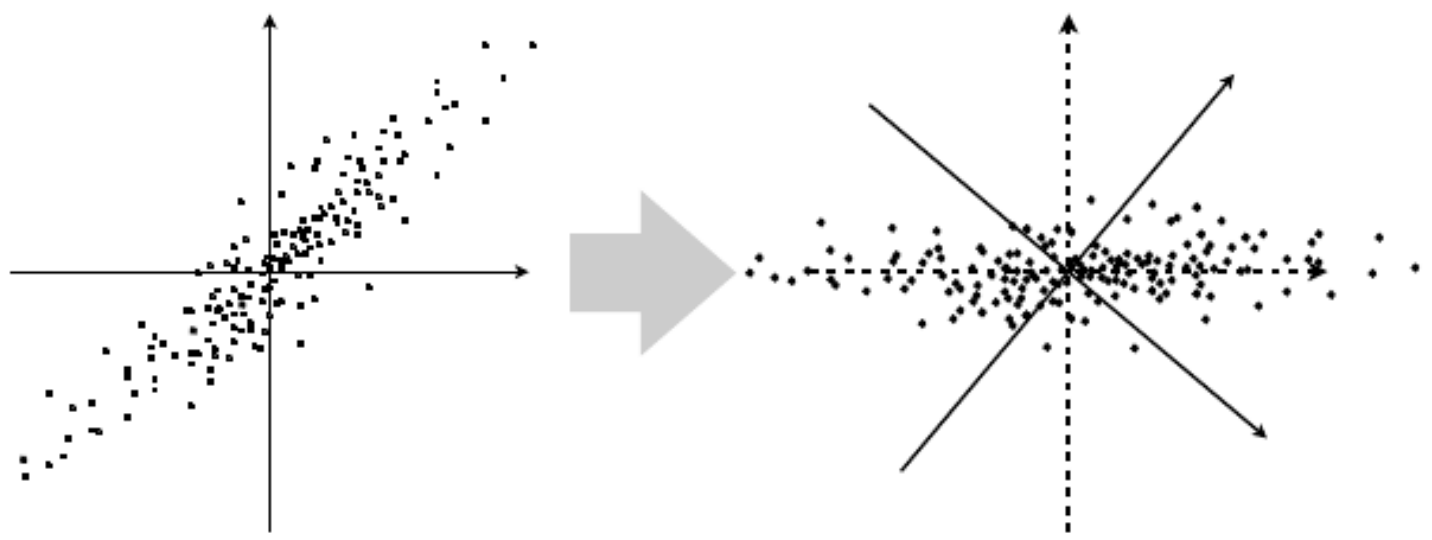

Figure 5.8: Illustration of the data representation learned via PCA.

In section 2.12, we saw that we could learn a one-dimensional representation

sparse representations often use over-complete representations: the representation dimension is greater than the original dimensionality of the data.

that best reconstructs the original data (in the sense of mean squared error) and that this representation actually corresponds to the first principal component of the data. Thus we can use PCA as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible (again, as measured by least-squares reconstruction error). In the following, we will take a look at other properties of the PCA representation. Specifically, we will study how the PCA representation can be said to decorrelate the original data representation $\boldsymbol{X}$.

Let us consider the $n \times m$-dimensional design matrix $\boldsymbol{X}$. We will assume that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = \mathbf{0}$. If this is not the case, the data can easily be centered (mean removed). The unbiased sample covariance matrix associated with $\boldsymbol{X}$ is given by:

$$\mathrm{Var}[\boldsymbol{x}] = \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} \tag{5.40}$$

One important aspect of PCA is that it finds a representation (through linear transformation) $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$ where $\mathrm{Var}[\boldsymbol{z}]$ is diagonal. To do this, we will make use of the singular value decomposition (SVD) of $\boldsymbol{X}$: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$, where $\boldsymbol{\Sigma}$ is an $n \times m$-dimensional rectangular diagonal matrix with the singular values of $\boldsymbol{X}$ on the main diagonal, $\boldsymbol{U}$ is an $n \times n$ matrix whose columns are orthonormal (i.e. unit length and orthogonal) and $\boldsymbol{W}$ is an $m \times m$ matrix also composed of orthonormal column vectors.

Using the SVD of $\boldsymbol{X}$, we can re-express the variance of $\boldsymbol{X}$ as:

$$\begin{aligned}
\mathrm{Var}[\boldsymbol{x}] &= \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} && (5.41)\\
&= \frac{1}{n-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.42)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.43)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^2 \boldsymbol{W}^\top, && (5.44)
\end{aligned}$$

where we use the orthonormality of $\boldsymbol{U}$ ($\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$) and define $\boldsymbol{\Sigma}^2$ as an $m \times m$-dimensional diagonal matrix with the squares of the singular values of $\boldsymbol{X}$ on the diagonal, i.e. the $i$th diagonal elements is given by $\Sigma_{i,i}^2$. This shows that if we

take $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, we can ensure that the covariance of $\boldsymbol{z}$ is diagonal as required.

$$\begin{aligned}
\mathrm{Var}[\boldsymbol{z}] &= \frac{1}{n-1}\boldsymbol{Z}^\top\boldsymbol{Z} && (5.45)\\
&= \frac{1}{n-1}\boldsymbol{W}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{W} && (5.46)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{W}^\top\boldsymbol{\Sigma}^2\boldsymbol{W}\boldsymbol{W}^\top && (5.47)\\
&= \frac{1}{n-1}\boldsymbol{\Sigma}^2 && (5.48)
\end{aligned}$$

Similar to our analysis of the variance of $\boldsymbol{X}$ above, we exploit the orthonormality of $\boldsymbol{W}$ (i.e., $\boldsymbol{W}^\top\boldsymbol{W} = \boldsymbol{I}$). Our use of SVD to solve for the PCA components of $\boldsymbol{X}$ (i.e. elements of $\boldsymbol{z}$) reveals an interesting connection to the eigen-decomposition of a matrix related to $\boldsymbol{X}$. Specifically, the columns of $\boldsymbol{W}$ are the eigenvectors of the $n \times n$-dimensional matrix $\boldsymbol{X}^\top\boldsymbol{X}$.

The above analysis shows that when we project the data $\boldsymbol{x}$ to $\boldsymbol{z}$, via the linear transformation $\boldsymbol{W}$, the resulting representation has a diagonal covariance matrix (as given by $\boldsymbol{\Sigma}^2$) which immediately implies that the individual elements of $\boldsymbol{z}$ are mutually uncorrelated.

This ability of PCA to transform data into a representation where the elements are mutually uncorrelated is a very important property of PCA. It is a simple example of a representation that attempt to *disentangle the unknown factors of variation* underlying the data. In the case of PCA, this *disentangling* takes the form of finding a rotation of the input space (mediated via the transformation $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$, as illustrated in Fig. 5.8. While correlation is an important category of dependency between elements of the data, we are also interested in learning representations that *disentangle* more complicated forms of feature dependencies. For this, we will need more than what can be done with a simple linear transformation. These issues are discussed below in Sec. 5.12 and later in detail in Chapter 16.

## 5.10 Weakly Supervised Learning

Weakly supervised learning is another class of learning methods that stands between supervised and unsupervised learning. It refers to a setting where the datasets consists of $(\boldsymbol{x}, y)$ pairs, as in supervised learning, but where the labels $y$ are either unreliably present (i.e. with missing values) or noisy (i.e. where the label given is not the true label).

Methods for working with weakly labeled data have recently grown in importance due to the—largely untapped—potential for using large quantities of readily

available weakly labeled data in a transfer learning paradigm to help solve problems where large, clean datasets are hard to come-by. The Internet has become a major source of this kind of noisy data.

For example, although we would like to train a computer vision system with labels indicating the presence and location of every object (and which pixels correspond to which object) in every image, such labeling is very human-labor intensive. Instead, we want to take advantage of images for which only the main object is identified, like the ImageNet dataset (Deng *et al.*, 2009), or worse, of video for which some general and high-level semantic spoken caption is approximately temporally aligned with the corresponding frames of the video, like the DVS data (Descriptive Video service) which has recently been released (Torabi *et al.*, 2015).

## 5.11 Building a Machine Learning Algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine a specification of a dataset, a cost function, an optimization procedure and a model.

For example, the linear regression algorithm combines a dataset consisting of $\boldsymbol{X}$ and $\boldsymbol{y}$, the cost function

$$J(\boldsymbol{w}, b) = -\mathbb{E}_{,y\sim\hat{p}} \quad p_{\text{model}}(y \mid \boldsymbol{x}),$$

and the model specification $p_{\text{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{x}^\top \boldsymbol{w} + b, 1)$. Typically we then observe that $J$ simplifies to the mean squared error and we can choose to optimize this in closed form by solving the normal equations with the Moore-Penrose pseudo-inverse.

By realizing that we can modify any of these components, we can obtain a very wide variety of algorithms.

The cost function typically includes at least one term that causes the learning process to perform statistical estimation. The most common cost function is the negative log-likelihood, so that minimizing the cost function causes maxmimum likelihood estimation. This main term of the cost function often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{,y\sim\hat{p}} \quad L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

where $y$ is the per-example loss $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$. We can design many different cost functions just by taking the expectation across the training set of different per-example loss functions.

The cost function may also include additional terms, such as regularization terms. For example, we can add weight decay to the linear regression cost function to obtain

$$J(\boldsymbol{w}, b) = \lambda ||\boldsymbol{w}||_2^2 - \mathbb{E}_{,y\sim p} \quad p_{\text{model}}(y \mid \boldsymbol{x}).$$

This still allows closed-form optimization.

A common and simple way to optimize such additive objective functions is *stochastic gradient descent* or SGD. SGD is a form of gradient descent where we do not use the true gradient but instead a stochastic estimator of the gradient (whose expected value should equal the true gradient). In the machine learning context, the estimator is formed by considering one example or a *minibatch* containing a few examples at a time, randomly chosen, and only using the gradient for these examples to update the parameters. For example, with the above definitions of example-wise loss $L$, the parameters could be iteratively updated as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \frac{\partial L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{5.49}$$

where $i$ is the index of a randomly chosen training example. See Sec. 8.3.2 for a deeper treatment of SGD from an optimization perspective.

If we change the model to be non-linear, then most cost functions can no longer be optimized in close form. This requires us to choose an iterative numerical optimization procedure, such as gradient descent.

This recipe supports both supervised and unsupervised learning. The linear regression example shows how to support supervised learning. Unsupervised learning can be supported by defining a dataset that contains only $\boldsymbol{X}$ and providing an appropriate unsupervised cost and model. For example, we can obtain the first PCA vector by specifying that our loss function is

$$J(\boldsymbol{w}) = \mathbb{E}_{\sim p} \quad ||\boldsymbol{x} - r(x; \boldsymbol{w})||_2^2$$

while our model is defined to have $\boldsymbol{w}$ with norm one and reconstruction function $r(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{w}\boldsymbol{x}$.

In some cases, the cost function may be a function that we cannot actually evaluate, for computational reasons. In these cases, we can still approximately minimize it using iterative numerical optimization so long as we have some way of approximating its gradients.

Most machine learning algorithms make use of this recipe, though it may not immediately be obvious. If a machine learning algorithm seems especially unique or hand-designed, it can usually be understood as using a special-case optimizer. Some models such as decision trees or $k$-means require special-case optimizers because their cost functions have flat regions that make them inappropriate for minimization by gradient-based optimizers. Recognizing that most

machine learning algorithms can be described using this recipe helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications.

## 5.12 The Curse of Dimensionality and Statistical Limitations of Local Generalization

Many of the most basic difficulties—both computational and statistical—in machine learning arise from the use of a large number of input variables. Some tasks become exponentially more difficult as this number increases. These challenges are fundamental to the field, and many of the ideas in deep learning are designed specifically to respond to these challenges.

### 5.12.1 The Curse of Dimensionality

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. This phenomenon is known as the *curse of dimensionality*. Of particular concern is that the number of possible distinct configurations of the variables of interest increases exponentially as the dimensionality increases.

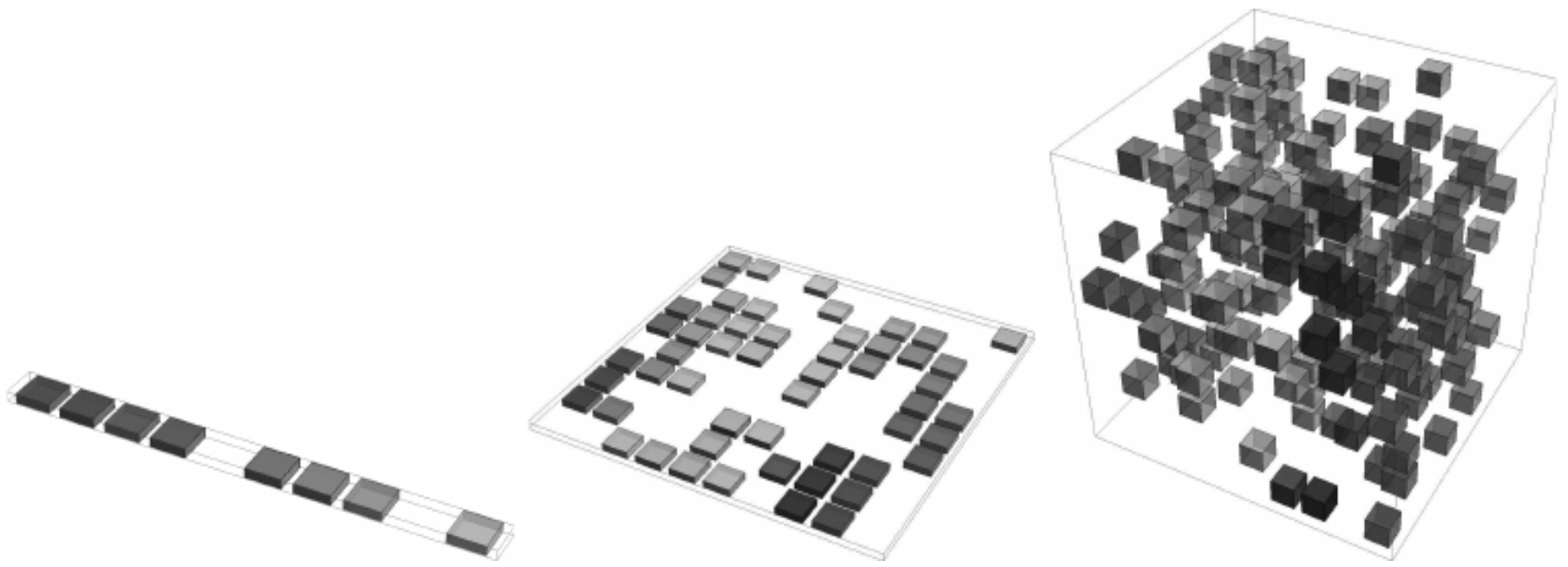

Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. In the figure we first consider one-dimensional data (left), i.e., one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (cells, in the figure), learning algorithms can easily generalize correctly, i.e., estimate the value of the target function within each region (and possibly interpolate between neighboring regions). With 2 dimensions (center), but still caring to distinguish 10 different values of each variable, we need to keep track of up to 10×10=100 regions, and we need at least that many examples to cover all those regions. With 3 dimensions (right) this grows to 10 = 1000 regions and at least that many examples. For $d$ dimensions and $V$ values to be distinguished along each axis, it looks like we need $O(V)$ regions and examples. This is an instance of the curse of dimensionality. However, note that if the data distribution is concentrated on a smaller set of regions, we may actually not need to cover all the possible regions, only those where probability is non-negligible. *Figure graciously provided by, and with authorization from, Nicolas Chapados.*

The curse of dimensionality rears its ugly head in many places in computer science, and especially so in machine learning.

One challenge posed by the curse of dimensionality is a statistical challenge. As illustrated in Figure 5.9, a statistical challenge arises because the number of possible configurations of the variables of interest is much larger than the number of training examples. To understand the issue, let us consider that the input space is organized into a grid, like in the figure. In low dimensions we can describe this space with a low number of grid cells that are mostly occupied by the data. A reasonable way to formulate the smoothness prior is with the assumption of local constancy around the training examples. In the context of our grid cells example, a straightforward way to implement local constancy is to assume that our learner should provide the same answer to two examples falling in the same grid cell. It is a form of local constancy assumption, a notion that we develop further in the next section. When generalizing to a new data point, we can usually tell what to do simply by inspecting the training examples that lie in the same cell as the new input. For example, if estimating the probability density at some

point $\boldsymbol{x}$, we can just return the number of training examples in the same unit volume cell as $\boldsymbol{x}$, divided by the total number of training examples. If we wish to classify an example, we can return the most common class of training examples in the same cell. If we are doing regression we can average the target values observed over the examples in that cell. But what about the cells for which we have seen no example? Because in high-dimensional spaces the number of configurations is going to be huge, much larger than our number of examples, most configurations will have no training example associated with it. How could we possibly say something meaningful about these new configurations? A simple answer is to extend the local constancy assumption into a smoothness assumption, as explained next.

### 5.12.2 Local Constancy and Smoothness Regularization

As argued previously, and especially in high-dimensional spaces (because of the curse of dimensionality introduced above), machine learning algorithms need priors, i.e., a preference over the space of solutions, in order to generalize to new configurations not seen in the training set. The specification of these preferences includes the choice of model family, as well as any regularizer or other aspects of the algorithm that influence the final outcome of training. We consider here a particular family of preferences which underlie many classical machine learning algorithms, and which we call the *smoothness prior* or the *local constancy prior*. We find that when the function to be learned has many ups and downs, and this is typically the case in high-dimensional spaces because of the curse of dimensionality (see above), then the smoothness prior is insufficient to achieve good generalization. We argue that more assumptions are needed in order to generalize better, in this setting. Deep learning algorithms typically introduce such additional assumptions. This starts with the classical multi-layer neural networks studied in the next chapter (Chapter 6), and in Chapter 16 we return to the advantages that representation learning, distributed representations and depth can bring towards generalization, even in high-dimensional spaces.

There are many different ways to implicitly or explicitly express a prior belief that the learned function should be smooth or locally constant. All of these different methods are designed to encourage the learning process to learn a function $f^*$ that satisfies the condition

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon) \tag{5.50}$$

for most configurations $\boldsymbol{x}$ and small change $\epsilon$. In other words, if we know a good answer for an input $\boldsymbol{x}$ (for example, if $\boldsymbol{x}$ is a labeled training example) then that answer is probably good in the neighborhood of $\boldsymbol{x}$. If we have several good answers in some neighborhood we would combine them (by some form of

averaging or interpolation) to produce an answer that agrees with as many of them as much as possible.

An extreme example of the local constancy approach is the $k$-nearest neighbors family of learning algorithms. These predictors are literally constant over each region $R$ containing all the points $\boldsymbol{x}$ that have the same set of $k$ nearest neighbors in the training set. Note that for $k = 1$, the number of distinguishable regions cannot be more than the number of training examples.

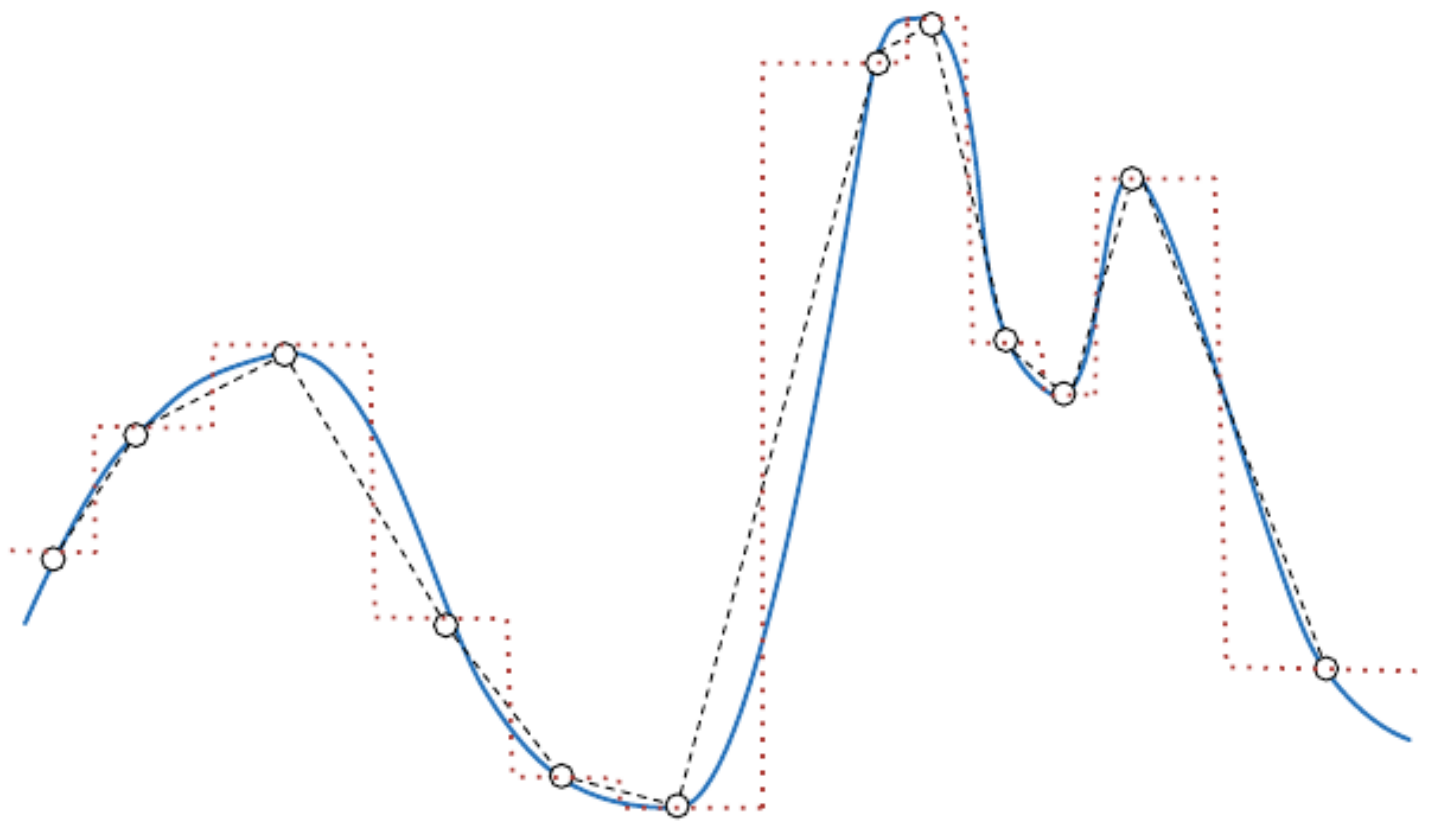

Figure 5.10: Illustration of interpolation and kernel-based methods, which construct a smooth function by interpolating in various ways between the training examples (circles), which act like knot points controlling the shape of the implicit regions that separate them as well as the values to output within each region. Depending on the type of kernel, one obtains a piecewise constant (histogram-like, in dotted red), a piecewise linear (dashed black) or a smoother kernel (bold blue). The underlying assumption is that the target function is as smooth or locally as constant as possible. This assumption allows to *generalize locally*, i.e., to extend the answer known at some point $\boldsymbol{x}$ to nearby points, and this works very well so long as, like in the figure, there are enough examples to cover most of the ups and downs of the target function.

To obtain even more smoothness, we can *interpolate* between neighboring training examples, as illustrated in Figure 5.10. For example, *non-parametric kernel density estimation methods* and *kernel regression* methods construct a learned function $f$ of the form of Eq. 5.38 for classification or regression, or alternatively, e.g., in the Parzen regression estimator, of the form

$$f(\boldsymbol{x}) = b + \sum_{i=1}^{n} \alpha_i \frac{k(\boldsymbol{x}, \boldsymbol{x}^{(i)})}{\sum_{j=1}^{n} k(\boldsymbol{x}, x^{(j)})}.$$

If the kernel function $k$ is discrete (e.g. 0 or 1), then this can include the above

cases where $f$ is piecewise constant and a discrete set of regions (no more than one per training example) can be distinguished. However, better results can often be obtained if $k$ is smooth, e.g., the Gaussian kernel from Eq. 5.39. With $k$ a *local kernel* (Bengio *et al.*, 2006b; Bengio and LeCun, 2007b; Bengio, 2009)[6] , we can think of each $\boldsymbol{x}^{(i)}$ as a *template* and the kernel function as a *similarity function* that *matches a template and a test example.*

With the Gaussian kernel, we do not have a piecewise constant function but instead a continuous and smooth function. In fact, the choice of $k$ can be shown to correspond to a particular form of smoothness. Equivalently, we can think of many of these estimators as the result of smoothing the *empirical distribution* by convolving it with a function associated with the kernel, e.g., the Gaussian kernel density estimator is the empirical distribution convolved with the Gaussian density.

Although in classical non-parametric estimators the $\alpha_i$ of Eq. 5.38 are fixed (e.g. to $1/n$ for density estimation and to $y^{(i)}$ for supervised learning from examples $(\boldsymbol{x}^{(i)}, y^{(i)})$), they can be optimized, and this is the basis of more modern non-parametric kernel methods (Schölkopf and Smola, 2002) such as the Support Vector Machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995) (see also Section 5.8.2).

However, as illustrated in Figure 5.10, even though these smooth kernel methods generalize better, the main thing that has changed is that one can basically interpolate between the neighboring examples, in some space associated with the kernel. One can then think of the training examples as control knots which locally specify the shape of each region and the associated output.

Decision trees also suffer from the limitations of exclusively smoothness-based learning. Because each example only informs the region in which it falls about which output to produce, *one cannot have more regions than training examples.* If the target function can be well approximated by cutting the input space into $N$ regions (with a different answer in each region), then at least $N$ examples are needed (and a multiple of $N$ is needed to achieve some level of statistical confidence in the predicted output). All this is also true if the tree is used for density estimation (the output is simply an estimate of the density within the region, which can be obtained by the ratio of the number of training examples in the region by the region volume) or whether a non-constant (e.g. linear) predictor is associated with each leaf (then more examples are needed within each leaf node, but the relationship between number of regions and number of examples remains linear). We examine below how this may hurt the generalization ability of decision trees and other learning algorithms that are based only on the smoothness or local constancy priors, when the input is high-dimensional, i.e., because of the curse of

i.e., with ( ) large when = and decreasing as they get farther apart

dimensionality.

In all cases, the smoothness assumption (Eq. 5.50) allows the learner to *generalize locally*. Since we assume that the target function obeys $f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon)$ most of the time for small $\epsilon$, we can generalize the empirical distribution (or the $(\boldsymbol{x}, y)$ training pairs) to the neighborhood of the training examples. If $(\boldsymbol{x}^{(i)}, y^{(i)})$ is a supervised (input,target) training example, then we expect $f^*(\boldsymbol{x}^{(i)}) \approx y^{(i)}$, and therefore if $\boldsymbol{x}$ is a near neighbor of $\boldsymbol{x}^{(i)}$, we expect that $f^*(\boldsymbol{x}) \approx y^{(i)}$. By considering more neighbors, we can obtain better generalization, by better executing the smoothness assumption.

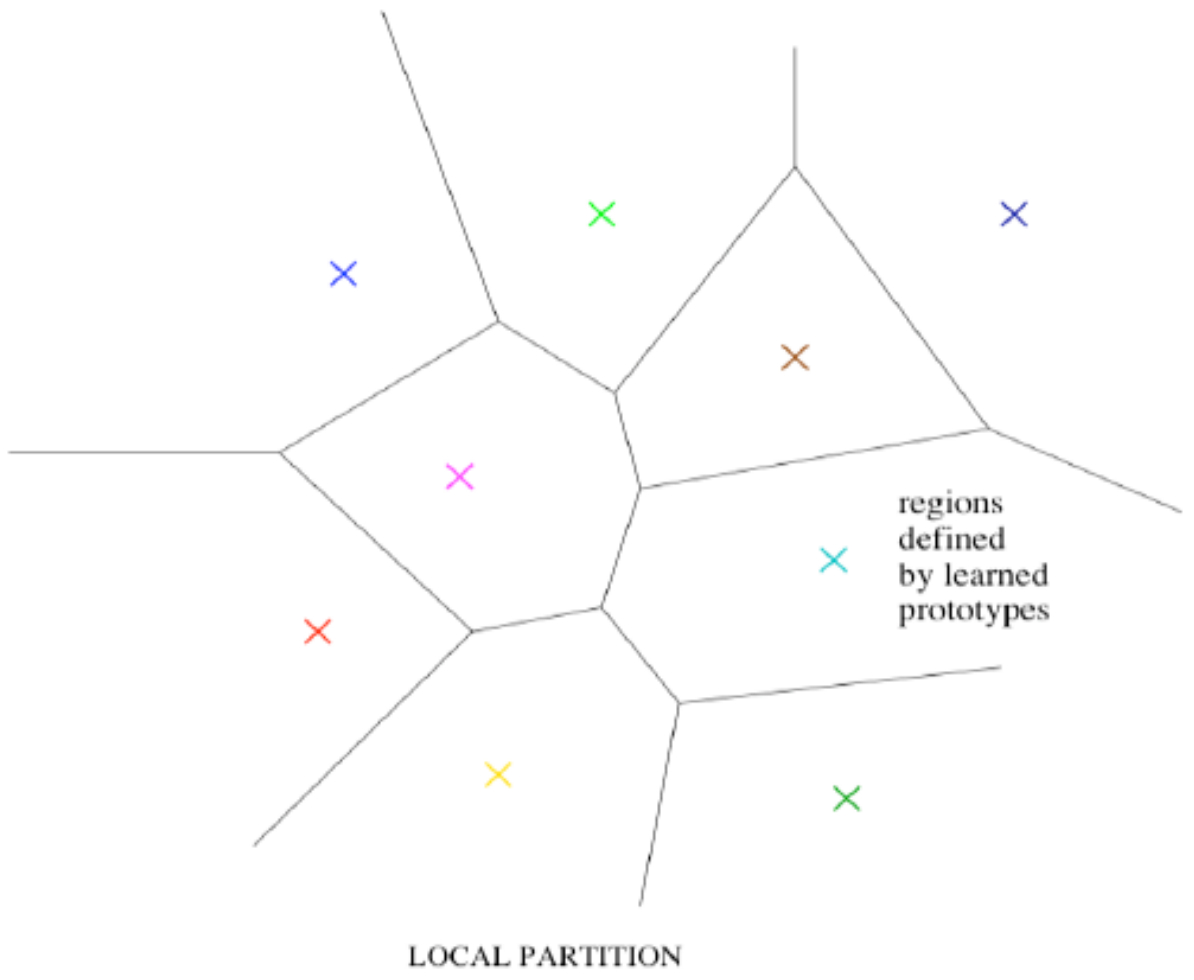


Figure 5.11: Illustration of how non-parametric learning algorithms that exploit only the smoothness or local constancy priors typically break up the input space into regions, with examples in those regions being used both to define the region boundaries and what the output should be within each region. The figure shows the case of clustering or 1-nearest-neighbor classifiers, for which each training example (cross of a different color) defines a region or a template (here, the different regions form a Voronoi tessellation). The number of these contiguous regions cannot grow faster than the number of training examples. In the case of a decision tree, the regions are recursively obtained by axis-aligned cuts within existing regions, but for these and for kernel machines with a local kernel (such as the Gaussian kernel), the same property holds, and generalization can only be *local*: each training example only informs the learner about how to generalize in some neighborhood around it.

In general, to distinguish $O(N)$ regions in input space, all of these methods require $O(N)$ examples (and typically there are $O(N)$ parameters associated with the $O(N)$ regions). This is illustrated in Figure 5.11 in the case of a nearest-

neighbor or clustering scenario, where each training example can be used to define one region. Is there a way to represent a complex function that has many more regions to be distinguished than the number of training examples? Clearly, assuming only smoothness of the underlying function will not allow a learner to do that. For example, imagine that the target function is a kind of checkerboard, i.e., with a lot of variations, but a simple structure to them, and imagine that the number of training examples is substantially less than the number of black and white regions. Based on local generalization and the smoothness or local constancy prior, we could get the correct answer within a constant-colour region, but we could not correctly predict the checkerboard pattern. The only thing that an example tells us, with this prior, is that nearby points should have the same colour, and the only way to get the checkerboard right is to cover all of its cells with at least one example.

The smoothness assumption and the associated non-parametric learning algorithms work extremely well *so long as there are enough examples to cover most of the ups and downs of the target function.* This is generally true when the function to be learned is smooth enough, which is typically the case for low-dimensional data. And if it is not very smooth (we want to distinguish a huge number of regions compared to the number of examples), is there any hope to generalize well?

Both of these questions are answered positively in Chapter 16. The key insight is that a very large number of regions, e.g., $O(2^N)$, can be defined with $O(N)$ examples, so long as we introduce some dependencies between the regions via additional priors about the underlying data generating distribution. In this way, we can actually generalize non-locally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006c). A neural network can actually learn a checkerboard pattern. Similarly, some recurrent neural networks can learn the $n$-bit parity (at least for some not too large values of $n$). Of course we could also solve the checkerboard task by making a much stronger assumption, e.g., that the target function is periodic. However, neural networks can generalize to a much wider variety of structures, and indeed our AI tasks have structure that is much too complex to be limited to periodicity, so we want learning algorithms that embody more general-purpose assumptions. The core idea in deep learning is that we assume that the data was generated by the *composition of factors* or features, potentially at multiple levels in a hierarchy. These apparently mild assumptions allow an exponential gain in the relationship between the number of examples and the number of regions that can be distinguished, as discussed in Chapter 16. Priors that are based on compositionality, such as arising from learning distributed representations and from a deep composition of representations, can give an exponential advantage, which can hopefully counter the exponential curse of dimensionality. Chapter 16 dis-

cusses these questions from the angle of representation learning and the objective of *disentangling the underlying factors of variation.*

### 5.12.3 Manifold Learning and the Curse of Dimensionality

We consider here a particular type of machine learning task called *manifold learning.* Although they have been introduced to reduce the curse of dimensionality. We will argue that they allow one to visualize and highlight how the smoothness prior is not sufficient to generalize in high-dimensional spaces. Chapter 17 is devoted to the manifold perspective on representation learning and goes in much greater details in this topic as well as in actual manifold learning algorithms based on neural networks.

A *manifold* is a connected region, i.e., a set of points, associated with a neighborhood around each point, which makes it locally look like a Euclidean space. The notion of neighbor implies the existence of transformations that can be applied to *move on the manifold* from one position to a neighboring one. Although there is a formal mathematical meaning to this term, in machine learning it tends to be used more loosely to talk about a connected set of points that can be well approximated by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space. Each dimension corresponds to a local direction of variation, i.e., moving along the manifold in some direction. The manifolds we talk about in machine learning are subsets of points, also called a submanifold, of the embedding space (which is also a manifold).

**Manifold learning** algorithms assume that the data distribution is concentrated in a small number of dimensions, i.e., that the set of high-probability configurations can be approximated by a low-dimensional manifold. Figure 5.8 (left) illustrates a distribution that is concentrated near a linear manifold (the manifold is along a 1-dimensional straight line). Manifold learning was introduced in the case of continuous-valued data and the unsupervised learning setting, although this probability concentration idea can be generalized to both discrete data and the supervised learning setting: the key assumption remains that probability mass is highly concentrated.

Is this assumption reasonable? It seems to be true for almost all of the AI tasks such as those involving images, sounds and text. To be convinced of this we will invoke (a) the observation that probability mass is concentrated and (b) the observed objects can generally be transformed into other plausible configurations via some small changes (which indicates a notion of direction of variation while staying on the "manifold"). For (a), consider that if the assumption of probability concentration was false, then sampling uniformly at random from in the set of all configurations (e.g., uniformly in $\mathbb{R}^n$) should produce probable (data-like) configurations reasonably often. But this is not what we observe in practice. For

example, generate pixel configurations for an image by independently picking the grey level (or a binary 0 versus 1) for each pixel. What kind of images do you get? You get "white noise" images, that look like the old television sets when no signal is coming in, as illustrated in Figure 5.12 (left). What is the probability that you would obtain something that looks like a natural image, with this procedure? Almost zero, because the set of probable configurations (near the manifold of natural images) occupies a very small volume out of the total set of pixel configurations. Similarly, if you generate a document by picking letters randomly, what is the probability that you will get a meaningful English-language text? Almost zero, again, because most of the long sequences of letters do not correspond to a natural language sequence: the distribution of natural language sequences occupies a very small volume in the total space of sequences of letters.

Figure 5.12: Sampling images uniformly at random, e.g., by randomly picking each pixel according to a uniform distribution, gives rise to white noise images such as illustrated on the left. Although there is a non-zero probability to generate something that looks like a natural image (like those on the right), that probability is exponentially tiny (exponential in the number of pixels!). This suggests that natural images are very "special", and that they occupy a tiny volume of the space of images.

The above thought experiments, which are in agreement with the many experimental results of the manifold learning literature, e.g. (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003;

Weinberger and Saul, 2004), clearly establish that for a large class of datasets of interest in AI, the *manifold hypothesis* is true: the data generating distribution concentrates in a small number of dimensions, as in the cartoon of Figure 17.4, from Chapter 17. That chapter explores the relationships between representation learning and manifold learning: if the data distribution concentrates on a smaller number of dimensions, then we can think of these dimensions as natural coordinates for the data, and we can think of representation learning algorithms as ways to map the input space to a new and often lower-dimensional space which captures the leading dimensions of variation present in the data as axes or dimensions of the representation.

An initial hope of early work on manifold learning (Roweis and Saul, 2000; Tenenbaum *et al.*, 2000) was to *reduce the effect of the curse of dimensionality*, by first reducing the data to a lower dimensional representation (e.g. mapping $(x_1, x_2)$ to $z_1$ in Figure 5.8 (right)), and then applying ordinary machine learning in that transformed space. This dimensionality reduction can be achieved by learning a transformation (generally non-linear, unlike with PCA introduced in Section 5.9.1) of the data that is *invertible for most training examples*, i.e., that keeps the information in the input example. It is only possible to reconstruct input examples from their low-dimensional representation because they lie on a lower-dimensional manifold, of course. This is basically how *auto-encoders* (Chapter 15) are trained.

The hope was that by non-linearly projecting the data in a new space of lower dimension, we would reduce the curse of dimensionality by only looking at relevant dimensions, i.e., a smaller set of regions of interest (cells, in Figure 5.9). This can indeed be the case, however, as discussed in Chapter 17, the manifolds can be highly curved and have a very large number of twists, requiring still a very large number of regions to be distinguished (every up and down of each corner of the manifold). Even if we were to reduce the dimensionality of an input from a 10,000-dimensional vector describing a $100 \times 100$ pixel binary image to a 100-dimensional vector, $2^{100}$ is still too large to hope to cover with a training set. Likewise, the space is too large to cover using only local generalization from training examples—the smoothness prior may be useful but is not sufficient. It may also be that although the effective dimensionality of the data could be small, some examples could fall outside of the main manifold and that we do not want to systematically lose that information. A *sparse representation* then becomes a possible way to represent data that is mostly low-dimensional, although occasionally occupying more dimensions. This can be achieved with a high-dimensional representation whose elements are 0 most of the time. We can see that the effective dimension (the number of non-zeros) then can change depending on where we are in input space, which can be useful. Sparse representations are discussed in Section 15.8.

The next part of the book introduces specific deep learning algorithms that aim at discovering representations that are useful for some task, i.e., trying to extract the directions of variations that matter for the task of interest, often in a supervised setting. The last part of the book concentrates more on unsupervised representation learning algorithms, which attempt to capture all of the directions of variation that are salient in the data distribution.

# Part II

# Deep Networks: Modern Practices

This part of the book summarizes the state of modern deep learning as it is used to solve practical applications.

Deep learning has a long history and many aspirations. Several approaches have been proposed that have yet to entirely bear fruit. Several ambitious goals have yet to be realized. These less-developed branches of deep learning appear in the final part of the book.

This part focuses only on those approaches that are essentially working technologies that are already used heavily in industry.

Modern deep learning provides a very powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large model and dataset of labeled training examples. Other tasks, that can not be described as associating one vector to another, or that are difficult enough that to do them a person would require time to think and reflect, remain beyond the scope of deep learning for now.

This part of the book describes the core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. Our description includes details such as, how to efficiently model specific kinds of inputs, how to process image inputs with convolutional networks as well as how to process sequence inputs with recurrent and recursive networks. Moreover, we provide guidance for how to preprocess the data for various tasks and how to choose the values of the various settings that govern the behavior of these algorithms.

These chapters are the most important for a practitioner – someone who wants to begin implementing and using deep learning algorithms to solve real-world problems today.

# Chapter 6

# Feedforward Deep Networks

*Feedforward deep networks*, also known as *multilayer perceptrons (MLPs)*, are the quintessential deep networks. They are parametric functions defined by composing together many parametric functions. Each of these component functions has multiple inputs and multiple outputs. In neural network terminology, we refer to each sub-function as a *layer* of the network, and each scalar output of one of these functions as a *unit* or sometimes as a *feature*. Even though each unit implements a relatively simple mapping or transformation of its input, the function represented by the entire network can become arbitrarily complex.

Not every deep learning algorithm can be understood in terms of defining a single, deterministic function like feedforward deep networks, but all of them share the property of containing many layers of many units. We can think of the number of units in each layer as being the *width* of a machine learning model, and the number of layers as its *depth*. Feedforward deep networks provide a conceptually simple example of an algorithm that captures the many advantages that come from having significant width and depth. Feedforward deep networks are also the key technology underlying most of the contemporary commercial applications of deep learning to large datasets.

In Chapter 5, we encountered several different traditional machine learning algorithms, including linear regression, linear classifiers, logistic regression and kernel machines. All of these algorithms work by applying a linear transformation to a fixed set of features. These algorithms can learn non-linear functions, but the non-linear part is fixed. In other words, the functions are non-linear in the space of inputs $\boldsymbol{x}$, but they are linear in some other pre-defined space.

Neural networks allow us to learn new kinds of non-linearity. Another way to view this idea is that neural networks allow us to learn the features provided to a linear model. From this point of view, neural networks allow us to automate the design of features—a task that until recently was performed gradually and

collectively, by the combined efforts of an entire community of researchers.

## 6.1 MLPs from the 1980's

Feedforward supervised neural networks were among the first and most successful non-linear learning algorithms (Rumelhart *et al.*, 1986e,c). These networks learn at least one function defining the features, as well as a (typically linear) function mapping from features to output. The layers of the network that correspond to features rather than outputs are called *hidden layers*. This is because the correct values of the features are unknown. The features must be created by the training algorithm. The input and output of the network is by contrast *observed* or *visible* in the training data. Figure 6.1 shows a classical MLP architecture from the 1980's, with a single hidden layer. A deeper version is obtained by simply having more hidden layers. Modern versions include changes in the non-linearities used and training procedure.

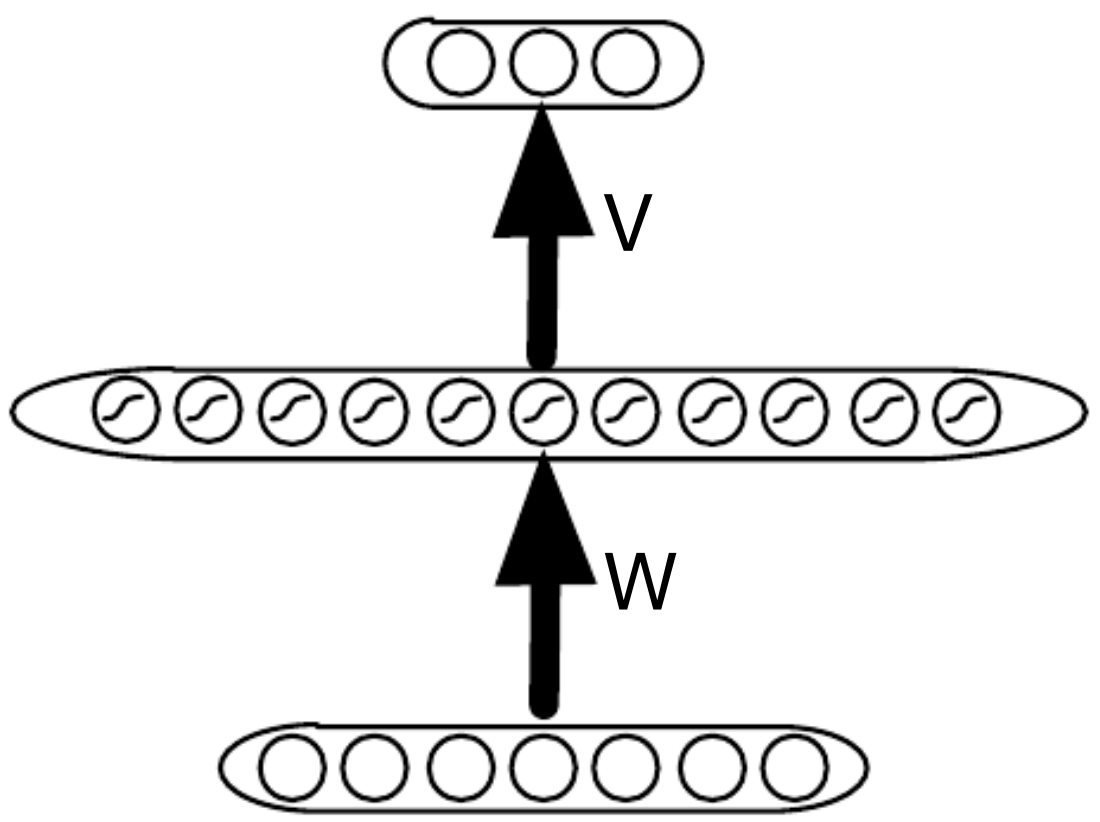


Figure 6.1: Shallow MLP, with one sigmoid hidden layer, computing vector-valued hidden unit vector $\boldsymbol{h} = \text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x})$ with weight matrix $\boldsymbol{W}$ and offset vector $\boldsymbol{c}$. The output vector is obtained via another learned affine transformation $\hat{\boldsymbol{y}} = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}$, with weight matrix $\boldsymbol{V}$ and output offset vector $\boldsymbol{b}$. The vector of hidden unit values $\boldsymbol{h}$ provides a new set of features, i.e., a new representation, derived from the raw input $\boldsymbol{x}$.

Example 6.1.1 introduces the equations for a shallow MLP for regression similar to those introduced in the 1980's, illustrated in Figure 6.1, which we will generalize below in the next few sections.

**Example 6.1.1. Shallow Multi-Layer Neural Network for Regression**

Based on the above definitions, we could pick the family of input-output functions to be

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{b} + \boldsymbol{V}\text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x}),$$

illustrated in Figure 6.1, where $\text{sigmoid}(a) = 1/(1 + e^{-a})$ is applied element-wise, the input is the vector $\boldsymbol{x} \in \mathbb{R}^{n_i}$, the hidden layer outputs are the elements of the vector $\boldsymbol{h} = \text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x})$ with $n_h$ entries, the parameters are $\boldsymbol{\theta} = (\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{V}, \boldsymbol{W})$ (with $\boldsymbol{\theta}$ also viewed as the flattened vectorized version of the tuple) with $\boldsymbol{b} \in \mathbb{R}^{n_o}$ a vector the same dimension as the output ($n_o$), $\boldsymbol{c} \in \mathbb{R}^{n_h}$ of the same dimension as $\boldsymbol{h}$ (number of hidden units), $\boldsymbol{V} \in \mathbb{R}^{n_o \times n_h}$ and $\boldsymbol{W} \in \mathbb{R}^{n_h \times n_i}$ being weight matrices.
The loss function for this classical example could be the squared error $L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$ (see Section 6.2 discussing how it makes $\hat{\boldsymbol{y}}$ an estimator of $\mathbb{E}[\boldsymbol{Y} \mid \boldsymbol{x}]$). The regularizer could be the ordinary $L^2$ weight decay $||\boldsymbol{\omega}||^2 = (\sum_{ij} W_{ij}^2 + \sum_{ki} V_{ki}^2)$, where we define the set of weights $\boldsymbol{\omega}$ as the concatenation of the elements of matrices $\boldsymbol{W}$ and $\boldsymbol{V}$. The $L^2$ weight decay thus penalizes the squared norm of the weights, with $\lambda$ a scalar that is larger to penalize stronger weights, thus yielding smaller weights. During training, we minimize a cost function obtained by adding together the squared loss and the regularization term:

$$J(\boldsymbol{\theta}) = \lambda||\boldsymbol{\omega}||^2 + \frac{1}{n}\sum_{t=1}^{n} ||\boldsymbol{y}^{(t)} - (\boldsymbol{b} + \boldsymbol{V}\text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x}^{(t)}))||^2.$$

where $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$ is the $t$-th training example, an (input,target) pair. Finally, the classical training procedure in this example is stochastic gradient descent, which iteratively updates $\boldsymbol{\theta}$ according to

$$\begin{aligned} \boldsymbol{\omega} &\leftarrow \boldsymbol{\omega} - \epsilon\Big(2\lambda\boldsymbol{\omega} + \nabla_{\boldsymbol{\omega}} L(f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}), \boldsymbol{y}^{(t)})\Big) \\ \boldsymbol{\beta} &\leftarrow \boldsymbol{\beta} - \epsilon\nabla_{\boldsymbol{\beta}} L(f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}), \boldsymbol{y}^{(t)}), \end{aligned}$$

where $\boldsymbol{\beta} = (\boldsymbol{b}, \boldsymbol{c})$ contains the offset[1] parameters, $\boldsymbol{\omega} = (\boldsymbol{W}, \boldsymbol{V})$ the weight matrices, $\epsilon$ is a learning rate and $t$ is incremented after each training example, modulo $n$. Section 6.4.3 shows how gradients can be computed efficiently thanks to backpropagation.

MLPs can learn powerful non-linear transformations: in fact, with enough hidden units they can represent arbitrarily complex but smooth functions, they can be universal approximators, as described below in Section 6.6. This is achieved by composing simple but non-linear learned transformations. By transforming the data non-linearly into a new space, a classification problem that was not linearly separable (not solvable by a linear classifier) can become separable, as illustrated in Figures 6.2 and 6.3.

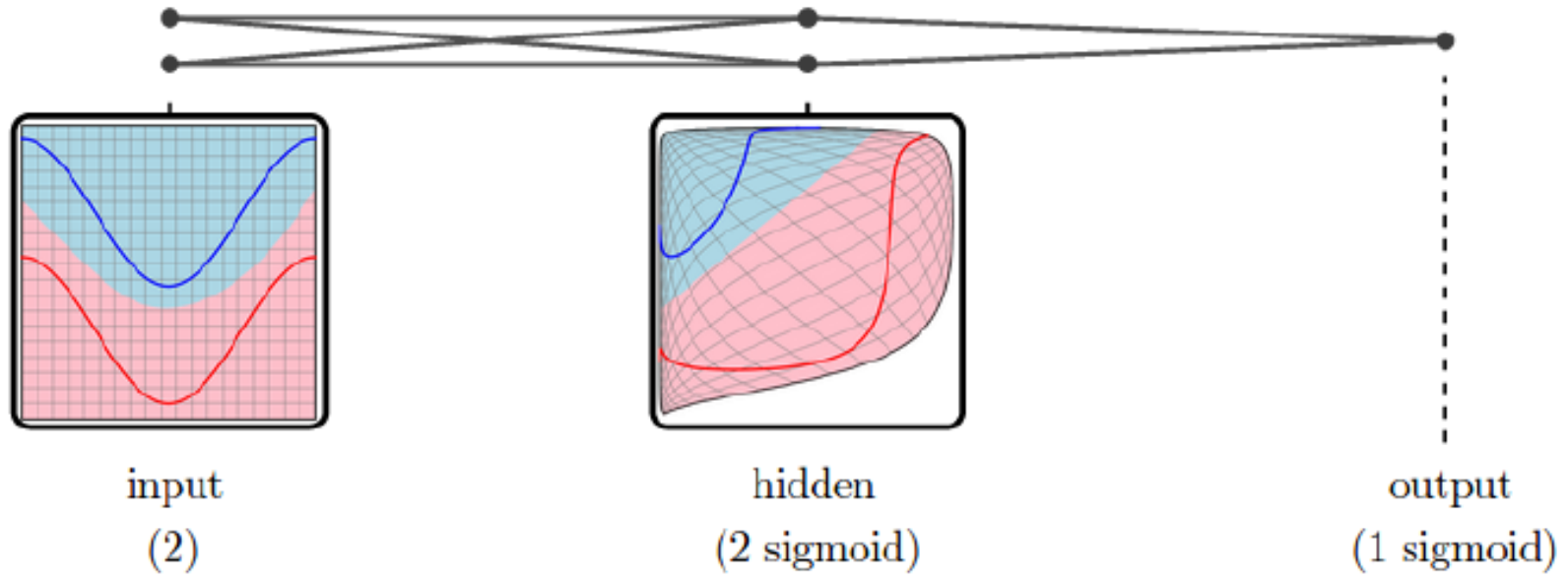


Figure 6.2: Each layer of a trained neural network non-linearly transforms its input, distorting the space so that the task becomes easier to perform, e.g., linear classification in the new feature space, in the above figures. The figure shows how a simple neural network with 2 inputs, 2 hidden units and one output can transform the 2-D input space so that the examples from the two classes become linearly separable. The red and blue solid curves are where the training examples come from (with color indicating class). The paler red (resp. blue) region indicates the region that should be labeled as red (resp. blue), with the blue-to-red interface corresponding to a good decision surface. On the left, we see in a black square the good decision surface in the original input space, and see that it is non-linear (i.e., a linear classifier could not do a good job if applied directly on the raw inputs). The raw input space is also mapped by a regular grid in the left square, and the grid gets transformed non-linearly, i.e., warped differently in different parts of the space (consider how the grid was transformed), to obtain the middle square, i.e., in the space of hidden units: every $(x\ ,x\ )$ point in the left block is mapped to a point $(h\ ,h\ )$ in the middle block using the parameters of the hidden units. We see that when the data are properly transformed non-linearly by the first hidden layer, the good decision surface becomes a linear one, which can be implemented by the output layer (which is a linear classifier when viewed as a function of the last hidden layer). Reproduced with permission by Chris Olah from , where many more insightful visualizations can be found.

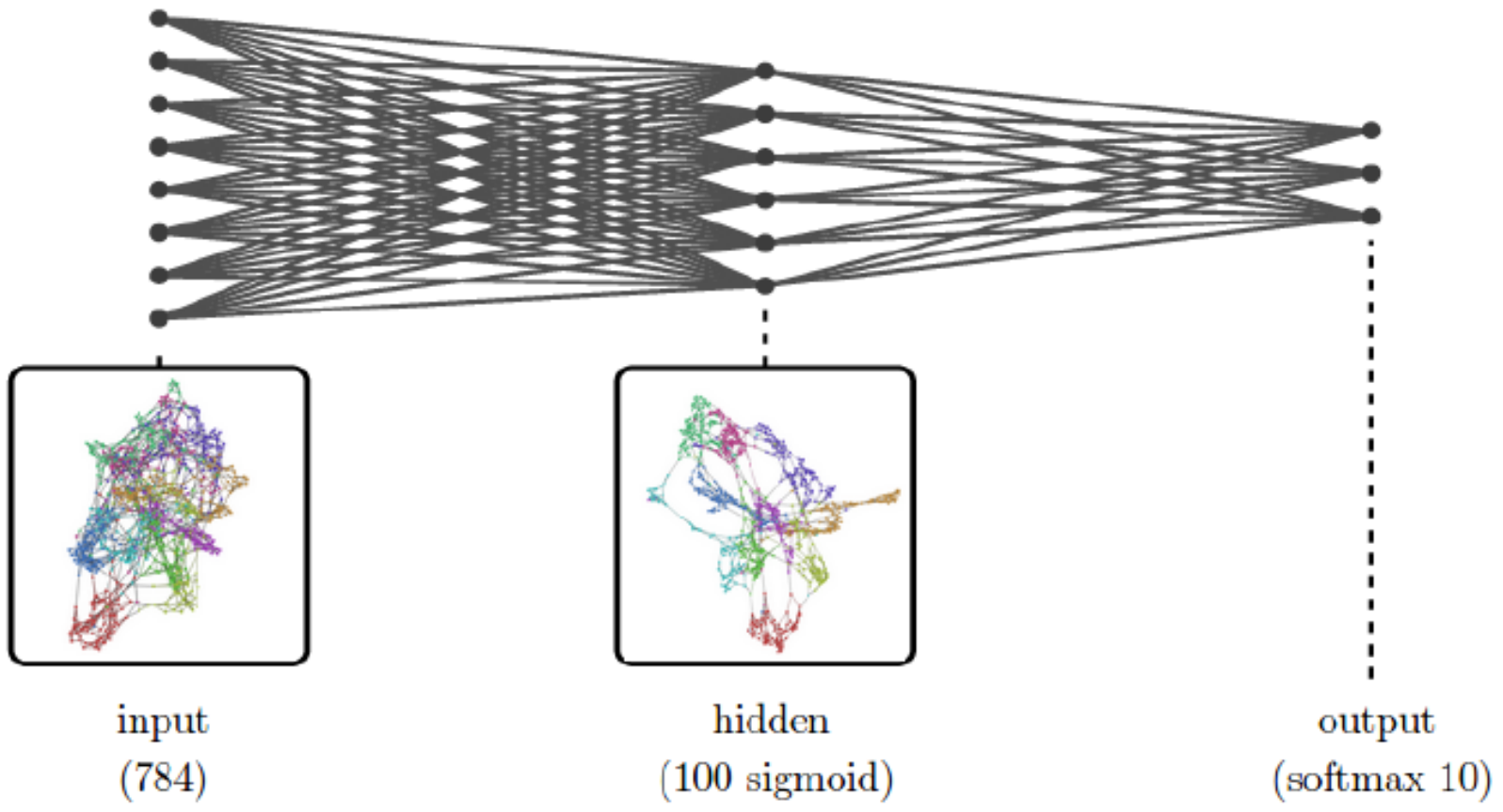


Figure 6.3: Like Figure 6.2, this figure shows how the hidden layer of an MLP can non-linearly warp the input space so as to make the classification task easier for a linear classifier (the output layer of a shallow MLP). The MLP now has 784 inputs (pixels of a $28 \times 28$ image of a digit), 100 hidden units and 10 outputs corresponding to the 10 digit classes. We cannot directly visualize with a 2-D grid the input and hidden layer spaces, but we can visualize a 2-D approximation obtained by dimensionality reduction. The squares below the input and hidden layer show where the training examples are in this reduced space, with one point per example, colored according to the digit class. Again, we see that the digits of different classes can be more easily separated in the feature space of the hidden layer than in the raw pixel space, with digits of the same class tending to form better separated clusters. Reproduced with permission by Chris Olah from , where more detail about this experiment can be found.

## 6.2 Estimating Conditional Statistics

To gently move from linear predictors to non-linear ones, let us consider the squared error loss function studied in the previous chapter, where the learning task is the estimation of the expected value of $\mathbf{y}$ given $\mathbf{x}$. In the context of linear regression, the conditional expectation of $\boldsymbol{y}$ is used as the mean of a Gaussian distribution that we fit with maximum likelihood. We can generalize linear regression to regression via any function $f$ by defining the mean squared error of $f$:

$$\mathbb{E}[||\boldsymbol{y} - f(\boldsymbol{x})||^2]$$

where the expectation is over the training set during training, and over the data generating distribution to obtain generalization error.

We can generalize its interpretation beyond the case where $f$ is linear or affine, uncovering an interesting property: minimizing it yields an estimator of the conditional expectation of the output variable $\mathbf{y}$ given the input variable $\mathbf{x}$, i.e.,

$$\arg\min_{f \in \mathbb{H}} \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}[||\boldsymbol{y} - f(\boldsymbol{x})||^2] = \mathbb{E}_{p(\ \ ,y)}[\mathbf{y}|\boldsymbol{x}]. \tag{6.1}$$

provided that our set of function $\mathbb{H}$ contains $\mathbb{E}_{p(\ ,\ )}[\mathbf{y} \mid \boldsymbol{x}]$. (If you would like to work out the proof yourself, it is easy to do using calculus of variations, which we describe in Chapter 19.6.2).

Similarly, we can generalize conditional maximum likelihood (introduced in Section 5.6.1) to other distributions than the Gaussian, as discussed below when defining the objective function for MLPs.

## 6.3 Parametrizing a Learned Predictor

There are many ways to define the family of input-output functions, cost function (including optional regularizers) and optimization procedure. The most common ones are described below, while more advanced ones are left to later chapters.

### 6.3.1 Family of Functions

A motivation for the family of functions defined by multi-layer neural networks is to *compose simple transformations in order to obtain highly non-linear ones*. In particular, MLPs compose affine transformations and element-wise non-linearities. As discussed in Section 6.6 below, with the appropriate choice of parameters, multi-layer neural networks can in principle approximate any smooth function, with more hidden units allowing one to achieve better approximations.

A multi-layer neural network with more than one hidden layer can be defined by generalizing the above structure, e.g., as follows, where we chose to use hyperbolic tangent[2] activation functions instead of sigmoid activation functions:

$$\boldsymbol{h}^k = \tanh(\boldsymbol{b}^k + \boldsymbol{W}^k \boldsymbol{h}^{k-1})$$

where $\boldsymbol{h}^0 = \boldsymbol{x}$ is the input of the neural net, $\boldsymbol{h}^k$ (for $k > 0$) is the output of the $k$-th hidden layer, which has weight matrix $\boldsymbol{W}^k$ and offset (or bias) vector $\boldsymbol{b}^k$. If we want the output $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ to lie in some desired range, then we typically define an *output non-linearity* (which we did not have in the above Example 6.1.1). The non-linearity for the output layer is generally different from the tanh, depending on the type of output to be predicted and the associated loss function (see below).

There are several other non-linearities besides the sigmoid and the hyperbolic tangent which have been successfully used with neural networks. In particular, we introduce some piece-wise linear units below such as the the rectified linear unit ($\max(0, b + \boldsymbol{w} \cdot \boldsymbol{x})$) and the maxout unit ($\max_i(b_i + \boldsymbol{W}_{:,i} \cdot \boldsymbol{x})$) which have been particularly successful in the case of deep feedforward or convolutional networks. A longer discussion of these can be found in Section 6.8.

These and other non-linear neural network activation functions commonly found in the literature are summarized below. Most of them are typically combined with an affine transformation $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$ and applied element-wise:

$$\boldsymbol{h} = \phi(\boldsymbol{a}) \Leftrightarrow h_i = \phi(a_i) = \phi(b_i + \boldsymbol{W}_{i,:}\boldsymbol{x}). \tag{6.2}$$

- **Rectifier** or **rectified linear unit (ReLU)** or **positive part**: transformation of the output of the previous layer: $\phi(a) = \max(0, a)$, also written $\phi(a) = (a)^+$. This is by far the most popular hidden unit in current feedforward networks. Some other effective variants are based on using a non-zero slope $\alpha_i$ when $a < 0$: $h_i = \phi(a, \alpha_i) = \max(0, a) + \alpha_i \min(0, a)$. where $\alpha_i$ can be a small fixed value like 0.01 A *leaky ReLU* (Maas *et al.*, 2013) fixes $\alpha_i$ to a small value like 0.01 while a *parametric ReLU* or *PReLU* treats $\alpha_i$ as a learnable parameter (He *et al.*, 2015).

- **Hyperbolic tangent**: $\phi(a) = \tanh(a)$.

- **Sigmoid**: $\phi(a) = 1/(1 + e^{-a})$.

- **Softmax**: This is a vector-to-vector transformation $\phi(\boldsymbol{a}) = \text{softmax}(\boldsymbol{a}) = e^{a_i} / \sum_j e^{a_j}$ such that $\sum_i \phi_i(\boldsymbol{a}) = 1$ and $\phi_i(\boldsymbol{a}) > 0$, i.e., the softmax output can be considered as a probability distribution over a finite set of outcomes.

which is linearly related to the sigmoid via tanh( ) = 2 sigmoid(2 ) 1 and typically yields easier optimization with stochastic gradient descent (Glorot and Bengio, 2010a).

Note that it is not applied element-wise but on a whole vector of "scores". It is mostly used as output non-linearity for predicting discrete probabilities over output categories. See definition and discussion below, around Eq. 6.4.

- **Radial basis function** or **RBF** unit: this one is not applied after a general affine transformation but acts on $\boldsymbol{x}$ using a different form that corresponds to a template matching, i.e., $h_i = \exp\left(-||\boldsymbol{w}_i - \boldsymbol{x}||^2/\sigma_i^2\right)$ (or typically with all the $\sigma_i$ set to the same value). This is heavily used in kernel SVMs (Boser *et al.*, 1992; Schölkopf *et al.*, 1999) and has the advantage that such units can be easily initialized (Powell, 1987; Niranjan and Fallside, 1990) as a random (or selected) subset of the input examples, i.e., $\boldsymbol{w}_i = \boldsymbol{x}^{(t)}$ for some assignment of examples $t$ to hidden unit templates $i$.

- **Softplus**: $\phi(a) = \zeta(a) = \log(1 + e^a)$. This is a smooth version of the rectifier, introduced in Dugas *et al.* (2001) for function approximation and in Nair and Hinton (2010a) in RBMs. Glorot *et al.* (2011a) compared the softplus and rectifier and found better results with the latter, in spite of the very similar shape and the differentiability and non-zero derivative of the softplus everywhere, contrary to the rectifier.

- **Hard tanh**: this is shaped similarly to the tanh and the rectifier but unlike the latter, it is bounded, $\phi(a) = \max(-1, \min(1, a))$. It was introduced by Collobert (2004).

- **Absolute value rectification**: $\phi(a) = |a|$ (may be applied on the affine dot product or on the output of a tanh unit). It is also a rectifier and has been used for object recognition from images (Jarrett *et al.*, 2009a), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

- **Maxout**: this is discussed in more detail in Section 6.8. It generalizes the rectifier but introduces multiple weight vectors $\boldsymbol{w}_i$ (called filters) for each hidden unit. $h_i = \max_i (b_i + \boldsymbol{w}_i \cdot \boldsymbol{x})$.

This is not an exhaustive list but covers most of the non-linearities and unit computations seen in the deep learning and neural nets literature. Many variants are possible.

As discussed in Section 6.4.3, the structure (also called *architecture*) of the family of input-output functions can be varied in many ways, which calls for a generic principle for efficiently computing gradients, described in that section. For example, a common variation is to connect layers that are not adjacent, with so-called skip connections, which are found in the visual cortex (where the word "layer" should be replaced by the word "area"). Other common variations depart

from a full connectivity between adjacent layers. For example, each unit at layer $k$ may be connected to only a subset of units at layer $k-1$. A particular case of such form of sparse connectivity is discussed in chapter 9 with *convolutional networks*. The set of connections between units of the whole network needs to form a directed acyclic graph in order to define a meaningful computation (see the flow graph formalism below, Section 6.4.3). *Recurrent networks*, treated in 10, are typically depicted using graphs containing cycles. Such graphs are using a different kind of graphical language. The cycles indicate that the value of a unit at time step $t+1$ is a function of the value of the unit at time step $t$. These cyclical graphs in the recurrent network language can be unrolled into directed acyclic graphs containing multiple time steps in order to obtain a traditional computational graph.

### 6.3.2 Loss Function and Conditional Log-Likelihood

In the 80's and 90's the most commonly used loss function was the *squared error* $L(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) = ||f_{\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{y}||^2$. As discussed in Section 6.2, if $f$ is unrestricted (non-parametric), minimizing the expected value of the loss function over some data-generating distribution $P(\mathbf{x}, \mathbf{y})$ yields $f(\boldsymbol{x}) = \mathbb{E}[\mathbf{y} \mid \mathbf{x} = \boldsymbol{x}]$, the true conditional expectation of $\mathbf{y}$ given $\mathbf{x}$. This tells us what the neural network is trying to learn. Replacing the squared error by an absolute value makes the neural network try to estimate not the conditional expectation but the conditional median[3].

However, when $y$ is a discrete label, i.e., for classification problems, other loss functions such as the Bernoulli negative log-likelihood[4] have been found to be more appropriate than the squared error. In the case where $y \in \{0, 1\}$ is binary this gives

$$L(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y) = -y \log f_{\boldsymbol{\theta}}(\boldsymbol{x}) - (1-y) \log(1 - f_{\boldsymbol{\theta}}(\boldsymbol{x})) \tag{6.3}$$

also known as *cross entropy objective function*. It can be shown that the optimal (non-parametric) $f$ minimizing this loss function is $f(\boldsymbol{x}) = P(\mathrm{y} = 1 \mid \boldsymbol{x})$. In other words, when maximizing the conditional log-likelihood objective function, we are training the neural net output to estimate conditional probabilities as well as possible in the sense of the KL divergence (see Section 3.9, Eq. 3.4). Note that in order for the above expression of the criterion to make sense, $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ must be strictly between 0 and 1 (an undefined or infinite value would otherwise arise). To achieve this, it is common to use the sigmoid as non-linearity for

Showing this is another interesting exercise.

Many authors use the term "cross entropy" to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood is a cross entropy between the empirical distribution defined by the training set and the model. For example, mean squared error is the cross entropy between the empirical distribution and a Gaussian model.

the output layer, which matches well with the binomial negative log-likelihood cost function[5]. As explained below (Softmax subsection), the log-likelihood of a Bernoulli variable whose mean is parameterized by a sigmoidal unit allows gradients to pass through the output non-linearity even when the neural network produces a confidently wrong answer, unlike the squared error criterion coupled with a sigmoid or softmax non-linearity.

### Learning a Conditional Probability Model

More generally, one can define a loss function as corresponding to a conditional log-likelihood, i.e., the negative log-likelihood (NLL) cost function

$$L_{\text{NLL}}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}\right) = -\log P(\mathbf{y} = \boldsymbol{y} \mid \mathbf{x} = \boldsymbol{x}; \boldsymbol{\theta}).$$

See Section 5.6.1 (and the one before) which shows that this criterion corresponds to minimizing the KL divergence between the model $P$ of the conditional probability of $\mathbf{y}$ given $\mathbf{x}$ and the data generating distribution $Q$, approximated here by the finite training set, i.e., the empirical distribution of pairs $(\boldsymbol{x}, \boldsymbol{y})$. Hence, minimizing this objective, as the amount of data increases, yields an estimator of the true conditional probability of $\mathbf{y}$ given $\mathbf{x}$.

For example, if $\mathbf{y}$ is a continuous random variable and we assume that, given $\mathbf{x}$, it has a Gaussian distribution with mean $f_{\boldsymbol{\theta}}(\mathbf{x})$ and variance $\sigma^2$, then

$$-\log P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}) = \frac{1}{2}(f_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbf{y})^2/\sigma^2 + \log(2\pi\sigma^2).$$

Up to an additive and multiplicative constant (which would give the same choice of $\boldsymbol{\theta}$), minimizing this negative log-likelihood is therefore equivalent to minimizing the squared error loss. Once we understand this principle, we can readily generalize it to other distributions, as appropriate. For example, it is straightforward to generalize the univariate Gaussian to the multivariate case, and under appropriate parametrization consider the variance to be a parameter or even a parametrized function of $\boldsymbol{x}$ (for example with output units that are guaranteed to be positive, or forming a positive definite matrix, as outlined below, Section 6.3.2).

Similarly, for discrete variables, the binomial negative log-likelihood cost function corresponds to the conditional log-likelihood associated with the Bernoulli distribution (also known as cross entropy) with probability $p = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ of generating $\mathrm{y} = 1$ given $\mathbf{x} = \boldsymbol{x}$ (and probability $1 - p$ of generating $\mathrm{y} = 0$):

$$\begin{aligned} L_{\text{NLL}} &= -\log P(\mathrm{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) = -\mathbf{1}_{\mathrm{y}=1} \log p - \mathbf{1}_{\mathrm{y}=0} \log(1-p) \\ &= -\mathrm{y} \log f_{\boldsymbol{\theta}}(\boldsymbol{x}) - (1-\mathrm{y}) \log(1 - f_{\boldsymbol{\theta}}(\boldsymbol{x})). \end{aligned}$$

---

In reference to statistical models, this "match" between the loss function and the output non-linearity is similar to the choice of a in generalized linear models (McCullagh and Nelder, 1989).

where $\mathbf{1}_{\mathrm{y}=1}$ is the usual binary indicator.

### Softmax

When y is discrete and has a finite domain (say $\{1, \ldots, n\}$) but is not binary, the Bernoulli distribution is extended to the multinoulli distribution (defined in Section 3.10.2). This distribution is specified by a vector of $n-1$ probabilities whose sum is less than or equal to 1, each element of which provides the probability $p_i = P(\mathrm{y} = i \mid \boldsymbol{x})$. We can then recover $P(\mathrm{y} = N \mid \boldsymbol{x})$ as $1 - \sum_{i=1}^{n-1} P(\mathrm{y} = i \mid \boldsymbol{x})$. Alternatively, one can specify a vector of $n$ probabilities whose sum is exactly 1. The two options have the same representational power but different learning dynamics.

The *softmax* non-linearity (Bridle, 1990): was designed for the purpose of specifying multinoulli distributions:

$$\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{a}) \iff p_i = \frac{e^{a_i}}{\sum_j e^{a_j}}. \tag{6.4}$$

where typically $\boldsymbol{a}$ is a set of activations coming from the lower layers of the network. We often use the overparameterized $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}$, but we may hardcode $a_n$ to 0 in order to specify only $n-1$ of the output probabilities. We can think of $a$ as a vector of scores whose elements $a_i$ are associated with each category $i$, with larger relative scores yielding exponentially larger probabilities. The corresponding loss function is therefore $L_{\mathrm{NLL}}(\boldsymbol{p}, y) = -\log p_y$. Note how minimizing this loss will push $a_y$ up (increase the score $a_y$ associated with the correct label $y$) while pushing down $a_i$ for $i \neq y$ (decreasing the score of the other labels, in the context $\boldsymbol{x}$). The first effect comes from the numerator of the softmax while the second effect comes from the normalizing denominator. These forces cancel on a specific example only if $p_y = 1$ and they cancel in average over examples (say sharing the same $x$) if $p_i$ equals the fraction of times that $\mathrm{y} = i$ for this value $\boldsymbol{x}$. To see this, consider the gradient with respect to the scores $\boldsymbol{a}$:

$$\begin{aligned}
\frac{\partial}{\partial a_k} L_{\mathrm{NLL}}(\boldsymbol{p}, y) &= \frac{\partial}{\partial a_k}(-\log p_y) = \frac{\partial}{\partial a_k}(-a_y + \log \sum_j e^{a_j}) \\
&= -\mathbf{1}_{y=k} + \frac{e^{a_k}}{\sum_j e^{a_j}} \\
&= p_k - 1_{y=k} \quad \text{or} \\
\frac{\partial}{\partial \boldsymbol{a}} L_{\mathrm{NLL}}(\boldsymbol{p}, y) &= (\boldsymbol{p} - \boldsymbol{e}_y)
\end{aligned} \tag{6.5}$$

where $\boldsymbol{e}_y = [0, \ldots, 0, 1, 0, \ldots, 0]$ is the one-hot vector with a 1 at position $y$. Examples that share the same $\boldsymbol{x}$ share the same $\boldsymbol{a}$, so the average gradient on $\boldsymbol{a}$

over these examples is 0 when the average of the above expression cancels out, i.e., $\boldsymbol{p} = \mathbb{E}_y[\boldsymbol{e}_y \mid \boldsymbol{x}]$ where the expectation is over these examples. Thus the optimal $p_i$ for these examples is the average number of times that $y = i$ among those examples. Over an infinite number of examples, we would find that the gradient is 0 when $p_i$ perfectly estimates the true $P(y = i \mid \boldsymbol{x})$. What the above gradient decomposition teaches us as well is the division of the total gradient into (1) a term due to the numerator (the $\boldsymbol{e}_y$) and dependent on the actually observed target $y$ and (2) a term independent of $y$ but which corresponds to the gradient of the softmax denominator. The same principles and the role of the normalization constant (or "partition function") can be seen at play in the training of Markov Random Fields, Boltzmann machines and RBMs, in Chapter 13.

The softmax has other interesting properties. First of all, the gradient of $logp(y = i \mid \boldsymbol{x})$ with respect to $\boldsymbol{a}$ only saturates in the case when $p(y = i \mid \boldsymbol{x})$ is already nearly maximal, i.e., approaching 1. Specifically, let us consider the case where the correct label is $i$, i.e. y $= i$. The element of the gradient associated with an erroneous label, say $j \neq i$, is

$$\frac{\partial}{\partial a_j} L_{\text{NLL}}(\boldsymbol{p}, y) = p_j \,. \tag{6.6}$$

So if the model correctly predicts a low probability that the y $= j$, i.e. that $p_j \approx 0$, then the gradient is also close to zero. But if the model incorrectly and confidently predicts that $j$ is the correct class, i.e., $p_j \approx 1$, there will be a strong push to reduce $a_j$. Conversely, if the model incorrectly and confidently predicts that the correct class $y$ should have a low probability, i.e., $p_y \approx 0$, there will be a strong push (a gradient of about -1) to push $a_y$ up. One way to see these is to imagine doing gradient descent on the $a_j$'s themselves (that is what backprop is really based on): the update on $a_j$ would be proportional to minus one times the gradient on $a_j$, so a positive gradient on $a_j$ (e.g., incorrectly confident that $p_j \approx 1$) pushes $a_j$ down, while a negative gradient on $a_j$ (e.g., incorrectly confident that $p_y \approx 0$) pushes $a_y$ up. In fact note how $a_y$ is *always pushed up* because $p_y - 1_{y=y} = p_y - 1 < 0$, and the other scores $a_j$ (for $j \neq y$) are *always pushed down, because their gradient is* $p_j > 0$.

There are other loss functions such as the squared error applied to softmax (or sigmoid) outputs (which was popular in the 80's and 90's) which have vanishing gradient when an output unit saturates (when the derivative of the non-linearity is near 0), *even if the output is completely wrong* (Solla *et al.*, 1988). This may be a problem because it means that the parameters will basically not change, even though the output is wrong.

To see how the squared error interacts with the softmax output, we need to introduce a one-hot encoding of the label, $\boldsymbol{y} = \boldsymbol{e}_i = [0, \ldots, 0, 1, 0, \ldots, 0]$, i.e for the label y $= i$, we have $\boldsymbol{y}_i = 1$ and $\boldsymbol{y}_j = 0, \forall j \neq i$. We will again consider that

we have the output of the network to be $\boldsymbol{p} = \text{softmax}(\boldsymbol{a})$, where, as before, $\boldsymbol{a}$ is the input to the softmax function ( e.g. $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}$ with $\boldsymbol{h}$ the output of the last hidden layer).

For the squared error loss $L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) = ||\boldsymbol{p}(\boldsymbol{a}) - \boldsymbol{y}||^2$, the gradient of the loss with respect to the input vector to the softmax, $\boldsymbol{a}$, is given by:

$$\begin{aligned} \frac{\partial}{\partial a_i} L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) &= \frac{\partial L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y})}{\partial \boldsymbol{p}(\boldsymbol{a})} \frac{\partial \boldsymbol{p}(\boldsymbol{a})}{\partial a_i} \\ &= \sum_j 2(p_j(\boldsymbol{a}) - \boldsymbol{y}_j) p_j (1_{i=j} - p_i). \end{aligned} \tag{6.7}$$

So if the model incorrectly predicts a low probability for the correct class y $= i$, i.e., if $p_y = p_i \approx 0$, then the score for the correct class, $a_y$, does not get pushed up in spite of a large error, i.e., $\frac{\partial}{\partial a_y} L_2(\boldsymbol{p}(\boldsymbol{a}), \boldsymbol{y}) \approx 0$. For this reason, practitioners prefer to use the negative log-likelihood (cross entropy) cost function, with the softmax non-linearity (as well as with the sigmoid non-linearity), rather than applying the squared error criterion to these probabilities.

Another uesful property of the softmax is that its output is invariant to adding a scalar to all of its inputs:

$$\text{softmax}(\boldsymbol{a}) = \text{softmax}(\boldsymbol{a} + b).$$

This property is used to implement the numerically stable variant of the softmax, which exploits the fact that $\text{softmax}(\boldsymbol{a}) = \text{softmax}(\boldsymbol{a} - \max_i a_i)$. This allows us to evaluate softmax with only small numerical errors even when $a$ contains extremely large or extremely negative numbers.

Finally, it is interesting to think of the softmax as a way to create a form of *competition* between the units (typically output units, but not necessarily) that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the *lateral inhibition* that is believed to exist between nearby neurons in cortex. At the extreme (when the difference between the maximal $a_i$ and the others is large in magnitude) it becomes a form of *winner-take-all* (one of the outputs is nearly 1 and the others are nearly 0). A more computationally expensive form of competition is found with *sparse coding*, described in Section 19.3.

### Neural Net Outputs as Parameters of a Conditional Distribution

In general, for any parametric probability distribution $p(\mathbf{y} \mid \boldsymbol{\omega})$ with parameters $\boldsymbol{\omega}$, we can construct a *conditional distribution* $p(\mathbf{y} \mid \mathbf{x})$ by making $\boldsymbol{\omega}$ a parametrized function of $\mathbf{x}$ and learning that function:

$$p(\mathbf{y} \mid \boldsymbol{\omega} = f_{\boldsymbol{\theta}}(\mathbf{x}))$$

where $f_{\boldsymbol{\theta}}(\mathbf{x})$ is the *output* of a predictor, $\mathbf{x}$ is its input, and $\mathbf{y}$ can be thought of as a *"target"*. The use of the word "target" comes from the common cases of classification and regression, where $f_{\boldsymbol{\theta}}(\mathbf{x})$ is really a prediction associated with random variable $\mathbf{y}$, or with its expected value. However, in general $\omega = f_{\boldsymbol{\theta}}(\mathbf{x})$ may contain parameters of the distribution of $\mathbf{y}$ other than its expected value. For example, it could contain its variance or covariance, in the case where $\mathbf{y}$ is conditionally Gaussian. In the above examples, with the squared error loss, $\omega$ is the mean of the Gaussian which captures the conditional distribution of $\mathbf{y}$ (which means that the variance is considered fixed, not a function of $\mathbf{x}$). In the common classification case, $\omega$ contains the probabilities associated with the various events of interest.

Once we view things in this way, if we apply the principle of maximum likelihood in the conditional case (Section 5.6.1), we automatically get as the natural cost function the negative log-likelihood $L(\mathbf{x}, \mathbf{y}) = -\log p(\mathbf{y} \mid \omega = f_{\boldsymbol{\theta}}(\mathbf{x}))$. Besides the expected value of $\mathbf{y}$, there could be other parameters of the conditional distribution of $\mathbf{y}$ that control the distribution of $\mathbf{y}$, given $\mathbf{x}$. For example, we may wish to learn the variance of a conditional Gaussian for $\mathbf{y}$, given $\mathbf{x}$, and that variance could be a function that varies with $\mathbf{x}$ or that is a constant with respect to $\mathbf{x}$. If the variance $\sigma^2$ of $\mathbf{y}$ given $\mathbf{x}$ is not a function of $\mathbf{x}$, its maximum likelihood value can be computed analytically because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations $\mathbf{y}$ and their expected value (here estimated by $f_{\boldsymbol{\theta}}(\mathbf{x})$). In the scalar case, we could estimate $\sigma$ as follows:

$$\sigma^2 \leftarrow \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{y}^{(t)} - f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}))^2 \tag{6.8}$$

where $^{(t)}$ indicates the $t$-th training example $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$. In other words, the conditional variance can simply be estimated from the mean squared error. If $\mathbf{y}$ is a $d$-vector and the conditional covariance is $\sigma^2$ times the identity, then the above formula should be modified as follows, again by setting the gradient of the log-likelihood with respect to $\sigma$ to zero:

$$\sigma^2 \leftarrow \frac{1}{nd} \sum_{i=1}^{n} ||\boldsymbol{y}^{(t)} - f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)})||^2 . \tag{6.9}$$

In the multivariate case with a diagonal covariance matrix with entries $\sigma_i^2$, we obtain

$$\sigma_i^2 \leftarrow \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{y}_i^{(t)} - f_{\boldsymbol{\theta},i}(\boldsymbol{x}^{(t)}))^2. \tag{6.10}$$

In the multivariate case with a full covariancae matrix, we have

$$\boldsymbol{\Sigma} \leftarrow \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}_i^{(t)} - f_{\boldsymbol{\theta},i}(\boldsymbol{x}^{(t)}))(\boldsymbol{y}_i^{(t)} - f_{\boldsymbol{\theta},i}(\boldsymbol{x}^{(t)}))^\top \tag{6.11}$$

If the variance $\boldsymbol{\Sigma}(\boldsymbol{x})$ is a function of $\boldsymbol{x}$, there is no known method for maximizing the likelihood in closed form, but we can compute the gradient necessary for use with an iterative optimization procedure. If $\boldsymbol{\Sigma}(\boldsymbol{x})$ is diagonal or scalar, only positivity must be enforced, e.g., using the *softplus* non-linearity:

$$\sigma_i(\boldsymbol{x}) = \text{softplus}(g_{\boldsymbol{\theta}}(\boldsymbol{x})).$$

where $g_{\boldsymbol{\theta}}(\boldsymbol{x})$ may be a neural network that takes $\boldsymbol{x}$ as input. A positive non-linearity may also be useful in the case where $\sigma$ is a function of $\boldsymbol{x}$, if we do not seek the maximum likelihood solution (for example we do not have immediate observed targets associated with that Gaussian distribution, because the samples from the Gaussian are used as input for further computation). Then we can make the free parameter $\boldsymbol{\omega}$ defining the variance the argument of the positive non-linearity, e.g., $\sigma_i(x) = \text{softplus}(\omega_i)$. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive-definiteness of the predicted covariance matrix. This can be achieved by writing $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^\top(\boldsymbol{x})$, where $\boldsymbol{B}$ is an unconstrained square matrix. One practical issue if the the matrix is full is that computing the likelihood is expensive, requiring $O(d^3)$ computation for the determinant and inverse of $\boldsymbol{\Sigma}(\boldsymbol{x})$ (or equivalently, and more commonly done, its eigendecomposition or that of $\boldsymbol{B}(\boldsymbol{x})$).

Besides the Gaussian, a simple and common example is the case where y is binary (i.e. Bernoulli distributed), where it is enough to specify $\omega = p(\mathrm{y} = 1 \mid \mathbf{x})$. In the multinoulli case (multiple discrete values), $\omega$ is generally specified by a vector of probabilities (one per possible discrete value) summing to 1, e.g., via the softmax non-linearity discussed above.

Another interesting and powerful example of output distribution for neural networks is the *mixture model*, and in particular the *Gaussian mixture model*, introduced in Section 3.10.6. Neural networks that compute the parameters of a mixture model were introduced in Jacobs *et al.* (1991); Bishop (1994). In the case of the Gaussian mixture model with $n$ components,

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x})\mathcal{N}(\boldsymbol{y} \mid \mu_i(\boldsymbol{x}), \boldsymbol{\Sigma}_i(\boldsymbol{x})).$$

The neural network must have three outputs: $p(\mathrm{c} = i \mid \boldsymbol{x})$, $\mu_i(\boldsymbol{x})$ and $\boldsymbol{\Sigma}_i(\boldsymbol{x})$. These outputs must satisfy different constraints:

1. Mixture components $p(\mathrm{c} = i \mid \boldsymbol{x})$: these form a multinoulli distribution over the $n$ different components associated with latent[6] variable c, and can typically be obtained by a softmax over an $n$-dimensional vector, to guarantee that these outputs are positive and sum to 1.

2. Means $\mu_i(\boldsymbol{x})$: these indicate the center or mean associated with the $i$-th Gaussian component, and are unconstrained (typically with no non-linearity at all for these output units). If $\mathbf{y}$ is a $d$-vector, then the network must output an $n \times d$ matrix containing all $n$ of these $d$-dimensional vectors.

3. Covariances $\boldsymbol{\Sigma}_i(\boldsymbol{x})$: these specify the covariance matrix for each component $i$. For the general case of an unconditional (does not depend on $\boldsymbol{x}$) but full covariance matrix, see Eq. 6.11 to set it by maximum likelihood. In many models the variance is both unconditional and diagonal (like assumed with Eq. 6.10) or even scalar (like assumed with Eq. 6.8 or 6.9).

It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be finicky, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to *clip gradients* (see Section 10.7.7 and Mikolov (2012); Pascanu and Bengio (2012); Graves (2013); Pascanu *et al.* (2013a)), while another is to scale the gradients heuristically (Murray and Larochelle, 2014).

### Multiple Output Variables

When $\mathbf{y}$ is actually a tuple formed by multiple random variables $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k)$, then one has to choose an appropriate form for their joint distribution, conditional on $\mathbf{x} = \boldsymbol{x}$. The simplest and most common choice is to assume that the $\mathbf{y}_i$ are conditionally independent, i.e.,

$$p(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k \mid \boldsymbol{x}) = \prod_{i=1}^{k} p(\mathbf{y}_i \mid \boldsymbol{x}).$$

This brings us back to the single variable case, especially since the log-likelihood now decomposes into a sum of terms $\log p(\mathbf{y}_i \mid \boldsymbol{x})$. If each $p(\mathbf{y}_i \mid \boldsymbol{x})$ is separately parametrized (e.g. a different neural network), then we can train these neural networks independently. However, a more common and powerful choice assumes that the different variables $\mathbf{y}_i$ share some common factors, given $\boldsymbol{x}$, that can be

c is called latent because we do not observe it in the data: given input and target , it is not 100% clear which Gaussian component was responsible for , but we can imagine that was generated by picking one of them, and make that unobserved choice a random variable.

represented in some hidden layer of the network (such as the top hidden layer). See Sections 6.7 and 7.12 for a deeper treatment of the notion of underlying factors of variation and multi-task training: each $(\mathbf{x}, \mathbf{y}_i)$ pair of random variables can be associated with a different learning task, but it might be possible to exploit what these tasks have in common. See also Figure 7.6 illustrating these concepts.

If the conditional independence assumption is considered too strong, what can we do? At this point it is useful to step back and consider everything we know about learning a joint probability distribution. Since any probability distribution $p(\mathbf{y}; \boldsymbol{\omega})$ parametrized by parameters $\boldsymbol{\omega}$ can be turned into a conditional distribution $p(\mathbf{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ (by making $\boldsymbol{\omega}$ a function $\boldsymbol{\omega} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ parametrized by $\boldsymbol{\theta}$), we can go beyond the simple parametric distributions we have seen above (Gaussian, Bernoulli, multinoulli), and use more complex joint distributions. If the set of values that $\mathbf{y}$ can take is small enough (e.g., we have 8 binary variables $\mathbf{y}_i$, i.e., a joint distribution involving $2^8 = 256$ possible values), then we can simply model all these joint occurences as separate values, e.g., with a softmax and multinoulli over all these configurations. However, when the set of values that $\mathbf{y}_i$ can take cannot be easily enumerated and the joint distribution is not unimodal or factorized, we need other tools. The third part of this book is about the frontier of research in deep learning, and much of it is devoted to modeling such complex joint distributions, also called *graphical models*: see Chapters 13, 18, 19, 20. In particular, Section 12.5 discusses how sophisticated joint probability models with parameters $\boldsymbol{\omega}$ can be coupled with neural networks that compute $\boldsymbol{\omega}$ as a function of inputs $\boldsymbol{x}$, yielding *structured output* models conditioned with deep learning.

### 6.3.3 Cost Functions for Neural Networks

Typically, the training criteria for neural networks are primarily based on maximum likelihood. In the case of supervised learning, this will be the conditional version of maximum likelihood when we perform supervised learning.

In addition to the negative log-likelihood cost, we also often add some sort of a regularization term. Many of the regularization terms that apply to linear models also apply to neural networks. For example, weight decay applies to neural networks as well as to linear models.

These ideas have all been described for machine learning models in general in Chapter 5. When designing cost functions for neural networks specifically, we can often specialize the regularization terms for neural networks in various ways, such as controlling the properties of the individual hidden units. These strategies are covered in Chapter 7.

In practice, a good choice for the criterion is maximum likelihood regularized with dropout, possibly also with weight decay.

### 6.3.4 Optimization Procedure

Previously, we have seen simple machine learning models which could sometimes be fit in closed form. Neural networks must essentially always be optimized with iterative procedures. Optimization of neural networks is so difficult that the choice of optimization procedure is often tightly intertwined with the choice of model. In other words, we often design the model to make optimization easier. Chapter 8 is devoted to the iterative optimization procedures used to train neural networks and other deep models, including optimization strategies that involve designing the model to be easier to optimize.

In practice, a good choice for the optimization algorithm for a feedforward network is usually stochastic gradient descent with momentum. Typically, to make the model easier to optimize, it is best to use piecewise linear hidden units.

## 6.4 Flow Graphs and Back-Propagation

The term *back-propagation* is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually it just means the method for computing gradients in such networks. Furthermore, it is generally understood as something very specific to multi-layer neural networks, but once its derivation is understood, it can easily be generalized to arbitrary families of parametrized functions (for which computing a gradient is meaningful) and their corresponding computational graph. We describe this generalization here, focusing on the case of interest in machine learning where the output of the function to differentiate (e.g., the training criterion $J$) is a scalar and we are interested in its derivative with respect to a set of parameters (considered to be the elements of a vector $\boldsymbol{\theta}$), or equivalently, a set of inputs[7]. The partial derivative of $J$ with respect to $\boldsymbol{\theta}$ (called the gradient) tells us whether $\boldsymbol{\theta}$ should be increased or decreased in order to decrease $J$, and is a crucial tool in optimizing the training objective. It can be readily proven that the back-propagation algorithm for computing gradients has optimal computational complexity in the sense that there is no algorithm that can compute the gradient faster (in the $O(\cdot)$ sense, i.e., up to an additive and multiplicative constant).

The basic idea of the back-propagation algorithm is that the partial derivative of the cost $J$ with respect to parameters $\boldsymbol{\theta}$ can be *decomposed recursively* by taking into consideration the composition of functions that relate $\boldsymbol{\theta}$ to $J$, via intermediate quantities that mediate that influence, e.g., the activations of hidden units in a deep neural network.

---

It is useful to know which inputs contributed most to the output or error made, and the sign of the derivative is also interesting in that context.

### 6.4.1 Chain Rule

The basic mathematical tool for considering derivatives through compositions of functions is the **chain rule**, illustrated in Figure 6.4. The partial derivative $\frac{\partial y}{\partial x}$ measures the locally linear influence of a variable $x$ on another one $y$, while we denote $\nabla_{\boldsymbol{\theta}} J$ for the gradient vector of a scalar $J$ with respect to some vector of variables $\boldsymbol{\theta}$. If $x$ influences $y$ which influences $z$, we are interested in how a tiny change in $x$ propagates into a tiny change in $z$ via a tiny change in $y$. In our case of interest, the "output" is the cost, or objective function $z = J(g(\boldsymbol{\theta}))$, we want the gradient with respect to some parameters $x = \boldsymbol{\theta}$, and there are intermediate quantities $y = g(\boldsymbol{\theta})$ such as neural net activations. The gradient of interest can then be decomposed, according to the chain rule, into

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta})) \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{6.12}$$

which works also when $J$, $g$ or $\boldsymbol{\theta}$ are vectors rather than scalars (in which case the corresponding partial derivatives are understood as Jacobian matrices of the appropriate dimensions). In the purely scalar case we can understand the chain rule as follows: a small change in $\boldsymbol{\theta}$ will propagate into a small change in $g(\boldsymbol{\theta})$ by getting multiplied by $\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$. Similarly, a small change in $g(\boldsymbol{\theta})$ will propagate into a small change in $J(g(\boldsymbol{\theta}))$ by getting multiplied by $\nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta}))$. Hence a small change in $\boldsymbol{\theta}$ first gets multiplied by $\frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ to obtain the change in $g(\boldsymbol{\theta})$ and this then gets multiplied by $\nabla_{g(\boldsymbol{\theta})} J(g(\boldsymbol{\theta}))$ to obtain the change in $J(g(\boldsymbol{\theta}))$. Hence the ratio of the change in $J(g(\boldsymbol{\theta}))$ to the change in $\boldsymbol{\theta}$ is the product of these partial derivatives.

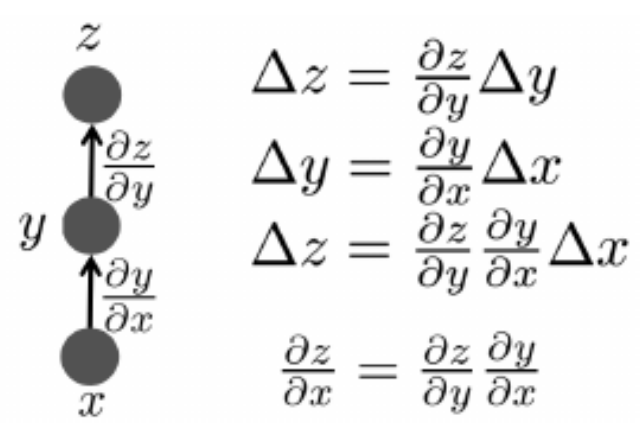


Figure 6.4: The chain rule, illustrated in the simplest possible case, with $z$ a scalar function of $y$, which is itself a scalar function of $x$. A small change $\Delta x$ in $x$ gets turned into a small change $\Delta y$ in $y$ through the partial derivative —, from the first-order Taylor approximation of $y(x)$, and similarly for $z(y)$. Plugging the equation for $\Delta y$ into the equation for $\Delta z$ yields the chain rule.

Now, if $g$ is a vector, we can rewrite the above as follows:

$$\nabla_{\boldsymbol{\theta}} J(g(\boldsymbol{\theta})) = \sum_i \frac{\partial J(g(\boldsymbol{\theta}))}{\partial g_i(\boldsymbol{\theta})} \frac{\partial g_i(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

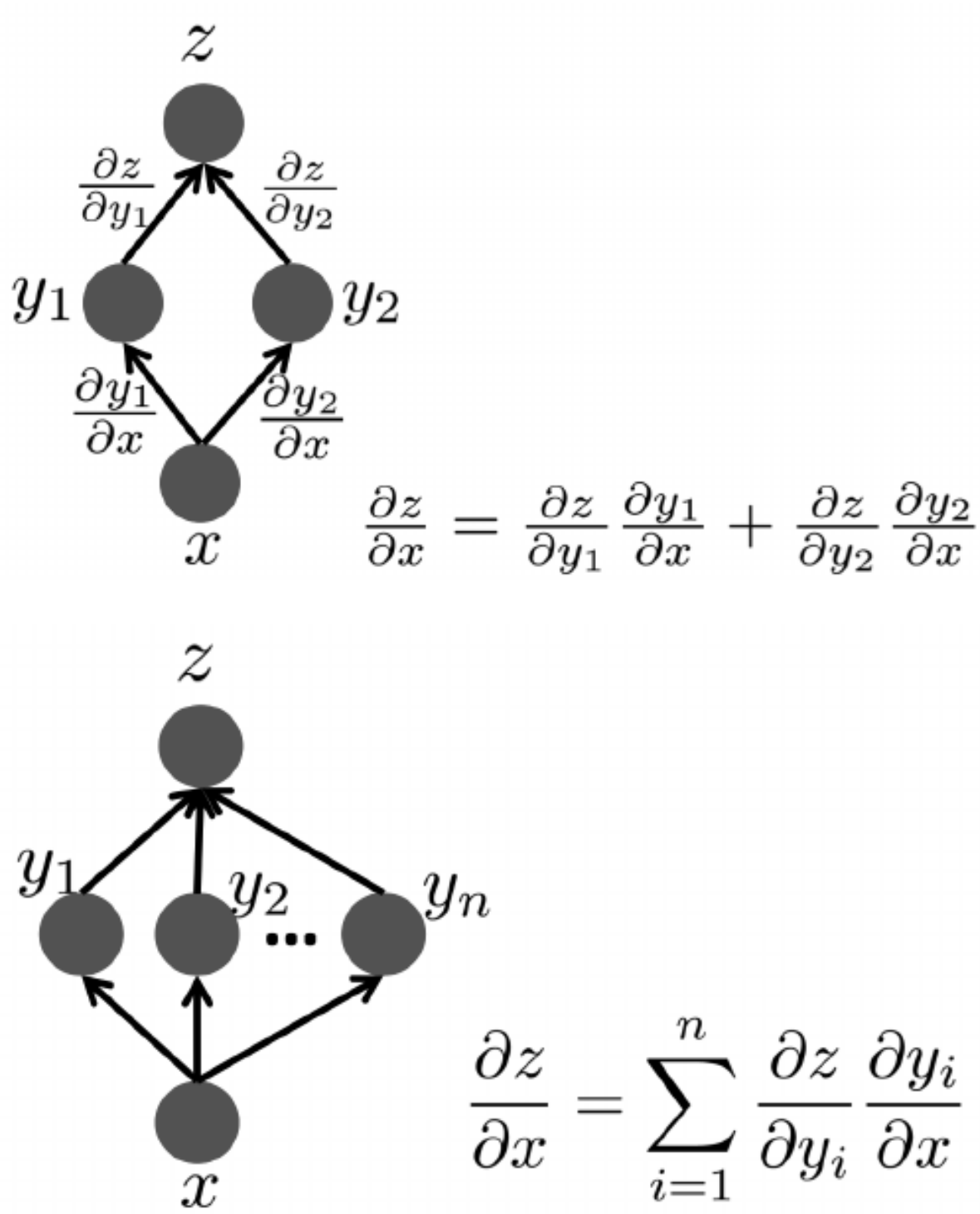


Figure 6.5: Top: The chain rule, when there are two intermediate variables $y$ and $y$ between $x$ and $z$, creating two paths for changes in $x$ to propagate and yield changes in $z$. Bottom: more general case, with $n$ intermediate variables $y$ to $y$ .

which sums over the influences of $\boldsymbol{\theta}$ on $J(g(\boldsymbol{\theta}))$ through all the intermediate variables $g_i(\boldsymbol{\theta})$. This is illustrated in Figure 6.5 with $x = \boldsymbol{\theta}$, $y_i = g_i(\boldsymbol{\theta})$, and $z = J(g(\boldsymbol{\theta}))$.

### 6.4.2 Back-Propagation in an MLP

Example 6.1.1 illustrated the case of an MLP with a single hidden layer. In this section we extend back-propagation to a deep MLP. This MLP is the same as before, but with multiple hidden layers rather than a single hidden layer. For this purpose, we will recursively apply the chain rule illustrated in Figure 6.5. The algorithm proceeds by first computing the gradient of the cost $J$ with respect to output units, and these are used to compute the gradient of $J$ with respect to the top hidden layer activations, which directly influence the outputs. We can then

**Algorithm 6.1** *Forward* computation associated with input $\boldsymbol{x}$ for a deep neural network with ordinary affine layers composed with an arbitrary elementwise differentiable (almost everywhere) non-linearity $f$. There are $M$ such layers, each mapping their vector-valued input $\boldsymbol{h}_k$ to a pre-activation vector $\boldsymbol{a}_k$ via a weight matrix $\boldsymbol{W}^{(k)}$ which is then transformed via $f$ into $\boldsymbol{h}_{k+1}$. The input vector $\boldsymbol{x}$ corresponds to $\boldsymbol{h}_0$ and the predicted outputs $\hat{\boldsymbol{y}}$ corresponds to $\boldsymbol{h}_M$. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on a target $\boldsymbol{y}$ (see Section 6.3.2 for examples of loss functions). The loss may be added to a regularizer $\Omega$ (see Section 6.3.3 and Chapter 7) to obtain the example-wise cost $J$. Algorithm 6.2 shows how to compute gradients of $J$ with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For computational efficiency on modern computers (especially GPUs), it is important to implement these equations on minibatches. Rather than using a vector $\boldsymbol{h}^{(k)}$ (and similary $\boldsymbol{a}^{(k)}$) to represent the activations on one example, an efficient implementation should use a matrix with the additional dimension representing the index of an example within the minibatch. Accordingly, $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ should have an additional dimension for the example index in the minibatch. The cost function $J$ remains a scalar because it is the average cost across examples in the minibatch.

$\boldsymbol{h}_0 = \boldsymbol{x}$
**for** $k = 1 \ldots, M$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
**end for**
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(M)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega$

continue computing the gradients of lower level hidden units one at a time in the same way The gradients on hidden and output units can be used to compute the gradient of $J$ with respect to the parameters (e.g. weights and biases) of each layer (i.e., that directly contribute to the output of that layer).

Algorithm 6.1 describes in matrix-vector form the forward propagation computation for a classical multi-layer network with $M$ layers, where each layer computes an affine transformation (defined by a bias vector $\boldsymbol{b}^{(k)}$ and a weight matrix $\boldsymbol{W}^{(k)}$) followed by a non-linearity $f$. In general, the non-linearity may be different on different layers. Typically at least the output layer has a different type than the other layers (see Section 6.3.1). Each unit at layer $k$ computes an output $\boldsymbol{h}_i^{(k)}$

**Algorithm 6.2** *Backward* computation for the deep neural network of Algorithm 6.1, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer $k$, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods. Note that although this algorithm as presented has pedagogical value, it is not a flexible way of implementing back-propagation because it is tied to the particulars of the network architecture and computation. Instead, consider the generalized form of back-propagation described in Sec. 6.4.3 below, which can accommodate any computational flow graph.

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, y) + \lambda \nabla_{\hat{\boldsymbol{y}}} \Omega$
(typically $\Omega$ is only a function of parameters not activations, so the last term would be zero)
**for** $k = M$ down to 1 **do**
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^k} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, where needed):
  $\nabla_{\boldsymbol{b}^k} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^k} \Omega$
  $\nabla_{\boldsymbol{W}^k} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^k} \Omega$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^k} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
**end for**

as follows:

$$
\begin{aligned}
a_i^{(k)} &= b_i^{(k)} + \sum_j W_{ij}^{(k)} \, h_j^{(k-1)} \\
h_i^{(k)} &= f(a^{(k)})v
\end{aligned}
\tag{6.13}
$$

where we separate the affine transformation from the non-linear activation operations for ease of exposition of the back-propagation computations.

These are described in matrix-vector form by Algorithm 6.2 and proceed from the output layer towards the first hidden layer, as outlined above. Note that this way of implementing back-propagation is not flexible enough to accommodate changes in architecture, and is considered obsolete, replaced by the generic form of back-propagation with automatic differentation, described next, and which is not specific to the machine learning context.

### 6.4.3 Back-Propagation in a General Flow Graph

In this section we call the intermediate quantities between inputs (parameters $\boldsymbol{\theta}$) and output (cost $J$) of the graph nodes $u_j$ (indexed by $j$) and consider the general case in which they form a directed acyclic graph that has $J$ as its final node $u_N$, that depends of all the other nodes $u_j$. The back-propagation algorithm exploits the chain rule for derivatives to compute $\frac{\partial J}{\partial u_j}$ when $\frac{\partial J}{\partial u_i}$ has already been computed for successors $u_i$ of $u_j$ in the graph, e.g., the hidden units in the next layer downstream. This recursion can be initialized by noting that $\frac{\partial J}{\partial u_N} = \frac{\partial J}{\partial J} = 1$ and at each step only requires to use the partial derivatives associated with each arc of the graph, $\frac{\partial u_i}{\partial u_j}$, when $u_i$ is a successor of $u_j$.

**Algorithm 6.3** Flow graph *forward* computation. Each node computes numerical value $u_i$ by applying a function $f_i$ to its argument list $\boldsymbol{a}^{(i)}$ that comprises the values of previous nodes $u_j$, $j < i$, with $j \in \text{parents}(i)$. The input to the flow graph is the vector $\boldsymbol{x}$, and is set into the first $M$ nodes $u_1$ to $u_M$. The output of the flow graph is read off the last (output) node $u_N$.

**for** $i = 1 \ldots, M$ **do**
  $u_i \leftarrow x_i$
**end for**
**for** $i = M+1 \ldots, N$ **do**
  $\boldsymbol{a}^{(i)} \leftarrow (u_j)_{j \in \text{parents}(i)}$
  $u_i \leftarrow f_i(\boldsymbol{a}^{(i)})$
**end for**
**return** $u_N$

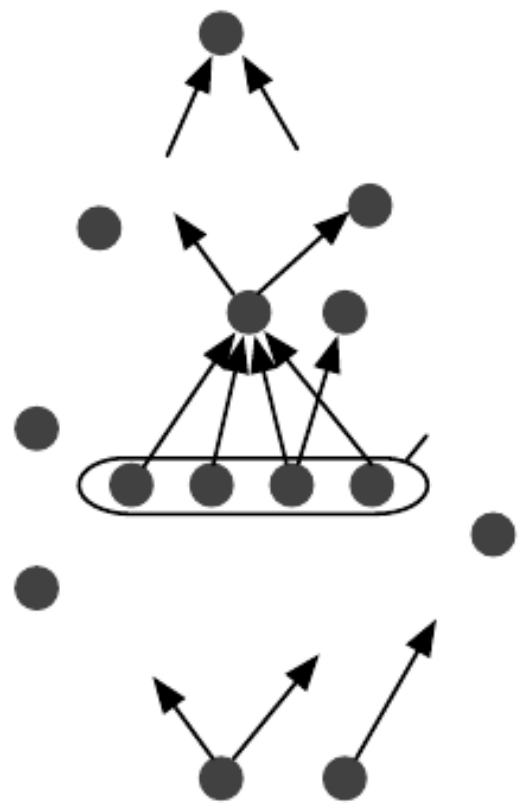

Figure 6.6: Illustration of recursive forward computation, where at each node $u$ we compute a value $u = f(\boldsymbol{a})$, with $\boldsymbol{a}$ being the list of values from parents $u$ of node $u$. Following Algorithm 6.3, the overall inputs to the graph are $u \ldots, u$ (e.g., the parameters we may want to tune during training), and there is a single scalar output $u$ (e.g., the loss which we want to minimize).

More generally than multi-layered networks, we can think about decomposing a function $J(\boldsymbol{\theta})$ into a more complicated graph of computations. This graph is called a **flow graph**. Each node $u_i$ of the graph denotes a numerical quantity that is obtained by performing a computation requiring the values $u_j$ of other nodes, with $j < i$. The nodes satisfy a partial order which dictates in what order the computation can proceed. In practical implementations of such functions (e.g. with the criterion $J(\boldsymbol{\theta})$ or its value estimated on a minibatch), the final computation is obtained as the *composition of simple functions* taken from a given set (such as the set of numerical operations that the library can perform on arrays of numbers).

We will define the back-propagation in a general flow-graph, using the following generic notation: $u_i = f_i(\boldsymbol{a}^{(i)})$, where $\boldsymbol{a}^{(i)}$ is a list of arguments for the application of $f_i$ to the values $u_j$ for the parents of $i$ in the graph: $\boldsymbol{a}^{(i)} = (u_j)_{j \in \text{parents}(i)}$. This is illustrated in Figure 6.6.

The overall computation of the function represented by the flow graph can thus be summarized by the forward computation algorithm, Algorithm 6.3.

In addition to having some code that tells us how to compute $f_i(\boldsymbol{a}^{(i)})$ for some values in the vector $\boldsymbol{a}^{(i)}$, we also need some code that tells us how to compute its partial derivatives, $\frac{\partial f_i(\boldsymbol{a}^{i})}{\partial a_{ik}}$ with respect to any immediate argument $a_{ik}$. Let

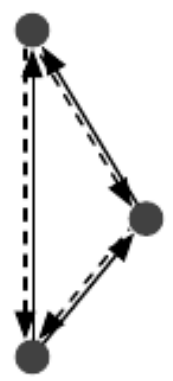

Figure 6.7: Illustration of indirect effect and direct effect of variable $u$ on variable $u$ in a flow graph, which means that the derivative of $u$ with respect to $u$ must include the sum of two terms, one for the direct effect (derivative of $u$ with respect to its first argument) and one for the indirect effect through $u$ (involving the product of the derivative of $u$ with respect to $u$ times the derivative of $u$ with respect to $u$ ). Forward computation of $u$ 's (as in Figure 6.6) is indicated with upward full arrows, while backward computation (of derivatives with respect to $u$ 's, as in Figure 6.8) is indicated with downward dashed arrows.

$k = \pi(i, j)$ denote the index of $u_j$ in the list $\boldsymbol{a}^{(i)}$. Note that $u_j$ could influence $u_i$ through multiple paths. Whereas $\frac{\partial u_i}{\partial u_j}$ would denote the total gradient adding up all of these influences, $\frac{\partial f_i(\boldsymbol{a}^{(i)})}{\partial a_{ik}}$ only denotes the derivative of $f_i$ with respect to its specific $k$-th argument, keeping the other arguments fixed, i.e., only considering the influence through the arc from $u_j$ to $u_i$.

In general, when manipulating partial derivatives, it is important to distinguish two kinds of derivatives: those that consider all the other arguments fixed (this is typically what we mean by partial derivative), and those that allow for indirect influences going through other arguments. The latter is typically what we mean by total derivatives, except that in our case there will simultaneously be multiple such arguments (the different parameters, or source nodes of the graph) for which we wish to compute the total derivative, holding the other source nodes fixed. For that reason, we still use the partial derivative notation for these, but we need to be aware of the different kinds of derivatives we are handling, which will depend on what type of node (internal or source) is under consideration.

For example consider $f_3(a_{3,1}, a_{3,2}) = e^{a_{3,1}+a_{3,2}}$ and $f_2(a_{2,1}) = a_{2,1}^2$, while $u_3 = f_3(u_2, u_1)$ and $u_2 = f_2(u_1)$, illustrated in Figure 6.7. The direct derivative of $f_3$ with respect to its argument $a_{3,2}$ (keeping $a_{3,1}$ fixed) is $\frac{\partial f_3}{\partial a_{3,2}} = e^{a_{3,1}+a_{3,2}}$. On the other hand, if we consider the variables $u_3$ and $u_1$ to which these arguments correspond, there are two paths from $u_1$ to $u_3$, and we obtain as the total derivative the sum of partial derivatives over these two paths, $\frac{\partial u_3}{\partial u_1} = e^{u_1+u_2}(1 + 2u_1)$. The results are different because $\frac{\partial u_3}{\partial u_1}$ involves not just the direct dependency of $u_3$ on $u_1$ but also the indirect dependency through $u_2$.

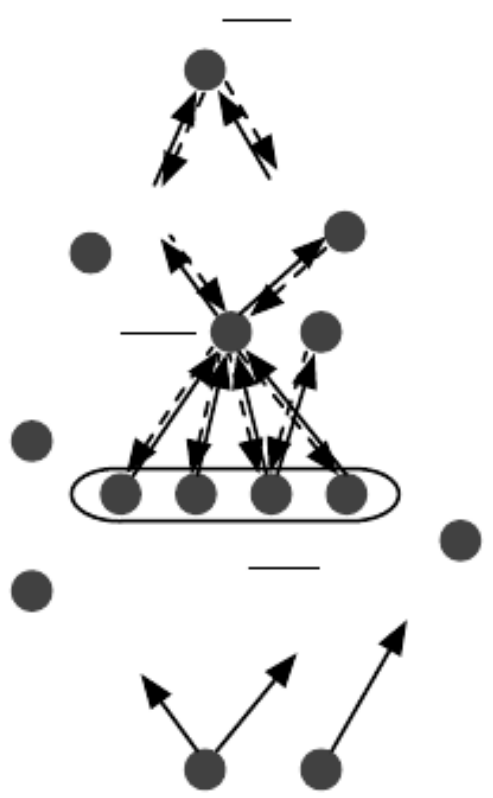

Figure 6.8: Illustration of recursive backward computation, where we associate to each node $j$ not just the values $u$ computed in the forward pass (Figure 6.6, bold upward arrows) but also the gradient —— with respect to the output scalar node $u$ . These gradients are recursively computed in exactly the opposite order, as described in Algorithm 6.4 by using the already computed —— of the children $i$ of $j$ (dashed downward arrows).

Armed with this understanding, we can define the back-propagation algorithm as follows, in Algorithm 6.4, which would be computed *after* the forward propagation (Algorithm 6.3) has been performed. Note the recursive nature of the application of the chain rule, in Algorithm 6.4: we compute the gradient on node $j$ by re-using the already computed gradient for children nodes $i$, starting the recurrence from the trivial $\frac{\partial u_N}{\partial u_N} = 1$ that sets the gradient for the output node. This is illustrated in Figure 6.8.

This recursion is a form of efficient factorization of the total gradient, i.e., it is an application of the principles of dynamic programming[8]. Indeed, the derivative of the output node with respect to any node can also be written down in this intractable form:

$$\frac{\partial u_N}{\partial u_i} = \sum_{\text{paths } u_{k_1}, \ldots, u_{k_n}:\, k_1 = i, k_n = N} \prod_{j=2}^{n} \frac{\partial u_{k_j}}{\partial u_{k_j}}$$

where the paths $u_{k_1}, \ldots, u_{k_n}$ go from the node $k_1 = i$ to the final node $k_n = N$

Here we refer to "dynamic programming" in the sense of table-filling algorithms that avoid re-computing frequently used subexpressions. In the context of machine learning, "dynamic programming" can also refer to iterating Bellman's equations. That is not the kind of dynamic programming we refer to here.

**Algorithm 6.4** *Back-propagation* computation of a flow graph (full, upward arrows, Figs.6.8 and 6.6), which itself produces an additional flow graph (dashed, backward arrows). See the forward propagation in a flow-graph (Algorithm 6.3, to be performed first) and the required data structure. In addition, a quantity $\frac{\partial u_N}{\partial u_i}$ needs to be stored (and computed) at each node, for the purpose of gradient back-propagation. Below, the notation $\pi(i,j)$ is the index of $u_j$ as an argument to $f_i$. The back-propagation algorithm efficiently computes $\frac{\partial u_N}{\partial u_i}$ for all $i$'s (traversing the graph backwards this time), and in particular we are interested in the derivatives of the output node $u_N$ with respect to the "inputs" $u_1 \ldots, u_M$ (which could be the parameters, in a learning setup). The cost of the overall computation is proportional to the number of arcs in the graph, assuming that the partial derivative associated with each arc requires a constant time. This is of the same order as the number of computations for the forward propagation.

$\frac{\partial u_N}{\partial u_N} \leftarrow 1$
**for** $j = N-1$ down to 1 **do**
  $\frac{\partial u_N}{\partial u_j} \leftarrow \sum_{i:j \in \text{parents}(i)} \frac{\partial u_N}{\partial u_i} \frac{\partial f_i(a_i)}{\partial a_{i,\pi(i,j)}}$
**end for**
**return** $\left(\frac{\partial u_N}{\partial u_i}\right)_{i=1}^{M}$

in the flow graph and $\frac{\partial u_{k_j}}{\partial u_{k_j}}$ refers only to the immediate derivative considering $u_{k_j}$ as the argument number $\pi(k_j, k_{j-1})$ of $a_{k_j}$ into $u_{k_j}$, i.e.,

$$\frac{\partial u_{k_j}}{\partial u_{k_j}} = \frac{\partial f_{k_j}(a_{k_j})}{\partial a_{k_j,\pi(k_j,k_j)}}.$$

Computing the sum as above would be intractable because the number of possible paths can be exponential in the depth of the graph. The back-propagation algorithm is efficient because it employs a dynamic programming strategy to reuse rather than re-compute partial sums associated with the gradients on intermediate nodes.

Although the above was stated as if the $u_i$'s were scalars, exactly the same procedure can be run with $u_i$'s being tuples of numbers (more easily represented by vectors). The same equations remain valid. Earlier, we wrote these equations using scalar multiplication of scalar partial derivatives. Using vectors, these multiplications turn into matrix-vector products. We multiply the row vector of gradients $\frac{\partial u_N}{\partial u_i}$ by a Jacobian matrix of partial derivatives associated with the $j \to i$ arc of the graph, $\frac{\partial f_i(a_i)}{\partial a_{i,\pi(i,j)}}$.

The quintessential neural network training scenario is the case where each $^{(i)}$ is a matrix containing $m$ examples in a minibatch and $n$ hidden unit activa-

tion values. In this case, both forward propagation and back-propagation can be expressed as a product between a matrix of activations or gradients and a matrix of weights. From a computational point of view, this is much more efficient than training on a single example at a time. When we do not use the minibatch version of the algorithm, $^{(i)}$ is only a vector. The main operation used in forward and back-propagation is then a matrix-vector product, between the weight matrix and the vector of activations or gradients. Matrix-matrix products are typically implemented with a high degree of parallel computation (e.g. in BLAS library implementations) which is essential for obtaining good performance on modern multicore CPUs and GPUs. Matrix-matrix products for minibatch forward and back-propagation allow parallelization across both examples and units, while matrix-vector products for the processing of a single example allow only parallelization across units.

### 6.4.4 Symbolic Back-propagation and Automatic Differentiation

The algorithm for generalized back-propagation (Alg. 6.4) was presented with the interpretation that actual computations take place at each step of the algorithm. This generalized form of back-propagation is just a particular way to perform *automatic differentiation* (Rall, 1981) in computational flow graphs defined by Algorithm 6.3. Automatic differentiation automatically obtains derivatives of a given expression and has numerous uses in machine learning (Baydin *et al.*, 2015). As an alternative (and often as a debugging tool) derivatives could be obtained by numerical methods based on measuring the effects of small changes, called *numerical differentiation* (Lyness and Moler, 1967). For example, a finite difference approximation of the gradient follows from the definition of derivative as a ratio of the change in output that results in a change in input, divided by the change in input. Methods based on random perturbations also exist which randomly jiggle all the input variables (e.g. parameters) and associate these random input changes with the resulting overall change in the output variable in order to estimate the gradient (Spall, 1992).

However, for obtaining a gradient (i.e., with respect to many variables, e.g., parameters of a neural network), back-propagation has two advantages over numerical differentiation: (1) it performs exact computation (up to machine precision), and (2) it is computationally much more efficient, obtaining all the required derivatives in one go. Instead, numerical differentiation methods either require to redo the forward propagation separately for each parameter (keeping the other ones fixed) or they yield stochastic estimators (from a random perturbation of all parameters) whose variances grows linearly with the number of parameters. Automatic differentiation of a function with $d$ inputs and $m$ outputs can be done either by carrying derivatives forward, known as forward mode computation, or

carrying them backwards, known as backward mode computation. The former is more efficient when $d < m$ and the latter is more efficient when $d > m$. In our use case, the output is a scalar (the cost), and the backward approach, also called reverse accumulation. Hence for machine learning applications where we want to compute gradients (partial derivatives with respect to a scalar cost), back-propagation is much more efficient than the approach of propagating derivatives forward in the graph.

Although Algorithm 6.4 can thus be seen as a particular form of automatic differentiation, its implementation can be done in different ways, the most general one involving *symbolic differentiation*. Symbolic differentation exploits our knowledge of how to compute derivatives of elementary operations, such as arithmetic operations, or any computation performed in the computer and specified by a symbolic expression. Symbolic differentiation takes a symbolic expression (for computing a function) and returns another symbolic expression (for computing the required derivatives). Automatic differentiation techniques such as the forward or backward mode allow one to perform these computations efficiently, avoiding a potentially exponential blow-up in the size of the computational graph for computing the derivatives.

The popular library[9] for deep learning, as well as most other open source deep learning libraries are doing a limited form of automatic differentiation restricted to the "programs" obtained by composing a predefined set of operations, each corresponding to a "module". The set of these modules is designed such that many neural network architectures and computations can be performed by composing the building blocks represented by each of these modules. Each module is defined by two main functions:

1. One that computes the outputs $\boldsymbol{y}$ of the module given its inputs $\boldsymbol{x}$, e.g., with an " " function

$$\boldsymbol{y} = \text{module.} \qquad (\boldsymbol{x}),$$

   and the input $(\boldsymbol{x})$, output $(\boldsymbol{y})$ and potentially intermediate results of this computation are *stored in the module*.

2. One that computes the gradient $\frac{\partial J}{\partial \boldsymbol{x}}$ of a scalar (typically the minibatch cost $J$) with respect to the inputs $\boldsymbol{x}$, given the gradient $\frac{\partial J}{\partial \boldsymbol{y}}$ with respect to the outputs, e.g., with a " " function

$$\nabla_{\boldsymbol{x}} J = \text{module.} \qquad (\nabla_{\boldsymbol{y}} J)\,.$$

---

See .

The function thus implicitly knows the Jacobian of the $\boldsymbol{x}$ to $\boldsymbol{y}$ mapping, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$, at $\boldsymbol{x}$. Specifically, the function specifies how to multiply this Jacobian by a vector passed to the function as an argument. It also needs to have access to the value of $\boldsymbol{x}$ that was previously fed as input to , so that the Jacobian is computed at that value. For avoiding recomputations, $\boldsymbol{y}$ and other values that were computed in are also typically stored where the module has access to these values.

During execution of the back-propagation algorithm, will be called with $\nabla_{\boldsymbol{y}} J$ given as argument. When called in this manner, computes

$$\nabla_{\boldsymbol{x}} J = \left( \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^{\top} \nabla_{\boldsymbol{y}} J.$$

In practice, implementations work in parallel over a whole minibatch (transforming matrix-vector operations into matrix-matrix operations) and may operate on objects which are not vectors (maybe higher-order tensors like those involved with images or sequences of vectors). Furthermore, the function does not have to explicitly compute the Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ and perform an actual matrix multiplication: it can do that matrix multiplication implicitly, which is often more efficient. For example, if the true Jacobian is diagonal, then the actual number of computations required is much less than the size of the Jacobian matrix.

To keep computations efficient and avoid the overhead of the glue required to compose modules together, neural net packages such as define modules that perform coarse-grained operations such as the cross-entropy loss, a convolution, the affine operation associated with a fully-connected neural network layer, or a softmax. It means that if one wants to write differentiable code for some computation that is not covered by the existing set of modules, one has to write their own code for a new module, providing both the code for and the code for . This is in contrast with standard automatic differentiation systems, which know how to compute derivatives through all the operations in a general-purpose programming language such as C.

This is how the (Bergstra *et al.*, 2010b; Bastien *et al.*, 2012) library[10] handles derivatives. It actually generates a symbolic representation of the gradient computation that is expressed in the language as the symbolic representation of the function to be differentiated. It therefore performs symbolic differentation and automatic differentiation, and can compute higher-order derivatives by repeated calls to the differentation operator.

Like , Theano only covers a predefined set of operations (i.e., a language that is a subset of usual programming languages), but it is a much larger and fine-grained set of operations, covering most of the operations on tensors

[10] See .

and linear algebra defined in Python's library of numerical computation. It is thus very rare that a user would need to write a new module for , except if they want to provide an alternative implementation (say, more efficient or numerically stable in some cases). Another immediate advantage of is that it can take advantage of the other tools of *symbolic computation* (Buchberger *et al.*, 1983), such as simplification (to make computation faster and more memory-efficient) and transformations that make the computation more numerically stable (Bergstra *et al.*, 2010b). These simplification operations make it still very efficient in terms of computation and memory usage even with a set of fine-grained operations such as individual tensor additions and multiplications. also provides a *compiler* of the resulting expressions into C for CPUs and GPUs, i.e., the same high-level expression can be implemented in different ways depending of the underlying hardware.

## 6.5 Back-propagation through Random Operations and Graphical Models

Whereas traditional neural networks perform deterministic computation, they can be extended to perform stochastic computation. In this case, we can think of the network as defining a sampling process that deterministically transforms some random values. Provided the computations are continuous and differentiable, we can then generally apply backpropagation as usual, with the underlying random values as inputs to the network.

As an example, let us consider the operation consisting of drawing samples z from a Gaussian distribution with mean $\mu$ and variance $\sigma^2$:

$$\mathrm{z} \sim \mathcal{N}(\mu, \sigma^2).$$

Because an individual sample of z is not produced by a function, but rather by a sampling process whose output changes every time we query it, it may seem counterintuitive to take the derivatives of z with respect to the parameters of its distribution, $\mu$ and $\sigma^2$. However, we can rewrite the sampling process as transforming an underlying random value $\eta \sim \mathcal{N}(0, 1)$ to obtain a sample from the desired distribution:

$$z = \mu + \sigma\eta \tag{6.14}$$

We are now able to backpropagate through the sampling operation, by regarding it as a deterministic operation with an extra input $\eta$. Crucially, the extra input is a random variable whose distribution is not a function of any of the variables whose derivatives we want to calculate. The result tells us how an infinitesimal change in $\mu$ or $\sigma$ would change the output if we could repeat the sampling operation again with the same value of $\eta$.

Being able to backpropagate through this sampling operation allows us to incorporate it into a larger graph; e.g. we can compute the derivatives of some loss function $J(z)$. Moreover, we can introduce functions that shape the distribution, e.g. $\mu = f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\sigma = g(\boldsymbol{x}; \boldsymbol{\theta})$ and use back-propagation through this functions to derive $\nabla_{\boldsymbol{\theta}} J(z)$.

The principle used in this Gaussian sampling example is true in general: given a value $z$ sampled from distribution $p(\mathrm{z} \mid \boldsymbol{\omega})$ whose parameters $\boldsymbol{\omega}$ may depend on other quantities of interest, we can rewrite

$$z \sim p(\mathrm{z} \mid \boldsymbol{\omega})$$

as

$$z = f(\boldsymbol{\omega}, \boldsymbol{\eta})$$

where $\boldsymbol{\eta}$ is a source of randomness that is independent of any of the variables that influence $\boldsymbol{\omega}$.

Gradient-based optimization can then be applied, so long as $z$ is continuous-valued, or more precisely when $f$ is continuous and differentiable almost everywhere. Otherwise, gradient-based optimization will be blind to the effect of any parameters that influence the cost via $z$ and $\boldsymbol{\omega}$.

When $z$ and $f$ are discrete-valued, it may still be possible to estimate a gradient on $\boldsymbol{\omega}$, using reinforcement learning algorithms such as variants of the REINFORCE algorithm (Williams, 1992), discussed below.

In neural network applications, we typically choose $\boldsymbol{\eta}$ to be drawn from some simple distribution, such as a unit uniform or unit Gaussian distribution, and achieve more complex distributions by allowing the deterministic portion of the network to reshape its input. This is actually how the random generators for parametric distributions are implemented in software, by performing operations on approximately independent sources of noise (such as random bits). So long as the function $f$ in the above equation is differentiable with respect to $\boldsymbol{\omega}$, we can back-propagate through the sampling operation.

### 6.5.1 Back-propagating through discrete stochastic operations

The idea of propagating gradients or optimizing through stochastic operations is old (Price, 1958; Bonnet, 1964), first used for machine learning in the context of reinforcement learning (Williams, 1992), variational approximations (Opper and Archambeau, 2009), and more recently, stochastic or generative neural networks (Bengio *et al.*, 2013a; Kingma, 2013; Kingma and Welling, 2014b,a; Rezende *et al.*, 2014; Goodfellow *et al.*, 2014c). Many networks, such as denoising autoencoders or networks regularized with dropout, are also naturally designed to take

noise as an input without requiring any special reparameterization to make the noise independent from the model.

When $z$ is discrete, for a given sampled value of the injected noise $\boldsymbol{\eta}$, a small change of $\boldsymbol{\omega}$ would generally not change $z$, and thus would not change our loss function $J$, so straightforward back-propagation is not applicable. Since the early days of research on propagating gradients through stochastic operations, that case was studied, mostly using reinforcement learning ideas. A simple but powerful set of solutions is the REINFORCE algorithm (Williams, 1992). The idea is that $J$ generally becomes a continuous function of $\boldsymbol{\omega}$ when we average over the possible samples of the injected noise $\boldsymbol{\eta}$. Although that sum is typically not tractable when $z$ is high-dimensional (or many discrete stochastic decisions are taken), it can be estimated unbiasedly using a Monte-Carlo average, and that gives us a stochastic gradient which can be used with SGD or other stochastic gradient-based optimization techniques.

The simplest version of REINFORCE can be derived by simply differentiating the integral of interest:

$$
\begin{aligned}
E[J(z)] &= \sum_z J(z)p(z) \\
\frac{\partial E[J(z)]}{\partial \boldsymbol{\omega}} &= \sum_z J(z)\frac{\partial p(z)}{\partial \boldsymbol{\omega}} \\
&= \sum_z J(z)p(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}} \\
&\approx \frac{1}{N} \sum_{z_i \sim p(z),\, i=1}^{N} J(z_i)\frac{\partial \log p(z_i)}{\partial \boldsymbol{\omega}}
\end{aligned}
\tag{6.15}
$$

where the second line is true because we have assumed that $z$ is discrete, which means that the derivative of $f$ is zero, the third line exploits the derivative rule for the logarithm, $\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}} = \frac{1}{p(z)}\frac{\partial p(z)}{\partial \boldsymbol{\omega}}$, and the last line gives as an unbiased Monte-Carlo estimator of the gradient. For example, if $z$ consists of a set of binomial random variables $z_i$ which are independent of each other given $\boldsymbol{\omega}$, with $p_i = p(z_i = 1|\boldsymbol{\omega}) = \text{sigmoid}(\boldsymbol{\omega}_i)$, then $\frac{\log p(z)}{\partial \boldsymbol{\omega}_i}$ is the binomial cross-entropy gradient, $z_i(1-p_i) + (1-z_i)p_i$. [11]

One issue with the above simple REINFORCE estimator is that it has a very high variance, so that many samples of $z$ need to be drawn to obtain a good estimator of the gradient, or equivalently, if only one sample is drawn, SGD will converge very slowly and will require a smaller learning rate. It is possible to

Note that ( ) depends on and is typically input-dependent, so that ( ) is different for each input . Everywhere we write ( ) in this section, we might as well write ( ) or ( ).

considerably reduce the variance of that estimator by using *variance reduction* methods (Wilson, 1984; L'Ecuyer, 1994). The idea is to modify the estimator so that its expected value remains unchanged but its varianced get reduced. In the context of REINFORCE, the proposed variance reduction methods involve the computation of a *baseline* that is used to offset $J(z)$. Note that any offset $b(\boldsymbol{\omega})$ that does not depend on $z$ would not change the expectation of the estimated gradient because

$$\begin{aligned} E_{p(z)}\left[\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}}\right] &= \sum_z p(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}} \\ &= \sum_z \frac{p(z)}{\partial \boldsymbol{\omega}} \\ &= \frac{\partial}{\partial \boldsymbol{\omega}} \sum_z p(z) = \frac{\partial}{\partial \boldsymbol{\omega}} 1 = 0, \end{aligned} \tag{6.16}$$

which means that

$$\begin{aligned} E_{p(z)}\left[(J(z) - b(\boldsymbol{\omega}))\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}}\right] &= E_{p(z)}\left[J(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}}\right] - b(\boldsymbol{\omega})E_{p(z)}\left[\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}}\right] \\ &= E_{p(z)}\left[J(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}}\right]. \end{aligned} \tag{6.17}$$

Furthermore, we can obtain the optimal $b(\boldsymbol{\omega})$ by computing the variance of $(J(z) - b(\boldsymbol{\omega}))\frac{\log p(z)}{\partial \boldsymbol{\omega}}$ under $p(z)$ and minimizing with respect to $b(\boldsymbol{\omega})$. What we find[12] is that this optimal baseline $b_i^*(\boldsymbol{\omega})$ is different for each element $\omega_i$ of the vector $\boldsymbol{\omega}$:

$$b_i^*(\boldsymbol{\omega}) = \frac{E_{p(z)}\left[J(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2\right]}{E_{p(z)}\left[\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2\right]}. \tag{6.18}$$

The gradient estimator with respect to $\omega_i$ then becomes

$$(J(z) - b_i(\boldsymbol{\omega}))\frac{\partial \log p(z)}{\partial \omega_i}$$

where $b_i(\boldsymbol{\omega})$ estimates the above $b_i^*(\boldsymbol{\omega})$. This can be done by using additional outputs for the neural network that computes $\boldsymbol{\omega}$. In addition, it outputs an estimator of $E_{p(z)}[J(z)\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2]$ and an estimator of $E_{p(z)}\left[\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2\right]$ for each element of $\boldsymbol{\omega}$. These extra outputs can be trained with the mean squared error objective, using respectively $\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2$ and $\frac{\partial \log p(z)}{\partial \boldsymbol{\omega}_i}^2$ as targets when $z$ is sampled

this is left as an exercise

from $p(z)$, for a given $\boldsymbol{\omega}$. Mnih and Gregor (2014) preferred to use a single shared output (across all elements $i$ of $\boldsymbol{\omega}$) trained with the target $J(z)$, using as baseline $b(\boldsymbol{\omega}) \approx E_{p(z)}[J(z)]$.

Variance reduction methods have been introduced in the reinforcement learning context (Sutton *et al.*, 2000; Weaver and Tao, 2001), generalizing previous work on the case of binary reward by Dayan (1990). See Bengio *et al.* (2013a); Mnih and Gregor (2014); Ba *et al.* (2014); Mnih *et al.* (2014); Xu *et al.* (2015a) for examples of modern uses of the REINFORCE algorithm with reduced variance in the context of deep learning. In addition to the use of an input-dependent baseline $b(\boldsymbol{\omega})$, Mnih and Gregor (2014) found that the scale of $(J(z) - b(\boldsymbol{\omega}))$ could be adjusted during training by dividing it by its standard deviation estimated by a moving average during training, as a kind of adaptive learning rate, to counter the effect of important variations that occur during the course of training in the magnitude of this quantity. Mnih and Gregor (2014) called this heuristic *variance normalization*.

Once we have estimated the gradient of the expected loss with respect to $\boldsymbol{\omega}$, we can back-propagate it as usual in the upstream parts of the computational graph that lead to $\boldsymbol{\omega}$, in order to obtain an estimated gradient over the variables of interest (typically parameters of the model). However, REINFORCE-based estimators remain fairly noisy and can be understood as estimating the gradient by correlating choices of $z$ with corresponding values of $J(z)$. If a good value of $z$ is unlikely under the current parametrization, it might take a long time to obtain it by chance, and get the required signal that this configuration should be reinforced.

## 6.6 Universal Approximation Properties and Depth

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in a convex optimization problem when applied to linear models. Unfortunately, we often want to learn non-linear functions.

At first glance, we might presume that learning a non-linear function requires designing a specialized model family for the kind of non-linearity we want to learn. However, it turns out that feedforward networks with hidden layers provide a universal approximation framework. Specifically, the *universal approximation theorem* (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another

with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990). The concept of Borel measurability is beyond the scope of this book; for our purposes it suffices to say that any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another. Interestingly, universal approximation theorems have also been proven for a wider class of non-linearities which includes the now commonly used rectified linear unit (Leshno *et al.*, 1993). The results on the expressive power of RBMs also tell us about networks of rectified linear units (Martens *et al.*, 2013).

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. However, we are not guaranteed that the training algorithm will be able to *learn* that function. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function due to overfitting. Recall from Chapter 5.3.1 that the "no free lunch" theorem shows that there is no universal machine learning algorithm. Even though feedforward networks provide a universal system for representing functions, there is no universal procedure for examining a training set and choosing the right set of functions among the family of functions our approximator can represent: there could be many functions within our family that fit well the data and we need to choose one (this is basically the overfitting scenario).

Another but related problem facing our universal approximation scheme is the size of the model needed to represent a given function. The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but it does not say how large this network will be. Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions. Unfortunately, in the worse case, an exponential number of hidden units (to basically record every input configuration that needs to be distinguished) may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $\boldsymbol{v} \in \{0, 1\}^n$ is $2^{2^n}$ and selecting one such function requires $2^n$ bits, which will in general require $O(2^n)$ degrees of freedom.

In summary, a feedforward network with a single layer is sufficient to represent any function, but it may be infeasibly large and may fail to learn and generalize correctly. Both of these failure modes suggest that we may want to use deeper

models.

First, we may want to choose a model with more than one hidden layer in order to avoid needing to make the model infeasibly large. There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value $d$, but require a much larger model if depth is restricted to be less than or equal to $d$. In many cases, the number of hidden units required by the shallow model is exponential in $n$. Such results have been first proven for circuits of logic gates (Håstad, 1986), then for linear threshold units with non-negative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), which are more like the actual neural networks we care about. This result was extended by Maass (1992); Maass *et al.* (1994), showing advantages of using continuous-valued activations over binary ones. Results about the existence of functions that are difficult to represent with a shallow polynomial circuit (Delalleau and Bengio, 2011) were also shown, which may be interesting because of the proposal to use such circuits to represent distributions, called sum-product networks or SPNs (Poon and Domingos, 2011). Note that the probabilistic interpretation of SPNs includes constraints that may limit their expressive power (Martens and Medabalimi, 2014). This last result also proves the existence of a depth hierarchy for SPNs. More recently, following on results demonstrating the universal approximation properties of shallow networks of rectifier units (Leshno *et al.*, 1993), results were obtained about the expressive power of deep networks of rectifier units (Pascanu *et al.*, 2013b; Montufar *et al.*, 2014). These showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network.

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better

generalization for a wide variety of tasks (Bengio *et al.*, 2007b; Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Goodfellow *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012b; Sermanet *et al.*, 2013; Farabet *et al.*, 2013a; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a). See Fig. 6.9 for an example of some of these empirical results. This suggests that using deep architectures does indeed express a useful prior over the space of functions the model learn.

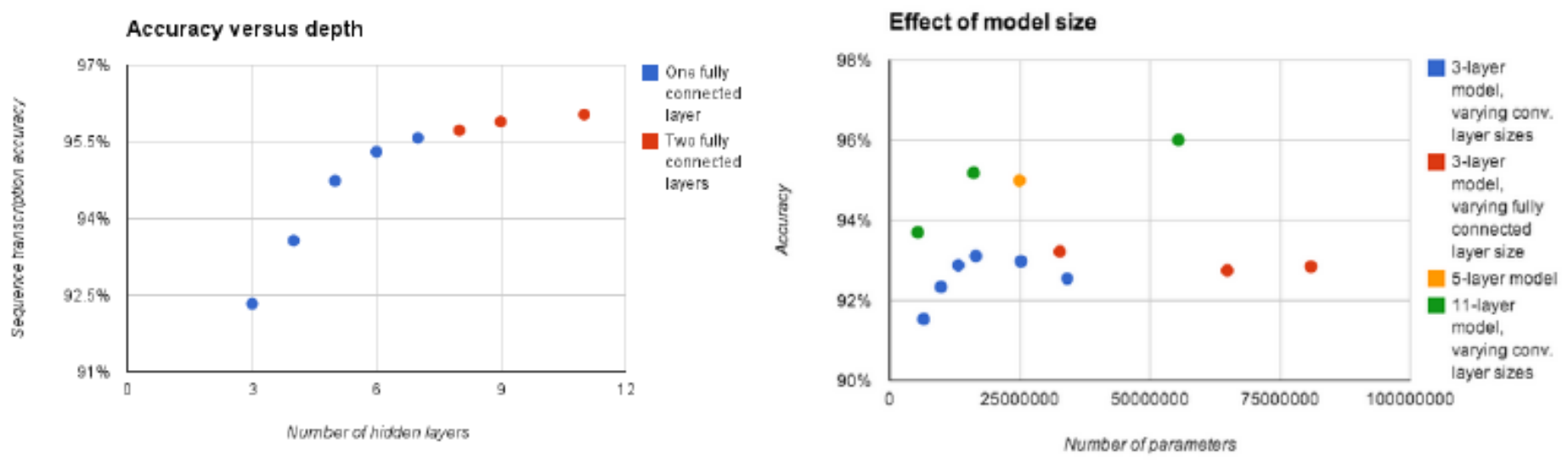


Figure 6.9: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Reproduced with permission from Goodfellow *et al.* (2014d). Left) The test set accuracy consistently increases with increasing depth. Right) This effect cannot be explained simply by the model being larger; one can also increase the model size by increasing the width of each layer. The test accuracy cannot be increased nearly as well by increasing the width, only by increasing the depth. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

## 6.7 Feature / Representation Learning

Let us consider again the single layer networks such as the perceptron, linear regression and logistic regression: such linear models are appealing because training them involves a convex optimization problem[13]. Convex optimization comes with some convergence guarantees towards a global optimum, irrespective of initial conditions. Simple and well-understood optimization algorithms are available

or even one for which an analytic solution can be computed, with linear regression or the case of some Gaussian process regression models

in this case. However, this limits the representational capacity too much: many tasks, for a given choice of input representation $\boldsymbol{x}$ (the raw input features), cannot be solved by using only a linear predictor. What are our options to avoid that limitation?

1. One option is to use a *kernel machine* (Williams and Rasmussen, 1996; Schölkopf *et al.*, 1999), i.e., to consider a fixed mapping from $\boldsymbol{x}$ to $\phi(\boldsymbol{x})$, where $\phi(\boldsymbol{x})$ is of much higher dimension. In this case, $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = b + w \cdot \phi(\boldsymbol{x})$ can be linear in the parameters (and in $\phi(x)$) and optimization remains convex (or even analytic). By exploiting the *kernel trick*, we can computationally handle a high-dimensional $\phi(\boldsymbol{x})$ (or even an infinite-dimensional one) so long as the kernel $k(\boldsymbol{u}, \boldsymbol{v}) = \phi(\boldsymbol{u}) \cdot \phi(\boldsymbol{v})$ (where $\cdot$ is the appropriate dot product for the space of $\phi(\cdot)$) can be computed efficiently. If $\phi(\boldsymbol{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization is not at all guaranteed: it will depend on the appropriateness of the choice of $\phi$ as a feature space for our task. Kernel machines theory clearly identifies the choice of $\phi$ to the choice of a prior. This leads to kernel engineering, which is equivalent to feature engineering, discussed next. The other type of kernel (that is very commonly used) embodies a very broad prior, such as smoothness, e.g., the Gaussian (or RBF) kernel $k(\boldsymbol{u}, \boldsymbol{v}) = \exp\left(-||\boldsymbol{u} - \boldsymbol{v}||/\sigma^2\right)$. Unfortunately, this prior may be insufficient, i.e., too broad and sensitive to the curse of dimensionality, as introduced in Section 5.12.1 and developed in more detail in Chapter 16.

2. Another option is to *manually engineer the representation or features* $\phi(\boldsymbol{x})$. Most industrial applications of machine learning rely on hand-crafted features and most of the research and development effort (as well as a very large fraction of the scientific literature in machine learning and its applications) goes into designing new features that are most appropriate to the task at hand. Clearly, faced with a problem to solve and some prior knowledge in the form of representations that are believed to be relevant, the prior knowledge can be very useful. This approach is therefore common in practice, but is not completely satisfying because it involves a very task-specific engineering work and a laborious never-ending effort to improve systems by designing better features. If there were some more general feature learning approaches that could be applied to a large set of related tasks (such as those involved in AI), we would certainly like to take advantage of them. Since humans seem to be able to learn a lot of new tasks (for which they were not programmed by evolution), it seems that such broad priors do exist. This whole question is discussed in more detail in Bengio and LeCun (2007a), and motivates the third option.

3. The third option is to *learn the features*, or *learn the representation*. In a sense, it allows one to interpolate between the almost agnostic approach of a kernel machine with a general-purpose smoothness kernel (such as RBF SVMs and other non-parametric statistical models) and full designer-provided knowledge in the form of a fixed representation that is perfectly tailored to the task. This is equivalent to the idea of *learning the kernel*, except that whereas most kernel learning methods only allow very few degrees of freedom in the learned kernel, representation learning methods such as those discussed in this book (including multi-layer neural networks) allow the feature function $\phi(\cdot)$ to be very rich (with a number of parameters that can be in the millions or more, depending on the amount of data available). This is equivalent to *learning the hidden layers*, in the case of a multi-layer neural network. Besides smoothness (which comes for example from regularizers such as weight decay), other priors can be incorporated in this feature learning. The most celebrated of these priors is *depth*, discussed above (Section 6.6). Other priors are discussed in Chapter 16.

This whole discussion is clearly not specific to neural networks and supervised learning, and is one of the central motivations for this book.

## 6.8 Piecewise Linear Hidden Units

Most of the recent improvement in the performance of deep neural networks can be attributed to increases in computational power and the size of datasets. The machine learning algorithms involved in recent state-of-the-art systems have mostly existed since the 1980s, with a few recent conceptual advances contributing significantly to increased performance.

One of the main algorithmic improvements that has had a significant impact is the use of piecewise linear units, such as absolute value rectifiers and rectified linear units. Such units consist of two linear pieces and their behavior is driven by a single weight vector. Jarrett *et al.* (2009b) observed that "using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system" among several different factors of neural network architecture design.

For small datasets, Jarrett *et al.* (2009b) observed that using rectifying non-linearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, allowing the classifier layer at the top to learn how to map different feature vectors to class identities.

When more data is available, learning begins to extract enough useful knowledge to exceed the performance of randomly chosen parameters. Glorot *et al.*

(2011b) showed that learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions. Because the behavior of the unit is linear over half of its domain, it is easy for an optimization algorithm to tell how to improve the behavior of a unit, even when the unit's activations are far from optimal. Just as piecewise linear networks are good at propagating information forward, back-propagation in such a network is also piecewise linear and propagates information about the error derivatives to all of the gradients in the network. Each piecewise linear function can be decomposed into different regions corresponding to different linear pieces. When we change a parameter of the network, the resulting change in the network's activity is linear until the point that it causes some unit to go from one linear piece to another. Traditional units such as sigmoids are more prone to discarding information due to saturation both in forward propagation and in back-propagation. The response of such a network to a change in a single parameter may be highly nonlinear even in a small neighborhood.

Glorot *et al.* (2011b) motivate rectified linear units from biological considerations. The half-rectifying non-linearity was intended to capture these properties of biological neurons: 1) For some inputs, biological neurons are completely inactive. 2) For some inputs, a biological neuron's output is proportional to its input. 3) Most of the time, biological neurons operate in the regime where they are inactive (e.g., they should have *sparse activations).*

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. This problem can be mitigated by initializing the biases to a small positive number, but it is still possible for a rectified linear unit to learn to de-activate and then never be activated again. Goodfellow *et al.* (2013a) introduced maxout units and showed that maxout units can successfully learn in conditions where rectified linear units become stuck. Maxout units are also piecewise linear, but unlike rectified linear units, each piece of the linear function has its own weight vector, so whichever piece is active can always learn. Due to the greater number of weight vectors, maxout units typically need extra regularization such as dropout, though they can work satisfactorily if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013). Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by $n$ different linear filters can be summarized without losing information by taking the max over each group of $k$ features, then the next layer can get by with $k$ times fewer weights. Because each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist forgetting how to perform tasks that they were trained on in the past. Neural networks trained with stochastic gradient descent are generally

believed to suffer from a phenomenon called *catastrophic forgetting* but maxout units tend to exhibit only mild forgetting (Goodfellow *et al.*, 2014a). Maxout units can also be seen as *learning the activation function* itself rather than just the relationship between units. With large enough $k$, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, maxout with two pieces can learn to implement the rectified linear activation function or the absolute value rectification function.

Another way to avoid the zero-gradient problem of rectifiers is with the leaky (Maas *et al.*, 2013) or or parametric ReLU (He *et al.*, 2015), introduced above. By having a small slope rather than a zero slope when the argument of the rectifier is negative, gradients pass all the time. PReLUs helped to yield a top-5 test error (4.94%) lower than humans (5.1%) for the first time on the ImageNet benchmark (He *et al.*, 2015).

This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation–a particular straightforward kind of such linear activation. This is discussed further in Section 10.7.4.

In addition to helping to propagate information and making optimization easier, piecewise linear units also have some nice properties that can make them easier to regularize. This is discussed further in Section 7.11.

Sigmoidal non-linearities still perform well in some contexts and are a popular choice when a hidden unit must compute a number guaranteed to be in a bounded interval (like in the (0,1) interval), but piecewise linear units are now by far the most popular kind of hidden units.

## 6.9 Historical Notes

Section 1.2 already gave an overview of the history of neural networks and deep learning. Here we focus on historical notes regarding back-propagation and the connectionist ideas that are still at the heart of today's research in deep learning.

The chain rule was invented in the 17th century (Leibniz, 1676; L'Hôpital, 1696) and gradient descent in the 19th centry (Cauchy, 1847b). Efficient applications of the chain rule which exploit the dynamic programming structure described in this chapter are found already in the 1960's and 1970's, mostly for control applications (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and

Ho, 1969; Dreyfus, 1973) but also for sensitivity analysis (Linnainmaa, 1976). Bringing these ideas to the optimization of weights of artificial neural networks with continuous-valued outputs was introduced by Werbos (1981) and rediscovered independently in different ways as well as actually simulated successfully by LeCun (1985); Parker (1985); Rumelhart *et al.* (1986a). The book *Parallel Distributed Processing* (Rumelhart *et al.*, 1986d) presented these findings in a chapter (Rumelhart *et al.*, 1986b) that contributed greatly to the popularization of back-propagation and initiated a very active period of research in multi-layer neural networks. However, the ideas put forward by the authors of that book and in particular by Rumelhart and Hinton go much beyond back-propagation. They include crucial ideas about the possible computational implementation of several central aspects of cognition and learning, which came under the name of "connectionism" because of the importance given the connections between neurons as the locus of learning and memory. In particular, these ideas include the notion of distributed representation, introduced in Chapter 1 and developed a lot more in part III of this book, with Chapter 16, which is at the heart of the generalization ability of neural networks. As discussed with the historical survey in Section 1.2, the boom of AI and machine learning research which followed on the connectionist ideas reached a peak in the early 1990's, as far as neural networks are concerned, while other machine learning techniques become more popular in the late 1990's and remained so for the first decade of this century. Neural networks research in the AI and machine learning community almost vanished then, only to be reborn ten years later (starting in 2006) with a novel focus on the depth of representation and the current wave of research on deep learning. In addition to back-propagation and distributed representations, the connectionists brought the idea of iterative inference (they used different words), viewing neural computation in the brain as a way to look for a configuration of neurons that best satisfy all the relevant pieces of knowledge implicitly captured in the weights of the neural network. This view turns out to be central in the topics covered in part III of this book regarding probabilistic models and inference.

# Chapter 7

# Regularization of Deep or Distributed Models

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization.

Chapter 5 introduced the basic concepts of generalization, underfitting, overfitting, bias, variance and regularization. If you are not already familiar with these notions, please refer to that chapter before continuing with this one.

In this chapter, we describe regularization in more detail, focusing on general strategies for regularizing deep or distributed models. Most deep learning algorithms can be seen as different ways of developing the specifics of these general strategies.

Some sections of this chapter deal with standard concepts in machine learning. If you are already familiar with these concepts, feel free to skip the relevant sections. However, most of this chapter is concerned with the extension of these basic concepts to the particular case of neural networks.

Regularization is any component of the model, training process or prediction procedure which is included to account for limitations of the training data, including its finiteness. There are many regularization strategies. Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values. If chosen carefully, these extra constraints and penalties can lead to improved performance on the test set. Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge. Other times, these constraints and penalties are

designed to express a generic preference for a simpler model class in order to promote generalization. Sometimes penalties and constraints are necessary to make an underdetermined problem determined. Other forms of regularization, known as ensemble methods, combine multiple hypotheses that explain the training data.

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, that is it reduces variance significantly while not overly increasing the bias. When we discussed generalization and overfitting in Chapter 5, we focused on three situations, where the model family being trained either (1) excluded the true data generating process—corresponding to underfitting and inducing bias, or (2) matched to the true data generating process—the "just right" model space, or (3) includes the generating process but also many other possible generating processes—the regime where variance dominates the estimation error (e.g. as measured by the MSE—see Section. 5.5).

Note that, in practice, an overly complex model family does not necessarily include (or even come close to) the target function or the true data generating process. We almost never have access to the true data generating process so we can never know if the model family being estimated includes the generating process or not. But since, in deep learning, we are often trying to work with data such as images, audio sequences and text, we can probably safely assume that our model family does not include the data generating process. We can assume that—to some extent – we are always trying to fit a square peg (the data generating process) into a round hole (our model family) and using the data to do that as best we can.

What this means is that controlling the complexity of the model is not going to be a simple question of finding the model of the right size, of the right number of parameters. Instead, we might find—and indeed in practical deep learning scenarios, we almost always do find – that the best fitting model (in the sense of minimizing generalization error) is one that possesses a large number of parameters that are not entirely free to span their domain.

As we will see there are a great many forms of regularization available to the deep learning practitioner. In fact, developing more effective regularizers has been one of the major research efforts in the field.

Most machine learning tasks can be viewed in terms of learning to represent a function $\hat{f}(\boldsymbol{x})$ parametrized by a vector of parameters $\boldsymbol{\theta}$. The data consists of inputs $\boldsymbol{x}^{(i)}$ and (for some tasks) targets $y^{(i)}$ for $i \in \{1, \ldots, m\}$. In the case of classification, each $y^{(i)}$ is an integer class label in $\{1, \ldots, k\}$. For regression tasks, each $y^{(i)}$ is a real number or a real-valued vector. We then denote the target in bold, as $\boldsymbol{y}^{(i)}$. For density estimation tasks, there are simply no targets. Sometimes

we have variables $\boldsymbol{y}$ that may naturally be considered as targets to be predicted, and we wish to estimate the joint distribution over $\boldsymbol{x}$ and $\boldsymbol{y}$. In this case, we simply define a new vector $\boldsymbol{x}'$ containing $\boldsymbol{x}$ and $\boldsymbol{y}$ concatenated together. We may then perform density estimation over $\boldsymbol{x}'$. We may group these examples into a *design matrix* $\boldsymbol{X}$ and a vector of targets $\boldsymbol{y}$ (when $y^{(i)}$ is a scalar), or a matrix of targets $\boldsymbol{Y}$ (when $\boldsymbol{y}^{(i)}$ is a vector).

In deep learning, we are mainly interested in the case where the function $\hat{f}(\boldsymbol{x})$ has a large number of parameters and as a result possesses a high capacity to fit relatively complicated functions. This means that deep learning algorithms usually require either a large training set or careful regularization (intended either to reduce the effective capacity of the model or to guide the model toward a specific solution using prior information) or both.

## 7.1 Regularization from a Bayesian Perspective

Many common methods of regularization can be interpreted from the Bayesian perspective. As we discussed in Sec. 5.7, Bayesian estimation theory takes a fundamentally different approach to model estimation than the frequentist view by regarding the model parameters themselves as uncertain and therefore treating them as random variables.

There are a number of immediate consequences of assuming a Bayesian world view. The first is that if we are using probability distributions to assess uncertainty in the model parameters then we should be able to express our uncertainty about the model parameters before we see any data[1]. This is the role of the *prior distribution*. The second consequence comes out of the marginalization equation over probability functions: when using the model to make predictions about outcomes, one should ideally average over the probable parameter values, or more precisely sum over all the possible values, weighted by their posterior distribution.

There is a deep connection between the Bayesian perspective on estimation and the process of regularization. This is not surprising since at the root both are concerned with making predictions relative to the true data generating distribution while taking into account the finiteness of the data. What this means is that both are open to combining information sources. That is, both are interested in combining the information that can be extracted from the training data with other, or "prior" sources of information. As we will see, many forms of regularization can be given a Bayesian interpretation.

If we are given a dataset $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$, the question is what it tells us about the unknown parameter $\boldsymbol{\theta}$, and this is characterized by the posterior distribution

[1] We should have a lot of uncertainty about the parameters before we see the data, but some configurations may a priori be impossible, while others could seem more plausible, a priori.

on the model parameter $\boldsymbol{\theta}$. The posterior $p(\boldsymbol{\theta} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ can be obtained by combining the data likelihood $p(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)} \mid \boldsymbol{\theta})$ with the prior belief on the parameter (before seeing the data) encapsulated by $p(\boldsymbol{\theta})$:

$$\log p(\boldsymbol{\theta} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\boldsymbol{x}^{(i)} \mid \boldsymbol{\theta}) + \text{ constant} \tag{7.1}$$

where the constant is $-\log Z$, with $Z$ the normalization constant which does not depend on $\boldsymbol{\theta}$ but does depend on the data. When maximizing over $\boldsymbol{\theta}$, this constant does not matter. In the context of maximum likelihood learning, the introduction of the prior distribution plays the same role as a regularizer in that it can be seen as a term (the first one above) added to the objective function that is added (to the second term, the log-likelihood) in hopes of achieving better generalization, despite of its detrimental effect on the likelihood of the training data.

In the following section, we will detail how the addition of a prior is equivalent to certain regularization strategies. However we must be a bit careful in establishing the relationship between the prior and a regularizer. Regularizers are more general than priors. Priors are distributions and as such are subject to constraints such as they must always be positive and must sum to one over their domain. Regularizers have no such explicit constraints and can depend on the data, not just on the parameters. Another problem in interpreting all regularizers as priors is that the equivalence implies the overly restrictive constraint that all unregularized objective functions be interpretable as log-likelihood functions. Nevertheless, it remains true that many of the most popular forms of regularization can be equated to a Bayesian prior.

## 7.2 Classical Regularization: Parameter Norm Penalty

Regularization has been used for decades prior to the advent of deep learning. Statistical and machine learning models traditionally represented simpler functions. Because the functions themselves had less capacity, the regularization did not need to be as sophisticated. We use the term *classical regularization* to refer to the techniques used in the general machine learning and statistics literature.

Most classical regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function $J$. We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{\theta}) \tag{7.2}$$

where $\alpha$ is a hyperparameter that weights the relative contribution of the norm penalty term, $\Omega$, relative to the standard objective function $J(\boldsymbol{x}; \boldsymbol{\theta})$. The hyper-

parameter $\alpha$ should be a non-negative real number. Setting $\alpha$ to 0 results in no regularization. Larger values of $\alpha$ correspond to more regularization.

When our training algorithm minimizes the regularized objective function $\tilde{J}$ it will decrease both the original objective $J$ on the training data and some measure of the size of the parameters $\boldsymbol{\theta}$ (or some subset of the parameters). Different choices for the parameter norm $\Omega$ can result in different solutions being preferred. In this section, we discuss the effects of the various norms when used as penalties on the model parameters.

Before delving into the regularization behavior of different norms, we note that for neural networks, we typically choose to use a parameter norm penalty $\Omega$ that only penalizes the interaction weights, i.e we leave the offsets unregularized. The offsets typically require less data to fit accurately than the weights. Each weight specifies how two variables interact. Fitting the weigth well requires observing both variables in a variety of conditions. Each offset controls only a single variable. This means that we do not induce too much variance by leaving the offsets unregularized. Also, regularizing the offsets can introduce a significant amount of underfitting.

### 7.2.1 [2] Parameter Regularization

We have already seen one the simplest and most common kinds of classical regularization: the $L^2$ parameter norm penalty commonly known as *weight decay*. This regularization strategy drives the parameters closer to the origin [2] by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ to the objective function. Here, $\boldsymbol{w}$ is the subset of parameters called *weights*. By weights, we mean parameters of a linear transformation. Not all parameters are weights. For example, the biases parameterize the offset of an affine transformation, so they are not considered weights in this context.

In various contexts, $L^2$ regularization is also known as *ridge regression* or *Tikhonov regularization*.

In the context of neural networks, it is sometimes desirable to use a separate weight decay penalty with a different $\alpha$ coefficient for each layer of the network. Because $\alpha$ is a hyperparameter and it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use the same weight decay at all layers just to reduce the search space.

We can gain some insight into the behavior of weight decay regularization

[2] More generally, we could regularize the parameters to be near any specific point in space and, surprisingly, still get a regularization effect, but better results will be obtained for a value closer to the true one, with zero being a default value that makes sense when we do not know if the correct value should be positive or negative. Since it is far more common to regularize the model parameters towards zero, we will focus on this special case in our exposition.

by studying the gradient of the regularized objective function. To simplify the presentation, we assume no offset term, so $\boldsymbol{\theta}$ is just $\boldsymbol{w}$. Such a model has the following gradient of the total objective function:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.3}$$

To take a single gradient step to update the weights, we perform this update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left( \alpha \boldsymbol{w} + \nabla \boldsymbol{w} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \right).$$

Written another way, the update is:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla \boldsymbol{w} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, towards zero, just before performing the usual gradient update. This describes what happens in a single step. But what happens over the entire course of training?

We will further simplify the analysis by considering a quadratic approximation to the objective function in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$. (If the objective function is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect).

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.4}$$

where $\boldsymbol{H}$ is the Hessian matrix of $J$ with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. There is no first order term in this quadratic approximation, because $\boldsymbol{w}^*$ is defined to be a minimum, where the gradient vanishes. Likewise, because $\boldsymbol{w}^*$ is a minimum, we can conclude that $\boldsymbol{H}$ is positive semi-definite.

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.5}$$

If we replace the exact gradient in equation 7.3 with the approximate gradient in equation 7.5, we can write an equation for the location of the minimum of the regularized objective function:

$$\alpha \boldsymbol{w} + \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) = 0 \tag{7.6}$$

$$(\boldsymbol{H} + \alpha \boldsymbol{I})\boldsymbol{w} = \boldsymbol{H}\boldsymbol{w}^* \tag{7.7}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H} \boldsymbol{w}^*. \tag{7.8}$$

From this, we see that the presence of the regularization term moves the optimum from $\boldsymbol{w}^*$ to $\tilde{\boldsymbol{w}}$. As $\alpha$ approaches 0, $\tilde{\boldsymbol{w}}$ approaches $\boldsymbol{w}^*$. But what happens as $\alpha$ grows? Because $\boldsymbol{H}$ is real and symmetric, we can decompose it into a diagonal

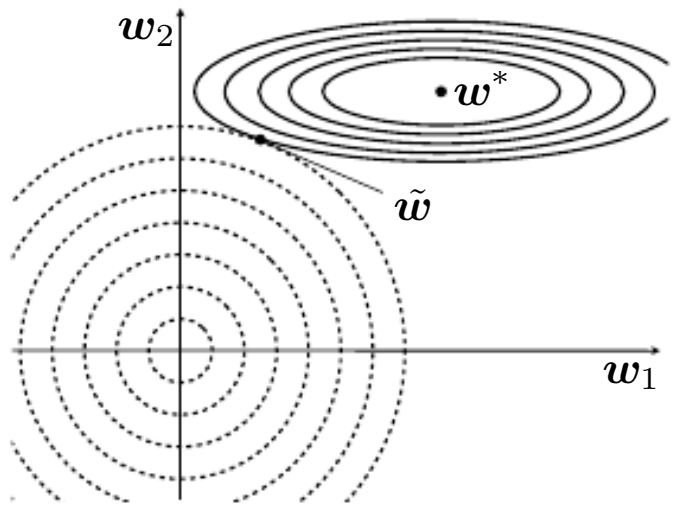


Figure 7.1: An illustration of the effect of L2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the $L$ regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of $J$ is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}$ . Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls $w$ close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}$ . The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of $w$ relatively little.

matrix $\boldsymbol{\Lambda}$ and an orthonormal basis of eigenvectors, $\boldsymbol{Q}$, such that $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Applying the decomposition to equation 7.8, we obtain:

$$\begin{aligned}
\tilde{\boldsymbol{w}} &= (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\
&= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})\boldsymbol{Q}^\top\right]^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\
&= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\Lambda\boldsymbol{Q}^\top\boldsymbol{w}^*, \\
\boldsymbol{Q}^\top\tilde{\boldsymbol{w}} &= (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\Lambda\boldsymbol{Q}^\top\boldsymbol{w}^* . \qquad (7.9)
\end{aligned}$$

If we interpret the $\boldsymbol{Q}^\top\tilde{\boldsymbol{w}}$ as rotating our solution parameters $\tilde{\boldsymbol{w}}$ into the basis defined by the eigenvectors $\boldsymbol{Q}$ of $\boldsymbol{H}$, then we see that the effect of weight decay is to rescale the coefficients of eigenvectors. Specifically the $i$th component is rescaled by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$. (You may wish to review how this kind of scaling works, first explained in Fig. 2.3).

Along the directions where the eigenvalues of $\boldsymbol{H}$ are relatively large, for example, where $\lambda_i \gg \alpha$, the effect of regularization is relatively small. However, components with $\lambda_i \ll \alpha$ will be shrunk to have nearly zero magnitude. This effect is illustrated in Fig. 7.1.

Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact. In directions that do not

contribute to reducing the objective function, a small eigenvalue of the Hessian tell us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training. This effect of suppressing contributions to the parameter vector along these principle directions of the Hessian $\boldsymbol{H}$ is captured in the concept of the *effective number of parameters*, defined to be

$$\gamma = \sum_i \frac{\lambda_i}{\lambda_i + \alpha}. \tag{7.10}$$

As $\alpha$ is increased, the effective number of parameters decreases.

Another way to gain some intuition for the effect of $L^2$ regularization is to study its effect on linear regression. The unregularized objective function for linear regression is the sum of squared errors:

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}).$$

When we add $L^2$ regularization, the objective function changes to

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha \boldsymbol{w}^\top \boldsymbol{w}.$$

This changes the normal equations for the solution from

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

to

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}.$$

We can see $L^2$ regularization causes the learning algorithm to "perceive" the input $\boldsymbol{X}$ as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

### 7.2.2 $^1$ Regularization

While $L^2$ weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use $L^1$ regularization.

Formally, $L^1$ regularization on the model parameter $\boldsymbol{w}$ is defined as:

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |\boldsymbol{w}_i|, \tag{7.11}$$

that is, as the sum of absolute values of the individual parameters.[3] We will now discuss the effect of $L^1$ regularization on the simple linear model, with no offset term, that we studied in our analysis of $L^2$ regularization. In particular, we are interested in delineating the differences between $L^1$ and $L^2$ forms of regularization. Thus, the gradient (actually the sub-gradient) on the regularized objective function $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is as follows:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \beta \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w}) \tag{7.12}$$

where $\text{sign}(\boldsymbol{w})$ is simply the sign of $\boldsymbol{w}$ applied element-wise.

By inspecting Eqn. 7.12, we can see immediately that the effect of $L^1$ regularization is quite different from that of $L^2$ regularization. Specifically, we can see that the regularization contribution to the gradient no longer scales linearly with $\boldsymbol{w}$, instead it is a constant factor with a sign equal to $\text{sign}(\boldsymbol{w})$. One consequence of this form of the gradient is that we will not necessarily see clean solutions to quadratic forms of $\nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w})$ as we did for $L^2$ regularization. Instead, the solutions are going to be much more aligned to the basis space in which the problem is embedded.

To see this, and for the sake of comparison with $L^2$ regularization, we will again study a simplified setting of a quadratic approximation to the objective function in the neighborhood of the empirical optimum $\boldsymbol{w}^*$. (Once again, if the per-example loss is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect). The gradient of this approximation is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.13}$$

where, again, $\boldsymbol{H}$ is the Hessian matrix of $J$ with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. We will also make the further simplifying assumption that the Hessian is diagonal, $\boldsymbol{H} = \text{diag}([\gamma_1, \ldots, \gamma_N])$, where each $\gamma_i > 0$. With this rather restrictive assumption, the solution of the minimum of the $L^1$ regularized objective function decomposes into a system of equations of the form:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2}\gamma_i (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \beta |\boldsymbol{w}_i|.$$

It admits an optimal solution (for each dimension $i$), with the following form:

$$\boldsymbol{w}_i = \text{sign}(\boldsymbol{w}_i^*) \max(|\boldsymbol{w}_i^*| - \frac{\beta}{\gamma_i}, 0).$$

[3] As with $L^2$ regularization, we could regularize the parameters towards a value that is not zero, but instead towards some parameter value $\boldsymbol{w}^{(o)}$. In that case the $L^1$ regularization would introduce the term $\Omega(\boldsymbol{\theta}) = \boldsymbol{w} \quad \boldsymbol{w}^{(o)}{}_1 = \beta \quad {}_i \boldsymbol{w}_i \quad \boldsymbol{w}_i^{(o)}$.

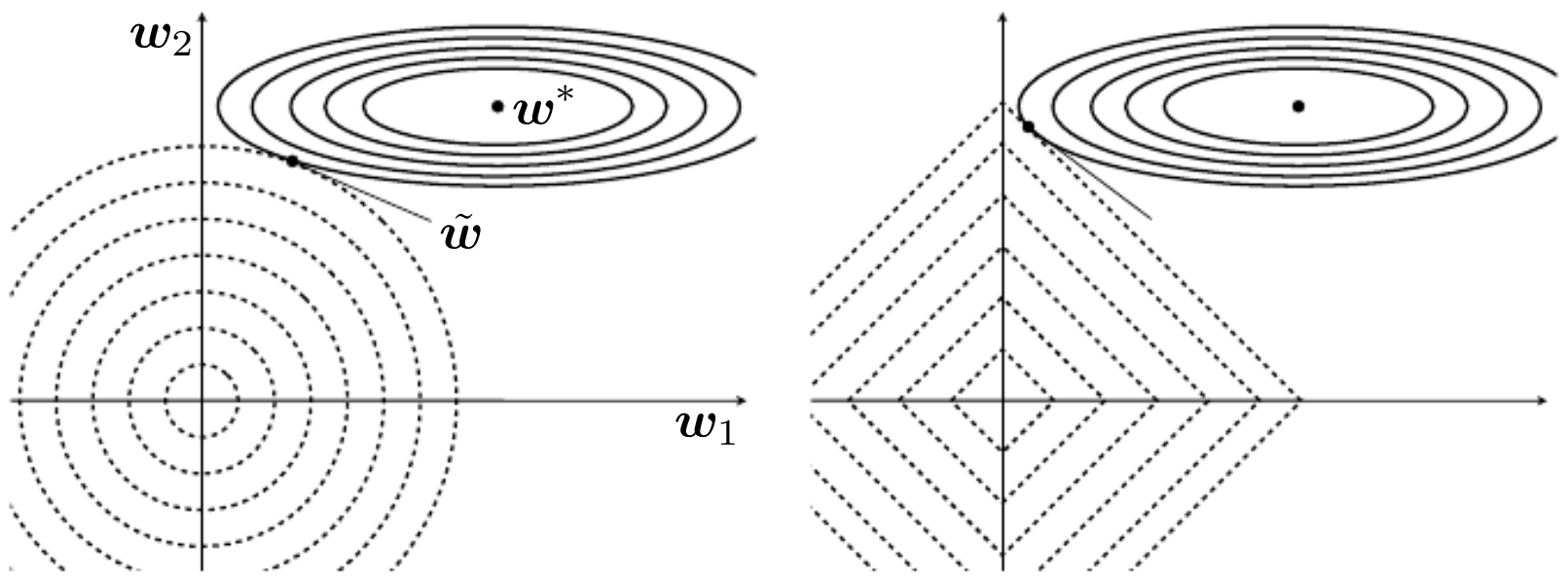


Figure 7.2: An illustration of the effect of $L$ regularization (right) on the value of the optimal $\boldsymbol{W}$, in comparison to the effect of $L$ regularization (left). Please refer to Fig. 7.1 for an explanation of the drawing. Note that with L1 regularization, the solution is more likely to end up close to or at an axis, where some of the parameters are set to 0. Compared to L2 regularization, we see that the equal-cost contour lines of the L1 regularizer make it easier for the solution to slide to such axis-aligned solutions.

Let's consider the situation where $\boldsymbol{w}_i^* > 0$ for all $i$, there are two possible outcomes. **Case 1:** $\boldsymbol{w}_i^* \leq \frac{\beta}{\gamma_i}$, here the optimal value of $\boldsymbol{w}_i$ under the regularized objective is simply $\boldsymbol{w}_i = 0$, this occurs because the contribution of $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ to the regularized objective $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is overwhelmed—in direction $i$, by the $L^1$ regularization which pushes the value of $\boldsymbol{w}_i$ to zero. **Case 2:** $\boldsymbol{w}_i^* > \frac{\beta}{\gamma_i}$, here the regularization does not move the optimal value of $\boldsymbol{w}$ to zero but instead it just shifts it in that direction by a distance equal to $\frac{\beta}{\gamma_i}$. This is illustrated in Fig. 7.2. A similar argument can be made when $\boldsymbol{w}_i^* < 0$, but with the $L_1$ penalty making $\boldsymbol{w}_i$ less negative by $\frac{\beta}{\gamma_i}$, or 0.

In comparison to $L^2$ regularization, $L^1$ regularization results in a solution that is more *sparse*. Sparsity in this context implies that there are some parameters have an optimal value of zero, due to the $L^1$ regularization term in the objective function. As we discussed, for each element $i$ of the parameter vector, this happens when $|\boldsymbol{w}_i^*| \leq \frac{\beta}{\gamma_i}$. Comparing this to the situation for $L^2$ regularization, where (under the same assumptions of a diagonal Hessian $\boldsymbol{H}$) we get $\boldsymbol{w}_{L^2} = \frac{\gamma_i}{\gamma_i + \alpha}\boldsymbol{w}^*$, which is nonzero as long as $\boldsymbol{w}^*$ is nonzero.

In Fig. 7.2, we see that even when the optimal value of $\boldsymbol{w}$ is nonzero, $L^1$ regularization acts to punish small values of parameters just as harshly as larger values, leading to optimal solutions with more parameters having value zero and more larger valued parameters.

The sparsity property induced by $L^1$ regularization has been used extensively as a feature selection mechanism. In particular, the well known LASSO (Tibshirani, 1995) (least absolute shrinkage and selection operator) model integrates an $L^1$ penalty with a linear model and a least squares cost function.

### 7.2.3 Bayesian Interpretation of the Parameter Norm Penalty

Parameter norm penalties are often amenable to being interpreted as a Bayesian prior. Recall that parameter norm penalties are effected by adding a term $\Omega(\boldsymbol{w})$ to the unregularized objective function $J$.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{w}) \tag{7.14}$$

where $\alpha$ is a hyperparameter that weights the relative contribution of the norm penalty term.

We can view the minimization of the regularized objective function above as equivalent to finding the maximum *a posteriori* (MAP) estimate of the parameters: $p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w})$, where the unregularized $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is taken as the log-likelihood and the regularization term $\alpha\Omega(\boldsymbol{w})$ plays the role of the parameter prior distribution. Difference choices of regularizers correspond to different priors.

In the case of $L^2$ regularization, minimizing with $\alpha\Omega(\boldsymbol{w}) = \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2$ is functionally equivalent to maximizing the log of the posterior distribution (or minimizing the negative log posterior) where the prior is given by a Gaussian distribution.

$$\log p(\boldsymbol{w}; \boldsymbol{\mu}, \Sigma) = -\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu})^\top \Sigma^{-1}(\boldsymbol{w} - \boldsymbol{\mu}) - \frac{1}{2}\log|\Sigma| - \frac{d}{2}\log(2\pi)$$

where $d$ is the dimension of $\boldsymbol{w}$. Ignoring terms that are not a function of $\boldsymbol{w}$ (and therefore do not affect the MAP value), we can see that the by choosing $\boldsymbol{\mu} = \boldsymbol{0}$ and $\Sigma^{-1} = \alpha\boldsymbol{I}$, we recover the functional form of $L^2$ regularization: $p(\boldsymbol{w}; \boldsymbol{\mu}, \Sigma) \propto e^{-\frac{\alpha}{2}\|\boldsymbol{w}\|_2^2}$. Thus $L^2$ regularization can be interpreted as assuming independent Gaussian prior distributions over all the model parameters, each with precision (which is the inverse of variance) $\alpha$.

For $L^1$ regularization, minimizing with $\alpha\Omega(\boldsymbol{w}) = \alpha\sum_i \|\boldsymbol{w}_i\|$ is equivalent to maximizing the log of the posterior distribution with an isotropic Laplace distribution over $\boldsymbol{w}$.

$$\log p(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\eta}) = \sum_i \log \mathrm{Laplace}(w_i; \boldsymbol{\mu}_i, \boldsymbol{\eta}_i) = \sum_i -\frac{|\boldsymbol{w}_i - \boldsymbol{\mu}_i|}{\eta_i} - \log(2\boldsymbol{\eta}_i)$$

Once again we can ignore the second term here because it does not depend on the elements of $\boldsymbol{w}$, so $L^1$ regularization is equivalent to optimizing a MAP objective with a log prior given by $\sum_i \log \mathrm{Laplace}(w_i; 0, \lambda^{-1})$.

## 7.3 Classical Regularization as Constrained Optimization

Classical regularization adds a penalty term to the training objective:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}).$$

Recall from Sec. 4.4 that we can minimize a function subject to constraints by constructing a generalized Lagrange function (see 4.4), consisting of the original objective function plus a set of penalties. Each penalty is a product between a coefficient, called a Karush–Kuhn–Tucker (KKT) multiplier [4], and a function representing whether the constraint is satisfied. If we wanted to constrain $\Omega(\boldsymbol{\theta})$ to be less than some constant $k$, we could construct a generalized Lagrange function

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k).$$

The solution to the constrained problem is given by

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha).$$

Solving this problem requires modifying both $\boldsymbol{\theta}$ and $\alpha$. Specifically, $\alpha$ must increase whenever $||\boldsymbol{\theta}||_p > k$ and decrease whenever $||\boldsymbol{\theta}||_p < k$. However, after we have solved the problem, we can fix $\alpha^*$ and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^*\Omega(\boldsymbol{\theta}). \tag{7.15}$$

This is exactly the same as the regularized training problem of minimizing $\tilde{J}$. Note that the value of $\alpha^*$ does not directly tell us the value of $k$. In principle, one can solve for $k$; but the relationship between $k$ and $\alpha^*$ depends on the form of $J$. We can thus think of classical regularization as imposing a constraint on the weights, but with an unknown size of the constraint region. Larger $\alpha$ will result in a smaller constraint region. Smaller $\alpha$ will result in a larger constraint region.

Sometimes we may wish to use explicit constraints rather than penalties. As described in Sec. 4.4, we can modify algorithms such as stochastic gradient descent to take a step downhill on $J(\boldsymbol{\theta})$ and then project $\boldsymbol{\theta}$ back to the nearest point that satisfies $\Omega(\boldsymbol{\theta}) < k$. This can be useful if we have an idea of what value of $k$ is appropriate and do not want to spend time searching for the value of $\alpha$ that corresponds to this $k$.

Another reason to use explicit constraints and reprojection rather than enforcing constraints with penalties is that penalties can cause non-convex optimization

[4] KKT multipliers generalize Lagrange multipliers to allow for inequality constraints.

procedures to get stuck in local minima corresponding to small $\boldsymbol{\theta}$. When training neural networks, this usually manifests as neural networks that train with several "dead units". These are units that do not contribute much to the behavior of the function learned by the network because the weights going into or out of them are all very small. When training with a penalty on the norm of the weights, these configurations can be locally optimal, even if it is possible to significantly reduce $J$ by making the weights larger. (This concern about local minima obviously does not apply when $\tilde{J}$ is convex)

Finally, explicit constraints with reprojection can be useful because they impose some stability on the optimization procedure. When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients which then induce a large update to the weights. If these updates consistently increase the size of the weights, then $\boldsymbol{\theta}$ rapidly moves away from the origin until numerical overflow occurs. Explicit constraints with reprojection allow us to terminate this feedback loop after the weights have reached a certain magnitude. Hinton *et al.* (2012c) recommend using constraints combined with a high learning rate to allow rapid exploration of parameter space while maintaining some stability.

## 7.4 Regularization and Under-Constrained Problems

In some cases, regularization is necessary for machine learning problems to be properly defined. Many linear models in machine learning, including linear regression and PCA, depend on inverting the matrix $\boldsymbol{X}^\top \boldsymbol{X}$. This is not possible whenever $\boldsymbol{X}^\top \boldsymbol{X}$ is singular. This matrix can be singular whenever the data truly has no variance in some direction, or when there are fewer examples (rows of $\boldsymbol{X}$) than input features (columns of $\boldsymbol{X}$). In this case, many forms of regularization correspond to inverting $\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I}$ instead. This regularized matrix is guaranteed to be invertible.

These linear problems have closed form solutions when the relevant matrix is invertible. It is also possible for a problem with no closed form solution to be underdetermined. An example is logistic regression applied to a problem where the classes are linearly separable. If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, then $2\boldsymbol{w}$ will also achieve perfect classification and higher likelihood. An iterative optimization procedure like stochastic gradient descent will continually increase the magnitude of $\boldsymbol{w}$ and, in theory, will never halt. In practice, a numerical implementation of gradient descent will eventually reach sufficiently large weights to cause numerical overflow, at which point its behavior will depend on how the programmer has decided to handle values that are not real numbers.

Most forms of regularization are able to guarantee the convergence of iterative

methods applied to underdetermined problems. For example, weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient. Likewise, early stopping based on the validation set classification rate will cause the training algorithm to terminate soon after the validation set classification accuracy has stopped increasing. Even if the problem is linearly separable and there is no overfitting, the validation set classification accuracy will eventually saturate to 100%, resulting in termination of the early stopping procedure.

The idea of using regularization to solve underdetermined problems extends beyond machine learning. The same idea is useful for several basic linear algebra problems.

As we saw in Chapter 2.9, we can solve underdetermined linear equations using the Moore-Penrose pseudoinverse.

One definition of the pseudoinverse $\boldsymbol{X}^+$ of a matrix $\boldsymbol{X}$ is to perform linear regression with an infinitesimal amount of $L^2$ regularization:

$$\boldsymbol{X}^+ = \lim_{\alpha \searrow 0} (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top .$$

When a true inverse for $\boldsymbol{X}$ exists, then $\boldsymbol{w} = \boldsymbol{X}^+ \boldsymbol{y}$ returns the weights that exactly solve the regression problem. When $\boldsymbol{X}$ is not invertible because no exact solution exists, this returns the $\boldsymbol{w}$ corresponding to the least possible mean squared error. When $\boldsymbol{X}$ is not invertible because many solutions exactly solve the regression problem, this returns $\boldsymbol{w}$ with the minimum possible $L^2$ norm.

Recall that the Moore-Penrose pseudoinverse can be computed easily using the singular value decomposition. Because the SVD is robust to underdetermined problems resulting from too few observations or too little underlying variance, it is useful for implementing stable variants of many closed-form linear machine learning algorithms. The stability of these algorithms can be viewed as a result of applying the minimum amount of regularization necessary to make the problem become determined.

## 7.5 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high dimensional input $\boldsymbol{x}$ and summarize it with a single category identity

$y$. This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new $(\boldsymbol{x}, y)$ pairs easily just by transforming the $\boldsymbol{x}$ inputs in our training set.

This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimation problem.

Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include an enormous variety of factors of variation, many of which can be easily simulated. Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using convolution and pooling. Many other operations such as rotating the image or scaling the image have also proven quite effective. One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between 'b' and 'd' and the difference between '6' and '9', so horizontal flips and $180^\circ$ rotations are not appropriate ways of augmenting datasets for these tasks. There are also transformations that we would like our classifiers to be invariant to, but which are not easy to perform. For example, out-of-plane rotation can not be implemented as a simple geometric operation on the input pixels.

Injecting noise in a neural network can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise, however (Tang and Eliasmith, 2010). One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms such as the denoising auto-encoder (Vincent *et al.*, 2008). Noise injection also works when the noise is applied to the hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction. Poole *et al.* (2014) recently showed that this approach can be highly effective provided that the magnitude of the noise is carefully tuned. Dropout, a powerful regularization strategy that will be described in Sec. 7.11, can be seen as a process of constructing new inputs by *multiplying* by noise.

When comparing machine learning benchmark results, it is important to take the effect of dataset augmentation into account. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique. To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, it

is necessary to make sure that both algorithms were evaluated using the same hand-designed dataset augmentation schemes. Suppose that algorithm A performs poorly with no dataset augmentation and algorithm B performs well when combined with numerous synthetic transformations of the input. In such a case it is likely the synthetic transformations caused the improved performance, rather than the use of machine learning algorithm B. Sometimes deciding whether an experiment has been properly controlled requires subjective judgment. For example, machine learning algorithms that inject noise into the input are performing a form of dataset augmentation. Usually, operations that are generally applicable (such as adding Gaussian noise to the input) are considered part of the machine learning algorithm, while operations that are specific to one application domain (such as randomly cropping an image) are considered to be separate pre-processing steps.

## 7.6 Classical Regularization as Noise Robustness

In the machine learning literature, there have been two ways that noise has been used as part of a regularization strategy. The first and most popular way is by adding noise to the input. While this can be interpreted simply as form of dataset augmentation (as described above in Sec. 7.5), we can also interpret it as being equivalent to more traditional forms of regularization.

The second way that noise has been used in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks (Jim *et al.*, 1996; Graves, 2011a). This can be interpreted as a stochastic implementation of a Bayesian inference over the weights. The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty (Graves, 2011a).

In this section, we review these two strategies and provide some insight into how noise can act to regularize the model.

### 7.6.1 Injecting Noise at the Input

Some classical regularization techniques can be derived in terms of training on noisy inputs [5]. Let us study a regression setting, where we are interested in learning a model $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a scalar. The cost function we will use is the least-squares error between the model prediction $\hat{y}(\boldsymbol{x})$ and the true value $y$.

$$J = \mathbb{E}_{p(x,y)} \ (\hat{y}(\boldsymbol{x}) - \boldsymbol{y})^2 \ , \tag{7.16}$$

[5] The analysis in this section is mainly based on that in Bishop (1995a,b)

where we are given a dataset of $m$ input / output pairs $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$ and the training objective is to minimize the objective function, which is the empirical average of the squared error on the training data.

With each input presentation to the model, we also include a random perturbation $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \nu \boldsymbol{I})$, so that the error function becomes

$$\begin{aligned}
\tilde{J}_{\boldsymbol{x}} &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ (\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) - y)^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \hat{y}^2(\boldsymbol{x}+\boldsymbol{\epsilon}) - 2y\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) + y^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \hat{y}^2(\boldsymbol{x}+\boldsymbol{\epsilon}) \right] - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ y\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ y^2 \right]
\end{aligned} \tag{7.17}$$

Assuming that the noise is small, we can model its effect using the Taylor series expansion of $\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon})$ around $\hat{y}(\boldsymbol{x})$.

$$\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) = \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} + O(\boldsymbol{\epsilon}^3) \tag{7.18}$$

Substituting this approximation for $\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon})$ into the objective function (Eq. 7.17) and using the fact that $\mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{\epsilon}] = 0$ and that $\mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top] = \nu \boldsymbol{I}$ to simplify[6], we get:

$$\begin{aligned}
\tilde{J}_{\boldsymbol{x}} \approx\ & \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \left( \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right)^2 \right] \\
& - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ y\hat{y}(\boldsymbol{x}) + y\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2} y\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ y^2 \right] \\
=\ & \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ (\hat{y}(\boldsymbol{x}) - y)^2 \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} + \left( \boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) \right)^2 + O(\boldsymbol{\epsilon}^3) \right] \\
& - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \frac{1}{2} y\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right] \\
=\ & J + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[ (\hat{y}(\boldsymbol{x}) - y) \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x}) \right] + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[ \|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2 \right]
\end{aligned} \tag{7.19}$$

If we minimize this objective function, by taking the functional gradient of $\hat{y}(\boldsymbol{x})$ and setting the result to zero, we can see that

$$\hat{y}(\boldsymbol{x}) = \mathbb{E}_{p(y|\boldsymbol{x})}[y] + O(\nu).$$

This implies that the expectation in the second last term in Eq. 7.19,

$$\mathbb{E}_{p(\boldsymbol{x},y)} \left[ (\hat{y}(\boldsymbol{x}) - y) \nabla_{\boldsymbol{x}}^2 \hat{y}(\boldsymbol{x}) \right],$$

[6] In this derivation we have used two properties of the trace operator: (1) that a scalar is equal to its trace; (2) that, for a square matrix $AB$, $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$. These are discussed in Sec. 2.10.

reduces to $O(\nu)$ because the expectation of the difference $(\hat{y}(\boldsymbol{x}) - y)$ reduces to $O(\nu)$.

This leaves us with the objective function of the form

$$\tilde{J}_{\boldsymbol{x}} = \mathbb{E}_{p(\boldsymbol{x},y)} \left[(\hat{y}(\boldsymbol{x}) - y)^2\right] + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[\|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2\right] + O(\nu^2).$$

For small $\nu$, the minimization of $J$ with added noise on the input (with covariance $\nu \boldsymbol{I}$) is equivalent to minimization of $J$ with an additional *regularization* term given by $\nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[\|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2\right]$.

This regularization term has the effect of penalizing large gradients of the function $\hat{y}(\boldsymbol{x})$. That is, it has the effect of reducing the *sensitivity* of the output of the network with respect to small variations in its input $\boldsymbol{x}$. We can interpret this as attempting to build in some local robustness into the model and thereby promote generalization. We note also that for linear networks, this regularization term reduces to simple weight decay (as discussed in Sec. 7.2.1).

### 7.6.2 Injecting Noise at the Weights

Rather than injecting noise as part of the input, one could also add noise directly to the model parameters. As we shall see, this can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization. Adding noise to the weights has been shown to be an effective regularization strategy in the context of recurrent neural networks[7] (Jim *et al.*, 1996; Graves, 2011b). In the following, we will present an analysis of the effect of weight noise on a standard feedforward neural network (as introduced in Chapter 6).

As we did in the last section, we again study the regression setting, where we wish to train a function $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a scalar using the least-squares cost function between the model predictions $\hat{y}(\boldsymbol{x})$ and the true values $y$:

$$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\boldsymbol{x}) - \boldsymbol{y})^2\right]. \tag{7.20}$$

We again assume we are given a dataset of $m$ input / output pairs $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

We now assume that with each input presentation we also include a random perturbation $\boldsymbol{\epsilon_W} \sim (\boldsymbol{0}, \eta \boldsymbol{I})$ of the network weights. Let us imagine that we have a standard $L$-layer MLP, we denote the perturbed model as $\hat{y}_{\boldsymbol{\epsilon}}(\boldsymbol{x})$. Despite the injection of noise, we are still interested in minimizing the squared error of the output of the network. The objective function thus becomes:

$$\begin{aligned} \tilde{J}_{\boldsymbol{W}} &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[(\hat{y}_{\boldsymbol{\epsilon_W}}(\boldsymbol{x}) - y)^2\right] \\ &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \hat{y}_{\boldsymbol{\epsilon}}^2(\boldsymbol{x}) - 2y\hat{y}_{\boldsymbol{\epsilon}}(\boldsymbol{x}) + y^2 \end{aligned} \tag{7.21}$$

[7] Recurrent neural networks will be discussed in detail in Chapter 10

Assuming small noise, we can consider the Taylor series expansion of $\hat{y}_{\boldsymbol{\epsilon}}(\boldsymbol{x})$ around the unperturbed function $\hat{y}(\boldsymbol{x})$.

$$\hat{y}_{\boldsymbol{\epsilon}}(\boldsymbol{x}) = \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) + \frac{1}{2} \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon}_{\boldsymbol{W}} + O(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{3}) \tag{7.22}$$

From here, we follow the same basic strategy that was laid-out in the previous section in analyzing the effect of adding noise to the input. That is, we substitute the Taylor series expansion of $\hat{y}_{\boldsymbol{\epsilon}}(\boldsymbol{x})$ into the objective function in Eq. 7.21.

$$\begin{aligned}
\tilde{J}_{\boldsymbol{W}} &\approx \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \left( \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) + \frac{1}{2} \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon}_{\boldsymbol{W}} \right)^2 \right] \\
&\quad \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ -2y \left( \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) + \frac{1}{2} \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon}_{\boldsymbol{W}} \right) \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ y^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ (\hat{y}(\boldsymbol{x}) - y)^2 \right] - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \frac{1}{2} y \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon} \right] \\
&\quad + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[ \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \boldsymbol{\epsilon}_{\boldsymbol{W}} + \left( \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top} \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) \right)^2 + O(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{3}) \right]. \qquad (7.23) \\
&\qquad (7.24)
\end{aligned}$$

Where we have used the fact that $\mathbb{E}_{\boldsymbol{\epsilon}}(\boldsymbol{\epsilon}_{\boldsymbol{W}}) = \boldsymbol{0}$ to drop terms that are linear in $\boldsymbol{\epsilon}_{\boldsymbol{W}}$. Incorporating the assumption that $\mathbb{E}_{\boldsymbol{\epsilon}}(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{2}) = \eta \boldsymbol{I}$, we have:

$$\tilde{J}_{\boldsymbol{W}} \approx J + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[ (\hat{y}(\boldsymbol{x}) - y) \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \right] + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[ \| \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) \|^2 \right] \tag{7.25}$$

Again, if we minimize this objective function, we find that the optimal value of $\hat{y}(\boldsymbol{x})$ is:

$$\hat{y}(\boldsymbol{x}) = \mathbb{E}_{p(y|\boldsymbol{x})} [y] + O(\eta),$$

implying that the expectation in the middle term in Eq. 7.25, $\mathbb{E}_{p(\boldsymbol{x},y)} \left[ (\hat{y}(\boldsymbol{x}) - y) \nabla_{\boldsymbol{W}}^{2} \hat{y}(\boldsymbol{x}) \right]$, reduces to $O(\eta)$ because the expectation of the difference $(\hat{y}(\boldsymbol{x}) - y)$ is reduces to $O(\eta)$.

This leaves us with the objective function of the form

$$\tilde{J}_{\boldsymbol{W}} = \mathbb{E}_{p(\boldsymbol{x},y)} \left[ (\hat{y}(\boldsymbol{x}) - y)^2 \right] + \eta \mathbb{E}_{p(\boldsymbol{x},y)} \left[ \| \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) \|^2 \right] + O(\eta^2).$$

For small $\eta$, the minimization of $J$ with added weight noise (with covariance $\eta \boldsymbol{I}$) is equivalent to minimization of $J$ with an additional *regularization* term: $\eta \mathbb{E}_{p(\boldsymbol{x},y)} \left[ \| \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) \|^2 \right]$. This form of regularization encourages the parameters to go to regions of parameter space where weights have a relatively small influence on the output. In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights. Regularization strategies

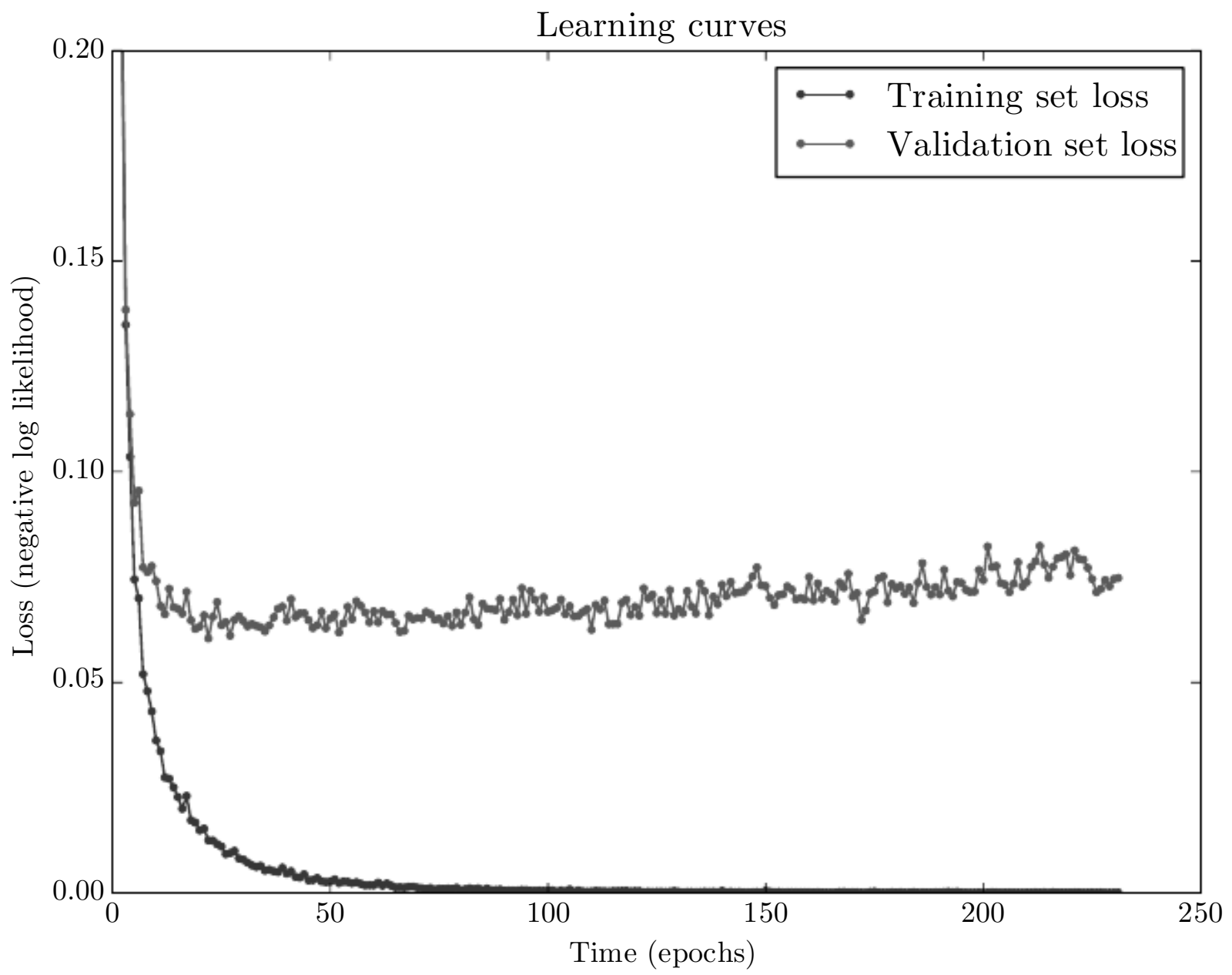


Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or *epochs*). In this example, we train a maxout network on MNIST, regularized with dropout. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

with this kind of behavior have been studied before (Hochreiter and Schmidhuber, 1995). In the simplified case of linear regression (where, for instance, $\hat{y}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$), this regularization term collapses into $\eta \mathbb{E}_{p(\boldsymbol{x})} \left[ \|\boldsymbol{x}\|^2 \right]$, which is not a function of parameters and therefore does not contribute to the gradient of $\tilde{J}_{\boldsymbol{W}}$ w.r.t the model parameters.

## 7.7 Early Stopping as a Form of Regularization

When training large models with large enough capacity, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See Fig. 7.3 for an example of this behavior. This behavior occurs very reliably.

This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Instead of running our optimization algorithm until we reach a (local) minimum of validation error, we run it until the error on the validation set has not improved for some amount of time. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. This procedure is specified more formally in Alg. 7.1.

**Algorithm 7.1** The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let $n$ be the number of steps between evaluations.
Let $p$ be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
**while** $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for $n$ steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  **if** $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  **else**
    $j \leftarrow j + 1$
  **end if**
**end while**
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is $i^*$

This strategy is known as *early stopping*. It is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity.

One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter. We can see in Fig. 7.3 that this hyperparameter has a U-shaped validation set performance curve, just like most other model capacity control parameters, albeit with the addition of irregular oscillations. In this case, we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set precisely. Most of the time, setting hyperparameters requires an expensive guess and check process, where we must set a hyperparameter at the start of training, then run training for several steps to see its effect. The "training time" hyperparameter is unique in that by definition a single run of training tries out many values of the hyperparameter. The only significant cost to choosing this hyperparameter automatically via early stopping is running the validation set evaluation periodically during training. One typically chooses a validation set size and a period for computing validation error such that this monitoring only adds a fraction of the training time to the total computational cost, for example by making the number of training examples seen between validation error measurements a multiple of the validation set size. Even better, with access to another processor, one uses it to perform the monitoring work, allowing one to use larger validation sets without slowing down the training process.

An additional cost to early stopping is the need to maintain a copy of the best parameters. This cost is generally negligible, because it is acceptable to store these parameters in a slower and larger form of memory (for example, training in GPU memory, but storing the optimal parameters in host memory or on a disk drive). Since the best parameters are written to infrequently and never read during training, these occasional slow writes are have little effect on the total training time.

Early stopping is a very unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics. This is in contrast to weight decay, where one must be careful not to use too much weight decay and trap the network in a bad local minimum corresponding to a solution with pathologically small weights.

Early stopping may be used either alone or in conjunction with other regularization strategies. Even when using regularization strategies that modify the objective function to encourage better generalization, it is rare for the best generalization to occur at a local minimum of the training objective.

Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed. In the second, extra

training step, all of the training data is included. There are two basic strategies one can use for this second training procedure.

One strategy is to initialize the model again and retrain on all of the data. In this second training pass, we train for the same number of steps as the early stopping procedure determined was optimal in the first pass. There are some subtleties associated with this procedure. For example, there is not a good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset. On the second round of training, each pass through the dataset will require more parameter updates because the training set is bigger. Usually, if overfitting is a serious concern, you will want to retrain for the same number of epochs, rather than the same number of parameter updates. If the primary difficulty is optimization rather than generalization, then retraining for the same number of parameter updates makes more sense (but it is also less likely that you need to use a regularization method like early stopping in the first place). This algorithm is described more formally in Alg. 7.2.

**Algorithm 7.2** A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (Alg. 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns $i^*$, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for $i^*$ steps.

Another strategy for using all of the data is to keep the parameters obtained from the first round of training and then *continue* training but now using all of the data. At this stage, we now no longer have a guide for when to stop in terms of a number of steps. Instead, we can monitor the average loss function on the validation set, and continue training until it falls below the value of the training set objective at which the early stopping procedure halted. This strategy avoids the high cost of retraining the model from scratch, but is not as well-behaved. For example, there is not any guarantee that the objective on the validation set will ever reach the target value, so this strategy is not even guaranteed to terminate. This procedure is presented more formally in Alg. 7.3.

**Early stopping and the use of surrogate loss functions:** A useful property of early stopping is that it can help to mitigate the problems caused by a mismatch

**Algorithm 7.3** Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (Alg. 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
**while** $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for $n$ steps.
**end while**

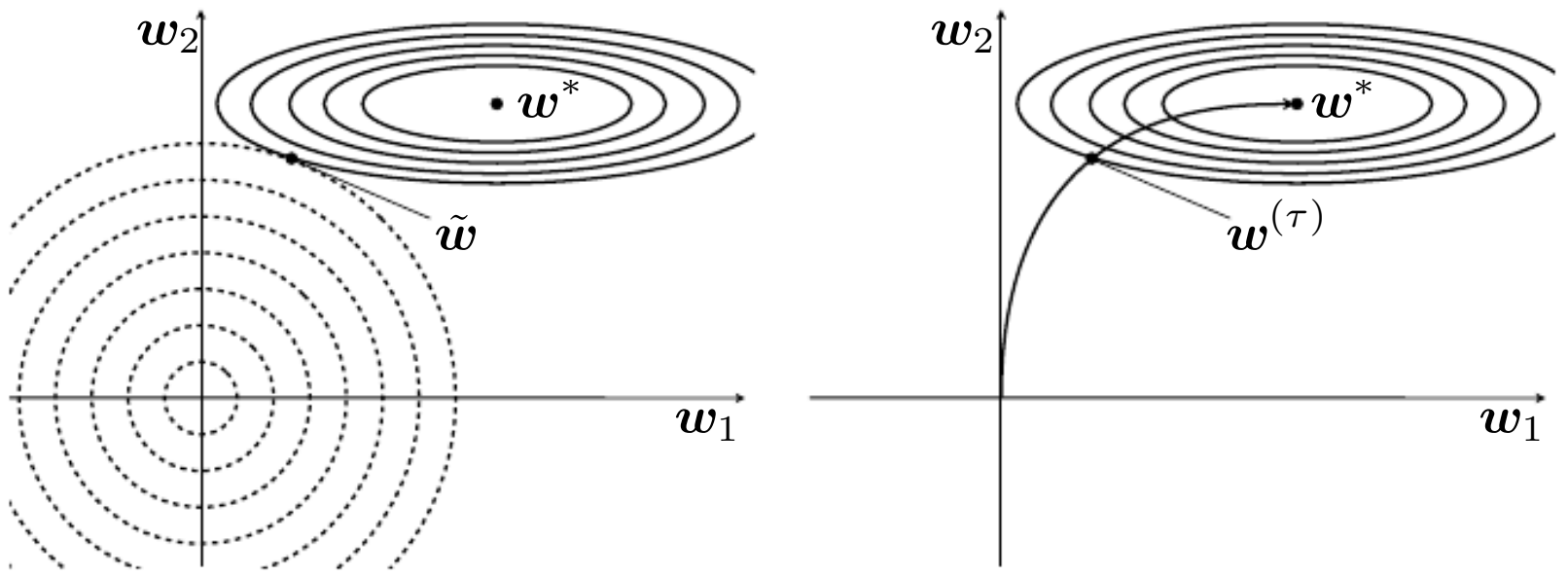


Figure 7.4: An illustration of the effect of early stopping (Right) as a form of regularization on the value of the optimal $\boldsymbol{w}$, as compared to L2 regularization (Left) discussed in Sec. 7.2.1.

between the surrogate loss function whose gradient we follow downhill and the underlying performance measure that we actually care about. For example, 0-1 classification loss has a derivative that is zero or undefined everywhere, so it is not appropriate for gradient-based optimization. We therefore train with a surrogate such as the log-likelihood of the correct class label. However, 0-1 loss is inexpensive to compute, so it can easily be used as an early stopping criterion. Even though the training 0-1 loss may have reached 0, the training log-likelihood can improve, yielding further improvements in validation set 0-1 loss.

Early stopping is also useful because it reduces the computational cost of the training procedure. It is a form of regularization that does not require adding additional terms to the surrogate loss function, so we get the benefit of regularization without the cost of any additional gradient computations. It also means

that we do not spend time approaching the exact local minimum of the surrogate loss.

**How early stopping acts as a regularizer:** So far we have stated that early stopping *is* a regularization strategy, but we have supported this claim only by showing learning curves where the validation set error has a U-shaped curve. What is the actual mechanism by which early stopping regularizes the model?[8]

Early stopping has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter value $\boldsymbol{\theta}_o$. More specifically, imagine taking $\tau$ optimization steps (corresponding to $\tau$ training iterations) and taking $\eta$ as the learning rate. We can view the product $\eta\tau$ as a measure of effective capacity. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from $\boldsymbol{\theta}_o$. In this sense, $\eta\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay.

Indeed, we can show how—in the case of a simple linear model with a quadratic error function and simple gradient descent—early stopping is equivalent to L2 regularization.

In order to compare with classical $L^2$ regularization, we examine a simple setting where the only parameters are linear weights ($\boldsymbol{\theta} = \boldsymbol{w}$). We can model the cost function $J$ with a quadratic approximation in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \tag{7.26}$$

where, as before, $\boldsymbol{H}$ is the Hessian matrix of $J$ with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. Given the assumption that $\boldsymbol{w}^*$ is a minimum of $J(\boldsymbol{w})$, we know that $\boldsymbol{H}$ is positive semi-definite. Under a local Taylor series approximation, the gradient is given by:

$$\nabla_{\boldsymbol{w}}\hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.27}$$

We are going to study the trajectory followed by the parameter vector during training. For simplicity, let us set the initial parameter vector to the origin[9], that

[8] Material for this section was taken from Bishop (1995a); Sjöberg and Ljung (1995); for further details regarding the interpretation of early-stopping as a regularizer, please consult these works.

[9] For neural networks, to obtain symmetry breaking between hidden units, we cannot initialize all the parameters at 0, as discussed in Section 8.7.2. However, the argument holds for any other initial value $\boldsymbol{w}^{(0)}$.

is $\boldsymbol{w}^{(0)} = \mathbf{0}$. Let us suppose that we update the parameters via gradient descent:

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}^{(\tau-1)}) \tag{7.28}$$

$$= \boldsymbol{w}^{(\tau-1)} - \eta \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.29}$$

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \eta \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.30}$$

Let us now rewrite this expression in the space of the eigenvectors of $\boldsymbol{H}$, exploiting the eigendecomposition of $\boldsymbol{H}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, where $\boldsymbol{\Lambda}$ is a diagonal matrix and $\boldsymbol{Q}$ is an orthonormal basis of eigenvectors.

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \eta \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*)$$

$$\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) = (\boldsymbol{I} - \eta \boldsymbol{\Lambda})\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*)$$

Assuming that $\boldsymbol{w}^0 = 0$ and $|1 - \eta\lambda_i| < 1$, the parameter trajectory during training after $\tau$ parameter updates is as follows:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \eta\boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.31}$$

Now, the expression for $\boldsymbol{Q}^\top \tilde{\boldsymbol{w}}$ in Eqn. 7.9 for $L^2$ regularization can rearrange as:

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*$$

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha]\boldsymbol{Q}^\top \boldsymbol{w}^* \tag{7.32}$$

Comparing Eqs 7.31 and 7.32, we see that if

$$(\boldsymbol{I} - \eta\boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha,$$

then $L^2$ regularization and early stopping can be seen to be equivalent (at least under the quadratic approximation of the objective function). Going even further, by taking logs and using the series expansion for $\log(1+x)$, we can conclude that if all $\lambda_i$ are small (that is, $\eta\lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx \frac{1}{\eta\alpha},$$

$$\alpha \approx \frac{1}{\tau\eta}. \tag{7.33}$$

That is, under these assumptions, the number of training iterations $\tau$ plays a role inversely proportional to the $L^2$ regularization parameter, and the inverse of $\tau\eta$ plays the role of the weight decay coefficient.

Parameter values corresponding to directions of significant curvature (of the objective function) are regularized less than directions of less curvature. Of course, in the context of early stopping, this really means that parameters that correspond to directions of significant curvature tend to learn early relative to parameters corresponding to directions of less curvature.

## 7.8 Parameter Tying and Parameter Sharing

Thus far, in this chapter, when we have discussed adding constraints or penalties to the parameters, we have always done so with respect to a fixed region or point. For example, $L^2$ regularization (or weight decay) penalizes model parameters for deviating from the fixed value of zero. However, sometimes we may need other ways to express our prior knowledge about suitable values of the model parameters. Sometimes we might not know precisely what values the parameters should take but we know, from knowledge of the domain and model architecture, that there should be some dependencies between the model parameters.

A common type of dependency that we often want to express is that certain parameters should be close to one another.

Consider the following scenario: we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions. Formally, we have model $a$ with parameters $\boldsymbol{w}^{(a)}$ and model $b$ with parameters $\boldsymbol{w}^{(b)}$. The two models map the input to two different, but related outputs: $\hat{y}_a = f(\boldsymbol{w}^{(a)}, \boldsymbol{x})$ and $\hat{y}_b = g(\boldsymbol{w}^{(b)}, \boldsymbol{x})$.

Let us imagine that the tasks are similar enough (perhaps with similar input and output distributions) that we believe the model parameters should be close to each other: $\forall i,\ w_i^{(a)}$ should be close to $w_i^{(b)}$. We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form: $\Omega(\boldsymbol{w}^{(a)}, \boldsymbol{w}^{(b)}) = \|\boldsymbol{w}^{(a)} - \boldsymbol{w}^{(b)}\|_2^2$. Here we used an $L^2$ penalty, but other choices are also possible.

This kind of approach was proposed in Lasserre *et al.* (2006), where they regularized the parameters of one model, trained as a classifier in a supervised paradigm, with the parameters of another model, this one trained in an unsupervised paradigm (to capture the distribution of the observed input data). The architectures were constructed such that many of the parameters in the classifier model could be paired to corresponding parameters in the unsupervised model.

While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: *to force sets of parameters to be equal*. This method of regularization is often referred to as *parameter sharing*, where we interpret the various models or model components as sharing a unique set of parameters. A significant advantage of parameter sharing over regularizing the parameters to be close (via a norm penalty) is that only a subset of the parameters (the unique set) need to be stored in memory. In certain models—such as the convolutional neural network—this can lead to significant reduction in the memory footprint of the model.

**Convolutional Neural Networks** By far the most popular and extensive use of parameter sharing occurs in *convolutional neural networks* (CNNs) applied to computer vision.

Natural images have many statistical properties that are invariant to translation. For example, a photo of a cat remains a photo of a cat if it is translated one pixel to the right. CNNs take this property into account by sharing parameters across multiple image locations. The same feature (a hidden unit with the same weights) is computed over different locations in the input. This means that we can find a cat with the same cat detector whether the cat appears at column $i$ or column $i+1$ in the image.

Parameter sharing has allowed CNNs to dramatically lower the number of unique model parameters and have allowed them to significantly increase network sizes without requiring a corresponding increase in training data. It remains one of the best examples of how to effectively incorporate domain knowledge into the network architecture.

CNNs will be discussed in more detail in Chapter 9.

## 7.9 Sparse Representations

The previous sections of this chapter were concerned with direct regularization of the model parameters. In this section we will describe a different kind of regularization strategy where the effect on the model parameters is only indirect. Specifically we consider representational sparsity as a form of regularization. We have already discussed (in sec. 7.2.2) how $L^1$ penalization induces a sparse parametrization – meaning that a significant number of the parameters is zero (or close to it). Representational sparsity, on the other hand, describes a representation where a significant number of elements are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

$$\underbrace{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}}_{\boldsymbol{y} \in \mathbb{R}^m} = \underbrace{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}}_{\boldsymbol{A} \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}}_{\boldsymbol{x} \in \mathbb{R}^n}$$

$$\underbrace{\begin{bmatrix} -14 \\ 1 \\ 1 \\ 2 \\ 23 \end{bmatrix}}_{\boldsymbol{y} \in \mathbb{R}^m} = \underbrace{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}}_{\boldsymbol{B} \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}_{\boldsymbol{h} \in \mathbb{R}^n}$$

In the first expression, we have an example of a sparsely parametrized linear regression model. In the second, we have linear regression with a sparse representation $\boldsymbol{h}$ of the data $\boldsymbol{x}$. That is, $\boldsymbol{h}$ is a function of $\boldsymbol{x}$ that, in some sense, represents the information present in $\boldsymbol{x}$, but does so with a sparse vector.

The distinction between representation sparsity and parameter sparsity is important, but more generally regularizing the representation and regularizing the model parameters can be related to each other in a fairly natural way. Ultimately, the goal of either strategy is to regularize the function mapping the input to the output space (regardless of the task). Sometimes it is more natural to express constraints or penalties in the form of prior information. However sometimes the connection between the parameters and the function being regularized is not well understood and it is difficult to know, *a priori*, how to specify a penalty or constraint on the model parameters that matches our prior knowledge of the function being learning. In these situations it may well be more natural or more effective to regularize the representations learned by the model.

Representational regularization is mediated by the same sorts of mechanisms that we have used in parameter regularization. Specifically both soft norm penalties and hard constraints may be considered - though norm penalties have been the more popular of the two in the deep learning community.

Norm penalty regularization of representations is performed by adding to the loss function $J$ a norm penalty on the *representation*, denoted $\Omega(\boldsymbol{h})$. As before, we denote the regularized loss function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{h}) \tag{7.34}$$

where $\alpha$ ($\alpha \geq 0$) weights the relative contribution of the norm penalty term, with larger values of $\alpha$ corresponding to more regularization.

Just as an $L^1$ penalty on the parameters induces parameter sparsity, an $L^1$ penalty on the elements of the representation induces representational sparsity: $\Omega(\boldsymbol{h}) = |\boldsymbol{h}|_1 = \sum_i |h_i|$. Of course, the $L^1$ penalty is only one choice of penalty that can result in a sparse representation. Others include the Student-$t$ penalty (derived from a Student-$t$ distribution prior on the elements of the representation) (Olshausen and Field, 1996; Bergstra, 2011) and KL-divergence penalties (Lee *et al.*, 2008; Larochelle and Bengio, 2008a; Goodfellow *et al.*, 2009) that are especially

useful for representations with elements constrained to lie on the unit interval. In Sec. 13.7, sparsity inducing penalties are discussed in the context of auto-encoders.

## 7.10 Bagging and Other Ensemble Methods

*Bagging* (short for *bootstrap aggregating*) is a technique for reducing generalization error by combining several models (Breiman, 1994). The idea is to train several different models separately, then have all of the models vote on the output for test examples. This is an example of a general strategy in machine learning called *model averaging*. Techniques employing this strategy are known as *ensemble methods*.

The reason that model averaging works is that different models will usually not make all the same errors on the test set.

Consider for example a set of $k$ regression models. Suppose that each model makes an error $\epsilon_i$ on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Then the error made by the average prediction of all the ensemble models is $\frac{1}{k}\sum_i \epsilon_i$. The expected squared error of the ensemble predictor is

$$\begin{aligned}\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] &= \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right] \\ &= \frac{1}{k}v + \frac{k-1}{k}c.\end{aligned}$$

In the case where the errors are perfectly correlated and $c = v$, the mean squared error reduces to $v$, so the model averaging does not help at all. In the case where the errors are perfectly uncorrelated and $c = 0$, the expected squared error of the ensemble is only $\frac{1}{k}v$. This means that the expected squared error of the ensemble decreases linearly with the ensemble size. In other words, on average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.

Different ensemble methods construct the ensemble of models in different ways. For example, each member of the ensemble could be formed by training a completely different kind of model using a different algorithm or objective function. Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times.

Specifically, bagging involves constructing $k$ different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means

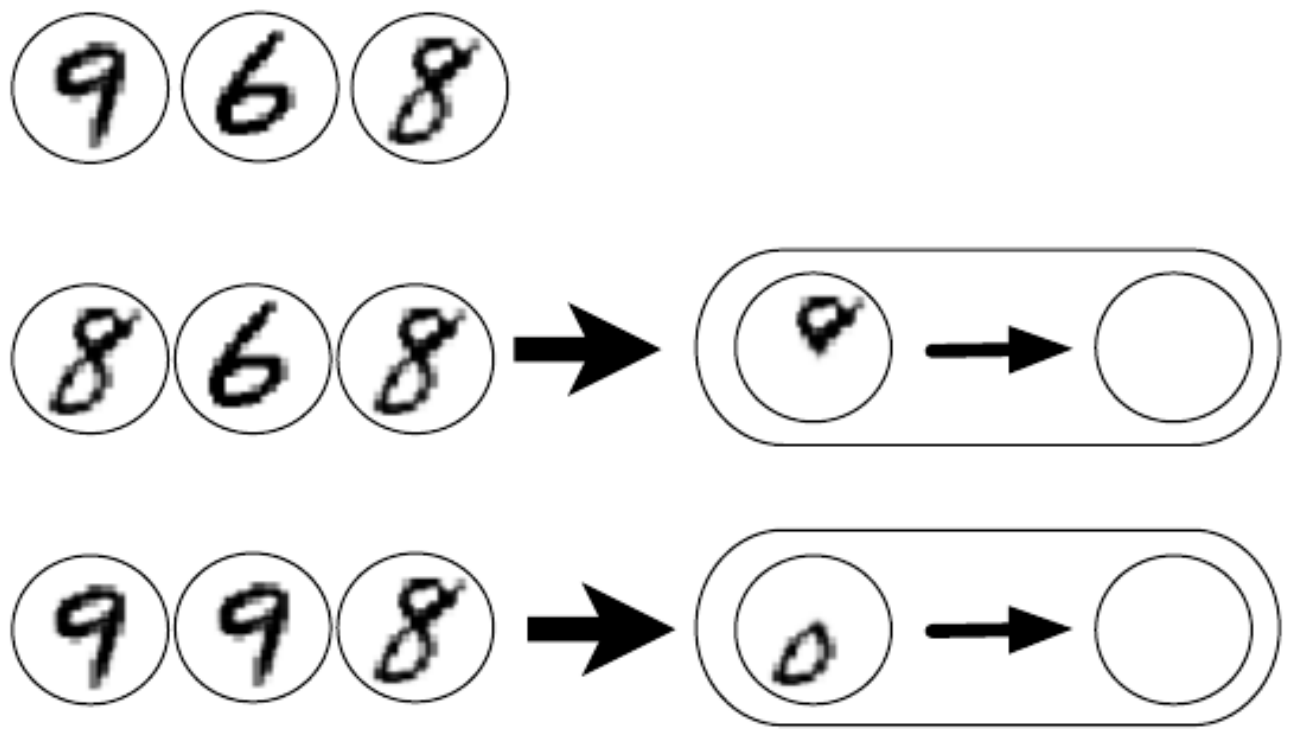

Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an '8' detector on the dataset depicted above, containing an '8', a '6' and a '9'. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the '9' and repeats the '8'. On this dataset, the detector learns that a loop on top of the digit corresponds to an '8'. On the second dataset, we repeat the '9' and omit the '6'. In this case, the detector learns that a loop on the bottom of the digit corresponds to an '8'. Each of these individual classification rules is brittle, but if we average their output then the detector is robust, achieving maximal confidence only when both loops of the '8' are present.

that, with high probability, each dataset is missing some of the examples from the original dataset and also contains several duplicate examples (in average around 2/3 of the examples from the original dataset are found in the resulting training set, if it has the same size as the original). Model $i$ is then trained on dataset $i$. The differences between which examples are included in each dataset result in differences between the trained models. See Fig. 7.5 for an example.

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all of the models are trained on the same dataset. Differences in random initialization, random selection of minibatches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

Model averaging is an extremely powerful and reliable method for reducing generalization error. Its use is usually discouraged when benchmarking algorithms for scientific papers, because any machine learning algorithm can benefit substantially from model averaging at the price of increased computation and memory. For this reason, benchmark comparisons are usually made using a single model.

Machine learning contests are usually won by methods using model averag-

ing over dozens of models. A recent prominent example is the Netflix Grand Prize (Koren, 2009).

Not all techniques for constructing ensembles are designed to make the ensemble more regularized than the individual models. For example, a technique called *boosting* (Freund and Schapire, 1996b,a) constructs an ensemble with higher capacity than the individual models. Boosting has been applied to build ensembles of neural networks (Schwenk and Bengio, 1998) by incrementally adding neural networks to the ensemble. Boosting has also been applied interpreting an individual neural network as an ensemble (Bengio *et al.*, 2006a), incrementally adding hidden units to the neural network.

## 7.11 Dropout

Because deep models with many parameters have a high degree of expressive power necessary to capture complex tasks such as speech or object recognition, they are capable of overfitting significantly. While this problem can be solved by using a very large dataset, large datasets are not always available. *Dropout* (Srivastava *et al.*, 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models.

Dropout can be thought of as a method of making bagging practical for large neural networks. Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing units from an underlying base network. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. This procedure requires some slight modification for models such as radial basis function networks, which take the difference between the unit's state and some reference value. Here, we will present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

For many classes of models that do not have nonlinear hidden units, the weight scaling inference rule is exact. For a simple example, consider a softmax regression classifier with $n$ input variables represented by the vector $\mathbf{v}$:

$$P(\mathrm{y} = y \mid \mathbf{v}) = \mathrm{softmax}\left(\boldsymbol{W}^\top \mathbf{v} + \boldsymbol{b}\right)_y .$$

We can index into the family of sub-models by element-wise multiplication of the

input with a binary vector $d$:

$$P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d}) = \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y .$$

The ensemble predictor is defined by re-normalizing the geometric mean over all ensemble members' predictions:

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})} \tag{7.35}$$

where

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}.$$

To see that the weight scaling rule is exact, we can simplify $\tilde{P}_{\text{ensemble}}$:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \frac{\exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}{\sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}}$$

$$= \frac{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}}{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}}$$

Because $\tilde{P}$ will be normalized, we can safely ignore multiplication by factors that are constant with respect to $y$:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\boldsymbol{d} \in \{0,1\}^n} \boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)$$

$$= \exp\left(\frac{1}{2} \boldsymbol{W}_{y,:}^\top \mathbf{v} + \boldsymbol{b}\right)$$

Substituting this back into equation 7.35 we obtain a softmax classifier with weights $\frac{1}{2}\boldsymbol{W}$.

The weight scaling rule is also exact in other settings, including regression networks with conditionally normal outputs, and deep networks that have hidden layers without nonlinearities. However, the weight scaling rule is only an approximation for deep models that have non-linearities. Though the approximation has not been theoretically characterized, it often works well, empirically. Goodfellow *et al.* (2013a) found experimentally that the weight scaling approximation can work better (in terms of classification accuracy) than Monte Carlo approximations to the ensemble predictor. This held true even when the Monte Carlo approximation was allowed to sample up to 1,000 sub-networks. Gal and Ghahramani (2015) found that some models obtain better classification accuracy using twenty samples and the Monte Carlo approximation. It appears that the optimal choice of inference approximation is problem-dependent.

Srivastava *et al.* (2014) showed that dropout is more effective than other standard computationally inexpensive regularizers, such as weight decay, filter norm constraints and sparse activity regularization. Dropout may also be combined with more expensive forms of regularization such as unsupervised pretraining to yield an improvement.

One advantage of dropout is that it is very computationally cheap. Using dropout during training requires only $O(n)$ computation per example per update, to generate $n$ random binary numbers and multiply them by the state. Depending on the implementation, it may also require $O(n)$ memory to store these binary numbers until the backpropagation stage. Running inference in the trained model has the same cost per-example as if dropout were not used, though we must pay the cost of dividing the weights by 2 once before beginning to run inference on examples.

One significant advantage of dropout is that it does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent. This includes feedforward neural networks, probabilistic models such as restricted Boltzmann machines (Srivastava *et al.*, 2014), and recurrent neural networks (Bayer and Osendorfer, 2014; Pascanu *et al.*, 2014a). On the other hand, if one wants to take advantage of the regularization effect of approaches that try to capture the structure of the input distribution, such as unsupervised pretraining, it is often necessary to change the architecture of the model.

Though the cost per-step of applying dropout to a specific model is negligible, the cost of using dropout in a complete system can be significant. Because dropout is a regularization technique, it reduces the effective capacity of a model. To offset this effect, we must increase the size of the model. Typically the optimal validation

set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm. For very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

When extremely few labeled training examples are available, dropout is less effective. Bayesian neural networks (Neal, 1996) outperform dropout on the Alternative Splicing Dataset (Xiong *et al.*, 2011) where fewer than 5,000 examples are available (Srivastava *et al.*, 2014). When additional unlabeled data is available, unsupervised feature learning can gain an advantage over dropout.

The stochasticity used while training with dropout is not a necessary part of the model's success. It is just a means of approximating the sum over all sub-models. Wang and Manning (2013) derived analytical approximations to this marginalization. Their approximation, known as *fast dropout* resulted in faster convergence time due to the reduced stochasticity in the computation of the gradient. This method can also be applied at test time, as a more principled (but also more computationally expensive) approximation to the average over all sub-networks than the weight scaling approximation. Fast dropout has been used to match the performance of standard dropout on small neural network problems, but has not yet to be applied to a large problem.

Dropout has inspired other stochastic approaches to training exponentially large ensembles of models that share weights. DropConnect is a special case of dropout where each product between a single scalar weight and a single hidden unit state is considered a unit that can be dropped (Wan *et al.*, 2013). Stochastic pooling is a form of randomized pooling (see chapter 9.3) for building ensembles of convolutional networks with each convolutional network attending to different spatial locations of each feature map. So far, dropout remains the most widely used implicit ensemble method.

## 7.12 Multi-Task Learning

Multi-task learning (Caruana, 1993) is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks. In the same way that additional training examples put more pressure on the parameters of the model towards values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained towards good values (assuming the sharing is justified), often yielding better generalization.

Figure 7.6 illustrates a very common form of multi-task learning, in which different supervised tasks (predicting $\mathbf{y}_i$ given $\mathbf{x}$) share the same input $\mathbf{x}$, as well

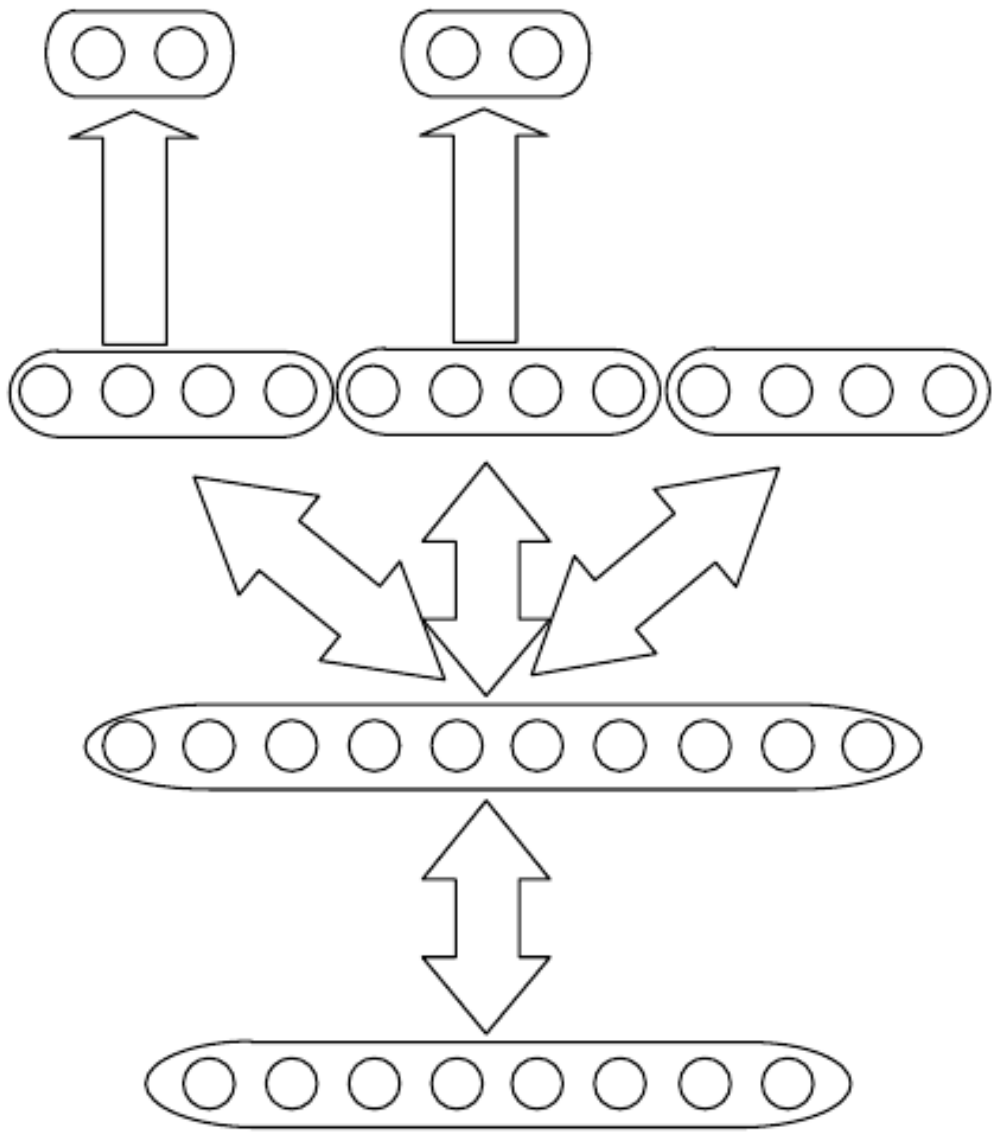

Figure 7.6: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $\boldsymbol{h}$ and $\boldsymbol{h}$ in the figure) can be learned on top of those yielding a shared representation $\boldsymbol{h}$ . The underlying assumption is that there exists a common pool of factors that explain the variations in the input $\mathbf{x}$, while each task is associated with a subset of these factors. In the figure, it is additionally assumed that top-level hidden units $\boldsymbol{h}$ and $\boldsymbol{h}$ are specialized to each task (respectively predicting $\mathbf{y}$ and $\mathbf{y}$ ) while some intermediate-level representation $\boldsymbol{h}$ is shared across all tasks. Note that in the unsupervised learning context, it makes sense for some of the top-level factors to be associated with none of the output tasks ($\boldsymbol{h}$ ): these are the factors that explain some of the input variations but are not relevant for predicting $\mathbf{y}$ or $\mathbf{y}$ .

as some intermediate-level representation $\boldsymbol{h}_{\text{shared}}$ capturing a common pool of factors. The model can generally be divided into two kinds of parts and associated parameters:

1. Task-specific parameters (which only benefit from the examples of their task to achieve good generalization). Example: upper layers of a neural network, in Figure 7.6.

2. Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). Example: lower layers of a neural network, in Figure 7.6.

Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be greatly improved (in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models). Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

From the point of view of deep learning, the underlying prior regarding the data is the following: *among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.*

## 7.13 Adversarial Training

In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set. It is natural therefore to wonder whether these models have obtained a true human-level understanding of these tasks. In order to probe the level of understanding a network has of the underlying task, we can search for examples that the model misclassifies. Szegedy *et al.* (2014b) found that even neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intentionally constructed by using an optimization procedure to search for an input $\boldsymbol{x}'$ near a data point $\boldsymbol{x}$ such that the model output is very different at $\boldsymbol{x}'$. In many case, $\boldsymbol{x}'$ can be so similar to $\boldsymbol{x}$ that a human observer cannot tell the difference between the original example and the *adversarial example*, but the network can make highly different predictions. See Fig. 7.7 for an example.

Adversarial examples have many implications, for example, in computer security, that are beyond the scope of this chapter. However, they are interesting in the context of regularization because one can reduce the error rate on the original i.i.d. test set by training on adversarially perturbed examples from the training set (Szegedy *et al.*, 2014b).

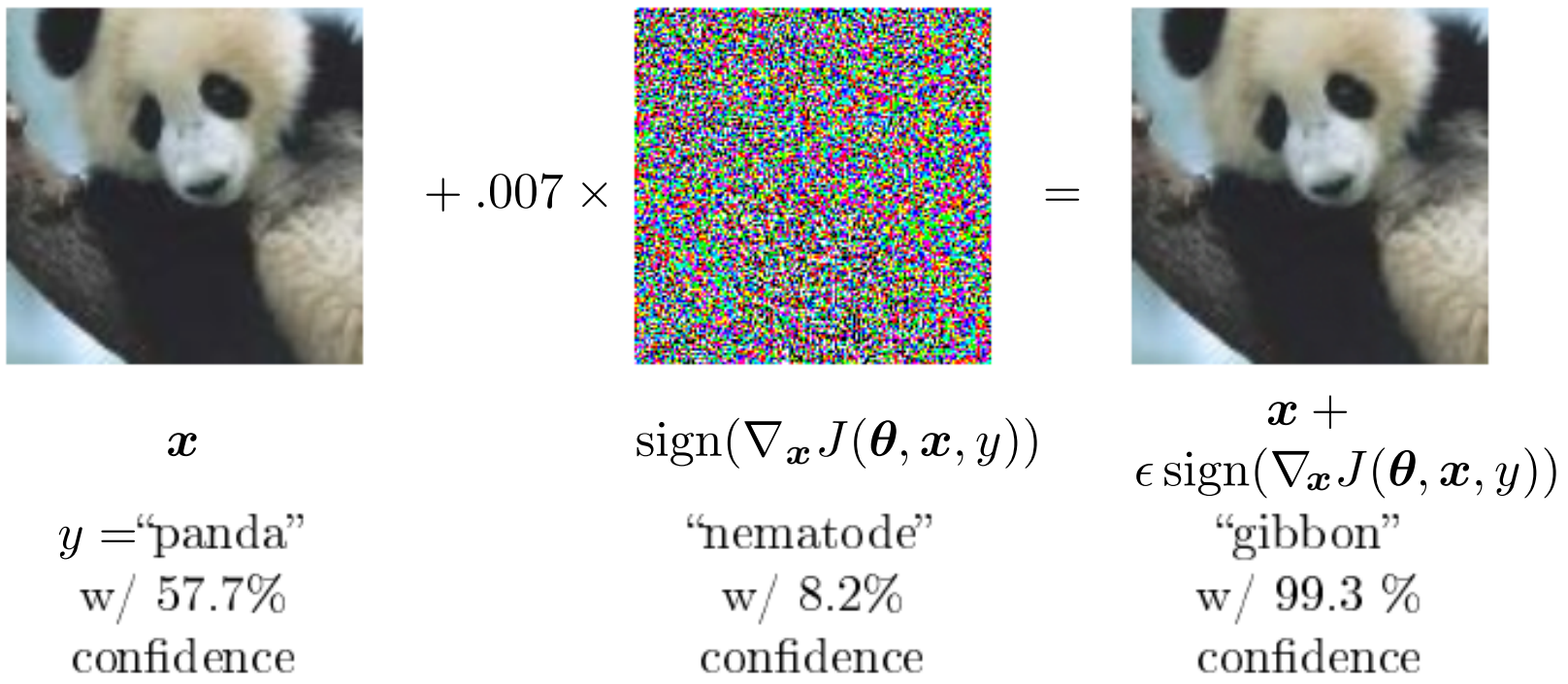


Figure 7.7: A demonstration of adversarial example generation applied to GoogLeNet (Szegedy *et al.*, 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Reproduced with permission from Goodfellow *et al.* (2014b).

Goodfellow *et al.* (2014b) showed that one of the primary causes of these adversarial examples is excessive linearity. Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result. These linear functions are easy to optimize. Unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. If we change each input by $\epsilon$, then a linear function with weights $\boldsymbol{w}$ can change by as much as $\epsilon|\boldsymbol{w}|$, which can be a very large amount if $\boldsymbol{w}$ is high-dimensional. Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data. This can be seen as a way of introducing explicitly the local smoothness prior into supervised neural nets.

This phenomenon helps to illustrate the power of using a large function family in combination with aggressive regularization. Purely linear models, like logistic regression, are not able to resist adversarial examples because they are forced to be linear. Neural networks are able to represent functions that can range from nearly linear to nearly locally constant and thus have the flexibility to capture linear trends in the training data while still learning to resist local perturbation.

# Chapter 8

# Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, we often solve optimization problems analytically in order to prove that an algorithm has a certain property. Inference in a probabilistic model can be cast as an optimization problem. Of all of the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds on machines in order to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it. This chapter presents these optimization techniques for neural network training.

If you're unfamiliar with the basic principles of gradient-based optimization, we suggest reviewing Chapter 4. That chapter includes a brief overview of numerical optimization in general.

This chapter focuses on one particular case of optimization: minimizing an objective function $J(\boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$, which is also implicitly a function of the training data. Typically, that objective function can be rewritten as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{8.1}$$

where $L$ is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, $y$ is the target output and $\hat{p}_{\text{data}}$ is the empirical distribution, in the supervised learning case. However, we would usually prefer to minimize the corresponding objective function where the expectation is taken across *the data generating distribution* $p_{\text{data}}$ rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{8.2}$$

## 8.1 Optimization for Model Training

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly—we care about some performance measure $P$ that we do not know how to directly influence, so instead we reduce some objective function $J(\boldsymbol{\theta})$ in hope that it will improve $P$. This is in contrast to pure optimization, where minimizing $J$ is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions.

### 8.1.1 Empirical Risk Minimization

Suppose that we have input feature vector $\boldsymbol{x}$ and targets $y$, sampled from some unknown joint distribution $p(\boldsymbol{x}, y)$, as well as some loss function $L(\boldsymbol{x}, y)$. Our ultimate goal is to minimize $\mathbb{E}_{\boldsymbol{x},y\sim p(\boldsymbol{x},y)}[L(\boldsymbol{x}, y)]$. This quantity is known as the *risk*. We emphasize here that the expectation is taken over the true underlying distribution $p$, so the risk is a form of generalization error. If we knew the true distribution $p(\boldsymbol{x}, y)$, this would be an optimization task solvable by an optimization algorithm. However, when we do not know $p(\boldsymbol{x}, y)$ but only have a training set of samples from it, we have a machine learning problem.

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. We now minimize the *empirical risk*

$$\mathbb{E}_{\boldsymbol{x},y\sim\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

where $m$ is the number of training examples.

The training process based on minimizing this average training error is known as *empirical risk minimization*. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases significantly as well. A variety of theoretical results establish conditions under which the true risk can be expected to decrease by various amounts.

However, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. In many cases, empirical risk minimization is not really feasible. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we

rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we actually optimize is even more different from the quantity that we truly want to optimize.

### 8.1.2 Surrogate Loss Functions

Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier (Marcotte and Savard, 1992). Practically, note that we cannot use gradient-based optimization on the 0-1 loss because its derivative is 0 almost everywhere. In such situations, one typically optimizes a *surrogate loss function* instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used with neural networks. It allows the model to estimate the conditional probability of the classes, given the input, and if we can do that well, then we can pick the classes that yield the less classification error in expectation. In other cases, the actual loss function is one that is very expensive to obtain. Note that in some cases, a surrogate loss function actually results in being able to learn more. For example, the test set 0-1 loss often continues to decrease for a long time after the training set 0-1 loss has reached zero, when training using the log-likelihood surrogate. This is because even when the expected 0-1 loss is 0, one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.

As discussed in Section 7.7, although one uses a surrogate loss function for the optimization objective, one can still use the actual loss of interest (if it is computable) to monitor progress and perform early stopping, a form of regularization. A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. Instead, using early stopping, they halt whenever overfitting begins to occur. This is often in the middle of a wide, flat region, but it can also occur on a steep part of the surrogate loss function. This is in contrast to general optimization, where converge is usually defined by arriving at a point that is very near a (local) minimum.

### 8.1.3 Batch and Minibatch Algorithms

One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a

sum over the training examples. Optimization algorithms for machine learning typically compute each update to the parameters based on a subset of the terms of the objective function, not based on the complete objective function itself.

For example, maximum likelihood estimation problems, when viewed in log space, decompose into a sum over each example:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.3}$$

Most of the properties of the objective function $J$ used by most of our optimization algorithms are also expectations over the training set. For example, the most commonly used property is the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.4}$$

Computing this expectation exactly is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

Recall that the standard error of the mean estimated from $n$ samples is given by $\hat{\sigma}/\sqrt{n}$, where $\hat{\sigma}$ is the estimated standard deviation. This means that there are less than linear returns to using more examples to estimate the gradient. Compare two hypothetical estimates of the gradient, one based on 100 examples and another based on 10,000 examples. The latter requires 100 times more computation than the former, but reduces the standard error of the mean only by a factor of 10. Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

Another way to intuitively understand the appeal of statistically estimating the gradient from a small number of samples is to consider that there may be redundancy in the training set. In the worst case, all $m$ samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with 1 sample, using $m$ times less computation than the naive approach. In practice, we are unlikely to truly encounter this worst-case situation, but we may find large numbers of examples that all make very functionally similar contributions to the gradient.

Optimization algorithms that use the entire training set are called *batch* or *deterministic* gradient methods, because they process all of the training examples

simultaneously in a large batch. Optimization algorithms that use only a single example at a time are sometimes called *stochastic* or sometimes *online* methods (the term online is usually reserved for the case where the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes will be made). Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training examples. These were traditionally called *minibatch* or *minibatch stochastic* methods and it is now common to simply call them *stochastic* methods.

The canonical example of a stochastic method is stochastic gradient descent, presented in detail in Sec. 8.3.2.

Minibatch sizes are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPU, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1, though this might take a very long time to train and require a small learning rate to maintain stability.

Different kinds of algorithms use different kinds of information in different ways. Some algorithms are more sensitive to sampling error than others, either because they use information that is difficult to estimate accurately with few samples, or because they use information in ways that amplify sampling errors more. Methods that compute updates based only on the gradient $\boldsymbol{g}$ are usually relatively robust and can handle smaller batch sizes like 100. Second order methods, that use also the Hessian matrix $\boldsymbol{H}$ and compute updates such as $\boldsymbol{H}^{-1}\boldsymbol{g}$, typically require much larger batch sizes like 10,000. To see this, consider that when $\boldsymbol{H}$ and its inverse are poorly conditioned, then very small changes in the estimate of

$\boldsymbol{g}$ can cause large changes in the update $\boldsymbol{H}^{-1}\boldsymbol{g}$. This is further compounded by estimation error in $\boldsymbol{H}$ itself.

Computing the expected gradient from a set of samples requires that those samples be independent. Many datasets are most naturally arranged in a way where successive examples are highly correlated. For example, we might have a dataset of medical data with a long list of blood sample test results. This list might be arranged so that first we have five blood samples taken at different times from the first patient, then we have three blood samples taken from the second patient, then the blood samples from the third patient, and so on. If we were to draw examples in order from this list, then each of our minibatches would be extremely biased, because it would represent primarily one patient out of the many patients in the dataset. In cases such as these where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches. For very large datasets, for example datasets containing billions of examples on a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion. This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to re-use this ordering every time it passes through the training data, however, this deviation from true random selection does not seem to have a significant detrimental effect.

Many optimization problems in machine learning decompose over examples well enough that we can compute entire separate updates over different examples in parallel. In other words, we can compute the update that minimizes $J(\boldsymbol{x})$ for one minibatch of examples $\boldsymbol{x}$ at the same time that we compute the update for several other minibatches. This is discussed further in Chapter 12.1.3.

### 8.1.4 Online Gradient Descent Minimizes Generalization Error

In machine learning, typically we minimize a objective function defined as an expectation of some per-example loss across the training set, such as in Eq. 8.1, but we really care about minimizing the corresponding generalization error, as in Eq. 8.2.

Usually, we use an optimization algorithm based on minibatch estimates of the gradient. As we argue below, during the first stages of learning, this is equivalent to minimizing the generalization error directly. However, after we have pass through the training data once and begin to repeat minibatches, the two criteria diverge.

Let us consider the "online" learning case, where examples or minibatches are drawn from a *stream* of data. In other words, instead of a fixed-size training set,

we are in the situation similar to a living being who sees a new example at each instant, with every example $(\boldsymbol{x}, y)$ coming from the data generating distribution $p(\boldsymbol{x}, y)$ (the same argument could be made in the unsupervised case, where there is no $y$).

Consider a loss function $L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$ whose expected value over $p(\boldsymbol{x}, y)$ we would like to minimize with respect to the parameters $\boldsymbol{\theta}$. In other words, the generalization error (Eq. 8.2) of the current predictor $f(\cdot; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$ can be rewritten as

$$J^*(\boldsymbol{\theta}) = \int L(f(\boldsymbol{x}; \boldsymbol{\theta}), y) dp(\boldsymbol{x}, y)$$

and under continuity assumptions of $p$, its exact gradient is

$$\boldsymbol{g} = \frac{\partial J^*(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \int \frac{\partial L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)}{\partial \boldsymbol{\theta}} dp(\boldsymbol{x}, y),$$

similarly to what we have seen in Eqs. 8.3 and 8.4 for the log-likelihood. Hence, we can obtain an unbiased estimator of the exact gradient of generalization error by sampling one example $(\boldsymbol{x}, y)$ (or equivalently a minibatch) from the data generating process $p$, and computing the gradient of the loss with respect to the parameters for that example (or that minibatch),

$$\hat{\boldsymbol{g}} = \frac{\partial L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)}{\partial \boldsymbol{\theta}}.$$

It should be clear that this stochastic gradient estimator $\hat{\boldsymbol{g}}$ is a noisy but unbiased estimator of the *exact gradient of generalization error*, $\boldsymbol{g}$. Hence, updating $\boldsymbol{\theta}$ in the direction of $\hat{\boldsymbol{g}}$ performs SGD on the generalization error. Of course, this is only possible if the examples are not repeated (unlike in the usual scenario for many machine learning applications, where several epochs through the training set are performed). With some datasets growing rapidly in size, faster than computing power, this online scenario is actually quite plausible. In that setting, overfitting is not an issue, while underfitting and computational efficiency matter a lot. See also Bottou and Bousquet (2008) for a discussion of the effect of computational bottlenecks on generalization error, as the number of training examples grows.

## 8.2 Challenges in Neural Network Optimization

Optimization in general is an extremely difficult task. Traditionally, machine learning has avoided this difficulty by carefully designing the objective function and constraints to ensure that the optimization problem is convex. When training neural networks, we must confront the general non-convex case, which introduces many difficulties. In this section, we summarize several of the most prominent challenges.

### 8.2.1 Ill-Conditioning

Some challenges arise even when optimizing convex functions. Of these, the most prominent is ill-conditioning of the Hessian matrix. Suppose we update our parameters using a gradient descent step $\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \boldsymbol{g}$ where $\alpha$ is a learning rate and $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. A second-order Taylor series expansion predicts that the value of the cost function at the new point is given by

$$J(\boldsymbol{\theta}') \approx J(\boldsymbol{\theta}) - \alpha \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\alpha^2 \boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$$

where $\boldsymbol{H}$ is the Hessian of $J$ with respect to $\boldsymbol{\theta}$. The $-\alpha \boldsymbol{g}^\top \boldsymbol{g}$ term is always negative—if the cost function were a linear function of the parameters, gradient descent would always move downhill. However, the second-order term $\frac{1}{2}\alpha^2 \boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$ can be negative or positive depending on the eigenvalues of $\boldsymbol{H}$ and the alignment of the corresponding eigenvectors with $\boldsymbol{g}$. On steps where $\boldsymbol{g}$ aligns closely with large, positive eigenvalues of $\boldsymbol{H}$, the learning rate must be very small, or the second-order term will result in gradient descent accidentally moving uphill.

This is a very general problem in most numerical optimization, convex or otherwise, and is described in more detail in Sec. 4.2.

The ill-conditioning problem is generally believed to be present in neural networks. It can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function. We can monitor the squared gradient norm $\boldsymbol{g}^\top \boldsymbol{g}$ and the $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$ term grows by more than order of magnitude.

Though ill-conditioning is present in other settings besides neural network training, some of the techniques used to combat it in other contexts are less applicable to neural networks. For example, Newton's method is an excellent tool for minimizing convex functions with poorly conditioned Hessian matrices, but in the subsequent sections we will argue that Newton's method requires significant modification before it can be applied to neural networks.

### 8.2.2 Local Minima

One of the most prominent features of a convex optimization problem is that a strictly convex function contains only one local minimum, which is also the global minimum. A weakly convex function may contain multiple global minima, but they are conveniently grouped into a single convex set in which all solutions are equivalent. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind.

With non-convex functions, such as neural nets, it is possible to have many local minima.

Neural networks and any models with multiple equivalently parameterized latent variables all have multiple local minima because of the *model identifiability* problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other. For example, we could take a neural network and modify layer 1 by swapping the income weight vector for unit $i$ with the incoming weight vector for unit $j$, then doing the same for the outgoing weight vectors. If we have $m$ layers with $n$ units each, then there are $n!^m$ ways of arranging the hidden units. This kind of non-identifability is known as *weight space symmetry.*

In addition to weight space symmetry, many kinds of neural networks have additional causes of non-identifiability. For example, in any rectified linear or maxout network, we can scale all of the incoming weights and biases of a unit by $\alpha$ if we also scale all of its outgoing weights by $1/\alpha$. This means that—if the cost function does not include terms such of weight decay that depend directly on the weights rather than the models' outputs—every local minimum of a rectified linear or maxout network lies on an $(m \times n)$-dimensional hyperbola of equivalent local minima.

These model identifiability issues mean that there can be an extremely large or even uncountably infinite amount of local minima in a neural network cost function. However, all of these local minima arising from non-identifiability are equivalent to each other in cost function value. As a result, these local minima are not a problematic form of non-convexity.

Local minima can be problematic if they have high cost. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992). If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

It remains an open question whether there are many local minima of high cost for networks of practical interest and whether optimization algorithms encounter them. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case. The problem remains an active area of research, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than a local minimum that also has very low value (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014).

Many practitioners attribute nearly all difficulty with neural network optimization to local minima. We encourage practitioners to carefully test for specific problems. To verify that a problem arises from local minima, plot the norm of the

gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is not local minima.

### 8.2.3 Ill-Conditioning

In Sec. 4.2 we discussed the general notion of poor conditioning in numerical computation. We defined the condition number of a matrix as the ratio of its largest to its smallest eigenvalue (in magnitude). In optimization, ill-conditioning refers to the difficulty that arises when the objective function to be minimized has eigenvalues that are tiny or even 0. In that case, inverting the Hessian matrix or optimizing the objective function both become numerically difficult. The condition number of the Hessian is directly linked to the number of training iterations needed by gradient descent, for example.

### 8.2.4 Plateaus, Saddle Points and Other Flat Regions

In a convex problem, any point with zero gradient is a global minimum. In a non-convex problem, a point with zero gradient may be a global minimum. It may also be a local minimum, as discussed above. Traditionally, local minima received much attention and fear of them was one of the reasons for the "neural networks winter" of 1995-2005.

However, local minima and maxima may in fact be rare compared to another kind of point with zero gradient: a saddle point. Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value. We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher dimensional spaces, local minima are rare and saddle points are exponentially more common. To understand the intution behind this, observe that a local minimum has only positive eigenvalues. A saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In 1-dimensional space, it is easy to obtain a local minimum by tossing a coin and getting heads once. In $n$-dimensional space, it is exponentially unlikely that all $n$ coin tosses will be heads. See Dauphin *et al.* (2014) for a review of the relevant theoretical work.

An amazing property of many random functions is that the eigenvalues become

more likely to be positive as we reach regions of lower cost. In our coin tossing analogy, this means we are more likely to have our coin come up heads $n$ times if we are at a critical point with low cost. This means that local minima are much more likely to have low cost than high cost. Critical points with high cost are far more likely to be saddle points.

This happens for many classes of random functions. Does it happen for neural networks? Baldi and Hornik (1989) showed theoretically that shallow auto-encoders with no non-linearities have global minima and saddle points but no local minima with higher cost than the global minimum. Saxe *et al.* (2013) showed theoretically the same result for deep networks without non-linearities. Such networks are essentially just multiple matrices composed together. Their output is a linear function of their input, but they are useful to study as a model of non-linear neural networks because their loss function is a non-convex function of their parameters. Dauphin *et al.* (2014) showed experimentally that real neural networks also have this behavior. Choromanska *et al.* (2014) provided additional theoretical arguments, showing that another class of high-dimensional random functions related to neural networks does so as well.

What are the implications of the proliferation of saddle points for training algorithms? This remains unclear. Some practitioners believe that saddle points could cause a serious problem. Currently, no evidence exists to suggest that first-order methods are hindered by saddle points. Goodfellow *et al.* (2015) provided visualizations of several learning trajectories of state-of-the-art neural networks and found no evidence of gradient descent slowing down near saddle points. Goodfellow *et al.* (2015) also argue that continuous-time gradient descent may be shown analytically to be repelled from, rather than attracted to, a nearby saddle point, but the situation may be different for more realistic uses of gradient descent.

Gradient descent is designed to move "downhill" and is not explicitly designed to seek a critical point. Newton's method, however, is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point. The proliferation of saddle points in high dimensional spaces presumably explains why second-order methods have not succeeded in replacing gradient descent for neural network training. Dauphin *et al.* (2014) introduced a *saddle-free Newton method* for second-order optimization and showed that it improves significantly over the traditional version. Second-order methods remain difficult to scale to large neural networks, but this saddle-free approach holds promise if it could be scaled.

There are other kinds of points with zero gradient besides minima and saddle points. There are also maxima, which are much like saddle points from the perspective of optimization—many algorithms are not attracted to them, but

unmodified Newton's method is. Maxima become exponentially rare in high dimensional space, just like minima do.

There may also be wide, flat regions of constant value. In these locations, the gradient and also the Hessian are all zero. Such degenerate locations pose major problems for all numerical optimization algorithms. In a convex problem, a wide, flat region must consist entirely of global minima, but in a general optimization problem, such a region could correspond to a high value of the objective function.

### 8.2.5 Cliffs and Exploding Gradients

Whereas the issues of ill-conditioning and saddle points discussed in the previous sections arise because of the second-order structure of the objective function (as a function of the parameters), neural networks involve stronger non-linearities which do not fit well with this picture. In particular, the second-order Taylor series approximation of the objective function yields a symmetric view of the landscape around the minimum, oriented according to the axes defined by the principal eigenvectors of the Hessian matrix. See Fig. 8.7 for an illustration of the Hessian matrix eigenvector directions around a local minimum. Second-order methods and momentum or gradient-averaging methods introduced in Sec. 8.5 are able to reduce the difficulty due to ill-conditioning by increasing the size of the steps in the low-curvature directions (the "valley", in Figure 8.1) and decreasing the size of the steps in the high-curvature directions (the steep sides of the valley, in the figure).

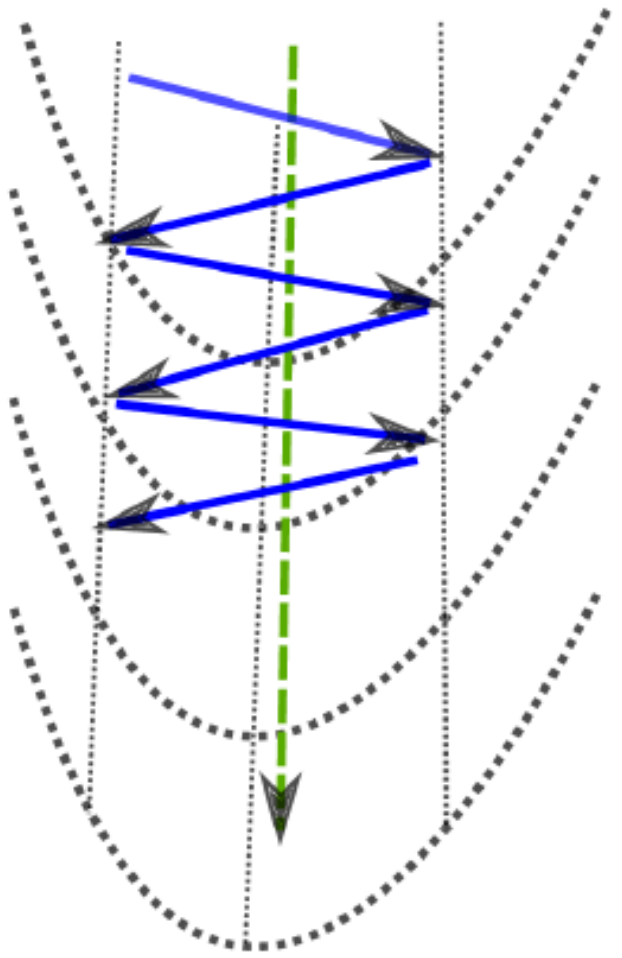

Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and other directions have a low curvature, corresponding to the smooth slope of the valley (going down, dashed arrow). Most second-order methods, as well as momentum or gradient averaging methods are meant to address that problem, by increasing the step size in the direction of the valley (where it is most paying in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations (blue full arrows). The objective is to smoothly go down, staying at the bottom of the valley (green dashed arrow).

However, although classical second order methods can help, due to higher order derivatives, the objective function may have a lot more non-linearity, as shown in Figure 8.2. The objective function often does not have the nice symmetrical shapes that the second-order "valley" picture builds in our mind. Instead, there are cliffs where the gradient rises sharply. When the parameters approach a cliff region, the gradient update step can move the learner towards a very bad configuration, ruining much of the progress made during recent training iterations.

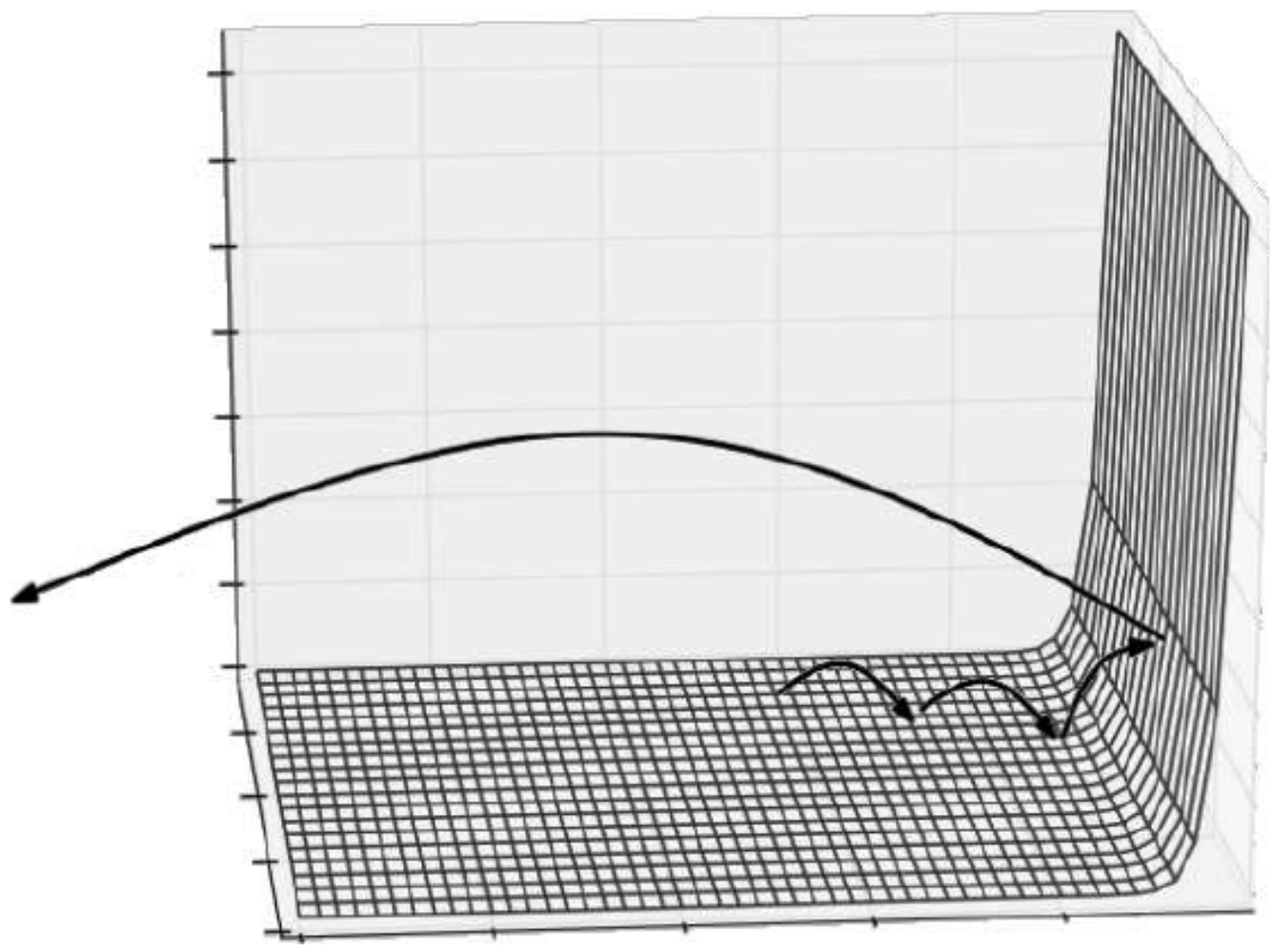

Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly ruining a lot of the optimization work that had been done. Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

As illustrated in Figure 8.3, the cliff can be dangerous whether we approach it from above or from below, but fortunately there are some fairly straightforward heuristics that allow one to avoid its most serious consequences. The basic idea is to limit the size of the jumps that one would make. Indeed, one should keep in mind that when we use the gradient to make an update of the parameters, we are relying on the assumption of *infinitesimal moves*. There is no guarantee that making a finite step of the parameters $\boldsymbol{\theta}$ in the direction of the gradient will yield an improvement. The only thing that is guaranteed is that a *small enough* step in that direction will be helpful. As we can see from Figure 8.3, in the presence of a cliff (and in general in the presence of very large gradients), the decrease in the objective function expected from going in the direction of the gradient is only valid for a very small step. In fact, because the objective function is usually bounded in its actual value (within a finite domain), when the gradient is large at $\boldsymbol{\theta}$, it typically only remains like this (especially, keeping its sign) in a small region around $\boldsymbol{\theta}$. Otherwise, the value of the objective function would have to change a lot: if the slope was consistently large in some direction as we would move in that direction, we would be able to decrease the objective function value by a

very large amount by following it, simply because the total change is the integral over some path of the directional derivatives along that path.

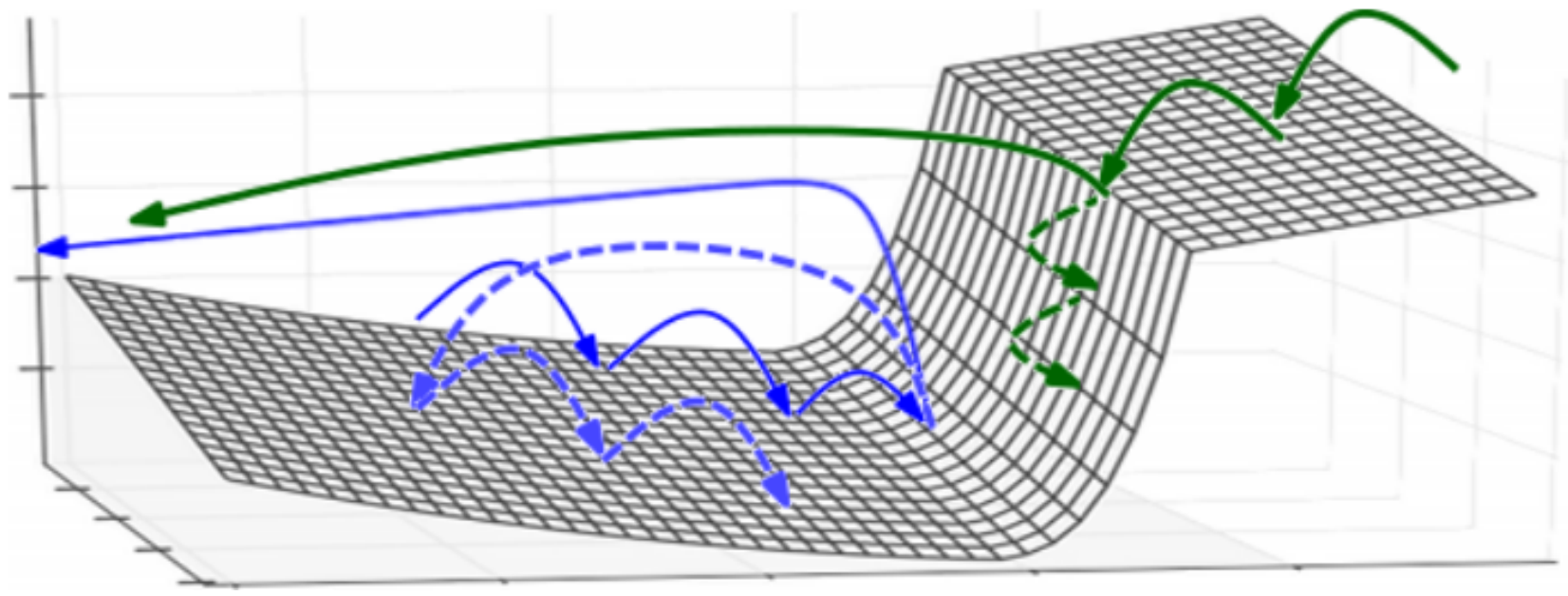

Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

The gradient clipping heuristics are described in more detail in Sec. 10.7.7. The basic idea is to bound the magnitude of the update step, i.e., not trust the gradient too much when it is very large in magnitude. The context in which such cliffs have been shown to arise in particular is that of recurrent neural networks, when considering long sequences, as discussed in the next section.

### 8.2.6 An Introduction to Learning Long-Term Dependencies

Parametrized dynamical systems such as recurrent neural networks (Chapter 10) face a particular optimization problem which is different but related to that of training very deep networks. We introduce this issue here and refer to reader to Sec. 10.7 for a deeper treatment along with a discussion of approaches that have been proposed to reduce this difficulty.

The simplest explanation of the problem, which is shared among very deep nets and recurrent nets, is that in both cases the final output is the composition of a large number of non-linear transformations. Even though each of these non-linear stages may be relatively smooth (e.g. the composition of an affine transformation with a hyperbolic tangent or sigmoid), their composition is going to be much

"more non-linear", in the sense that derivatives through the whole composition will tend to be either very small or very large, with many strong variations. This arises simply because the Jacobian (matrix of derivatives) of a composition is the product of the Jacobians of each stage. If

$$f = f_T \circ f_{T-1} \circ \ldots, f_2 \circ f_1$$

then the Jacobian matrix of derivatives of $f(x)$ with respect to its input vector $\boldsymbol{x}$ is the product

$$f' = f'_T f'_{T-1} \ldots, f'_2 f_1 \tag{8.5}$$

where

$$f' = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

and

$$f'_t = \frac{\partial f_t(\boldsymbol{a}_t)}{\partial \boldsymbol{a}_t},$$

where $\boldsymbol{a}_t = f_{t-1}(f_{t-2}(\ldots, f_2(f_1(\boldsymbol{x}))))$. When taking derivatives through compositions, we obtain matrix multiplication of the corresponding Jacobians. This is illustrated in Figure 8.4.

Figure 8.4: When composing many non-linearities (like the activation non-linearity in a deep or recurrent neural network), the result is highly non-linear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many ups and downs (not shown here).

In the scalar case, we can imagine that multiplying many numbers together tends to be either very large or very small. In the special case where all the numbers in the product have the same value $\alpha$, this is obvious, since $\alpha^T$ goes to 0 if $\alpha < 1$ and goes to $\infty$ if $\alpha > 1$, as $T$ increases. The more general case of non-identical numbers be understood by taking the logarithm of these numbers, considering them to be random, and computing the variance of the sum of these logarithms. Clearly, although some cancellation can happen, the variance grows with $T$, and in fact if those numbers are independent, the variance grows linearly with $T$, i.e., the size of the sum (which is the standard deviation) grows as $\sqrt{T}$, which means that the product grows roughly as $e^T$ (consider the variance of log-normal variate $X$ if $\log X$ is normal with mean 0 and variance $T$).

It would be interesting to push this analysis to the case of multiplying square matrices instead of multiplying numbers, but one might expect qualitatively similar conclusions, i.e., the size of the product somehow grows with the number of matrices, and that it grows exponentially. In the case of matrices, one can get a new form of cancellation due to leading eigenvectors being well aligned or not. The product of matrices will blow up only if, among their leading eigenvectors with eigenvalue greater than 1, there is enough "in common" (in the sense of the appropriate dot products of leading eigenvectors of one matrix and another).

However, this analysis was for the case where these numbers are independent. In the case of an ordinary recurrent neural network (developed in more detail in Chapter 10), these Jacobian matrices are highly related to each other. Each layer-wise Jacobian is actually the product of two matrices: (a) the recurrent matrix $\boldsymbol{W}$ and (b) the diagonal matrix whose entries are the derivatives of the non-linearities associated with the hidden units, which vary depending on the time step. This makes it likely that successive Jacobians have similar eigenvectors, making the product of these Jacobians explode or vanish even faster.

The consequence of the exponential convergence of these products of Jacobians towards either very small or very large values is that it makes the learning of *long-term dependencies* particularly difficult, as we explain below and was independently introduced in Hochreiter (1991) and Bengio *et al.* (1993, 1994) for the first time.

Consider a fairly general parametrized dynamical system (which includes classical recurrent networks as a special case, as well as all their known variants), processing a sequence of inputs, $x_1, \ldots, x_t, \ldots$, involving iterating over the transition operator:

$$s_t = F_{\boldsymbol{\theta}}(s_{t-1}, x_t) \tag{8.6}$$

where $s_t$ is called the state of the system and $F_{\boldsymbol{\theta}}$ is the recurrent function that maps the previous state and current input to the next state. The state can be used to produce an output via an output function,

$$o_t = g_{\omega}(s_t), \tag{8.7}$$

and a loss $L_t$ is computed at each time step $t$ as a function of $o_t$ and possibly of some targets $y_t$. Let us consider the gradient of a loss $L_T$ at time $T$ with respect to the parameters $\boldsymbol{\theta}$ of the recurrent function $F_{\boldsymbol{\theta}}$. One particular way to

decompose the gradient $\frac{\partial L}{\partial \boldsymbol{\theta}}$ using the chain rule is the following:

$$\frac{\partial L_T}{\partial \boldsymbol{\theta}} = \sum_{t \leq T} \frac{\partial L_T}{\partial s_t} \frac{\partial s_t}{\partial \boldsymbol{\theta}}$$
$$\frac{\partial L_T}{\partial \boldsymbol{\theta}} = \sum_{t \leq T} \frac{\partial L_T}{\partial s_T} \frac{\partial s_T}{\partial s_t} \frac{\partial F_{\boldsymbol{\theta}}(s_{t-1}, x_t)}{\partial \boldsymbol{\theta}} \tag{8.8}$$

where the last Jacobian matrix only accounts for the immediate effect of $\boldsymbol{\theta}$ as a parameter of $F_{\boldsymbol{\theta}}$ when computing $s_t = F_{\boldsymbol{\theta}}(s_{t-1}, x_t)$, i.e., not taking into account the indirect effect of $\boldsymbol{\theta}$ via $s_{t-1}$ (otherwise there would be double counting and the result would be incorrect). To see that this decomposition is correct, please refer to the notions of gradient computation in a flow graph introduced in Sec. 6.4.3, and note that we can construct a graph in which $\boldsymbol{\theta}$ influences each $s_t$, each of which influences $L_T$ via $s_T$. Now let us note that each Jacobian matrix $\frac{\partial s}{\partial s}$ can be decomposed as follows:

$$\frac{\partial s_T}{\partial s_t} = \frac{\partial s_T}{\partial s_{T-1}} \frac{\partial s_{T-1}}{\partial s_{T-2}} \cdots \frac{\partial s_{t+1}}{\partial s_t} \tag{8.9}$$

which is of the same form as Eq. 8.5 discussed above, i.e., which tends to either vanish or explode.

As a consequence, we see from Eq. 8.8 that $\frac{\partial L}{\partial \boldsymbol{\theta}}$ is a weighted sum of terms over spans $T-t$, *with weights that are exponentially smaller (or larger)* for longer-term dependencies relating the state at $t$ to the state at $T$. As shown in Bengio *et al.* (1994), in order for a recurrent network to *reliably store memories*, the Jacobians $\frac{\partial s}{\partial s_1}$ relating each state to the next must have a determinant that is less than 1 (i.e., yielding to the formation of *attractors* in the corresponding dynamical system). Hence, *when the model is able to capture long-term dependencies it is also in a situation where gradients vanish and long-term dependencies have an exponentially smaller weight than short-term dependencies in the total gradient.* It does not mean that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments in Bengio *et al.* (1994) show that as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful learning rapidly reaching 0 after only 10 or 20 steps in the case of the ordinary recurrent net and stochastic gradient descent (Sec. 8.3.2).

For a deeper treatment of the dynamical systems view of recurrent networks, consider Doya (1993); Bengio *et al.* (1994); Siegelmann and Sontag (1995), with a review in Pascanu *et al.* (2013a). Sec. 10.7 discusses various approaches that

have been proposed to reduce the difficulty of learning long-term dependencies (in some cases allowing one to reach to hundreds of steps), but it remains one of the main challenges in deep learning.

### 8.2.7 Inexact Gradients

Most optimization algorithms are primarily motivated by the case where we have exact knowledge of the gradient or Hessian matrix. In practice, we usually only have a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling best estimates at least insofar as using a minibatch of training examples to compute the gradient.

In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient. These issues mostly arise with the more advanced models in Part III of this book. For example, persistent contrastive divergence gives a technique for approximating the gradient of the intractable log-likelihood of a Boltzmann machine.

Various neural network optimization algorithms are designed to account for these imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

### 8.2.8 Theoretical Limits of Optimization

Several theoretical results show that there are limits on the performance of any optimization algorithm we might design for neural networks (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997). Typically these results have little bearing on the use of neural networks in practice.

Some theoretical results apply only to the case where the units of a neural network output discrete values. However, most neural network units output smoothly increasing values that make optimization via local search feasible. Some theoretical results show that there exist problem classes that are intractable, but it can be difficult to tell whether a particular problem falls into that class. Other results show that finding a solution for a network of a given size is intractable, but in practice we can find a solution easily by using a larger network for which many more parameter settings correspond to an acceptable solution. Moreover, in the context of neural network training, we usually do not care about finding the exact minimum of a function, but only in reducing its value sufficiently to obtain good generalization error. Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.

## 8.3 Optimization Algorithms I: Basic Algorithms

In Sec. 6.4.3, we discussed the backpropagation algorithm (backprop): that is, how to efficiently compute the gradient of the loss with respect to the model parameters. The backpropagation algorithm does *not* specify how we use this gradient to update the weights of the model.

In this section we introduce a number of gradient-based *learning algorithms* that have been proposed to optimize the parameters of deep learning models.

### 8.3.1 Gradient Descent

Gradient descent is the most basic gradient-based algorithm one might apply to train a deep model. The algorithm is also sometimes called *batch gradient descent* or *deterministic gradient descent* in neural network papers because it updates the parameters only after having seen a batch of all the training examples and the gradient is computed exactly and deterministically. This method involves updating the model parameters $\boldsymbol{\theta}$ [1] with a small step in the direction of the gradient of the objective function, i.e., for neural networks that includes the terms for all the training examples as well as any regularization terms. For the case of supervised learning with data pairs $[\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}]$ we have:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}), \tag{8.10}$$

where $\epsilon$ is the *learning rate*, an optimization hyperparameter that controls the size of the step the the parameters take in the direction of the gradient. Following the gradient in this way is only guaranteed to reduce the loss if $\epsilon$ is smaller than a threshold value[2]. Note that the learning rate $\epsilon$ does not need to be decreased towards 0, in the batch gradient case. We do get convergence with a fixed $\epsilon$ because the gradients get smaller and smaller (approaching 0) as we approach the local minimum of a function whose gradients are Lipschitz continuous.

In fact, once the algorithm has reached a convex basin of attraction, convergence is fast, with the magnitude of the difference to the minimum decreasing quickly. If the smallest second derivative in any direction (the smallest eigenvalue of the Hessian) is at least $\mu$ and the largest second derivative is at most $L$, then *linear convergence* is achieved: the *excess error*, or difference between the current value of the objective function and the value at the local minimum, is at least reduced by a fixed factor $(1 - \frac{\mu}{\mathcal{L}})$ after each update. The excess error goes down

[1] in the case of a deep neural network, these parameters would include the weights and biases associated with each layer

[2] greater than 1 , where is the Lipschitz constant or the largest second derivative in any direction, and this statement is true only if the gradient is Lipschitz-continuous

exponentially towards zero, or in other words, the error decreases exponentially fast towards its (locally) minimum value. If there is no lower bound on the smallest second derivative $\mu$, then convergence is sub-linear and the error decreases in $O(1/k)$ after $k$ steps (Bertsekas, 2004). In addition, the number of training iterations to reach a particular error level is proportional to $\frac{\mathcal{L}}{\mu}$. Very slow convergence can thus occur when the Hessian is ill-conditioned ($\mu$ is tiny or 0).

In spite of its impressive convergence properties (when the Hessian is not ill-conditioned), batch gradient descent is rarely used in machine learning because it does not exploit the particular structure of the objective function, which is written as a large sum of generally i.i.d. terms associated with each training example. Exploiting this structure is what allows *stochastic* gradient descent, discussed next, to achieve much faster practical convergence, as analyzed theoretically by Bottou and Bousquet (2008).

### 8.3.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) and its variants are probably the most used optimization algorithm for machine learning in general and for deep learning in particular. It is very similar to (batch) gradient descent except that it uses a stochastic (i.e., noisy) estimator of the gradient to perform its update. With machine learning, this is typically obtained by sampling one or a small subset of $m$ of the training examples and computing their gradient, as shown in Algorithm 8.1. When the examples are i.i.d., it means that the expected value $E[\hat{\boldsymbol{g}}]$ of this estimated gradient (averaging over different draws of the examples used to compute the estimated gradient) equals the true gradient. Thus, the gradient estimator is unbiased:

$$E[\hat{\boldsymbol{g}}] = \boldsymbol{g},$$

where $\boldsymbol{g}$ is the total gradient.

When $m = 1$, Algorithm 8.1 is sometimes called *online gradient descent*. When $m > 1$ but $m$ a fraction of the number of training examples, this algorithm is sometimes called *minibatch* SGD. See Sec. 8.1.3 about minibatch optimization algorithms.

A crucial hyper-parameter that is introduced when one applies SGD is the learning rate ($\eta_k$ in Algorithm 8.1). Whereas ordinary gradient descent can work with a fixed learning rate, it is necessary to allow SGD's learning rate to decrease at an appropriate rate during training, if one wants to converge to a minimum. This is because the SGD gradient estimator introduces a source of noise (the random sampling of $m$ training examples) that does not become 0 even when we arrive at a minimum (whereas the true gradient becomes small and then 0 when we approach and reach a minimum). A sufficient condition to guarantee

Stochastic gradient descent (SGD) update at training iteration $k$

Learning rate $\eta$.
Initial parameter $\boldsymbol{\theta}$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\hat{\boldsymbol{g}} =$
$i = 1$ to $m$
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})/m$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_k - \eta \hat{\boldsymbol{g}}$

convergence is that

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad \text{and}$$
$$\sum_{k=1}^{\infty} \eta_k^2 < \infty. \tag{8.11}$$

For a deeper treatment of SGD, see Bottou (1998), which covers the case when the objective function is not convex in the parameters.

The most important property of SGD and related minibatch or online gradient-based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large, reaching to the online or streaming limit, where each example is only seen once.

Again, let us use $k$ to denote the number of iterations, $\mu$ the smallest eigenvalue of the Hessian and $\mathcal{L}$ its largest eigenvalue. We have seen that batch gradient ex-cess error converges at a rate $O((1-\frac{\mu}{\mathcal{L}})^k)$ in the strongly convex case (in a convex basin of attraction where the smallest second derivatives are lower bounded by $\mu$) and in $O(1/k)$ in the convex case (or $O(1/k^2)$ with accelerated convergence or Nesterov momentum (Nesterov, 1983), discussed below, Sec. 8.3.4). Unfortunately, the linear convergence (the error going down towards its minimum exponentially fast) of the strongly convex case is lost in the stochastic setup. With SGD, error converges in $O(1/\sqrt{k})$ in the convex case and in $O(1/k)$ in the strongly convex case, and these bounds cannot be improved unless extra conditions are assumed.

Because many more stochastic updates can be performed for the price of one deterministic update, stochastic gradient converges initially much faster than deterministic (or batch) gradient descent. On the other hand, after some number

of iterations, deterministic gradient descent (or equivalently, using larger and larger minibatches) will converge to lower values of the objective function, because of its faster rate. However, in actual applications, for the conditions in which large neural networks are trained on large datasets, SGD variants remain the choice of practitioners. Training error can in principle be greatly improved by following SGD by a deterministic gradient-based optimization method, but note that it may be at the cost of worse generalization error. Indeed, generalization error cannot go down faster than the $O(1/k)$ rate, which corresponds to the statistical rate of convergence (in the best possible case, which is the online case, when every example is new), or the Cramér-Rao bound (Cramér, 1946; Rao, 1945). As pointed out by Bottou and Bousquet (2008), this makes it unclear whether it is worthwhile pursuing a faster rate by optimization techniques that would make training error converge at a faster asymptotic rate. Without a faster convergence on the test set, the faster training set convergence is likely to correspond to overfitting.

However, it is not just the asymptotic rate of convergence that matters. The "constants" hidden by the $O()$ notation matter, in this case, and the speed of convergence early in the process also matters. Note that second order methods and adaptive learning rate methods described below can have an important effect on these constants, even though they do not change the asymptotic worst-case asymptotic rate of convergence. Although learning theory usually considers estimation error (variance) and approximation error (bias), one should also consider the effect of finite computational resources (training time), as in Bottou and Bousquet (2008). Although SGD approaches the minimum slowly in an asymptotic sense (when we consider what happens close to the minimum, because of the added noise), it approaches the region of the minimum exponentially fast! This was shown by Nedic and Bertsekas (2000): constant step-size SGD approaches some error level (higher than at the minimum) exponentially fast (multiplying the excess error by a fixed factor after each iteration, in average).

### 8.3.3 Momentum

While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. This is especially true in situations where the gradient is small. When the gradient is consistent across consecutive minibatches, we know that we can afford to take larger steps in this direction.

The method of Momentum Polyak (1964) is designed to accelerate learning, especially in the face of small and consistent gradients. The intuition behind momentum, as the name suggests, is derived from a physical interpretation of the optimization process. Imagine you have a small ball (think of a marble) that represents the current position in parameter space (for our purposes here we can

imagine a 2-D parameter space). Now consider that the ball is on a gentle slope, while the instantaneous force pulling the ball down hill is relatively small, their contributions combine and the downhill velocity of the ball gradually begins to increase over time. The momentum method is designed to inject this kind of downhill acceleration into gradient-based optimization. The effect of momentum is illustrated in Fig. 8.5.

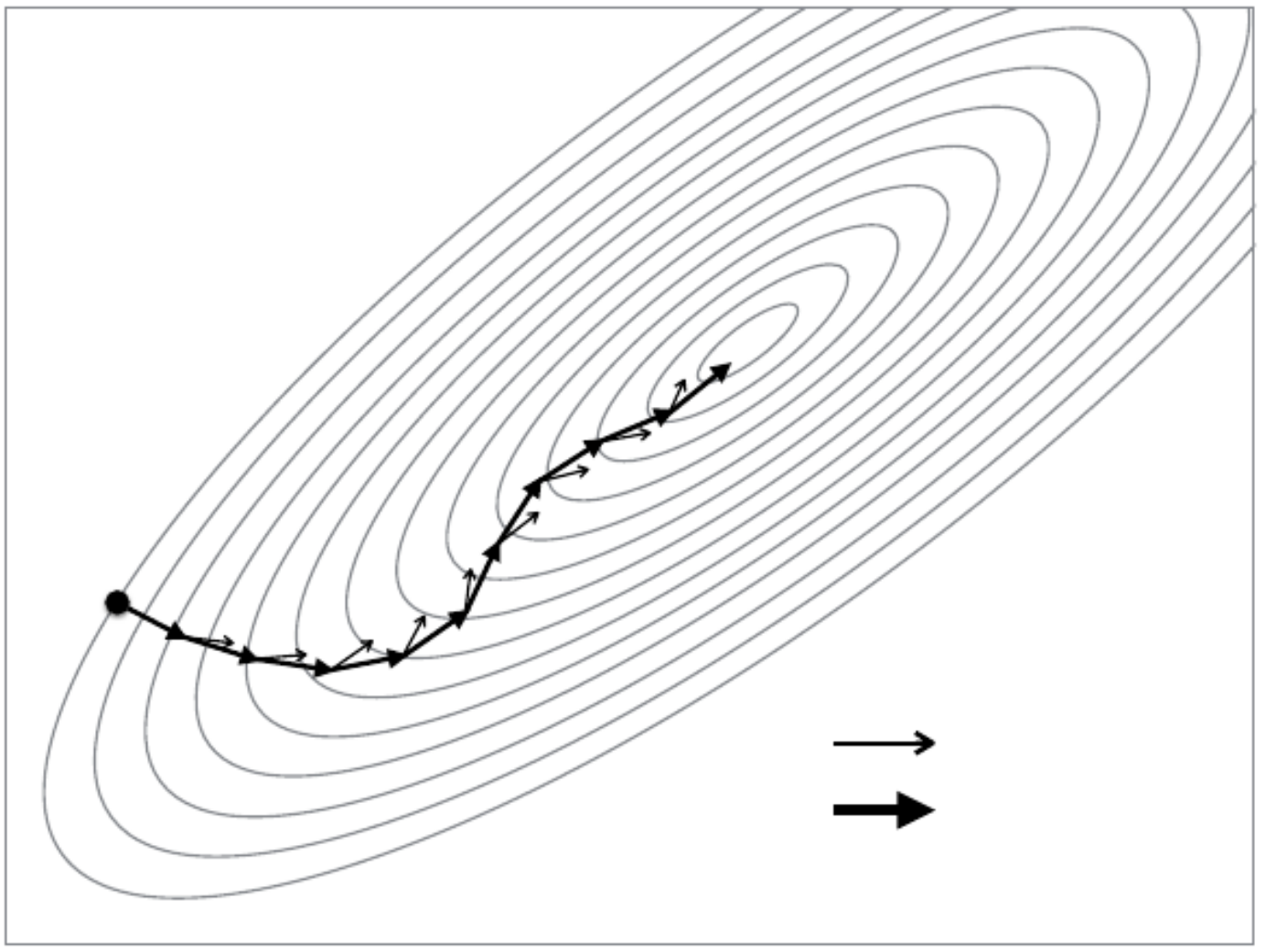

Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.

Formally, we introduce a variable $\boldsymbol{v}$ that plays the role of velocity (or momentum) that accumulates gradient. The update rule is given by:

$$\boldsymbol{v} \leftarrow +\alpha \boldsymbol{v} + \eta \nabla_{\boldsymbol{\theta}} \quad \frac{1}{m} \sum_{t=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}) \Bigg)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left( \frac{1}{n} \sum_{t=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}) \right)$. The larger $\alpha$ is relative to $\eta$, the more previous gradients affect the current direction. The overall learning rate, which in the case of SGD, was a simple hyperparameter, is here a relatively complicated function of $\alpha$ and $\eta$. The SGD+momentum algorithm is given in Algorithm 8.2.

**Stochastic gradient descent (SGD) with momentum**

Learning rate $\eta$, momentum parameter $\alpha$.
Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient estimate: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

### 8.3.4 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov.

$$\boldsymbol{v} \leftarrow +\alpha \boldsymbol{v} + \eta \nabla_{\boldsymbol{\theta}} \left[ \frac{1}{m} \sum_{t=1}^{m} L\Big( \boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(t)} \Big) \right],$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

where the parameters $\alpha$ and $\eta$ play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a *correction factor* to the standard method of momentum. Figure 8.6 illustrates the difference between Nesterov momentum and standard momentum. The complete Nesterov momentum algorithm is presented in Algorithm 8.3.

In the batch gradient case, Nesterov momentum in the convex basin of attraction (not necessarily strictly convex[3]) brings the rate of convergence of the excess error from $O(1/k)$ (after $k$ steps, in the batch gradient case) to $O(1/k^2)$ as shown by Nesterov (1983). In the strictly convex case, convergence goes from from $O(1 - \frac{\mu}{\mathcal{L}})$ (batch gradient) to $O(1 - \sqrt{\frac{\mu}{\mathcal{L}}})$. Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

[3] which means that the Hessian can be ill-conditioned

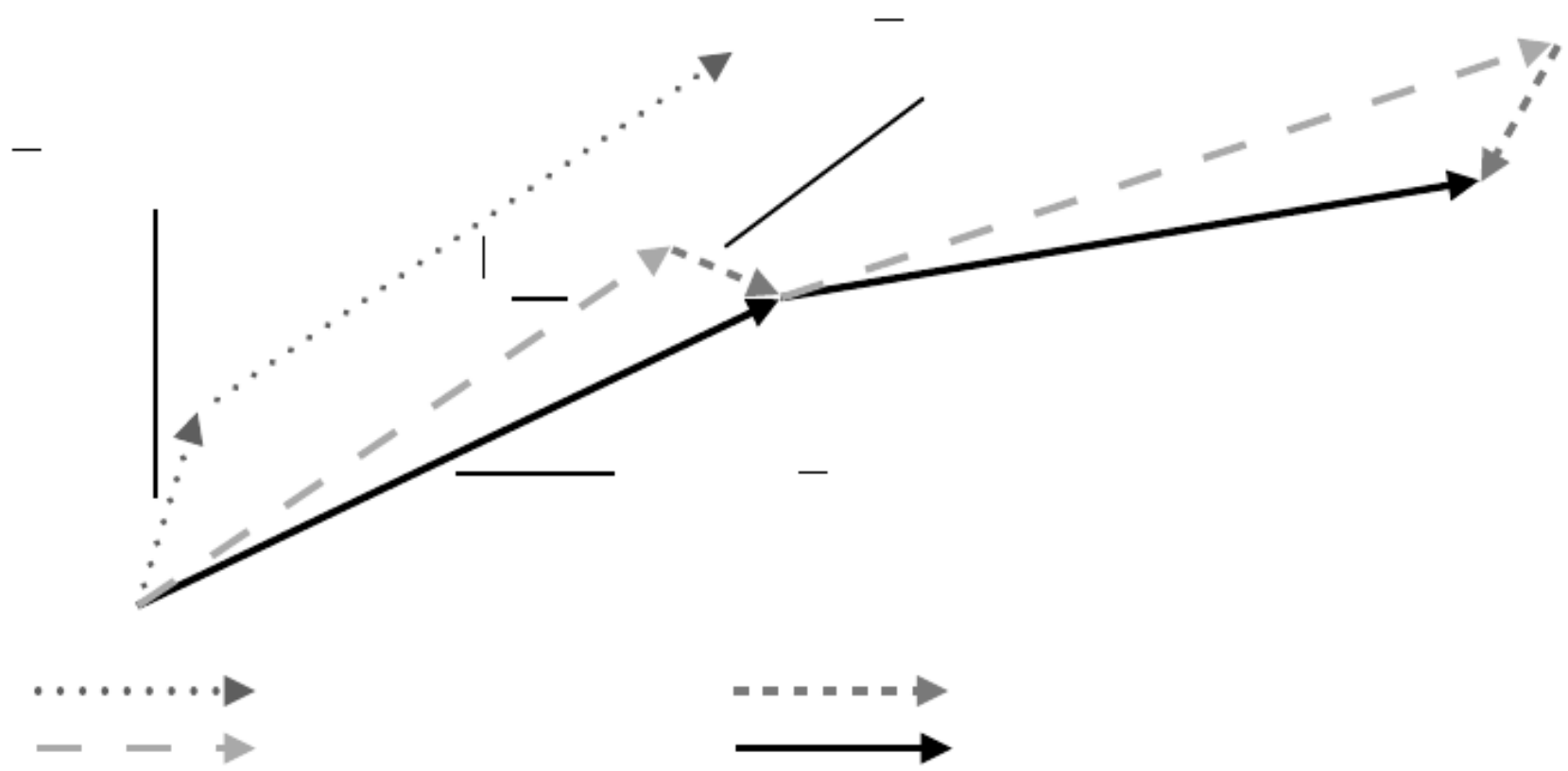

Figure 8.6: An illustration of the difference between Nesterov momentum and standard momentum. Nesterov momentum incorporates the gradient after the velocity is already applied. This figure is derived from a similar image in Geoff Hinton's Coursera lectures.

## 8.4 Optimization Algorithms II: Adaptive Learning Rates

Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. In reality, as we've discussed in Sec.s 4.3 and 8.2, we often have a subset of parameters to which the cost is much more sensitive. These directions in parameter space will limit the size of the SGD learning rate and consequently limit the progress that can be made in the other, less sensitive directions. While the use of momentum can go some way to alleviate these issues, it does so by introducing another hyperparameter that may be just as difficult to set as the original learning rate. In the face of this, it is natural to ask if there is another way. Can learning rates be set automatically and independently for each parameter?

The delta-bar-delta algorithm (Jacobs, 1988) is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase, if it changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini-batch-based) methods have

Stochastic gradient descent (SGD) with Nesterov momentum

Learning rate $\eta$, momentum parameter $\alpha$.
Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\tilde{\boldsymbol{\theta}}} L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

been introduced that adapt the learning rates of model parameters. This section will briefly review a few of these algorithms.

### 8.4.1 AdaGrad

The AdaGrad algorithm, shown in Algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to an accumulated sum of squared partial derivatives over all training iterations. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the convex optimization context, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that — for training deep neural network models — the accumulation of squared gradients *from the beginning of training* results in a premature and excessive decrease in the effective learning rate.

### 8.4.2 RMSprop

The RMSprop algorithm (Hinton, 2012) addresses the deficiency of AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. As we have previously discussed (especially in Sec. 8.2 ), in deep networks, the optimization surface is far from convex. Directions in parameter space with strong partial derivatives early in training may flatten out as training progresses. The introduction of the exponentially weighted moving average allows the effec-

The Adagrad algorithm

Global learning rate $\eta$,
Initial parameter $\boldsymbol{\theta}$
Initialize gradient accumulation variable $\boldsymbol{r} = \quad ,$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g}^2$ (square is applied element-wise)
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\boldsymbol{r}}}\boldsymbol{g}$. % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}_t$

tive learning rates to adapt to the changing local topology of the loss surface.

RMSprop is shown in its standard form in Algorithm 8.5 and combined with Nesterov momentum in Algorithm 8.6. Note that compared to AdaGrad, the use of the moving average does introduce a new hyperparameter, $\rho$, that controls the length scale of the moving average.

Empirically RMSprop has shown to be an effective and practical optimization algorithm for deep neural networks. It is easy to implement and relatively simple to use (i.e. there does not appear to be a great sensitivity to the algorithm's hyperparameters). It is currently one of the "go to" optimization methods being employed routinely by deep learning researchers.

### 8.4.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in Algorithm 8.7. In the context of the earlier algorithms, it is perhaps best seen as a variant on RMSprop+momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSprop is to apply momentum to the rescaled gradients which is not particularly well motivated. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second order moments to account for their initialization at the origin (see Algorithm 8.7) . RMSprop also incorporates an estimate of the (uncentered) second order moment, however it lacks the correction term. Thus, unlike in Adam, the RMSprop second-order moment

**The RMSprop algorithm**

Global learning rate $\eta$, decay rate $\rho$.
Initial parameter $\boldsymbol{\theta}$
Initialize accumulation variables $\boldsymbol{r} = 0$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g}^2$

Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

estimate may have high bias early in training.

### 8.4.4 AdaDelta

AdaDelta is another recently introduced optimization algorithm that seeks to directly address the issues with AdaGrad. AdaDelta starts from an attempt to incorporate some second-order gradient information (see 4.3) into the optimization algorithm. In particular, consider the Newton's step for a single parameter $\boldsymbol{\theta}_j$ on the loss for a single example $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}$.[4]

$$\Delta\theta_j = -\frac{1}{\frac{\partial^2}{\partial \boldsymbol{\theta}^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})} \frac{\partial}{\partial \boldsymbol{\theta}_j} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})$$

$$\frac{1}{\frac{\partial^2}{\partial \boldsymbol{\theta}^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})} = \frac{\Delta\theta_j}{\frac{\partial}{\partial \boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})}$$

Thus, assuming a diagonal Hessian and a Newton update (which we do not have), its inverse could be estimated as the ratio of the increment $\Delta\boldsymbol{\theta}_j$ over the first

[4] Recall, from Chapter 4, that Newton's method — in the single dimension of — can be motivated by looking at the Taylor series expansion of the loss around the current point $^0$: $(( ^{()}; ^0 + \Delta ) ^{()}) \quad (( ^{()}; ^0) ^{()}) + — (( ^{()}; ^0) ^{()}) \Delta + \frac{1}{2} — (( ^{()}; ^0) ^{()}) \Delta ^2$. This expression reaches its extremum, with respect to $\Delta$ when its derivative (w.r.t. $\Delta$ ) is equal to zero. Using this, we can solve for the optimal step $\Delta = \frac{— (( ; ) )}{— (( ; ) )}$.

RMSprop algorithm with Nesterov momentum

Global learning rate $\eta$, decay rate $\rho$, momentum coefficient $\alpha$.
Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} =$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g}^2$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

partial derivative of the loss. AdaDelta separately estimates this ratio as the ratio of RMS estimates, using the square-roots of an exponentially weighted moving average over squares of increments (in the numerator) and partial derivatives (in the denominator). The complete AdaDelta algorithm is shown in Fig. 8.8.

### 8.4.5 Choosing the Right Optimization Algorithm

In this section, we discussed a series of related algorithms that each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. At this point, a natural question is: which algorithm should one choose? Unfortunately, there is currently no consensus on this point. Tom Schaul (2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that this family of algorithms (represented by RMSprop and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam. The choice of which algorithm to use, as this point, seems to depend as much on the users familiarity with the algorithm (for ease of hyperparameter tuning) as it does on any established notion of superior performance.

**The Adam algorithm**

Step-size $\alpha$
Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$
Initial parameter $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \quad , \boldsymbol{r} = \quad ,$
Initialize timestep $t = 0$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$
Get biased first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Get biased second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g}^2$
Compute bias-corrected first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1}$
Compute bias-corrected second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2}$
Compute update: $\Delta\boldsymbol{\theta} = -\alpha \frac{\boldsymbol{s}}{\sqrt{\boldsymbol{r}}+\epsilon}\boldsymbol{g}$ % (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

## 8.5 Optimization Algorithms III: Approximate Second-Order Methods

In this section we discuss the application of second-order methods to the training of deep networks. For simplicity of exposition, the only objective function we will consider is the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}).$$

However the methods we discuss here extend readily to more general objective functions that, for instance, include parameter regularization terms such as those discussed in Chapter 7.

### 8.5.1 Newton's Method

In section 4.3, we discussed the difference between first-order gradient methods and second-order gradient methods. Namely, that second-order gradient methods

**The Adadelta algorithm**

Decay rate $\rho$, constant $\epsilon$
Initial parameter $\boldsymbol{\theta}$
Initialize accumulation variables $\boldsymbol{r} = \quad , \boldsymbol{s} = \quad ,$
Stopping criterion not met
Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
Set $\boldsymbol{g} =$
$i = 1$ to $m$
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g}^2$
Compute update: $\Delta\boldsymbol{\theta} = -\frac{\sqrt{\boldsymbol{s}+\epsilon}}{\sqrt{\boldsymbol{r}+\epsilon}}\boldsymbol{g}$ % (operations applied element-wise)
Accumulate update: $\boldsymbol{s} \leftarrow \rho \boldsymbol{s} + (1-\rho)\left[\Delta\boldsymbol{\theta}\right]^2$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

use information about the partial derivatives of the partial derivatives of the loss - i.e. second-order gradient information.

In multiple dimensions, we may need to examine all of the second derivatives of the function. These derivatives can be collected together into a matrix called the *Hessian matrix*. The Hessian matrix of a function $J(\boldsymbol{\theta})$, denoted $\boldsymbol{H}(J)(\boldsymbol{\theta})$, or simply as $\boldsymbol{H}$, is defined as

$$\boldsymbol{H}(J)(\boldsymbol{\theta})_{i,j} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} J(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.12}$$

Here the Hessian is computed with respect to the parameters $\boldsymbol{\theta}$ of $J$. Equivalently, the Hessian is the Jacobian of the gradient.

Newton's method is an optimization scheme based on using a second-order *Taylor series expansion* to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0). \tag{8.13}$$

If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(J(\boldsymbol{\theta}_0)\right)\right]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) \tag{8.14}$$

Thus for a locally quadratic function (with positive definite $H$), by rescaling the gradient by $H^{-1}$, Newton's method jumps directly to the minimum. If the

objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton's method, given in Algorithm 8.9.

---

Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$.

---

Initial parameter $\boldsymbol{\theta}_0$
stopping criterion not met
Initialize the gradient $\boldsymbol{g} =$
Initialize the gradient $\boldsymbol{H} =$
$i = 1$ to $m$ % loop over the training set.
Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute gradient: $\boldsymbol{H} \leftarrow \boldsymbol{H} + \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian inverse: $\boldsymbol{H}^{-1}$
Compute update: $\Delta\boldsymbol{\theta}_t = \boldsymbol{H}^{-1}\boldsymbol{g}$
Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t$

---

One way to understand how Newton's method works is to consider the Hessian $\boldsymbol{H}$ as a transformation from the Euclidean space where the problem is defined, to a space where gradient optimization is simplified (at least under the quadratic assumption). For real, symmetric $H$, its eigendecomposition is given by $\boldsymbol{H} = \boldsymbol{Q} \quad \boldsymbol{Q}^\top$, where $\boldsymbol{Q}$ is an orthogonal matrix[5].

We can now consider a change of variables from the original parameters $\boldsymbol{\theta}$ to an alternative parametrization $\boldsymbol{\phi}$ via the transformation $\boldsymbol{\phi} = \quad {}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta}$. Our goal is to re-express the quadratic approximation in $\boldsymbol{\phi}$-space. For this, we need to know how the gradient $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$ changes under this transformation. For that purpose, we need to compute $\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0)$, which, by the chain rule of calculus, is given by:

$$\begin{aligned}\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) &= \left[\frac{\partial\boldsymbol{\theta}}{\partial\boldsymbol{\phi}}\right]^\top \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) \\ &= \boldsymbol{Q} \quad {}^{\frac{1}{2}} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) \end{aligned} \tag{8.15}$$

Or equivalently, $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) = \quad {}^{\frac{1}{2}}\boldsymbol{Q}^\top \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0)$. Using the eigendecomposition of $\boldsymbol{H}$ and the change in variables to $\boldsymbol{\phi}$, we can re-express the quadratic approximation

[5] Recall that an orthogonal matrix is square matrix with real-valued elements and whose columns and rows are orthogonal unit vectors, i.e. for orthogonal matrix , , : : $= 1$, : : $= 1$ and for $=$ , : : $= 0$ and : : $= 0$.

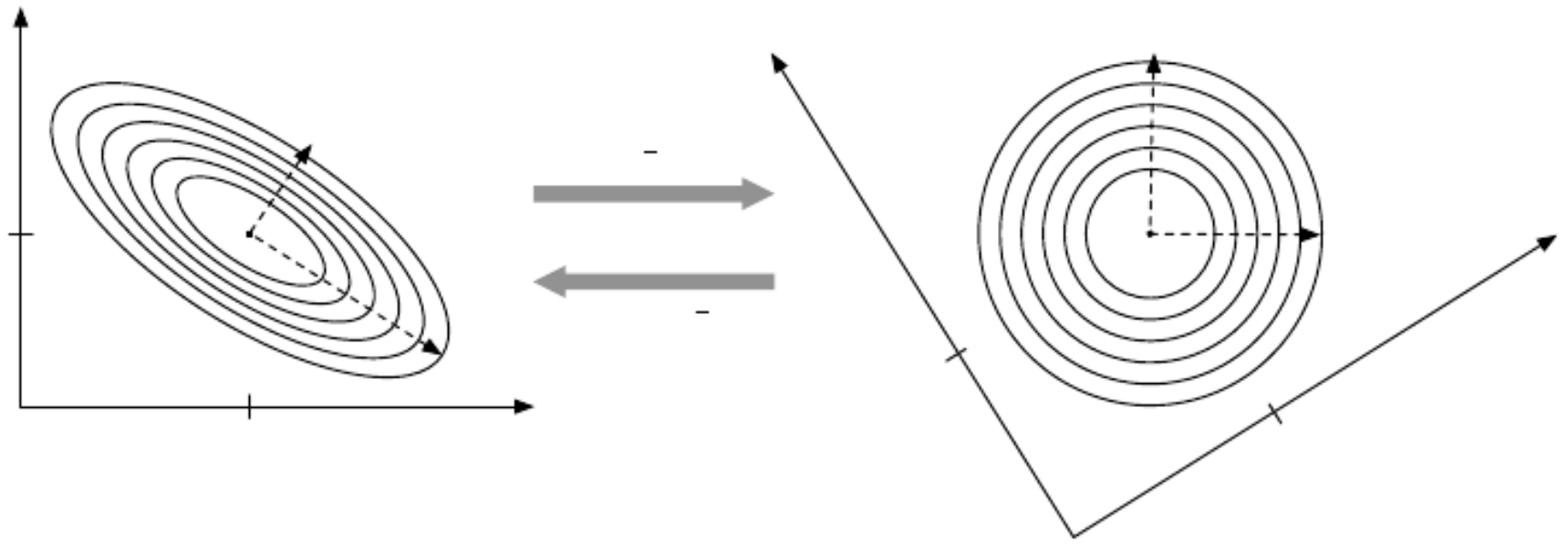

Figure 8.7: Newton's method maps an arbitrary and possibly ill-conditioned (see Sec.4.2) quadratic objective function in parameter space $\boldsymbol{\theta}$ into an alternative parameter space $\boldsymbol{\phi}$ where the quadratic objective function is isometric.

in Eq. 8.13.

$$
\begin{aligned}
f(\boldsymbol{\theta}) &\approx f(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= f(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{Q} \quad \tfrac{1}{2} \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{Q} \quad \boldsymbol{Q}^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) \\
&= f(\boldsymbol{\theta}_0) + \left( \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta} - \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta}_0 \right)^\top \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) \\
&\qquad + \frac{1}{2} \left( \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta} - \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta}_0 \right)^\top \left( \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta} - \quad \tfrac{1}{2} \boldsymbol{Q}^\top \boldsymbol{\theta}_0 \right) \\
&= f(\boldsymbol{\theta}_0) + (\boldsymbol{\phi} - \boldsymbol{\phi}_0)^\top \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\phi} - \boldsymbol{\phi}_0)^\top (\boldsymbol{\phi} - \boldsymbol{\phi}_0) \,. \qquad (8.16)
\end{aligned}
$$

Computing the gradient with respect to $\boldsymbol{\phi}$ and setting this to zero yields the Newton update in $\boldsymbol{\phi}$-space.

$$
\boldsymbol{\phi}^* = \boldsymbol{\phi}_0 - \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0), \qquad (8.17)
$$

That is, in $\boldsymbol{\phi}$-space, the Newton update is transformed into a standard gradient step with unit learning rate. Figure 8.7 illustrates the effect of the transformation from $\boldsymbol{\theta}$-space to $\boldsymbol{\phi}$-space on the newton update.

For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step iterative procedure. First, update or compute the inverse Hessian (i.e. by updating the quadratic approximation). Second, update the parameters according to Eq. 8.14.

In Sec. 8.2.4, we discussed how Newton's method is only appropriate when the local quadratic approximation holds. In deep learning, the surface of the objective

function is typically non-convex with many features, such as saddle points, that are problematic for Newton's method. If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction. This situation can be avoided by regularizing the Hessian. Common regularization strategies include adding a constant, $\alpha$, along the diagonal of the Hessian. The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0).$$

This regularization strategy is used in approximations to Newton's method, such as the Levenberg–Marquardt algorithm (Levenberg, 1944; Marquardt, 1963), and works fairly well as long as the negative eigenvalues of the Hessian are still relatively close to zero. In cases where there are more extreme directions of curvature, the value of $\alpha$ would have to be sufficiently large to offset the negative eigenvalues. However, as $\alpha$ increases in size, the Hessian becomes dominated by the $\alpha \boldsymbol{I}$ diagonal and the direction chosen by Newton's method converges to the standard gradient.

Beyond the challenges created by certain topological features of the objective function, such as saddle points, the application of Newton's method for training large neural networks is limited by the significant computational burden it imposes. The number of elements in the Hessian is squared in the number of parameters, so with $K$ parameters (and for even moderately sized networks the number of parameters $K$ can be in the millions), Newton's method would require the inversion of a $K \times K$ matrix — with computational complexity of $O(K^3)$. Also, since the parameters will change with every update, the inverse Hessian has to be computed *at every training iteration.* As a consequence, only networks with very small number of parameters can be practically trained via Newton's method. In the remainder of this section, we will discuss alternatives that attempt to gain some of the advantages of Newton's method while side-stepping the computational hurdles.

### 8.5.2 Conjugate Gradients

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending *conjugate directions.* The inspiration for this approach follows from a careful study of the weakness of the method of steepest descent (see Sec. 4.3 for details.), where line searches are applied iteratively in the direction associated with the gradient. Figure 8.8 illustrates how the method of steepest descent, when applied in a quadratic bowl, progresses in a rather ineffective back-and-forth, zig-zag pattern. This happens because each line search direction, i.e. the gradient, is guaranteed to be orthogonal to the previous line search direction.

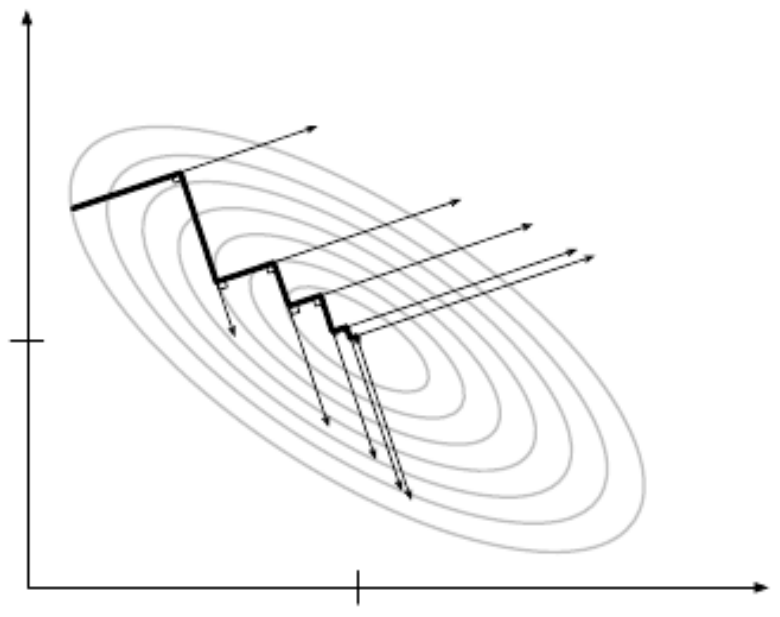

Figure 8.8: The method of steepest descent applied to a quadratic cost surface. The arrows show directions of the gradient. Note the back-and-forth progress made toward the optimum. By definition, at the minimum of the objective along a given direction, the gradient is orthogonal to that direction.

Let the previous search direction be $\boldsymbol{d}_{t-1}$. At the minimum, where the line search terminates, the directional derivative is zero in direction $\boldsymbol{d}_{t-1}$: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \boldsymbol{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $\boldsymbol{d}_t = \nabla_\theta J(\boldsymbol{\theta})$ will have no contribution in the direction $\boldsymbol{d}_{t-1}$. Thus $\boldsymbol{d}_t$ is orthogonal to $\boldsymbol{d}_{t-1}$. This relationship between $\boldsymbol{d}_{t-1}$ and $\boldsymbol{d}_t$ is illustrated in Fig. 8.8 for multiple iterations of steepest descent. As demonstrated in the figure, the choice of orthogonal directions of descent do not preserve the minimum along the previous search directions. This gives rise to the zig-zag pattern of progress, where by descending to the minimum in the current gradient direction, we must re-minimize the objective in the previous gradient direction. Thus, by following the gradient at the end of each line search we are, in a sense, undoing progress we've already made in the direction of the previous line search. The method of conjugate gradients seeks to address this problem with steepest descent.

In the method of conjugate gradients, we seek to find a search direction that is *conjugate* to the previous line search direction, i.e. it will not undo progress made in that direction. At training iteration $t$, the next search direction $\boldsymbol{d}_t$ takes the form:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1} \tag{8.18}$$

were $\beta_t$ is a coefficient whose magnitude controls how much of the direction, $\boldsymbol{d}_{t-1}$, we should add back to the current search direction.

Two directions, $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$, are defined as conjugate if $\boldsymbol{d}_t^\top \boldsymbol{H}(J) \boldsymbol{d}_{t-1} = 0$,

that is, if they are orthogonal in the $\phi$-space defined above in Sec. 8.5.1:

$$\begin{aligned} \boldsymbol{d}_t^\top \boldsymbol{H} \boldsymbol{d}_{t-1} &= 0 \\ \boldsymbol{d}_t^\top \boldsymbol{Q} \ \ \boldsymbol{Q}^\top \boldsymbol{d}_{t-1} &= 0 \\ \left( \ \ \tfrac{1}{2}\boldsymbol{Q}^\top \boldsymbol{d}_{t-1} \right)^\top \left( \ \ \tfrac{1}{2}\boldsymbol{Q}^\top \boldsymbol{d}_{t-1} \right) &= 0 \\ \boldsymbol{d}_t^{(\phi)\top} \boldsymbol{d}_{t-1}^{(\phi)} &= 0, \end{aligned}$$

where we have defined $\boldsymbol{d}_{t-1}^{(\phi)} = \ \ \tfrac{1}{2}\boldsymbol{Q}^\top \boldsymbol{d}_t$ as the direction $\boldsymbol{d}_{t-1}$ transformed to $\phi$-space. The method of conjugate gradients can be interpreted as the application of the method of steepest descent in $\phi$-space.

We motivated the method of conjugate gradients by arguing that it was more computationally viable for large problems that Newton's method. However, so far, we have only shown that the conjugate directions can be computed using the eigendecomposition of the Hessian, $\boldsymbol{H}$. Computing the Hessian, its inverse or its decomposition are exactly the calculations we are hoping to avoid. Can we calculate the conjugate directions without resorting to these calculations? Fortunately the answer to that is yes.

Two popular methods for computing the $\beta_t$ are:

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.19}$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.20}$$

As illustrated in Fig. 8.9, for a quadratic surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude, i.e., we stay at the minimum along the previous directions. As a consequence, in a $k$-dimensional parameter space, conjugate gradients only requires $k$ line searches to achieve the minimum. The conjugate gradient algorithm is given in Algorithm 8.10.

So far we've discussed the method of conjugate gradients as its applied to quadratic objective functions. Of course, our primary interest in this chapter is to explore optimization methods for training neural networks and other related deep learning models where correspondiing objective function is far from quadratic. Perhaps surprisingly, the method of conjugate gradients is still applicable in this setting, though with some modification.

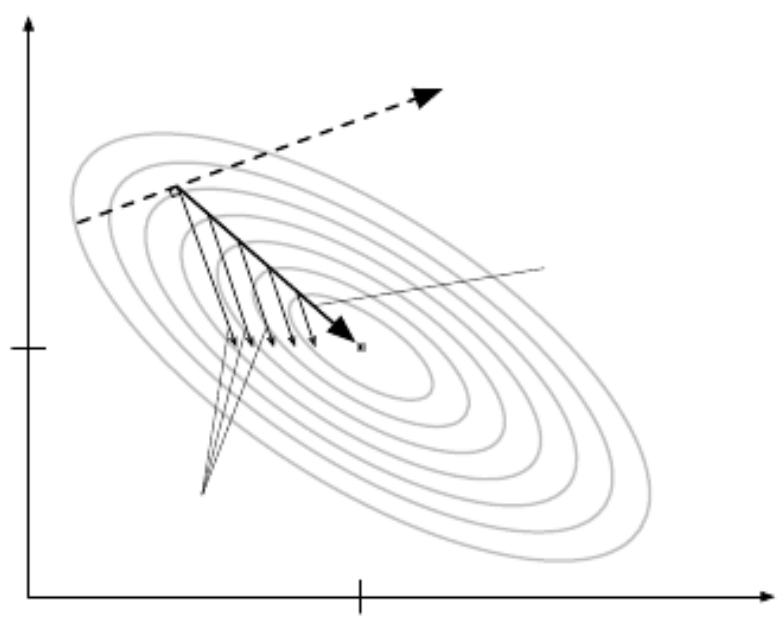

Figure 8.9: The conjugate direction ensures that the contribution to gradient along the original descent direction remains close to zero. As a result, the gradients evaluated along the conjugate direction are all orthogonal to the original descent direction. Figure adapted from (LeCun *et al.*, 1998a).

Without any assurance that the objective is quadratic, the conjugate directions are no longer assured to remain at the minimum of the objective for previous directions. As a result, the so-called *nonlinear conjugate gradients* algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.

Practionners report reasonable results in applications of the nonlinear conjugate gradients algorithm to training neural networks, though it is often beneficial to initialize the optimization with a few iterations of stochastic gradient descent before commencing nonlinear conjugate gradients. Also, while the (nonlinear) conjugate gradients algorithm has traditionally been cast as a batch method, minibatch versions have been used successfully for the training of neural networks Le *et al.* (2011).

### 8.5.3 BFGS

Like the method of conjugate gradients, the BFGS algorithm (the Broyden – Fletcher – Goldfarb – Shanno algorithm) attempts to bring some of the advantages of Newton's method without the computational burden. However, BFGS takes a more direct approach to the approximation of Newton's update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [\boldsymbol{H}(J(\boldsymbol{\theta}_0))]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.21}$$

The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian $\boldsymbol{H}(J)(\boldsymbol{\theta}_0)$. The approach adopted by Quasi-Newton

| Conjugate gradient method |
|---|
| Initial parameters $\boldsymbol{\theta}_0$ |
| Initialize $\boldsymbol{\rho}_0 =$ |
| stopping criterion not met |
| Initialize the gradient $\boldsymbol{g}_t =$ |
| $i = 1$ to $m$ % loop over the training set. |
| Compute gradient: $\boldsymbol{g}_t \leftarrow \boldsymbol{g}_t + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$ |
| backpropagation) |
| Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak — Ribière) |
| Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$ |
| Perform line search to find: $\eta^* = \operatorname{argmin}_\eta \frac{1}{m}\sum_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$ |
| Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$ |

methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix $\boldsymbol{M}_t$ that is iteratively refined by low rank updates to become a better approximation of $\boldsymbol{H}(J)$.

From Newton's update, in Eq. 8.21, we can see that the parameters at learning steps $t$ are related via the secant condition (also known as the quasi-Newton condition):

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\boldsymbol{H}^{-1} \left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t+1}) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)\right) \tag{8.22}$$

The approximation to the Hessian inverse used in the BFGS procedure is constructed so as to satisfy this condition, with $\boldsymbol{M}$ in place of $\boldsymbol{H}^{-1}$. Specifically, $\boldsymbol{M}$ is updated according to:

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + \left(1 + \frac{\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1}\boldsymbol{\phi}}{\boldsymbol{\phi}^\top \boldsymbol{\phi}}\right)\frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{\boldsymbol{\phi}^\top \boldsymbol{\phi}} - \left(\frac{\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1} + \boldsymbol{M}_{t-1}\boldsymbol{\phi}^\top}{\boldsymbol{\phi}^\top \boldsymbol{\phi}}\right), \tag{8.23}$$

where $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$, $\boldsymbol{\phi} = \boldsymbol{g}_t - \boldsymbol{g}_{t-1}$ and $= \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$. Eq. 8.23 shows that the BFGS procedure iteratively refines the approximation of the Hessian with rank one updates — the last three terms are all rank-1. This mean that if $\boldsymbol{\theta} \in \mathbb{R}^n$, then the computational complexity of the update is $O(n^2)$. The derivation of the BFGS approximation is given in many textbooks on optimization, including Luenberger (1984).

Once the inverse Hessian approximation is updated, that is $\boldsymbol{M}_t$ is calculated, the direction of descent $\boldsymbol{\rho}_t$ is determined by $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$. A line search is performed in this direction to determine the size of the step, $\eta^*$, taken in this direction. The final update to the parameters is given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t.$$

**BFGS method**

Initial parameters $\boldsymbol{\theta}_0$
Initialize inverse Hessian $\boldsymbol{M}_0 = \boldsymbol{I}$
stopping criterion not met
Compute gradient: $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ (via batch backpropagation)
Compute $\boldsymbol{\phi} = \boldsymbol{g}_t - \boldsymbol{g}_{t-1}, \quad = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$
Approx $\boldsymbol{H}^{-1}$: $\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + \left(1 + \frac{\boldsymbol{\phi}\ \boldsymbol{M}_{\ 1}\boldsymbol{\phi}}{\boldsymbol{\phi}}\right)\frac{\boldsymbol{\phi}\ \boldsymbol{\phi}}{\boldsymbol{\phi}} - \left(\frac{\boldsymbol{\phi}\ \ \boldsymbol{M}_{\ 1} + \boldsymbol{M}_{\ 1}\boldsymbol{\phi}}{\boldsymbol{\phi}}\right)$
Compute search direction: $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$
Perform line search to find: $\eta^* = \text{argmin}_\eta\ J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$
Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
$*$

The complete BFGS algorithm is presented in Algorithm 8.11.

Like the method of conjugate gradients, the BFGS algorithm iterates a series of line searches with the direction incorporating second-order information. However unlike conjugate gradients, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum along the line. Thus, relative to conjugate gradients, BFGS has the advantage that it can spending less time refining each line search. On the other hand, the BFGS algorithm must store the inverse Hessian matrix, $\boldsymbol{M}$, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically having millions of parameters.

The memory costs of the BFGS algorithm can be significantly decreased by avoiding storing the complete inverse Hessian approximation $\boldsymbol{M}$. Alternatively, by replacing the $\boldsymbol{M}_{t-1}$ in Eq. 8.23 with an identity matrix, the BFGS search direction update formula becomes:

$$\boldsymbol{\rho}_t = -\boldsymbol{g}_t + b \quad + a\boldsymbol{\phi}, \tag{8.24}$$

where the scalars $a$ and $b$ are given by:

$$a = -\left(1 + \frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{{}^\top \boldsymbol{\phi}}\right)\frac{{}^\top g_t}{{}^\top \boldsymbol{\phi}} + \frac{\boldsymbol{\phi}^\top g_t}{{}^\top \boldsymbol{\phi}}$$
$$b = \frac{{}^\top g_t}{{}^\top \boldsymbol{\phi}}$$

with $\boldsymbol{\phi}$ and as defined above. If used with exact line searches, the directions defined by Eq: 8.24 are mutually conjugate. However, unlike the method of conjugate gradients, this procedure remains well behaved when the the minimum of

the line search is reached only approximately. This strategy can be generalized to include more information about the Hessian by storing previous values of $\boldsymbol{\phi}$ and

.

## 8.6 Optimization Algorithms IV: Natural Gradient Methods

Let us take a moment and consider the optimization problem we face when attempting to train a deep learning model. One perspective is that we are trying to adapt the parameters of the model to improve the performance of the network, as measured by the objective function. From this perspective, moving in the direction of steepest descent in parameter space is sensible, as it is in some sense, the direction where we can make the fastest progress. However, as we have seen when considering the Newton's method, the direction of instantaneous steepest descent, i.e. the gradient, in parameter space does not always correspond to the direction of greatest progress when moving in a straight line. One way to view the problem with the direction of steepest descent in parameter space is that it depends on the specific and arbitrary parametrization of the model. For example, replacing $\theta$ by a scaled version of $\theta$ would yield a different optimization trajectory, with gradient descent. We would like our optimization method to be as much as possible invariant to the specific choice of model parametrization. The method of natural gradients is an attempt to specify such a parametrization invariant optimization scheme.

Natural gradient methods are motivated by the insight that when considering directions of descent of the cost function, it is more 'natural' to consider this in the space of the functions represented by the model, rather than in the space of parameters. Every value of $\theta$ corresponds to an input-output function, and the set of functions achievable by a neural network corresponds to a low-dimensional manifold in the space of functions (of dimension no greater than the number of parameters). The natural metric in the space of function is given by how the output changes (at every possible input $\boldsymbol{x}$) as we move on that manifold.

Deep learning approaches to machine learning tasks, such as classification and regression, typically cast the output of the deep learning model as a probability distribution. For example, for a classification task, the model output is cast as the distribution over class labels $\boldsymbol{y}$ given an input $\boldsymbol{x}$. Natural gradient methods aim to find the direction of steepest descent in the space of the probability distribution output by the model. One way to formalize this is to first specify the natural

"finite difference", $\Delta_N$ as

$$\Delta_N \equiv \arg\min_{\Delta\boldsymbol{\theta}} \mathbb{E}_{\hat{p}_{\text{data}}} \left[-\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(x)\right] \tag{8.25}$$
$$\text{s.t.} \mathrm{KL}\left(p_{\boldsymbol{\theta}}(x) \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(x)\right) = \Delta\mathrm{KL}.$$

Under the constraint of unit displacement $\Delta\mathrm{KL}$ of the output distribution under the KL divergence, the natural gradient finds the direction that minimizes the objective function, typically the negative log likelihood.

Taking $\Delta\boldsymbol{\theta} \to 0$, and assuming we have a discrete and bounded domain $\mathcal{X}$ over the probability distribution, we can express the Taylor series expansion of the log probability $\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ around $\boldsymbol{\theta}$ as

$$\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}} \approx \log p_{\boldsymbol{\theta}} + \left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top \left(\nabla^2 \log p_{\boldsymbol{\theta}}\right)\Delta\boldsymbol{\theta}. \tag{8.26}$$

First note that

$$\sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} = \sum_{\mathcal{X}} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} 1 = 0, \tag{8.27}$$

which also makes sense intuitively: when the examples comes from the model distribution, the model is the best possible for fitting them, and the expected log-likelihood gradient is 0. Substituting the expression in Eq. 8.26 into the KL divergence $\mathrm{KL}\left(p_{\boldsymbol{\theta}} \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}\right)$ we have:

$$\begin{aligned}
\mathrm{KL}\left(p_{\boldsymbol{\theta}} \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}\right) &= \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} - \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}} \\
&\approx \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} \\
&\quad - \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \left[\log p_{\boldsymbol{\theta}} + \left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top \left(\nabla^2 \log p_{\boldsymbol{\theta}}\right)\Delta\boldsymbol{\theta}\right] \\
&= -\frac{1}{2}\Delta\boldsymbol{\theta}^\top \mathbb{E}_p \left[\nabla^2 \log p_{\boldsymbol{\theta}}\right]\Delta\boldsymbol{\theta}
\end{aligned}$$

where we have used Eq. 8.27 to cancel the $\left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta}$ term and the quantity $\mathbb{E}_p \left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]$ is the negative expected Hessian matrix of $\log p_{\boldsymbol{\theta}}$, which can be equivalently expressed in a form known as the Fisher information matrix (FIM)

$$\text{FIM}: \mathbb{E}_p \left[\left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \left(\nabla \log p_{\boldsymbol{\theta}}\right).\right] \tag{8.28}$$

To see this equivalence, note that, like in Eq. 8.27,

$$0 = \nabla^2_{\boldsymbol{\theta}} \sum_{\mathcal{X}} p_\theta = \sum_{\mathcal{X}} \nabla_{\boldsymbol{\theta}} \left(p_\theta \nabla_{\boldsymbol{\theta}} \log p_\theta\right) = \sum_{\mathcal{X}} p \left(\nabla_{\boldsymbol{\theta}} \log p_\theta\right)^\top \nabla_{\boldsymbol{\theta}} \log p_\theta + \sum_{\mathcal{X}} p \nabla^2_{\boldsymbol{\theta}} \log p_\theta \tag{8.29}$$

where the first term is the FIM and the second one is the expected second derivative.

Using the Taylor series expansion of $\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ in the log-likelihood objective given in Eq. 8.25 and expressing the constraint as a Lagrangian, we can express the natural gradient objective as:

$$L_N(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = \mathbb{E}_{\hat{p}_{\text{data}}}[-\log p_{\boldsymbol{\theta}}] + \mathbb{E}_{\hat{p}_{\text{data}}}[-\nabla \log p_{\boldsymbol{\theta}}]^\top + \frac{\lambda}{2}\Delta\boldsymbol{\theta}^\top \mathbb{E}_p\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]\Delta\boldsymbol{\theta}. \tag{8.30}$$

As we did in deriving Newton's method, we can ask what value of $\Delta\boldsymbol{\theta}$ minimizes this objective function. For this purpose, we can solve for $\nabla_{\Delta\boldsymbol{\theta}} L_N(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = 0$ and get the natural gradient update equation (with $\Delta\boldsymbol{\theta} = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \left(\mathbb{E}_p\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]\right)^{-1} \mathbb{E}_{\hat{p}_{\text{data}}}[-\nabla \log p_{\boldsymbol{\theta}}]. \tag{8.31}$$

Comparing Newton method's update (Eq. 8.14) to this one, we see that while Newton's method scaled the gradient by the inverse Hessian matrix, the method of natural gradient scales the gradient with the inverse of the Fisher information matrix.

It is worthwhile reflecting on the difference between the natural gradient and Newton's method. First, they are motivated from two quite different perspectives. Newton's method starts by making a quadratic approximation to the objective function and then attempts to jump directly to the minimum of this approximation. On the other hand, the natural gradient approach starts with the goal of finding the direction of steepest descent in the space of probability distributions. The close correspondence shown here has much more to do with the use of the Taylor series approximation, to 2nd-order, done here in an effort to turn the natural gradient into a practical algorithm.

The main difference between Newton's method and natural gradient is that the expectation of second derivatives (or of squared first derivatives) in the natural gradient is taken with respect to the model distribution $p_{\boldsymbol{\theta}}$, while the Hessian used in Newton's method is computed over the dataset (or perhaps a subsample of the dataset, i.e. a minibatch), which implies that the analogous expectation is taken with respect to the data distribution $\hat{p}_{\text{data}}$.

## 8.7 Optimization Strategies and Meta-Algorithms

Many optimization techniques are not exactly algorithms, but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms.

### 8.7.1 Coordinate Descent

In some cases, it may be possible to solve an optimization problem quickly by breaking it into separate pieces. If we minimize $f(\boldsymbol{x})$ with respect to a single variable $x_i$, then minimize it with respect to another variable $x_j$ and so on, we are guaranteed to arrive at a (local) minimum. This practice is known as *coordinate descent*, because we optimize one coordinate at a time. More generally, *block coordinate descent* refers to minimizing with respect to a subset of the variables simultaneously. The term "coordinate descent" is often used to refer to block coordinate descent as well as the strictly individual coordinate descent.

Coordinate descent makes the most sense when the different variables in the optimization problem can be clearly separated into groups that play relatively isolated roles, or when optimization with respect to one group of variables is significantly more efficient than optimization with respect to all of the variables. For example, the objective function most commonly used for sparse coding is not convex. However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters and the code representations. Minimizing the objective function with respect to either one of these sets of variables is a convex problem. Block coordinate descent thus gives us an optimization strategy that allows us to use efficient convex optimization algorithms.

Coordinate descent is not a very good strategy when the value of one variable strongly influences the optimal value of another variable, as in the function $f(\boldsymbol{x}) = (x_1 - x_2)^2 + \alpha\left(x_1^2 + y_1^2\right)$ where $\alpha$ is a positive constant. As $\alpha$ approaches 0, coordinate descent ceases to make any progress at all, while Newton's method could solve the problem in a single step.

### 8.7.2 Initialization Strategies

Some optimization algorithms are not iterative by nature and simply solve for a solution point. Other optimization algorithms are iterative by nature but, when applied to the right class of optimization problems, converge to acceptable solutions in an acceptable amount of time regardless of initialization. Deep learning training algorithms usually do not have this luxury. Training algorithms for deep learning models are usually iterative in nature and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable

cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Most initialization strategies are based on achieving some nice properties when the network is initialized. However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed. A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point.

Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units. If two units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way. Even if the model or training algorithm is capable of using stochasticity to compute different updates for different units (for example, if one trains with dropout) , it is usually best to initialize each unit to compute a different function from all of the other units. This may help to make sure that no input patterns are lost in the null space of forward propagation and no gradient patterns are lost in the null space of backpropagation. This goal of having each unit compute a different function motivates random initialization of the parameters. We could explicitly search for a large set of basis functions that are all mutually different from each other, but this often incurs a noticeable computation cost. For example, if we have at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix, and be guaranteed that each unit computes a very different function from each other unit. Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly. Extra parameters, for example, parameters encoding the conditional variance of a prediction, are usually set to heuristically chosen constants much like the biases are.

We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter a lot, but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both

the outcome of the optimization procedure and on the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or backpropagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Too large of initial weights may, however, result in exploding values during forward propagation or backpropagation. In recurrent networks, large weights can also result in *chaos* (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random). To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step). Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units. These competing factors determine the ideal initial scale of the weights.

The perspectives of regularization and optimization can give very different insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller. The use of an optimization algorithm such as stochastic gradient descent that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or due to triggering some early stopping criterion based on overfitting) expresses a prior that the final parameters should be close to the initial parameters. Recall from Sec. 7.7 that gradient descent with early stopping is equivalent to weight decay for some models. In the general case, gradient descent with early stopping is not the same as weight decay, but does provide a loose analogy for thinking about the effect of initialization. We can think of initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ as being similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. From this point of view, it makes sense to choose $\boldsymbol{\theta}_0$ to be near 0. This prior says that it is more likely that units do not interact with each other than that they do interact. Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. On the other hand, if we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and in pre-specified ways.

Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize the weights of a fully connected layer with $m$ inputs and $n$ outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{n}})$, while Glorot and Bengio

(2010b) suggest using the *normalized initialization*

$$W_{i,j} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}).$$

This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no non-linearities.

Saxe *et al.* (2013) recommend initializing to random orthogonal matrices, so that all singular values are 1. This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without non-linearities.

In order to account for the non-linearity, Saxe *et al.* (2013) recommend rescaling all initial weights by a gain factor $g$. This can offset the effect of the non-linearities on the eigenvalues of the Jacobian, though the interaction between the non-linearities and the eigenvectors of the Jacobian remains poorly characterized and presumably cannot be accounted for by adjusting the gain. Increasing $g$ pushes the network toward the regime where activations increase in norm as they propagate forward through the network and gradients increase in norm as the propagate backward. Sussillo (2014) showed that setting the gain factor correctly is sufficient to train networks as deep as 1,000 layers, without needing to use orthogonal initializations. A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or backpropagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can avoid the vanishing and exploding gradients problem altogether. Feedforward networks are qualitatively different from recurrent networks. Recurrent networks repeatedly use the same weight matrix for forward propagation and repeatedly use its transpose for backpropagation. If we use the same simplification to analyze recurrent nets as is commonly used to analyze feedforward nets, that is, if we assume that the recurrent net consists only of matrix multiplications composed together, then both forward and back-propagation behave very much like the power method, systematically driving the propagated values toward the principle singular vector of the weight matrix.

Unfortunately, these optimal criteria for initial weights often do not lead to optimal performance. This may be for three different reasons. First, we may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network. Second, the properties imposed at initialization may not persist after learning has begun to proceed. Third, the criteria might succeed at improving the speed of optimization but inadvertently

increase generalization error. In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

One drawback to scaling rules like $1/\sqrt{m}$ is that every individual weight becomes extremely small when the layers become large. Martens (2010) introduced an alternative initialization scheme called *sparse initialization* in which each unit is initialized to have exactly $k$ non-zero weights. The idea is to keep the total amount of input to the unit independent from $m$ without making the magnitude of individual weight elements shrink with $m$. This helps to achieve more diversity among the units at initialization. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values. Because it takes a long time for gradient descent to shrink "incorrect" large values, this initialization scheme can cause problems for units such as maxout units that have several filters that must be carefully coordinated with each other.

When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm described in Chapter 11.2.2, such as random search. The choice of whether to use dense or sparse initialization can also be made a hyperparameter. Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data. If the weights are too small, the range of activations across the minibatch will shrink as the activations propagate forward through the network. By repeatedly identifying the first layer with unacceptably small activations and increasing its weights, it is possible to eventually obtain a network with reasonable initial activations throughout. If learning is still too slow at this point, it can be useful to look at the range or standard deviation of the gradients as well as the activations. This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data, rather than on feedback from a trained model on the validation set.

So far we have focused on the initialization of the weights. Fortunately, initialization of other parameters is typically easier.

The approach for setting the biases must be coordinated with the approach for settings the weights. Setting the biases to zero is compatible with most weight initialization schemes. There are a few situations where we may set some biases to non-zero values:

- If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output. To do this, we

assume that the initial weights are small enough that the output of the unit is determined only by the bias. This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set. For example, if the output is a distribution over classes and this distribution is a highly skewed distribution with the marginal probability of class $i$ given by element $c_i$ of some vector $\boldsymbol{c}$, then we can set the bias vector $\boldsymbol{b}$ by solving the equation $\mathrm{softmax}(\boldsymbol{b}) = \boldsymbol{c}$. This applies not only to classifiers but also to models we will encounter in Part III of the book, such as autoencoders and Boltzmann machines. These models have layers whose output should resemble the input data $\boldsymbol{x}$, and it can be very helpful to initialize the biases of such layers to match the marginal distribution over $\boldsymbol{x}$.

- Sometimes we may want to choose the bias to avoid causing too much saturation at initialization. For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization. This approach is not compatible with weight initialization schemes that do not expect strong input from the biases though. For example, it is not recommended for use with random walk initialization (Sussillo, 2014).

- When one unit gates another unit (for example, the forget gate of an LSTM), we may want to set the bias of the gating unit to 1, in order to make the gate initially be open and avoid discarding the gradient through the unit that it gates (Jozefowicz *et al.*, 2015b).

Another common type of parameter is a variance or precision parameter. For example, we can perform linear regression with a conditional variance estimate using the model

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T\boldsymbol{x} + b, 1/\beta)$$

where $\beta$ is a precision parameter. We can usually initialize variance or precision parameters to 1 safely. Another approach is to assume the initial weights are zero, set the biases to produce the correct marginal mean of the output, and set the variance parameters to the marginal variance of the output in the training set.

Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning. A common strategy discussed in Part III of this book is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs. One can also perform supervised training on a related task. Even performing supervised training on an unrelated tasks can sometimes yield an initialization that offers faster convergence than a random initialization. Some of these initialization strategies may yield faster convergence and better generalization because

they encode information about the distribution in the initial parameters of the model. Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.

### 8.7.3 Greedy Supervised Pre-training

Sometimes, directly training a model to solve a specific task can be too ambitious if the model is complex and hard to optimize or if the task is very difficult. It is sometimes more effective to train a simpler model to solve the task, then make the model more complex. It can also be more effective to train the model to solve a simpler task, then move on to confront the final task. These strategies that involve training simple models on simple tasks before confronting the challenge of training the desired model to perform the desired task are collectively known as *pre-training*.

*Greedy* algorithms break a problem into many components, then solve for the optimal version of each component in isolation. Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution. However, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution, and the quality of a greedy solution is often acceptable if not optimal. Greedy algorithms may also be followed by a *fine-tuning* stage in which a joint optimization algorithm searches for an optimal solution to the full problem. Initializing the joint optimization algorithm with a greedy solution can greatly speed it up and improve the quality of the solution it finds.

Pre-training, and especially greedy pre-training, algorithms are ubiquitous in deep learning. In this section, we describe specifically those pre-training algorithms that break supervised learning problems into other simpler supervised learning problems. This approach is known as *greedy supervised pre-training*.

In the original (Bengio *et al.*, 2007b) version of greedy supervised pre-training, each stage consists in a supervised learning training task involving only a subset of the layers in the final neural network. An example of greedy supervised pre-training is illustrated in Figure 8.10, in which each added hidden layer is pre-trained as part of a shallow supervised MLP, taking as input the output of the previously trained hidden layer. Instead of pre-training one layer at a time, Simonyan and Zisserman (2015) pretrain a deep convolutional network (11 weight layers) and then use the first 4 and last 3 layers from this network to initialize even deeper networks (with up to 19 layers of weights). The middle layers are then initialized randomly and all the layers of the very deep network and jointly trained. Another option, explored by Yu *et al.* (2010) is to use the *outputs* of the previously trained MLPs, as well as the raw input, as inputs for each added stage.

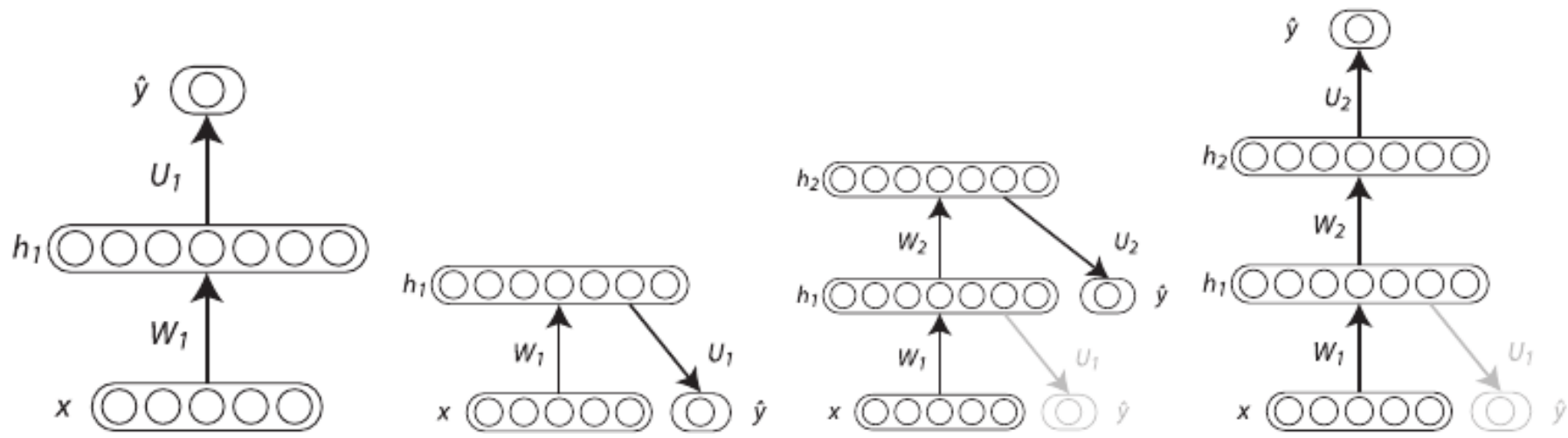


Figure 8.10: Illustration of one form of greedy supervised pre-training (Bengio *et al.*, 2007b). We start by training a sufficiently shallow architecture, like the first one on the left. For the sake of illustration, we redraw that architecture in the second figure from the left, because we are going to only keep the input-to-hidden layer and discard the hidden-to-output layer: we send the output of the first hidden layer as input to another supervised single hidden layer MLP that is trained with the same objective, thus adding a second hidden layer (third figure). This can be repeated for as many layers as desired. We can then redraw the figure (last on the right) to view it as an ordinary feedforward network. To further improve the optimization, jointly fine-tuning all the already pre-trained layers can be done, either only at the end or at each stage of this process.

Why would greedy supervised pre-training help? The hypothesis initially discussed by Bengio *et al.* (2007b) is that it *helps to provide better guidance to the intermediate levels of a deep hierarchy.* In general, pre-training may help both in terms of optimization and in terms of generalization.

An approach related to supervised pre-training extends the idea to the context of transfer learning: Yosinski *et al.* (2014) pre-train a deep convolutional net with 8 layers of weights on a set of tasks (a subset of the 1000 ImageNet object categories) and then initialize a same-size network with the first $k$ layers of the first net. All the layers of the second network (with the upper layers initialized randomly) are then jointly trained towards a different set of tasks (another subset of the 1000 ImageNet object categories), with fewer training examples then for the first set of tasks. See Sec. 16.2 for a deeper treatment of transfer learning with neural networks.

### 8.7.4 Designing Models to Aid Optimization

Most of the model families we study for deep learning are incredibly broad. While most of these model families result in objective functions that are non-convex any time we include at least one hidden layer, there are many other difficulties for optimization besides just non-convexity.

In principle, we could use activation functions that increase and decrease in jagged non-monotonic patterns. However, this would make optimization ex-

tremely difficult. In practice,

.

Most of the advances in neural network learning over the past 30 years have been obtained by changing the model family rather than changing the optimization procedure. Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state of the art neural network applications.

Specifically, modern neural networks reflect a *design choice* to use linear transformations between layers and activation functions that are differentiable and have significant slope. In particular, model innovations like the LSTM, rectified linear units and maxout units have all moved toward using more linear functions than previous models like deep networks based on sigmoidal units. These models have nice properties that make optimization easier. The gradient flows through many layers provided that the Jacobian of the linear transformation has reasonable singular values. Moreover, linear functions consistently increase in a single direction, so even if the model's output is very far from correct, it is clear simply from computing the gradient which direction its output should remove to reduce the loss function. In other words, modern neural nets have been designed so that their *local* gradient information corresponds reasonably well to moving toward a distant solution.

Other model design strategies can help to make optimization easier. For example, linear paths or skip connections between layers reduce the length of the shortest path from the lower layer's parameter to the output, and thus mitigate the vanishing gradient problem (Srivastava *et al.*, 2015). A related idea to skip connections is adding extra copies of the output that are attached to the intermediate hidden layers of the network, as in GoogLeNet (Szegedy *et al.*, 2014a), deeply-supervised nets (Lee *et al.*, 2014) and FitNets (Romero *et al.*, 2015). With GoogLeNet and deeply-supervised nets, these "auxiliary heads" are trained to perform the same task as the primary output as the top of the network in order to insure that the lower layers receive a large gradient. When training is complete the auxiliary heads may be discarded. This is an alternative to the pre-training strategies, which were introduced in the previous section. In this way, one can train jointly all the layers in a single phase but change the objective function and the architecture, so that intermediate layers (especially the lower ones) can get some hints about what they should do, via a shorter path. These hints provide an error signal to lower layers.

### 8.7.5 Continuation Methods and Curriculum Learning

Many optimization algorithms depend strongly on a good choice of initial parameters, for a variety of reasons. Most of our learning algorithms are based on

making small, local moves. Sometimes the correct direction of these local moves can be difficult to compute. We may be able to compute some properties of the objective function, such as its gradient, only approximately, with bias or variance in our estimate of the correct direction. In these cases, local descent may or may not define a reasonably short path to a valid solution, but we are not actually able to follow the local descent path. The objective function may have issues such as poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be very small. In these cases, local descent with steps of size $\epsilon$ may define a reasonably short path to the solution, but we are only able to compute the local descent direction with steps of size $\delta \ll \epsilon$. In these cases, local descent may or may not define a path to the solution, but the path contains many steps, so following the path incurs a high computational cost. Sometimes local information provides us no guide, when the function has a wide flat region, or if we manage to land exactly on a critical point (usually this latter scenario only happens to methods that solve explicitly for critical points, such as Newton's method). In these cases, local descent does not define a path to a solution at all. In other cases, local moves can be too greedy and lead us along a path that moves downhill but away from any solution. This can happen even without local minima. Imagine a mountain standing in a plain, with a deep valley on the south side of the mountain. If we begin our descent from the north face of the mountain, we will begin by traveling north, rather than south, even though the only minimum lies to our south. After descending the mountain, we may become stuck on the plain. If the plain has sufficient slope, we will still waste a lot of time moving around the perimeter of the mountain before finally descending into the valley. Currently, we do not understand which of these problems are most relevant to making neural network optimization difficult, and this is an active area of research. Regardless of which of these problems are most significant, all of them might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we are able to initialize learning within that well-behaved region.

*Continuation methods* are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space. The idea behind continuation methods is to construct a series of objective functions over the same parameters. In order to minimize a cost function $J(\boldsymbol{\theta})$, we will construct new cost functions $\{J^{(0)}, \ldots, J^{(n)}\}$. These cost functions are designed to be increasingly difficult, with $J^{(0)}$ being fairly easy to minimize, and $J^{(n)}$, the most difficult, being $J(\boldsymbol{\theta})$, the true cost function motivating the entire process. When we say that $J^{(i)}$ is easier than $J^{(i+1)}$, we mean that it is well behaved over more of $\boldsymbol{\theta}$ space. A random initialization is more likely to land in the region where local descent can

minimize the cost function successfully because this region is larger. The series of cost functions are designed so that a solution to one is a good initial point of the next. We thus begin by solving an easy problem then refine the solution to solve incrementally harder problems until we arrive at a solution to the true underlying problem.

Traditional continuation methods (predating the use of continuation methods for neural network training) are usually based on smoothing the objective function. See Wu (1997) for an example of such a method and a review of some related methods. Continuation methods are also closely related to simulated annealing, which adds noise to the parameters (Kirkpatrick *et al.*, 1983). Continuation methods have been extremely successful in recent years: see a recent overview of recent literature, especially for AI applications in Mobahi and Fisher III (2015).

33Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima. Specifically, they were designed to reach a global minimum despite the presence of many local minima. To do so, these continuation methods would construct easier cost functions by "blurring" the original cost function. This blurring operation can be done by approximating

$$J(\boldsymbol{\theta})^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\theta}, \boldsymbol{\theta}, \sigma(i)^2)} J(\boldsymbol{\theta}')$$

via sampling. The intuition for this approach is that some non-convex functions become approximately convex when blurred. In many cases, this blurring preserves enough information about the location of a global minimum that we can find the global minimum by solving progressively less blurred versions of the problem. See Fig. 8.11 for an illustration. This approach can break down in three different ways. First, it might successfully define a series of cost functions where the first is convex and the optimum tracks from one function to the next arriving at the global minimum, but it might require so many incremental cost functions that the cost of the entire procedure remains high. NP-hard optimization problems remain NP-hard, even when continuation methods are applicable. The other two ways that continuation methods fail both correspond to the method not being applicable. First, the function might not become convex, no matter how much it is blurred. Consider for example the function $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$. Second, the function may become convex as a result of blurring, but the minimum of this blurred function may track to a local rather than a global minimum of the original cost function.

Though continuation methods were mostly originally designed to deal with the problem of local minima, local minima are no longer believed to be the primary problem for neural network optimization. Fortunately, continuation methods can still help. The easier objective functions introduced by the continuation method can eliminate flat regions, decrease variance in gradient estimates, improve conditioning of the Hessian matrix, or do anything else that will either make local

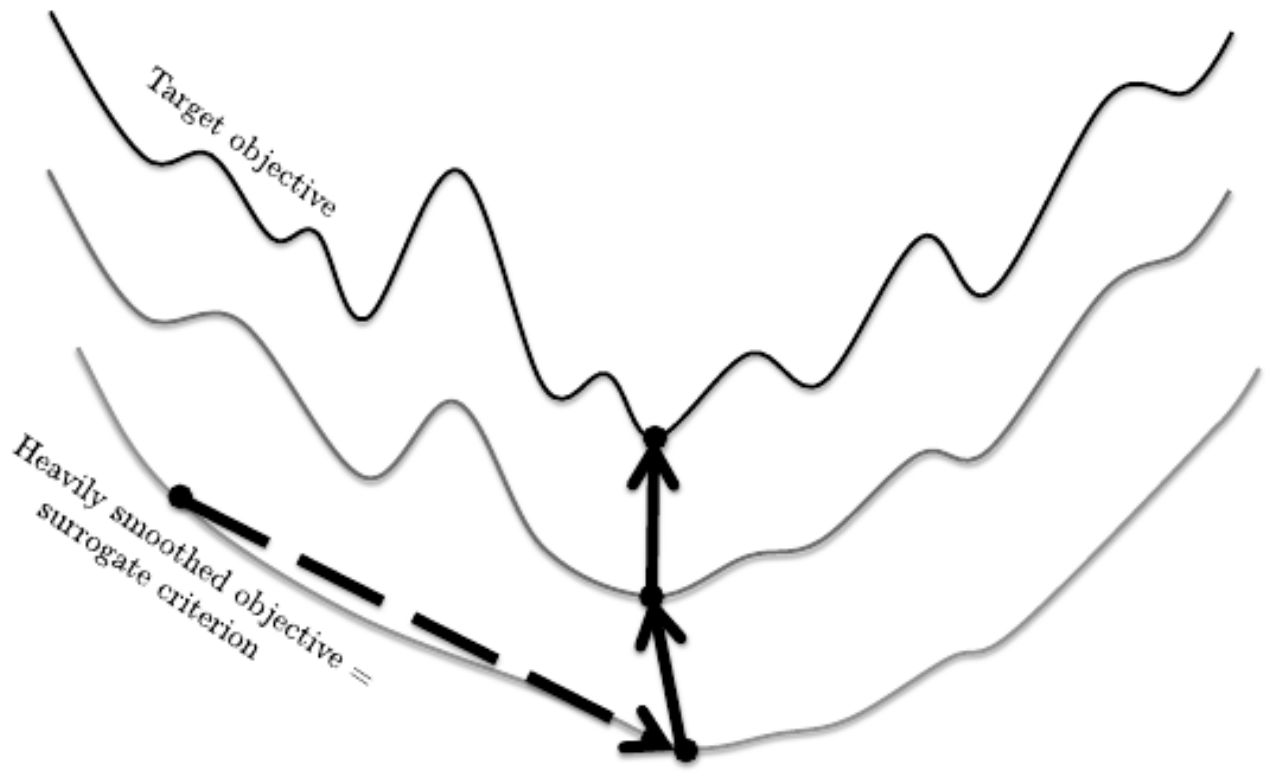


Figure 8.11: Continuation methods start by optimizing a smoothed version of the target objective function (possibly convex), then gradually reduce the amount of smoothing while tracking the local optimum. This approach tends to find better local minima than local descent on the target objective function starting from random initializations.

updates easier to compute or improve the correspondence between local update directions and progress toward a global solution.

Two fairly general approaches to global optimization are *continuation methods* (Wu, 1997), a deterministic approach and *simulated annealing* (Kirkpatrick *et al.*, 1983), a stochastic approach. They both proceed from the intuition that if we sufficiently blur a non-convex objective function (e.g. convolve it with a Gaussian) whose global minima are not at infinite values, then it becomes convex and finding the global optimum of that blurred objective function should be much easier. As illustrated in Figure 8.11, by gradually changing the objective function from a very blurred easy to optimize version to the original crisp and difficult objective function, we are actually likely to find better local minima. In the case of simulated annealing, the blurring occurs because of injecting noise. With injected noise, the state of the system can sometimes go uphill, and thus does not necessarily get stuck in a local minimum. With a lot of noise, the effective objective function (averaged over the noise) is flatter and convex, and if the amount of noise is reduced sufficiently slowly, then one can show convergence to the global minimum. However, the annealing schedule (the rate at which the noise level is decreased, or equivalently the temperature is decreased when you think of the physical annealing analogy) might need to be extremely slow, so an NP-hard optimization problem remains NP-hard.

Bengio *et al.* (2009) observed that an approach called *curriculum learning* or *shaping* can be interpreted as a continuation method. Curriculum learning

is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts. This basic strategy was previously known to accelerate progress in animal training (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009) and machine learning (Solomonoff, 1989; Elman, 1993; Sanger, 1994). Bengio *et al.* (2009) justified this strategy as a continuation method, where earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples (either by assigning their contributions to the cost function larger coefficients, or by sampling them more frequently), and experimentally demonstrated that better results could be obtained by following a curriculum on a large-scale neural language modeling task. Curriculum learning has been successful on a wide range of natural language (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) and computer vision (Kumar *et al.*, 2010; Lee and Grauman, 2011; Supancic and Ramanan, 2013) tasks. Curriculum learning was also verified as being consistent with the way in which humans *teach* (Khan *et al.*, 2011): teachers start by showing easier and more prototypical examples and then help the learner refine the decision surface with the less obvious cases. Curriculum-based strategies are *more effective* for teaching humans than strategies based on uniform sampling of examples, and can also increase the effectiveness of other teaching strategies Basu and Christensen (2013).

Another important contribution to research on curriculum learning arose in the context of training recurrent neural networks to capture long-term dependencies (Zaremba and Sutskever, 2014): it was found that much better results were obtained with a *stochastic curriculum*, in which a random mix of easy and difficult examples is always presented to the learner, but where the average proportion of the more difficult examples (here, those with longer-term dependencies) is gradually increased. Instead, with a deterministic curriculum, no improvement over the baseline (ordinary training from the fully training set) was observed.

# Chapter 9

# Convolutional Networks

*Convolutional networks* (also known as *convolutional neural networks* or *CNNs*) are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications . The name "convolutional neural network" indicates that the network employs a mathematical operation called *convolution*. Convolution is a specialized kind of linear operation. **Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.**

In this chapter, we will first describe what convolution is. Next, we will explain the motivation behind using convolution in a neural network. We will then describe an operation called *pooling*, which almost all convolutional networks employ. Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields such as engineering or pure mathematics. We will describe several variants on the convolution function that are widely used in practice for neural networks. We will also show how convolution may be applied to many kinds of data, with different numbers of dimensions. We then discuss means of making convolution more efficient. Convolutional networks standard out as an example of neuroscientific principles influencing deep learning. We will discuss these neuroscientific principles, then conclude with comments about the role convolutional networks have played in the history of deep learning.

## 9.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, let's start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time $t$. Both $x$ and $t$ are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where $a$ is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function $s$ providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called *convolution*. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In our example, $w$ needs to be a valid probability density function, or the output is not a weighted average. Also, $w$ needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function $x$) to the convolution is often referred to as the *input* and the second argument (in this example, the function $w$) as the *kernel*. The output is sometimes referred to as the *feature map*.

In our example, the idea of a laser sensor that can provide measurements at every instant in time is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides one measurement once per second. $t$ can then take on only integer values. If we now assume that $x$ and $w$ are defined only on integer $t$, we can

define the discrete convolution:

$$s[t] = (x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t-a]$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of learn-able parameters. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image $I$ as our input, we probably also want to use a two-dimensional kernel $K$:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n]$$

Note that convolution is commutative, meaning we can equivalently write:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n]$$

Usually the latter view is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of $m$ and $n$.

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the *cross-correlation*, which is the same as convolution but without flipping the kernel:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n]$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we will follow this convention of calling both operations convolution, and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant.

See Fig. 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.

Discrete convolution can be viewed as multiplication by a matrix. However, the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be

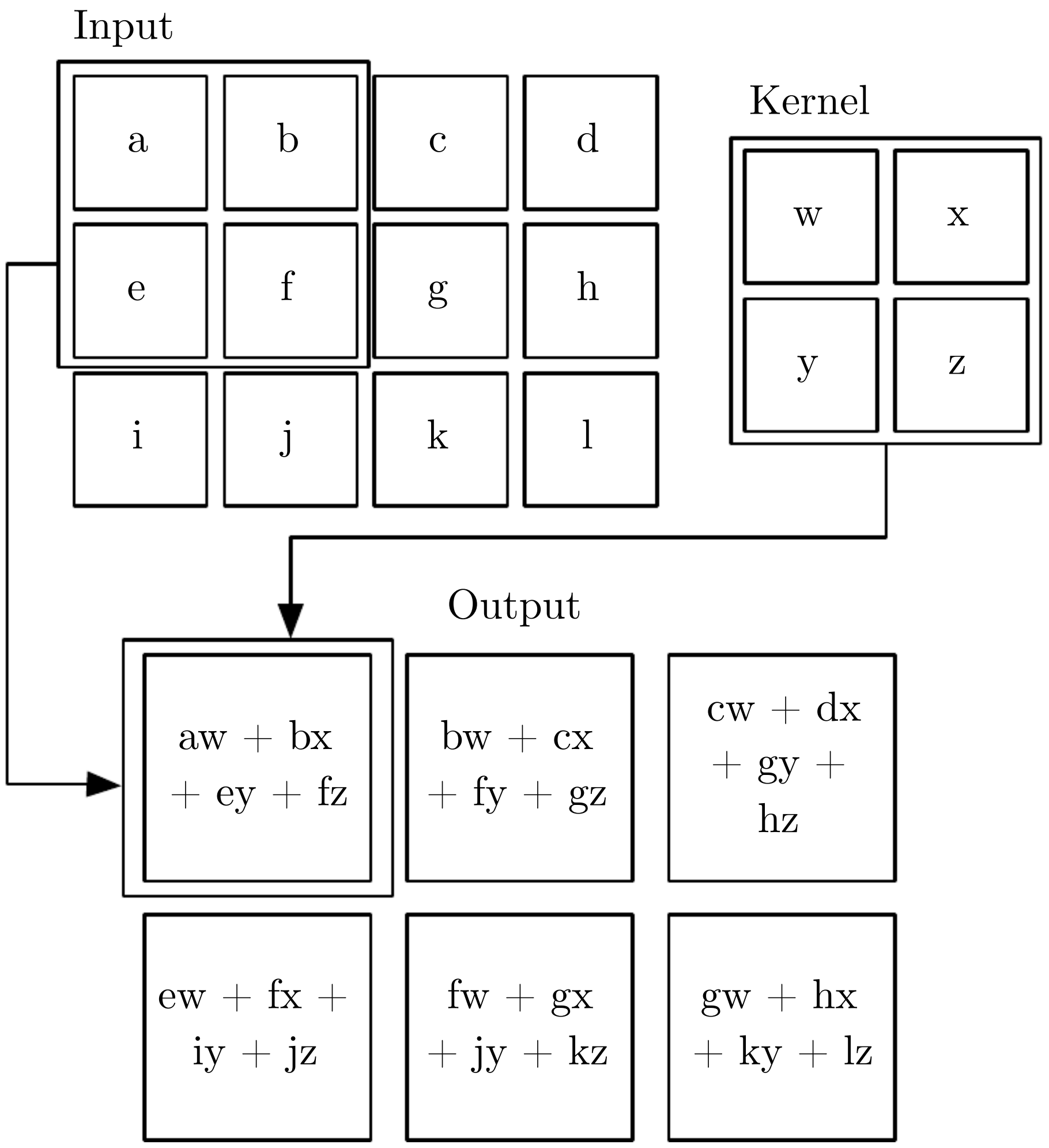


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

equal to the row above shifted by one element. This is known as a *Toeplitz matrix*. In two dimensions, a *doubly block circulant matrix* corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Viewing convolution as matrix multiplication usually does not help to implement convolution operations, but it is useful for understanding and designing neural networks. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network. Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

## 9.2 Motivation

Convolution leverages three important ideas that can help improve a machine learning system: *sparse interactions*, *parameter sharing* and *equivariant representations*. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

Traditional neural network layers use a matrix multiplication to describe the interaction between each input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have *sparse interactions* (also referred to as *sparse connectivity* or *sparse weights*). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are $m$ inputs and $n$ outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to $k$, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping $k$ several orders of magnitude smaller than $m$. For graphical demonstrations of sparse connectivity, see Fig. 9.2 and Fig. 9.3. In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in

Fig. 9.4. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

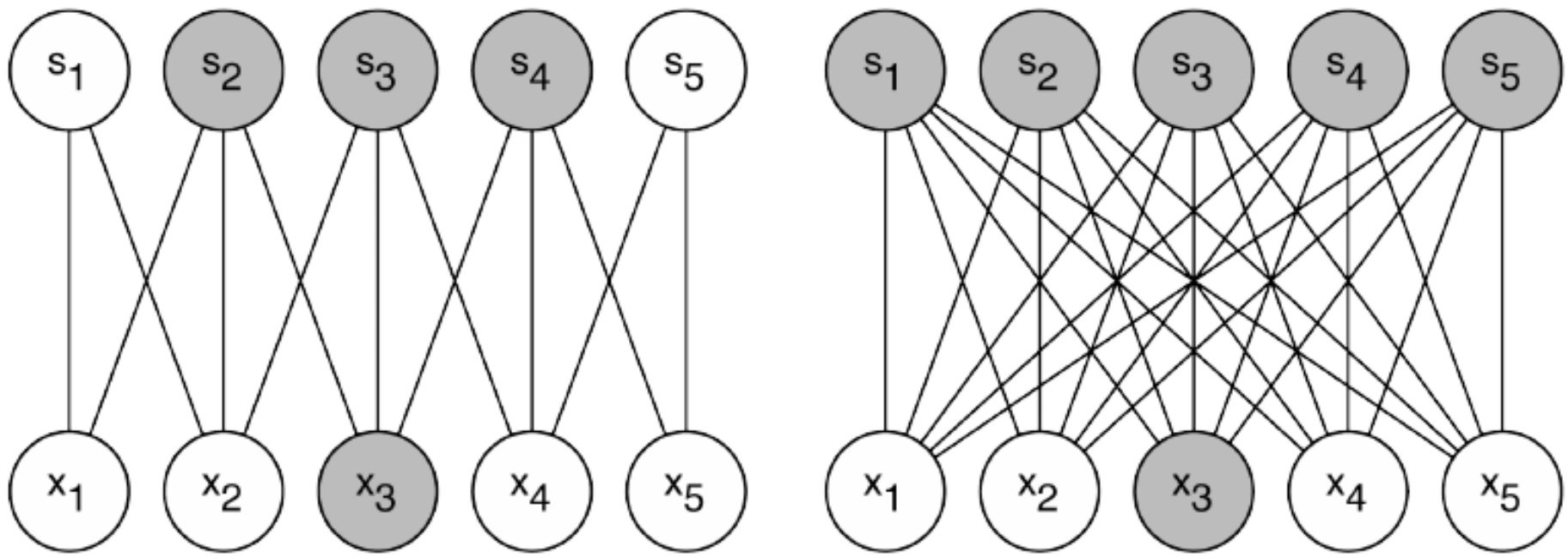


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, $X_3$, and also highlight the output units in **S** that are affected by this unit. *(Left)* When **S** is formed by convolution with a kernel of width 3, only three outputs are affected by $X_3$. *(Right)* When **S** is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by $X_3$.

*Parameter sharing* refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has *tied weights*, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation–it is still $O(k \times n)$–but it does further reduce the storage requirements of the model to $k$ parameters. Recall that $k$ is usual several orders of magnitude less than $m$. Since $m$ and $n$ are usually roughly the same size, $k$ is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see Fig. 9.5.

As an example of both of these first two principles in action, Fig. 9.6 shows how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

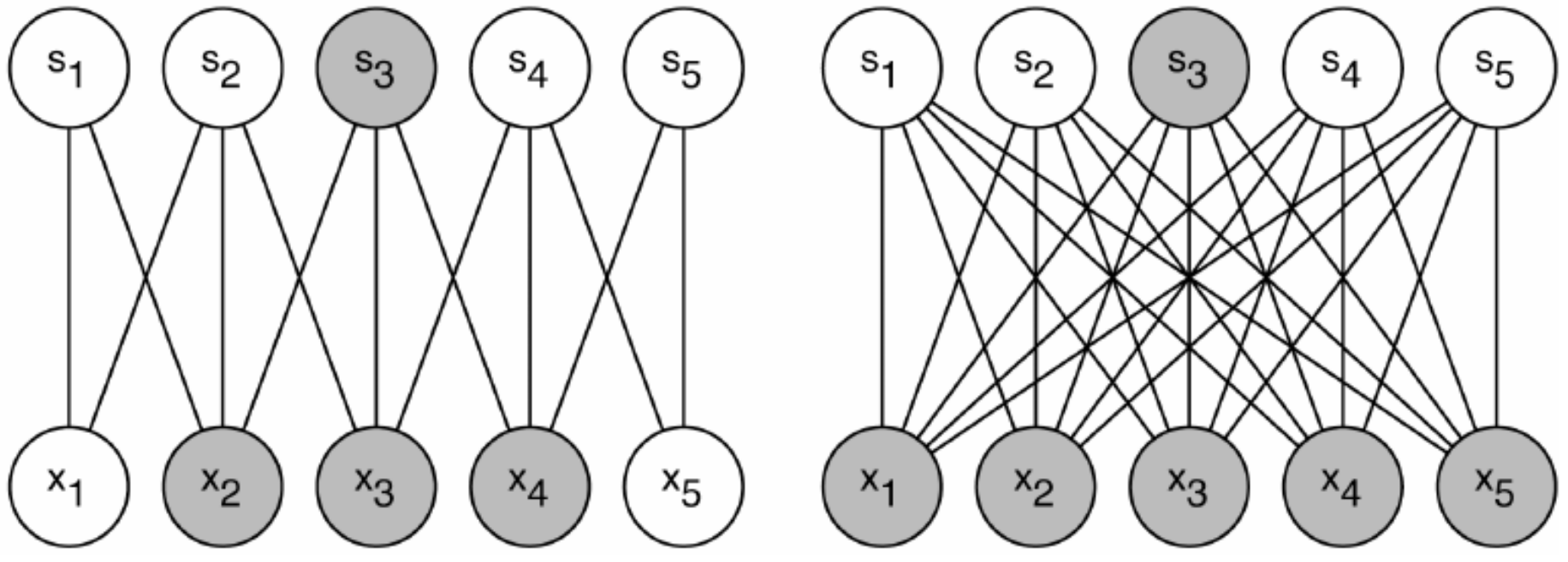


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, $S_3$, and also highlight the input units in $\mathbf{X}$ that affect this unit. These units are known as the *receptive field* of $S_3$. *(Left)* When $\mathbf{S}$ is formed by convolution with a kernel of width 3, only three inputs affect $S_3$ . *(Right)* When $\mathbf{S}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect $S_3$.

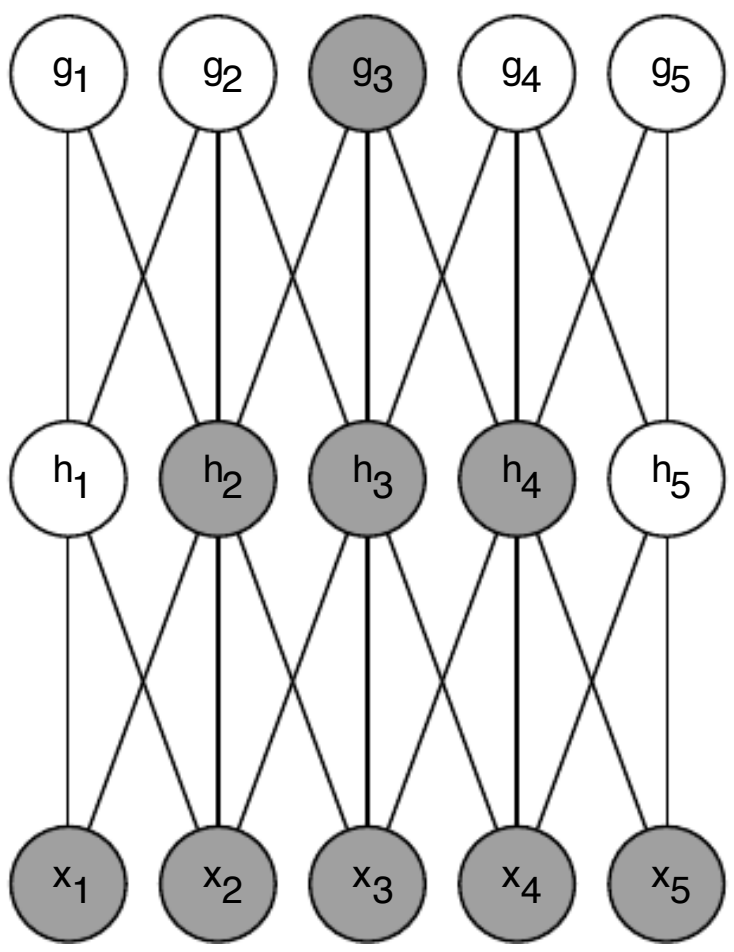


Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution or pooling. This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

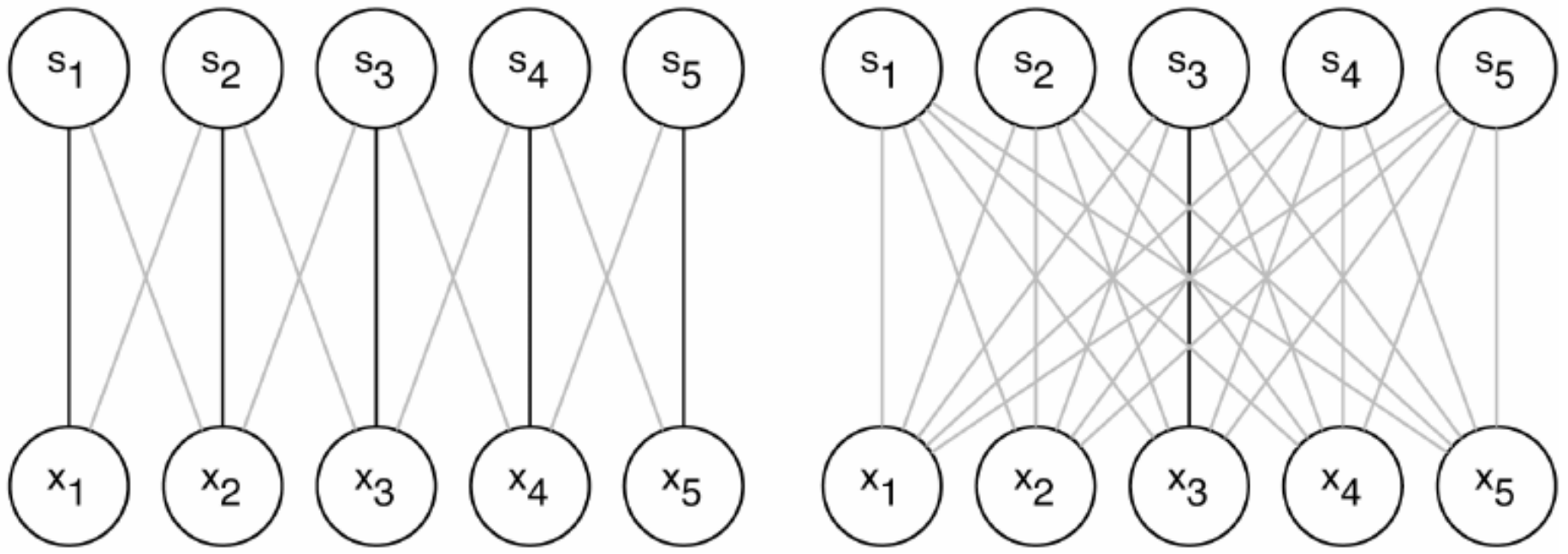


Figure 9.5: *Parameter sharing:* We highlight the connections that use a particular parameter in two different models. *(Left)* We highlight uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Right)* We highlight the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called *equivariance* to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g$ if $f(g(x)) = g(f(x))$. In the case of convolution, if we let $g$ be any function that translate the input, i.e., shifts it, then the convolution function is equivariant to $g$. For example, define $g(x)$ such that for all $i$, $g(x)[i] = x[i-1]$. This shifts every element of $x$ one unit to the right. If we apply this transformation to $x$, then apply convolution, the result will be the same as if we applied convolution to $x$, then applied the transformation to the output. When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that same local function is useful everywhere in the input. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations–the part of the network processing

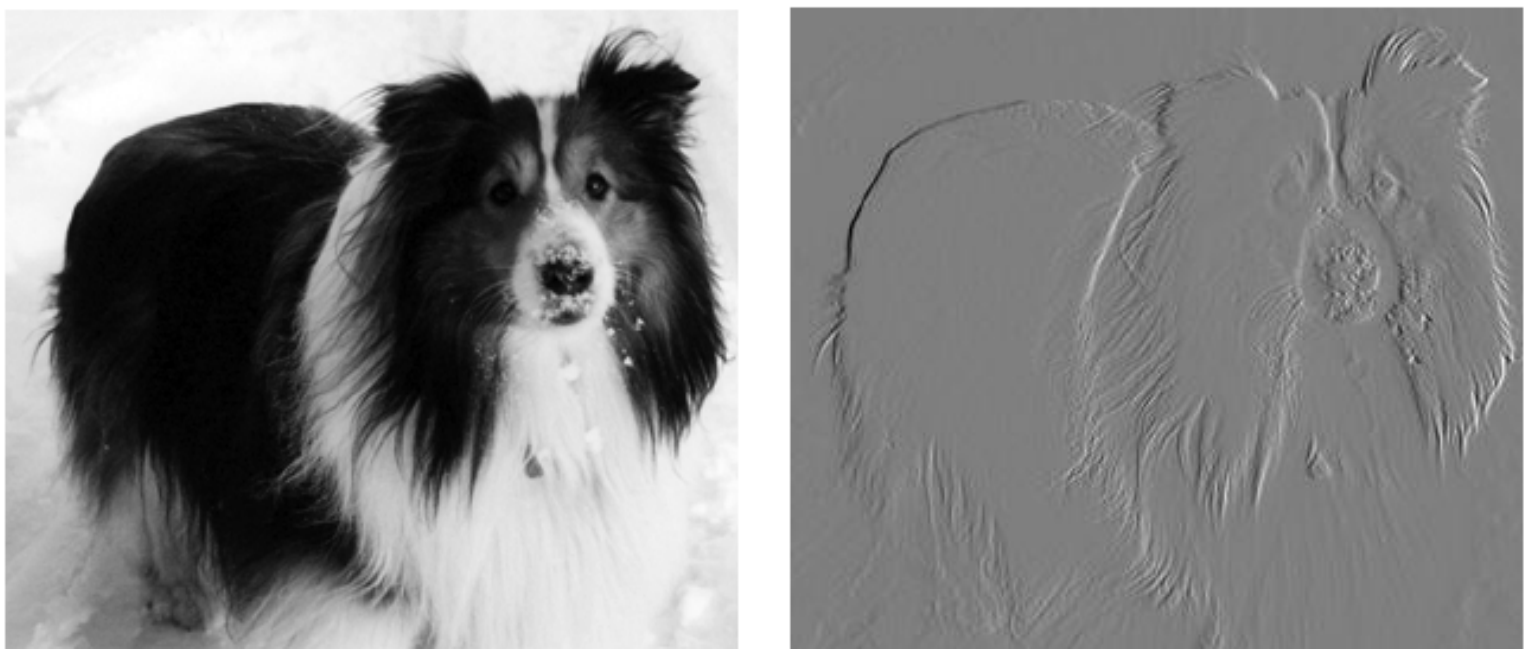

Figure 9.6: *Efficiency of edge detection.* The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing 2 elements, and requires $319 \times 280 \times 3 = 267,960$ floating point operations (two multiplications and one addition per output pixel) to compute. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over 8 billion, entries in the matrix, making convolution 4 billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over 16 billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small, local region across the entire input. (Photo credit: Paula Goodfellow)

the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

Note that convolution is not equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. We discuss this further in section 9.7.

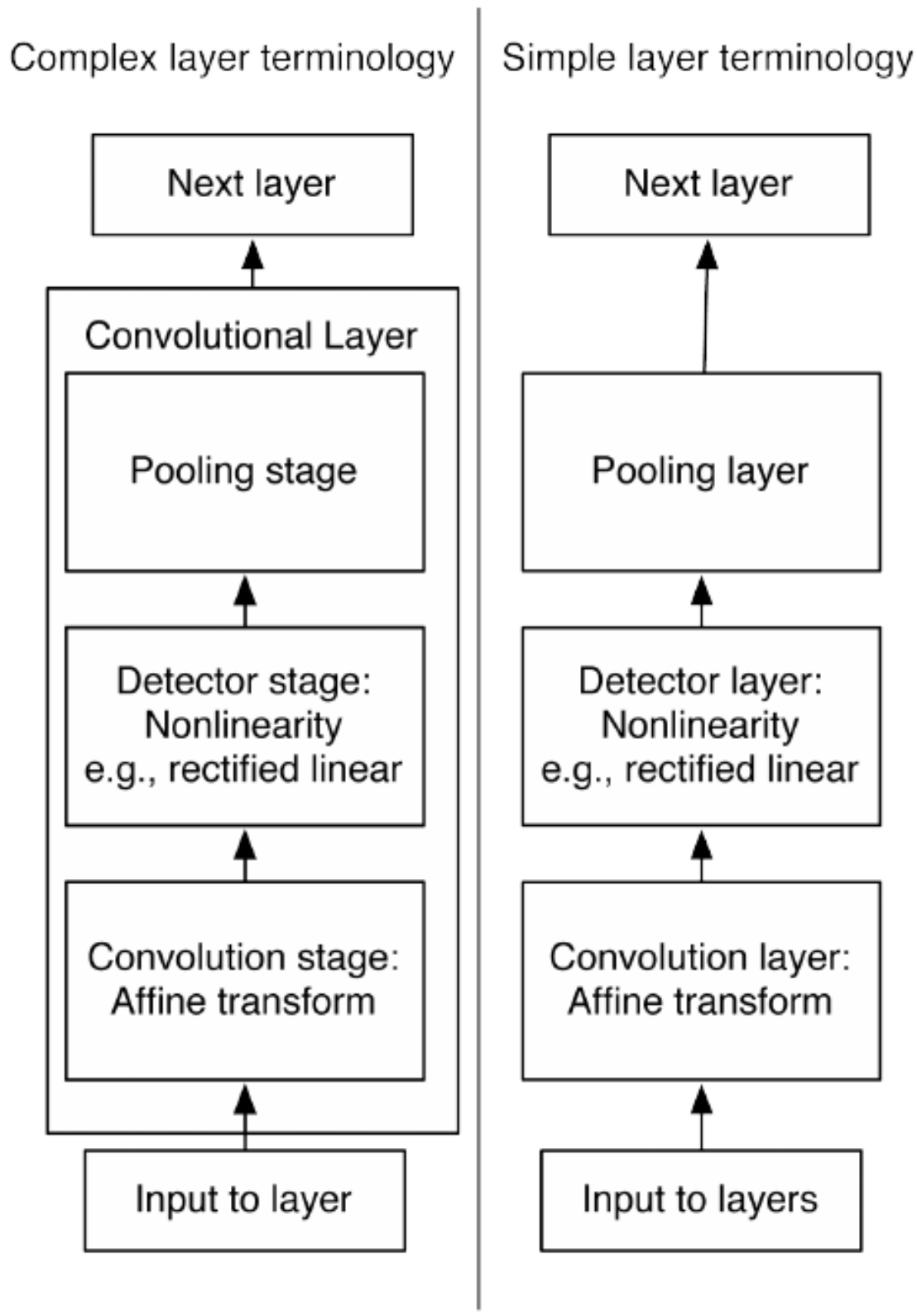


Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. *Left)* In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages." In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. *Right)* In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters.

## 9.3 Pooling

A typical layer of a convolutional network consists of three stages (see Fig. 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of presynaptic activations. In the second stage, each presynaptic activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the *detector* stage. In the third stage, we use a *pooling function* to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation become *invariant* to small translations of the input. This means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. See Fig. 9.8 for an example of how this works. **Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.** For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see Fig. 9.9).

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced $k$ pixels apart rather than 1 pixel apart. See Fig. 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly $k$ times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and

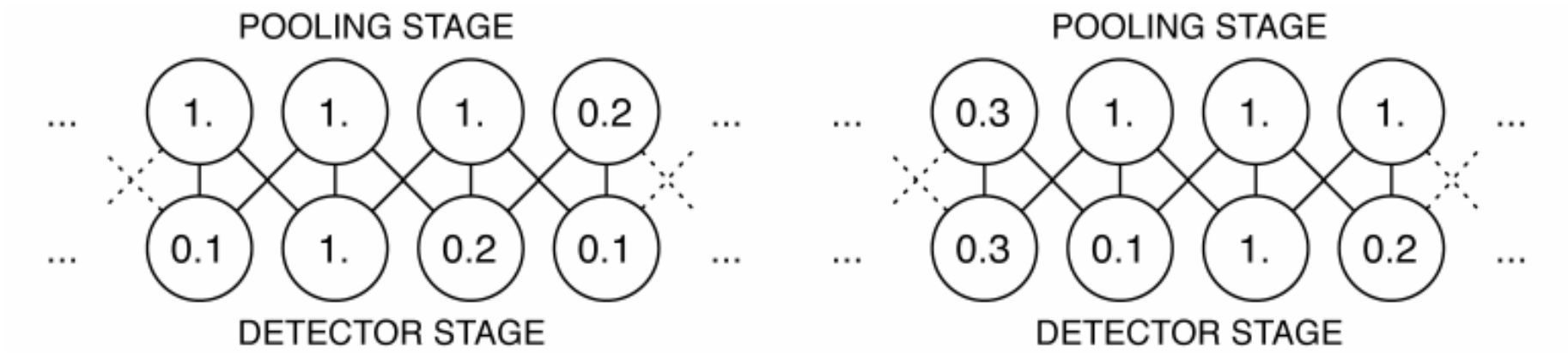


Figure 9.8: *Max pooling introduces invariance.* Left: A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of 1 between pooling regions and a pooling region width of 3. Right: A view of the same network, after the input has been shifted to the right by 1 pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

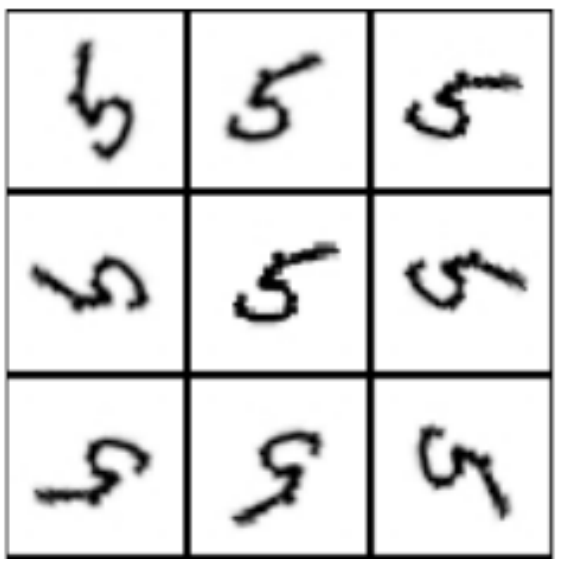

Figure 9.9: *Example of learned invariances:* If each of these filters drive units that appear in the same max-pooling region, then the pooling unit will detect "5"s in any rotation. By learning to have each filter be a different rotation of the "5" template, this pooling unit has learned to be invariant to rotation. This is in contrast to translation invariance, which is usually achieved by hard-coding the net to pool over shifted versions of a single learned filter.

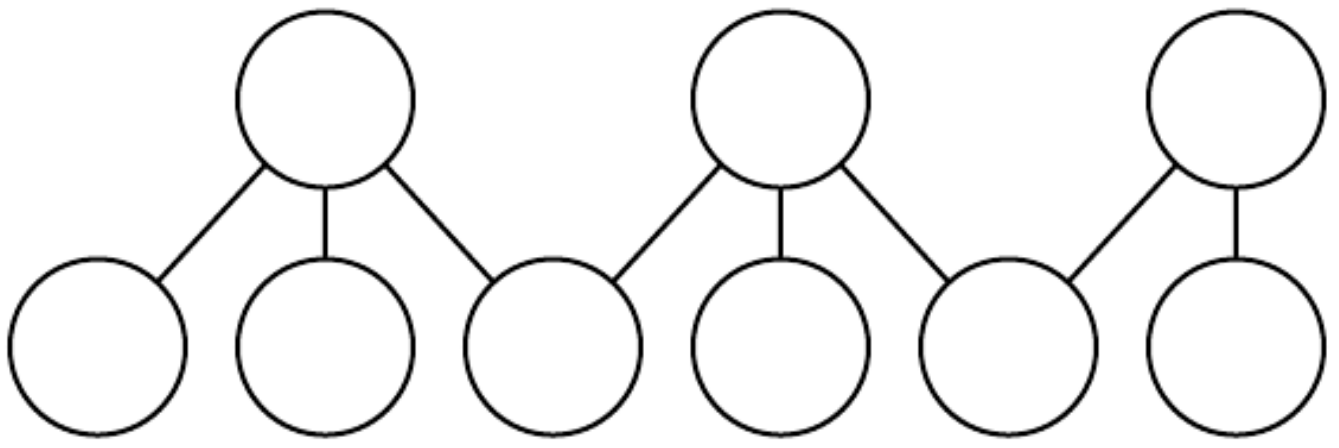

Figure 9.10: *Pooling with downsampling.* Here we use max-pooling with a pool width of 3 and a stride between pools of 2. This reduces the representation size by a factor of 2, which reduces the computational and statistical burden on the next layer. Note that the final pool has a smaller size, but must be included if we do not want to ignore some of the detector units.

reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of and offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau *et al.*, 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia *et al.*, 2012).

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders. These issues will be discussed further when we present these types of networks. Pooling in convolutional Boltzmann machines is presented in Chapter 20.7. The inverse-like operations on pooling units needed in some differentiable networks will be covered in Chapter 20.9.6.

## 9.4 Convolution and Pooling as an Infinitely Strong Prior

Recall the concept of a *prior probability distribution* from Chapter 5.3. This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.

An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data gives to those values.

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior says that the function the layer should learn contains only local interactions and is equivariant to translation. Likewise, the use of pooling is in infinitely strong prior that each unit should be invariant to small translations.

Of course, implementing a convolutional net as a fully connected net with an infinitely strong prior would be extremely computationally wasteful. But thinking of a convolutional net as a fully connected net with an infinitely strong prior can give us some insights into how convolutional nets work.

One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precision spatial information, then using pooling on all features can cause underfitting. (Some convolution network architectures (Szegedy *et al.*, 2014a) are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect) When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

Another key insight from this view is that we should only compare convolu-

tional models to other convolutional models in benchmarks of statistical learning performance. Models that do not use convolution would be able to learn even if we permuted all of the pixels in the image. For many image datasets, there are separate benchmarks for models that are *permutation invariant* and must discover the concept of topology via learning, and models that have the knowledge of spatial relationships hard-coded into them by their designer.

## 9.5 Variants of the Basic Convolution Function

When discussing convolution in the context of neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature. The functions used in practice differ slightly. Here we describe these differences in detail, and highlight some useful properties of the functions used in neural networks.

First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution in parallel. This is because convolution with a single kernel can only extract one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations.

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. (Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch)

Note that because convolutional networks usually use multi-channel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel-flipping is used. These multi-channel operations are only commutative if each operation has the same number of output channels as input channels.

Assume we have a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel $i$ of the output and a unit in channel $j$ of the input, with an offset of $k$ rows and $l$ columns between the output unit and the input unit. Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel $i$ at row $j$ and column $k$. Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$. If $\mathbf{Z}$ is produced by

convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n}$$

where the summation over $l$, $m$ and $n$ is over all values for which the tensor indexing operations inside the summation is valid.

We may also want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. If we want to sample only every $s$ pixels in each direction in the output, then we can defined a downsampled convolution function $c$ such that:

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[ V_{l,j\times s+m,k\times s+n} K_{i,l,m,n} \right]. \tag{9.1}$$

We refer to $s$ as the *stride* of this downsampled convolution. It is also possible to define a separate stride for each direction of motion.

One essential feature of any convolutional network implementation is the ability to implicitly zero-pad the input $\mathbf{V}$ in order to make it wider. Without this feature, the width of the representation shrinks by the kernel width - 1 at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels–both scenarios that significantly limit the expressive power of the network. See Fig. 9.11 for an example.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called *valid* convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image is of size $m \times m$ and the kernel is of size $k \times k$, the output will be of size $m - k + 1 \times m - k + 1$. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to $1 \times 1$, at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero-padding is added to keep the size of the output equal to the size of the input. MATLAB calls this *same* convolution. In this case, the network can contain as many convolutional layers as the available hardware

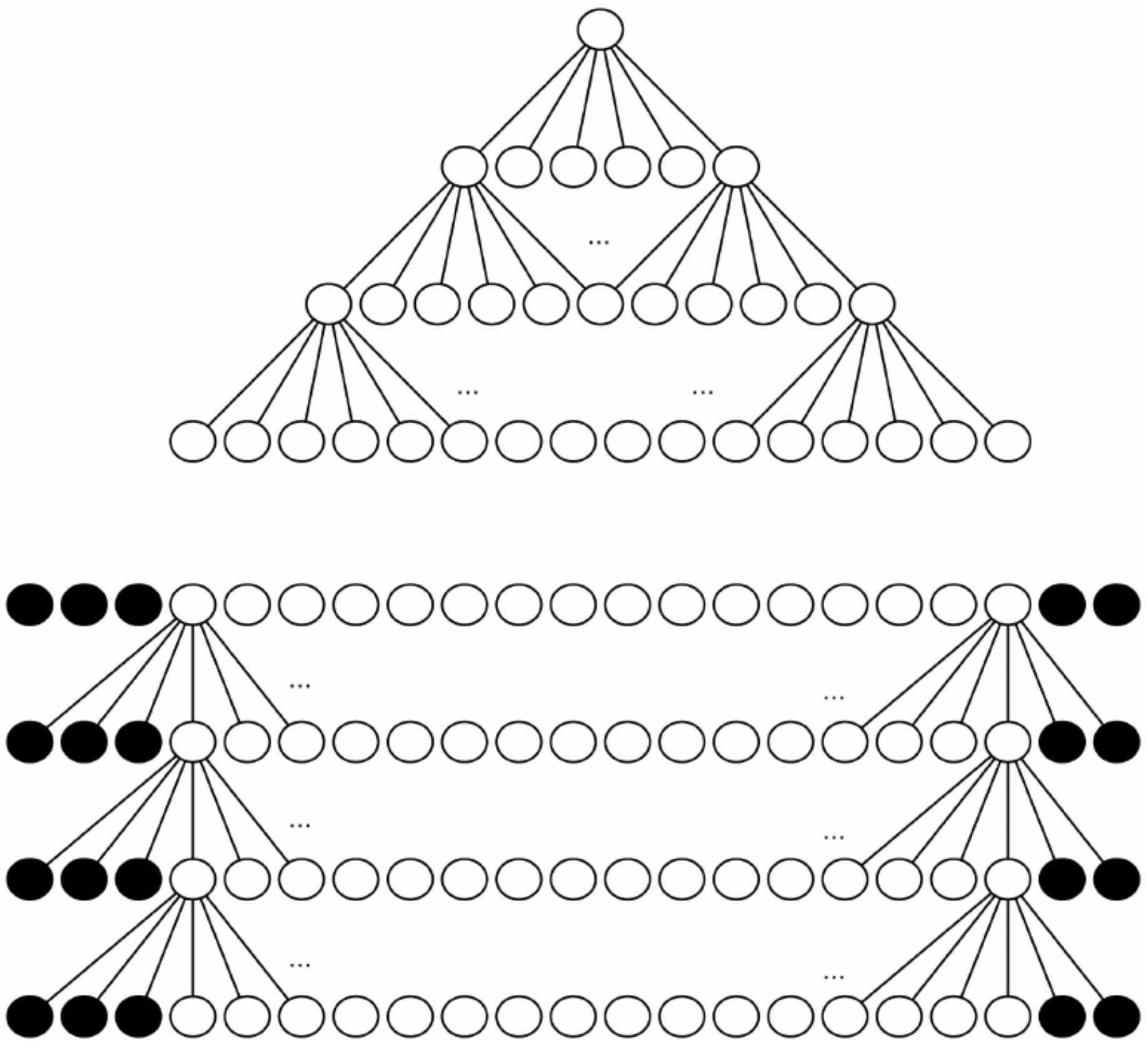

Figure 9.11: *The effect of zero padding on network size*: Consider a convolutional network with a kernel of width six at every layer. In this example, do not use any pooling, so only the convolution operation itself shrinks the network size. *Top)* In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive and some shrinking is inevitable in this kind of architecture. *Bottom)* By adding five implicit zeroes to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

can support, since the operation of convolution does not modify the architectural possibilities available to the next layer. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other extreme case, which MATLAB refers to as *full convolution*, in which enough zeroes are added for every pixel to be visited $k$ times in each direction, resulting in an output image of size $m+k-1 \times m+k-1$. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between "valid" and "same" convolution.

In some cases, we do not actually want to use convolution, but rather locally connected layers. In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified by a 6-D tensor $\mathbf{W}$. The indices into $\mathbf{W}$ are respectively: $i$, the output channel, $j$, the output row, $k$, the output column, $l$, the input channel, $m$, the row offset within the input, and $n$, the column offset within the input. The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} \left[ V_{l,j+m,k+n} w_{i,j,k,l,m,n} \right].$$

This is sometimes also called *unshared convolution*, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations.

Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space. For example, if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image.

It can also be useful to make versions of convolution or locally connected layers in which the connectivity is further restricted, for example to constraint that each output channel $i$ be a function of only a subset of the input channels $l$. A common way to do this is to make the first $m$ output channels connect to only the first $n$ input channels, the second $m$ output channels connect to only the second $n$ input channels, and so on. See Fig. 9.12 for an example. Modeling interactions between few channels allows the network to have fewer parameters in order to reduce memory consumption and increase statistical efficiency, and also reduces the amount of computation needed to perform forward and back-propagation. It accomplishes these goals without reducing the number of hidden units.

*Tiled convolution* (Gregor and LeCun, 2010; Le *et al.*, 2010) offers a compro-

Figure 9.12: A convolutional network with the first two output channels connected to only the first two input channels, and the second two output channels connected to only the second two input channels.

mise between a convolutional layer and a locally connected layer. Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space. This means that immediately neighboring locations will have different filters, like in a locally connected layer, but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels, rather than the size of the entire output feature map.

To define tiled convolution algebraically, let $k$ be a 6-D tensor, where two of the dimensions correspond to different locations in the output map. Rather than having a separate index for each location in the output map, output locations cycle through a set of $t$ different choices of kernel stack in each direction. If $t$ is equal to the output width, this is the same as a locally connected layer.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n,j\%t,k\%t}$$

It is straightforward to generalize this equation to use a different tiling range for each dimension.

Both locally connected layers and tiled convolutional layers have an interesting interaction with max-pooling: the detector units of these layers are driven by different filters. If these filters learn to detect different transformed versions of the same underlying features, then the max-pooled units become invariant to the learned transformation (see Fig. 9.9). Convolutional layers are hard-coded to be invariant specifically to translation.

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, one must be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs. In some simple cases, this operation can be performed using the convolution operation, but many cases of interest, including the case of stride greater than 1, do not have this property.

Recall that convolution is a linear operation and can thus be described as a matrix multiplication (if we first reshape the input tensor into a flat vector). The matrix involved is a function of the convolution kernel. The matrix is sparse and each element of the kernel is copied to several elements of the matrix. It is not usually practical to implement convolution in this way, but this view helps us to derive some of the other operations needed to implement a convolutional network.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to backpropagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed to compute the reconstruction in a convolutional autoencoder (or to perform the analogous role in a convolutional RBM, sparse coding model, etc.). Like the kernel gradient operation, this input gradient operation can be implemented using a convolution in some cases, but in the general case requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

It turns out that these three operations–convolution, backprop from output to weights, and backprop from output to inputs–are sufficient to compute all of the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution. See (Goodfellow, 2010) for a full derivation of the equations in the fully general multi-dimensional, multi-example case. To give a sense of how these equations work, we present the two dimensional, single example version here.

Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack $\mathbf{K}$ applied to multi-channel image $\mathbf{V}$ with stride $s$ is defined by $c(\mathbf{K}, \mathbf{V}, s)$ as in equation 9.1. Suppose we want to minimize some loss function $J(\mathbf{V}, \mathbf{K})$. During forward propagation, we will need to use $c$ itself to output $\mathbf{Z}$, which is then propagated through the rest of the network and used to compute $J$. . During backpropagation, we will receive a tensor $\mathbf{G}$ such that $G_{i,j,k} = \frac{\partial}{\partial Z} J(\mathbf{V}, \mathbf{K})$.

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,m \times s+k,n \times s+l}.$$

If this layer is not the bottom layer of the network, we'll need to compute the gradient with respect to $\mathbf{V}$ in order to backpropagate the error farther down. To do so, we can use a function

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) = \sum_{l,m|s \times l+m=j} \sum_{n,p|s \times n+p=k} \sum_{q} K_{q,i,m,p} G_{i,l,n}.$$

We could also use $h$ to define the reconstruction of a convolutional autoencoder, or the probability distribution over visible given hidden units in a convolutional RBM or sparse coding model. Suppose we have hidden units $\mathbf{H}$ in the same format as $\mathbf{Z}$ and we define a reconstruction

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s).$$

In order to train the autoencoder, we will receive the gradient with respect to $\mathbf{R}$ as a tensor $\mathbf{E}$. To train the decoder, we need to obtain the gradient with respect to $\mathbf{K}$. This is given by $g(\mathbf{H}, \mathbf{E}, s)$. To train the encoder, we need to obtain the gradient with respect to $\mathbf{H}$. This is given by $c(\mathbf{K}, \mathbf{E}, s)$. It is also possible to differentiate through $g$ using $c$ and $h$, but these operations are not needed for the backpropagation algorithm on any standard network architectures.

Generally, we do not use only a linear operation in order to transform from the inputs to the outputs in a convolutional layer. We generally also add some bias term to each output before applying the nonlinearity. This raises the question of how to share parameters among the biases. For locally connected layers it is natural to give each unit its own bias, and for tiled convolution, it is natural to share the biases with the same tiling pattern as the kernels. For convolutional layers, it is typical to have one bias per channel of the output and share it across all locations within each convolution map. However, if the input is of known, fixed size, it is also possible to learn a separate bias at each location of the output map. Separating the biases may slightly reduce the statistical efficiency of the model, but also allows the model to correct for differences in the image statistics at different locations. For example, when using implicit zero padding, detector units at the edge of the image receive less total input and may need larger biases.

## 9.6 Structured Outputs

Although we have mostly talked about convolutional networks used for classification tasks, with one output per category, a convolutional network can also be used to produce as output a high-dimensional structured object, such as the set of pixel-wise segmentation label predictions. In that case, both the input and the output of the network have an image topology. For each input pixel or block of $k \times k$ input pixels, the network can output a vector of scores, such as the probabilities associated with different categories.

Since a convolutional layer outputs tensor $\mathbf{S}$, with an output vector $\mathbf{S}_{i,j,:} =$ at each 2-D location $(i, j)$, we can simply avoid stacking fully-connected layers on top of the convolutional layers and obtain a spatially structured output. We can train a convolutional network with targets associated with each location, each corresponding to a group of output units. If we want to perform a classification

at each location (corresponding to an input pixel or a block of input pixels) we can separately apply the softmax function at each location, to the vector $\mathbf{S}_{i,j,:}$ (pre-activation values) associated with that location. We only need to make sure that the last convolutional layer has the same number of dimensions per location as the desired number of output categories.

One issue that often comes up is that the output plane can be smaller than the input plane. It could be smaller because of border effects, if we use the "valid" convolutions rather than the "full" convolutions. But the main cause for reduction is the presence of pooling layer. Each pooling layer typically aggregates and subsamples. For example, a $2 \times 2$ pooling layer will reduce the feature map dimensions by the same factor. Some authors are have preferred to avoid pooling altogether (Jain *et al.*, 2007), while assigning the same output label to blocks of input pixels was acceptable to others (Pinheiro and Collobert, 2014, 2015).

After some initial features have been extracted, we may interpret the successive convolutional layers of such a deep convolutional network as iteratively coordinating the answers given at each image location. This suggests using the same convolutions at each stage, thus sharing weights between the last layers of the deep net, as pointed out by Jain *et al.* (2007). This makes the sequence of computations performed by the successive convolutional layers with weights shared across layers a particular kind of recurrent network, as discussed by Pinheiro and Collobert (2014, 2015).

Once a prediction for each pixel is made, various methods can be used to further process these predictions in order to obtain a segmentation of the image into regions (Briggman *et al.*, 2009; Turaga *et al.*, 2010; Farabet *et al.*, 2013b). The general idea is to exploit the a priori assumption that large groups of contiguous pixels tend to be associated with the same label. In part 3 of this book we will talk about methods based on graphical models to capture the joint distribution of structured outputs (such as the labels to assign to individual pixels), conditional on the input. These methods, like conditional random fields (CRFs), can also be used for image segmentation, given the output of a convolutional network. Alternatively, the convolutional network can be trained with respect to an approximation of the CRF training objective (Ning *et al.*, 2005; Thompson *et al.*, 2014).

## 9.7 Data Types

The data used with a convolutional network usually consists of several channels, each channel being the observation of a different quantity at some point in space or time. See Table 9.1 for examples of data types with different dimensionalities and number of channels.

| | Single channel | Multi-channel |
|---|---|---|
| 1-D | Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step. | Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a "skeleton" over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character's skeleton. Each channel in the data we feed to the convolutional model represents the angle about one axis of one joint. |
| 2-D | Audio data that has been preprocessed with a Fourier transform: We can transform the audio waveform into a 2D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network's output. | Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and vertical axes of the image, conferring translation equivariance in both directions. |
| 3-D | Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans. | Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame. |

Table 9.1: Examples of different formats of data that can be used with convolutional networks.

So far we have discussed only the case where every example in the train and test data has the same spatial dimensions. One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues.

For example, consider a collection of images, where each image has a different width and height. It is unclear how to apply matrix multiplication. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Sometimes the output of the network is allowed to have variable size as well as the input, for example if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example if we want to assign a single class label to the entire image. In this case we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, in order to maintain a fixed number of pooled outputs.

Note that the use of convolution for processing variable sized inputs only makes sense for inputs that have variable size because they contain varying amounts of observation of the same kind of thing–different lengths of recordings over time, different widths of observations over space, etc. Convolution does not make sense if the input has variable size because it can optionally include different kinds of observations. For example, if we are processing college applications, and our features consist of both grades and standardized test scores, but not every applicant took the standardized test, then it does not make sense to convolve the same weights over both the features corresponding to the grades and the features corresponding to the test scores.

## 9.8 Efficient Convolution Algorithms

Modern convolutional network applications often involve networks containing more than one million units. Powerful implementations exploiting parallel computation resources, as discussed in Chapter 12.1 are essential. However, in many cases it is also possible to speed up convolution by selecting an appropriate convolution algorithm.

Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naive

implementation of discrete convolution.

When a $d$-dimensional kernel can be expressed as the outer product of $d$ vectors, one vector per dimension, the kernel is called *separable*. When the kernel is separable, naive convolution is inefficient. It is equivalent to compose $d$ one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one $k$-dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors. If the kernel is $w$ elements wide in each dimension, then naive multidimensional convolution requires $O(w^d)$ runtime and parameter storage space, while separable convolution requires $O(w \times d)$ runtime and parameter storage space. Of course, not every convolution can be represented in this way.

Devising faster ways of performing convolution or approximate convolution without harming the accuracy of the model is an active area of research. Even techniques that improve the efficiency of only forward propagation are useful because in the commercial setting, it is typical to devote many more resources to deployment of a network than to its training.

## 9.9 Random or Unsupervised Features

Typically, the most expensive part of convolutional network training is learning the features. The output layer is usually relatively inexpensive due to the small number of features provided as input to this layer after passing through several layers of pooling. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network. One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

There are two basic strategies for obtaining convolution kernels without supervised training. One is to simply initialize them randomly. The other is to learn them with an unsupervised criterion. This approach allows the features to be determined separately from the classifier layer at the top of the architecture. One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer. Learning the last layer is then typically a convex optimization problem, assuming the last layer is something like logistic regression or an SVM.

Random filters often work surprisingly well in convolutional networks (Jarrett *et al.*, 2009b; Saxe *et al.*, 2011; Pinto *et al.*, 2011; Cox and Pinto, 2011). Saxe *et al.* (2011) showed that layers consisting of convolution following by pooling naturally become frequency selective and translation invariant when assigned random weights. They argue that this provides an inexpensive way to choose the

architecture of a convolutional network: first evaluate the performance of several convolutional network architectures by training only the last layer, then take the best of these architectures and train the entire architecture using a more expensive approach.

An intermediate approach is to learn the features, but using methods that do not require full forward and back-propagation at every gradient step. As with multilayer perceptrons, we use greedy layer-wise unsupervised pretraining, to train the first layer in isolation, then extract all features from the first layer only once, then train the second layer in isolation given those features, and so on. The canonical example of this is the convolutional deep belief network (Lee *et al.*, 2009). Convolutional networks offer us the opportunity to take this strategy one step further than is possible with multilayer perceptrons. Instead of training an entire convolutional layer at a time, we can actually train a small but densely-connected unsupervised model (such as PSD, described in Chapter 15.8.2, or $k$-means) of a single image *patch*. We can then use the weight matrices from this patch-based model to define the kernels of a convolutional layer. This means that it is possible to use unsupervised learning to train a convolutional network *without ever using convolution during the training process*. Using this approach, we can train very large models and incur a high computational cost only at inference time (Ranzato *et al.*, 2007b; Jarrett *et al.*, 2009b; Kavukcuoglu *et al.*, 2010a; Coates *et al.*, 2013).

As with other approaches to unsupervised pretraining, it remains difficult to tease apart the cause of some of the benefits seen with this approach. Unsupervised pretraining may offer some regularization relative to supervised training, or it may simply allow us to train much larger architectures due to the reduced computational cost of the learning rule.

## 9.10 The Neuroscientific Basis for Convolutional Networks

Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence. Though convolutional networks have been guided by many other fields, some of the key design principles of neural networks were drawn from neuroscience.

The history of convolutional networks begins with neuroscientific experiments long before the relevant computational models were developed. Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many of the most basic facts about how the mammalian vision system works (Hubel and Wiesel, 1959, 1962, 1968). Their accomplishements were eventually recognized with a Nobel Prize. Their findings that have had the greatest

influence on contemporary deep learning models were based on recording the activity of individual neurons in cats. By anesthetizing the cat, they could immobilize the cat's eye and observe how neurons in the cat's brain responded to images projected in precise locations on a screen in front of the cat.

Their worked helped to characterize many aspects of brain function that are beyond the scope of this book. From the point of view of deep learning, we can focus on a simplified, cartoon view of brain function.

In this simplified view, we focus on a part of the brain called *V1*, also known as the *primary visual cortex*. V1 is the first area of the brain that begins to perform significantly advanced processing of visual input. In this cartoon view, images are formed by light arriving in the eye and stimulating the retina, the light-sensitive tissue in the back of the eye. The neurons in the retina perform some simple preprocessing of the image but do not substantially alter the way it is represented. The image then passes through the optic nerve and a brain region called the lateral geniculate nucleus. The main role, as far as we are concerned here, of both of these anatomical regions is primarily just to carry the signal from the eye to V1, which is located at the back of the head.

A convolutional network layer is designed to capture three properties of V1:

1. V1 is arranged in a spatial map. It actually has a two-dimensional structure mirroring the structure of the image in the retina. For example, light arriving at the lower half of the retina affects only the corresponding half of V1. Convolutional networks capture this property by having their features defined in terms of two dimensional maps.

2. V1 contains many *simple cells*. A simple cell's activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field. The detector units of a convolutional network are designed to emulate these properties of simple cells. V1 also contains many *complex cells*. These cells respond to features that are similar to those detected by simple cells, but complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks. Complex cells are also invariant to some changes in lighting that cannot be captured simply by pooling over spatial locations. These invariances have inspired some of the cross-channel pooling strategies in convolutional networks, such as maxout units (Goodfellow *et al.*, 2013a).

Though we know the most about V1, it is generally believed that the same basic principles apply to other brain regions. In our cartoon view of the visual system, the basic strategy of detection followed by pooling is repeatedly applied as we move deeper into the brain. As we pass through multiple anatomical layers of the brain, we eventually find cells that respond to some specific concept and are

invariant to many transformations of the input. These cells have been nicknamed "grandmother cells"— the idea is that a person could have a neuron that activates when seeing an image of their grandmother, regardless of whether she appears in the left or right side of the image, whether the image is a close-up of her face or zoomed out shot of her entire body, whether she is brightly lit, or in shadow, etc.

These grandmother cells have been shown to actually exist in the human brain, in a region called the medial temporal lobe (Quiroga *et al.*, 2005). Researchers tested whether individual neurons would respond to photos of famous individuals. They found what has come to be called the "Halle Berry neuron": an individual neuron that is activated by the concept of Halle Berry. This neuron fires when a person sees a photo of Halle Berry, a drawing of Halle Berry, or even text containing the words "Halle Berry." Of course, this has nothing to do with Halle Berry herself; other neurons responded to the presence of Bill Clinton, Jennifer Aniston, etc.

These medial temporal lobe neurons are somewhat more general than modern convolutional networks, which would not automatically generalize to identifying a person or object when reading its name. The closest analog to a convolutional network's last layer of features is a brain area called the inferotemporal cortex (IT). When viewing an object, information flows from the retina, through the LGN, to V1, then onward to V2, then V4, then IT. This happens within the first 100ms of glimpsing an object. If a person is allowed to continue looking at the object for more time, then information will begin to flow backwards as the brain uses top-down feedback to update the activations in the lower level brain areas. However, if we interrupt the person's gaze, and observe only the firing rates that result from the first 100ms of mostly feed-forward activation, then IT proves to be very similar to a convolutional network. Convolutional networks can predict IT firing rates, and also perform very similarly to (time limited) humans on object recognition tasks (DiCarlo, 2013).

That being said, there are many differences between convolutional networks and the mammalian vision system. Some of these differences are well known to computational neuroscientists, but outside the scope of this book. Some of these differences are not yet known, because many basic questions about how the mamalian vision system works. As a brief list:

- The human eye is mostly very low resolution, except for a tiny patch called the *fovea*. The fovea only observes an area about the size of a thumbnail held at arms length. Though we feel as if we can see an entire scene in high resolution, this is an illusion created by the subconscious part of our brain, as it stitches together several glimpses of small areas. Most convolutional networks actual receive large full resolution photographs as input.

- The human visual system is integrated with many other senses, such as

hearing, and factors like our moods and thoughts. Convolutional networks so far are purely visual.

- The human visual system does much more than just recognize objects. It is able to understand entire scenes including many objects and relationships between objects, and processes rich 3-D geometric information needed for our bodies to interface with the world. Convolutional networks have been applied to some of these problems but these applications are in their infancy.

- Even simple brain areas like V1 are heavily impacted by feedback from higher levels. Feedback has been explored extensively in neural network models but has not yet been shown to offer a compelling improvement.

- While feed-forward IT firing rates capture much of the same information as convolutional network features, it's not clear how similar the intermediate computations are. The brain probably uses very different activation and pooling functions. An individual neuron's activation probably is not well-characterized by a single linear filter response. A recent model of V1 involves multiple quadratic filters for each neuron (Rust *et al.*, 2005). Indeed our cartoon picture of "simple cells" and "complex cells" might create a non-existent distinction; simple cells and complex cells might both be the same kind of cell but with their "parameters" enabling a continuum of behaviors ranging from what we call "simple" to what we call "complex."

It's also worth mentioning that neuroscience has told us relatively little about how to *train* convolutional networks. Model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision (Marr and Poggio, 1976), but these models did not use the modern backpropagation algorithm and gradient descent. For example, the Neocognitron (Fukushima, 1980) incorporated most of the model architecture design elements of the modern convolutional network but relied on a layerwise unsupervised clustering algorithm.

Lang and Hinton (1988) introduced the use of backpropagation to train *time-delay neural networks* (TDNNs). To use contemporary terminology, TDNNs are one-dimensional convolutional networks applied to time series. Back-propagation applied to these models was not inspired by any neuroscientific observation and is considered by some to be biologically implausible. Following the success of backpropagation-based training of TDNNs, (LeCun *et al.*, 1989) developed the modern convolutional network by applying the same training algorithm to 2-D convolution applied to images.

So far we have described how simple cells are roughly linear and selective for certain features, complex cells are more non-linear and become invariant to some

transformations of these simple cell features, and stacks of layers that alternative between selectivity and invariance can yield grandmother cells for very specific phenomena. We have not yet described precisely what these individual cells detect. In a deep, nonlinear network, it can be difficult to understand the function of individual cells. Simple cells in the first layer are easier to analyze, because their responses are driven by a linear function. In an artificial neural network, we can just display an image of the kernel to see what the corresponding channel of a convolutional layer responds to. In a biological neural network, we do not have access to the weights themselves. Instead, we put an electrode in the neuron itself, display several samples of white noise images in front of the animal's retina, and record how each of these samples causes the neuron to activate. We can then fit a linear model to these responses in order to obtain an approximation of the neuron's weights. This approach is known as *reverse correlation* (Ringach and Shapley, 2004).

Reverse correlation shows us that most V1 cells have weights that are described by *Gabor functions*. The Gabor function describes the weight at a 2-D point in the image. We can think of an image as being a function of 2-D coordinates, $I(x, y)$. Likewise, we can think of a simple cell as sampling the image at a set of locations, defined by a set of $x$ coordinates and a set of $y$ coordinates, , and applying weights that are also a function of the location, $w(x, y)$. From this point of view, the response of a simple cell to an image is given by

$$s(I) = \sum_{x\in} \sum_{y\in} w(x, y) I(x, y).$$

Specifically, $w(x, y)$ takes the form of a Gabor function:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi),$$

where

$$x' = (x - x_0)\cos(\tau) + (y - y_0)\sin(\tau)$$

and

$$y' = -(x - y_0)\sin(\tau) + (y - y_0)\cos(\tau).$$

Here, $\alpha$, $\beta_x$, $\beta_y$, $f$, $\phi$, $x_0$, $y_0$, and $\tau$ are parameters that control the properties of the Gabor function. Fig. 9.13 shows some examples of Gabor functions with different settings of these parameters.

The parameters $x_0$, $y_0$, and $\tau$ define a coordinate system. We translate and rotate $x$ and $y$ to form $x'$ and $y'$. Specifically, the simple cell will respond to image features centered at the point $(x_0, y_0)$, and it will respond to changes in brightness as we move along a line rotated $\tau$ radians from the horizontal.

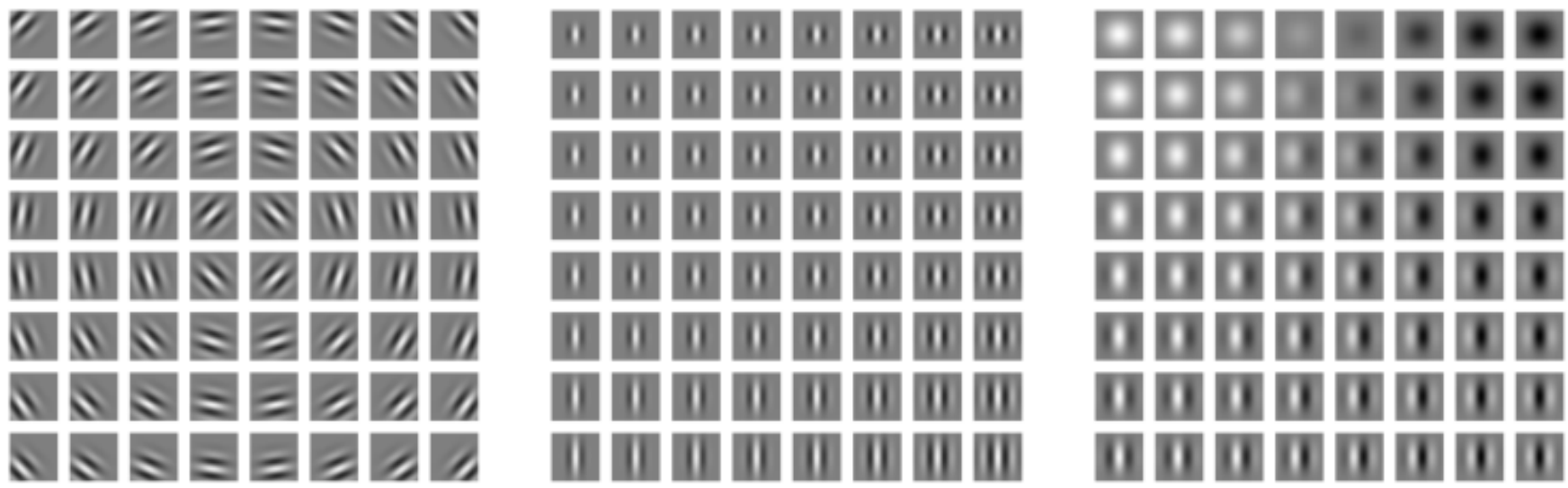

Figure 9.13: Gabor functions with a variety of parameter settings. White indicates large positive weight, black indicates large negative weight, and the background gray corresponds to zero weight. *Left)* Gabor functions with different values of the parameters that control the coordinate system: $x_0$, $y_0$, and $\tau$. Each gabor function in this grid is assigned a value of $x_0$ and $y_0$ proportional to its position in its grid, and $\tau$ is chosen so that each Gabor is sensitive to the direction radiating out from the center of the grid. For the other two plots, $x_0$, $y_0$, and $\tau$ are fixed to zero. *Center)* Gabor functions with different Gaussian scale parameters *beta* and $\beta$ . Gabor functions are arranged in increasing width (decreasing $\beta$ ) as we move left to right through the grid, and increasing height (decreasing $\beta$ ) as we move top to bottom. For the other two plots, the $\beta$ values are fixed to 1.5× the image width. *Right)* Gabor functions with different sinusoid parameters $f$ and $\phi$. As we move top to bottom, $f$ increases, and as we move left to right, $\phi$ increases. For the other two plots, $\phi$ is fixed to 0 and $f$ is fixed to 5× the image width.

Viewed as a function of $x'$ and $y'$, the function $w$ then responds to changes in brightness as we move along the $x'$ axis. It has two important factors: one is a Gaussian function and the other is a cosine function.

The Gaussian factor $\alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right)$ can be seen as a gating term that ensures the simple cell will only respond to values near where $x'$ and $y'$ are both zero, in other words, near the center of the cell's receptive field. The scaling factor $\alpha$ adjusts the total magnitude of the simple cell's response, while $\beta_x$ and $\beta_y$ control how quickly its receptive field falls off.

The cosine factor $\cos(fx' + \phi)$ controls how the simple cell responds to changing brightness along the $x'$ axis. The parameter $f$ controls the frequency of the cosine and $\phi$ controls its phase offset.

Altogether, this cartoon view of simple cells means that a simple cell responds to a specific spatial frequency of brightness in a specific direction at a specific location. They are most excited when the wave of brightness in the image has the same phase as the weights (i.e., when the image is bright where the weights are positive and dark where the weights are negative) and are most inhibited when the wave of brightness is fully out of phase with the weights (i.e., when the image is dark where the weights are positive and bright where the weights are negative).

The cartoon view of a complex cell is that it computes the $L^2$ norm of the 2-D vector containing two simple cell's responses: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. An important special case occurs when $s_1$ has all of the same parameters as $s_0$ except for $\phi$, and $\phi$ is set such that $s_1$ is one quarter cycle out of phase with $s_0$. In this case, $s_0$ and $s_1$ form a *quadrature pair*. A complex cell defined in this way responds when the Gaussian reweighted image $I(x, y) \exp(-\beta_x x'^2 - \beta_y y'^2)$ contains a high amplitude sinusoidal wave with frequency $f$ in direction $\tau$ near $(x_0, y_0)$, *regardless of the phase offset of this wave*. In other words, the complex cell is invariant to small translations of the image in direction $\tau$, or to negating the image (replacing black with white and vice versa).

Some of the most striking correspondences between neuroscience and machine learning come from visually comparing the features learned by machine learning models with those employed by V1. Olshausen and Field (1996) showed that a simple unsupervised learning algorithm, sparse coding, learns features with receptive fields similar to those of simple cells. Since then, we have found that an extremely wide variety of statistical learning algorithms learn features with Gabor-like functions when applied to natural images. This includes most deep learning algorithms, which learn these features in their first layer. Fig. 9.14 shows some examples. Because so many different learning algorithms learn edge detectors, it is difficult to conclude that any specific learning algorithm is the "right" model of the brain just based on the features that it learns (though it can certainly be a bad sign if an algorithm does *not* learn some sort of edge detector when applied to natural images). These features are an important part of the statistical structure of natural images and can be recovered by many different approaches to statistical modeling. See Hyvärinen *et al.* (2009) for a review of the field of natural image statistics.

## 9.11 Convolutional Networks and the History of Deep Learning

Convolutional networks have played an important role in the history of deep learning. They are a key example of a successful application of insights obtained by studying the brain to machine learning applications. They were also some of the first deep models to perform well, long before arbitrary deep models were considered viable. Convolutional networks were also some of the first neural networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. For example, in the 1990's, the neural network research group at AT&T developed a convolutional network for reading checks (LeCun *et al.*, 1998d). By the end of the 1990's, this system deployed by NEC was reading over 10% of all the checks in the US. A little bit later, several

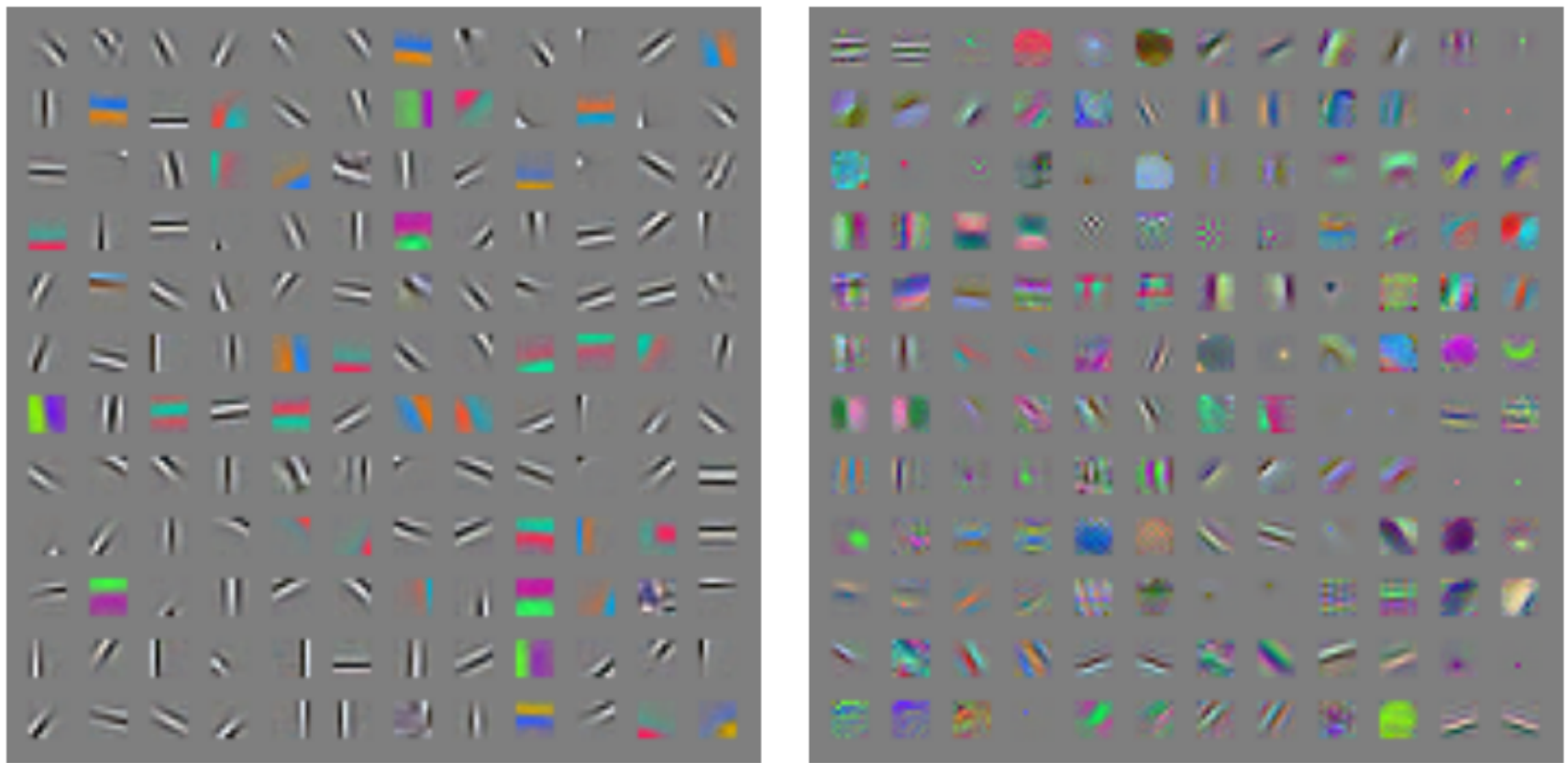

Figure 9.14: Many machine learning algorithms learn features that detect edges or specific colors of edges when applied to natural images. These feature detectors are reminiscent of the Gabor functions known to be present in primary visual cortex. *Left) Weights learned by an unsupervised learning algorithm (spike and slab sparse coding) applied to small image patches. Right)* Convolution kernels learned by the first layer of a fully supervised convolutional maxout network. Neighboring pairs of filters drive the same maxout unit.

OCR and handwriting recognitions systems based on convolutional nets were deployed by Microsoft (Simard *et al.*, 2003). See Chapter 12 for more details on such applications and more modern applications of convolutional networks. See LeCun *et al.* (2010) for a more in-depth history of convolutional networks up to 2010.

Convolutional networks were also used to win many contests. The current intensity of commercial interest in deep learning began when Krizhevsky *et al.* (2012a) won the ImageNet object recognition challenge, but convolutional networks had been used to win other machine learning and computer vision contests with less impact for years earlier.

Convolutional nets were some of first working deep networks trained with back-propagation. It is not entirely clear why convolutional networks succeeded when general backpropagation networks were considered to have failed. It may simply be that convolutional networks were more computationally efficient than fully connected networks, so it was easier to run multiple experiments with them and tune their implementation and hyperparameters. Larger networks also seem to be easier to train. With modern hardware, fully connected networks appear to perform reasonably on many tasks, even when using datasets that were available and activation functions that were popular during the times when fully connected

networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological, e.g., having trained for much longer (rather than concluding that training had stalled out of discouragement) might have yielded much better results. Whatever the case, it is fortunate that convolutional networks performed well decades ago. In many ways, they "carried the torch" for the rest of deep learning and paved the way to the acceptance of neural networks in general.

# Chapter 10

# Sequence Modeling: Recurrent and Recursive Nets

*Recurrent neural networks* (Rumelhart *et al.*, 1986c), or RNNs[1], are the main tool for handling sequential data, which involves variable length inputs or outputs. To go from multi-layer networks to recurrent networks, we need take advantage of one of the early ideas found in machine learning and statistical models of the 80's: *sharing parameters*[2] across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when, like in speech, the input sequence can be stretched non-linearly, i.e., some parts (like vowels) may last longer in different examples. It means that the absolute time step at which an event occurs is meaningless: it only makes sense to consider the event in some context that somehow captures what has happened before. Compared to a multi-layer network, the weights in an RNN are shared across different instances of the artificial neurons, each associated with different time steps. This allows us to apply the network to input sequences of different lengths because the same weights are re-used at each time step. This idea is made more explicit in the early work on *time-delay neural networks* (Lang and Hinton, 1988; Waibel *et al.*, 1989), where a fully connected network is replaced by one with local connections that are shared across different temporal instances of

[1]Unfortunately, the RNN acronym is sometimes also used for denoting Recursive Neural Networks. However, since the RNN acronym has been around for much longer, we suggest keeping this acronym for recurrent neural networks.

[2]see Section 7.8 for an introduction to the concept of parameter sharing

the hidden units. Such networks are among the ancestors of *convolutional neural networks*, covered in more detail in Section 9. Recurrent nets are described in more detail below, in Section 10.2, after introducing the key idea of unfolding a computational graph. As shown in Section 10.1 below, the computational graph (a notion previously introduced in Section 6.4.3 in the case of MLPs) associated with a recurrent network is structured like a chain. Recurrent neural networks have been generalized into *recursive neural networks*, in which the unfolded structure can be more general than a chain. With recursive networks, the unfolded network has the shape of a tree. Recursive neural networks are discussed in more detail in Section 10.6. For a good textbook on RNNs, see Graves (2012).

## 10.1 Unfolding Flow Graphs and Sharing Parameters

A flow graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Please refer to Section 6.4.3 for a general introduction. In this section we explain the idea of *unfolding* a recursive or recurrent computation into a flow graph that has a repetitive structure, typically corresponding to a chain of events.

For example, consider the classical form of a dynamical system:

$$\boldsymbol{s}_t = f_\theta(\boldsymbol{s}_{t-1}) \tag{10.1}$$

where $\boldsymbol{s}_t$ is called the state of the system. The unfolded flow graph of such a system looks like in Fig. 10.1.

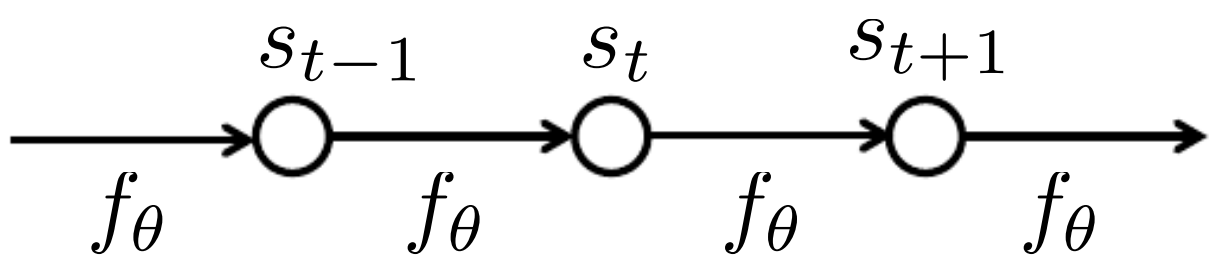


Figure 10.1: Classical dynamical system equation 10.1 illustrated as an unfolded flow graph. Each node represents the state at some time $t$ and function $f_\theta$ maps the state at $t$ to the state at $t+1$. The same parameters (the same function $f_\theta$) is used for all time steps.

As another example, let us consider a dynamical system driven by an external signal $\boldsymbol{x}_t$,

$$\boldsymbol{s}_t = f_\theta(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t) \tag{10.2}$$

illustrated in Fig. 10.2, where we see that the state now contains information about the whole past sequence, i.e., the above equation implicitly defines a function

$$\boldsymbol{s}_t = g_t(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1) \tag{10.3}$$

which maps the whole past sequence $(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1)$ to the current state. Equation 10.2 is actually part of the definition of a recurrent net. We can think of $\boldsymbol{s}_t$ as a kind of summary of the past sequence of inputs up to $t$. Note that this summary is in general necessarily lossy, since it maps an arbitrary length sequence $(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1)$ to a fixed length vector $\boldsymbol{s}_t$. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects. For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, it may not be necessary to distinctly keep track of all the bits of information, only those required to predict the rest of the sentence. The most demanding situation is when we ask $\boldsymbol{s}_t$ to be rich enough to allow one to approximately recover the input sequence, as in auto-encoder frameworks (Chapter 15).

If we had to define a different function $g_t$ for each possible sequence length (imagine a separate neural network, each with a different input size), each with its own parameters, we would not get any generalization to sequences of a size not seen in the training set. Furthermore, one would need to see a lot more training examples, because a separate model would have to be trained for each sequence length, and it would need a lot more parameters (proportionally to the size of the input sequence). It could not generalize what it learns from what happens at a position $t$ to what could happen at a position $t' \neq t$. By instead defining the state through a recurrent formulation as in Eq. 10.2, the same parameters are used for any sequence length, allowing much better generalization properties.

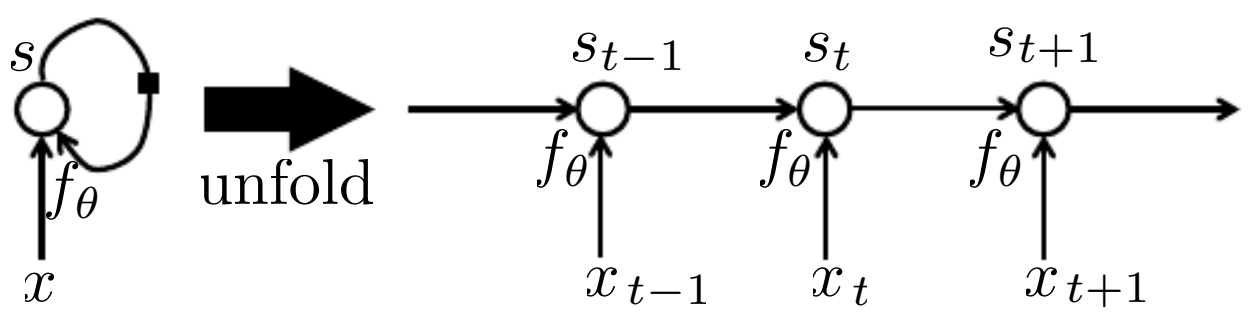


Figure 10.2: Left: input processing part of a recurrent neural network, seen as a circuit. The black square indicates a delay of 1 time step. Right: the same seen as an unfolded flow graph, where each node is now associated with one particular time instance.

Equation 10.2 can be drawn in two different ways. One is in a way that is inspired by how a physical implementation (such as a real neural network) might look like, i.e., like a circuit which operates in real time, as in the left of Fig. 10.2. The other is as a flow graph, in which the computations occurring at different time steps in the circuit are unfolded as different nodes of the flow graph, as in the right of Fig. 10.2. What we call *unfolding* is the operation that maps a circuit as in the left side of the figure to a flow graph with repeated pieces as

in the right side. Note how the unfolded graph now has a size that depends on the sequence length. The black square indicates a delay of 1 time step on the recurrent connection, from the state at time $t$ to the state at time $t+1$.

The other important observation to make from Fig. 10.2 is that *the same parameters ($\theta$) are shared* over different parts of the graph, corresponding here to different time steps.

## 10.2 Recurrent Neural Networks

Armed with the ideas introduced in the previous section, we can design a wide variety of recurrent circuits, which are compact and simple to understand visually. As we will explain, we can automatically obtain their equivalent unfolded graph, which are useful computationally and also help focus on the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients).

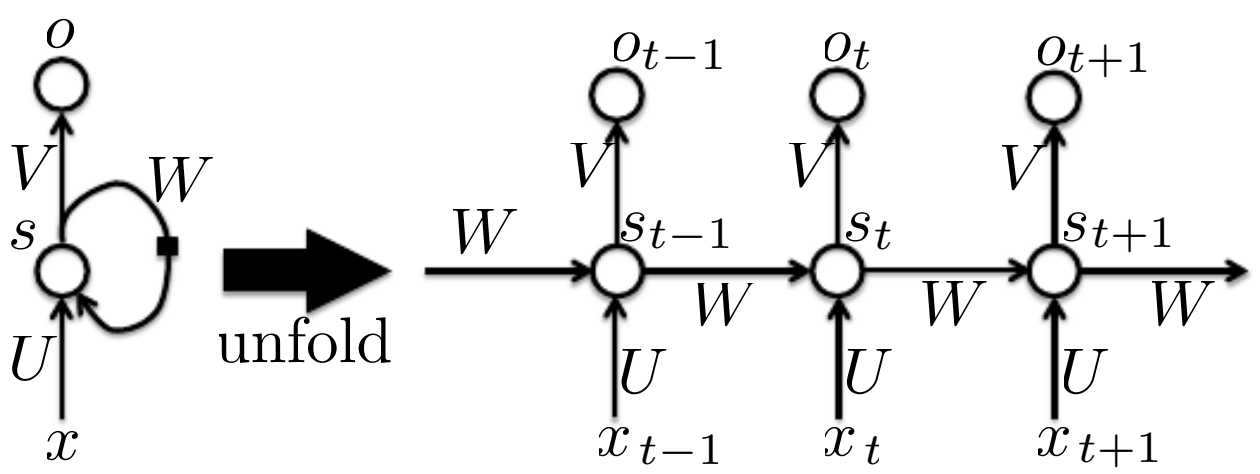


Figure 10.3: Left: ordinary recurrent network circuit with hidden-to-hidden recurrence, seen as a circuit, with weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$ for the three different kinds of connections (input-to-hidden, hidden-to-output and hidden-to-hidden, respectively). Each circle indicates a whole vector of activations. Right: the same seen as an time-unfolded flow graph, where each node is now associated with one particular time instance.

Some of the early circuit designs for recurrent neural networks are illustrated in Figs. 10.3, 10.4 and 10.6. Fig. 10.3 shows the simplest recurrent network architecture, whose equations are laid down below in Eq. 10.4. This kind of neural network has been shown to be a universal approximation machine for discrete sequences. This is loosely analogous to the universal approximator theorem for feedforward neural networks. Specifically, any function computable by a Turing machine can be computed by a recurrent network of a finite size. The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995; Hyotyniemi, 1996). Note that the functions

computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The outputs of the RNN must be discretized so that it can take a binary sequence in input and produce a binary sequence in output. Note also that the "input" of the Turing machine is a specification of the function to be computed, which is why only a finite-size recurrent net (Siegelmann and Sontag (1995) use 886 units) is sufficient for all problems. The RNN can simulate an unbounded stack by representing its activations and weights with rational numbers of unbounded precision.

On the other hand, the network with *output recurrence* (shown in Figure 10.4) and no hidden-to-hidden recurrence is strictly less powerful, since it requires that the output units capture the full state, which means that they represent an appropriate summary of the past as necessary to predict the future. Although this may be appropriate in some cases where the full state of the system is observed and can be provided as a target, this assumption is generally too strong. The advantage of this assumption, though, is that with the maximum likelihood criterion, all the time steps are decoupled and no back-propagation through time is necessary.

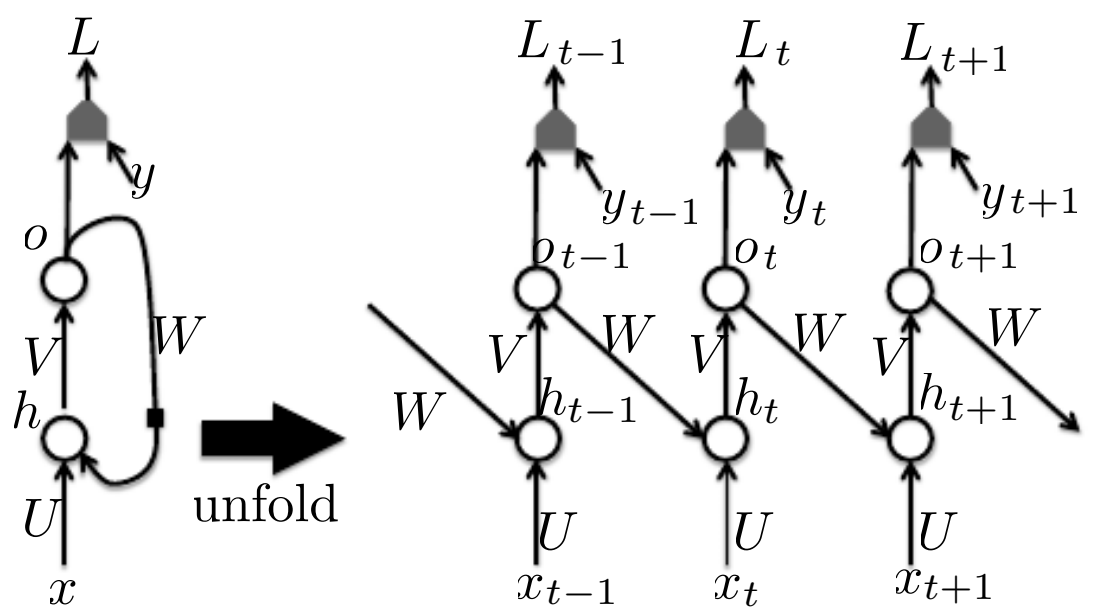


Figure 10.4: Left: RNN circuit whose recurrence is only through the output. Right: computational flow graph unfolded in time. At each $t$, the input is $\boldsymbol{x}_t$, the hidden layer activations $\boldsymbol{h}_t$, the output $\boldsymbol{o}_t$, the target $y_t$ and the loss $L_t$. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by Fig. 10.3 but may be easier to train because they can exploit "teacher forcing", i.e., constraining some of the units involved in the recurrent loop (here the output units) to take some target values during training. This architecture is less powerful because the only state information (carrying the information about the past) is the previous *prediction*. Unless the prediction is very high-dimensional and rich, this will usually miss important information from the past.

The recurrent network of Fig. 10.3 corresponds to the following forward propagation equations, if we assume that hyperbolic tangent non-linearities are used

in the hidden units and softmax is used in output (for classification problems):

$$\begin{aligned} \boldsymbol{a}_t &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t \\ \boldsymbol{s}_t &= \tanh(\boldsymbol{a}_t) \\ \boldsymbol{o}_t &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}_t \\ \boldsymbol{p}_t &= \text{softmax}(\boldsymbol{o}_t) \end{aligned} \tag{10.4}$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given input/target sequence pair $(\boldsymbol{x}, \boldsymbol{y})$ would then be just the sum of the losses over all the time steps, e.g.,

$$L(\boldsymbol{x}, \boldsymbol{y}) = \sum_t L_t = \sum_t -\log p_{t,y_t} \tag{10.5}$$

where $y_t$ is the category that should be associated with time step $t$ in the output sequence.

The circuit in Fig. 10.4 has only its output (the prediction of the previous target) as state, as a memory of the past. This potentially limits its expressive power, but also makes it easier to train. Indeed, the "intermediate state" of the corresponding unfolded deep network is not hidden anymore: targets are available to guide this intermediate representation, which should make it easier to train. In general, the state of the RNN must be sufficiently rich to store a summary of the past sequence that is enough to properly predict the future target values. Constraining the state to be the visible variable $y_t$ itself is therefore in general not enough to learn most tasks of interest, unless, given the sequence of inputs $\boldsymbol{x}_t$, $y_t$ contains all the required information about the past $y$'s that is required to predict the future $y$'s.

*Teacher forcing* is the training process in which the fed back inputs are not the predicted outputs but the targets themselves, as illustrated in Fig. 10.5. The disadvantage of strict teacher forcing is that if the network is going to be later used in an *open-loop* mode, i.e., with the network outputs (or samples from the output distribution) fed back as input, then the kind of inputs that the network will have seen during training could be quite different from the kind of inputs that it will see at test time when the network is run in generative mode, potentially yielding very poor generalizations. One way to mitigate this problem is to train with both teacher-forced inputs and with free-running inputs, e.g., predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths. In this way, the network can learn to take into account

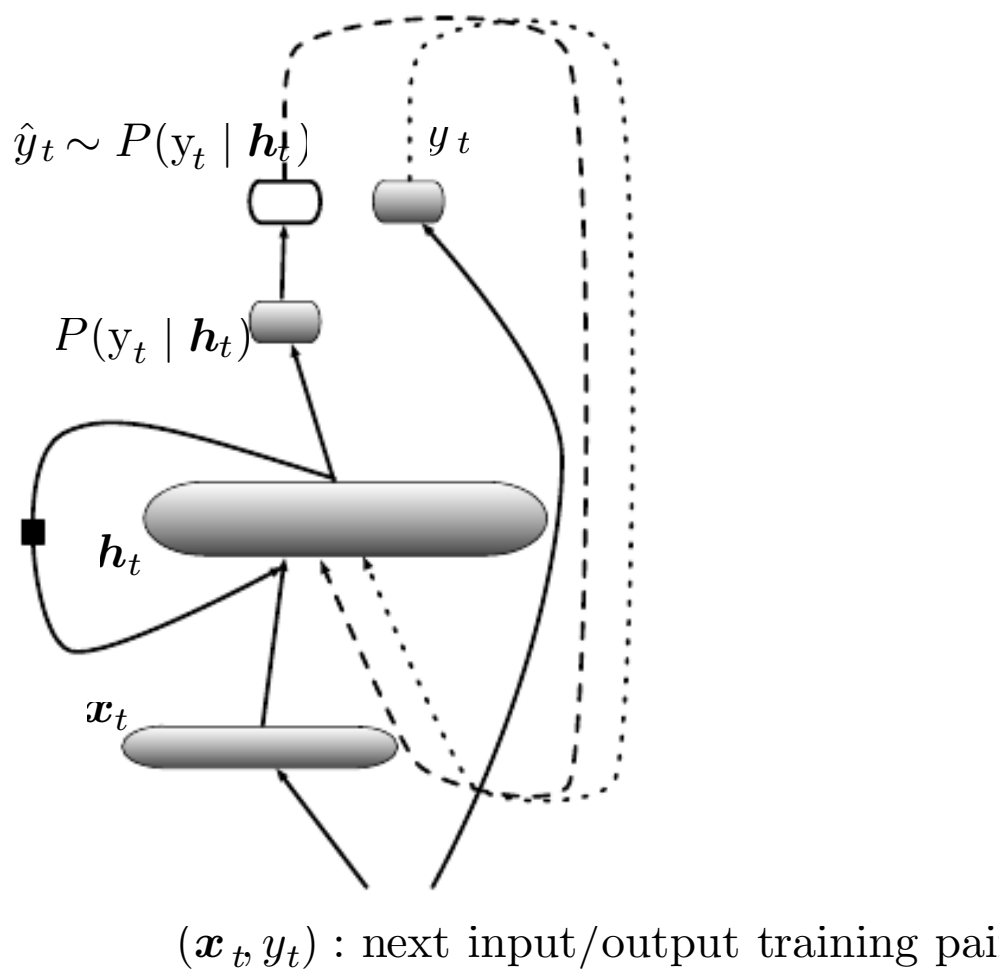


Figure 10.5: Illustration of *teacher forcing* for RNNs, which comes out naturally from the log-likelihood training objective (such as in Eq. 10.5). There are two ways in which the output variable can be fed back as input to update the next state $\boldsymbol{h}_t$: what is fed back is either the sample $\hat{y}_t$ generated from the RNN model's output distribution $P(\mathrm{y}_t \mid \boldsymbol{h}_t)$ (dashed arrow) or the actual "correct" output $y_t$ coming from the training data (dotted arrow) $(\boldsymbol{x}_t, y_t)$. The former is what is done when one *generates a sequence* from the model. The latter is teacher forcing, done during training.

input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back towards one that will make the network generate proper outputs after a few steps. Another approach (Bengio *et al.*, 2015) to mitigate the gap between the generative mode of RNNs and how they are trained (with teacher forcing, i.e., maximum likelihood) randomly chooses to use generated values or actual data values as input. This approach exploits a curriculum learning strategy to gradually use more of the generated values as input.

### 10.2.1 Computing the Gradient in a Recurrent Neural Network

Using the generalized back-propagation algorithm (for arbitrary flow graphs) introduced in Section 6.4.3, one can obtain the so-called **Back-Propagation Through Time** (BPTT) algorithm. Once we know how to compute gradients, we can in principle apply any of the general-purpose gradient-based techniques to train an RNN. These general-purpose techniques were introduced in Section 4.3 and developed in greater depth in Chapter 8.

Let us thus work out how to compute gradients by BPTT for the RNN equations above (Eqs. 10.4 and 10.5). The nodes of our flow graph will include the

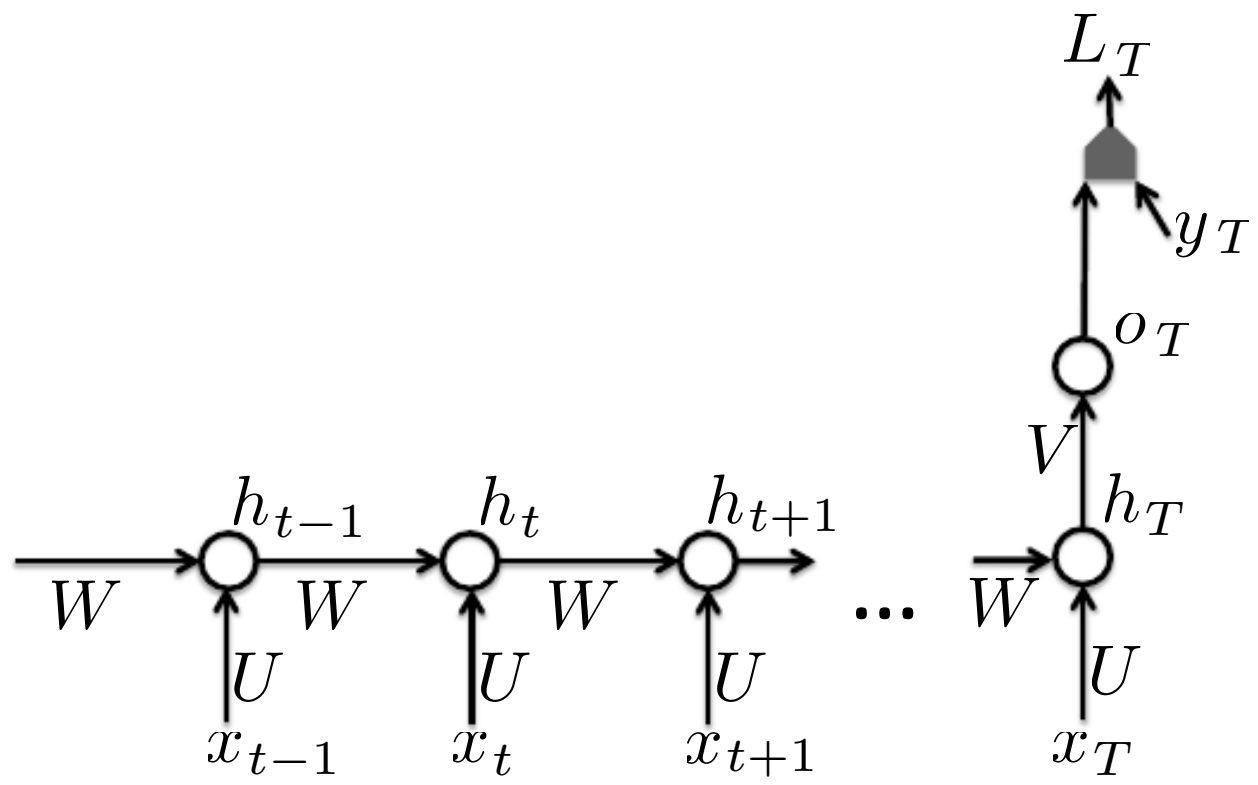


Figure 10.6: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (like in the figure) or the gradient on the output $\boldsymbol{o}_t$ can be obtained by back-propagating from further downstream modules.

parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by $t$ for $\boldsymbol{x}_t$, $\boldsymbol{s}_t$, $\boldsymbol{o}_t$ and $L_t$. For each node $a$ we need to compute the gradient $\nabla_{\boldsymbol{a}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L_t} = 1$$

and the gradient $\nabla_{\boldsymbol{o}_t} L$ on the outputs at time step $t$, for all $i, t$, is as follows:

$$(\nabla_{\boldsymbol{o}_t} L)_i = \frac{\partial L}{\partial o_{ti}} = \frac{\partial L}{\partial L_t} \frac{\partial L_t}{\partial o_{ti}} = p_{t,i} - \mathbf{1}_{i,y_t}$$

and work our way backwards, starting from the end of the sequence, say $T$, at which point $\boldsymbol{s}_T$ only has $\boldsymbol{o}_T$ as descendent:

$$\nabla_{\boldsymbol{s}_T} L = \nabla_{\boldsymbol{o}_T} L \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{s}_T} = \nabla_{\boldsymbol{o}_T} L \, \boldsymbol{V}.$$

Note how the above equation is vector-wise and corresponds to $\frac{\partial L}{\partial s_{Tj}} = \sum_i \frac{\partial L}{\partial o_{Ti}} V_{ij}$, scalar-wise. We can then iterate backwards in time to back-propagate gradients through time, from $t = T - 1$ down to $t = 1$, noting that $\boldsymbol{s}_t$ (for $t < T$) has as descendents both $\boldsymbol{o}_t$ and $\boldsymbol{s}_{t+1}$:

$$\nabla_{\boldsymbol{s}_t} L = \nabla_{\boldsymbol{s}_{t+1}} L \frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t} + \nabla_{\boldsymbol{o}_t} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{s}_t} = \nabla_{\boldsymbol{s}_{t+1}} L \, \mathrm{diag}(1 - \boldsymbol{s}_{t+1}^2) \boldsymbol{W} + \nabla_{\boldsymbol{o}_t} L \, \boldsymbol{V}$$

where $\mathrm{diag}(1 - \boldsymbol{s}_{t+1}^2)$ indicates the diagonal matrix containing the elements $1 - \boldsymbol{s}_{t+1,i}^2$, i.e., the derivative of the hyperbolic tangent associated with the hidden unit $i$ at time $t+1$.

Once the gradients on the internal nodes of the flow graph are obtained, we can obtain the gradients on the parameter nodes, which have descendents at all the time steps:

$$\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \nabla_{\boldsymbol{o}_t} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{c}} = \sum_t \nabla_{\boldsymbol{o}_t} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \nabla_{\boldsymbol{s}_t} L \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{b}} = \sum_t \nabla_{\boldsymbol{s}_t} L \,\mathrm{diag}(1 - \boldsymbol{s}_t^2) \\
\nabla_{\boldsymbol{V}} L &= \sum_t \nabla_{\boldsymbol{o}_t} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{V}} = \sum_t \nabla_{\boldsymbol{o}_t} L \, \boldsymbol{s}_t^\top \\
\nabla_{\boldsymbol{W}} L &= \sum_t \nabla_{\boldsymbol{s}_t} \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{W}} = \sum_t \nabla_{\boldsymbol{s}_t} L \,\mathrm{diag}(1 - \boldsymbol{s}_t^2) s_{t-1}^\top
\end{aligned}$$

Note in the above (and elsewhere) that whereas $\nabla_{\boldsymbol{s}_t} L$ refers to the full influence of $\boldsymbol{s}_t$ through all paths from $\boldsymbol{s}_t$ to $L$, $\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{W}}$ or $\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{b}}$ refers to the immediate effect of the denominator on the numerator, i.e., when we consider the denominator as a parent of the numerator and only that direct dependency is accounted for. Otherwise, we would get "double counting" of derivatives.

### 10.2.2 Recurrent Networks as Directed Graphical Models

Up to here, we have not clearly stated what the losses $L_t$ associated with the outputs $o_t$ of a recurrent net should be. We have delayed this discussion because there are many possible ways to use RNNs. In this section, we describe the most common case, where the RNN models a probability distribution over a sequence of observations.

When we use a predictive log-likelihood training objective, such as Eq. 10.5, we train the RNN to estimate the conditional distribution of the next sequence element $y_t$ given the past inputs. The conditioning input sequence $\{\boldsymbol{x}_\tau\}_{\tau<t}$ may contain the past values of $y$: we condition on past values of $y$ to predict future values of $y$. This is a natural consequence of a probabilistic interpretation of modeling the sequence of $y$'s, explained below. Sometimes, we include in the conditioning information not just the past $y$'s but also other conditioning variables. Decomposing the joint probability over the sequence of $y$'s as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence. When we do not feed past $y$'s as inputs that condition the next step prediction, the directed graphical model contains no edges from any $y_\tau$

in the past to the current $y_t$. In this case, the outputs $y$ are conditionally independent given the sequence of $\boldsymbol{x}$'s. When we do feed the actual $y$ values (not their prediction, but the actual observed or generated values) back into the network, the directed graphical model contains edges from all $y_\tau$ values in the past to the current $y_t$ value.

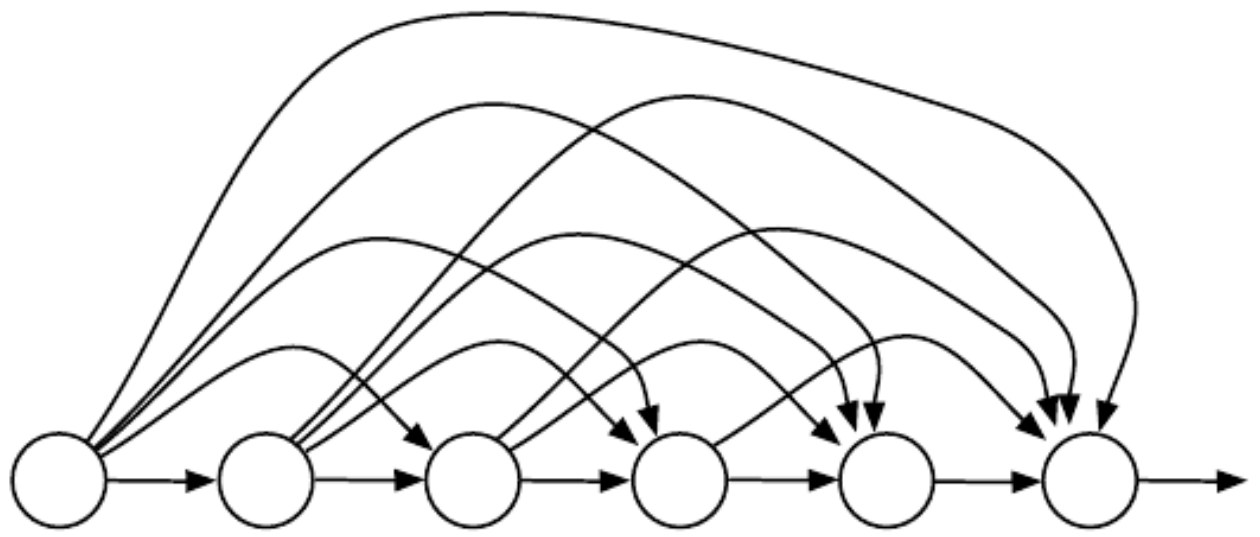

Figure 10.7: Fully connected graphical model for a sequence $y\ , y\ \ , \ldots, y_t, \ldots$: every past observation $y_\tau$ may influence the conditional distribution of some $y_t$ (for $t > \tau$), given the previous values. Parametrizing the graphical model directly according to this graph (as in Eq. 10.3) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in Fig. 10.8.

As a simple example, let us first consider the case where $\boldsymbol{x}_\tau$ only contains $y_\tau$: the target output at the next time is also the next input. The RNN then defines a directed graphical model over the random variables $\mathbf{y} = (\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_T)$. We parametrize the joint distribution of these observations using the chain rule (Eq. 3.1) for conditional probabilities:

$$P(\mathbf{y}) = P(\mathrm{y}_1, \ldots, \mathrm{y}_T) = \prod_{t=1}^{T} P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \mathrm{y}_{t-2}, \ldots, \mathrm{y}_1) \tag{10.6}$$

where the right-hand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of $\boldsymbol{y}$ according to such a model is

$$L = \sum_t L_t$$

where

$$L_t = -\log P(\mathrm{y}_t = y_t \mid y_{t-1}, y_{t-2}, \ldots, y_1).$$

Manipulating the structure (presence of arcs between nodes) of graphical models allow us to describe probability distributions in which some variables are conditionally independent. For example, a subset of $(\mathrm{y}_1, \ldots, \mathrm{y}_{t-1})$ might provide all

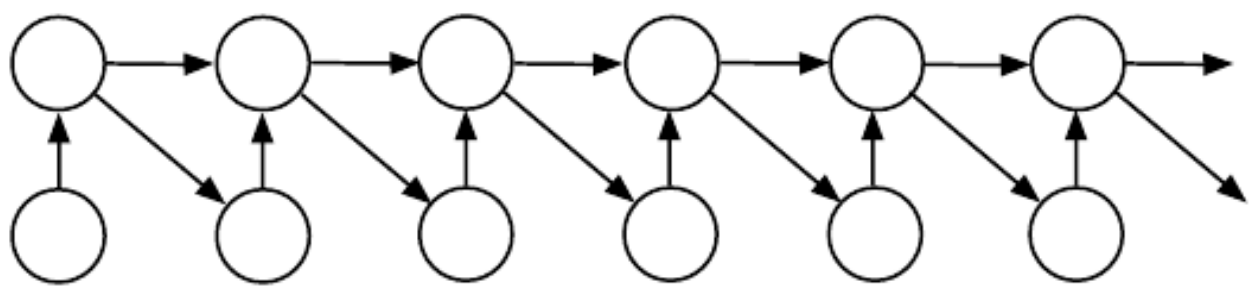

Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on Eq. 10.2. Every stage in the sequence (for $\boldsymbol{s}_t$ and $y_t$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

the information one can get to predict $\mathrm{y}_t$ given $(\mathrm{y}_1, \ldots, \mathrm{y}_{t-1})$. However, in some cases, we believe that all past inputs should have an influence on the next element of the sequence, and not say, the most recent ones. In that case the graphical model would be fully connected, as in Figure 10.7. RNNs as directed graphical models have such a fully connected structure, when we ignore (i.e., marginalize over) the intermediate states. However, it is more interesting to consider the graphical model structure of RNNs when we introduce a particular kind of latent variable $\boldsymbol{s}_t$ (that is deterministic in the past observations). This results in a very efficient parametrization of the joint distribution over the observations, because rather than having the number of parameters grow exponentially with time – as would be the case for an arbitrary joint distribution – due to parameter sharing the number of parameters in the RNN in actually constant w.r.t. time. This is illustrated by Eq. 10.2 and Figure 10.8. The model is trained so that the variable $\boldsymbol{s}_t$ summarizes whatever is required from the whole previous sequence (Eq. 10.3) to predict the next step. Some operations such as predicting missing values in the middle of the sequence remain computationally inefficient despite the recurrent network parameterization. If we incorporate the $\boldsymbol{s}_t$ nodes in the graphical model, we see that it decouples that past and the future, acting as an intermediate quantity between them. It also makes the graph locally connected (each state $\boldsymbol{s}_t$ only depends the previous $\boldsymbol{s}_{t-1}$ and on the current input) whereas the original graphical model (without nodes for the state) is fully connected.

The statistical complexity of the model may then be flexibly adjusted by changing the complexity of the function producing the state variable. Crucially, the complexity of this function (as measured by the number of free parameters) is not forced to increase with sequence length. This stands in contrast to the approach of controlling statistical complexity by modifying the graph structure of a directed graphical model. A graphical model defined over discrete variables using a table to implement the conditional probability distribution $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \mathrm{y}_{t-2}, \ldots, \mathrm{y}_1)$ would require an amount of parameters (table entries)

that grows exponentially with $t$. The graphical model of recurrent networks is directed and decomposes the probability distribution as a product of conditionals without explicitly cutting any arc in the graphical model. The recurrent net uses the same graphical model structure, but defines this conditional probability distribution via Eq. 10.2, using a number of parameters that is constant in $t$. The capacity of the model is reduced by parametrizing the transition probability in a recursive way that requires a fixed (and not exponential) number of parameters, due to a form of parameter sharing (see Section 7.8 for an introduction to the concept). Instead of reducing $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \ldots, \mathrm{y}_1)$ to something like $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \ldots, \mathrm{y}_{t-k})$ (assuming the $k$ previous ones as the parents), we keep the full dependency but we parametrize the conditional efficiently in a way that does not grow with $t$, exploiting parameter sharing. The price recurrent networks pay for their reduced number of parameters is that *optimizing* the parameters may be difficult, as discussed below (Sections 8.2.6 and 10.7).

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps: the above conditional probability distribution is *stationary*, i.e., the relation between the past and the next observation does not depend on $t$, only on the values of the past observations. It allows one to use the same model for sequences of different lengths. In principle, it would be possible to use $t$ as an extra input at each time step and let the learner discover any time-dependence while sharing as much as it can between different time steps. This would already be much better than using a different conditional probability distribution for each $t$, but the network would then have to extrapolate when faced with new values of $t$.

Let us consider how that stationarity assumption can be turned into an efficient recursive parametrization. When we introduce a function $g_t$ that summarizes the past inputs and a recursive function $f_\theta$ to implement it, the decomposition of the likelihood over a sequence of vectors $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$ is

$$P(\boldsymbol{X}) = \prod_{t=1}^{T} P(\boldsymbol{x}_t \mid g_t(\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_1))$$

where

$$\boldsymbol{s}_t = g_t(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1) = f_\theta(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t).$$

Note that if the self-recurrence function $f_\theta$ is *learned*, it can discard some of the information in some of the past values $\boldsymbol{x}_{t-k}$ that are not needed for predicting the future data. In fact, because the state generally has a fixed dimension smaller than the length of the sequences (times the dimension of the input), it *has to* discard some information. However, we leave it to the learning procedure to choose what information to keep and what information to throw away, so as minimize the objective function (e.g., predict future values correctly).

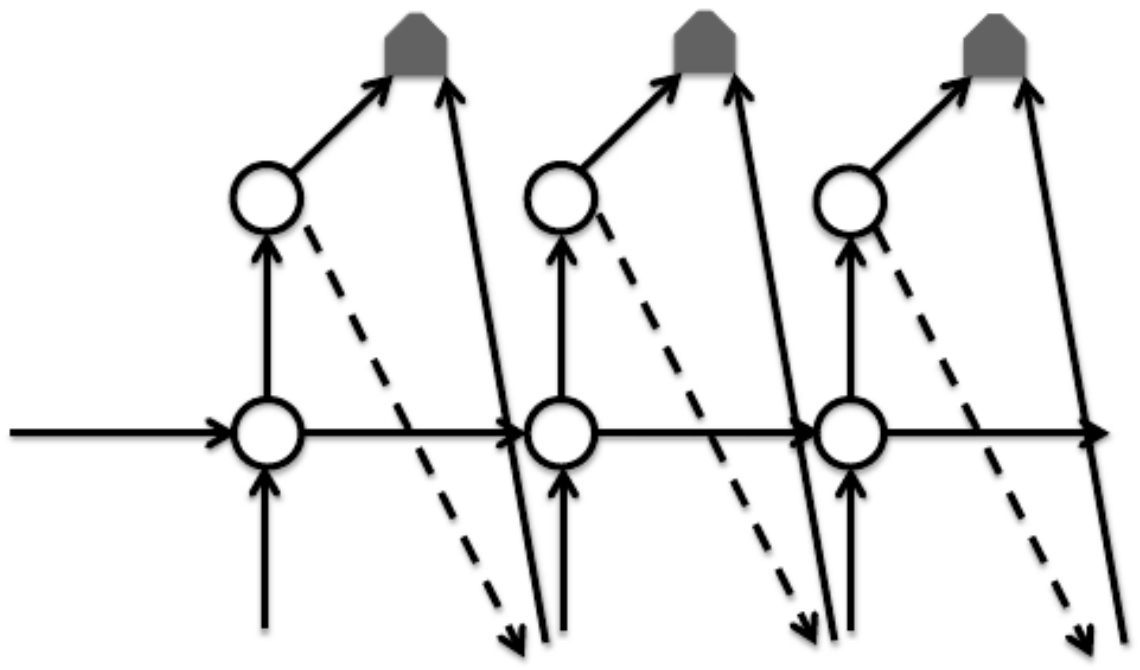

Figure 10.9: A recurrent neural network modeling $P(\boldsymbol{x}, \ldots, \boldsymbol{x}_T)$, able to generate sequences from this distribution. Each element $\boldsymbol{x}_t$ of the observed sequence serves both as input (for computing the state $\boldsymbol{s}_t$ at the current time step) and as target (for the prediction made at the previous time step). The output $\boldsymbol{o}_t$ encodes the parameters of a conditional distribution $P(\boldsymbol{x}_t \mid \boldsymbol{x}, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_t \mid \boldsymbol{o}_t)$ for $\boldsymbol{x}_t$, given the past sequence $\boldsymbol{x} \ldots, \boldsymbol{x}_t$. The loss $L_t$ is the negative log-likelihood associated with the output prediction (or more generally, distribution parameters) $\boldsymbol{o}_t$, when the actual observed target is $\boldsymbol{x}_t$. In training mode, one measures and minimizes the sum of the losses over observed sequence(s) $\boldsymbol{x}$. In generative mode, $\boldsymbol{x}_t$ is sampled from the conditional distribution $P(\boldsymbol{x}_t \mid \boldsymbol{x}, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_t \mid \boldsymbol{o}_t)$ (dashed arrows) and then that generated sample $\boldsymbol{x}_t$ is fed back as input for computing the next state $\boldsymbol{s}_t$, the next output $\boldsymbol{o}_t$, and generating the next sample $\boldsymbol{x}_t$, etc. The length of the sequence must also be generated (unless known in advance). This could be done by a special sigmoidal output unit that predicts a binary target (with associated cross-entropy loss) that encodes the fact that the next output is the last.

The above decompositions of the joint probability of a sequence of variables into ordered conditionals precisely corresponds to the sequence of computations performed by an RNN. The *target* to be predicted at each time step $t$ is the next element in the sequence, while the *input* at each time step is the previous element in the sequence (with all previous inputs summarized in the state), and the *output* is interpreted as parametrizing the probability distribution of the target given the state. This is illustrated in Fig. 10.9.

If the RNN is actually going to be used to generate sequences, one must also incorporate in the output information allowing to stochastically decide when to stop generating new output elements. This can be achieved in various ways. In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence. When that symbol is generated, a complete sequence has been generated. The target for that special symbol occurs exactly once per sequence, as the last target after the last output element $\boldsymbol{x}_T$. In general, one may train a binomial output associated with that stopping variable, for example using a sigmoid output non-linearity and the cross entropy loss, i.e., again negative log-likelihood for the event "end of the sequence". Another kind of solution is to model the integer $T$ itself, through any reasonable parametric distribution, and use the number of time steps left (and possibly the number of time steps since the beginning of the sequence) as extra inputs at each time step. Thus we would have decomposed $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T)$ into $P(T)$ and $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T \mid T)$. In general, one must therefore keep in mind that in order to fully generate a sequence we must not only generate the $\boldsymbol{x}_t$'s, but also the sequence length $T$, either implicitly through a series of continue/stop decisions (or a special "end-of-sequence" symbol), or explicitly through modeling the distribution of $T$ itself as an integer random variable. For example, the strategy of using a special end-of-sequence symbol is used when the RNN generates an output sequence, such as a sequence of words in machine translation (Kalchbrenner and Blunsom, 2013; Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014). The strategy of predicting $T$ itself is used for example by Goodfellow *et al.* (2014d).

If we take the RNN equations of the previous section (Eq. 10.4 and 10.5), they could correspond to a generative RNN if we simply make the target $\boldsymbol{y}_t$ equal to the next input $\boldsymbol{x}_{t+1}$ (and because the outputs are the result of a softmax, it must be that the input sequence is a sequence of symbols, i.e., $\boldsymbol{x}_t$ is a symbol or bounded integer).

Other types of data can clearly be modeled in a similar way, following the discussions about the encoding of outputs and the probabilistic interpretation of losses as negative log-likelihoods, in Sections 5.6 and 6.3.2.

### 10.2.3 Modeling a sequence conditioned on some context by making the generative RNN conditional

In the previous section we studied how an RNN could correspond to a directed graphical model over a sequence of random variables $\mathbf{y}$. Here we consider how one can condition this distribution on other variables. In general, as discussed in Section 6.3.2 (see especially the end of that section, in Subsection 6.3.2 ), when we can represent a parametric probability distribution $P(\mathbf{y} \mid \boldsymbol{\omega})$, we can make it conditional by making $\omega$ a function of the appropriate conditioning variable:

$$P(\mathbf{y} \mid \boldsymbol{\omega} = f(\mathbf{x})).$$

In the case of an RNN, this can be achieved in different ways. We review here the most common and obvious choices.

If $\mathbf{x}$ is a fixed-size vector, then we can simply make it an extra input of the RNN that generates the $\mathbf{y}$ sequence. Some common ways of providing an extra input to an RNN are:

1. as an extra input at each time step, or
2. as the initial state $\boldsymbol{s}_0$, or
3. both.

In general, one may need to add extra parameters (and parametrization) to map $\mathbf{x} = \boldsymbol{x}$ into the "extra bias" going either into only $\boldsymbol{s}_0$, into the other $\boldsymbol{s}_t$ $(t > 0)$, or into both. The first (and most commonly used) approach is illustrated in Fig. 10.10.

Speech is considered to be a sequence of phonemes or phonetic categories, roughly corresponding to one letter in latin alphabets. As an example, imagine that $\mathbf{x}$ is encoding the identity of a phoneme and the identity of a speaker. Let $\mathbf{y}$ represent an acoustic sequence corresponding to that phoneme, as pronounced by that speaker.

Consider the case where the input $\mathbf{x}$ is a sequence of the same length as the output sequence $\mathbf{y}$, and the $\mathbf{y}_t$'s are independent of each other when the past input sequence is given, i.e., $P(\mathbf{y}_t \mid \mathbf{y}_{t-1}, \ldots, \mathbf{y}_1, \mathbf{x}) = P(\mathbf{y}_t \mid \mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1)$. We therefore have a causal relationship between the $\mathbf{x}_t$'s and the predictions of the $\mathbf{y}_t$'s, in addition to the independence of the $\mathbf{y}_t$'s, given $\mathbf{x}$. Under these (pretty strong) assumptions, we can return to Fig. 10.3 and interpret the $t$-th output $\boldsymbol{o}_t$ as parameters for a conditional distribution for $\mathbf{y}_t$, given $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1$.

If we want to remove the conditional independence assumption, we can do so by making the past $\mathbf{y}_t$'s inputs into the state as well. That situation is illustrated in Fig. 10.11.

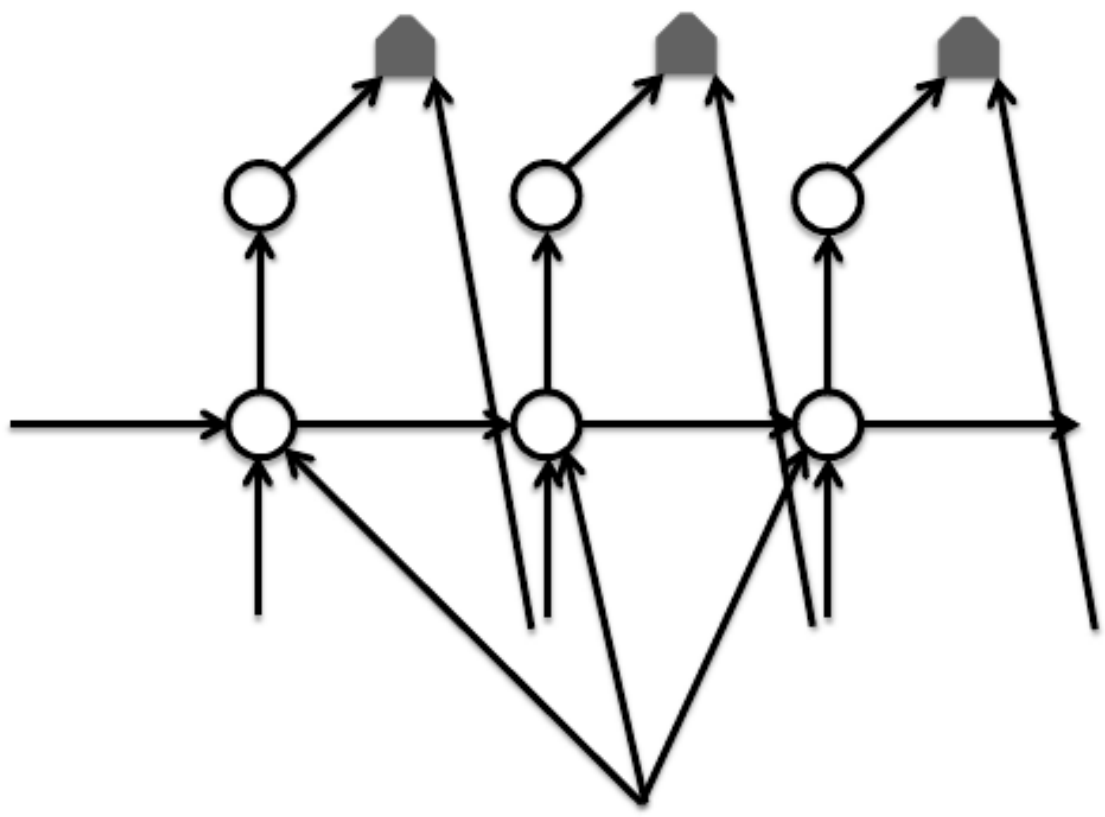

Figure 10.10: A conditional generative recurrent neural network maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. Each element $\boldsymbol{y}_t$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step). The generative semantics are the same as in the unconditional case (Fig. 10.9). There are only two differences. First, the state is now conditioned on the input $\boldsymbol{x}$. Second, the same parameters (weight matrix $\boldsymbol{R}$ in the figure) are used at every time step to parametrize that dependency.

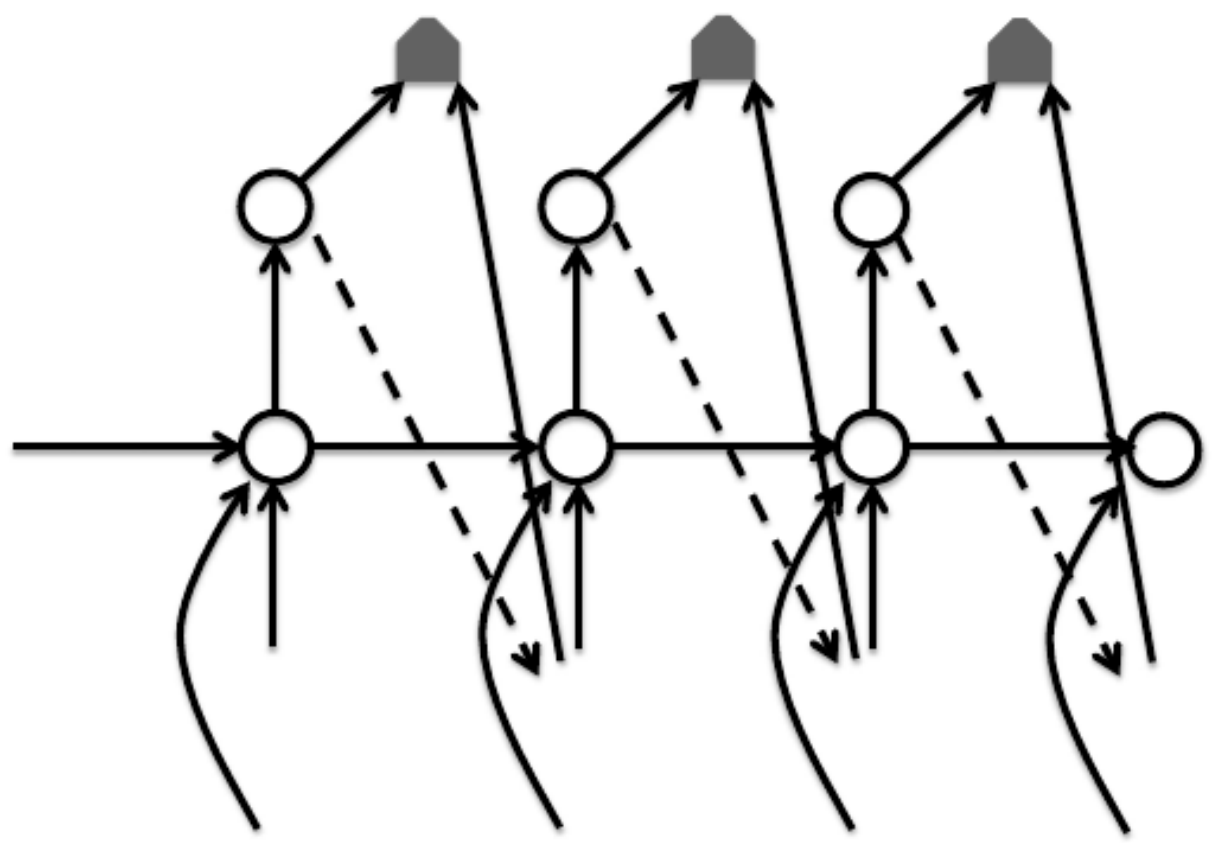

Figure 10.11: A conditional generative recurrent neural network mapping a variable-length sequence $\boldsymbol{x}$ into a distribution over sequences $\mathbf{y}$ of the same length. This architecture assumes that the predictions of $\mathbf{y}_t$'s are causally related to the $\mathbf{x}_t$'s, i.e., that we want to predict the $\mathbf{y}_t$'s only using the past $\mathbf{x}_t$'s. Note how the prediction of $\mathbf{y}_t$ is based on both the past $\mathbf{x}$'s and the past $\mathbf{y}$'s. The dashed arrows indicate that $\boldsymbol{y}_t$ can be generated by sampling from the output distribution $\boldsymbol{o}_t$. When $\boldsymbol{y}_t$ is clamped (known), it is used as a target in the loss $L_t$ which measures the log-probability that $\boldsymbol{y}_t$ would be sampled from the distribution $o_t$.

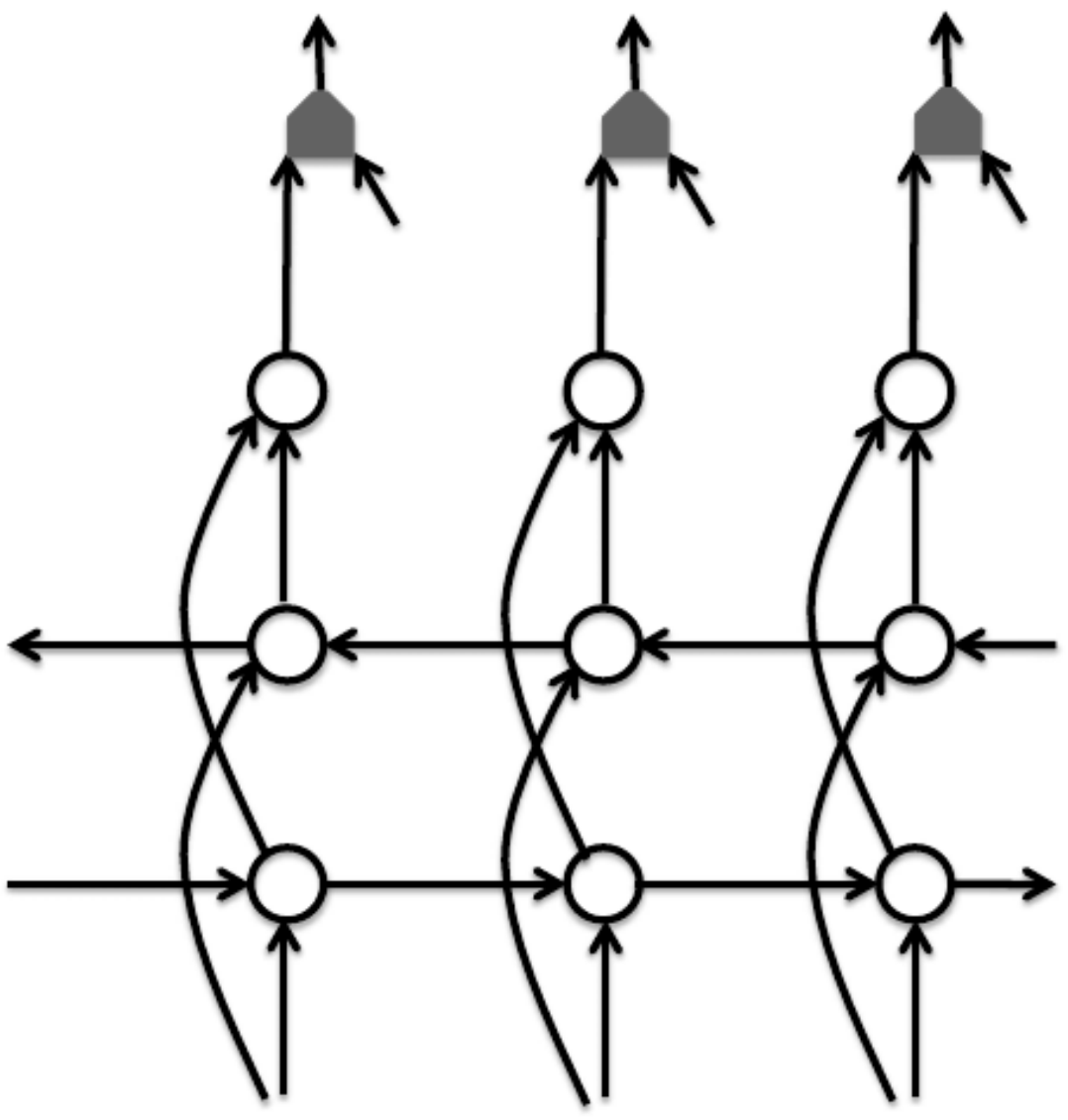

Figure 10.12: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss $L_t$ at each step $t$. The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while the $\boldsymbol{g}$ recurrence propagates information backward in time (towards the left). Thus at each point $t$, the output units $\boldsymbol{o}_t$ can benefit from a relevant summary of the past in its $\boldsymbol{h}_t$ input and from a relevant summary of the future in its $\boldsymbol{g}_t$ input.

## 10.3 Bidirectional RNNs

All of the recurrent networks we have considered up to now have a "causal" structure, meaning that the state at time $t$ only captures information from the past, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. However, in many applications we want to output at time $t$ a prediction regarding an output which may depend on *the whole input sequence.* For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them. This is also true of handwriting recognition and many other sequence-to-sequence learning tasks, described in the next section.

Bidirectional recurrent neural networks (or bidirectional RNNs) were invented to address that need (Schuster and Paliwal, 1997). They have been extremely successful (Graves, 2012) in applications where that need arises, such as handwriting recognition (Graves *et al.*, 2008; Graves and Schmidhuber, 2009), speech recognition (Graves and Schmidhuber, 2005; Graves *et al.*, 2013) and bioinformatics (Baldi *et al.*, 1999).

As the name suggests, the basic idea behind bidirectional RNNs is to combine a forward-going RNN and a backward-going RNN. Fig. 10.12 illustrates the typical bidirectional RNN, with $\boldsymbol{h}_t$ standing for the state of the forward-going RNN and $\boldsymbol{g}_t$ standing for the state of the backward-going RNN. This allows the units $\boldsymbol{o}_t$ to compute a representation that depends on *both the past and the future* but is most sensitive to the input values around time $t$, without having to specify a fixed-size window around $t$ (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

This idea can be naturally extended to 2-dimensional input, such as images, by having *four* RNNs, each one going in one of the four directions: up, down, left, right. At each point $(i, j)$ of a 2-D grid, an output $o_{i,j}$ could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN are able to learn to carry that information.

## 10.4 Encoder-Decoder Sequence-to-Sequence Architectures

We have seen in Fig. 10.6 how an RNN can map an input sequence to a fixed-size prediction. We have seen in Fig. 10.9 how an RNN can model a distribution over sequences and generate new ones from the estimated distribution. We have seen in Fig. 10.10 how one can condition on an input vector to learn to generate such

sequences. We have seen in Figures 10.11 and 10.12 how an RNN (unidirectional or bidirectional) can map an input sequence to an output sequence of the same length.

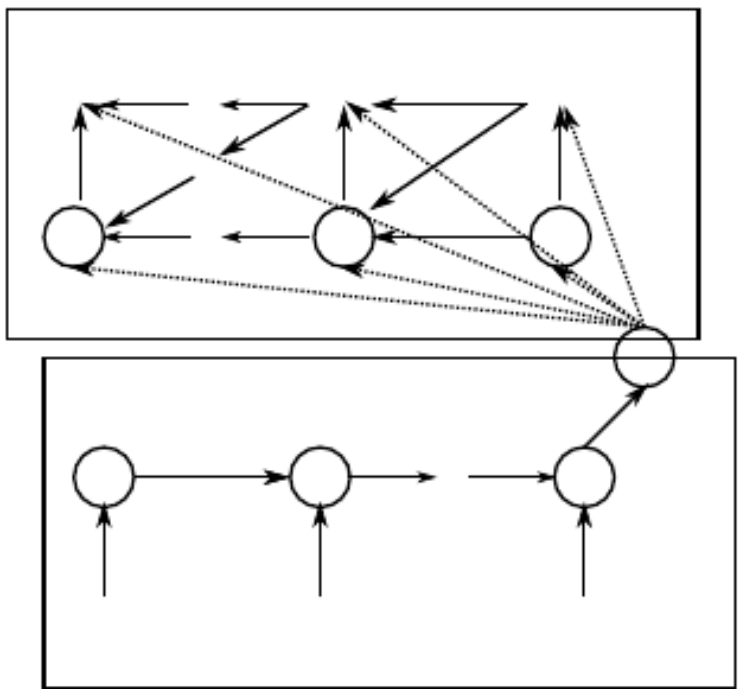

Figure 10.13: Example of encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}\ ,\ldots,\mathbf{y}_n\ )$ given an input sequence $(\mathbf{x}\ ,\mathbf{x}\ ,\ldots\mathbf{x}_n\ )$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable $C$ which represents a semantic summary of the input sequence and conditions computations in the decoder RNN.

Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation or question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related).

We often call the input to the RNN the "context." We want to produce a representation of this context, $C$. The context $C$ might be a vector or a sequence or sequence of vectors that summarize the input sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n_x})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed in Cho *et al.* (2014a) and shortly after in Sutskever *et al.* (2014b). These authors respectively called this architecture, illustrated in Fig. 10.13, the encoder-decoder or sequence-to-sequence architecture. The idea is very simple: (1) an *encoder* or *reader* or *input* RNN processes the input sequence. The encoder emits the context $C$, usually as a simple function of its final hidden state. (2) a *decoder* or *writer* or *output* RNN is conditioned on that fixed-length vector (just like in Fig. 10.10) to generate the output sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{n_y})$, where the lengths $n_x$ and $n_y$ can vary from

training pair to training pair. The two RNNs are trained jointly to maximize the average of $\log P(\mathbf{Y} = \boldsymbol{Y} | \mathbf{X} = \boldsymbol{X})$ over all the training pairs $(\boldsymbol{X}, \boldsymbol{Y})$. The last state $\boldsymbol{s}_{n_x}$ of the input RNN is typically used as a representation $C$ of the input sequence that conditions the output RNN. The output RNN can be conditioned in at least two ways, which can be combined, as we have seen earlier, i.e., either by producing a starting state for the output RNN or by producing an extra input at each time step of the output RNN. In any case, one inserts an extra set of parameters to map $C$ into a bias or initial state. Hence the two RNNs do not have to have the same hidden layer dimensionality, and sometimes it makes sense to make that mapping more complex and non-linear, e.g., an MLP could be used to map the output of the encoder RNN into an input for the decoder RNN.

One clear limitation of this architecture is when the output of the encoder RNN has a dimension that is too small to properly summarize a long sequence. This phenomenon was observed by Bahdanau *et al.* (2014) in the context of machine translation. They proposed to make $C$ a variable length sequence rather than a fixed-size vector. Additionally, they introduced an *attention mechanism* that learns to associate elements of the sequence $C$ to elements of the output sequence. See Sec. 12.4.6 for more details.

## 10.5 Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from input to hidden state,
2. from previous hidden state to next hidden state, and
3. from hidden state to output,

where the first two are actually brought together to map the input and previous state into the next state. With the RNN architecture of Fig. 10.3, each of these three blocks is associated with a single weight matrix. In other words, when the network is unfolded, each of these corresponds to a shallow transformation. By a shallow transformation, we mean a transformation that would be represented by a single layer within a deep MLP. Typically this is a transformation represented by a learned affined transformation followed by a fixed non-linearity.

Would it be advantageous to introduce depth in each of these operations? Experimental evidence (Graves *et al.*, 2013; Pascanu *et al.*, 2014a) strongly suggests so. The experimental evidence is in agreement with the idea that we need enough depth in order to perform the required mappings. See also Schmidhuber (1992); El Hihi and Bengio (1996); Jaeger (2007a) for earlier work on deep RNNs.

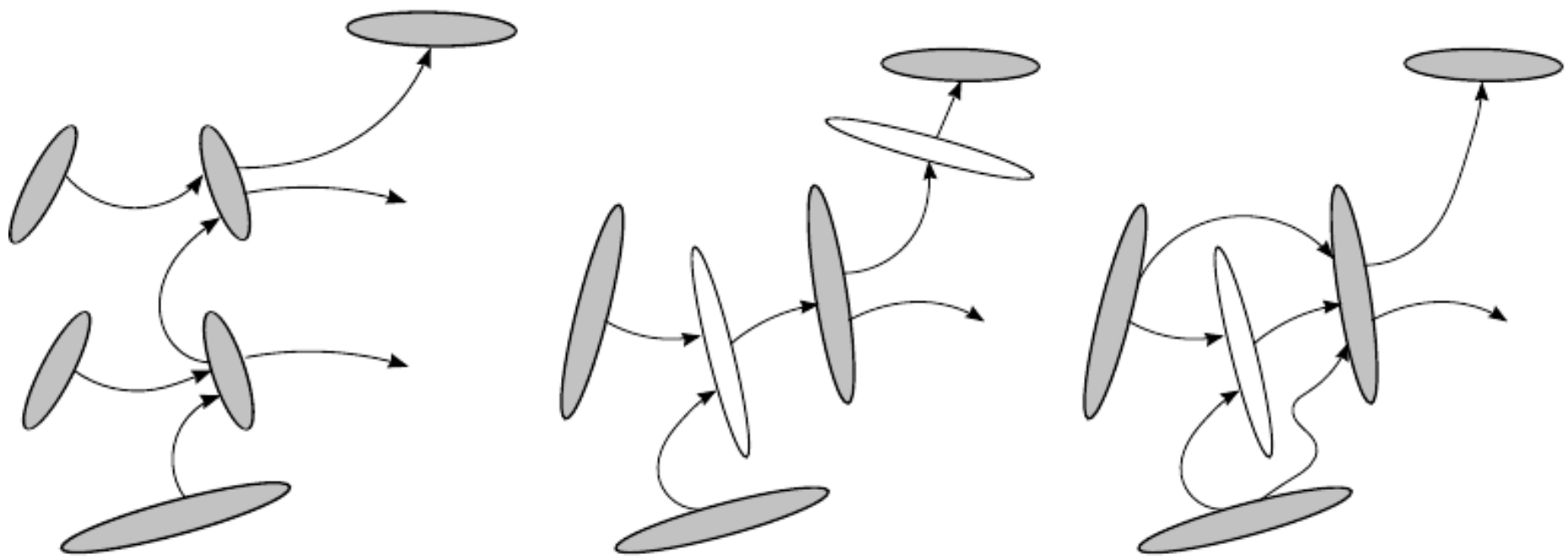

Figure 10.14: A recurrent neural network can be made deep in many ways. First, the hidden recurrent state can be broken down into groups organized hierarchically (left). Second, deeper computation (e.g., an MLP in the figure) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts (Middle). This may lengthen the shortest path linking different time steps. The path-lengthening effect can be mitigated by introduced skip connections (Right). Figures reproduced from Pascanu *et al.* (2014a) with permission.

El Hihi and Bengio (1996) first introduced the idea of decomposing the hidden state of an RNN into multiple groups of units that would operate at different time scales. Graves *et al.* (2013) were the first to show a significant benefit of decomposing the state of an RNN into groups of hidden units, with a restricted connectivity between the groups, e.g., as in Fig. 10.14 (left). Indeed, if there were no restriction at all and no pressure for some units to represent a slower time scale, then having $N$ groups of $M$ hidden units would be equivalent to having a single group of $NM$ hidden units. Koutnik *et al.* (2014) showed how the multiple time scales idea from El Hihi and Bengio (1996) can be advantageous on several sequential learning tasks: each group of hidden unit is updated at a different multiple of the time step index e.g., at every time step, at every 2nd step, at every 4th step, etc.

We can also think of the lower layers in this hierarchy as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state. Pascanu *et al.* (2014a) go a step further and propose to have a separate MLP (possibly deep) for each of the three blocks enumerated above, as illustrated in Fig. 10.14 (middle). It makes sense to allocate enough capacity in each of these three steps, but having a deep state-to-state transition may also hurt: it makes the shortest path from an event at time $t$ to an event at time $t' > t$ substantially longer, which make it more difficult to learn long-term dependencies (see Sections 8.2.6 and 10.7). For example if a one-hidden-layer MLP is used for the state-to-state transition, we have doubled the

length of that path, compared with the ordinary RNN of Fig. 10.3. However, as argued by Pascanu *et al.* (2014a), this can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in Fig. 10.14 (right).

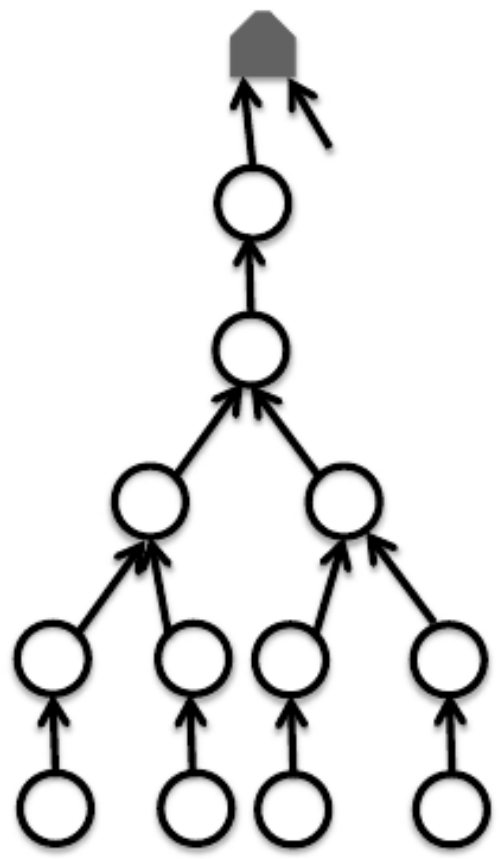

Figure 10.15: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. In the figure, a variable-size sequence $\boldsymbol{x}$ ,$\boldsymbol{x}$ ,... can be mapped to a fixed-size representation (the output $o$), with a fixed number of parameters (e.g. the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target $y$ is provided which is associated with the whole sequence.

## 10.6 Recursive Neural Networks

Recursive networks represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in Fig. 10.15. Recursive neural networks were introduced by Pollack (1990) and their potential use for learning to reason were nicely laid down by Bottou (2011). Recursive networks have been successfully applied in processing *data structures* as input to neural nets (Frasconi *et al.*, 1997, 1998), in natural language processing (Socher *et al.*, 2011a,c, 2013) as well as in computer vision (Socher *et al.*, 2011b).

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length $N$, the depth (measured as the number of compositions of non-

linear operations) can be drastically reduced from $N$ to $O(\log N)$, which might help deal with long-term dependencies. An open question is how to best structure the tree, though. One option is to have a tree structure which does not depend on the data, e.g., a balanced binary tree. Another is to use an external method, such as a natural language parser (Socher *et al.*, 2011a, 2013). Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested in Bottou (2011).

Many variants of the recursive net idea are possible. For example, in Frasconi *et al.* (1997, 1998), the data is associated with a tree structure in the first place, and inputs and/or targets with each node of the tree. The computation performed by each node does not have to be the traditional artificial neuron computation (affine transformation of all inputs followed by a monotone non-linearity). For example, Socher *et al.* (2013) propose using tensor operations and bilinear forms, which have previously been found useful to model relationships between concepts (Weston *et al.*, 2010; Bordes *et al.*, 2012) when the concepts are represented by continuous vectors (embeddings).

## 10.7 The Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks was introduced in Section 8.2.6. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared short-term ones. See Hochreiter (1991); Doya (1993); Bengio *et al.* (1994); Pascanu *et al.* (2013a) for a deeper treatment.

In this section we discuss various approaches that have been proposed to alleviate this difficulty with learning long-term dependencies.

### 10.7.1 Echo State Networks

Echo state networks are models whose weights are chosen to make the dynamics of forward propagation barely contractive.

The recurrent weights and input weights of a recurrent network are those that define the state representation captured by the model, i.e., how the state $\boldsymbol{s}_t$ (hidden units vector) at time $t$ (Eq. 10.2) captures and summarizes information from the previous inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t$. Since learning the recurrent and input weights is difficult, one option that has been proposed (Jaeger, 2003; Lee *et al.*,

2015; Maass *et al.*, 2002; Jaeger and Haas, 2004; Jaeger, 2007b) is to *set those weights such that the recurrent hidden units do a good job of capturing the history of past inputs*, and *only learn the output weights.* This is the idea that was independently proposed for *Echo State Networks* or ESNs (Jaeger and Haas, 2004; Jaeger, 2007b) and *Liquid State Machines* (Maass *et al.*, 2002). The latter is similar, except that it uses spiking neurons (with binary outputs) instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines are termed *reservoir computing* (Lukoševičius and Jaeger, 2009) to denote the fact that the hidden units form of reservoir of temporal features which may capture different aspects of the history of inputs.

One way to think about these reservoir computing recurrent networks is that they are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time $t$) into a fixed-length vector (the recurrent state $\boldsymbol{s}_t$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest. The training criterion is therefore convex in the parameters (which are just the output weights) and can actually be solved online in the linear regression case,using online updates for linear regression (Jaeger, 2003).

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to make the dynamical system associated with the recurrent net nearly be on the edge of stability, i.e., more precisely with values around 1 for the leading singular value of the Jacobian of the state-to-state transition function. As explained in 8.2.6, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $\boldsymbol{J}^{(t)} = \frac{\partial s_t}{\partial s_{t\ 1}}$. Of particular importance is the *spectral radius* of $\boldsymbol{J}^{(t)}$, defined to be the maximum of the absolute values of its eigenvalues (which can be complex-valued).

To understand the effect of the spectral radius, consider the simple case of back-propagation with a the Jacobian matrix $\boldsymbol{J}$ that does not change with $t$. This case happens, for example, when the network is purely linear. Suppose that $\boldsymbol{J}$ has an eigenvector $\boldsymbol{v}$ with corresponding eigenvalue $\lambda$. Consider what happens as we propagate a gradient vector backwards through time. If we begin with a gradient vector $\boldsymbol{g}$, then after one step of back-propagation, we will have $\boldsymbol{J}\boldsymbol{g}$, and after $n$ steps we will have $\boldsymbol{J}^n\boldsymbol{g}$. Now consider what happens if we instead back-propagate a perturbed version of $\boldsymbol{g}$. If we begin with $\boldsymbol{g}+\delta\boldsymbol{v}$, then after one step, we will have $\boldsymbol{J}(\boldsymbol{g}+\delta\boldsymbol{v})$. After $n$ steps, we will have $\boldsymbol{J}^n(\boldsymbol{g}+\delta\boldsymbol{v})$. From this we can see that back-propagation starting from $\boldsymbol{g}$ and back-propagation starting from $\boldsymbol{g}+\delta\boldsymbol{v}$ diverge by $\delta\boldsymbol{J}^n\boldsymbol{v}$ after $n$ steps of back-propagation. If $\boldsymbol{v}$ is chosen to be a unit eigenvector of $\boldsymbol{J}$ with eigenvalue $\lambda$, then multiplication by the Jacobian simply

scales the difference at each step. The two executions of back-propagation are separated by a distance of $\delta|\lambda|^n$. When $\boldsymbol{v}$ corresponds to the largest value of $|\lambda|$, this perturbation achieves the widest possible separation of an initial perturbation of size $\delta$.

When $|lambda| > 1$, the deviation size $\delta|\lambda|^n$ grows exponentially large. When $|\lambda| < 1$, the deviation size becomes exponentially small.

Of course, this example assumed that the Jacobian was the same at every time step, corresponding to a recurrent network with no non-linearity. Everything we have said about back-propagation in such a setting applies equally to forward propagation in a network with no non-linearity, where the state $\boldsymbol{s}^{(t+1)} = \boldsymbol{s}^{(t)\top}\boldsymbol{W}$.

When the spectral radius is less than one, we say that the mapping from $\boldsymbol{s}^{(t)}$ to $\boldsymbol{s}^{(t+1)}$ is *contractive*—a small changes gets *contracted*, becoming smaller after each time step. This necessarily makes the network forget information about the past when we use a finite level of precision (such as 32 bit integers) to store the state vector.

The Jacobian matrix tells us how a small change of $\boldsymbol{s}_t$ propagates one step forward, or equivalently, how the gradient on $\boldsymbol{s}_{t+1}$ propagates one step backward, during back-propagation. Note that neither $\boldsymbol{W}$ nor $\boldsymbol{J}$ need to be symmetric (although they are square and real), so they can have complex-valued eigenvalues and eigenvectors, with imaginary components corresponding to potentially oscillatory behavior (if the same Jacobian was applied iteratively). Even though $\boldsymbol{s}^{(t)}$ or a small variation of $\boldsymbol{s}^{(t)}$ of interest in back-propagation are real-valued, they can be expressed in such a complex-valued basis. What matters is what happens to the magnitude (complex absolute value) of these possibly complex-valued basis coefficients, when we multiply by the matrix by the vector. An eigenvalue with magnitude greater than one corresponds to magnification (exponential growth, if applied iteratevely) or shrinking (exponential decay, if applied iteratively).

With a non-linear map, the Jacobian is free to change at each step. The dynamics therefore become more complicated. However, it remains true that a small initial variation can turn into a large variation after a number of steps. One difference between the purely linear case and the non-linear case is that the use of a squashing non-linearity such as tanh can cause the recurrent dynamics to become bounded. Note that it is possible for back-propagation to retain unbounded dynamics even when forward propagation has bounded dynamics, for example, when a sequence of tanh units are all in the middle of their linear regime and are connected by weight matrices with spectral radius greater than 1.

The strategy proposed for reservoir computing machines (Jaeger, 2003). is to set the weights to make the Jacobians always slightly contractive. This is achieved by making the spectral radius of the weight matrix large but slightly less than 1: the maximum spectral radius of $\boldsymbol{J}$ is the spectral radius of $\boldsymbol{W}$ times the maximum

of the absolute value of the derivative of the non-linearity; when the latter is 1, such as with , it is thus recommended to set the spectral radius of $\boldsymbol{W}$ sligthly below 1.

However, in practice, good results are often found with a spectral radius of slightly larger than 1, such as 1.2 (Sutskever, 2012; Sutskever *et al.*, 2013). Keep in mind that with hyperbolic tangent units, the maximum derivative is 1, so that in order to guarantee a Jacobian spectral radius less than 1, the weight matrix should have spectral radius less than 1 as well. However, most derivatives of the hyperbolic tangent will be less than 1, which may explain Sutskever's empirical observation.

More recently, it has been shown that the techniques used to set the weights in ESNs could be used to *initialize* the weights in a fully trainable recurrent network (e.g., trained using back-propagation through time), helping to learn long-term dependencies (Sutskever, 2012; Sutskever *et al.*, 2013). In addition to setting the spectral radius to 1.2, Sutskever sets the recurrent weight matrix to be initially sparse, with only 15 non-zero input weights per hidden unit.

Note that when some eigenvalues of the Jacobian are exactly 1 and none are larger than 1, then some information can be kept in a stable way when moving forward in time, and there are paths in the unfolded computational graph through which back-propagated gradients neither vanish nor explode. The Jacobians tell us about differential propagation. They tell us how a vector of small changes gets propagated forward (when right-multiplying that vector with the product of the Jacobian matrices) or backward (when left-multiplying that vector with the product of the Jacobian matrices). Of course, because these networks are typically not linear, the forward propagation is different from this differential forward propagation, but the differential propagation is important to understand how small changes in input or parameters can influence the later states and outputs. The next two sections show methods to keep information for a very long time by making some paths in the unfolded graph correspond to "multiplying by 1" at each step.

### 10.7.2 Combining Short and Long Paths in the Unfolded Flow Graph

An old idea that has been proposed to deal with the difficulty of learning long-term dependencies is to use recurrent connections with long delays (Lin *et al.*, 1996). Whereas the ordinary recurrent connections are associated with a delay of 1 (relating the state at $t$ with the state at $t+1$), it is possible to construct recurrent networks with longer delays (Bengio, 1991), following the idea of incorporating delays in feedforward neural networks (Lang and Hinton, 1988) in order to capture temporal structure (with Time-Delay Neural Networks, which are the

1-D predecessors of Convolutional Neural Networks, discussed in Chapter 9).

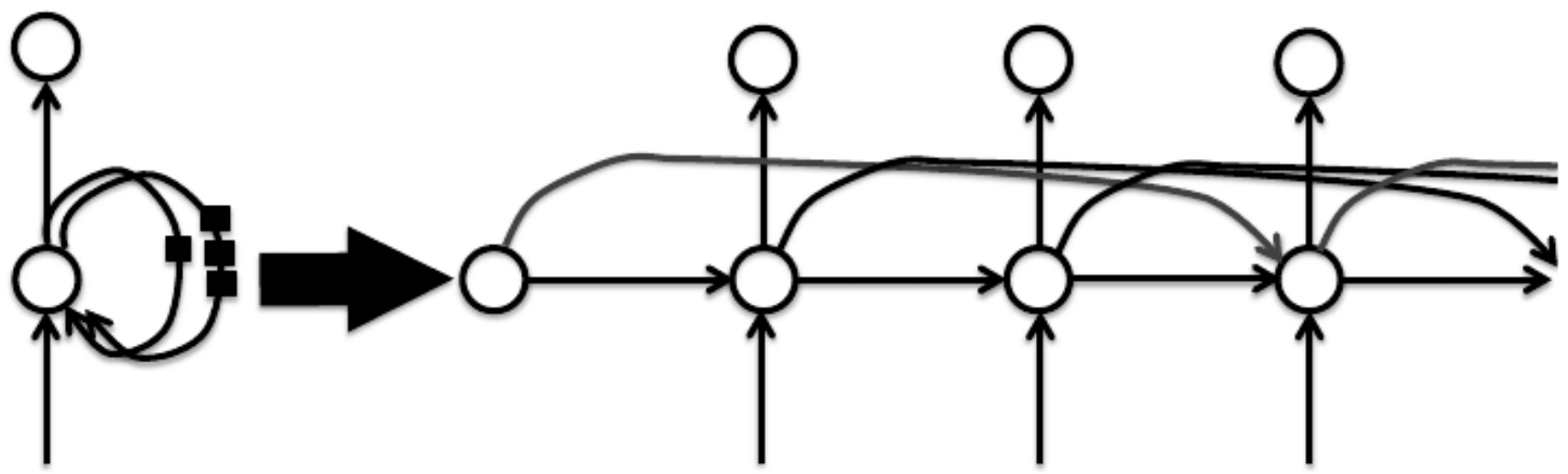

Figure 10.16: A recurrent neural networks with *delays*, in which some of the connections reach back in time to more than one time step. Left: connectivity of the recurrent net, with square boxes indicating the number of time delays associated with a connection. Right: unfolded recurrent network. In the figure there are regular recurrent connections with a delay of 1 time step ($W$) and recurrent connections with a delay of 3 time steps ($W$ ). The advantage of these longer-delay connections is that they allow to connect past states to future states through shorter paths (3 times shorter, here), going through these longer delay connections (in red).

As we have seen in Section 8.2.6, gradients may vanish or explode exponentially *with respect to the number of time steps.* If we have recurrent connections with a time-delay of $d$, then instead of the vanishing or explosion going as $O(\lambda^T)$ over $T$ time steps (where $\lambda$ is the largest eigenvalue of the Jacobians $\frac{\partial s_t}{\partial s_{t\ 1}}$), the unfolded recurrent network now has paths through which gradients grow as $O(\lambda^{T/d})$ because the number of effective steps is $T/d$. This allows the learning algorithm to capture longer dependencies although not all long-term dependencies may be well represented in this way. This idea was first explored in Lin *et al.* (1996) and is illustrated in Fig. 10.16.

### 10.7.3 Leaky Units and a Hierarchy of Different Time Scales

A related idea in order to obtain paths on which the product of derivatives is close to 1 is to have units with *linear* self-connections and a weight near 1 on these connections. The strength of that linear self-connection corresponds to a time scale and thus we can have different hidden units which operate at different time scales (Mozer, 1992). Depending on how close to 1 these self-connection weights are, information can travel forward and gradients backward with a different rate of "forgetting" or contraction to 0, i.e., a different *time scale.* One can view this idea as a smooth variant of the idea of having different delays in the connections

presented in the previous section. Such ideas were proposed in Mozer (1992); ElHihi and Bengio (1996), before a closely related idea discussed in the next section of *gating* these self-connections in order to let the network control at what rate each unit should be contracting. Leaky units were found to be useful (Jaeger *et al.*, 2007) in the context of echo state networks, introduced above (Sec. 10.7.1).

The idea of leaky units with a self-connection actually arises naturally when considering a *continuous-time* recurrent neural network such as

$$\dot{s}_i \tau_i = -s_i + \sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s} + \boldsymbol{U}_{i,:}\boldsymbol{x})$$

where $\sigma$ is the neural non-linearity (e.g., sigmoid or tanh), $\tau_i > 0$ is a time constant and $\dot{s}_i$ indicates the temporal derivative of unit $s_i$. A related equation is

$$\dot{s}_i \tau_i = -s_i + (b_i + \boldsymbol{W}_{i,:}\sigma(\boldsymbol{s}) + \boldsymbol{U}_{i,:}\boldsymbol{x})$$

where the state vector $\boldsymbol{s}$ (with elements $s_i$) now represents the pre-activation of the hidden units.

When discretizing in time such equations (which changes the meaning of $\tau$), one gets

$$s_{t+1,i} - s_{t,i} = -\frac{s_{t,i}}{\tau_i} + \frac{1}{\tau_i}\sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s}_t + \boldsymbol{U}_{i,:}\boldsymbol{x}_t)$$

$$s_{t+1,i} = (1 - \frac{1}{\tau_i})s_{t,i} + \frac{1}{\tau_i}\sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s}_t + \boldsymbol{U}_{i,:}\boldsymbol{x}_t). \tag{10.7}$$

We see that the new value of the state is a convex linear combination of the old value and of the value computed based on current inputs and recurrent weights, if $1 \leq \tau_i < \infty$. When $\tau_i = 1$, there is no linear self-recurrence, only the non-linear update which we find in ordinary recurrent networks. When $\tau_i > 1$, this linear recurrence allows gradients to propagate more easily. When $\tau_i$ is large, the state changes very slowly, integrating the past values associated with the input sequence.

By associating different time scales $\tau_i$ with different units, one obtains different paths corresponding to different forgetting rates. There are two basic strategies for setting these time constants. One strategy is to manually fix them to values that remain constant, for eample by sampling their values from some distribution once at initialization time. Another strategy is to make the time constants free parameters and learn them. Having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu *et al.*, 2013a).

Note that the time constant $\tau$ corresponds to a *self-weight* of $(1 - \frac{1}{\tau})$, *without any non-linearity involved in the self-recurrence.* If the recursive computation of the leaky unit is expanded, we find that it computes a weighted average of past input values, with weights that are exponentially decaying for older values, and

$\tau$ controls that exponential rate of decay. In the extreme case where $\tau \to \infty$, the weights converge to being all the same: the leaky unit just takes a simple *average* of contributions from the past. In that case, there is no associated vanishing or exploding effect. An alternative is to avoid the weight of $\frac{1}{\tau_i}$ in front of $\sigma\left(b_i + \boldsymbol{W}s^{(t)} + \boldsymbol{U}\boldsymbol{x}^{(t)}\right)$, thus making the state *sum* all the past values when $\tau_i$ is large, instead of averaging them.

### 10.7.4 The Long-Short-Term-Memory Architecture and Other Gated RNNs

Whereas in the previous section we consider creating paths where derivatives neither vanish nor explode too quickly by introducing self-loops, leaky units have self-weights that are not context-dependent: they are fixed, or learned, but remain constant during a whole test sequence.

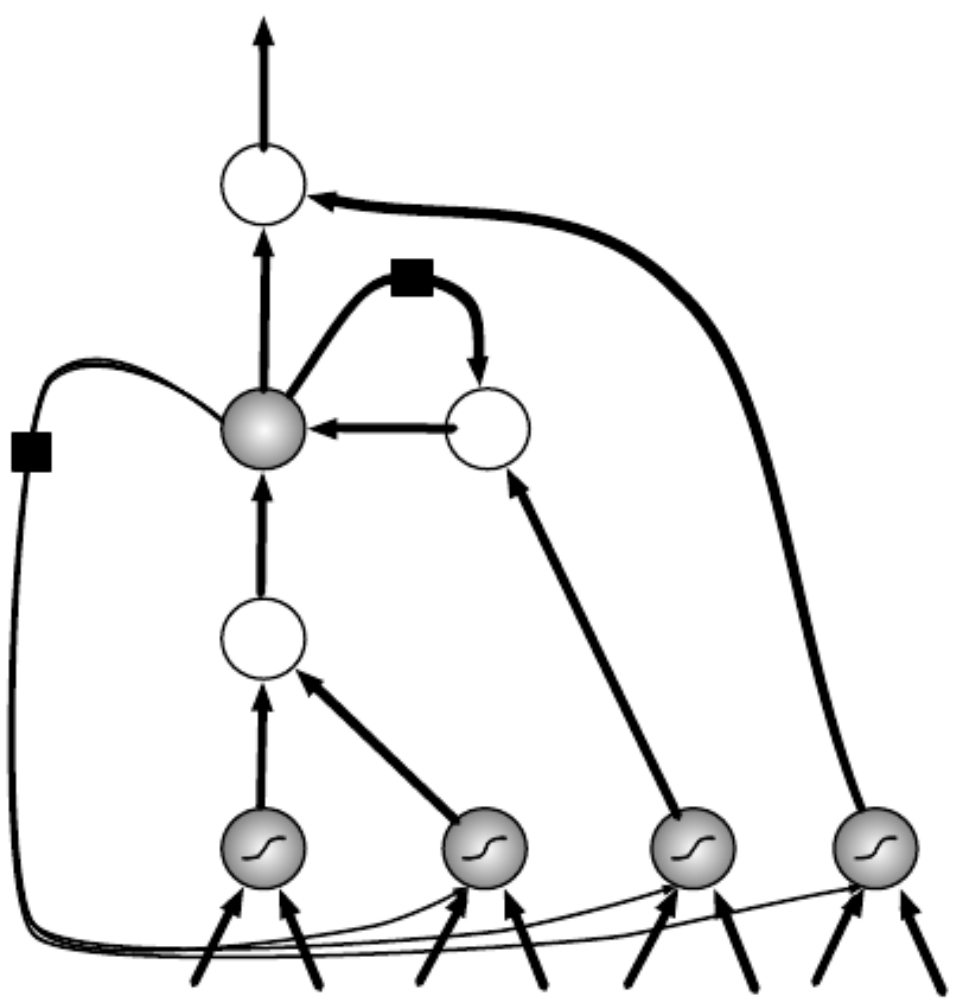

Figure 10.17: Block diagram of the LSTM recurrent network "cell". Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid non-linearity, while the input unit can have any squashing non-linearity. The state unit can also be used as extra input to the gating units. The black square indicates a delay of 1 time unit.

It is worthwhile to consider the role played by leaky units: they allow the

network to *accumulate* information (e.g. evidence for a particular feature or category) over a long duration. However, once that information gets used, it might be useful for the neural network to *forget* the old state. For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero and starting to count from fresh. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it.

### LSTM

This clever idea of conditioning the forgetting on the context is a core contribution of the Long-Short-Term-Memory (LSTM) algorithm (Hochreiter and Schmidhuber, 1997), described below. Several variants of the LSTM are found in the literature (Hochreiter and Schmidhuber, 1997; Graves, 2012; Graves *et al.*, 2013; Graves, 2013; Sutskever *et al.*, 2014a) but the principle is always to have a linear self-loop through which gradients can flow for long durations. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically (even for fixed parameters, but based on the input sequence). The LSTM has been found extremely successful in a number of applications, such as unconstrained handwriting recognition (Graves *et al.*, 2009), speech recognition (Graves *et al.*, 2013; Graves and Jaitly, 2014), handwriting generation (Graves, 2013), machine translation (Sutskever *et al.*, 2014a), image to text conversion (captioning) (Kiros *et al.*, 2014b; Vinyals *et al.*, 2014b; Xu *et al.*, 2015b) and parsing (Vinyals *et al.*, 2014a).

The LSTM block diagram is illustrated in Fig. 10.17. The corresponding forward (state update) equations are given below, in the case of a shallow recurrent network architecture. Deeper architectures have been successfully used in Graves *et al.* (2013); Pascanu *et al.* (2014a). Instead of a unit that simply applies an element-wise non-linearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_{t,i}$ that has a linear self-loop similar to the leaky units described in the previous section. However, here, the self-loop weight (or the associated time constant) is controlled by a *forget gate* unit $h^f_{t,i}$ (for time step $t$ and cell $i$), that sets this weight to a value between 0 and 1 via a sigmoid unit.

$$h^f_{t,i} = \text{sigmoid}(b^f_i + \sum_j U^f_{ij} x_{t,j} + \sum_j W^f_{ij} h_{t,j}), \tag{10.8}$$

where $\boldsymbol{x}_t$ is the current input vector and $\boldsymbol{h}_t$ is the current hidden layer vector,

containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f$ ,$\boldsymbol{U}^f$, $\boldsymbol{W}^f$ are respectively biases, input weights and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, following the pattern of Eq. 10.7, but with a conditional self-loop weight $h^f_{t,i}$ :

$$s_{t+1,i} = h^f_{t,i} s_{t,i} + h^e_{t,i} \sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h_{t,j}), \tag{10.9}$$

where $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights and recurrent weights into the LSTM cell. The *external input gate* unit $h^e_{t,i}$ is computed similarly to the forget gate (i.e., with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$h^e_{t,i} = \text{sigmoid}(b^e_i + \sum_j U^e_{ij} x_{t,j} + \sum_j W^e_{ij} h_{t,j}). \tag{10.10}$$

The output $h_{t+1,i}$ of the LSTM cell can also be shut off, via the *output gate* $h^o_{t,i}$, which also uses a sigmoid unit for gating:

$$\begin{aligned} h_{t+1,i} &= \tanh(s_{t+1,i}) h^o_{t,i} \\ h^o_{t,i} &= \text{sigmoid}(b^o_i + \sum_j U^o_{ij} x_{t,j} + \sum_j W^o_{ij} h_{t,j}) \end{aligned} \tag{10.11}$$

which has parameters $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_{t,i}$ as an extra input (with its weight) into the three gates of the $i$-th unit, as shown in Fig. 10.17. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial data sets designed for testing the ability to learn long-term dependencies Bengio *et al.* (1994); Hochreiter and Schmidhuber (1997); Hochreiter *et al.* (2000), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves *et al.*, 2013; Sutskever *et al.*, 2014a). Variants and alternatives to the LSTM have been studied and used and are discussed next.

### Other Gated RNNs

Which pieces of the LSTM architecture are actually necessary? What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behavior of different units?

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as Gated Recurrent Units (GRU) (Cho *et al.*,

2014b; Chung *et al.*, 2014, 2015; Jozefowicz *et al.*, 2015b; Chrupala *et al.*, 2015), which were successfully used in reaching the MOSES state-of-the-art for English-to-French machine translation (Cho *et al.*, 2014a). The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit, which is natural if we consider the continuous-time interpretation of the self-weight of the state, as in the equation for leaky units, Eq. 10.7. The update equations are the following:

$$h_{t+1,i} = h^{u}_{t,i} h_{t,i} + (1 - h^{u}_{t,i})\sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h^{r}_{t,j} h_{t,j}). \tag{10.12}$$

where $\boldsymbol{g}^u$ stands for "update" gate and $\boldsymbol{g}^r$ for "reset" gate. Their value is defined as usual:

$$h^{u}_{t,i} = \text{sigmoid}(b^{u}_{i} + \sum_j U^{u}_{ij} x_{t,j} + \sum_j W^{u}_{ij} h_{t,j}) \tag{10.13}$$

and

$$h^{r}_{t,i} = \text{sigmoid}(b^{r}_{i} + \sum_j U^{r}_{ij} x_{t,j} + \sum_j W^{r}_{ij} h_{t,j}). \tag{10.14}$$

The reset and updates gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it by the new "target state" value (towards which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional non-linear effect in the relationship between past state and future state.

Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across a number of hidden units. Or the product of a global gate (covering a whole group of units, e.g., a layer) and a local gate (per unit) could be used to combine global control and local control. However, several investigations over architectural variations of the LSTM and GRU found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 2015a). Greff *et al.* (2015) found that a crucial ingredient is the forget gate, while Jozefowicz *et al.* (2015a) found that adding a bias of 1 to the LSTM forget gate, a practice advocated by Gers *et al.* (2000), makes the LSTM as strong as the best of the explored architectural variants.

### 10.7.5 Explicit Memory

Intelligence requires knowledge and acquiring knowledge can be done via learning, which has motivated the development of large-scale deep architectures. However,

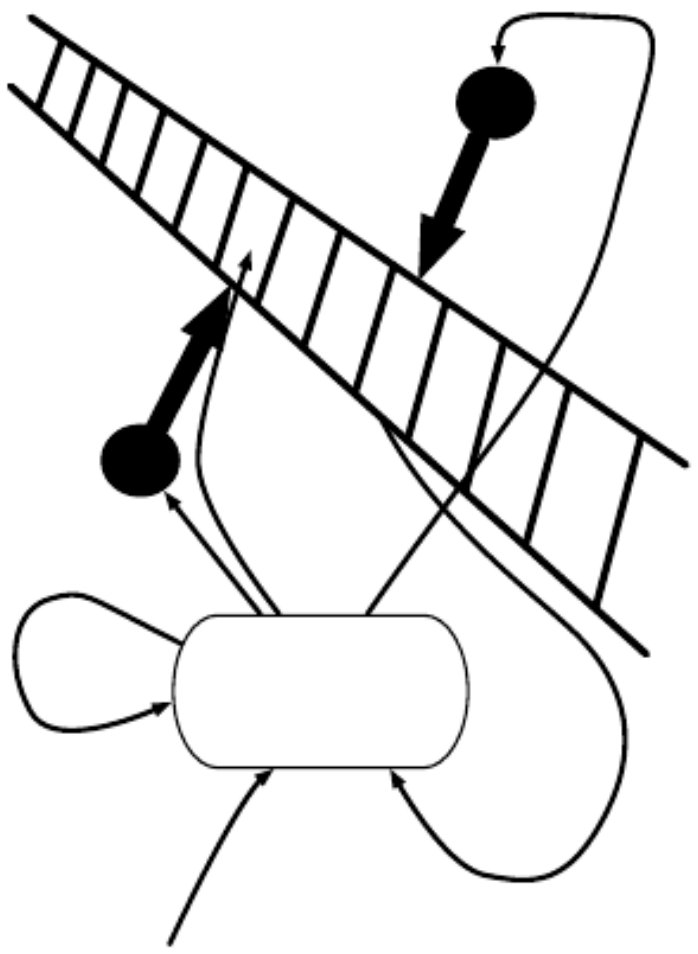

Figure 10.18: A schematic example of memory network architecture, in which we explicitly distinguish the "representation" part of the model (the "task network", here a recurrent net in the bottom) from the "memory" part of the model (the set of cells), which can store facts. From this representation, one learns to "control" the memory, decided from where to read and write (through the reading and writing mechanims, indicated by circles with arrows pointing at the reading and writing addresses).

there are different kinds of knowledge. Some knowledge can be implicit, subconscious, and difficult to verbalize—such as how to walk, or how a dog looks different from a cat. Other knowledge can be explicit, declarative, and relatively straightforward to put into words—every day commonsense knowledge, like "a cat is a kind of animal," or very specific facts that you need to know to accomplish your current goals, like "The meeting with the sales team is at 3:00 PM in room 141."

Neural networks excel at storing implicit knowledge. However, they struggle to memorize facts. Stochastic gradient descent requires many presentations of the same input before it can be stored in a neural networks parameters, and even then, that input will not be stored especially precisely. Graves *et al.* (2014b) hypothesized that this is because neural networks lack the equivalent of the *working memory* system that allows human beings to explicitly hold and manipulate pieces of information that are relevant to achieving some goal. Such explicit memory components would allow our systems not only to rapidly and "intentionally" store and retrieve specific facts but also to sequentially reason with them.

The core idea behind *memory networks* (Weston *et al.*, 2014) and *neural Turing machines* (Graves *et al.*, 2014b) is to add a set of memory cells that can be read from and written to via an addressing mechanism. Each memory cell can be thought of as an extension of the memory cells in LSTMs and GRUs. The difference is that the network outputs an internal state that chooses which cell to read from or write to, just as memory accesses in a digital computer read from or write to a specific address.

It is difficult to optimize functions that produce exact, integer addresses. To alleviate this problem, memory networks and NTMs actually read to or write from many memory cells simultaneously. To read, they take a weighted average of many cells. To write, they modify multiple cells by different amounts. The coefficients for these operations are chosen to be focused on a small number of cells, for example, by producing them via a softmax function. Using these weights with non-zero derivatives allows the functions controlling access to the memory to be optimized using gradient descent. The gradient on these coefficients indicates whether each of them should be increased or decreased, but the gradient will typically be large only for those memory addresses receiving a large coefficient.

These memory cells are typically augmented to contain a vector, rather than the single scalar stored by an LSTM or GRU memory cell. There are two reasons to increase the size of the memory cell. One reason is that we have increased the cost of accessing a memory cell. We pay the computational cost of producing a coefficient for many cells, but we expect these coefficients to cluster around a small number of cells. By reading a vector value, rather than a scalar value, we can offset some of this cost. Another reason to use vector-valued memory cells is

that they allow for *content-based addressing*, where the weight used to read to or write from a cell is a function of that cell. Vector-valued cells allow us to retrieve a complete vector-valued memory if we are able to produce a pattern that matches some but not all of its elements. This is analogous to the way that people can recall the lyrics of a song based on a few words. We can think of a content-based read instruction as saying, "Retrieve the lyrics of the song that has the chorus 'We all live on a yellow submarine." ' Content-based addressing is more useful when we make the objects to be retrieved large—if every letter of the song was stored in a separate memory cell, we would not be able to find them this way. By comparison, *location-based addressing* is not allowed to refer to the content of the memory. We can think of a location-based read instruction as saying "Retrieve the lyrics of the song in slot 347." Location-based addressing can often be a perfectly sensible mechanism even when the memory cells are small.

If the content of a cell in a memory network is copied (not forgotten) at most time steps, then the information it contains can be propagated forward in time and the gradients propagated backward in time without either vanishing or exploding. This is illustrated in Fig. 10.18, where we see that a "task neural network" is coupled with a memory. Although that task neural network could be feedforward or recurrent (the latter being the case in the figure), the overall system is a recurrent network. The task network can choose to read from or write to specific memory addresses. Explicit memory seems to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be because information and gradients can be propagated (forward in time or backwards in time, respectively) for very long durations.

As an alternative to back-propagation through weighted averages of memory cells, we can interpret the weights as probabilities and stochastically read just one cell. A variant of the REINFORCE algorithm (Williams, 1992) can estimate a noisy gradient on the addressing weights (Zaremba and Sutskever, 2015). So far, training these stochastic architectures that make hard decisions remains harder than training deterministic algorithms that make soft decisions.

Whether it is soft (allowing back-propagation) or stochastic and hard, the mechanism for choosing an address is in its form identical to the *attention mechanism* which had been previously introduced in the context of machine translation (Bahdanau *et al.*, 2014) and discussed in Sec. 12.4.6. The idea of attention mechanisms for neural networks was introduced even earlier, in the context of handwriting generation (Graves, 2013), in which the focus attention is moving monotonically in the input sequence. In the case of machine translation and memory networks, at each step, the focus of attention can move to a completely different place, compared to the previous step.

### 10.7.6 Better Optimization

A central optimization difficulty with RNNs regards the learning of long-term dependencies (Hochreiter, 1991; Bengio *et al.*, 1993, 1994). This difficulty has been explained in detail in Section 8.2.6. The gist of the problem is that the composition of the non-linear recurrence with itself over many many time steps yields a highly non-linear function whose derivatives (e.g. of the state at $T$ w.r.t. the state at $t < T$, i.e. the Jacobian matrix $\frac{\partial \boldsymbol{s}_T}{\partial \boldsymbol{s}_t}$) tend to either vanish or explode as $T-t$ increases, because it is equal to the product of the state transition Jacobian matrices $\frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t}$)

If it explodes, the parameter gradient $\nabla_{\boldsymbol{\theta}} L$ also explodes, yielding gradient-based parameter updates that are poor. A simple heuristic but practical solution to this problem is discussed in the next section (Sec. 10.7.7). However, as discussed in Bengio *et al.* (1994), if the state transition Jacobian matrix has eigenvalues that are larger than 1 in magnitude, then it can yield to "unstable" dynamics, in the sense that a bit of information cannot be stored reliably for a long time in the presence of input "noise". Indeed, the state transition Jacobian matrix eigenvalues indicate how a small change in some direction (the corresponding eigenvector) will be expanded (if the eigenvalue is greater than 1) or contracted (if it is less than 1).

An interesting idea proposed in Martens and Sutskever (2011) is that at the same time as first derivatives are becoming smaller in directions associated with long-term effects, *so may the higher derivatives.* In particular, if we use a second-order optimization method (such as the Hessian-free method of Martens and Sutskever (2011)), then we could differentially treat different directions: divide the small first derivative (gradient) by a small second derivative, while not scaling up in the directions where the second derivative is large (and hopefully, the first derivative as well). Whereas in the scalar case, if we add a large number and a small number, the small number is "lost", in the vector case, if we add a large vector with a small vector, it is still possible to recover the information about the direction of the small vector if we have access to information (such as in the second derivative matrix) that tells us how to rescale appropriately each direction.

One disadvantage of many second-order methods, including the Hessian-free method, is that they tend to be geared towards "batch" training (or fairly large minibatches) rather than "stochastic" updates (where only one example or a small minibatch of examples are examined before a parameter update is made). Although the experiments on recurrent networks applied to problems with long-term dependencies showed very encouraging results in Martens and Sutskever (2011), it was later shown that similar results could be obtained by much simpler methods (Sutskever, 2012; Sutskever *et al.*, 2013) involving better initialization, a cheap surrogate to second-order optimization (a variant on the momentum

technique, Section 8.5), and the clipping trick described below.

### 10.7.7 Clipping Gradients

As discussed in Section 8.2.5, strongly non-linear functions such as those computed by a recurrent net over many time steps tend to have derivatives that can be either very large or very small in magnitude. This is illustrated in Fig.s 8.2 and 8.3, in which we see that the objective function (as a function of the parameters) has a "landscape" in which one finds "cliffs": wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing a lot of the work that had been done to reach the current solution. The gradient tells us the direction that corresponds to the steepest descent within an infinitesimal region surrounding the current parameters. Outside of this infinitesimal region, the cost function may begin to curve back upwards. The update must be chosen to be small enough to avoid traversing too much upward curvature. We typically use learning rates that decay slowly enough that consecutive steps have approximately the same learning rate. A step size that is appropriate for a relatively linear part of the landscape is often inappropriate and causes uphill motion if we enter a more curved part of the landscape on the next step.

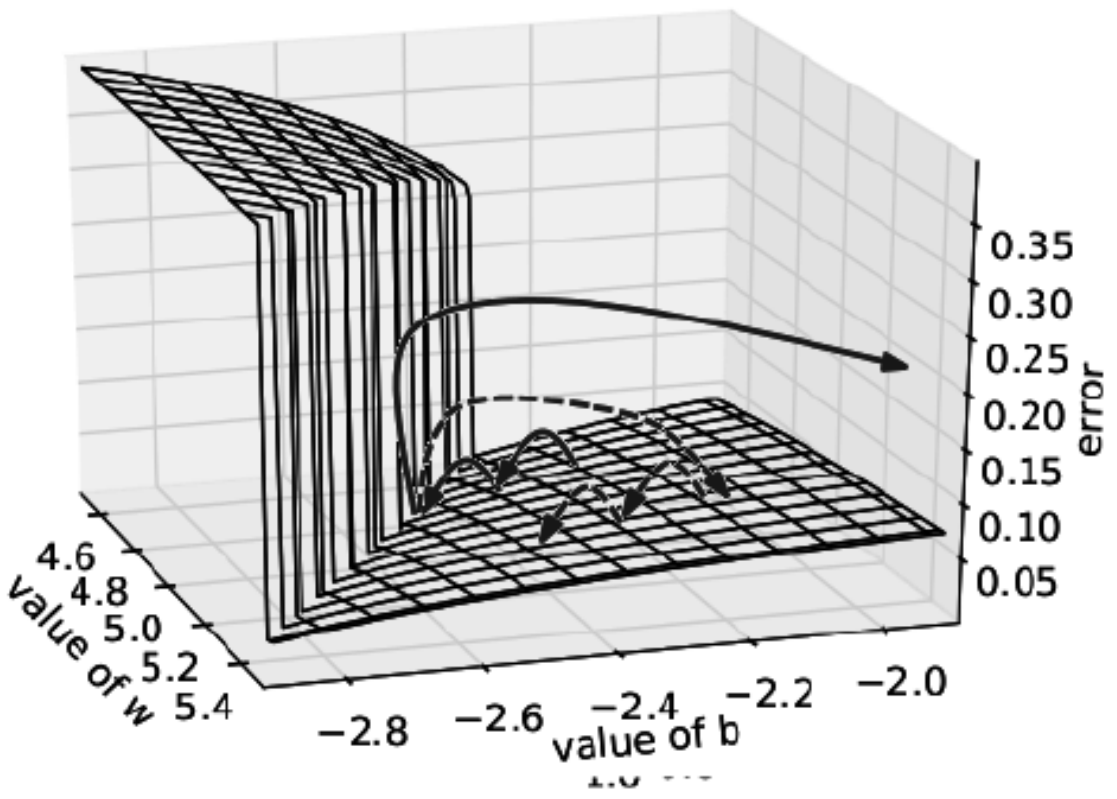


Figure 10.19: Example of the effect of gradient clipping in a recurrent network with two parameters $\boldsymbol{w}$ and $\boldsymbol{b}$. Vertical axis is the objective function to minimize. Note the cliff where the gradient explodes and from where gradient descent can get pushed very far. Clipping the gradient when its norm is above a threshold (Pascanu *et al.*, 2013a) prevents this catastrophic outcome and helps training recurrent nets with long-term dependencies to be captured.

A simple type of solution has been in used by practitioners for many years: *clipping the gradient*. There are different instances of this idea (Mikolov, 2012; Pascanu *et al.*, 2013a). One option is to clip the parameter gradient from a mini-batch *element-wise* (Mikolov, 2012) just before the parameter update. Another is to *clip the norm* $||\boldsymbol{g}||$ *of the gradient* $\boldsymbol{g}$ (Pascanu *et al.*, 2013a) just before the parameter update:

$$\begin{aligned} &\text{if } ||\boldsymbol{g}|| > v \\ &\qquad \boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||} \end{aligned} \tag{10.15}$$

where $v$ is the norm threshold and $\boldsymbol{g}$ is used to update parameters. Because the gradient of all the parameters (including different groups of parameters, such as weights and biases) is renormalized jointly with a single scaling factor, the latter method has the advantage that it guarantees that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Although the parameter update has the same direction as the true gradient, with gradient norm clipping, the parameter update vector norm is now bounded. This bounded gradient avoids performing a detrimental step when the gradient explodes. In fact, even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well. If the explosion is so severe that the gradient is numerically or (considered infinite or not-a-number), then

a random step of size $v$ can be taken and will typically move away from the numerically unstable configuration. Clipping the gradient norm per-minibatch will not change the direction of the gradient for an individual minibatch. However, taking the average of the norm-clipped gradient from many minibatches is not equivalent to clipping the norm of the true gradient (the gradient formed from using all examples). Examples that have large gradient norm, as well as examples that appear in the same minibatch as such examples, will have their contribution to the final direction diminished. This stands in contrast to traditional minibatch gradient descent, where the true gradient direction is equal to the average over all minibatch gradients. Put another way, traditional stochastic gradient descent uses an unbiased estimate of the gradient, while gradient descent with norm clipping introduces a heuristic bias that we know empirically to be useful. With element-wise clipping, the direction of the update is not aligned with the true gradient or the minibatch gradient, but it is still a descent direction. It has also been proposed (Graves, 2013) to clip the back-propagated gradient (with respect to hidden units) but no comparison has been published between these variants; we conjecture that all these methods behave similarly.

### 10.7.8 Regularizing to Encourage Information Flow

Whereas clipping helps dealing with exploding gradients, it does not help with vanishing gradients. To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms, described above in Section 10.7.4. Another idea is to regularize or constrain the parameters so as to encourage "information flow". In particular, we would like the gradient vector $\nabla_{\boldsymbol{s}_t} L$ being back-propagated to maintain its magnitude (even if there is only a loss at the end of the sequence), i.e., we want

$$\nabla_{\boldsymbol{s}_t} L \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s}_{t-1}}$$

to be as large as

$$\nabla_{\boldsymbol{s}_t} L.$$

With this objective, Pascanu *et al.* (2013a) propose the following regularizer:

$$\Omega = \sum_t \left( \frac{\left|\left|\nabla_{\boldsymbol{s}_t} L \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s}_{t-1}}\right|\right|}{\left|\left|\nabla_{\boldsymbol{s}_t} L\right|\right|} - 1 \right)^2. \tag{10.16}$$

It looks like computing the gradient of this regularizer is difficult, but Pascanu *et al.* (2013a) propose an approximation in which we consider the back-propagated vectors $\nabla_{\boldsymbol{s}_t} L$ as if they were constants (for the purpose of this regularizer, i.e., no need to back-prop through them). The experiments with this regularizer suggest that, if combined with the norm clipping heuristic (which handles gradient explosion), it can considerably increase the span of the dependencies that an RNN can learn. Because it keeps the RNN dynamics on the edge of explosive gradients, the gradient clipping is particularly important: otherwise gradient explosion prevents learning to succeed.

### 10.7.9 Organizing the State at Multiple Time Scales

Another promising approach to handle long-term dependencies is the old idea of organizing the state of the RNN at multiple time-scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales. This is illustrated in Fig. 10.20.

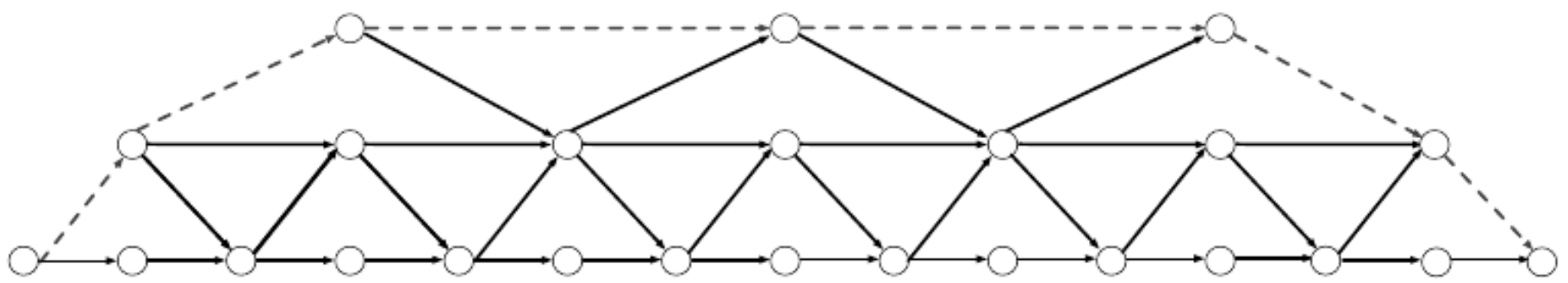

Figure 10.20: Example of a multi-scale recurrent net architecture (unfolded in time), with higher levels operating at a slower time scale. Information can flow unhampered (either forward or backward in time) over longer durations at the higher levels, thus creating long-paths (such as the dotted path) through which long-term dependencies between elements of the input/output sequence can be captured.

There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky (as in Eq. 10.7), but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013a). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units, as in Fig. 10.20. This is the approach of El Hihi and Bengio (1996); Koutnik *et al.* (2014) and it also worked well on a number of benchmark datasets.

# Chapter 11

# Practical methodology

Successfully applying deep learning techniques requires more than just a good knowledge of what algorithms exist and the principles that explain how they work. A good machine learning practitioner also needs to know how to choose an algorithm for a particular application and how to monitor and respond to feedback obtained from experiments in order to improve a machine learning system. During day to day development of machine learning systems, practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model. All of these operations are at the very least time-consuming to try out, so it is important to be able to determine the right course of action rather than blindly guessing.

Most of this book is about different machine learning models, training algorithms, and objective functions. This may give the impression that the most important ingredient to being a machine learning expert is knowing a wide variety of machine learning techniques and being good at different kinds of math. In practice, one can usually do much better with a correct application of a commonplace algorithm than by sloppily applying an obscure algorithm. Correct application of an algorithm depends on mastering some fairly simple methodology. Many of the recommendations in this chapter are adapted from a lecture by Andrew Ng (Ng, 2015).

We recommend the following practical design process:

- Determine your goals—what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the application of interest.

- Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.

- Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether it is due to overfitting, underfitting, or a defect in the data or software.

- Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.

Determining your goals, in terms of which error metric to use, is a necessary first step because your error metric will guide all of your future actions. You should also have an idea of what level of performance you desire. Keep in mind that for most applications, it is impossible to achieve absolute zero error. The Bayes error defines the minimum error rate that you can hope to achieve, even if you have infinite training data and can recover the true probability distribution. This is because your input features may not contain complete information about the output variable, or because the system might be intrinsically stochastic. You will also be limited by having a finite amount of training data. When your goal is to answer a scientific question about which algorithm performs better on a fixed benchmark, the benchmark specification usually determines the training set and you are not allowed to collect more data. When your goal is to build the best possible real-world product or service, you can typically collect more data but must determine the value of reducing error further and weigh this against the cost of collecting more data. Data collection can require time, money, or human suffering (for example, if your data collection process involves performing invasive medical tests). Typically, in the academic setting, we have some estimate of the error rate that is attainable based on previously published benchmark results. In the real-word setting, we have some idea of the error rate that is necessary for an application to be safe, cost-effective, or appealing to consumers. Once you have determined your realistic desired error rate, your design decisions will be guided by reaching this error rate.

When should one decide to gather more data? More data always helps generalization, but the cost of collecting that data may not be justified by the incremental improvement. It is possible to forecast the effect of increasing the amount of data, typically using an empirical methodology resting on the assumption that the relationship between training set size and generalization error has a smooth and saturating shape. This makes it possible to approximately extrapolate from experiments done with less data than what is currently available. Note that adding a small fraction of the total number of examples will typically not have a noticeable impact on generalization error. It is therefore recommended to experiment with training set sizes on a logarithmic scale, for example doubling the number of examples between consecutive experiments.

This chapter contains further guidelines regarding the exploration of other aspects of the machine learning experiments, in particular concerning the choice of hyper-parameters, which often change the model size (which influences computational costs) or its capacity (its ability to extract more information from examples, often related to the number of free parameters), and yield different effects on the training set and validation or test sets.

## 11.1 Default Baseline Models

The first step in any practical application is to establish a reasonable end-to-end system. Note that the defaults suggested below are likely to change in the future as research progresses.

Depending on the complexity of your problem, you may even want to begin without using deep learning. If your problem has a chance of being solved by just choosing a few linear weights correctly, you may want to begin with a simple statistical model like Naive Bayes.

If you know that your problem falls into an "AI-complete" category like object recognition, speech recognition, machine translation, and so on, then you are likely to do well by beginning with an appropriate deep learning model.

First, choose the general category of model based on the structure of your data. If you want to perform supervised learning with fixed-size vectors as input, use a feed-forward network with fully connected layers. If the input has known topological structure (for example, if the input is an image), use a convolutional network. In these cases, you should begin by using some kind of piecewise linear unit (ReLUs or their generalizations like Leaky ReLUs, PreLus and maxout). If your input or output is a sequence, use a gated recurrent net (LSTM or GRU).

A reasonable choice of optimization algorithm is SGD with momentum with a decaying learning rate (popular decay schemes that perform better or worse on different problems include decaying linearly until reaching a fixed minimum learning rate, decaying exponentially, or decreasing the learning rate by a factor of 2-10 each time validation error plateaus). Another very reasonable alternative is Adam, although we expect that the technology for adaptive momentum and learning rates will improve in the future, so keep an eye on that.

Unless your training set contains tens of millions of examples or more, you should include some mild forms of regularization from the start. Early stopping should be used almost universally. Dropout is an excellent regularizer that is easy to implement and compatible with many models and training algorithms. Batch normalization is another excellent addition to standard neural networks and it has been shown to help substantially and is becoming commonly used. The authors of the paper introducing batch normalization (Ioffe and Szegedy, 2015) argue that

it acts as regularization strategy.

If your task is similar to another task that has been studied extensively, you will probably do well by first copying the model and algorithm that is already known to perform best on the previously studied task. You may even want to copy a trained model from that task. For example, it is common to use the features from a convolutional network trained on ImageNet to solve other computer vision tasks.

A common question is whether to begin by using unsupervised pretraining. This is somewhat domain specific. Some domains, such as natural language processing, are known to benefit tremendously from unsupervised learning techniques such as learning unsupervised word embeddings. In other domains, such as computer vision, current unsupervised learning techniques do not bring a benefit, except in the semi-supervised setting, when the number of labeled examples is very small (Kingma *et al.*, 2014; Rasmus *et al.*, 2015). If your application is in a context where unsupervised learning is known to be important, then include it in your first end-to-end baseline. Otherwise, only use unsupervised learning in your first attempt if the task you want to solve is unsupervised. You can always try adding unsupervised learning later if you observe that your initial baseline overfits.

## 11.2 Selecting Hyperparameters

Most deep learning algorithms come with many hyperparameters that control many aspects of the algorithm's behavior. Some of these hyperparameters affect the time and memory cost of running the algorithm. Some of these hyperparameters affect the quality of the model recovered by the training process and its ability to infer correct results when deployed on new inputs.

There are two basic approaches to choosing these hyparameters: choosing them manually and choosing them automatically. Choosing the hyperparameters manually requires understanding what the hyperparameters do and how machine learning models achieve good generalization. Automatic hyperparameter selection algorithms greatly reduce the need to understand these ideas, but they are often much more computationally costly.

### 11.2.1 Manual Hyperparameter Tuning

To set hyperparameters manually, one must understand the relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime). This means establishing a solid foundation on the fundamental ideas concerning the effective capacity of a learning algorithm from Chapter 5.

The goal of manual hyperparameter search is usually to find the lowest generalization error subject to some runtime and memory budget. Understanding the runtime and memory effect of most hyperparameters is a relatively straightforward software engineering exercise[1] that we do not describe in detail here. Instead, we focus on how to reduce generalization error by modifying hyperparameters.

The primary goal of manual hyperparameter search is to adjust the effective capacity of the model to match the complexity of the task. Effective capacity is constrained by three factors: the representational capacity of the model, the ability of the learning algorithm to successfully minimize the cost function used to train the model, and the degree to which the cost function and training procedure regularize the model. A model with more layers and more hidden units per layer has higher representational capacity—it is capable of representing more complicated functions. It can not necessarily actually learn all of these functions though, if the training algorithm cannot discover that certain functions do a good job of minimizing the training cost, or if regularization terms such as weight decay forbid some of these functions.

The generalization error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters, as in Fig. 5.3. At one extreme, the hyperparameter value corresponds to low capacity, and generalization error is high because training error is high. This is the underfitting regime. At the other extreme, the hyperparameter value corresponds to high capacity, and the generalization error is high because the gap between training and test error is high. Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error, by adding a medium generalization gap to a medium amount of training error.

For some hyperparameters, overfitting occurs when the value of the hyperparameter is large. For example, models with a larger number of parameters tend to have higher capacity. For other hyperparameters, overfitting occurs when the value of the hyperparameter is small. For example, the smallest allowable weight decay coefficient of 0 corresponds to the greatest effective capacity of the learning algorithm.

Not every hyperparameter will be able to explore the entire U-shaped curve. Many hyperparameters are discrete, such as the number of units in a layer or the number of linear pieces in a maxout unit, so it is only possible to visit a few points along the curve. Some hyperparameters are binary. Usually these hyperparameters are switches that specify whether to use some optional component of the learning algorithm, such as normalization of the input, or not. These hy-

[1] it becomes less straighforward when we consider a distributed computation setup, taking into account bandwidth and communication bottlenecks as well as reliability issues on large computer clusters

perparameters can only explore two points on the curve. Other hyperparameters have some minimum or maximum value that prevents them from exploring some part of the curve. For example, the minimum weight decay coefficient is 0. This means that if the model is underfitting when weight decay is 0, we can not enter the overfitting region by modifying the weight decay coefficient.

The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate. It also behaves somewhat differently from capacity control hyperparameters. The learning rate has a U-shaped curve for *training* error, illustraed in Fig. 11.1. When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. In the idealized quadratic case, this occurs if the learning rate is at least twice as large as its optimal value (LeCun *et al.*, 1998a). When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error. This effect is poorly understood (it would not happen for a convex loss function).

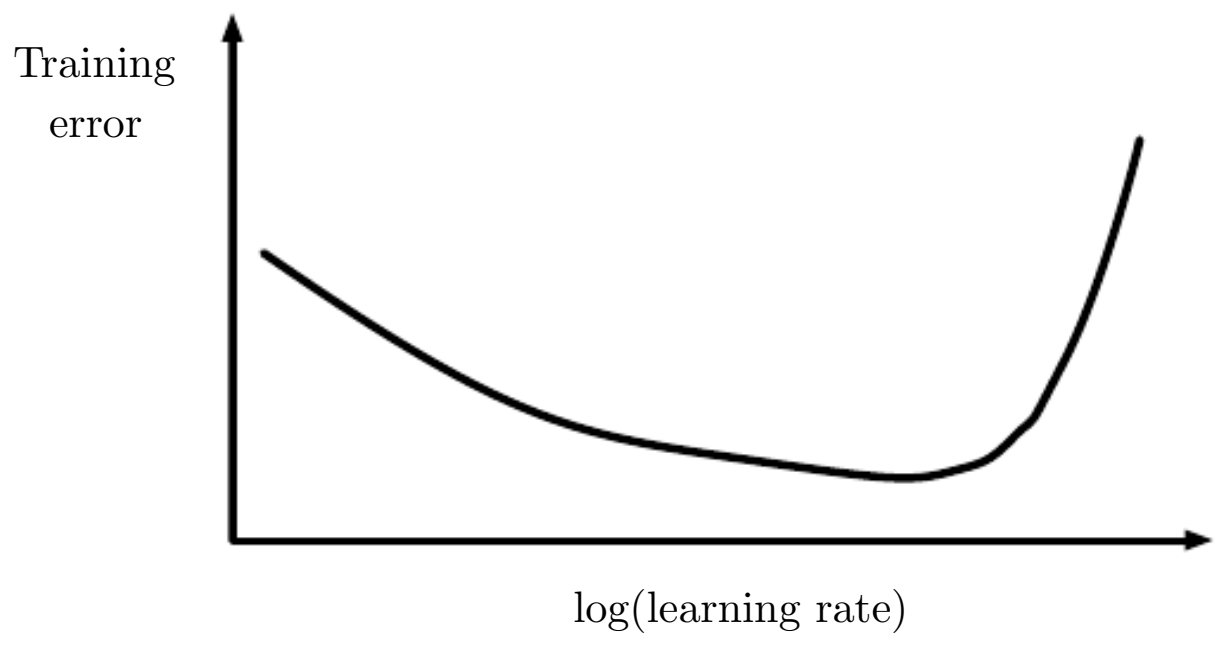


Figure 11.1: Typical relationship between the logarithm of the learning rate and the training error. Notice the sharp rise in error when the learning is above an optimal value. This is for a fixed training time, as a smaller learning rate may sometimes only slow down training by a factor proportional by the learning rate reduction. Generalization error can follow this curve or be complicated by regularization effects arising out of having a too large or too small learning rates, since poor optimization can, to some degree, reduce or prevent overfitting, and even points with equivalent training error can have different generalization error.

Tuning the parameters other than the learning rate requires monitoring both training and test error to diagnose whether your model is overfitting or underfitting, then adjusting its capacity appropriately.

If your error on the training set is higher than your target error rate, you have no choice but to increase the model capacity.

If your error on the test set is higher than than your target error rate, you can now take two kinds of actions. The test error is the sum of the training error and the gap between training and test error. The optimal test error is found by trading off these quantities. Neural networks typically perform best when the training error is very low (and thus, when capacity is high) and the test error is primarily driven by the gap between train and test error. Your goal is to reduce this gap without increasing training error faster than the gap decreases. To reduce the gap, change regularization hyperparameters to reduce effective model capacity, such as by adding dropout or weight decay. Usually the best performance comes from a large model that is regularized well, for example by using dropout.

For most hyperparameters, you can get a sense of how to set them by reasoning about whether they increase or decrease model capacity. Some examples are included in Table 11.1.

While manually tuning hyperparameters, do not lose sight of your end goal: good performance on the test set. Adding regularization is only one way to achieve this goal. As long as you have low training error, you can always reduce generalization error by collecting more training data. The brute force way to practically guarantee success is to continually increase model capacity and training set size until the task is solved. This approach does of course increase the computational cost of training and inference, so it is only feasible given appropriate resources. In principle, this approach could fail due to optimization difficulties, but for many problems optimization does not seem to be a significant barrier, provided that the model is chosen appropriately.

### 11.2.2 Automatic Hyperparameter Optimization Algorithms

The ideal learning algorithm just takes a dataset and outputs a function, without requiring hand-tuning of hyperparameters. The popularity of several learning algorithms such as logistic regression and SVMs stems from the fact that they can get away with one or two hyperparameters. For many years, the widespread use of neural networks has been hampered by the existence of many hyperparameters[2]. Manual hyperparameter tuning can work very well when the user has a good starting point, such as one determined by others having worked on the same type of application and architecture, or when the user has months or years of experience in exploring hyperparameter values for neural networks applied to similar tasks.

If we think about the way in which the user of a learning algorithm searches for good values of the hyperparameters, we realize that an optimization is taking place: we are trying to find a value of the hyperparameters that optimizes an objective function, such as validation error, sometimes under constraint (such as

[2] from 3 or 4 to possibly dozens if one wants to separately set hyperparameters for different layers

| Hyper-parameter | Increases capacity when... | Reason | Caveats |
|---|---|---|---|
| Number of hidden units | increased | Increasing the number of hidden units increases the representational capacity of the model. | Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model. |
| Learning rate | tuned optimally | An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure | |
| Convolution kernel width | increased | Increasing the kernel width increases the number of parameters in the model | A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost. |
| Implicit zero padding | increased | Adding implicit zeros before convolution keeps the representation size large | Increased time and memory cost of most operations. Can be used to offset the output narrowing effect of wide kernels. |
| Weight decay coefficient | decreased | Decreasing the weight decay coefficient frees the model parameters to become larger | |
| Dropout rate | decreased | Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set | |

Table 11.1: The effect of various hyperparameters on model capacity.

a budget for training time, memory or recognition time). It is therefore possible, in principle, to apply optimization algorithms to *encapsulate a learning algorithm that has hyperparameters and yield a hyperparameter-free learning algorithm.* We call these procedures *hyperparameter optimization algorithms.* Unfortunately, they typically also involve hyperparameters, such as the range of values of be explored, but these tend to have much less influence on the final result.

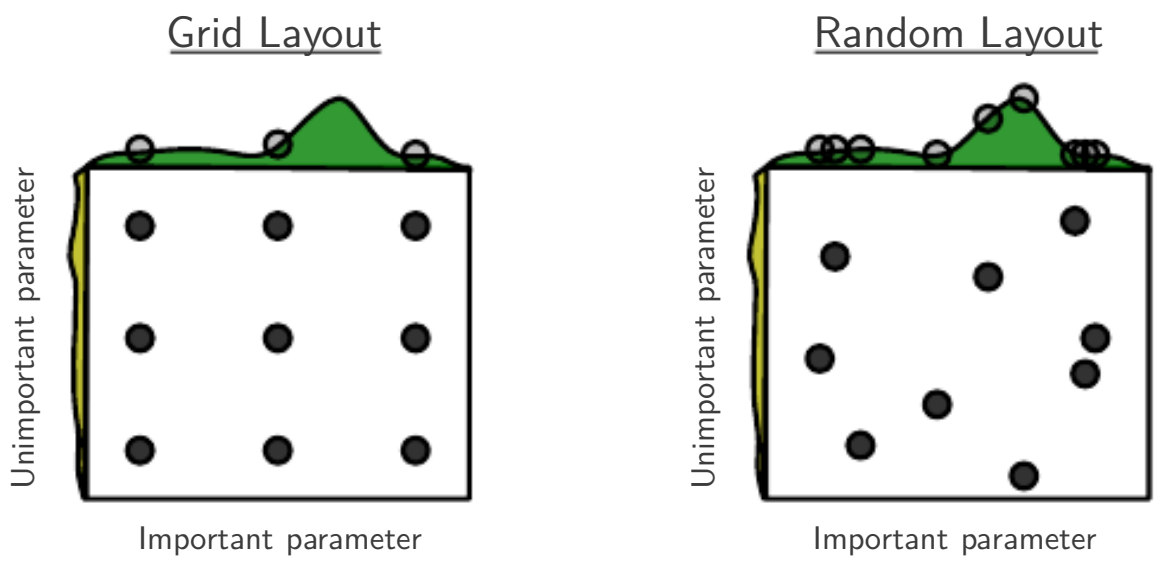


Figure 11.2: Comparison of grid search (left) and random search (right). For illustration purposes we display 2 hyperparameters but we are typically interested in having many more. To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets. To perform random search, we provide a probability distribution over joint hyperparameter configurations. Usually most of these hyperparameters are independent from each other. Common choices for the distribution over a single hyperparameter include uniform and log-uniform (to sample from a log-uniform distribution, take the exp of a sample from a uniform distribution). The search algorithm then randomly samples joint hyperparameter configurations and runs training with each of them. Both grid search and random search evaluate the validation set error and return the best configuration. The figure illustrates the typical case where only some hyperparameters have a significant influence on the result: in that case, grid search wastes an amount of computation that is exponential in the number of non-influential hyperparameters.

### 11.2.3 Grid Search

When the number of hyperparameters is 1 or 2, at most 3, the common practice is to perform what is called a *grid search.* For ordered hyperparameters (e.g., discrete-valued ones like number of hidden units, or continuous-valued ones like learning rate), one would select a small finite set of values which will be explored. See the left of Figure 11.2 for an illustration of a grid of hyperparameter values. How should these lists of values be chosen? In the case of numerical (ordered) hyperparameters, the smallest and largest element of each list is chosen conservatively, based on prior experience with similar experiments, to make sure that the optimal value is very likely to be in the selected range. Typically, a grid search

involves picking values approximately on a *logarithmic scale*, e.g., a learning rate taken within the set $\{.1, .01, 10^{-3}, 10^{-4}, 10^{-5}\}$, or a number of hidden units taken with the set $\{50, 100, 200, 500, 1000, 2000\}$. In effect, practitioners who do a grid search often do it repeatedly, zooming in on one or more regions in the space of hyperparameter values, in order to search for good configurations. When the optimal value found is on at an extreme of the range of values, it means that the range should be extended beyond that optimal value.

The obvious problem with a grid search is that its computational cost grows exponentially with the number of hyperparameters. If there are $m$ hyperparameters, each taking at most $n$ values, then the number of training and evaluation trials required grows as $O(n^m)$. Fortunately, the trials may be run in parallel and exploit loose parallelism (with almost no need for communication between nodes) —but due to the exponential cost of grid search, even parallelization may not provide a satisfactory size of search.

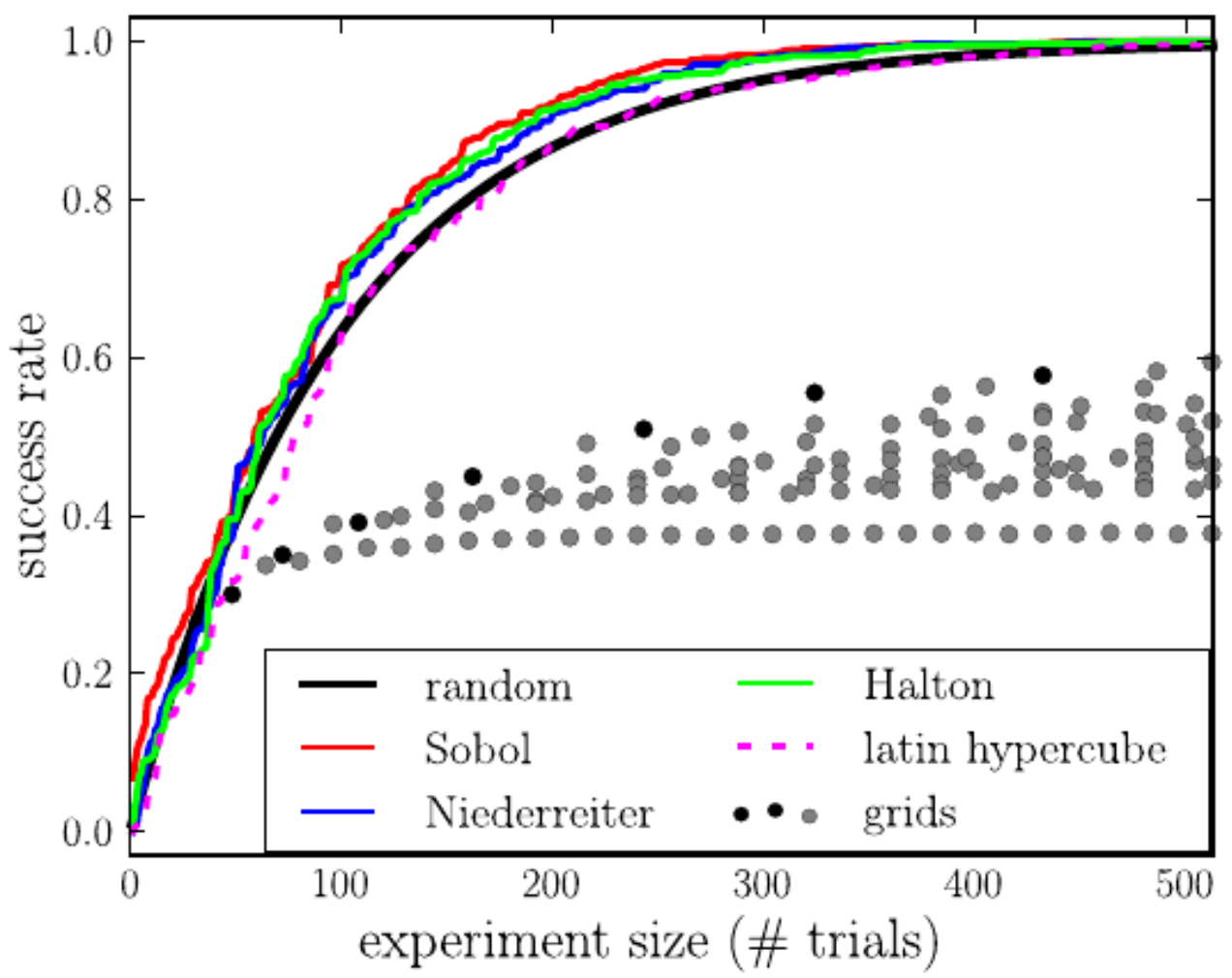


Figure 11.3: Experimental comparison of the convergence of grid search (circles) and random search (black curve) as well as quasi-random search (which better fill the space than a purely random search), see Bergstra and Bengio (2012). The vertical axis is the validation set accuracy and the horizontal axis is the number of training runs. We see that random or quasi-random search can converge much faster than grid search, for the reason illustrated in Figure 11.2.

### 11.2.4 Random Search

Fortunately, there is an alternative to grid search that is as simple to program, more convenient to use, and converges much faster to good values of the hyperparameters: random search (Bergstra and Bengio, 2012).

A random search proceeds as follows. First we define a marginal distribution for each hyperparameter, e.g., a Bernoulli or multinoulli for binary or symbolic hyperparameters, or a uniform distribution on a log-scale. For example, to continue with the examples of number of hidden units and learning rate illustrated for the grid search, we could sample these hyperparameters as follows:

$$\begin{aligned} \texttt{log_learning_rate} &\sim U(-1, -5) \\ \texttt{learning_rate} &= 10^{\text{log_learning_rate}}. \end{aligned} \tag{11.1}$$

where $U(a, b)$ indicates a sample of the uniform distribution in the interval $(a, b)$. Similarly the `log_number_of_hidden_units` may be sampled from $U(\log(50), \log(2000))$.

Note how, unlike in the case of a grid search, one *should not discretize* or bin the values of the hyperpameters. This allows one to explore a larger set of values, and does not incur additional computational cost. In fact, as illustrated in Fig. 11.2, a random search can be exponentially more efficient than a grid search, when there are several hyperparameters whose value do not change much in the result. This is studied at length in Bergstra and Bengio (2012), where the following advantages of of random search are highlighted:

1. The best validation error found as a function of the number of runs tends to converge faster with a random search than a grid search, as illustrated in the experiment from Bergstra and Bengio (2012) shown in Figure 11.3.

2. With random search, if some runs do not complete, it is not necessary to restart them in order to obtain interpretable and usable results, contrary to grid search.

3. If one wants to explore more finely and sample more runs, it is not necessary to restart from scratch: more runs can be added to the past runs, because all the runs are i.i.d., contrary to the grid case.

The main reason why random search finds good solutions faster than grid search is that the there are no wasted experimental runs, unlike in the case of grid search, when two values of a hyperparameter (given values of the other hyperparameters) would give the same result. In the case of grid search, the other hyperparameters would have the same values for these two runs, whereas with random search, they would have different values. Hence if the change between these two values does not marginally make much difference, grid search will unnecessarily repeat

two equivalent experiments whereas random search will still give two independent explorations of the other hyperparameters.

### 11.2.5 Model-Based Hyperparameter Optimization

The search for good hyperparameters can be cast as an optimization problem. The decision variables are the hyperpararameters. The cost to be optimized is the validation set error that results from training using these hyperparameters. In simplified settings where it is feasible to compute the gradient of some differentiable error measure on the validation set with respect to the hyperparameters, we can simply follow this gradient (Bengio *et al.*, 1999; Bengio, 2000; Maclaurin *et al.*, 2015). Unfortunately, in most practical settings, this gradient is unavailable, either due to its high computation and memory cost, or due to hyperparameters having intrinsically non-differentiable interactions with the validation set error, as in the case of discrete-valued hyperparameters.

To compensate for this lack of a gradient, we can build a model of the validation set error, then propose new hyperparameter guesses by performing optimization within this model. Most model-based algorithms for hyperparameter search use a Bayesian regression model to estimate both the expected value of the validation set error for each hyperparameter and the uncertainty around this expectation. Optimization thus involves a tradeoff between exploration (proposing hyperparameters for which there is high uncertainty, which may lead to a large improvement but may also perform poorly) and exploitation (proposing hyperparameters which the model is confident will perform as well as any hyperparameters it has seen so far—usually hyperparameters that are very similar to ones it has seen before). Contemporary approaches to hyperparameter optimization include Spearmint (Snoek *et al.*, 2012), TPE (Bergstra *et al.*, 2011) and SMAC (Hutter *et al.*, 2011).

Currently, we cannot unambiguously recommend Bayesian hyperparameter optimization as an established tool for achieving better deep learning results or for obtaining those results with less effort. Bayesian hyperparameter optimization sometimes performs comparably to human experts, sometimes better, but fails catastrophically on other problems. It may be worth trying to see if it works on a particular problem but is not yet sufficiently mature or reliable. That being said, hyperparameter optimization is an important field of research that, while often driven primarily by the needs of deep learning, holds the potential to benefit not only the entire field of machine learning but the discipline of engineering in general.

One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from

the experiment. This is much less efficient, in the sense of how much information can be gleaned early in an experiment, than manual search by a human practitioner, since one can usually tell early on if some set of hyperparameters is completely pathological. Swersky *et al.* (2014) have introduced an early version of an algorithm that maintains a set of multiple experiments. At various time points, the hyperparameter optimization algorithm can choose to begin a new experiment, to "freeze" a running experiment that is not promising, or to "thaw" and resume an experiment that was earlier frozen but now appears promising given more information.

## 11.3 Debugging Strategies

When a machine learning system performs poorly, it is usually difficult to tell whether the poor performance is intrinsic to the algorithm itself or whether there is a bug in the implementation of the algorithm. Machine learning systems are difficult to debug for a variety of reasons.

In most cases, we do not know a priori what the intended behavior of the algorithm is. In fact, the entire point of using machine learning is that it will discover useful behavior that we were not able to specify ourselves. If we train a neural network on a *new* classification task and it achieves 5% test error, we have no straightforward way of knowing if this is the expected behavior or sub-optimal behavior.

A further difficulty is that most machine learning models have multiple parts that are each adaptive. If one part is broken, the other parts can adapt and still achieve roughly acceptable performance. For example, suppose that we are training a neural net with several layers parameterized by weights $\boldsymbol{W}$ and offsets $\boldsymbol{b}$. Suppose further that we have manually implemented the gradient descent rule for each parameter separately, and we made an error in the update for the biases:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha$$

where $\alpha$ is the learning rate. This erroneous update does not use the gradient at all. It causes the biases to constantly become negative throughout learning, which is clearly not a correct implementation of any reasonable learning algorithm. The bug may not be apparent just from examining the output of the model though. Depending on the distribution of the input, the weights may be able to adapt to compensate for the negative biases.

Most debugging strategies for neural nets are designed to get around one or both of these two difficulties. Either we design a case that is so simple that the correct behavior actually can be predicted, or we design a test that exercises one part of the neural net implementation in isolation.

Some important debugging tests include:

*Visualize the model in action* : If you're training a model to detect objects in images, view some images with the bounding boxes on them. If you're training a generative model of speech, listen to some of the speech samples it produces. This may seem obvious, but it is easy to fall into the practice of only looking at quantitative performance measurements like accuracy or log-likelihood. Directly observing the machine learning model performing its task will help you to determine whether the quantitative performance numbers you're achieving seem reasonable. Evaluation bugs can be some of the most devastating bugs because they can mislead you into believing your system is performing well when it is not.

*Visualize the worst mistakes* : most models are able to output some sort of confidence score for the task they perform. For example, classifiers based on a softmax output layer give the probability assigned to each class. The probability assigned to the most likely class thus gives a confidence score. By viewing the training set examples that are the hardest to model correctly, you can often discover problems with the way the data has been preprocessed or labeled. For example, the Street View transcription system originally had a problem where the address number detection system would crop the image too tightly and omit some of the digits. The transcription network then assigned very low probability to the correct answer on these images. Sorting the images to identify the most confident mistakes showed that there was a systematic problem with the cropping. Modifying the detection system to crop much wider images resulted in much better performance of the overall system, even though the transcription network needed to be able to process greater variation in the position and scale of the address numbers.

*Compare train and test error, evaluate if overfitting or underfitting*: If both train and test error are high and similar in value to each other, this could mean that the model is underfitting, or it could mean that a bug is preventing proper fitting. If training error is low but test error is high, this usually just indicates that the model is overfitting, unless there is some difference in the way the test data is prepared or there is a problem with saving and reloading the model between training and test evaluation. To better ascertain whether one is underfitting or overfitting, it suffices to *vary one of the hyperparameters* and note the effect on the validation error. First, note whether the hyperparameter is one whose increases tends to increase effective capacity or decrease it (see Table 11.1 for some examples). The simplest one is the number of training iterations, which increases effective capacity. Second, verify if the increase in effective capacity (corresponding to the increase or decrease in the hyperparameter value) yields an increase or decrease in validation error. If the former, it means that we are overfitting (at least for that hyperparameter). If the latter, it means that we are

underfitting. It is therefore a good practice to always monitor the evolution of training and validation error as a function of the number of training iterations, since that immediately answers the question, and also tells us if the optimization is going astray (when training error goes up).

*Fit a tiny dataset*: If you have high error on the training set, determine whether it is due to genuine underfitting or a bug. Usually even small models can be guaranteed to fit a sufficiently small dataset. For example, a classification dataset with only one example can be fit just by setting the biases of the output layer correctly. Usually if you cannot train a classifier to correctly label a single example, an auto-encoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is some sort of bug preventing successful optimization on the training set. When trying it out with just a few examples, more of the parameters are exercised and this tests the settings of the optimization hyperparameters, like learning rate.

*Compare symbolic derivatives to numerical derivatives*: If you are using a software framework that requires you to implement your own gradient computations, or if you are adding a new symbolic operation to a symbolic differentiation environment and must define its gradient, then a common source of error is implementing this gradient expression incorrectly. One way to verify that these derivatives are correct is to compare the derivatives computed by your implementation of symbolic differentiation to the derivatives computed by a *finite differences*. Because

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

we can approximate the derivative by using a small, finite $\epsilon$:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

We can improve the accuracy of the approximation by using the *centered difference*:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}.$$

Note that $\epsilon$ cannot be chosen too small because, due to finite-precision computation (and especially the presence of non-linearities), the difference $f(x+\epsilon) - f(x)$ or $f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)$ might end up being numerically zero. This means that the Taylor expansion error can be non-negligeable, in some cases.

Usually, we will want to test the gradient or Jacobian of a vector-valued function $g : \mathbb{R}^m \to \mathbb{R}^n$. Unfortunately, finite differencing only allows us to take a single derivative at a time. We can either run finite differencing $mn$ times to evaluate all of the partial derivatives of $g$, or we can use random projections to construct a 1-dimensional function to test with a single call to finite differencing.

For example, we can apply finite differencing to $f(x)$ where $f(x) = \boldsymbol{u}^T g(\boldsymbol{v}x)$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ are randomly chosen vectors. Computing $f'(x)$ correctly requires being able to back-propagate through $g$ correctly. It is usually a good idea to repeat this test for more than one value of $\boldsymbol{u}$ and $\boldsymbol{v}$ to reduce the chance that the test overlooks mistakes that are orthogonal to the random projection.

If one has access to numerical computation on complex numbers, then there is a very efficient way to numerically estimate the gradient, when feeding the model with complex-valued parameters. Indeed, note that

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2)$$
$$\text{real}(f(x + i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}(\frac{f(x + i\epsilon)}{\epsilon}) = f'(x) + O(\epsilon^2), \tag{11.2}$$

where $i = \sqrt{-1}$. Unlike in the real case above, there is no cancellation effect due to taking the difference between the value of $f$ at different points. Hence, it is possible to use tiny values of $\epsilon$ like $\epsilon = 10^{-150}$, which make the $O(\epsilon^2)$ error insignificant for all practical purposes. This was first remarked by Squire and Trapp (1998).

*Monitor histograms of activations and gradient*: It is often useful to visualize statistics of neural network activations and gradients, collected over a large chunk of data (maybe one epoch). The pre-activation value of hidden units can tell us if the units saturate, or how often they do. For example, for rectifiers, how often are they off? Are there units that are always off? For tanh units, the average of the absolute value of the pre-activations tells us how saturated the unit is. In a deep network where the propagated gradients quickly grow or quickly vanish, optimization may be hampered. Finally, it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves. As suggested by Bottou (2015), we would like the magnitude of parameter updates over a minibatch to represent something like 1% of the magnitude of the parameter, not 50% or 0.001% (which would make the parameters move too slowly). It may be that some groups of parameters are moving at a good pace while others are stalled. When the data have a sparse nature (like in natural language), some parameters may be very rarely updated, and this should be kept in mind when monitoring their evolution. See Glorot and Bengio (2010a) for an early example of using such monitoring to study the effect of initial values and nonlinearities, along with the evolution of training

Finally, many deep learning algorithms provide some sort of guarantee about the results produced at each step. For example, in Part III, we will see some approximate inference algorithms that work by using algebraic solutions to optimization problems. Typically these can be debugged by testing each of their guarantees. Some guarantees that some optimization algorithms offer include

that the objective function will never increase after one step of the algorithm, that the gradient with respect to some subset of variables will be zero after each step of the algorithm, and that the gradient with respect to all variables will be zero at convergence. Usually due to rounding error, these conditions will not hold exactly in a digital computer, so the debugging test should include some tolerance parameter.

# Chapter 12

# Applications

In this chapter, we describe how to put deep learning models to practical use. We begin by discussing the large scale neural network implementations required for most serious AI applications. Next, we review several specific application areas that deep learning has been used to solve. While one goal of deep learning is to design algorithms that are capable of solving a broad variety of tasks, so far some degree of specialization is needed. For example, vision tasks require processing a large number of input features (pixels) per example. Language tasks require modeling a large number of possible values (words in the vocabulary) per input feature.

## 12.1 Large Scale Deep Learning

Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior. It truly is important to emphasize the fact that the number of neurons must be *large*. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use. As we saw in Chapter 1.2.3, network sizes have grown exponentially for the past three decades, yet artificial neural networks are only as large as the nervous systems of insects.

Because the size of neural networks is of paramount importance, deep learning requires high performance hardware and software infrastructure.

### 12.1.1 Fast CPU Implementations

Traditionally, neural networks were trained using the CPU of a single machine. Today, this approach is generally considered insufficient. We now mostly use GPU computing or the CPUs of many machines networked together. Before moving to these expensive setups, researchers worked hard to demonstrate that CPUs could not manage the high computational workload required by neural networks.

A description of how to implement efficient numerical CPU code is beyond the scope of this book, but we emphasize here that careful implementation for specific CPU families can yield large improvements. For example, in 2011, the best CPUs available could run neural network workloads faster when using fixed-point arithmetic rather than floating-point arithmetic. By creating a carefully tuned fixed-point implementation, Vanhoucke *et al.* (2011) obtained a $3\times$ speedup over a strong floating-point system. Each new model of CPU has different performance characteristics, so sometimes floating-point implementations can be faster too. The important principle is that careful specialization of numerical computation routines can yield a large payoff. Other strategies, besides choosing whether to use fixed or floating point, including optimizing data structures to avoid cache misses and using vector instructions. Many machine learning researchers neglect these implementation details, but when they restrict the size of the network one can train, they in turn restrict the machine learning capabilities of the network.

### 12.1.2 GPU Implementations

Most modern neural network implementations are based on graphics processing units. Graphics processing units (GPUs) are specialized hardware components that were originally developed for graphics applications. The consumer market for video gaming systems spurred development of graphics processing hardware. The performance characteristics needed for good video gaming systems turn out to be beneficial for neural networks as well.

Video game rendering requires performing many operations in parallel quickly. Models of characters and environments are specified in terms of lists of 3-D coordinates of vertices. Graphics cards must perform matrix multiplication and division on many vertices in parallel to convert these 3-D coordinates into 2-D on-screen coordinates. The graphics card must then perform many computations at each pixel in parallel to determine the color of each pixel. In both cases, the computations are fairly simple and do not involving much branching compared to the computational workload that a CPU usually encounters. For example, each vertex in the same rigid object will be multiplied by the same matrix; there is no need to evaluate an if statement per-vertex to determine which matrix to multiply by. The computations are also entirely independent of each other, and thus may be

parallelized easily. The computations also involve processing massive buffers of memory, containing bitmaps describing the texture (color pattern) of each object to be rendered. Together, this results in graphics cards having been designed to have a high degree of parallelism and high memory bandwidth, at the cost of having a lower clock speed and less branching capability relative to traditional CPUs.

Neural networks also benefit from the same performance characteristics. Neural networks usually involve large and numerous buffers of parameters, activation values, and gradient values, each of which must be completely updated during every step of training. These buffers are large enough to fall outside the cache of a traditional desktop computer so the memory bandwidth of the system often becomes the rate limiting factor. GPUs offer a compelling advantage over CPUs due to their high memory bandwidth. Neural network training algorithms typically do not involve much branching or sophisticated control, so they are appropriate for neural network hardware. Since neural networks can be divided into multiple individual "neurons" that can be processed independently from the other neurons in the same layer, neural networks easily benefit from the parallelism of GPU computing.

GPU hardware was originally so specialized that it could only be used for graphics tasks. Over time, GPU hardware became more flexible, allowing custom subroutines to be used to transform the coordinates of vertices or assign colors to pixels. In principle, there was no requirement that these pixel values actually be based on a rendering task. These GPUs could be used for scientific computing by writing the output of a computation to a buffer of pixel values. Steinkrau *et al.* (2005) implemented a two-layer fully connected neural network on an early GPU and reported a 3X speedup over their CPU-based baseline. Shortly thereafter, Chellapilla *et al.* (2006) demonstrated that the same technique could be used to accelerate supervised convolutional networks.

The popularity of graphics cards for neural network training exploded after the advent of *General Purpose GPUs*. These GP-GPUs could execute arbitrary code, not just rendering subroutines. NVIDIA's CUDA programming language provided a way to write this arbitrary code in a C-like language. With their relatively convenient programming model, massive parallelism, and high memory bandwidth, GP-GPUs now offer an ideal platform for neural network programming. This platform was rapidly adopted by deep learning researchers soon after it became available (Raina *et al.*, 2009; Ciresan *et al.*, 2010).

Writing efficient code for GP-GPUs remains a difficult task best left to specialists. The techniques required to obtain good performance on GPU are very different from those used on CPU. For example, good CPU-based code is usually designed to read information from the cache as much as possible. On GPU, most

writable memory locations are not cached, so it can actually be faster to compute the same value twice, rather than compute it once and read it back from memory. GPU code is also inherently multi-threaded and the different threads must be coordinated with each other carefully. For example, memory operations are faster if they can be *coalesced*. Coalesced reads or writes occur when several threads can each read or write a value that they need simultaneously, as part of a single memory transaction. Different models of GPUs are able to coalesce different kinds of read or write patterns. Typically, memory operations are easier to coalesce if among $n$ threads, thread $i$ accesses byte $i + j$ of memory, and $j$ is a multiple of some power of 2. The exact specifications differ between models of GPU. Another common consideration for GPUs is making sure that each thread in a group executes the same instruction simultaneously. This means that branching can be difficult on GPU. Threads are divided into small groups called *warps*. Each thread in a warp executes the same instruction during each cycle, so if different threads within the same warp need to execute different code paths, these different code paths must be traversed sequentially rather than in parallel.

Due to the difficulty of writing high performance GPU code, researchers should structure their workflow to avoid needing to write new GPU code in order to test new models or algorithms. Typically, one can do this by building a software library of high performance operations like convolution and matrix multiplication, then specifying models in terms of calls to this library of operations. For example, the machine learning library Pylearn2 (Warde-Farley *et al.*, 2011) specifies all of its machine learning algorithms in terms of calls to the Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012) and cuda-convnet (Krizhevsky, 2010), which provide these high-performance operations. This factored approach can also ease support for multiple kinds of hardware. For example, the same Theano program can run on either CPU or GPU, without needing to change any of the calls to Theano itself. Other libraries like Torch (Collobert *et al.*, 2011b) provide similar features.

### 12.1.3 Large Scale Distributed Implementations

In many cases, the computational resources available on a single machine are insufficient. We therefore want to distribute the workload of training and inference across many machines.

Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as *data parallelism*.

It is also possible to get *model parallelism*, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training.

Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD, but usually we get less than linear

returns in terms of optimization performance. It would be better to allow multiple machines to compute multiple gradient descent steps in parallel. Unfortunately, the standard defintion of gradient descent is as a completely sequential algorithm: the gradient at step $t$ is a function of the parameters produced by step $t-1$.

This can be solved using *asynchronous stochastic gradient descent* (Bengio *et al.*, 2001a; Recht *et al.*, 2011). In this approach, several processor cores share the memory representing the parameters. Each core reads parameters without a lock, then computes a gradient, then increments the parameters without a lock. This reduces the average amount of improvement that each gradient descent step yields, because some of the cores overwrite each other's progress, but the increased rate of production of steps causes the learning process to be faster overall. Dean *et al.* (2012) pioneered the multi-machine implementation of this lock-free approach to gradient descent, where the parameters are managed by a *parameter server* rather than stored in shared memory. Distributed asynchronous gradient descent remains the primary strategy for training large deep networks and is used by most major deep learning groups in industry (Chilimbi *et al.*, 2014; Wu *et al.*, 2015). Academic deep learning researchers typically cannot afford the same scale of distributed learning systems but some research has focused on how to build distributed networks with relatively low-cost hardware available in the university setting (Coates *et al.*, 2013).

### 12.1.4 Model Compression

In many commercial applications, it is much more important that the time and memory cost of running inference in a machine learning model be low than that the time and memory cost of training be low. For applications that do not require personalization, it is possible to train a model once, then deploy it to be used by billions of users. In many cases, the end user is more resource-constrained than the developer. For example, one might train a speech recognition network with a powerful computer cluster, then deploy it on mobile phones.

A key strategy for reducing the cost of inference is *model compression* (Buciluǎ *et al.*, 2006). The basic idea of model compression is to replace the original, expensive model with a smaller model that requires less resources to store and evaluate.

Model compression is applicable when the size of the original model is driven primarily by a need to prevent overfitting. In most cases, the model with the lowest generalization error is an ensemble of several independently trained models. Evaluating all $n$ ensemble members is expensive. Sometimes, even a single model generalizes better if it is large (for example, if it is regularized with dropout).

These large models learn some function $f(\boldsymbol{x})$, but do so using many more parameters than are necessary for the task. Their size is necessary only due to

the limited number of training examples. As soon as we have fit this function $f(\boldsymbol{x})$, we can generate a training set containing infinitely many examples, simply by applying $f$ to randomly sampled points $\boldsymbol{x}$. We then train the new, smaller, model to match $f(\boldsymbol{x})$ on these points. In order to most efficiently use the capacity of the new, small model, it is best to sample the new $\boldsymbol{x}$ points from a distribution resembling the actual test inputs that will be supplied to the model later. This can be done by corrupting training examples or by drawing points from a generative model trained on the original training set.

Alternatively, one can train the smaller model only on the original training points, but train it to copy other features of the model, such as its posterior distribution over the incorrect classes (Hinton *et al.*, 2014, 2015).

### 12.1.5 Dynamic Structure

One strategy for accelerating data processing systems in general is to build systems that have *dynamic structure* in the graph describing the computation needed to process an input. Data processing systems can dynamically determine which subset of many neural networks should be run on a given input. Individual neural networks can also exhibit dynamic structure internally by determining which subset of features (hidden units) to compute given information from the input. This form of dynamic structure inside neural networks is sometimes called *conditional computation* (Bengio, 2013a; Bengio *et al.*, 2013a) , though many kinds of dynamic structure predate this term. Since many components of the architecture may be relevant only for a small amount of possible inputs, the system can run faster by computing these features only when they are needed.

Dynamic structure of computations is a basic computer science principle applied generally throughout the software engineering discipline. The simplest versions of dynamic structure applied to neural networks are based on determining which subset of some group of neural networks (or other machine learning models) should be applied to a particular input.

A venerable strategy for accelerating inference in a classifier is to use a *cascade* of classifiers. The basic idea is that we are trying to detect the presence of a rare object (or event). To know for sure that the object is present, we must use a sophisticated classifier with high capacity, that is expensive to run. However, because the object is rare, we can usually reject inputs as not containing the object with much less computation. In these situations, we can train a sequence of classifiers. The first classifiers in the sequence have low capacity, and are trained to have high recall. In other words, they are trained to make sure we do not wrongly reject an input when the object is present. The final classifier is trained to have high precision. At test time, we run inference by running the classifiers in a sequence, abandoning any example as soon as any one element in the cascade

rejects it. Overall, this allows us to verify the presence of objects with high confidence, using a high capacity model, but does not force us to pay the cost of inference in a high capacity model for every example. The system as a whole has somewhat high capacity just from the use of many models, even if all of the individual models have the same capacity. It is also possible to design the cascade so that models that come later have higher capacity. Viola and Jones (2001) used a cascade of boosted decision trees to implement a fast and robust face detector suitable for use in handheld digital cameras. Their classifier localizes a face using essentially a sliding window approach in which many windows are examined and rejected if they do not contain faces. Another version of cascades uses the earlier models to implement a sort of hard attention mechanism: the early members of the cascade localize an object and later members of the cascade performing further processing on it. For example, Google transcribes address numbers from Street View imagery using a two-step cascade that first locates the address number with one machine learning model and then transcribes it with another (Goodfellow *et al.*, 2014d).

Decision trees themselves are an example of dynamic structure, because each node in the tree determines which of its subtrees should be evaluated for each input. A simple way to accomplish the union of deep learning and dynamic structure is to train a decision tree in which each node uses a neural network to make the splitting decision (Guo and Gelfand, 1992), though this has typically not been done with the primary goal of accelerating inference computations.

In the same spirit, one can use a neural network, called the *gater* to select which one out of several *expert networks* will be used to compute the output, given the current input. The first version of this idea is called the *mixture of experts* (Nowlan, 1990; Jacobs *et al.*, 1991), in which the gater outputs a set of probabilities or weights[1], one per expert, and the final output is obtained by the weighted combination of the experts outputs. In that case, there is no computational saving, but if a single expert is chosen by the gater for each example, we obtain the *hard mixture of experts* (Collobert *et al.*, 2001, 2002), which can considerably speed-up training and inference time.

One major obstacle to using dynamically structured systems is the decreased degree of parallelism that results from the system following different code branches for different inputs. This means that few operations in the network can be described as matrix multiplication or batch convolution on a minibatch of examples. We can write more specialized sub-routines that convolve each example with different kernels or multiply each row of a design matrix by a different set of columns of weights. Unfortunately, these more specialized subroutines are difficult to implement efficiently. CPU implementations will be slow due to the lack of cache

[1] obtained via a softmax non-linearity

coherence and GPU implementations will be slow due to the lack of coalesced memory transactions and the need to serialize warps when members of a warp take different branches. In some cases, these issues can be mitigated by partitioning the examples into groups that all take the same branch, and processing these groups of examples simultaneously. This can be an acceptable strategy for minimizing the time required to process a fixed amount of examples in an offline setting. In a real-time setting where examples must be processed continuously, partitioning the workload can result in load-balancing issues. For example, if we assign one machine to process the first step in a cascade and another machine to process the last step in a cascade, then the first will tend to be overloaded and the last will tend to be underloaded. Similar issues arise if each machine is assigned to implement different nodes of a neural decision tree.

### 12.1.6 Specialized Hardware Implementations of Deep Networks

Since the early days of neural networks research, hardware designers have worked on specialized hardware implementations that could speed up training and/or inference of neural network algorithms. See early and more recent reviews of specialized hardware for deep networks (Lindsey and Lindblad, 1994; Beiu *et al.*, 2003; Misra and Saha, 2010).

Different forms of specialized hardware (Graf and Jackel, 1989; Mead and Ismail, 2012; Kim *et al.*, 2009; Pham *et al.*, 2012; Chen *et al.*, 2014a,b) have been considered over the last decades, starting with ASICs (application-specific integrated circuit), either with digital (based on binary representations of numbers), analog (Graf and Jackel, 1989; Mead and Ismail, 2012) (based on physical implementations of continuous values as voltages or currents) or hybrid implementations (combining digital and analog components), and in recent years with the more flexible FPGA (field programmable gated array) implementations (where the particulars of the circuit can be written on the chip after it has been built).

Whereas software implementations on general-purpose processing units (CPUs and GPUs) typically use 32 or 64 bits precision floating point representations of numbers, it has long been known that it was possible to use less precision, at least at inference time (Holt and Baker, 1991; Holi and Hwang, 1993; Presley and Haggard, 1994; Simard and Graf, 1994; Wawrzynek *et al.*, 1996; Savich *et al.*, 2007). This has become a more pressing issue in recent years as deep learning has gained in popularity in industrial products, and as the great impact of faster hardware was demonstrated with GPUs. Another factor that motivates current research on specialized hardware for deep networks is that the rate of progress of a single CPU or GPU core has slowed down, and most recent improvements in computing speed have come from parallelization across cores (either in CPUs or GPUs). This is very different from the situation of the 1990's (the previous neural

network era) where the hardware implementations of neural networks (which might take two years from inception to availability of a chip) could not keep up with the rapid progress and low prices of general-purpose CPUs. Building specialized hardware is thus a way to push the envelope further, at a time when new hardware designs are being developed for low-power devices such as phones, aiming for general-public applications of deep learning (e.g., with speech, computer vision or natural language).

Recent work on low-precision implementations of backprop-based neural nets (Vanhoucke *et al.*, 2011; Courbariaux *et al.*, 2015; Gupta *et al.*, 2015) suggests that between 8 and 16 bits of precision can suffice for using or training deep neural networks with back-propagation. What is clear is that more precision is required during training than at inference time, and that some forms of dynamic fixed point representation of numbers can be used to reduce how many bits are required per number. Whereas fixed point numbers are restricted to a fixed range (which corresponds to a given exponent in a floating point representation), dynamic fixed point representations share that range among a set of numbers (such as all the weights in one layer). Using fixed point rather than floating point representations and using less bits per number reduces the surface area, power requirements and computing time needed for performing multiplications, and multiplications are the most demanding of the operations needed to use or train a modern deep network with backprop.

## 12.2 Computer Vision

Computer vision has traditionally been one of the most active research areas for deep learning applications. Many of the most popular standard benchmark tasks for deep learning algorithms are forms of object recognition or optical character recognition.

Computer vision is a very broad field encompassing a wide variety of ways of processing images, and an amazing diversity of applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities. As an example of the latter category, one recent computer vision application is to recognize sound waves from the vibrations they induce in objects visible in a video (Davis *et al.*, 2014). Most deep learning research on computer vision has not focused on such exotic applications that expand the realm of what is possible with imagery but rather a small core of AI goals aimed at replicating human abilities. Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of

symbols from an image, or labeling each pixel in an image with the identity of the object it belongs to. Because generative modeling has been a guiding principle of deep learning research, there is also a large body of work on image synthesis using deep models. While image synthesis *ex nihilo* is usually not considered a computer vision endeavor, models capable of image synthesis are usally useful for image restoration, a computer vision task involving repairing defects in images or removing objects from images.

### 12.2.1 Preprocessing

Many application areas require sophisticated preprocessing because the original input comes in a form that is difficult for many deep learning architectures to represent. Computer vision usually requires relatively little of this kind of preprocessing. The images should be standardized so that their pixels all lie in the same, reasonable range, like [0,1] or [-1, 1]. Mixing images that lie in [0,1] with images that lie in [0, 255] will usually result in failure. This is the only kind of preprocessing that is strictly necessary. Many computer vision architectures require images of a standard size, so images must be cropped or scaled to fit that size. However, even this rescaling is not always strictly necessary. some convolutional models are able to process variable size input if their output varies in size with the input or if they use Some convolutional models accept variably-sized inputs and dynamically adjust the size of their pooling regions to keep the output size constant (Waibel *et al.*, 1989). Other convolutional models have variable-sized output that automatically scales in size with the input, such as models that denoise or label each pixel in an image (Hadsell *et al.*, 2007).

Many other kinds of preprocessing are less necessary but help to reduce the size of the model required to obtain good accuracy on the training set, the amount of time required for training, or the size of the training set required to obtain good accuracy on the test set.

Any form of preprocessing that removes some of the complexity from the vision task will accomplish both of these goals. Simpler tasks do not require as large of models to solve, and simpler solutions are more likely to generalize well. Preprocessing of this kind is usually designed to remove some kind of variability in the input data that is easy for a human designer to describe and that the human designer is confident has no relevance to the task. When training with large datasets and large models, this kind of preprocessing is often unnecessary, and it is best to just let the model learn which kinds of variability it should become invariant to. For example, the AlexNet system for classifying ImageNet only has one preprocessing step: subtracting the mean across training examples of each pixel (Krizhevsky *et al.*, 2012a).

Another approach to preprocessing is to artificially introduce more variation

into the training set. *Dataset augmentation* gives the model more training data without requiring the collection of as much real data. This kind of preprocessing usually increases the optimal model size and the amount of time required for training.

### Contrast Normalization

One of the most obvious sources of variation that can be safely removed for many tasks is the amount of contrast in the image. Contrast simply refers to the magnitude of the difference between the bright and the dark pixels in an image. There are many ways of quantifying the contrast of an image. In the context of deep learning, contrast usually refers to the standard deviation of the pixels in an image or region of an image. Suupose we have an image represented by a tensor $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$, with $X_{i,j,0}$ being the red intensity at row $i$ and column $j$, $X_{i,j,1}$ giving the green intensity and $X_{i,j,2}$ giving the green intensity. Then the contrast of the entire image is given by

$$\sqrt{\frac{1}{3rc} \sum_{i=1}^{r} \sum_{j=1}^{c} \sum_{k=1}^{3} \left(X_{i,j,k} - \bar{\mathbf{X}}\right)^2}$$

where $\bar{\mathbf{X}}$ is the mean intensity of the entire image:

$$\bar{\mathbf{X}} = \frac{1}{3rc} \sum_{i=1}^{r} \sum_{j=1}^{c} \sum_{k=1}^{3} X_{i,j,k}.$$

*Global contrast normalization* (GCN) aims to prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant $s$. This approach is complicated by the fact that no scaling factor can change the contrast of a zero-contrast image (one whose pixels all have equal intensity). Images with very low but non-zero contrast often have little information content. Dividing by the true stand deviation usually accomplishes nothing more than amplifying sensor noise or compression artifacts in such cases. This motivates introducing a small, positive regularization parameter $\lambda$ to bias the estimate of the standard deviation. Alternately, one can constrain the denominator to be at least $\epsilon$. Given an input image $\mathbf{X}$, GCN produces an output image $\mathbf{X}'$, defined such that

$$X'_{i,j,k} = s \frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc} \sum_{i=1}^{r} \sum_{j=1}^{c} \sum_{k=1}^{3} \left(x_{i,j,k} - \bar{X}\right)^2}\right\}}. \tag{12.1}$$

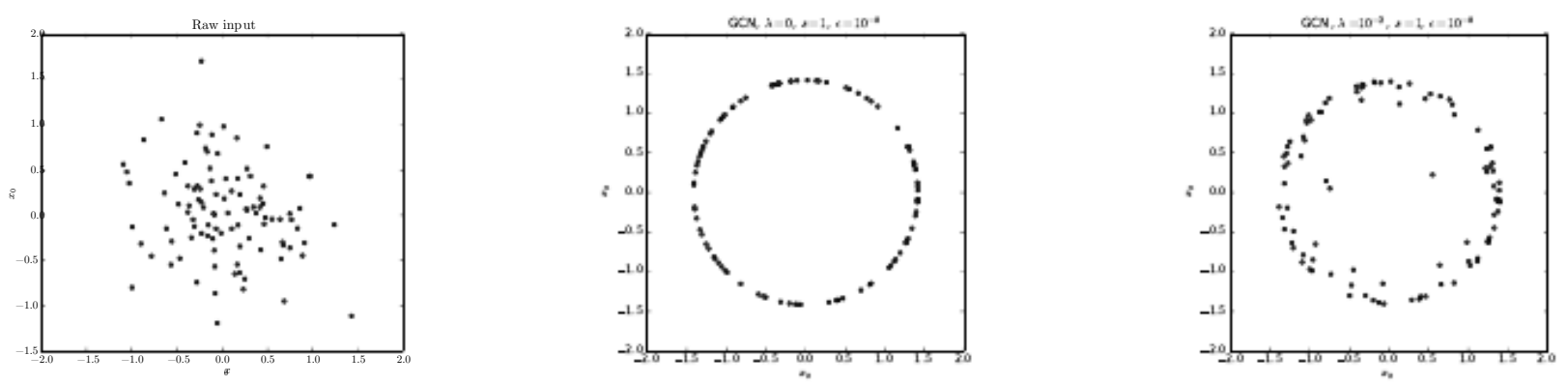


Figure 12.1: GCN maps examples onto a sphere. *Left)* Raw input data may have any norm. *Center)* GCN with $\lambda = 0$ maps all non-zero examples perfectly onto a sphere. *Right)* Regularized GCN, with $\lambda > 0$, draws examples toward the sphere but does not completely discard the variation in their norm.

Datasets consisting of large images cropped to interesting objects are unlikely to contain any images with nearly constant intensity. In these cases, it is safe to practically ignore the small denominator problem by setting $\lambda = 0$ and avoid division by 0 in extremely rare cases by setting $\epsilon$ to an extremely low value like $10^{-8}$. This is the approach used by Goodfellow *et al.* (2013a) on the CIFAR-10 dataset. Small images cropped randomly are more likely to have nearly constant intensity, making aggressive regularization more useful. Coates *et al.* (2011) used $\epsilon = 0$ and $\lambda = 10$ on small, randomly selected patches drawn from CIFAR-10.

The scale parameter $s$ can usually be set to 1, as done by Coates *et al.* (2011), or chosen to make each individual pixel have standard deviation across examples close to 1, as done by Goodfellow *et al.* (2013a).

Note that the standard deviation in equation 12.1 is just a rescaling of the $L^2$ norm of the image. It is preferable to define GCN in terms of standard deviation so that the same $s$ may be used regardless of image size. However, this observation can be useful because it helps to understand GCN as mapping examples to a spherical shell. See Fig. 12.1 for an illustration. This can be a useful property because neural networks are often better at responding to directions in space rather than exact locations. Responding to multiple distances in the same direction requires hidden units with collinear weight vectors but different biases. Such coordination can be difficult for the learning algorithm to discover. Additionally, many shallow graphical models have problems with representing multiple separated modes along the same line. GCN avoids these problems by reducing each example to a direction rather than a direction and a distance.

Counterintuitively, there is a preprocessing operation known as *sphering* and it is not the same operation as GCN. Sphering does not refer to making the data lie on a spherical shell, but rather to rescaling the principal components to have equal variance, so that the multivariate normal distribution used by PCA has spherical contours. Sphering is more commonly known as *whitening*.

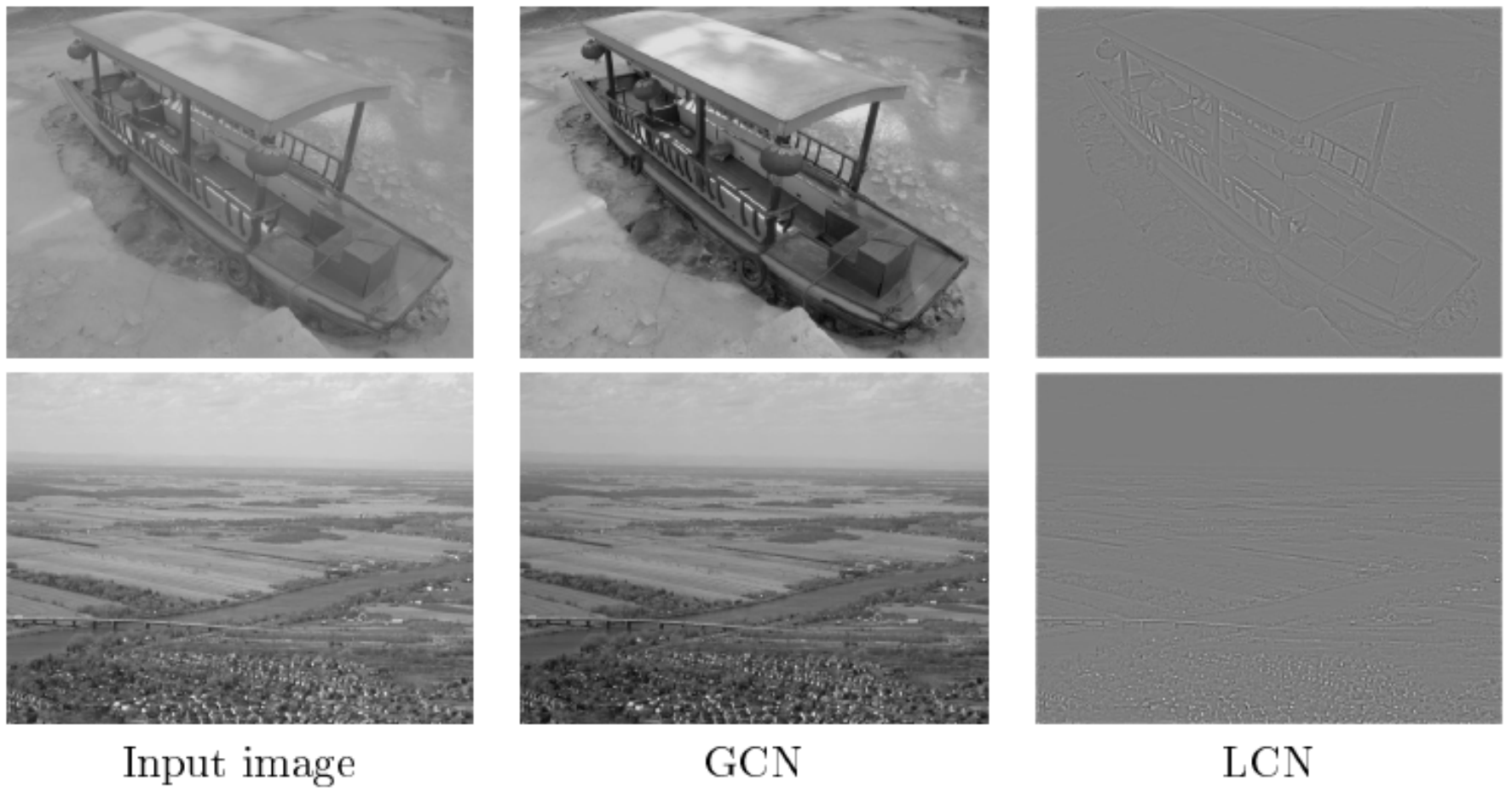


Figure 12.2: A comparison of global and local contrast normalization. Visually, the effects of global contrast normalization are subtle. It places all images on roughly the same scale, which reduces the burden on the learning algorithm to handle multiple scales. Local contrast normalization modifies the image much more, discarding all regions of constant intensity. This allows the model to focus on just the edges. Regions of fine texture, such as the houses in the second row, may lose some detail due to the bandwidth of the normalization kernel being too high.

Global contrast normalization will often fail to highlight image features we would like to stand out, such as edges and corners. If we have a scene with a large dark area and a large bright area (such as a city square with half the image in the shadow of a building) then global contrast normalization will ensure there is a large difference between the brightness of the dark area and the brightness of the light area. It will not, however, ensure that edges within the dark region stand out.

This motivates *local contrast normalization*. Local contrast normalization ensures that the contrast is normalized across each small window, rather than over the image as a whole. See Fig. 12.2 for a comparison of global and local contrast normalization.

Various definitions of local contrast normalization are possible. In all cases, one modifies each pixel by subtracting a mean of nearby pixels and dividing by a standard deviation of nearby pixels. In some cases, this is literally the mean and standard deviation of all pixels in a rectangular window centered on the pixel to be modified (Pinto *et al.*, 2008). In other cases, this is a weighted mean and weighted standard deviation using Gaussian weights centered on the pixel to be modified. In the case of color images, some strategies process different color

channels separately while others combine information from different channels to normalize each pixel (Sermanet *et al.*, 2012).

Local contrast normalization can usually be implemented efficiently by using separable convolution (see Sec. 9.8) to compute feature maps of local means and local standard deviations, then using elementwise subtraction and elementwise division on different feature maps.

Local contrast normalization is a differentiable operation and can also be used as a nonlinearity applied to the hidden layers of a network, as well as a preprocessing operation applied to the input.

As with global contrast normalization, we typically need to regularize local contrast normalization to avoid division by zero. In fact, because local contrast normalization typically acts on smaller windows, it is even more important to regularize. Smaller windows are more likely to contain values that are all nearly the same as each other, and thus more likely to have zero standard deviation.

### Dataset Augmentation

As described in Sec. 7.5, it is easy to improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class. Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations. As described before, classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset. In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation. These schemes include random perturbation of the colors in an image (Krizhevsky *et al.*, 2012a) and non-linear geometric distortions of the input (LeCun *et al.*, 1998b).

## 12.3 Speech Recognition

The task of speech recognition consists in mapping an acoustic signal corresponding to a spoken natural language utterance into the corresponding sequence of words intended by the speaker. If we denote by $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_T)$ the input sequence of acoustic vectors (describing the recorded sounds in discrete time units such as the traditional 20ms frames), and by $\boldsymbol{y} = (y_1, y_2, \ldots y_N)$ the target output or linguistic sequence (e.g., whose elements are words or characters from a natural language), the Automatic Speech Recognition (ASR) task can be described as looking for a function $f_{\text{ASR}} \approx f^*_{\text{ASR}}$, where $f^*_{\text{ASR}}$ finds the most likely linguistic

sequence $\boldsymbol{y}$ given the acoustic sequence $\boldsymbol{X}$:

$$f^*_{\text{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X}) \tag{12.2}$$

where $P^*$ is the true conditional distribution relating the inputs $\boldsymbol{X}$ to the targets $\boldsymbol{y}$.

Since the 80's and until about 2009-2012, the state-of-the art for speech recognition was held by systems combining hidden Markov models (HMMs) and Gaussian Mixture Models (GMMs). GMMs modeled the association between acoustic features and phonemes, while HMMs modeled the sequential structure of speech (which sequence of Gaussian distributions characterize a phoneme, a word?). HMMs are briefly described in Sec. 19.4.2 (Rabiner, 1989), along with GMMs. Speech recognition was one of the first areas to which neural networks were applied, in the late 80's and early 90's (Bourlard and Wellekens, 1989; Waibel *et al.*, 1989; Robinson and Fallside, 1991; Bengio *et al.*, 1991, 1992b; Konig *et al.*, 1996). Neural net based speech recognition performanced approximately matched the performance of GMMs+HMMs systems at that time. For example, Robinson and Fallside (1991) achieved 26% phoneme error rate on the TIMIT (Garofolo *et al.*, 1993) corpus (with 39 phonemes to discriminate against), which was better or comparable to HMM-based systems. TIMIT has been since then a benchmark for phoneme recognition, playing a role similar to MNIST for object recognition. However, with the complex engineering involved in software systems for speech recognition and effort that had been invested in building these systems on the basis of GMMs+HMMs, the industry did not see a compelling argument for switching to neural networks: as a consequence, both academic and industrial research in using neural nets for speech recognition remained mostly dormant until the late 2000's.

It turned out that with *much larger models*, *deeper models* (more hidden layers), and training with much larget datasets, neural nets could very advantageously replace the GMMs for the task of associating acoustic features to phonemes (or sub-phonemic states). Starting in 2009, unsupervised pretraining was used to build stacks of RBMs taking spectral acoustic representations in a fixed-size input window (around a center frame) and predicting the conditional probabilities of HMM states for that center frame. Early results on the TIMIT dataset suggested that training such deep networks actually helped to significantly improve the HMM recognition rate on TIMIT (Mohamed *et al.*, 2012), bringing down the phoneme error rate from about 26% to 23%. This was quickly followed up by work to expand the architecture from phoneme recognition (which is what TIMIT is focused on) to large-vocabulary speech recognition (Dahl *et al.*, 2012), which cares not just about recognizing phonemes but about recognizing sequences of words from a large vocabulary. By that time, several of the major speech groups

in industry had started exploring deep learning in collaboration with academic researchers. Hinton *et al.* (2012a) describe the breakthroughs achieved by these collaborators, which are now deployed in products such as mobile phones.

As it turned out later, as these groups explored larger and larger labeled datasets and incorporated some of the methods for initializing, training, and setting up the architecture of deep nets, they realized that the unsupervised pre-training phase was either unnecessary or did not bring any significant improvement.

These breakthroughs in recognition performance for word error rate in speech recognition were unprecedented (around 30% improvement) and were following a long period of about ten years during which error rates did not improve much with the traditional GMM+HMM technology, in spite of the continuously growing size of training sets (see Figure 2.4 of Deng and Yu (2014)). This created a rapid shift in the speech recognition community towards deep learning, at conferences such as ICASSP. In a matter of two years, most of the industrial products for speech recognition incorporated that innovation and this interest spurred a new wave of explorations for deep learning algorithms and architectures, which is still ongoing.

One of these innovations was the use of convolutional networks (Chapter 9) instead of fully-connected feedforward networks (Sainath *et al.*, 2013). In that case the input spectrogram is seen not as one long vector but as an image, with one axis corresponding to time and the other to frequency of spectral components.

Another important push, still ongoing, has been towards end-to-end deep learning speech recognition systems, without the need for the HMM. The first major breakthrough in this direction came from Graves *et al.* (2013) which trained a deep LSTM RNN (see Sec. 10.7.4), using MAP inference over the frame-to-phoneme alignment, as in LeCun *et al.* (1998c) and in the the CTC framework (Graves *et al.*, 2006; Graves, 2012), described in more detail in Sec. 19.4.1. A deep RNN (Graves *et al.*, 2013) has state variables from several layers at each time step, giving the unfolded graph two kinds of depth: ordinary depth due to a stack of layers, and depth due to time unfolding. This work brought the phoneme error rate on TIMIT to a record low of 17.7%. See Pascanu *et al.* (2014a); Chung *et al.* (2014) for other variants of deep RNNs, applied in other settings.

Following that push, the idea was introduced of using an attention mechanism to let the system learn how to "align" the acoustic-level information with the phonetic-level information Chorowski *et al.* (2014).

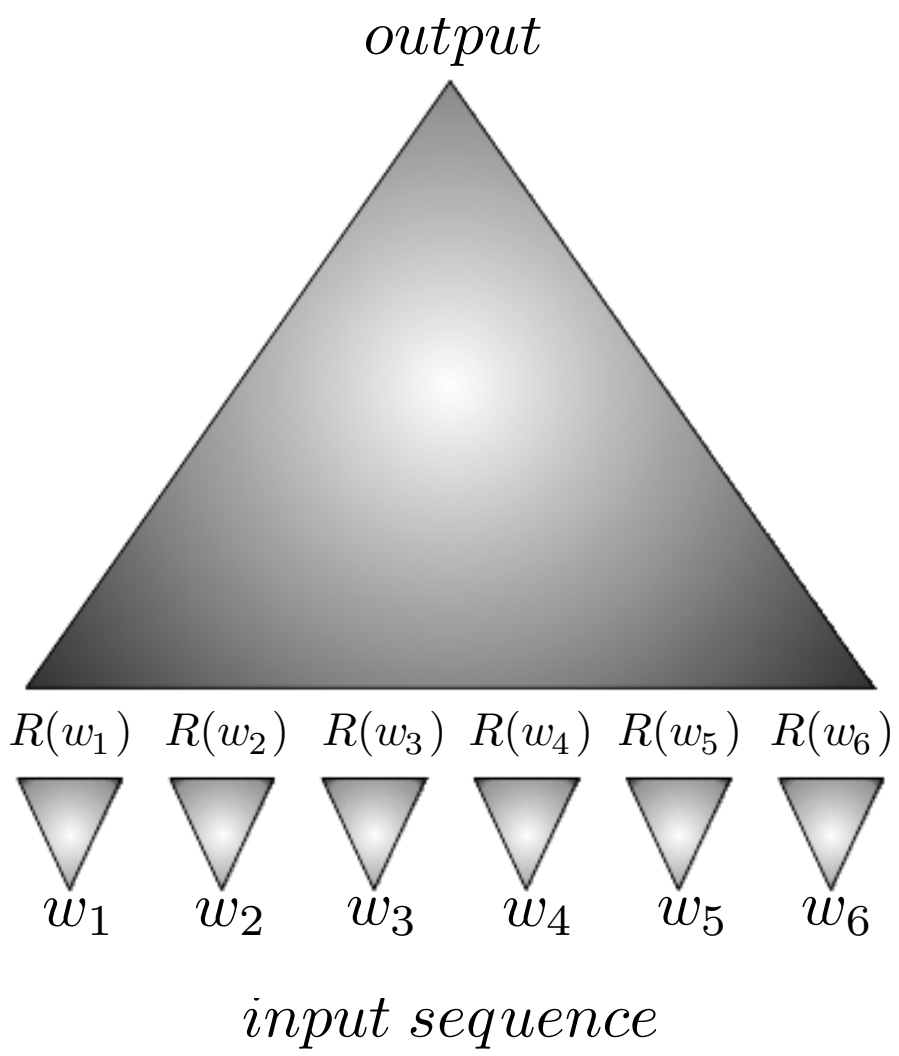


Figure 12.3: Neural language models and their extensions can always be decomposed into two components: (1) the word embeddings, i.e., a mapping from any word index (a symbol) to a learned vector and (2) other parameters dedicated to the task at hand (such as predicting the next word, or translating one sentence into another), based on those representations. The training objective is defined in terms of the output of the second component. It is the second component that drives the learning of the word embeddings in such a way as to make similar words (according to the task) share attributes or dimensions in their embeddings.

## 12.4 Natural Language Processing and Neural Language Models

Natural language processing includes applications such as language modeling and machine translation. As with the other applications discussed in this chapter, very generic neural network techniques can be successfully applied to natural language processing. However, to achieve excellent performance and scale well to large applications, some domain-specific strategies become important. Natural language modeling usually forces us to use some of the many techniques that are specialized for processing sequential data. In many case, we choose to regard natural language as a sequence of words, rather than a sequence of individual characters. In this case, because the total number of possible words is so large, we are modeling an extremely high-dimensional and sparse discrete space. Several strategies have been developed to make models of such a space efficient, both in a computational and in a statistical sense.

### 12.4.1 Historical Perspective

The idea of distributed representations for symbols was introduced by Rumelhart *et al.* (1986a) in one of the first explorations of back-propagation, with symbols corresponding to the identity of family members and the neural network capturing the family relationships between family members, e.g., with examples of the form (Colin, Mother, Victoria). It turned out that the first layer of the neural network learned a representation of each family member, with learned features, e.g. for Colin, representing which family tree Colin was in, what branch of that tree he was in, what generation he was from, etc. One can think of these learned features as a set of attributes and the rest of the neural network computing micro-rules relating these attributes together in order to obtain the desired predictions, e.g., who is the mother of Colin? A similar idea was the basis of the research on neural language model started by Bengio *et al.* (2001b), where this time each symbol represented a word in a natural languge vocabulary, and the task was to predict the next word given a few previous ones. Instead of having a small set of symbols, we have a vocabulary with tens or hundreds of thousands of words (and nowadays it goes up to the million, when considering proper names and misspellings). This raises serious computational challenges, discussed below in Sec. 12.4.4. The basic idea of a neural language models and their extensions, e.g., for machine translation, is illustrated in Figure 12.3 and a specific instance (which was used by Bengio *et al.* (2001b)) is illustrated in Figure 12.3. Figure 12.3 explains the basic of idea of splitting the model into two parts, one for the word embeddings (mapping symbols to vectors) and one for the task to be performed. Sometimes, different maps can be used, e.g., for input words and output words, as in Figure 12.3, or in

neural machine translation models (Sec. 12.4.6).

Earlier work had looked at modeling sequences of characters in text using neural networks (Miikkulainen and Dyer, 1991; Schmidhuber, 1996), but it turned out that working with word symbols worked better as a language model and, more importantly, immediately yielded *word embeddings*, i.e., interpretable representations of words, as illustrated in Figures 12.5 and 12.6. Actually, these compelling 2-dimensional visualization arose thanks to the the development of the t-SNE algorithm (van der Maaten and Hinton, 2008a) in 2008, and the first visualization of word embeddings was made by Joseph Turian in 2009: these t-SNE visualizations of word embedding quickly became a standard tool to understand the learned word representations.

The early efforts at training language models typically yielded neural nets that did not beat an $n$-gram model by themselves, but when adding the probability prediction coming from the neural net and from the $n$-gram model, one would typically get substantial gains in log-likelihood (Bengio *et al.*, 2003a).

An important development after the demonstration that neural language models could be used to improve upon classical $n$-gram models in terms of negative log-likelihood (also called perplexity in the language modeling literature) has been the demonstration that these improved language models could yield an improvement in word error rate for state-of-the-art speech recognition systems (Schwenk and Gauvain, 2002a, 2005; Schwenk, 2007). The same technique (replacing the $n$-gram by a combination of $n$-gram and neural language model) was then used to improve classical statistical machine translation systems (Schwenk *et al.*, 2006; Schwenk, 2010).

More developments of the original model are described in the sections below.

### 12.4.2 $n$-grams

$n$-grams are sequences of $n$ tokens, where tokens can represent words or other discrete entities depending on the application. $n$-gram models are estimators of conditional probabilities based on counting relative frequencies of occurrences of $n$-grams. $n$-gram models have been the core building block of statistical language modeling for many decades (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999). Like RNNs, they are based on the product rule (or chain rule) decomposition of the joint probability into conditionals, Eq. 10.6. All of these models use estimates of $P(x_t \mid x_1, \ldots, x_{t-1})$ to compute $P(x_1, \ldots, x_T)$. Models based on $n$-grams have the following additional properties:

1. They estimate these conditional probabilities based only on the last $n-1$ values (to predict the next one)

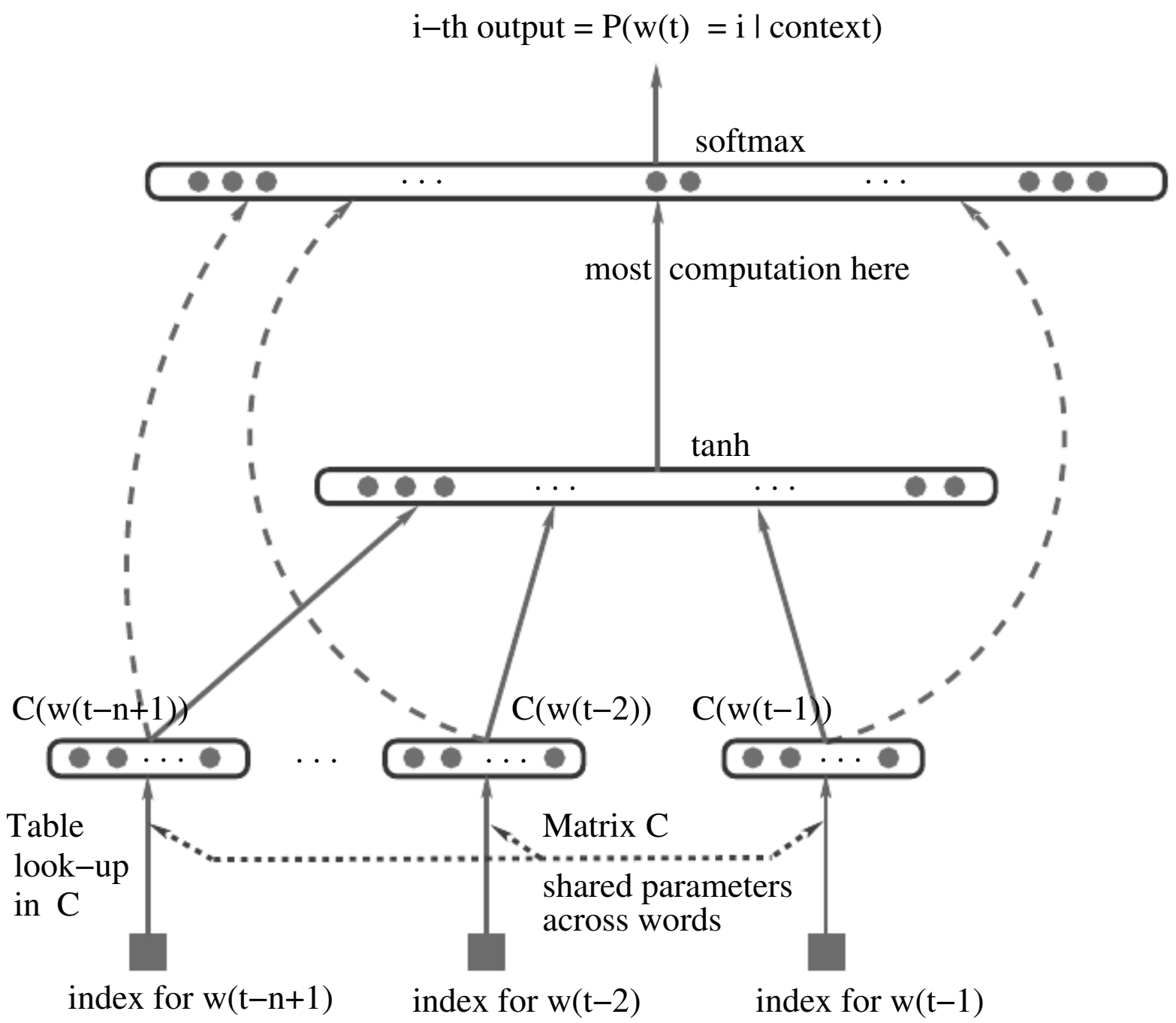


Figure 12.4: This is the original architecture for a neural language model that was developed by Bengio *et al.* (2001b) and is a special case of the general architecture for neural language-related models illustrated in Figure 12.3. Here the "second component" is an ordinary MLP with a single hidden layer and a very large softmax output layer predicting the probability of the next word (given the previous words seen in input).

2. They assume that the data is discrete. Specifically, $x_t$ is a token taken from a finite set $\mathbb{V}$ (for vocabulary).

3. The conditional probability estimates are obtained from frequency counts of all the observed $n$-grams.

The names *unigram* (for $n$=1), *bigram* (for $n$=2), *trigram* (for $n$=3), and *$n$-gram* in general, derive from the Latin prefixes for the corresponding numbers and the Greek suffix "-gram" denoting something that is written. Typically the grams in an $n$-gram model are words or characters.

To train an $n$-gram model with maximum likelihood, we simply gather counts of $n$-grams. The probability of an $n$-gram, $p_n(x_1, \ldots, x_n)$, is given by the number of times we saw that $n$-gram in the training data, divided by the total number of $n$-grams in the training set.

Usually we train both an $n$-gram model and an $n-1$ gram model simultaneously. This makes it easy to compute

$$p(x_t \mid x_{t-n+1}, \ldots, x_t) = p_n(x_{t-n+1}, \ldots, x_t)/p_{n-1}(x_{t-n+1}, x_{t-1}) \tag{12.3}$$

simply by looking up two counts. For this to exactly reproduce inference in $p_n$, we must omit the final character from each sequence when we train $p_{n-1}$.

As an example, we demonstrate how to use trigram and bigram counts to estimate the probability of the sentence "`THE DOG RAN AWAY`." The first words of the sentence cannot be handled by the default formula based on conditional probability because there is no context at the beginning of the sentence. Instead, we must use the marginal probability over words at the start of the sentence. We thus evaluate $p_3(\texttt{THE DOG RAN})$. Finally, the last word may be predicted using the typical case, of using the conditional distribution $p(\texttt{AWAY} \mid \texttt{DOG RAN})$. Putting this together with Eq.
refeq:ml-ngram, we obtain:

$$p(\texttt{THE DOG RAN AWAY}) = p_3(\texttt{THE DOG RAN})p_3(\texttt{DOG RAN AWAY})/p_2(\texttt{DOG RAN}).$$

A fundamental limitation of maximum likelihood for $n$-gram models is that $p_n$ is very likely to be zero in many cases, even though the tuple $(x_{t-n+1}, \ldots, x_t)$ may appear in the test set. This can cause two different kinds of catastrophic outcomes. When $p_{n-1}$ is zero, the ratio is undefined, so the model does not even produce a sensible output. When $p_{n-1}$ is non-zero but $p_n$ is zero, the test log-likelihood is $-\infty$. To avoid such catastrophic outcomes, most $n$-gram models employ some form of *smoothing*. Smoothing techniques shift probability mass from the observed tuples to unobserved ones that are similar. See Chen and Goodman (1999) for a review and empirical comparisons. One basic technique consists in

adding non-zero probability mass to all of the possible next symbol values. This method can be justified as Bayesian inference with a uniform or Dirichlet prior over the the count parameters. Another very popular idea consists in backing off, or mixing (as in mixture model), the higher-order $n$-gram predictor with all the lower-order ones (with smaller $n$). Back-off methods look-up the lower-order $n$-grams if the frequency of the context $x_{t-1}, \ldots, x_{t-n+1}$ is too small. More formally, they estimate the distribution over $x_t$ by using contexts $x_{t-n+k}, \ldots, x_{t-1}$, for increasing $k$, until a sufficiently reliable estimate is found.

$n$-gram models can be interpreted as non-parametric or as parametric. The vocabulary size $|\mathbb{V}|$ and the $n$ impose a hard upper limit on the number of count parameters that will be stored, so asymptotically, as the size of the training set approaches infinity, the number of parameters remains constant. In this view, $n$-gram models are parametric, but with very sparse parameters. In realistic use cases, most $n$-grams are not observed during training, and the size of the model description scales with the size of the training set. In this view, $n$-gram models are non-parametric. Finally, the learning algorithm encompassing both hyperparameter tuning and parameter learning is trivially non-parametric, in the sense that almost any learning algorithm with configurable capacity is non-parametric, because the hyperparameter selection algorithm can increase $n$ arbitrarily.

Classical $n$-gram models suffer from the curse of dimensionality (a general problem in machine learning. There are $|\mathbb{V}|^n$ possible $n$-grams to count and only a limited amount of training data. To overcome this problem, a language model must be able to share knowledge between one word and other semantically similar words.

To overcome this problem, *class-based language models* (Brown *et al.*, 1992; Ney and Kneser, 1993; Niesler *et al.*, 1998) introduce the notion of word categories in order to share statistical strength between words that are semantically related. The idea is to use a clustering algorithm to partition the set of words into clusters or classes, based on their co-occurence frequencies with other words. The model can then use word class IDs rather than individual words to represent the context on the right side of the conditioning bar. Composite models combining word-based and class-based models via mixing or back-off are also possible. Although word classes clearly gives a way to generalize between sequences in which some word is replaced by another of the same class, much information is lost in this representation.

### 12.4.3 How Neural Language Models can Generalize Better

Neural language models are a class of deep model designed to overcome the curse of dimensionality problem. Unlike class-based $n$-gram models, neural language models are able to recognize that two words are similar without losing the ability

to encode each word as distinct from the other.

The fundamental reason why neural language models can break the barrier encountered with models based on $n$-grams is that neural language models can share statistical strength between one word (and its context) and other similar words and contexts. This sharing is enabled by learning a distributed representation of each word. For example, if the word `dog` and the word `cat` map to representations that share many attributes (except maybe some indicator of being feline or not, for example), then sentences that contain the word `cat` can inform the predictions that will be made by the model for sentences that contain the word `dog`, and vice-versa. Because there are many such attributes, there are many ways in which generalization can happen, transfering information from each training sentence to an exponentially large number of semantically related sentences. The curse of dimensionality requires the model to generalize to a number of sentences that is exponential in the sentence length. The model counters this curse by relating each training sentence to an exponential number of similar sentences.

We sometimes call these word representations "word embeddings." In this interpretation, we view the raw symbols as points in a space of dimension equal to the vocabulary size. The word representations embed those points in a semantic space of lower dimension. In the original space, every word is represented by a one-hot vector, so every pair of words is at Euclidean distance $\sqrt{2}$ from each other. In the embedding space, semantically similar words (or any pair of words sharing some "features" learned by the model) are close to each other. Figure 12.5 demonstrates that semantically related words appear near each other in the embedding space.

Figure 12.6 zooms in on specific areas of such a picture of word embeddings to show more clearly how semantically similar words end up with representations that are close to each other.

### 12.4.4 High-Dimensional Outputs

A common problem in natural language applications is that it can be very computationally expensive to represent an output distribution over the choice of a word, because the vocabulary size is large. The naive approach to representing such a distribution is to apply an affine transformation from a hidden representation to the output space, then apply the softmax function. Suppose we have a vocabulary $\mathbb{V}$ with size $|\mathbb{V}|$. The weight matrix describing the linear component of this affine transformation is very large, because its output dimension is $|\mathbb{V}|$. This imposes a high memory cost to represent the matrix, and a high computational cost to multiply by it. Because the softmax is normalized across all $|\mathbb{V}|$ outputs, it is necessary to perform the full matrix multiplication at training time as well as test time—we cannot calculate only the dot product with the weight vector for the

Figure 12.5: Word embeddings obtained from neural language models tend to cluster by semantic categories, as visualized here via t-SNE dimensionality reduction and coloring of words by such categories. Reproduced with permission by Chris Olah from `http://colah.github.io/`, where many more insightful visualizations can be found. Keep in mind that these embeddings are 2-D for the purpose of visualization. In real applications, embeddings typically have higher dimensionality and can simultaneously capture many kinds of similarity between words.

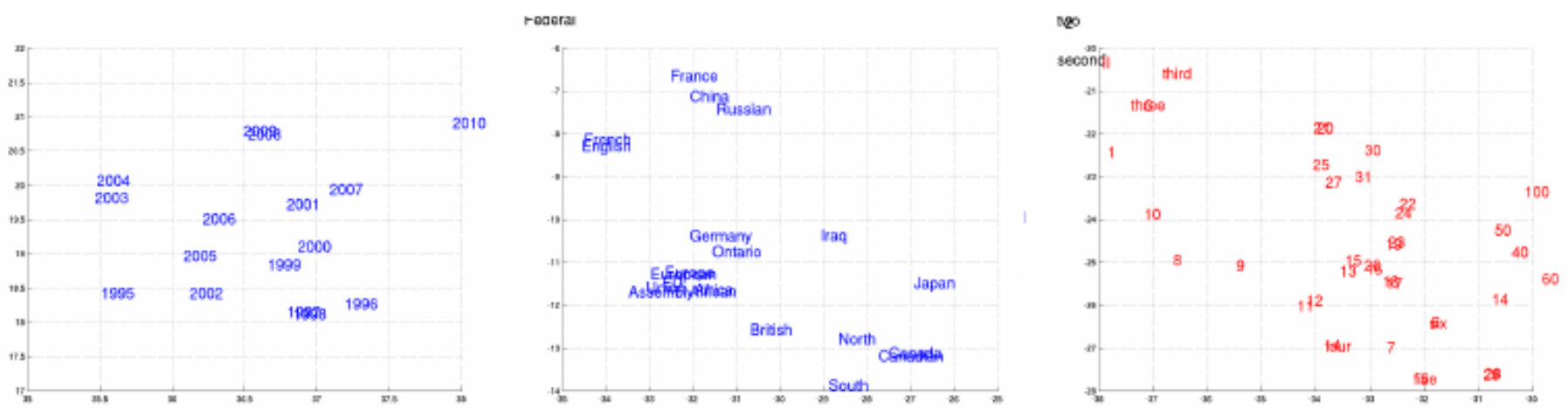


Figure 12.6: Two-dimensional visualizations of word embeddings obtained from a neural machine translation model (Bahdanau *et al.*, 2014), zooming in on specific areas where semantically related words have embedding vectors that are close to each other. Years appear on the left, countries in the middle, and numbers on the right. Compare Fig. 12.5 for a broader perspective. Keep in mind that these embeddings are 2-D for the purpose of visualization. In real applications, embeddings typically have higher dimensionality and can simultaneously capture many kinds of similarity between words.

correct output. The computational costs of the output layer thus arise both at training time (to compute the likelihood and its gradient) and at test time (to compute probabilities for all or selected words).

Suppose that $\boldsymbol{h}$ is the top hidden layer used to predict the output probabilities $\boldsymbol{p}$. If we parameterize the transformation from $\boldsymbol{h}$ to $\boldsymbol{p}$ with learned weights $\boldsymbol{W}$ and learned biases $\boldsymbol{b}$, then the affine-softmax output layer performs the following computations:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\}, \tag{12.4}$$

$$p_i = \frac{e^{a_i}}{\sum_{i' \in \{1,\ldots,|\mathbb{V}|\}} e^{a_{i'}}}. \tag{12.5}$$

If $\boldsymbol{h}$ contains $n_h$ elements then the above operation is $O(|\mathbb{V}| n_h)$. With $n_h$ in the thousands and $|\mathbb{V}|$ in the hundreds of thousands, this computation dominates the computation of most neural language models.

### Use of a Short List

The first neural language models dealt with this problem (Bengio *et al.*, 2001a, 2003a) by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002b) and Schwenk (2007) built upon this approach by splitting the vocabulary $\mathbb{V}$ into a *shortlist* $\mathbb{L}$ of most frequent words (handled by the neural net) and a tail $\mathbb{T} = \mathbb{V} \backslash \mathbb{L}$ of more rare words (handled by an $n$-gram model). To be able to combine the two predictions, the neural net also has to predict the probability that a word appearing after context $C$ belongs to the tail list. This may be achieved by adding an extra sigmoid output unit to provide an estimate of $P(i \in \mathbb{T} \mid C)$. The extra output can then be used to achieve an estimate of the probability distribution over all words in $\mathbb{V}$ as follows:

$$\begin{aligned} P(y = i \mid C) = & 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) \\ & + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C) \end{aligned} \tag{12.6}$$

where $P(y = i \mid C, i \in \mathbb{L})$ is provided by the neural language model and $P(y = i \mid C, i \in \mathbb{T})$ is provided by the $n$-gram model. With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.

An obvious disadvantage of the short list approach is that the potential generalization advantage of the neural language models is limited to the most frequent words. This disadvantage has stimulated the exploration of alternative methods to deal with high-dimensional outputs, described below.

### Hierarchical Softmax

When trying to parametrize and compute a multinoulli probability distribution over a large set (e.g. hundreds of thousands of words) of dimension $|\mathbb{V}|$, a classical approach (Goodman, 2001) is to decompose probabilities hierarchically. Instead of having a number of computations proportional to $|\mathbb{V}|$ (and also proportional to the number of hidden units $n_h$, in our case), the $|\mathbb{V}|$ factor can be reduced to as low as $\log |\mathbb{V}|$. Bengio (2002) and Morin and Bengio (2005) introduced this factorized approach to the context of neural language models.

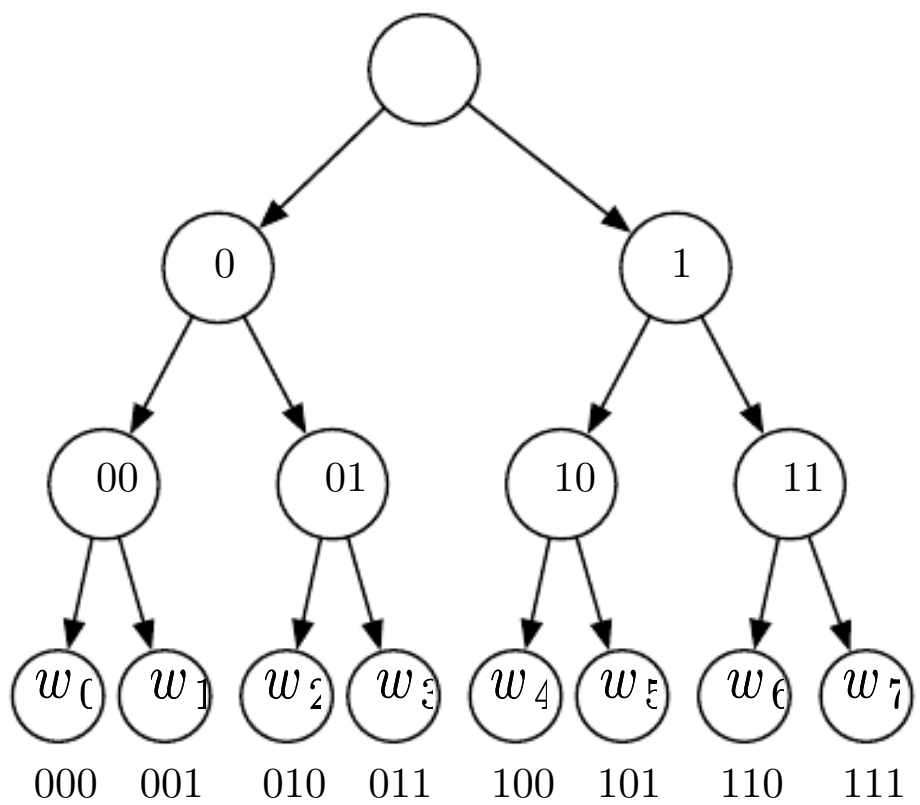


Figure 12.7: Illustration of a hierarchy of word categories, with actual words at the leaves and groups of words at the internal nodes. Any node can be indexed by the sequence of binary decisions (0=left, 1=right) to reach the node from the root. If the tree is sufficiently balanced, the maximum depth (number of binary decisions) is on the order of the logarithm of the number of words $|\mathbb{V}|$: the choice of one out of $|\mathbb{V}|$ words can be obtained by doing $O(\log |\mathbb{V}|)$ operations (one for each of the nodes on the path from the root).

One can think of this hierarchy as building categories of words, then categories of categories of words, then categories of categories of categories of words, etc. Figure 12.7 illustrates a simple example with 8 words $w_0, \ldots, w_7$ organized into a 3-level hierarchy. Super-class 0 contains the classes 00 and 01, which respectively contain the words $(w_0, w_1)$ and $(w_2, w_3)$. Similarly, super-class 1 contains the classes 10 and 11, which respectively contain the words $(w_4, w_5)$ and $(w_6, w_7)$. Hence, computing the probability of a word $y$ can be done by computing three binomial probabilities, associated with the left or right binary decisions associated with the nodes from the root to a node $y$. Let $b_i$ be the $i$-th binary decision when traversing the tree towards the value $y$. Thus, the probability of sampling and output y can be decomposed into a product of conditional probabilities, using

the chain rule for conditional probabilities, with each node indexed by the prefix of these bits. For example, node 10 in Figure 12.7 corresponds to the prefix $(b_0(w_4) = 1, b_1(w_4) = 0)$, and the probability of $w_4$ can be decomposed as follows:

$$\begin{aligned} P(\mathrm{y} = w_4) &= P(\mathrm{b}_0 = 1, \mathrm{b}_1 = 0, \mathrm{b}_2 = 0) \\ &= P(\mathrm{b}_0 = 1) P(\mathrm{b}_1 = 0 \mid \mathrm{b}_0 = 1) P(\mathrm{b}_2 = 0 \mid \mathrm{b}_0 = 1, \mathrm{b}_1 = 0). \end{aligned} \tag{12.7}$$

Each of these conditional probabilities can be computed at one node, starting from the root, and associated with the arc going from a parent node to one of its children nodes.

For neural language models, these probabilities are typically conditioned on some context, so in general we decompose the log-likelihood of the next word $y$ given its context as follows:

$$\log P(y \mid C) = \sum_i \log P(b_i \mid b_1, b_2, \ldots, b_{i-1}, C)$$

where the sum runs over all $k$ bits of $y$. This can be obtained by computing the sigmoid of $k$ dot products, each with a weight vector indexed by the identifier $n = b_1, b_2, \ldots, b_{i-1}$ associated with some internal node $n$ on the path to $y$:

$$\begin{aligned} p_n &= \mathrm{sigmoid}(c_n + \boldsymbol{v}_n \cdot \boldsymbol{h}_C) \\ \log P(b_i \mid b_1, b_2, \ldots, b_{i-1}, C) &= b_i \log p_n + (1 - b_i) \log(1 - p_n) \end{aligned} \tag{12.8}$$

which corresponds to the usual Bernoulli cross-entropy for logistic regression and probabilistic binary classification in neural networks (Sec.s 6.3.2 and 6.3.2).

Since the output log-likelihood can be computed efficiently (as low as $\log |\mathbb{V}|$ rather than $|\mathbb{V}|$), so can its gradient with respect to the output parameters (the $\boldsymbol{v}_n$ and $c_n$ above) as well as with respect to the hidden layer activations $\boldsymbol{h}_C$.

Note that in principle we could optimize the tree structure to minimize the expected number of computations, following Shannon's theorem, i.e., by structuring the tree so that the number of bits associated with a word be approximately equal to the logarithm of the frequency of that word. However, in practice, this is typically not worth it because the computation of the output probabilities is only one part of the total computation. For example, if there are several hidden layers of width $n_h$, then the associated computations grow as $O(n_h^2)$ while the output computations grow as $O(n_h L)$ where $L$ is the average number of bits of output words (weighted by their frequency). Hence, there is not much advantage in making $L$ much less than $n_h$. Consider that $n_h$ is typically large (e.g., around a thousand or more), and the vocabulary sizes are typically not more than the order of a million. Since $\log_2(10^6)$ is about 20, we could get $L$ on the order of 20 for such a large vocabulary, but in fact it would not make much of a difference

to take $L$ on the order of 1000 (the same as $n_h$), which means that a 2-level tree (which has average depth $L$ on the order of $\sqrt{|\mathbb{V}|}$) is sufficient to reap most of the benefit of a hierarchical softmax, with the typical vocabulary sizes and hidden layer sizes that are currently used. A 2-level tree corresponds to simply defining a set of mutually exclusive words classes.

One question that remains somewhat open is how to best define these word classes, or how to define the word hierarchy in general. Early work used existing hierarchies (Morin and Bengio, 2005) but it can also be learned, ideally jointly with the neural language model, although an exact optimization of the log-likelihood appears intractable because the choice of a word hierarchy is a discrete one, not amenable to gradient-based optimization. However, one could use discrete optimization to approximately optimize the partition of words into word classes.

An important advantage of the hierarchical softmax is that it brings computational benefits both at training time and at test time, if at test time we want to compute the probability of specific words. Of course computing the probability of all $|\mathbb{V}|$ words will remain expensive. Another interesting question is how to pick the most likely word, in a given context, and unfortunately the tree structure does not provide an efficient and exact answer. However, in practice (e.g., for translation or speech recognition), we want to pick the best *sequence* of words, and this typically requires a heuristic search such as the beam search (Sec. 19.5.1).

A disadvantage is that in practice the hierarchical softmax tends to give worse test results than sampling-based methods such as described below, although this may be due to a poor choice of word classes.

### Importance Sampling

An idea that is almost as old as the hierarchical softmax, for speeding up training of neural language models, is the use of a sampling technique to approximate the "negative phase" contribution of the gradient, i.e., the "counter-examples" on which the model should give a low score (or high energy), compared to the observed word. See Sec. 18.1 for the decomposition of the log-likelihood into a "positive phase" term (pushing the score of the correct word up, or pushing down its energy, which is easy here because we only have to consider one word) and a "negative phase" term (pushing down the score of all the other words, in proportion to the probability that the model gives them). Using the notation introduced in

Eq. 12.4, the gradient can be written as follows:

$$\begin{aligned}\frac{\partial \log P(y \mid C)}{\partial \theta} &= \frac{\partial \log \mathrm{softmax}_y(\boldsymbol{a})}{\partial \theta} \\ &= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \\ &= \frac{\partial}{\partial \theta}(a_y - \log \sum_i e^{a_i}) \\ &= \frac{\partial a_y}{\partial \theta} - \sum_i P(i \mid C) \frac{\partial a_i}{\partial \theta}\end{aligned} \tag{12.9}$$

where $\boldsymbol{a}$ is the vector of pre-softmax activations (or scores), with one element per word. The first term is the "positive phase" term (pushing $a_y$ up) while the second term is the "negative phase" term (pushing $a_i$ down for all $i$, with weight $P(i \mid C)$. Since the negative phase term is an expectation, we can estimated by a Monte-Carlo sample. However, that would require sampling from the model itself, i.e., computing $P(i \mid C)$ for all $i$ in the vocabulary, which is precisely what we are trying to avoid.

The solution proposed by Bengio and Sénécal (2003); Bengio and Sénécal (2008) is to sample from another distribution, called the proposal distribution (denoted $q$), and use appropriate weights to correct for that. This is called *importance sampling* and is introduced in Sec. 17.1.2. But even exact importance sampling is not appropriate because it requires computing weights $p_i / q_i$, where $p_i = P(i \mid C)$, which can only be computed if all the scores $a_i$ are computed. The solution adopted is called *biased importance sampling*, where the importance weights are normalized to sum to 1, i.e., when negative word $n_i$ is sampled, the associated gradient is weighted by

$$w_i = \frac{p_{n_i} / q_{n_i}}{\sum_{j=1}^{N} p_{n_j} / q_{n_j}}.$$

These weights are used to give the appropriate importance to the $N$ negative samples from $q$ used to form the estimated negative phase contribution to the gradient:

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C) \frac{\partial a_i}{\partial \theta}) \approx \frac{1}{N} \sum_{i=1}^{N} w_i \frac{\partial a_{n_i}}{\partial \theta}.$$

A unigram or a bigram distribution works well as the proposal distribution $q$, because can be easily estimated from the data as well as sampled from very efficiently.

A related application of importance sampling to speed-up training of a larger class of model was introduced by Dauphin *et al.* (2011). These are models where

the output is not necessarily a 1-of-n choice (which one can think of as an integer, or as a one-hot vector), but more generally a sparse vector, where only a few of the entries are non-zero. This occurs for example when the output is a *bag-of-words*, i.e., a sparse vector where the non-zeros indicate the presence (0 or 1) or the frequency (a small count) of each word of a document. In the paper, the authors study the case of denoising auto-encoders with a bag-of-words as input. Whereas the sparsity of the input can be easily exploited (by ignoring the zeros in the computation), it is not so clear for the output (reconstruction) units. Because an auto-encoder also predicts its input, the target for the reconstruction is sparse but the prediction (probabilities that any particular word is present) is not. The algorithm ends up minimizing reconstruction error (minus log-likelihood) for the "positive words" (those that are non-zero in the target) and an equal number of "negative words" chosen randomly, but with their gradients reweighted appropriately according to importance sampling.

In all of these cases, the computational complexity of gradient estimation for the output layer is reduced to be proportional to the number of negative samples rather than proportional to the size of the output vector.

### Noise-Contrastive Estimation and Ranking Loss

Other approaches based on sampling have been proposed to reduce the computational cost of training neural language models with large vocabularies.

An early one is the ranking loss proposed by Collobert and Weston (2008), in which we view the output of the neural language model for each word as a score and ask that the score of the correct word $a_y$ be ranked high in comparison to the other scores $a_i$. The ranking loss proposed then is

$$L = \sum_i \max(0, 1 - a_y + a_i). \tag{12.10}$$

Note that the gradient is zero for the i-th term if the score of the observed word, $a_y$ is greater than the score of negative word $a_i$ by a margin of 1. One issue with this criterion is that it does not provide estimated conditional probabilities, which are useful in some applications, e.g., speech recognition, or (conditional) text generation.

A more recently used training objective for neural language model is noise-contrastive estimation, which is introduced in Sec. 18.6. The idea is to turn the training task into a probabilistic classification problem where the learner tries to identify whether a given value is sampled from the data generating distribution (under the probability estimated by the trained model) or from a fixed "noise" model. This approach has been successfully applied to neural language models (Mnih and Teh, 2012; Mnih and Kavukcuoglu, 2013). That probabilistic

classifier output (when the output is the observed word $y$) can be obtained by combining the score of the observed word $a_y$ with a learned parameter that estimates the log of the normalizing constant of the softmax. The training objective also requires that one samples a word from the "noise" distribution, which acts like a proposal distribution for importance sampling.

### 12.4.5 Combining Neural Language Models with $n$-grams

A major advantage of $n$-grams over neural networks is that $n$-grams achieve high model capacity (by storing the frequencies of very many tuples) while requiring very little computation to process an example (by looking up only a few tuples that match the current context). If we use hash tables or trees to access the counts, the computation used for $n$-grams is almost independent of capacity. In comparison, doubling a neural network's number of parameters typically also roughly doubles its computation time (exceptions include models that avoid using all parameters on each pass, like embedding layers that index a single embedding, and models that can add parameters while reducing the degree of parameter sharing, like tiled convolutional networks).

One easy way to add capacity is thus to combine both approaches in an ensemble consisting of a neural language model and an $n$-gram language model (Bengio *et al.*, 2001b, 2003a). As with any ensemble, this technique can also reduce test error, and many ways of combining the ensemble members' predictions are possible (uniform weighting, weights chosen on a validation set, etc.) Later, this ensemble idea was extended Mikolov *et al.* (2011a) the idea of an ensemble approach to include not just two models but a large array of models. It is also possible to pair a neural network with a maximum entropy model and train both jointly (Mikolov *et al.*, 2011b). This approach can be viewed as training a neural network with an extra set of inputs that are connected directly to the output, and not connected to any other part of the model. The extra inputs are indicators for the presence of particular $n$-grams in the input context, so these variables are very high-dimensional and very sparse. The increase in model capacity is huge—the new portion of the architecture contains up to $|sV|^n$ parameters—but the amount of added computation needed to process an input is minimal because the extra inputs are very sparse.

### 12.4.6 Neural Machine Translation

The early use of neural networks for machine translation (Schwenk *et al.*, 2006; Schwenk, 2010) took advantage of the good performance of neural language models in order to replace one component of a machine translation system, the statistical language model, which was traditionally done by an $n$-gram-based model. These

$n$-gram based model include not just traditional back-off $n$-gram models (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999) but also so-called maximum entropy language models (Berger *et al.*, 1996), in which an affine / softmax layer predicts the next word given the presence of frequent $n$-grams in the context (as outlined in the previous section).

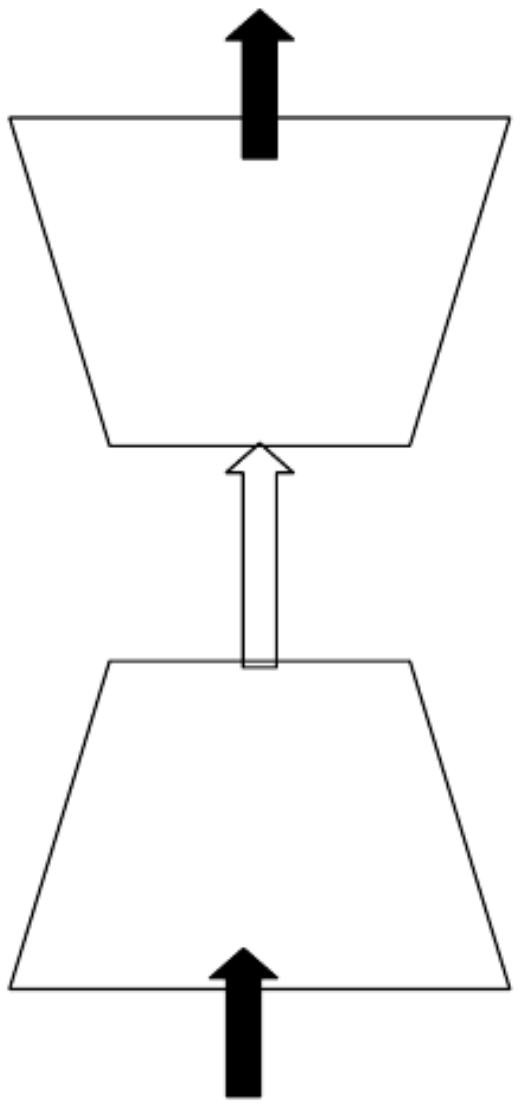

Figure 12.8: The encoder-decoder architecture to map back and forth between a surface representation (e.g., sequence of words, image) and a semantic representation. By coupling the encoder for one modality (e.g. French to "meaning") with the decoder for another modality (e.g. "meaning" to English), we can train systems that translate from one modality to another. This idea has been applied successfully not just to machine translation but also to caption generation from images.

However, once we have an architecture for learning neural language model that captures the joint probability $P(\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_T)$ of a sequence of words, we can in principle make the distribution conditional on some generic context $C$ by making some of its parameters a function of $C$, as explained in Sec. 6.3.2. For example, Devlin *et al.* (2014) beat the state-of-the-art in some statistical machine translation benchmarks by using an MLP to score a phrase $\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k$ in the target language given a phrase $\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n$ in the source language, i.e., estimate $P(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k \mid \mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n)$ and use it to replace the traditional phrase table (that estimates

the same quantity) based on $n$-grams. To make this translation more flexible, we would like to use a model that can accommodate variable length inputs and variable length outputs. If an RNN is used to capture $P(\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_T)$, we can make the initial state of the RNN or the biases used at each time step for the hidden units be a function of some generic context $C$. If $C$ is obtained from a source sentence in another language, we can thus train our neural language to translate from one language to another. If we think of $C$ as a semantic summary of the source sentence, it can for example be obtained as the final state of another RNN (the encoder RNN or "reader"), as in Cho *et al.* (2014a); Sutskever *et al.* (2014b); Jean *et al.* (2014), or as the top layer of a convolutional network, as in (Kalchbrenner and Blunsom, 2013). This general idea of an encoder-decoder framework for machine translation is illustrated in Figure 12.8.

This raises the question of representing not just words but sequences of words. The idea of learning a semantic representation of phrases and even sentences so that the representation of the source and target sentences are close to each other and can be mapped from one to the other has been explored (Kalchbrenner and Blunsom, 2013; Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014), first using a combination of convolutions and RNNs (Kalchbrenner and Blunsom, 2013) and then using both RNNs for encoding the source sentence and for generating the output target-language sentence (Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014).

### Using an Attention Mechanism and Aligning Pieces of Data

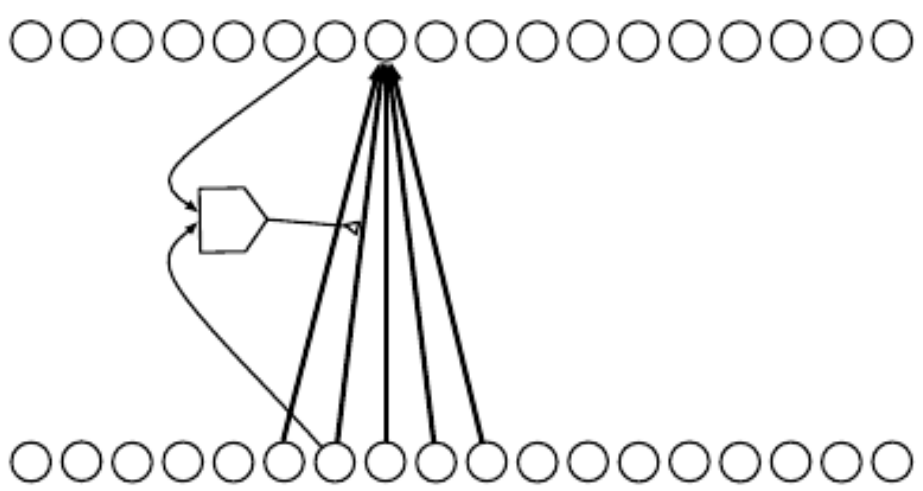

Figure 12.9: Illustration of the attention mechanism used in a neural machine translation system introduced in Bahdanau *et al.* (2014).

Using a fixed-size representation to capture all the semantic details of a very long sentence of say 60 words is very difficult. It can be achieved by training a sufficiently large RNN well enough and for long enough, as demonstrated by Cho *et al.* (2014a); Sutskever *et al.* (2014b). However, this is not how humans translate long sequences of words. What they usually do, after having read the whole

sentence or paragraph (to get the context and the jist of what is being expressed), they produce the translated words one at a time, each time focusing on a different part of the input sentence in order to gather the semantic details that are required to produce the next output word. That is exactly the idea that Bahdanau *et al.* (2014) first introduced and that is illustrated in Figure 12.9.

We can think of an attention-based system as having three components:

1. A process that "*reads*" raw data (such as source words in source sentence), and converts them into distributed representations, with one feature vector associated with each word position.

2. A list of feature vectors storing the output of the reader. This can be understood as a "*memory*" containing a sequence of facts, which can be retrieved later, not necessarily in the same order, nor having to visit all of them.

3. A process that "*exploits*" the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).

The third component generates the translated sentence.

When words in a sentence written in one language are aligned with corresponding words in a translated sentence in another language, it becomes possible to relate the corresponding word embeddings. Earlier work showed that one could learn a kind of translation matrix relating the word embeddings in one language with the word embeddings in another (Kočiský *et al.*, 2014), yielding lower alignmnent error rates than traditional approaches based on the frequency counts in the phrase table. There is even earlier work on learning cross-lingual word vectors (Klementiev *et al.*, 2012). Many extensions to this approach are possible. For example, more efficient cross-lingual alignment (Gouws *et al.*, 2014) allows training on larger datasets.

## 12.5 Structured Outputs

In principle, if we have good models $P(\mathbf{Y} \mid \boldsymbol{\omega})$ of the joint distribution of random variables $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_k)$, with parameters $\boldsymbol{\omega}$ we can use them to build conditional models $P(\mathbf{Y} \mid \mathbf{X})$ conditioned on some input variables $\mathbf{X}$ by making $\boldsymbol{\omega}$ a parametrized function of the input $\mathbf{X}$. In Chapter 10, in particular with Sec. 10.4, we saw how an RNN can represent a joint distribution over elements of a sequence that can be conditioned. For example, with machine translation (in the previous section, 12.4.6), we condition on another sequence (the source sentence). Conditional joint distribution models are sometimes called "structured output models",

to distinguish them from the more usual supervised learning tasks where the outputs represent a single random variable (like a class) or conditionally independent random variables (like different attributes, discussed in Sec. 6.3.2).

In the third part of this book, we will go beyond RNNs as means of capturing the joint distribution between output variables, conditioned on input variables. For example Restricted Boltzmann Machines (RBMs) were made conditional by Taylor *et al.* (2007); Taylor and Hinton (2009) in the context of modeling motion style and by Boulanger-Lewandowski *et al.* (2012) in the context of symbolic sequences describing polyphonic music. In both of these examples, we actually use an RBM as a "output model" for an RNN, i.e., at each time step in the sequence, we want to output a distribution over a group of random variables (articulators for Taylor and Hinton (2009), and musical notes for Boulanger-Lewandowski *et al.* (2012)), given the current state of the RNN. With the RNN-RBM (Boulanger-Lewandowski *et al.*, 2012), a generative model is set up to model a sequence of frames $\boldsymbol{x}_t$ with the following structure. An RNN captures the past context through a state variable $\boldsymbol{h}_t$ that follows a deterministic recurrence

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t).$$

In addition, at each time step, a joint distribution over the elements of the next frame $\boldsymbol{x}_{t+1}$ is formed using an RBM with parameters $\boldsymbol{\omega}_t$:

$$P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t).$$

The RBM parameters $\boldsymbol{\omega}_t = (\boldsymbol{\omega}_{\text{RBM}}, \boldsymbol{\omega}'_t)$ are composed of two subsets of parameters, the parameters $\boldsymbol{\omega}_{\text{RBM}}$ that are ordinary parameters (namely the weights of the RBM and the visible units biases) and the parameters $\boldsymbol{\omega}'_t$ that are conditioned on the RNN state $\boldsymbol{h}_t$ (namely, the hidden units biases):

$$\boldsymbol{\omega}'_t = g(\boldsymbol{h}_t).$$

where both $f$ and $g$ have free parameters that are updated by SGD, along with $\boldsymbol{\omega}_{\text{RBM}}$. Since $\boldsymbol{\omega}_t$ is a function of the past frames, we are modeling the joint distribution of the sequence of frames:

$$\begin{aligned} P(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) &= \prod_t P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1) \\ &= \prod_t P(\boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t) = \prod_t P(\mathbf{x}_{t+1} \mid \boldsymbol{h}_t). \end{aligned} \tag{12.11}$$

Our goal is to construct a vector $\boldsymbol{\delta\omega}$ that approximates the log-likelihood gradient on the RBM parameters. Specifically, we want the condition

$$\boldsymbol{\delta\omega} \approx \nabla_{\boldsymbol{\omega}} \log P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega})$$

to hold. The RBM log-likelihood gradient is intractable, but we can achieve the desired approximation by setting $\boldsymbol{\delta\omega}$ to the output of an algorithm called contrastive divergence. The contrastive divergence algorithm estimates the true log-likelihood gradient, but the estimate is only an approximation. In the same way that we have decomposed the RBM parameters $\omega_t$ into the two groups $\boldsymbol{\omega}_{\text{RBM}}$ and $\boldsymbol{\omega}'_t$, we can decompose $\boldsymbol{\delta\omega}$ into the corresponding estimated gradients $\boldsymbol{\delta\omega}_{\text{RBM}}$ and $\boldsymbol{\delta\omega}'_t$. The estimated gradient $\boldsymbol{\delta\omega}_{\text{RBM}}$ can be used to update $\boldsymbol{\omega}_{\text{RBM}}$ directly (e.g., by SGD) while $\boldsymbol{\delta\omega}'_t$ can be back-propagated through the RNN (as if it was the true gradient of $\log P(\mathbf{x}_{t+1} = \boldsymbol{x}_{t+1} \mid \boldsymbol{\omega}_t)$ with respect to $\boldsymbol{\omega}'_t$), thereby providing the estimated gradient on the parameters of $f$ and $g$. The use of the contrastive divergence update to train conditional RBMs was introduced by Taylor *et al.* (2007).

## 12.6 Other Applications

In this section we cover a few other types of applications of deep learning that are different from the standard object recognition, speech recognition and natural language processing tasks discussed above. The third part of this book will expand that scope even further to include tasks requiring the ability to generate samples or conditional high-dimensional samples (unlike "the next word", in language models).

### 12.6.1 Dimensionality Reduction and Information Retrieval

Dimensionality reduction was one of the first applications of representation learning and deep learning, and was one of the early motivations for studying auto-encoders. For example, Hinton and Salakhutdinov (2006) trained a stack of RBMs and then used their weights to initialize a deep auto-encoder with gradually smaller hidden layers, culminating in a bottleneck of 30 units. The layer widths were 784, 100, 500, 250 and 30 units per layer in the encoder, and the same in reverse for the decoder. The resulting code yielded less reconstruction error than PCA into 30 dimensions and the learned representation was qualitatively easier to interpret and relate to the underlying categories, with these categories manifesting as well-separated clusters.

Lower-dimensional representations can improve performance on many tasks, such as classification. Models of smaller spaces consume less computational resources. Many forms of imensionality reduction operation place semantically related examples near each other, as observed by Salakhutdinov and Hinton (2007) and Torralba *et al.* (2008). The hints provided by the mapping to the lower-dimensional space aid generalization.

One task that benefits even more than usual from dimensionality reduction is *information retrieval*, the task of finding entries in a database that resemble a

query entry. This task derives the usual benefits from dimensionality reduction that other tasks do, but also derives the additional benefit that search can become extremely efficient in certain kinds of low dimensional spaces. Specifically, if we train the dimensionality reduction algorithm to produce a code that is low-dimensional and *binary*, then we can store all database entries in a hash table mapping binary code vectors to entries, and perform information retrieval by returning all database entries that have the same binary code as the query. We can also search over slightly less similar entries very efficiently, just by flipping individual bits from the encoding of the query. This approach to information retrieval via dimensionality reduction and binarization is called *semantic hashing* (Salakhutdinov and Hinton, 2007), and has been applied to both textual input (Salakhutdinov and Hinton, 2007), and images (Torralba *et al.*, 2008; Weiss *et al.*, 2008).

To produce binary codes for semantic hashing, one typically uses an encoding function with sigmoids on the final layer. The sigmoid units must be trained to be saturated to nearly 0 or nearly 1 for all input values. One trick that can accomplish this is simply to inject additive noise just before the sigmoid non-linearity during training. The magnitude of the noise should increase over time. To fight that noise and preserve as much information as possible, the network must increase the magnitude of the inputs to the sigmoid function, until saturation occurs.

The idea of learning a hashing function has been further explored in several directions, including the idea of training the representations so as to optimize a loss more directly linked to the task of finding nearby examples in the hash table (Norouzi and Fleet, 2011).

There are other applications of deep learning in information retrieval. In particular, using neural language models and trained word embeddings can make natural language queries more robust because two semantically similar queries would tend to be associated to nearby representations, if these are based on their word embeddings. For example Sordoni *et al.* (2015) suggest a better query based on the recent queries and results previously obtained, using a novel hierarchical recurrent neural network architecture in which each a low-level recurrent net processes the sequence of words in one query and outputs a representation that is the input for a query-level recurrent network, with one "time step" per query in the sequence of queries.

### 12.6.2 Recommender Systems

One of the major families of applications of machine learning in the information technology sector is the ability to make recommendations of items to potential users or customers. Two major types of applications can be distinguished: online advertising and item recommendations (often these recommendations are still for the purpose of selling a product). Both rely on predicting the association between

a user and an item, either to predict the probability of some action (the user buying the product, or some proxy for this action) or the expected gain (which may depend on the value of the product) if an ad is shown or a recommendation is made regarding that product, to that user. The internet is currently financed in great part by various forms of online advertising, and there are major parts of the economy that rely on buying online. Companies such as Amazon and eBay are known to use machine learning and specifically deep learning[2] for their product recommendations. Sometimes, the items are not products that are actually for sale. Examples include the recommendation of posts on social networks, recommending movies to watch, recommending jokes, recommending advice from experts, matching players for video games, or matching people in dating services.

Often, this association problem is handled like a supervised learning problem: given some information about the item and about the user, predict the proxy of interest (user clicks on ad, user enters a rating, user clicks on a "like" button, user buys product, user spends some amount of money on the product, user spends time visiting a page for the product, etc). This often ends up being either a regression problem (predicting some conditional expected value) or a probabilistic classification problem (predicting the conditional probability of some discrete event).

The early work on recommender systems relied on the minimal information available as inputs for these predictions: the user ID and the item ID. In this context, the only way to generalize is to rely on the similarity between the patterns of values of the target variable for different users or for different items. If two users both like A, B and C, and if user 1 likes item D, then this should be a strong cue that user 2 will also like D. Algorithms based on this principle come under the name of *collaborative filtering*. Both non-parametric approaches (such as nearest-neighbor methods based on the estimated similarity between patterns of preferences) and parametric methods are possible, the latter often relying on learning a distributed representation (also called embedding) for each user and for each item. The simplest one of these, and a highly successful method that one often finds as a component of state-of-the-art systems, corresponds to a bilinear prediction of the target variable (such as a rating). The prediction is obtained by the dot product between the user embedding and the item embedding (possibly corrected by constants that depend only on either the user ID $u$ or the item ID $u$). Let $\hat{\boldsymbol{R}}$ be the matrix containing our predictions, $\boldsymbol{A}$ a matrix with user embeddings in its rows and $\boldsymbol{B}$ a matrix with item embeddings in its columns. Let $\boldsymbol{b}$ and $\boldsymbol{c}$ be vectors that contain respectively a kind of bias for each user (representing how grumpy or positive that user is in general) and for each item (representing its

[2] http://tamebay.com/2015/08/ebay-reveals-how-its-using-deep-learning-in-search.html, http://www.wired.com/2015/04/now-anyone-can-tap-ai-behind-amazons-recommendations/

general popularity). The bilinear prediction is thus obtained as follows:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \tag{12.12}$$

Typically one wants to minimize the squared error between predicted ratings $\hat{R}_{u,i}$ and actual ratings $\boldsymbol{R}_{u,i}$. User embeddings and item embeddings can then be conveniently visualized when they are first reduced to a low dimension (two or three), or they can be used to compare users or items against each other, just like word embeddings. One way to obtain these embeddings is by performing a singular value decomposition of the matrix $\boldsymbol{R}$ of actual targets (such as ratings). This corresponds to factorizing $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}'$ (or a normalized variant) into the product of two factors, the lower rank matrices $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}$ and $\boldsymbol{B} = \boldsymbol{V}'$. One problem with the SVD is that it treats the missing entries in an arbitrary way, as if they corresponded to a target value of 0. Instead we would like to avoid paying any cost for the predictions made on missing entries. The good news is that the sum of squared errors on the observed ratings can also be easily minimized by gradient-based optimization. It turned out that the SVD or bilinear prediction of Eq. 12.12 performed very well in the competition for the Netflix prize[3], aiming at predicting ratings for films, based only on previous ratings by a large set of anonymous users. Many machine learning experts participated in this competition, which took place between 2006 and 2009. It raised the level of research in recommender systems using advanced machine learning and yielded improvements in recommender systems. Even though it did not win by itself, the simple bilinear prediction or SVD was a component of the ensemble models presented by most of the competitors, including the winners (Töscher *et al.*, 2009; Koren, 2009).

Beyond these bilinear models based on distributed representations, one of the first uses of neural networks for collaborative filtering is based on the RBM (Salakhutdinov *et al.*, 2007). RBMs turned out to be an important element of the ensemble of methods that won the Netflix competition (Töscher *et al.*, 2009; Koren, 2009). More advanced variants on the idea of factorizing the ratings matrix have also been explored in the neural networks community (Salakhutdinov and Mnih, 2008).

However, there is a basic limitation of collaborative filtering systems: when a new item or a new user is considered, there is no way to evaluate its similarity with other items or users (respectively), or the degree of association between, say, that new user and existing items. This is called the problem of cold-start recommendations, and a general way of solving it is to introduce extra information about the individual users and items, such as user profile information or item features, and systems that use such information are called *content-based recommender systems*. The mapping from a rich set of user features or item features to their embedding

[3] `http://www.netflixprize.com/`

can be learned through a deep learning architecture (Huang *et al.*, 2013; Elkahky *et al.*, 2015)

Specialized deep learning architectures such as convolutional networks have also been applied to learn to extract features from rich content such as from musical audio tracks, for music recommendation (van den Oörd *et al.*, 2013). In that work, the convolutional net takes acoustic features as input and computes an embedding for the associated song. The dot product between this song embedding and the embedding for a user is then used to predict whether a user will listen to a particular song.

### Exploration Versus Exploitation

When making recommendations to users, an issue arises that goes beyond ordinary supervised learning, and into the realm of reinforcement learning, into what is called *contextual bandits* (Langford and Zhang, 2008; Lu *et al.*, 2010). The issue is that when we use the recommendation system to collect data, we get a biased and incomplete view of the preferences of users: we only see the responses of users to the items they were recommended and not to the other items. In addition, in some cases we may not get any information on users for whom no recommendation has been made (for example, with ad auctions, it may be that the price proposed for an ad was below a minimum price threshold, or does not win the auction, so the ad is not shown at all). More importantly, we get no information about what outcome would have resulted from recommending any of the other items. This would be like training a classifier by picking one class for each input case (typically the class with the highest probability) and then only getting as feedback whether this was the correct class or not. Clearly, each example conveys less information so more examples are necessary. Worse, if we are not careful, we could end up with a system that continues picking the wrong decisions even as more and more data is collected, because the correct decision initially had a very low probability: until the learner picks that correct decision, it doesn't learn about the correct decision. This is similar to the situation in reinforcement learning where only the reward for the selected actions are observed. The difference between the bandits situation and the more general reinforcement learning situation is that with bandits there is only action and one reward per learning episode, whereas in general there might be a sequence of actions and rewards, with no clear assignment of credit or blame to the different actions. The term *contextual* bandits refers to the case where the action is taken in the context of some input variable that can inform the decision (for example, we at least know the user identity, and we want to pick an item). The mapping from context to action is also called a *policy*. The feedback loop between the learner and the data distribution (which now depends on the actions of the learner) is a central research issue in the reinforcement learning and bandits

literature.

Reinforcement learning requires choosing a tradeoff between *exploration* and *exploitation*. Exploitation refers to taking actions that come from the current, best version of the learned policy—actions that we know will achieve a high reward. Exploration refers to taking actions specifically in order to obtain more training data. If we know that given context $\boldsymbol{x}$, action $a$ gives us a reward of 1, we do not know whether that is the best possible reward. We may want to exploit our current policy and continue taking action $a$ in order to be relatively sure of obtaining a reward of 1. However, we may also want to explore by trying action $a'$. We do not know what will happen if we try action $a'$. We hope to get a reward of 2, but we run the risk of getting a reward of 0. Either way, we at least gain some knowledge.

Exploration can be implemented in many ways, ranging from occasionally taking random actions intended to cover the entire space of possible actions, to model-based approaches that compute a choice of action based on its expected reward and the model's amount of uncertainty about that reward.

Many factors determine the extent to which we prefer exploration or exploitation. One of the most prominent factors is the time scale we are interested in. If the agent has only a short amount of time to accrue reward, then we prefer more exploitation. If the agent has a long time to accrue reward, then we begin with more exploration so that future actions can be planned more effectively with more knowledge. As time progresses and our learned policy improves, we move toward more exploitation.

Supervised learning has no tradeoff between exploration and exploitation because the supervision signal always specifies which output is correct for each input. There is no need to try out different outputs to determine if one is better than the model's current output—we always know that the label is the best output.

Besides the exploration-exploitation trade-off, the feedback loop between learning and the environment via actions and observed examples also makes it less trivial to evaluate and compare different policies on data that was generated using another policy, but solutions exist (Dudik *et al.*, 2011).

### 12.6.3 Knowledge Representation, Reasoning and Question Answering

Deep learning approaches have been very successful in language modeling, machine translation and natural language processing thanks to the concept of word vectors or word embeddings (Bengio *et al.*, 2001b), derived from the earlier idea of distributed representations for symbols (Rumelhart *et al.*, 1986a). This is still a kind of shallow representation that captures semantic knowledge at the level of individual words and concepts. How about representing phrases and relations

between words? How about representing facts and knowledge, so as to be able to answer questions? Machine learning is already used for this purpose in search engines but much more remains to be done to explore how to answer these questions.

### Knowledge, Relations and Question Answering

One such interesting question is how distributed representations can be trained to capture the *relations* between two entities. These relations allow use to formalize facts about objects and how objects interact with each other.

In mathematics, a *binary relation* is a set of ordered pairs of objects. Pairs that are in the set are said to have the relation while those who are not in the set do not. For example, we can define the relation "is less than" on the set of entities $\{1, 2, 3\}$ by defining the set of ordered pairs $\mathbb{S} = \{(1, 2), (1, 3), (2, 3)\}$. Once this relation is defined, we can use it like a verb. Because $(1, 2) \in \mathbb{S}$, we say that 1 is less than 2. Because $(2, 1) \notin \mathbb{S}$, we can not say that 2 is less than 1. Of course, the entities that are related to one another need not be numbers. We could define a relation `is_a_type_of` containing tuples like (`dog`, `mammal`).

In the context of AI, we think of a relation as a sentence in a syntactically simple and highly structured language. The relation plays the role of a verb, while two arguments to the relation play the role of its subject and object. These sentences take the form of a triplet of tokens

$$(\text{subject}, \text{verb}, \text{object}$$

with values

$$(\text{entity}_i, \text{relation}_j, \text{entity}_k).$$

We can also define an *attribute*, a concept analogous to a relation, but taking only one argument:

$$(\text{entity}_i, \text{attribute}_j).$$

For example, we could define the `has_fur` attribute, and apply it to entities like `dog`.

Many applications require representing relations and reasoning about them. How should we best do this within the context of neural networks?

Machine learning models of course require training data. We can infer relations between entities from training datasets consisting of unstructured natural language. There are also structured databases that identify relations explicitly. A common structure for these databases is the *relational database*, which stores this same kind of information, albeit not formatted as three token sentences. When a database is intended to convey commonsense knowledge about everyday life or expert knowledge about an application area to an artificial intelligence system,

we call the database a *knowledge base*. Knowledge bases range from general ones like `Freebase`, `OpenCyc`, `WordNet`, or `Wikibase`[4], etc. to more specialized knowledge bases, like `GeneOntology`[5]. Representations for entities and relations can be learned by considering each triplet in a knowledge base as a training example and maximizing a training objective that captures their joint distribution (Bordes *et al.*, 2013a).

In addition to training data, we also need to define a model family to train. A common approach is to extend neural language models to model entities and relations. Neural language models learn a vector that provides a distributed representation of each word. They also learn about interactions between words, such as which word is likely to come after a sequence of words, by learning functions of these vectors. We can extend this approach to entities and relations by learning an embedding vector for each relation. In fact, the parallel between modeling language and modeling knowledge encoded as relations is so close that researchers have trained representations of such entities by using *both* knowledge bases *and* natural language sentences (Bordes *et al.*, 2011, 2012; Wang *et al.*, 2014a) or combining data from multiple relational databases (Bordes *et al.*, 2013b). Many possibilities exist for the particular parametrization associated with such a model. Early work on learning about relations between entities (Paccanaro and Hinton, 2000) posited highly constrained parametric forms ("linear relational embeddings"), often using a different form of representation for the relation than for the entities. For example, Paccanaro and Hinton (2000); Bordes *et al.* (2011) used vectors for entities and matrices for relations, with the idea that a relation acts like an operator on entities. Alternatively, relations can be considered as any other entity (Bordes *et al.*, 2012), allowing to make statements about relations, but more flexibility is put in the machinery that combines them in order to model their joint distribution.

A practical short-term application of such models is *link prediction*: predicting missing arcs in the knowledge graph. This is a form of generalization to new facts, based on old facts. Most of the knowledge bases that currently exist have been constructed through manual labour, which tends to leave many and probably the majority of true relations absent from the knowledge base. See Wang *et al.* (2014b); Lin *et al.* (2015); Garcia-Duran *et al.* (2015) for examples of such an application. In general, only positive examples of facts are known, so the metrics used (and also the objective function) are those used in information retrieval, based on ranking and precision. For example, precision @10% counts how many time the "correct" fact appears among the 10% highest scoring ones, when we

[4]Respectively available from these web sites: `freebase.com`, `cyc.com/opencyc`, `wordnet.princeton.edu`, `wikiba.se`

[5]`geneontology.org`

consider all the corrupted variants of the fact (e.g., by replacing one of entities by any one in the set of entities). Another application of knowledge bases and distributed representations for them is *word-sense disambiguation* (Navigli and Velardi, 2005; Bordes *et al.*, 2012), which is the task of deciding which of the senses of a word is the appropriate one, in some context.

Eventually, knowledge of relations combined with a reasoning process and understanding of natural language could allow us to build a general question answering system. A general question answering system must be able to process input information and remember important facts, organized in a way that it can retrieve and reason about them later. This remains a difficult open problem which can only be solved in restricted "toy" environments. Currently, the best approach to remembering and retrieving specific declarative facts is to use an explicit memory mechanism, as described in Sec. 10.7.5. Memory networks were first proposed to solve a toy question answering task (Weston *et al.*, 2014). Kumar *et al.* (2015) have proposed an extension that uses GRU recurrent nets to read the input into the memory and to produce the answer given the contents of the memory.

# Part III

# Deep Learning Research

This part of the book describes the more ambitious and advanced approaches to deep learning, currently pursued by the research community.

In the previous parts of the book, we have shown how to solve supervised learning problems—how to learn to map one vector to another, given enough examples of the mapping.

Not all problems we might want to solve fall into this category. We may wish to generate new examples, or determine how likely some point is, or handle missing values and take advantage of a large set of unlabeled examples or examples from related tasks. Many deep learning algorithms have been designed to tackle such unsupervised learning problems, but none have truly solved the problem in the same way that deep learning has largely solved the supervised learning problem for a wide variety of tasks. In this part of the book, we describe the existing approaches to unsupervised learning and some of the popular thought about how we can make progress in this field.

Another shortcoming of the current state of the art for industrial applications is that our learning algorithms require large amounts of supervised data to achieve good accuracy. In this part of the book, we discuss some of the speculative approaches to reducing the amount of labeled data necessary for existing models to work well.

This section is the most important for a researcher—someone who wants to understand the breadth of perspectives that have been brought to the field of deep learning, and push the field forward towards true artificial intelligence.

# Chapter 13

# Structured Probabilistic Models for Deep Learning

Deep learning draws upon many modeling formalisms that researchers can use to guide their design efforts and describe their algorithms. One of these formalisms is the idea of *structured probabilistic models*. We have already discussed structured probabilistic models briefly in Chapter 3.14. That brief presentation was sufficient to understand how to use structured probabilistic models as a language to describe some of the algorithms in part II of this book. Now, in part III, structured probabilistic models are a key ingredient of many of the most important research topics in deep learning. In order to prepare to discuss these research ideas, this chapter describes structured probabilistic models in much greater detail. This chapter is intended to be self-contained; the reader does not need to review the earlier introduction before continuing with this chapter.

A structured probabilistic model is a way of describing a probability distribution, using a graph to describe which random variables in the probability distribution interact with each other directly. Here we use "graph" in the graph theory sense–a set of vertices connected to one another by a set of edges. Because the structure of the model is defined by a graph, these models are often also referred to as *graphical models*.

The graphical models research community is large and has developed many different models, training algorithms, and inference algorithms. In this chapter, we provide basic background on some of the most central ideas of graphical models, with an emphasis on the concepts that have proven most useful to the deep learning research community. If you already have a strong background in graphical models, you may wish to skip most of this chapter. However, even a graphical model expert may benefit from reading the final section of this chapter, section 13.6, in which we highlight some of the unique ways that graphical

models are used for deep learning algorithms. Deep learning practitioners tend to use very different model structures, learning algorithms, and inference procedures than are commonly used by the rest of the graphical models research community. In this chapter, we identify these differences in preferences and explain the reasons for them.

In this chapter we first describe the challenges of building large-scale probabilistic models in section 13.1. Next, we describe how to use a graph to describe the structure of a probability distribution in section 13.2. We then revisit the challenges we described in section 13.1 and show how the structured approach to probabilistic modeling can overcome these challenges in section 13.3. One of the major difficulties in graphical modeling is understanding which variables need to be able to interact directly, i.e., which graph structures are most suitable for a given problem. We outline two approaches to resolving this difficulty by learning about the dependencies in section 13.4. Finally, we close with a discussion of the unique emphasis that deep learning practitioners place on specific approaches to graphical modeling in section 13.6.

## 13.1 The Challenge of Unstructured Modeling

The goal of deep learning is to scale machine learning to the kinds of challenges needed to solve artificial intelligence. This means being able to understand high-dimensional data with rich structure. For example, we would like AI algorithms to be able to understand natural images[1], audio waveforms representing speech, and documents containing multiple words and punctuation characters.

Classification algortihms can take such a rich high-dimensional input and summarize it with a categorical label—what object is in a photo, what word is spoken in a recording, what topic a document is about. The process of classification discards most of the information in the input and produces on a single output (or a probability distribution over values of that single output). The classifier is also often able to ignore many parts of the input. For example, when recognizing an object in a photo, it is usually possible to ignore the background of the photo.

It is possible to ask probabilistic models to do many other tasks. These tasks are often more expensive than classification. Some of them require producing multiple output values. Most require a complete understanding of the entire structure of the input, with no option to ignore sections of it. These tasks include

- Density estimation: given an input $\boldsymbol{x}$, the machine learning system returns an estimate of $p(\boldsymbol{x})$. This requires only a single output, but it does require

[1] A *natural image* is an image that might captured by a camera in a reasonably ordinary environment, as opposed to synthetically rendered images, screenshots of web pages, etc.

a complete understanding of the entire input. If even one element of the vector is unusual, the system must assign it a low probability.

- Denoising: given a damaged or incorrectly observed input $\tilde{\boldsymbol{x}}$, the machine learning system returns an estimate of the original or correct $\boldsymbol{x}$. For example, the machine learning system might be asked to remove dust or scratches from an old photograph. This requires multiple outputs (every element of the estimated clean example $\boldsymbol{x}$) and an understanding of the entire input (since even one damaged area will still reveal the final estimate as being damaged).

- Missing value imputation: given the observations of some elements of $\boldsymbol{x}$, the model is asked to return estimates of or a probability distribution over some or all of the unobserved elements of $\boldsymbol{x}$. This requires multiple outputs, and because the model could be asked to restore any of the elements of $\boldsymbol{x}$, it must understand the entire input.

- Sampling: the model generates new samples from the distribution $p(\boldsymbol{x})$. Applications include speech synthesis, i.e. producing new waveforms that-sound like natural human speech. This requires multiple output values and a good model of the entire input. If the samples have even one element drawn from the wrong distribution, then the sampling process is wrong.

For an example of the sampling tasks on small natural images, see Fig. 13.1.

Modeling a rich distribution over thousands or millions of random variables is a challenging task, both computationally and statistically. Suppose we only wanted to model binary variables. This is the simplest possible case, and yet already it seems overwhelming. For a small, $32 \times 32$ pixel color (RGB) image, there are $2^{3072}$ possible binary images of this form. This number is over $10^{800}$ times larger than the estimated number of atoms in the universe.

In general, if we wish to model a distribution over a random vector $\mathbf{x}$ containing $n$ discrete variables capable of taking on $k$ values each, then the naive approach of representing $P(\mathbf{x})$ by storing a lookup table with one probability value per possible outcome requires $k^n$ parameters!

This is not feasible for several reasons:

- **Memory: the cost of storing the representation** : For all but very small values of $n$ and $k$, representing the distribution as a table will require too many values to store.

- **Statistical efficiency**: As the number of parameters in a model increases, so does the amount of training examples needed to choose the values of those parameters using a statistical estimator. Because the table-based model has

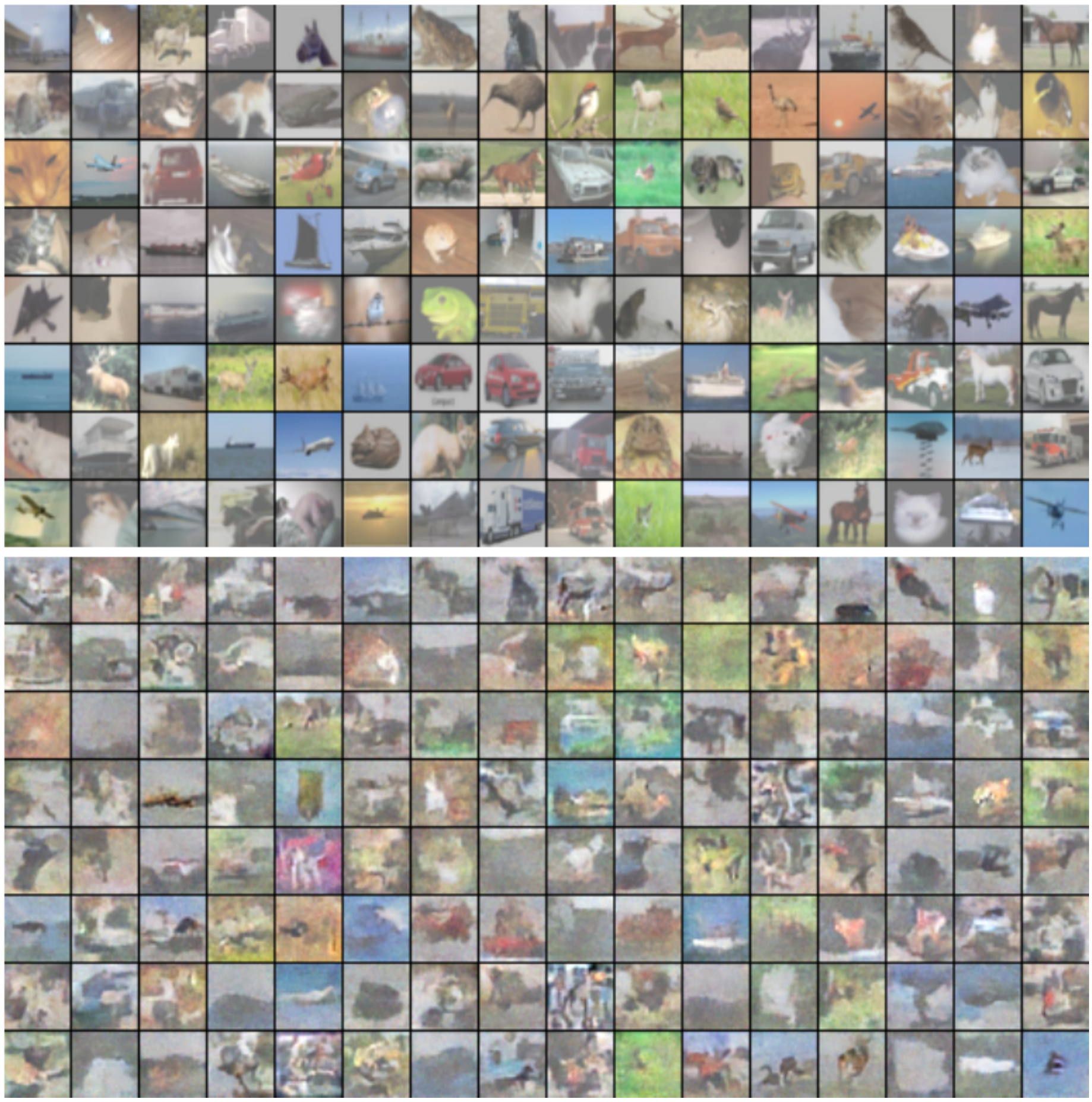

Figure 13.1: Probabilistic modeling of natural images. *Top:* Example $32 \times 32$ pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). *Bottom:* Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training example that is closest to it in Euclidean space. This comparison allows us to see that the model is truly synthesizing new images, rather than memorizing the training data. Contrast of both sets of images has been adjusted for display. Figure reproduced with permission from (Courville *et al.*, 2011).

an astronomical number of parameters, it will require an astronomically large training set to fit accurately. Any such model will overfit the training set very badly.

- **Runtime: the cost of inference**: Suppose we want to perform an *inference* task where we use our model of the joint distribution $P(\mathbf{x})$ to compute some other distribution, such as the marignal distribution $P(\mathrm{x}_1)$ or the conditional distribution $P(\mathrm{x}_2 \mid \mathrm{x}_1)$. Computing these distributions will require summing across the entire table, so the runtime of these operations is as high as the intractable memory cost of storing the model.

- **Runtime: the cost of sampling**: Likewise, suppose we want to draw a sample from the model. The naive way to do this is to sample some value $u \sim U(0, 1)$, then iterate through the table adding up the probability values until they exceed $u$ and return the outcome whose probability value was added last. This requires reading through the whole table in the worst case, so it has the same exponential cost as the other operations.

The problem with the table-based approach is that we are explicitly modeling every possible kind of interaction between every possible subset of variables. The probability distributions we encounter in real tasks are much simpler than this. Usually, most variables influence each other only indirectly.

For example, consider modeling the finishing times of a team in a relay race. Suppose the team consists of three runners, Alice, Bob, and Carol. At the start of the race, Alice carries a baton and begins running around a track. After completing her lap around the track, she hands the baton to Bob. Bob then runs his own lap and hands the baton to Carol, who runs the final lap. We can model each of their finishing times as a continuous random variable. Alice's finishing time does not depend on anyone else's, since she goes first. Bob's finishing time depends on Alice's, because Bob does not have the opportunity to start his lap until Alice has completed hers. If Alice finishes faster, Bob will finish faster, all else being equal. Finally, Carol's finishing time depends on both her teammates. If Alice is slow, Bob will probably finish late too, and Carol will have quite a late starting time and thus is likely to have a late finishing time as well. However, Carol's finishing time depends only *indirectly* on Alice's finishing time via Bob's. If we already know Bob's finishing time, we won't be able to estimate Carol's finishing time better by finding out what Alice's finishing time was. This means we can model the relay race using only two interactions: Alice's effect on Bob, and Bob's effect on Carol. We can omit the third, indirect interaction between Alice and Carol from our model.

Structured probabilistic models provide a formal framework for modeling only direct interactions between random variables. This allows the models to have

significantly fewer parameters which can in turn be estimated reliably from less data. These smaller models also have dramatically reduced computation cost in terms of storing the model, performing inference in the model, and drawing samples from the model.

## 13.2 Using Graphs to Describe Model Structure

Structured probabilistc models use graphs (in the graph theory sense of "nodes" or "vertices" connected by edges) to represent interactions between random variables. Each node represents a random variable. Each edge represents a direct interaction. These direct interactions imply other, indirect interactions, but only the direct interactions need to be explicitly modeled.

There is more than one way to describe the interactions in a probability distribution using a graph. In the following sections we describe some of the most popular and useful approaches.

### 13.2.1 Directed Models

One kind of structured probabilistic model is the *directed graphical model* otherwise known as the *belief network* or *Bayesian network*[2] (Pearl, 1985).

Directed graphical models are called "directed" because their edges are directed, that is, they point from one vertex to another. This direction is represented in the drawing with an arrow. The direction of the arrow indicates which variable's probability distribution is defined in terms of the other's. Drawing an arrow from a to b means that we define the probability distribution over b via a conditional distribution, with a as one of the variables on the right side of the conditioning bar. In other words, the distribution over b depends on the value of a.

Let's continue with the relay race example from Section 13.1. Suppose we name Alice's finishing time $\mathrm{t}_0$, Bob's finishing time $\mathrm{t}_1$, and Carol's finishing time $\mathrm{t}_2$. As we saw earlier, our estimate of $\mathrm{t}_1$ depends on $\mathrm{t}_0$. Our estimate of $\mathrm{t}_2$ depends directly on $\mathrm{t}_1$ but only indirectly on $\mathrm{t}_0$. We can draw this relationship in a directed graphical model, illustrated in Fig. 13.2.

Formally, a directed graphical model defined on variables $\mathbf{x}$ is defined by a directed acyclic graph $\mathcal{G}$ whose vertices are the random variables in the model, and a set of *local conditional probability distributions* $p(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i))$ where $Pa_{\mathcal{G}}(\mathrm{x}_i)$

[2] Judea Pearl suggested using the term Bayes Network when one wishes to "emphasize the judgmental" nature of the values computed by the network, i.e. to highlight that they usually represent degrees of belief rather than frequencies of events.

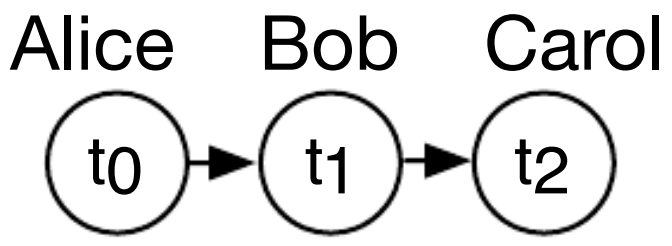


Figure 13.2: A directed graphical model depicting the relay race example. Alice's finishing time $t_0$ influences Bob's finishing time $t_1$, because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob's finishing time $t_1$ influences Carol's finishing time $t_2$.

gives the parents of $x_i$ in $\mathcal{G}$. The probability distribution over $\mathbf{x}$ is given by

$$p(\mathbf{x}) = \Pi_i p(x_i \mid Pa_{\mathcal{G}}(x_i)).$$

In our relay race example, this means that, using the graph drawn in Fig. 13.2,

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 \mid t_0)p(t_2 \mid t_1).$$

This is our first time seeing a structured probabilistic model in action. We can examine the cost of using it, in order to observe how structured modeling has many advantages relative to unstructured modeling.

Suppose we represented time by discretizing time ranging from minute 0 to minute 10 into 6 second chunks. This would make $t_0$, $t_1$, and $t_2$ each be discrete variables with 100 possible values. If we attempted to represent $p(t_0, t_1, t_2)$ with a table, it would need to store 999,999 values (100 values of $t_0$ $\times$ 100 values of $t_1$ $\times$ 100 values of $t_2$, minus 1, since the probability of one of the configurations is made redundant by the constraint that the sum of the probabilities be 1). If instead, we only make a table for each of the conditional probability distributions, then the distribution over $t_0$ requires 99 values, the table defining $t_1$ given $t_0$ requires 9900 values, and so does the table defining $t_2$ and $t_1$. This comes to a total of 19,899 values. This means that using the directed graphical model reduced our number of parameters by a factor of more than 50!

In general, to model $n$ discrete variables each having $k$ values, the cost of the single table approach scales like $O(k^n)$, as we've observed before. Now suppose we build a directed graphical model over these variables. If $m$ is the maximum number of variables appearing (on either side of the conditioning bar) in a single conditional probability distribution, then the cost of the tables for the directed model scales like $O(k^m)$. As long as we can design a model such that $m << n$, we get very dramatic savings.

In other words, so long as each variable has few parents in the graph, the distribution can be represented with very few parameters. Some restrictions on

the graph structure (e.g. it is a tree) can also guarantee that operations like computing marginal or conditional distributions over subsets of variables are efficient.

It's important to realize what kinds of information can be encoded in the graph, and what can't be. The graph just encodes simplifying assumptions about which variables are conditionally independent from each other. It's also possible to make other kinds of simplifying assumptions. For example, suppose we assume Bob always runs the same regardless of how Alice performed. (In reality, Alice's performance probably influences Bob's performance–depending on Bob's personality, if Alice runs especially fast in a given race, this might encourage Bob to push hard and match her exceptional performance, or it might make him overconfident and lazy). Then the only effect Alice has on Bob's finishing time is that we must add Alice's finishing time to the total amount of time we think Bob needs to run. This observation allows us to define a model with $O(k)$ parameters instead of $O(k^2)$. However, note that $\mathrm{t}_0$ and $\mathrm{t}_1$ are still directly dependent with this assumption, because $\mathrm{t}_1$ represents the absolute time at which Bob finishes, not the total time he himself spends running. This means our graph must still contain an arrow from $\mathrm{t}_0$ to $\mathrm{t}_1$. The assumption that Bob's personal running time is independent from all other factors cannot be encoded in a graph over $\mathrm{t}_0$, $\mathrm{t}_1$, and $\mathrm{t}_2$. Instead, we encode this information in the definition of the conditional distribution itself. The conditional distribution is no longer a $k \times k - 1$ element table indexed by $\mathrm{t}_0$ and $\mathrm{t}_1$ but is now a slightly more complicated formula using only $k - 1$ parameters. The directed graphical model syntax does not place any constraint on how we define our conditional distributions. It only defines which variables they are allowed to take in as arguments.

### 13.2.2 Undirected Models

Directed graphical models give us one language for describing structured probabilistic models. Another popular language is that of *undirected models*, otherwise known as *Markov random fields* (MRFs) or *Markov networks* (Kindermann, 1980). As their name implies, undirected models use graphs whose edges are undirected.

Directed models are most naturally applicable to situations where there is a clear reason to draw each arrow in one particular direction. Often these are situations where we understand the causality, and the causality only flows in one direction. One such situation is the relay race example. Earlier runners affects the finishing times of later runners; later runners do not affect the finishing times of earlier runners.

Not all situations we might want to model have such a clear direction to their interactions. When the interactions seem to have no intrinsic direction, or to operate in both directions, it may be more appropriate to use an undirected

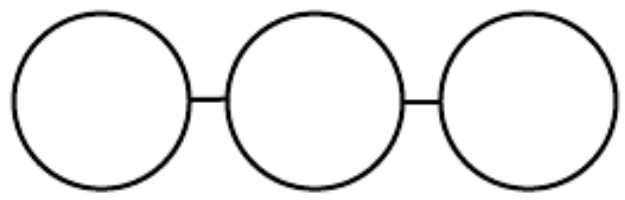

Figure 13.3: An undirected graph representing how your roommate's health h , your health h , and your work colleague's health h affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague don't know each other, they can only infect each other indirectly via you.

model.

As an example of such a situation, suppose we want to model a distribution over three binary variables: whether or not you are sick, whether or not your coworker is sick, and whether or not your roommate is sick. As in the relay race example, we can make simplifying assumptions about the kinds of interactions that take place. Assuming that your coworker and your roommate do not know each other, it is very unlikely that one of them will give the other a disease such as a cold directly. This event can be seen as so rare that it is acceptable not to model it. However, it is reasonably likely that either of them could give you a cold, and that you could pass it on to the other. We can model the indirect transmission of a cold from your coworker to your roommate by modeling the transmission of the cold from your coworker to you and the transmission of the cold from you to your roommate.

In this case, it's just as easy for you to cause your roommate to get sick as it is for your roommate to make you sick, so there is not a clean, uni-directional narrative on which to base the model. This motivates using an undirected model. As with directed models, if two nodes in an undirected model are connected by an edge, then the random variables corresponding to those nodes interact with each other directly. Unlike directed models, the edge in an undirected model has no arrow, and is not associated with a conditional probability distribution.

Let's call the random variable representing your health $\mathrm{h}_y$, the random variable representing your roommate's health $\mathrm{h}_r$, and the random variable representing your colleague's health $\mathrm{h}_c$. See Fig. 13.3 for a drawing of the graph representing this scenario.

Formally, an undirected graphical model is a structured probabilistic model defined on an undirected graph $\mathcal{G}$. For each clique $\mathcal{C}$ in the graph[3], a *factor* $\phi(\mathcal{C})$ (also called a *clique potential*) measures the affinity of the variables in that clique for being in each of their possible joint states. The factors are constrained to be

[3] A clique of the graph is a subset of nodes that are all connected to each other by an edge of the graph.

non-negative. Together they define an *unnormalized probability distribution*

$$\tilde{p}(\mathbf{x}) = \Pi_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}).$$

The unnormalized probability distribution is efficient to work with so long as all the cliques are small. It encodes the idea that states with higher affinity are more likely. However, unlike in a Bayesian network, there is little structure to the definition of the cliques, so there is nothing to guarantee that multiplying them together will yield a valid probability distribution. See Fig. 13.4 for an example of reading factorization information from an undirected graph.

Our example of the cold spreading between you, your roommate, and your colleague contains two cliques. One clique contains $\mathrm{h}_y$ and $\mathrm{h}_c$. The factor for this clique can be defined by a table, and might have values resembling these:

| | $\mathrm{h}_y = 0$ | $\mathrm{h}_y = 1$ |
|---|---|---|
| $\mathrm{h}_c = 0$ | 2 | 1 |
| $\mathrm{h}_c = 1$ | 1 | 10 |

A state of 1 indicates good health, while a state of 0 indicates poor health (having been infected with a cold). Both of you are usually healthy, so the corresponding state has the highest affinity. The state where only one of you is sick has the lowest affinity, because this is a rare state. The state where both of you are sick (because one of you has infected the other) is a higher affinity state, though still not as common as the state where both are healthy.

To complete the model, we would need to also define a similar factor for the clique containing $\mathrm{h}_y$ and $\mathrm{h}_r$.

### 13.2.3 The Partition Function

While the unnormalized probability distribution is guaranteed to be non-negative everywhere, it is not guaranteed to sum or integrate to 1. To obtain a valid probability distribution, we must use the corresponding normalized probability distribution[4]:

$$p(\mathbf{x}) = \frac{1}{Z} \tilde{p}(\mathbf{x})$$

where $Z$ is the value that results in the probability distribution summing or integrating to 1:

$$Z = \quad \tilde{p}(\mathbf{x}) d\mathbf{x}.$$

[4] A distribution defined by normalizing a product of clique potentials is also called a *Gibbs distribution*.

You can think of $Z$ as a constant when the $\phi$ functions are held constant. Note that if the $\phi$ functions have parameters, then $Z$ is a function of those parameters. It is common in the literature to write $Z$ with its arguments omitted to save space. $Z$ is known as the *partition function*, a term borrowed from statistical physics.

Since $Z$ is an integral or sum over all possible joint assignments of the state $\mathbf{x}$ it is often intractable to compute. In order to be able to obtain the normalized probability distribution of an undirected model, the model structure and the definitions of the $\phi$ functions must be conducive to computing $Z$ efficiently. In the context of deep learning, $Z$ is usually intractable, and we must resort to approximations. Such approximate algorithms are the topic of Chapter 18.

One important consideration to keep in mind when designing undirected models is that it is possible for $Z$ not to exist. This happens if some of the variables in the model are continuous and the integral of $\tilde{p}$ over their domain diverges. For example, suppose we want to model a single scalar variable $\mathrm{x} \in$ with a single clique potential $\phi(x) = x^2$. In this case,

$$Z = \quad x^2 dx.$$

Since this integral diverges, there is no probability distribution corresponding to this choice of $\phi(x)$. Sometimes the choice of some parameter of the $\phi$ functions determines whether the probability distribution is defined. For example, for $\phi(x; \beta) = \exp\ -\beta x^2$ , the $\beta$ parameter determines whether $Z$ exists. Positive $\beta$ results in a Gaussian distribution over x but all other values of $\beta$ make $\phi$ impossible to normalize.

One key difference between directed modeling and undirected modeling is that directed models are defined directly in terms of probability distributions from the start, while undirected models are defined more loosely by $\phi$ functions that are then converted into probability distributions. This changes the intuitions one must develop in order to work with these models. One key idea to keep in mind while working with undirected models is that the domain of each of the variables has dramatic effect on the kind of probability distribution that a given set of $\phi$ functions corresponds to. For example, consider an $n$-dimensional vector-valued random variable $\mathbf{x}$ and an undirected model parameterized by a vector of biases $\boldsymbol{b}$. Suppose we have one clique for each element of $\mathbf{x}$, $\phi_i(\mathrm{x}_i) = \exp(b_i \mathrm{x}_i)$. What kind of probability distribution does this result in? The answer is that we don't have enough information, because we have not yet specified the domain of $\mathbf{x}$. If $\mathbf{x} \in \quad ^n$, then the integral defining $Z$ diverges and no probability distribution exists. If $\mathbf{x} \in \{0, 1\}^n$, then $p(\mathbf{x})$ factorizes into $n$ independent distributions, with $p(\mathrm{x}_i = 1) = \mathrm{sigmoid}\,(b_i)$. If the domain of $\mathbf{x}$ is the set of elementary basis vectors $(\{[1, 0, \ldots, 0], [0, 1, \ldots, 0], \ldots, [0, 0, \ldots, 1]\}\ )$ then $p(\mathbf{x}) = \mathrm{softmax}(b)$, so a large value of $b_i$ actually reduces $p(\mathrm{x}_j = 1)$ for $j \neq i$. Often, it is possible to

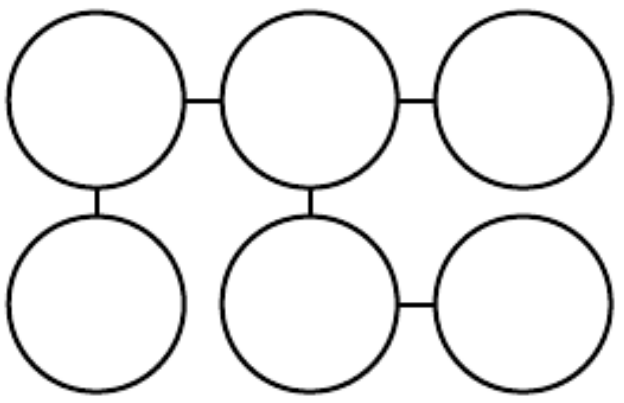

Figure 13.4: This graph implies that $p(A, B, C, D, E, F)$ can be written as $^{1}\phi \quad (A, B)\phi \quad (B, C)\phi \quad (A, D)\phi \quad (B, E)\phi \quad (E, F)$ for an appropriate choice of the $\phi$ functions.

leverage the effect of a carefully chosen domain of a variable in order to obtain complicated behavior from a relatively simple set of $\phi$ functions. We'll explore a practical application of this idea later, in Chapter 20.7.

### 13.2.4 Energy-Based Models

Many interesting theoretical results about undirected models depend on the assumption that $\forall \mathbf{x}, \tilde{p}(\mathbf{x}) > 0$. A convenient way to enforce this to use an *energy-based model* (EBM) where

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \tag{13.1}$$

and $E(\mathbf{x})$ is known as the *energy function*. Because $\exp(z)$ is positive for all $z$, this guarantees that no energy function will result in a probability of zero for any state $\mathbf{x}$. Being completely free to choose the energy function makes learning simpler. If we learned the clique potentials directly, we would need to use constrained optimization, and we would need to arbitrarily impose some specific minimal probability value. By learning the energy function, we can use unconstrained optimization[5], and the probabilities in the model can approach arbitrarily close to zero but never reach it.

Any distribution of the form given by equation 13.1 is an example of a *Boltzmann distribution*. For this reason, many energy-based models are called *Boltzmann machines*. There is no accepted guideline for when to call a model an energy-based model and when to call it a Boltzmann machines. The term Boltzmann machine was first introduced to describe a model with exclusively binary variables, but today many models such as the mean-covariance restricted Boltzmann machine incorporate real-valued variables as well.

[5] For some models, we may still need to use constrained optimization to make sure exists.

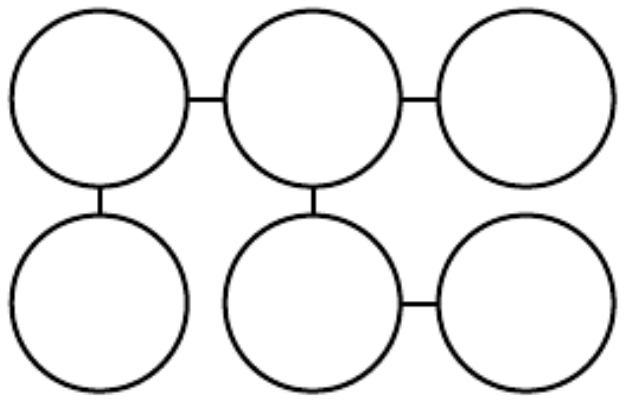

Figure 13.5: This graph implies that $E(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f})$ can be written as $E_{\mathrm{a\,b}}(\mathrm{a}, \mathrm{b}) + E_{\mathrm{b\,c}}(\mathrm{b}, \mathrm{c}) + E_{\mathrm{a\,d}}(\mathrm{a}, \mathrm{d}) + E_{\mathrm{b\,e}}(\mathrm{b}, \mathrm{e}) + E_{\mathrm{e\,f}}(\mathrm{e}, \mathrm{f})$ for an appropriate choice of the per-clique energy functions. Note that we can obtain the $\phi$ functions in Fig. 13.4 by setting each $\phi$ to the exp of the corresponding negative energy, e.g., $\phi_{\mathrm{a\,b}}(\mathrm{a}, \mathrm{b}) = \exp(-E(\mathrm{a}, \mathrm{b}))$.

Cliques in an undirected graph correspond to factors of the unnormalized probability function. Because $\exp(a)\exp(b) = \exp(a+b)$, this means that different cliques in the undirected graph correspond to the different terms of the energy function. In other words, an energy-based model is just a special kind of Markov network: the exponentiation makes each term in the energy function correspond to a factor for a different clique. See Fig. 13.5 for an example of how to read the form of the energy function from an undirected graph structure.

One part of the definition of an energy-based model serves no functional purpose from a machine learning point of view: the $-$ sign in Eq. 13.1. This $-$ sign could be incorporated into the definition of $E$, or for many functions $E$ the learning algorithm could simply learn parameters with opposite sign. The $-$ sign is present primarily to preserve compatibility between the machine learning literature and the physics literature. Many advances in probabilistic modeling were originally developed by statistical physicists, for whom $E$ refers to actual, physical energy and does not have arbitrary sign. Terminology such as "energy" and "partition function" remains associated with these techniques, even though their mathematical applicability is broader than the physics context in which they were developed. Some machine learning researchers (e.g., Smolensky (1986), who referred to negative energy as *harmony*) have chosen to emit the negation, but this is not the standard convention.

### 13.2.5 Separation and D-Separation

The edges in a graphical model tell us which variables directly interact. We often need to know which variables *indirectly* interact. Some of these indirect interactions can be enabled or disabled by observing other variables. More formally, we would like to know which subsets of variables are conditionally independent from each other, given the values of other subsets of variables.

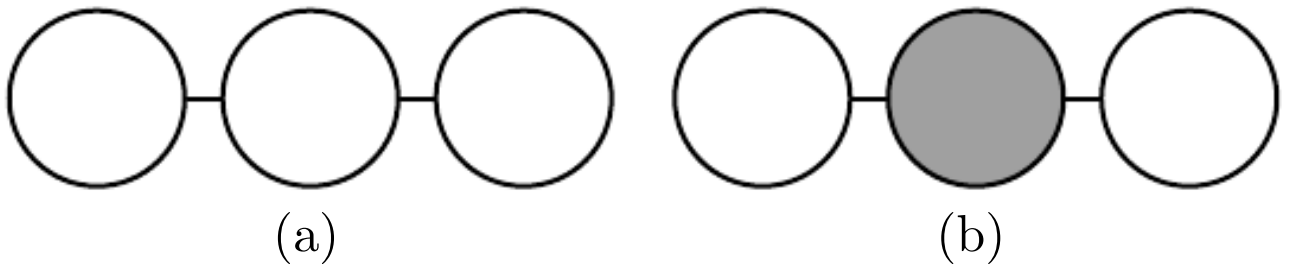


Figure 13.6: a) The path between random variable a and random variable b through s is active, because s is not observed. This means that a and b are not separated. b) Here s is shaded in, to indicate that it is observed. Because the only path between $ra$ and b is through s, and that path is inactive, we can conclude that a and b are separated given s.

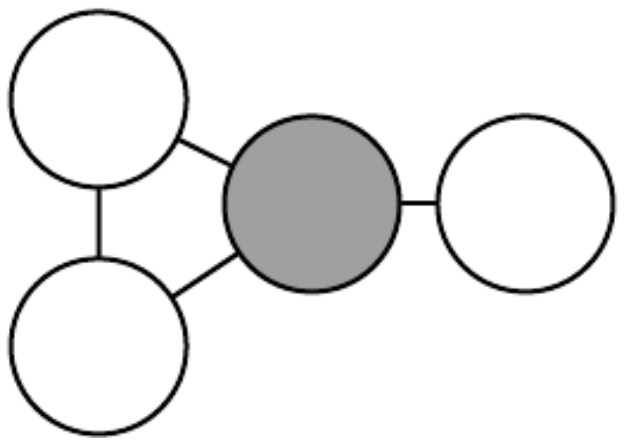

Figure 13.7: An example of reading separation properties from an undirected graph. Here b is shaded to indicate that it is observed. Because observing b blocks the only path from a to c, we say that a and c are separated from each other given b. The observation of b also blocks one path between a and d, but there is a second, active path between them. Therefore, a and d are not separated given b.

Identifying the conditional independences in a graph is very simple in the case of undirected models. In this case, conditional independence implied by the graph is called *separation*. We say that a set of variables is *separated* from another set of variables given a third set of variables if the graph structure implies that is independent from given . If two variables a and b are connected by a path involving only unobserved variables, then those variables are not separated. If no path exists between them, or all paths contain an observed variable, then they are separated. We refer to paths involving only unobserved variables as "active" and paths including an observed variable as "inactive."

When we draw a graph, we can indicate observed variables by shading them in. See Fig. 13.6 for a depiction of how active and inactive paths in an undirected look when drawn in this way. See Fig. 13.7 for an example of reading separation from an undirected graph.

Similar concepts apply to directed models, except that in the context of directed models, these concepts are referred to as *d-separation*. The "d" stands for "dependence." D-separation for directed graphs is defined the same as separation for undirected graphs: We say that a set of variables is d-separated from another set of variables given a third set of variables if the graph structure

implies that is independent from given .

As with undirected models, we can examine the independences implied by the graph by looking at what active paths exist in the graph. As before, two variables are dependent if there is an active path between them, and d-separated if no such path exists. In directed nets, determining whether a path is active is somewhat more complicated. See Fig. 13.8 for a guide to identifying active paths in a directed model. See Fig. 13.9 for an example of reading some properties from a graph.

It is important to remember that separation and d-separation tell us only about those conditional independences *that are implied by the graph.* There is no requirement that the graph imply all independences that are present. In particular, it is always legitimate to use the complete graph (the graph with all possible edges) to represent any distribution. In fact, some distributions contain independences that are not possible to represent with existing graphical notation. *Context-specific independences* are independences that are present dependent on the value of some variables in the network. For example, consider a model of three binary variables, a, b, and c. Suppose that when a is 0, b and c are independent, but when a is 1, b is deterministically equal to c. Encoding the behavior when $a = 1$ requires an edge connecting b and c. The graph then fails to indicate that b and c are independent when $a = 0$.

In general, a graph will never imply that an independence exists when it does not. However, a graph may fail to encode an independence.

### 13.2.6 Converting between Undirected and Directed Graphs

In common parlance, we often refer to certain model classes as being undirected or directed. For example, we typically refer to RBMs as undirected and sparse coding as directed. This way of speaking can be somewhat leading, because no probabilistic model is inherently directed or undirected. Instead, some models are most easily *described* using a directed graph, or most easily described using an undirected graph.

Ever probability distribution can be represented by either a directed model or by an undirected model. In the worst case, one can always represent any distribution by using a "complete graph." In the case of a directed model, the complete graph is any directed acyclic graph where we impose some ordering on the random variables, and each variable has all other variables that precede it in the ordering as its ancestors in the graph. For an undirected model, the complete graph is simply a graph containing a single clique encompassing all of the variables.

Of course, the utility of a graphical model is that the graph implies that some variables do not interact directly. The complete graph is not very useful because

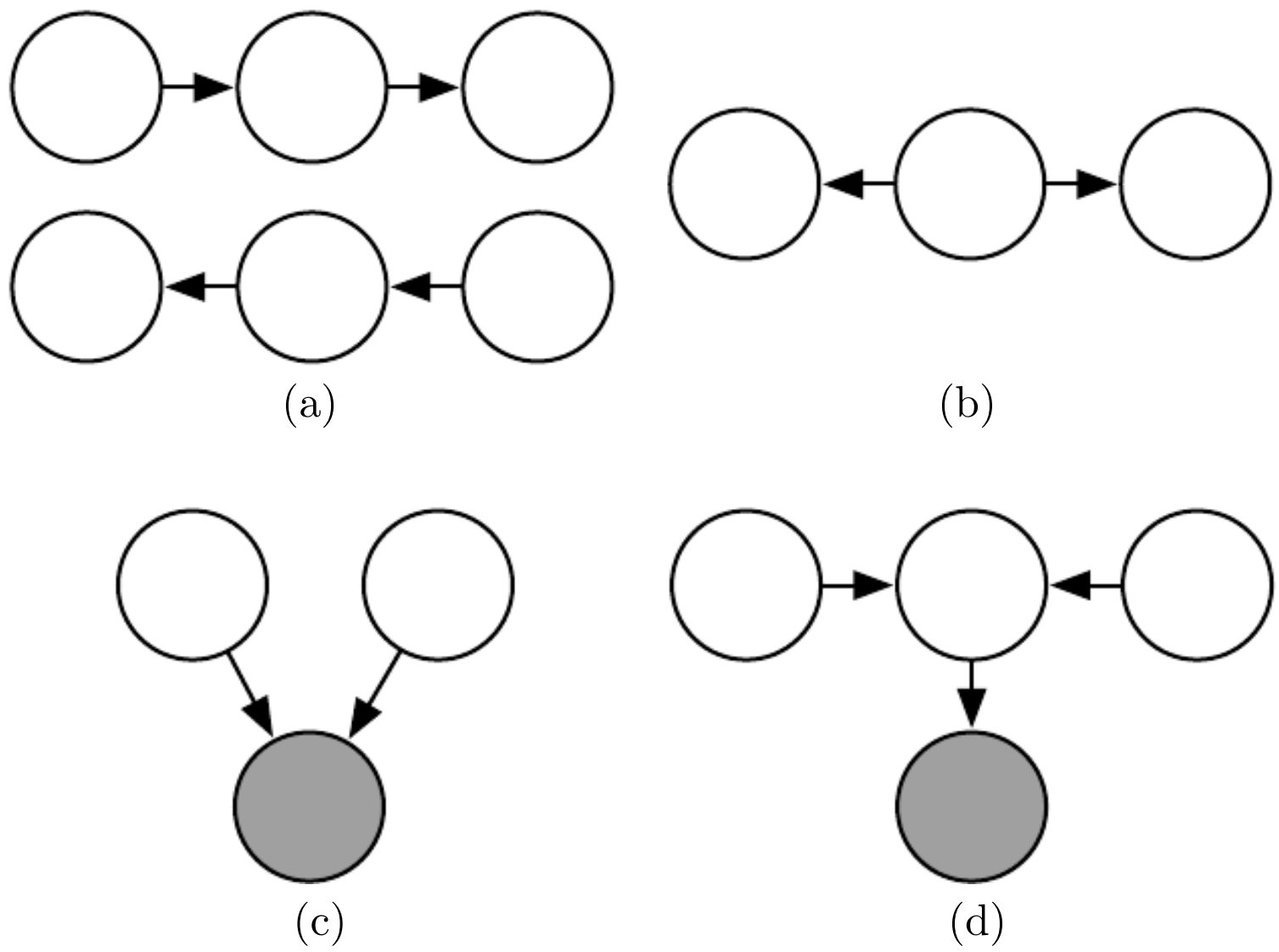


Figure 13.8: All of the kinds of active paths of length two that can exist between random variables a and *rb*. a) Any path with arrows proceeding directly from a to b or vice versa. This kind of path becomes blocked if s is observed. We have already seen this kind of path in the relay race example. b) a and b are connected by a *common cause* s. For example, suppose s is a variable indicating whether or not there is a hurricane and a and b measure the wind speed at two different nearby weather monitoring outposts. If we observe very high winds at station a, we might expect to also see high winds at b. This kind of path can be blocked by observing s. If we already know there is a hurricane, we expect to see high winds at b, regardless of what is observed at a. A lower than expected wind at a (for a hurricane) would not change our expectation of winds at b (knowing there is a hurricane). However, if s is not observed, then a and b are dependent, i.e., the path is inactive. c) a and b are both parents of s. This is called a *V-structure* or *the collider case*, and it causes a and a to be related by the *explaining away effect*. In this case, the path is actually active when s is observed. For example, suppose s is a variable indicating that your colleague is not at work. The variable a represents her being sick, while b represents her being on vacation. If you observe that she is not at work, you can presume she is probably sick or on vacation, but it's not especially likely that both have happened at the same time. If you find out that she is on vacation, this fact is sufficient to *explain* her absence, and you can infer that she is probably not also sick. d) The explaining away effect happens even if any descendant of s is observed! For example, suppose that c is a variable representing whether you have received a report from your colleague. If you notice that you have not received the report, this increases your estimate of the probability that she is not at work today, which in turn makes it more likely that she is either sick or on vacation. The only way to block a path through a V-structure is to observe none of the descendants of the shared child.

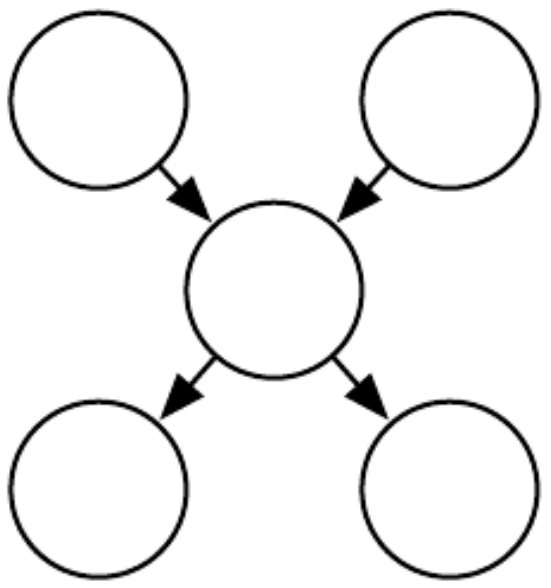

Figure 13.9

From this graph, we can read out several d-separation properties. Examples include:

- a and b are d-separated given the empty set.
- a and e are d-separated given c.
- d and e are d-separated given c.

We can also see that some variables are no longer d-separated when we observe some variables:

- a and b are not d-separated given c.
- a and b are not d-separated given d.

it does not imply any independences. TODO figure complete graph

When we represent a probability distribution with a graph, we want to choose a graph that implies as many independences as possible, without implying any independences that do not actually exist.

From this point of view, some distributions can be represented more efficiently using directed models, while other distributions can be represented more efficiently using undirected models. In other words, directed models can encode some independences that undirected models cannot encode, and vice versa.

Directed models are able to use one specific kind of substructure that undirected models cannot represent perfectly. This substructure is called an *immorality*. The structure occurs when two random variables a and b are both parents of a third random variable c, and there is no edge directly connecting a and b in either direction. (The name "immorality" may seem strange; it was coined in the graphical models literature as a joke about unmarried parents) To convert a directed model with graph $\mathcal{D}$ into an undirected model, we need to create a new graph $\mathcal{U}$. For every pair of variables x and y, we add an undirected edge connecting x and y to $\mathcal{U}$ if there is a directed edge (in either direction) connecting x and y in $\mathcal{D}$ or if x and y are both parents in $\mathcal{D}$ of a third variable z. The resulting $\mathcal{U}$ is known as a *moralized graph*. See Fig. 13.10 for examples of converting directed models to undirected models via moralization.

Likewise, undirected models can include substructures that no directed model can represent perfectly. Specifically, a directed graph $\mathcal{D}$ cannot capture all of the conditional independences implied by an undirected graph $\mathcal{U}$ if $\mathcal{U}$ contains a *loop* of length greater than three, unless that loop also contains a *chord*. A loop is a sequence of variables connected by undirected edges, with the last variable in the sequence connected back to the first variable in the sequence. A chord is a connection between any two non-consecutive variables in this sequence. If $\mathcal{U}$ has loops of length four or greater and does not have chords for these loops, we must add the chords before we can convert it to a directed model. Adding these chords discards some of the independence information that was encoded in $\mathcal{U}$. The graph formed by adding chords to $\mathcal{U}$ is known as a *chordal* or *triangulated* graph, because all the loops can now be described in terms of smaller, triangular loops. To build a directed graph $\mathcal{D}$ from the chordal graph, we need to also assign directions to the edges. When doing so, we must not create a directed cycle in $\mathcal{D}$, or the result does not define a valid directed probabilistic model. One way to assign directions to the edges in $\mathcal{D}$ is to impose an ordering on the random variables, then point each edge from the node that comes earlier in the ordering to the node that comes later in the ordering. TODO point to fig

IG HERE

TODO: started this above, need to scrap some some BNs encode indepen-

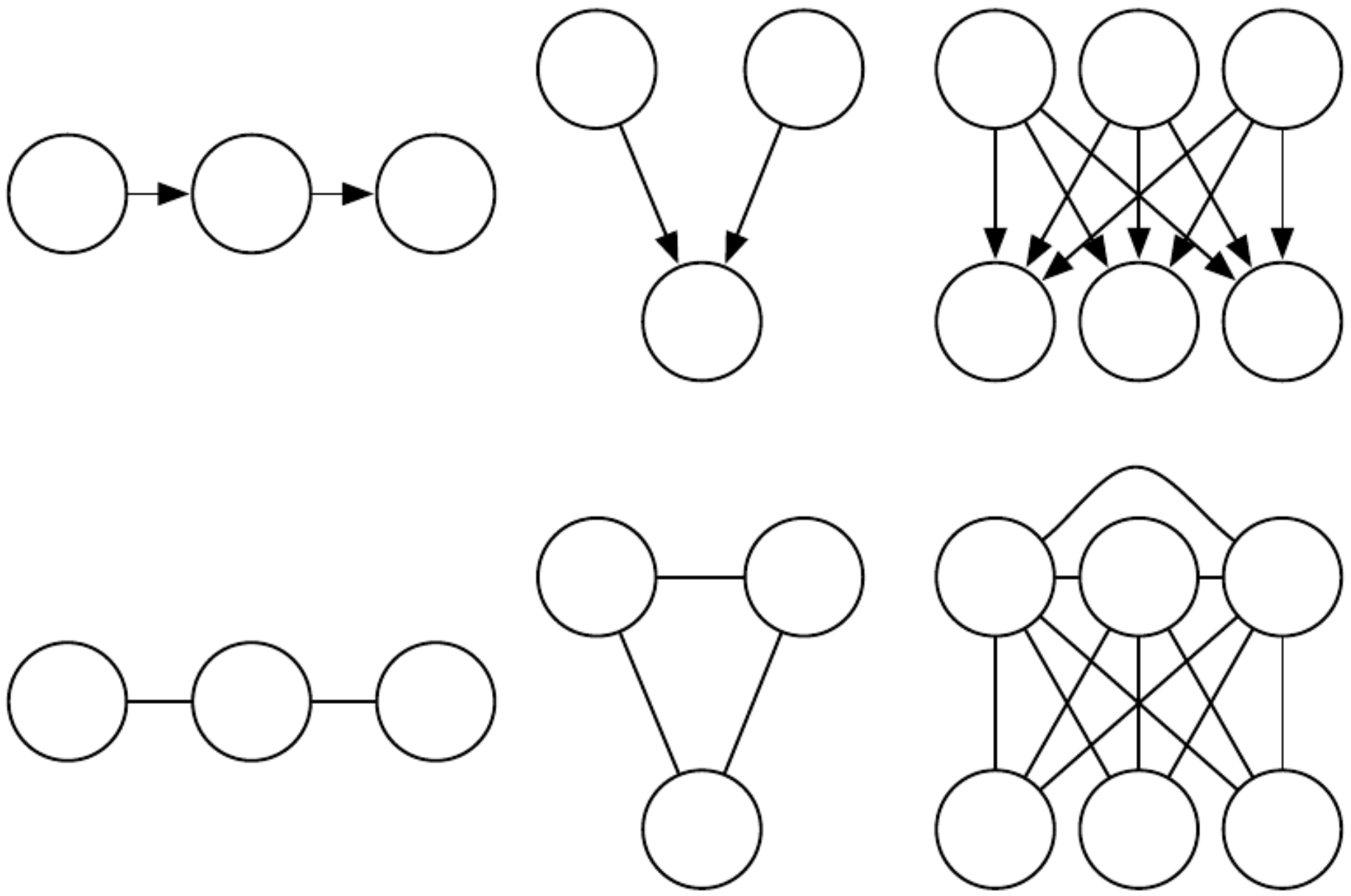

Figure 13.10: Examples of converting directed models to undirected models by constructing moralized graphs. *Left)* This simple chain can be converted to a moralized graph merely by replacing its directed edges with undirected edges. The resulting undirected model implies exactly the same set of independences and conditional independences. *Center)* This graph is the simplest directed model that cannot be converted to an undirected model without losing some independences. This graph consists entirely of a single immorality. Because a and b are parents of c, they are connected by an active path when c is observed. To capture this dependence, the undirected model must include a clique encompassing all three variables. This clique fails to encode the fact that $a \perp b$. *Right)* In general, moralization may add many edges to the graph, thus losing many implied independences. For example, this sparse coding graph requires adding moralizing edges between every pair of hidden units, thus introducing a quadratic number of new direct dependences.

dences that MNs can't encode, and vice versa example of BN that an MN can't encode: A and B are parents of C A is d-separated from B given the empty set The Markov net requires a clique over A, B, and C in order to capture the active path from A to B when C is observed This clique means that the graph cannot imply A is separated from B given the empty set example of a MN that a BN can't encode: A, B, C, D connected in a loop BN cannot have both A d-sep D given B, C and B d-sep C given A, D

In many cases, we may want to convert an undirected model to a directed model, or vice versa. To do so, we choose the graph in the new format that implies as many independences as possible, while not implying any independences that were not implied by the original graph.

To convert a directed model $\mathcal{D}$ to an undirected model $\mathcal{U}$, we re

TODO: conversion between directed and undirected models

### 13.2.7 Marginalizing Variables out of a Graph

TODO: marginalizing variables out of a graph

### 13.2.8 Factor Graphs

*Factor graphs* are another way of drawing undirected models that resolve an ambiguity in the graphical representation of standard undirected model syntax. In an undirected model, the scope of every $\phi$ function must be a subset of some clique in the graph. However, it is not necessary that there exist any $\phi$ whose scope contains the entirety of every clique. Factor graphs explicitly represent the scope of each $\phi$ function. Specifically, a factor graph is a graphical representation of an undirected model that consists of a bipartite undirected graph. Some of the nodes are drawn as circles. These nodes correspond to random variables as in a standard undirected model. The rest of the nodes are drawn as squares. These nodes correspond to the factors $\phi$ of the unnormalized probability distribution. Variables and factors may be connected with undirected edges. A variable and a factor are connected in the graph if and only if the variable is one of the arguments to the factor in the unnormalized probability distribution. No factor may be connected to another factor in the graph, nor can a variable be connected to a variable. See Fig. 13.11 for an example of how factor graphs can resolve ambiguity in the interpretation of undirected networks.

## 13.3 Advantages of Structured Modeling

TODO– note that we have already shown that some things are cheaper in the sections where we introduce the modeling syntax

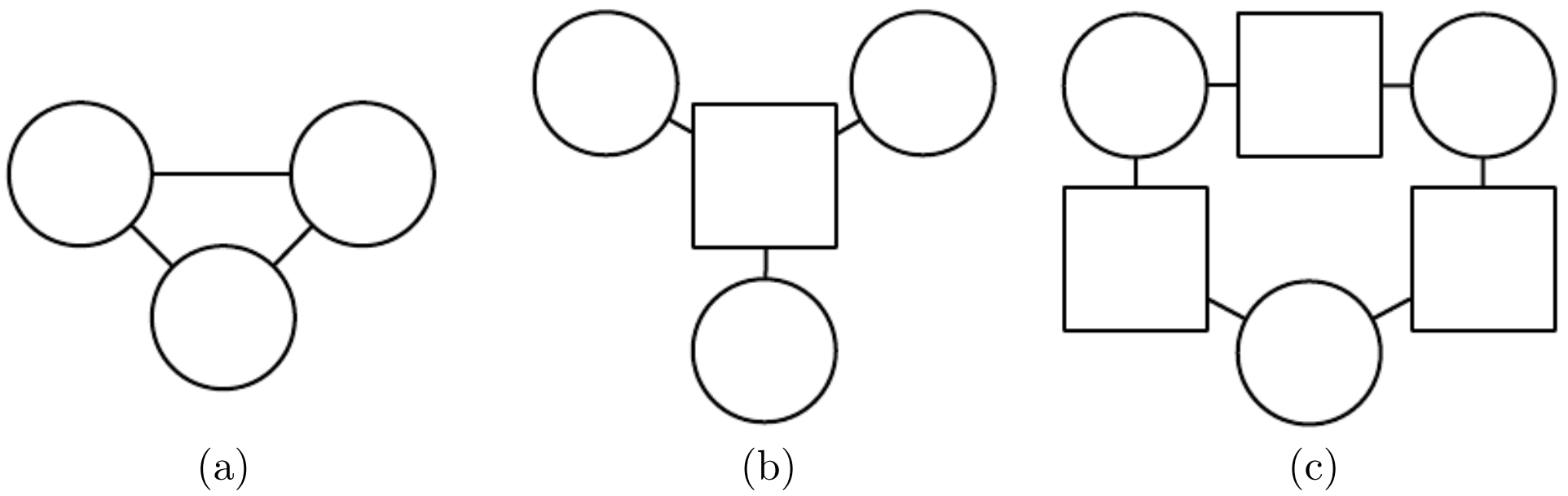


Figure 13.11: An example of how a factor graph can resolve ambiguity in the interpretation of undirected networks. a) An undirected network with a clique involving three variables a, b, and c. b) A factor graph corresponding to the same undirected model. This factor graph has one factor over all three variables. c) Another valid factor graph for the same undirected model. This factor graph has three factors, each over only two variables. Note that representation, inference, and learning are all asymptotically cheaper in (c) compared to (b), even though both require the same undirected graph to represent. TODO: make sure figure respects random variable notation

TODO: revisit each of the three challenges from sec:unstructured TODO: hammer point that graphical models convey information by leaving edges out TODO: need to show reduced cost of sampling, but first reader needs to know about ancestral and gibbs sampling.... TODO: benefit of separating representation from learning and inference

## 13.4 Learning about Dependencies

We describe here two types of random variables: observed or "visible" variables $\mathbf{v}$ and latent or "hidden" variables $\mathbf{h}$. The observed variables $\mathbf{v}$ correspond to the variables actually provided in the data set during training. $\mathbf{h}$ consists of variables that are introduced to the model in order to help it explain the structure in $\mathbf{v}$. Generally the exact semantics of $\mathbf{h}$ depend on the model parameters and are created by the learning algorithm. The motivation for this is twofold.

### 13.4.1 Latent Variables Versus Structure Learning

Often the different elements of $\mathbf{v}$ are highly dependent on each other. A good model of $\mathbf{v}$ which did not contain any latent variables would need to have very large numbers of parents per node in a Bayesian network or very large cliques in a Markov network. Just representing these higher order interactions is costly–both in a computational sense, because the number of parameters that must be stored

in memory scales exponentially with the number of members in a clique, but also in a statistical sense, because this exponential number of parameters requires a wealth of data to estimate accurately.

There is also the problem of learning which variables need to be in such large cliques. An entire field of machine learning called *structure learning* is devoted to this problem . For a good reference on structure learning, see (Koller and Friedman, 2009). Most structure learning techniques are a form of greedy search. A structure is proposed, a model with that structure is trained, then given a score. The score rewards high training set accuracy and penalizes model complexity. Candidate structures with a small number of edges added or removed are then proposed as the next step of the search, and the search proceeds to a new structure that is expected to increase the score.

Using latent variables instead of adaptive structure avoids the need to perform discrete searches and multiple rounds of training. A fixed structure over visible and hidden variables can use direct interactions between visible and hidden units to impose indirect interactions between visible units. Using simple parameter learning techniques we can learn a model with a fixed structure that imputes the right structure on the marginal $p(\mathrm{v})$.

### 13.4.2 Latent Variables for Feature Learning

Another advantage of using latent variables is that they often develop useful semantics.

As discussed in section 3.10.6, the mixture of Gaussians model learns a latent variable that corresponds to which category of examples the input was drawn from. This means that the latent variable in a mixture of Gaussians model can be used to do classification.

In Chapter 15 we saw how simple probabilistic models like sparse coding learn latent variables that can be used as input features for a classifier, or as coordinates along a manifold. Other models can be used in this same way, but deeper models and models with different kids of interactions can create even richer descriptions of the input. Most of the approaches mentioned in sec. 13.4.2 accomplish feature learning by learning latent variables. Often, given some model of $\mathbf{v}$ and $\mathbf{h}$, it turns out that $[\mathbf{h} \mid \mathbf{v}]$ TODO: uh-oh, is there a collision between set notation and expectation notation? or argmax $p(\boldsymbol{h}, \boldsymbol{v})$ is a good feature mapping for $\boldsymbol{v}$.

TODO: appropriate links to Monte Carlo methods chapter spun off from here

## 13.5 Inference and Approximate Inference over Latent Variables

As soon as we introduce latent variables in a graphical model, this raises the question: how to choose values of the latent variables $\boldsymbol{h}$ given values of the visible variables $\boldsymbol{x}$? This is what we call *inference*, in particular inference over the latent variables. The general question of inference is to guess some variables given others.

TODO: inference has definitely been introduced above... TODO: mention loopy BP, show how it is very expensive for DBMs

TODO: briefly explain what variational inference is and reference approximate inference chapter

### 13.5.1 Reparametrization Trick

Sometimes, in order to estimate the stochastic gradient of an expected loss over some random variable $\mathbf{h}$, with respect to parameters that influence $\mathbf{h}$, we would like to compute gradients through $\mathbf{h}$, i.e., on the parameters that influenced the probability distribution from which $\mathbf{h}$ was sampled. If $\mathbf{h}$ is continuous-valued, this is generally possible by using the *reparametrization trick*, i.e., rewriting

$$\mathbf{h} \sim p(\mathbf{h} \mid \theta) \tag{13.2}$$

as

$$\mathbf{h} = f(\theta, \eta) \tag{13.3}$$

where $\eta$ is some independent noise source of the appropriate dimension with density $p(\eta)$, and $f$ is a continuous (differentiable almost everywhere) function. The reparametrization trick is the idea that if the random variable to be integrated over is continuous, we can *back-propagate* through the process that gave rise to it in order to figure how to change that process.

For example, let us suppose we want to estimate the expected gradient

$$\frac{\partial}{\partial \theta} \quad L(\mathbf{h}) p(\mathbf{h} \mid \theta) d\mathbf{h} \tag{13.4}$$

where the parameters $\theta$ influences the random variable $\mathbf{h}$ which in term influence our loss $L$. A very efficient (Kingma and Welling, 2014b; Rezende *et al.*, 2014) way to achieve[6] this is to perform the reparametrization in Eq. 13.3 and the corresponding change of variable in the integral of Eq. 13.4, integrating over $\eta$ rather than $\mathbf{h}$:

$$\frac{\partial}{\partial \theta} \quad L(f(\theta, \eta)) p(eta) d\eta. \tag{13.5}$$

[6] compared to approaches that do not back-propagate through the generation of

We can now more easily enter the derivative in the integral, getting

$$g = \frac{\partial L(f(\theta, \eta))}{\partial \theta} p(eta) d\eta.$$

Finally, we get a stochastic gradient estimator

$$\hat{g} = \frac{\partial L(f(\theta, \eta))}{\partial \theta}$$

where we sampled $\eta \sim p(\eta)$ and $E[\hat{g}] = g$.

This trick was first introduced to machine learning by Williams (1992) in the context of back-propagation. The idea of back-propagating through a sampling process was independently re-discovered and re-popularized in the context of deep generative models more recently by Bengio (2013b); Bengio *et al.* (2013a); Kingma (2013). The underlying principle was actually discovered much earlier in the information theory community (Price, 1958; Bonnet, 1964). The theorems from Price (1958); Bonnet (1964) were first used in a variational context by Opper and Archambeau (2009); Challis and Barber (2012) and for efficient Bayesian inference by Salimans and Knowles (2013). It was further developed and analyzed by Kingma and Welling (2014b) and was used to train generative stochastic networks (GSNs) (Bengio *et al.*, 2014a,b), described in Section 20.12, which can be viewed as recurrent networks with noise injected both in input and hidden units (with each time step corresponding to one step of a generative Markov chain). The reparametrization trick was also used to estimate the parameter gradient in variational auto-encoders (Kingma and Welling, 2014a; Rezende *et al.*, 2014; Kingma *et al.*, 2014), which are described in Section 20.9.3.

## 13.6 The Deep Learning Approach to Structured Probabilistic Models

Deep learning practictioners generally use the same basic computational tools as other machine learning practitioners who work with structured probabilistic models. However, in the context of deep learning, we usually make different design decisions about how to combine these tools, resulting in overall algorithms and models that have a very different flavor from more traditional graphical models.

The most striking difference between the deep learning style of graphical model design and the traditional style of graphical model design is that the deep learning style heavily emphasizes the use of latent variables. Deep learning models typically have more latent variables than observed variables. Moreover, the practitioner typically does not intend for the latent variables to take on any specific semantics ahead of time— the training algorithm is free to invent the concepts it

needs to model a particular dataset. The latent variables are usually not very easy for a human to interpret after the fact, though visualization techniques may allow some rough characterization of what they represent. Complicated non-linear interactions between variables are accomplished via indirect connections that flow through multiple latent variables. By contrast, traditional graphical models usually contain variables that are at least occasionally observed, even if many of the variables are missing at random from some training examples. Complicated non-linear interactions between variables are modeled by using higher-order terms, with structure learning algorithms used to prune connections and control model capacity. When latent variables are used, they are often designed with some specific semantics in mind—the topic of a document, the intelligence of a student, the disease causing a patient's symptoms, etc. These models are often much more interpretable by human practitioners and often have more theoretical guarantees, yet are less able to scale to complex problems and are not reuseable in as many different contexts as deep models.

Another obvious difference is the kind of graph structure typically used in the deep learning approach. This is tightly linked with the choice of inference algorithm. Traditional approaches to graphical models typically aim to maintain the tractability of exact inference. When this constraint is too limiting, a popular exact inference algorithm is loopy belief propagation. Both of these approaches often work well with very sparsely connected graphs. By comparison, very few interesting deep models admit exact inference, and loopy belief propagation is almost never used for deep learning. Most deep models are designed to make Gibbs sampling or variational inference algorithms, rather than loopy belief propagation, efficient. Another consideration is that deep learning models contain a very large number of latent variables, making efficient numerical code essential. As a result of these design constraints, most deep learning models are organized into regular repeating patterns of units grouped into layers, but neighboring layers may be fully connected to each other. When sparse connections are used, they usually follow a regular pattern, such as the block connections used in convolutional models.

Finally, the deep learning approach to graphical modeling is characterized by a marked tolerance of the unknown. Rather than simplifying the model until all quantities we might want can be computed exactly, we increase the power of the model until it is just barely possible to train or use. We often use models whose marginal distributions cannot be computed, and are satisfied simply to draw approximate samples from these models. We often train models with an intractable objective function that we cannot even approximate in a reasonable amount of time, but we are still able to approximately train the model if we can efficiently obtain an estimate of the gradient of such a function. The deep learning

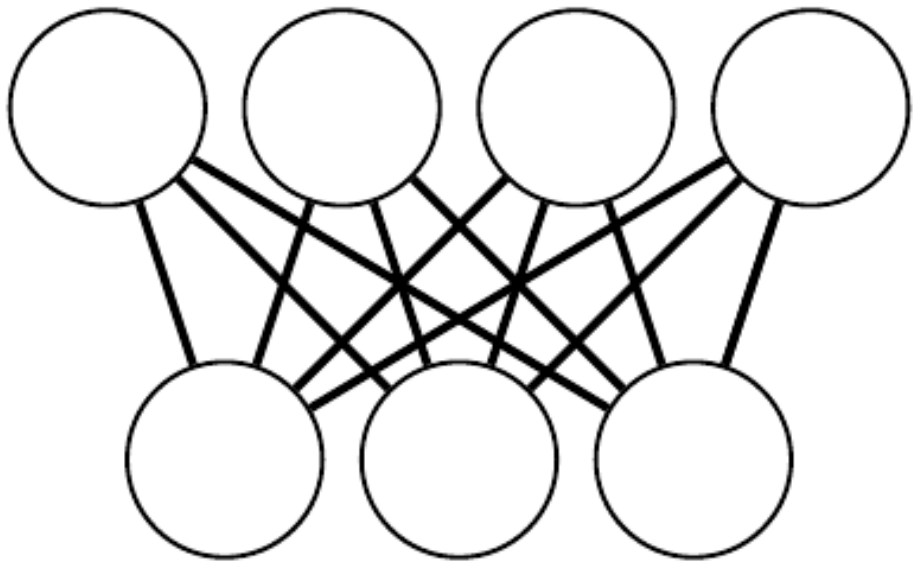

Figure 13.12: An example RBM drawn as a Markov network

approach is often to figure out what the minimum amount of information we absolutely need is, and then to figure out how to get a reasonable approximation of that information as quickly as possible.

### 13.6.1 Example: The Restricted Boltzmann Machine

TODO: rework this section. Add pointer to Chapter 20.2. TODO what do we want to exemplify here?

The *restricted Boltzmann machine* (RBM) (Smolensky, 1986) or *harmonium* is an example of a model that TODO what do we want to exemplify here?

It is an energy-based model with binary visible and hidden units. Its energy function is

$$E(v, h) = -b^\top v - c^\top h - v^\top W h$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. The model is depicted graphically in Fig. 13.12. As this figure makes clear, an important aspect of this model is that there are no direct interactions between any two visible units or between any two hidden units (hence the "restricted," a general Boltzmann machine may have arbitrary connections).

The restrictions on the RBM structure yield the nice properties

$$p(\mathbf{h} \mid \mathbf{v}) = \Pi_i p(\mathrm{h}_i \mid \mathbf{v})$$

and

$$p(\mathbf{v} \mid \mathbf{h}) = \Pi_i p(\mathrm{v}_i \mid \mathbf{h}).$$

The individual conditionals are simple to compute as well, for example

$$p(\mathrm{h}_i = 1 \mid \mathbf{v}) = \sigma \quad \mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i \quad .$$

Together these properties allow for efficient block Gibbs sampling, alternating between sampling all of $\mathbf{h}$ simultaneously and sampling all of $\mathbf{v}$ simultaneously.

Since the energy function itself is just a linear function of the parameters, it is easy to take the needed derivatives. For example,

$$\frac{\partial}{\partial \boldsymbol{W}_{i,j}} \quad , \; E(\mathbf{v}, \mathbf{h}) = -\mathrm{v}_i \mathrm{h}_j .$$

These two properties–efficient Gibbs sampling and efficient derivatives– make it possible to train the RBM with stochastic approximations to $\nabla_\theta \log Z$.

### 13.6.2 The Computational Challenge with High-Dimensional Distributions

TODO: this whole section should probably just be cut, IG thinks YB has written the same thing in 2-3 other places (ml.tex for sure, and maybe also manifolds.tex and prob.tex, possibly others IG hasn't read yet) YB doesn't seem to have read the intro part of this chapter which discusses these things in more detail, double check to make sure there's not anything left out above If this section is kept, it needs cleanup, i.e. a instead $A$, etc. If this section is cut, need to search for refs to it and move them to one of the other versions of it

High-dimensional random variables actually bring two challenges: a statistical challenge and a computational challenge.

The *statistical challenge* was introduced in Section 5.12 and regards generalization: the number of configurations we may want to distinguish can grow exponentially with the number of dimensions of interest, and this quickly becomes much larger than the number of examples one can possibly have (or use with bounded computational resources).

The *computational challenge* associated with high-dimensional distributions arises because many algorithms for learning or using a trained model (especially those based on estimating an explicit probability function) involve intractable computations that grow exponentially with the number of dimensions.

With probabilistic models, this computational challenge arises because of intractable sums (summing over an exponential number of configurations) or intractable maximizations (finding the best out of an intractable number of configurations), discussed mostly in the third part of this book.

- **Intractable inference**: inference is discussed mostly in Chapter 19. It regards the question of guessing the probable values of some variables $A$, given other variables $B$, with respect to a model that captures the joint distribution between $A$, $B$ and $C$. In order to even compute such conditional probabilities one needs to sum over the values of the variables $C$, as well as compute a normalization constant which sums over the values of $A$ and $C$.

- **Intractable normalization constants (the partition function)**: the partition function is discussed mostly in Chapter 18. Normalizing constants of probability functions come up in inference (above) as well as in learning. Many probabilistic models involve such a constant. Unfortunately, the parameters (which we want to tune) influence that constant, and computing the gradient of the partition function with respect to the parameters is generally as intractable as computing the partition function itself. Monte-Carlo Markov chain (MCMC) methods (Chapter 14) are often used to deal with the partition function (computing it or its gradient) but they may also suffer from the curse of dimensionality, when the number of modes of the distribution of interest is very large, and these modes are well separated (Section 14.2).

One way to confront these intractable computations is to approximate them, and many approaches have been proposed, discussed in the chapters listed above. Another interesting way would be to avoid these intractable computations altogether by design, and methods that do not require such computations are thus very appealing. Several generative models based on auto-encoders have been proposed in recent years, with that motivation, and are discussed at the end of Chapter 20.

# Chapter 14

# Monte Carlo Methods

TODO plan organization of chapter (spun off from graphical models chapter)

## 14.1 Markov Chain Monte Carlo Methods

Drawing a sample $x$ from the probability distribution $p(\mathrm{x})$ defined by a structured model is an important operation. The following techniques are described in (Koller and Friedman, 2009).

Sampling from an energy-based model is not straightforward. Suppose we have an EBM defining a distribution $p(\mathrm{a}, \mathrm{b})$. In order to sample a, we must draw it from $p(\mathrm{a} \mid \mathrm{b})$, and in order to sample b, we must draw it from $p(\mathrm{b} \mid \mathrm{a})$. It seems to be an intractable chicken-and-egg problem. Directed models avoid this because their $\mathcal{G}$ is directed and acyclical. In *ancestral sampling* one simply samples each of the variables in topological order, conditioning on each variable's parents, which are guaranteed to have already been sampled. This defines an efficient, single-pass method of obtaining a sample.

In an EBM, it turns out that we can get around this chicken and egg problem by sampling using a *Markov chain*. A Markov chain is defined by a *state* $\mathbf{x}$ and a transition distribution $T(\mathbf{x}' \mid \mathbf{x})$. Running the Markov chain means repeatedly updating the state $\mathbf{x}$ to a value $\mathbf{x}'$ sampled from $T(\mathbf{x}' \mid \mathbf{x})$.

Under certain distributions, a Markov chain is eventually guaranteed to draw $\mathbf{x}$ from an equilibrium distribution $\pi(\mathbf{x}')$, defined by the condition

$$\forall \boldsymbol{x}', \pi(\boldsymbol{x}') = \sum_{\boldsymbol{x}} T(rvx' \mid \boldsymbol{x}) \pi(\boldsymbol{x}).$$

TODO– this vector / matrix view needs a whole lot more exposition only literally a vector / matrix when the state is discrete unpack into multiple sentences,

the parenthetical is hard to parse is the term "stochastic matrix" defined anywhere? make sure it's in the index at least whoever finishes writing this section should also finish making the math notation consistent terms in this section need to be in the index

We can think of $\pi$ as a vector (with the probability for each possible value $\boldsymbol{x}$ in the element indexed by $x$, $\pi(x)$) and $T$ as a corresponding stochastic matrix (with row index $x'$ and column index $x$), i.e., with non-negative entries that sum to 1 over elements of a column. Then, the above equation becomes

$$T\pi = \pi$$

an eigenvector equation that says that $\pi$ is the eigenvector of $T$ with eigenvalue 1. It can be shown (Perron-Frobenius theorem) that this is the largest possible eigenvalue, and the only one with value 1 under mild conditions (for example $T(x' \mid x) > 0$). We can also see this equation as a fixed point equation for the update of the distribution associated with each step of the Markov chain. If we start a chain by picking $\mathrm{x}_0 \sim p_0$, then we get a distribution $p_1 = Tp_0$ after one step, and $p_t = Tp_{t-1} = T^t p_0$ after $t$ steps. If this recursion converges (the chain has a so-called *stationary distribution*), then it converges to a fixed point which is precisely $p_t = \pi$ for $t \to \infty$, and the dynamical systems view meets and agrees with the eigenvector view.

This condition guarantees that repeated applications of the transition sampling procedure don't change the *distribution* over the state of the Markov chain. Running the Markov chain until it reaches its equilibrium distribution is called "burning in" the Markov chain.

Unfortunately, there is no theory to predict how many steps the Markov chain must run before reaching its equilibrium distribution[1], nor any way to tell for sure that this event has happened. Also, even though successive samples come from the same distribution, they are highly correlated with each other, so to obtain multiple samples one should run the Markov chain for many steps between collecting each sample. Markov chains tend to get stuck in a single mode of $\pi(x)$ for several steps. The speed with which a Markov chain moves from mode to mode is called its mixing rate. Since burning in a Markov chain and getting it to mix well may take several sampling steps, sampling correctly from an EBM is still a somewhat costly procedure.

TODO: mention Metropolis-Hastings

Of course, all of this depends on ensuring $\pi(x) = p(x)$ . Fortunately, this is easy so long as $p(x)$ is defined by an EBM. The simplest method is to use *Gibbs sampling*, in which sampling from $T(\mathbf{x}' \mid \mathbf{x})$ is accomplished by selecting

[1] although in principle the ratio of the two leading eigenvalues of the transition operator gives us some clue, and the largest eigenvalue is 1.

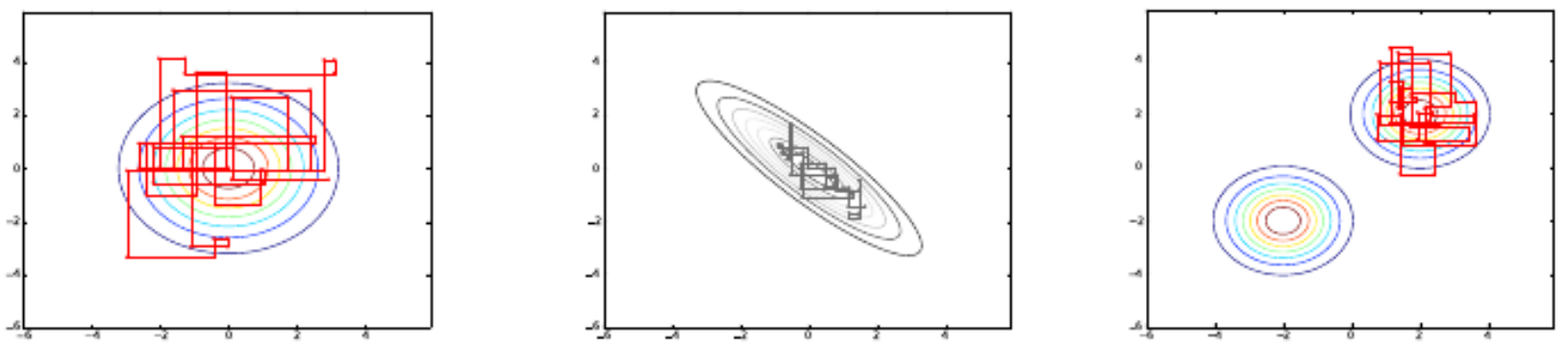

Figure 14.1: Paths followed by Gibbs sampling for three distributions, with the Markov chain initialized at the mode in both cases. Left) A multivariate normal distribution with two independent variables. Gibbs sampling *mixes* well because the variables are independent. Center) A multivariate normal distribution with highly correlated variables. The correlation between variables makes it difficult for the Markov chain to mix. Because each variable must be updated conditioned on the other, the correlation reduces the rate at which the Markov chain can move away from the starting point. Right) A mixture of Gaussians with widely separated modes that are not axis-aligned. Gibbs sampling mixes very slowly because it is difficult to change modes while altering only one variable at a time.

one variable $\mathrm{x}_i$ and sampling it from $p$ conditioned on its neighbors in $\mathcal{G}$. It is also possible to sample several variables at the same time so long as they are conditionally independent given all of their neighbors.

TODO: discussion of mixing example with 2 binary variables that prefer to both have the same state IG's graphic from lecture on adversarial nets

TODO: refer to this figure in the text:

TODO: refer to this figure in the text

### 14.1.1 Markov Chain Theory

TODO

State Perron's theorem

DEFINE detailed balance

### 14.1.2 Importance Sampling

TODO write this section

## 14.2 The Difficulty of Mixing between Well-Separated Modes

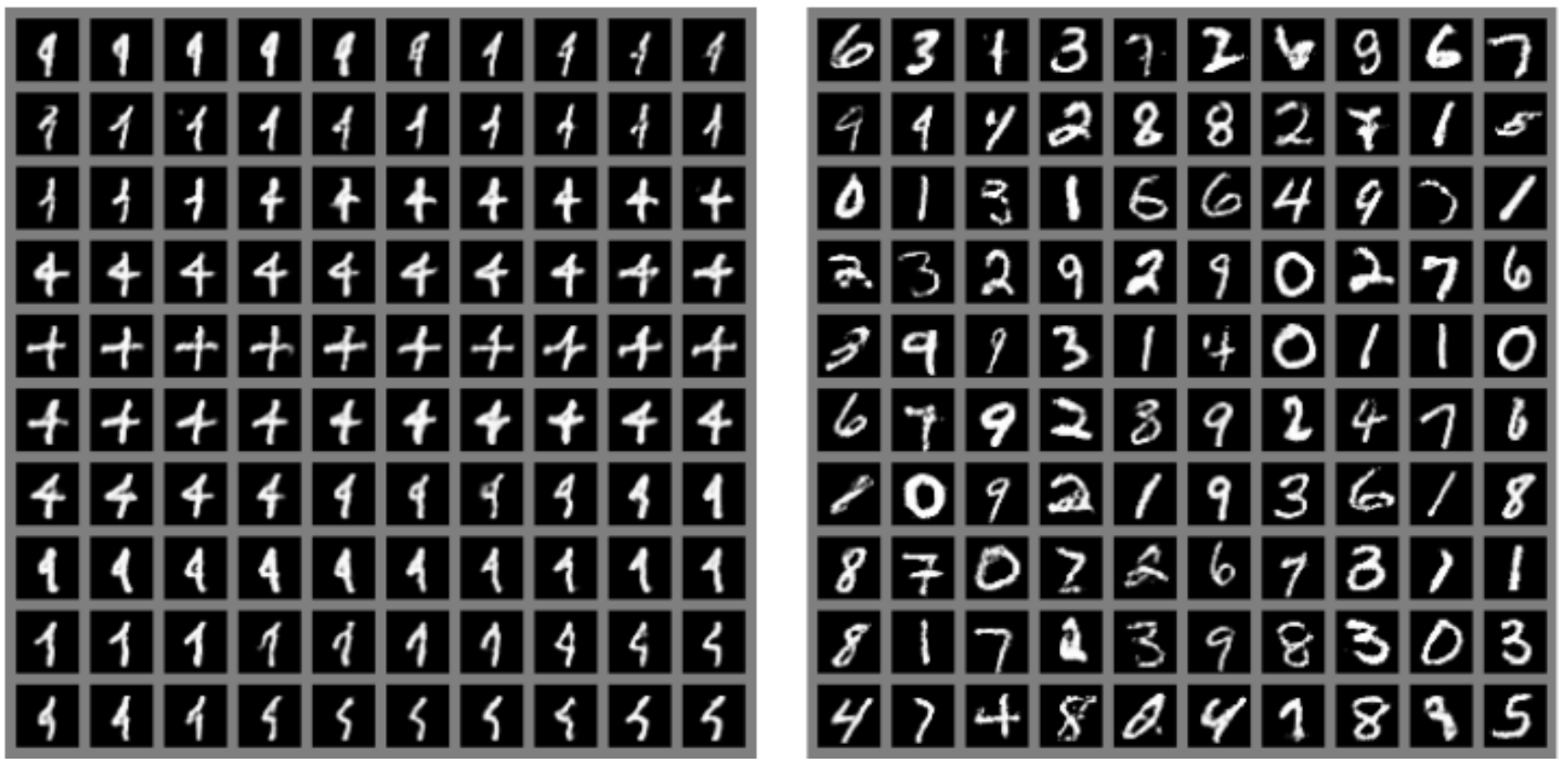

Figure 14.2: An illustration of the slow mixing problem in deep probabilistic models. Each panel should be read left to right, top to bottom. Left) Consecutive samples from Gibbs sampling applied to a deep Boltzmann machine trained on the MNIST dataset. Consecutive samples are similar to each other. Because the Gibbs sampling is performed in a deep graphical model, this similarity is based more on semantic rather than raw visual features, but it is still difficult for the Gibbs chain to transition from one mode of the distribution to another, for example by changing the digit identity. Right) Consecutive ancestral samples from a generative adversarial network. Because ancestral sampling generates each sample independently from the others, there is no mixing problem.

# Chapter 15

# Linear Factor Models and Auto-Encoders

Linear factor models are generative unsupervised learning models in which we imagine that some unobserved factors $\boldsymbol{h}$ explain the observed variables $\boldsymbol{x}$ through a linear transformation. Auto-encoders are unsupervised learning methods that learn a representation of the data, typically obtained by a non-linear parametric transformation of the data, i.e., from $\boldsymbol{x}$ to $\boldsymbol{h}$, typically a feedforward neural network, but not necessarily. They also learn a transformation going backwards from the representation to the data, from $\boldsymbol{h}$ to $\boldsymbol{x}$, like the linear factor models. Linear factor models therefore only specify a parametric decoder, whereas auto-encoder also specify a parametric encoder. Some linear factor models, like PCA, actually correspond to an auto-encoder (a linear one), but for others the encoder is implicitly defined via an inference mechanism that searches for an $\boldsymbol{h}$ that could have generated the observed $\boldsymbol{x}$.

The idea of auto-encoders has been part of the historical landscape of neural networks for decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994) but has really picked up speed in recent years. They remained somewhat marginal for many years, in part due to what was an incomplete understanding of the mathematical interpretation and geometrical underpinnings of auto-encoders, which are developed further in Chapters 17 and 20.12.

**An auto-encoder is simply a neural network that tries to copy its input to its output.** The architecture of an auto-encoder is typically decomposed into the following parts, illustrated in Figure 15.1:

- an input, $\boldsymbol{x}$
- an encoder function $f$
- a "*code*" or internal representation $\boldsymbol{h} = f(\boldsymbol{x})$

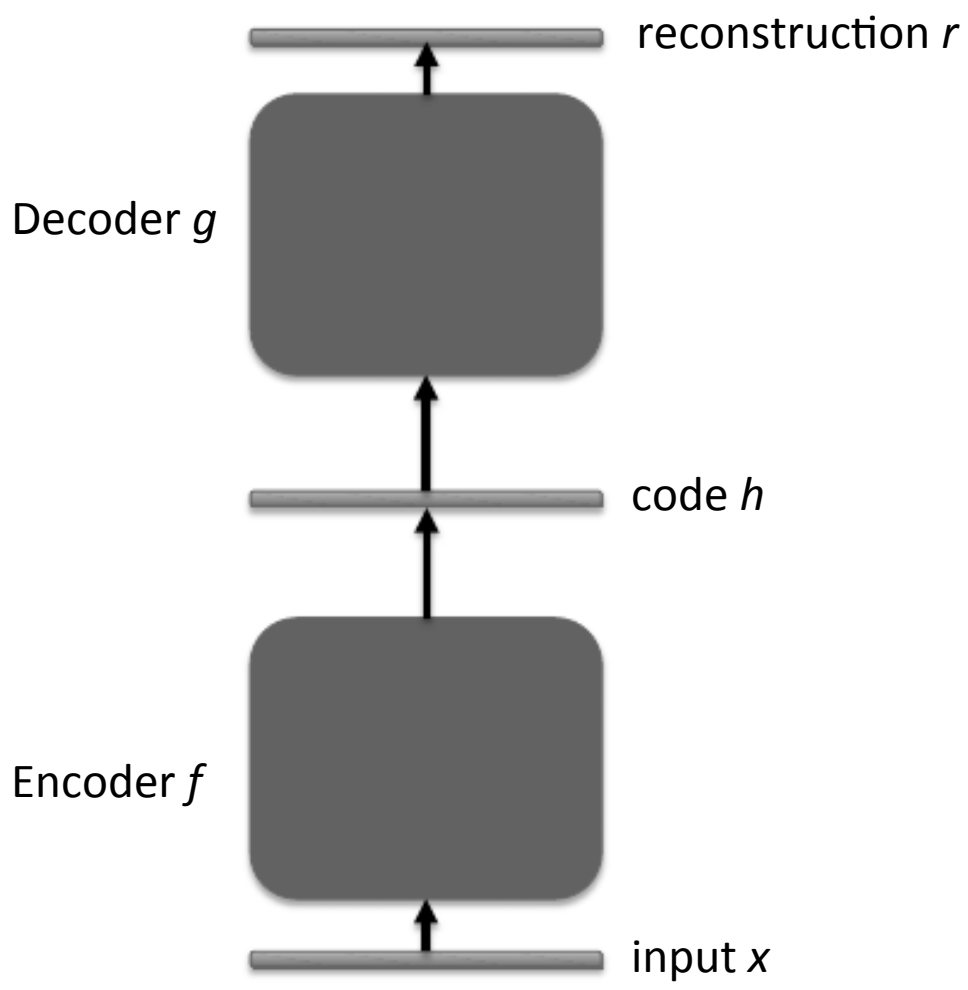


Figure 15.1: General schema of an auto-encoder, mapping an input $\boldsymbol{x}$ to an output (called reconstruction) $\boldsymbol{r}$ through an internal representation or code $\boldsymbol{h}$. The auto-encoder has two components: the encoder $f$ (mapping $\boldsymbol{x}$ to $\boldsymbol{h}$) and the decoder $g$ (mapping $\boldsymbol{h}$ to $\boldsymbol{r}$).

- a decoder function $g$
- an output, also called "*reconstruction*" $\boldsymbol{r} = g(\boldsymbol{h}) = g(f(\boldsymbol{x}))$
- a loss function $L$ computing a scalar $L(\boldsymbol{r}, \boldsymbol{x})$ measuring how good of a reconstruction $\boldsymbol{r}$ is of the given input $\boldsymbol{x}$. The objective is to minimize the expected value of $L$ over the training set of examples $\{\boldsymbol{x}\}$.

## 15.1 Regularized Auto-Encoders

Predicting the input may sound useless: what could prevent the auto-encoder from simply copying its input into its output? In the 20th century, this was achieved by constraining the architecture of the auto-encoder to avoid this, by forcing the dimension of the code $\boldsymbol{h}$ to be smaller than the dimension of the input $\boldsymbol{x}$.

Figure 15.2 illustrates the two typical cases of auto-encoders: undercomplete (with the dimension of the representation $\boldsymbol{h}$ smaller than the dimension of the input $\boldsymbol{x}$), andovercomplete (with the dimension of $\boldsymbol{h}$ larger than that of $\boldsymbol{x}$). Whereas early work with auto-encoders, just like PCA, uses an undercomplete bottleneck in the sequence of layers to avoid learning the identity function, more recent work

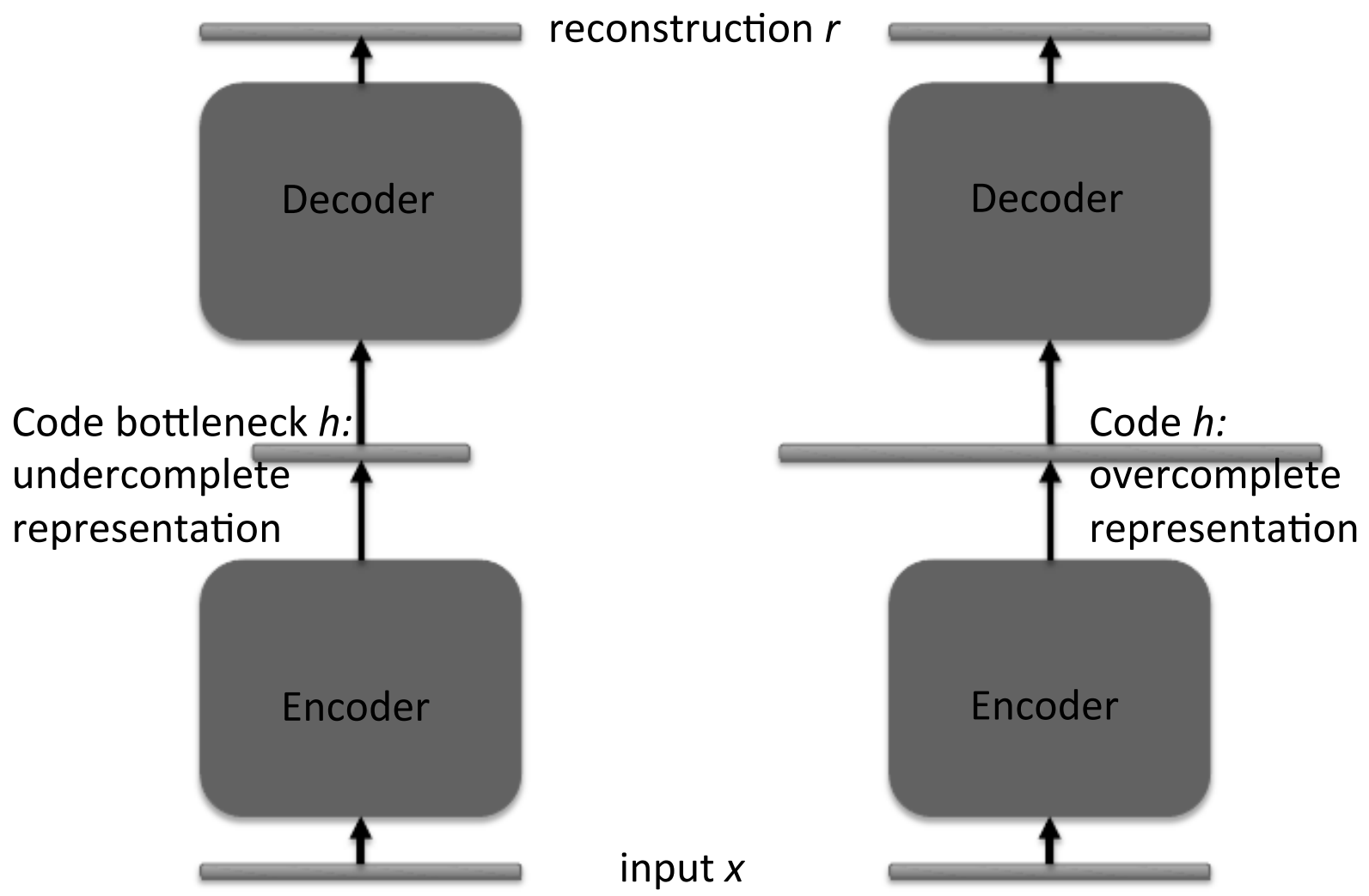


Figure 15.2: Left: undercomplete representation (dimension of code $\boldsymbol{h}$ is less than dimension of input $\boldsymbol{x}$). Right: overcomplete representation. Overcomplete auto-encoders require some other form of regularization (instead of the constraint on the dimension of $\boldsymbol{h}$) to avoid the trivial solution where $\boldsymbol{r} = \boldsymbol{x}$ for all $\boldsymbol{x}$.

allows overcomplete representations. What we have learned in recent years is that it is possible to make the auto-encoder meaningfully capture the structure of the input distribution even if the representation is overcomplete, with other forms of constraint or regularization. In fact, once you realize that auto-encoders can capture the input distribution (indirectly, not as a an explicit probability function), you also realize that it should need more capacity as one increases the complexity of the distribution to be captured (and the amount of data available): it should not be limited by the input dimension. This is a problem in particular with the shallow auto-encoders, which have a single hidden layer (for the code). Indeed, that hidden layer size controls both the dimensionality reduction constraint (the code size at the bottleneck) and the capacity (which allows to learn a more complex distribution).

Besides the **bottleneck** constraint, alternative constraints or regularization methods have been explored and can guarantee that the auto-encoder does something useful and not just learn some trivial identity-like function:

- **Sparsity of the representation or of its derivative**: even if the intermediate representation has a very high dimensionality, the effective local dimensionality (number of degrees of freedom that capture a coordinate sys-

tem among the probable $x$'s) could be much smaller if most of the elements of $h$ are zero (or any other constant, such that $||\frac{\partial h_i}{\partial \boldsymbol{x}}||$ is close to zero). When $||\frac{\partial h_i}{\partial \boldsymbol{x}}||$ is close to zero, $h_i$ does not participate in encoding local changes in $\boldsymbol{x}$. There is a geometrical interpretation of this situation in terms of *manifold learning* that is discussed in more depth in Chapter 17. The discussion in Chapter 16 also explains how an auto-encoder naturally tends towards learning a coordinate system for the actual factors of variation in the data. At least four types of "auto-encoders" clearly fall in this category of sparse representation:

- **Sparse coding** (Olshausen and Field, 1996) has been heavily studied as an unsupervised feature learning and feature inference mechanism. It is a linear factor model rather than an auto-encoder, because it has no explicit parametric encoder, and instead uses an iterative optimization procedure to compute the maximally likely code. Sparse coding looks for representations that are both sparse and explain the input through the decoder. Instead of the code being a parametric function of the input, it is considered like free variable that is obtained through an optimization, i.e., a particular form of inference:

$$\boldsymbol{h}^* = f(\boldsymbol{x}) = \arg\min_{\boldsymbol{h}} L(g(\boldsymbol{h}), \boldsymbol{x})) + \lambda \Omega(\boldsymbol{h}) \tag{15.1}$$

  where $L$ is the reconstruction loss, $f$ the (non-parametric) encoder, $g$ the (parametric) decoder, $\Omega(\boldsymbol{h})$ is a sparsity regularizer, and in practice the minimization can be approximate. Sparse coding has a manifold or geometric interpretation that is discussed in Section 15.8. It also has an interpretation as a directed graphical model, described in more details in Section 19.3. To achieve sparsity, the objective function to optimize includes a term that is minimized when the representation has many zero or near-zero values, such as the L1 penalty $|\boldsymbol{h}|_1 = \sum_i |h_i|$.
- An interesting variation of sparse coding combines the freedom to choose the representation through optimization and a parametric encoder. It is called **predictive sparse decomposition** (PSD) (Kavukcuoglu *et al.*, 2008a) and is briefly described in Section 15.8.2.
- At the other end of the spectrum are simply **sparse auto-encoders**, which combine with the standard auto-encoder schema a sparsity penalty which encourages the output of the encoder to be sparse. These are described in Section 15.8.1. Besides the L1 penalty, other sparsity penalties that have been explored include the Student-t penalty (Olshausen and Field, 1996; Bergstra, 2011), TODO: should the t be in

math mode, perhaps?

$$\sum_i \log(1 + \alpha^2 h_i^2)$$

(i.e. where $\alpha h_i$ has a Student-t prior density) and the KL-divergence penalty (Lee *et al.*, 2008; Goodfellow *et al.*, 2009; Larochelle and Bengio, 2008a)

$$-\sum_i (t \log h_i + (1-t)\log(1-h_i)),$$

with a target sparsity level $t$, for $h_i \in (0,1)$, e.g. through a sigmoid non-linearity.

– **Contractive autoencoders** (Rifai *et al.*, 2011b), covered in Section 15.10, explicitly penalize $||\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}||_F^2$, i.e., the sum of the squared norm of the vectors $\frac{\partial h_i(\boldsymbol{x})}{\partial \boldsymbol{x}}$ (each indicating how much each hidden unit $h_i$ responds to changes in $\boldsymbol{x}$ and what direction of change in $\boldsymbol{x}$ that unit is most sensitive to, around a particular $\boldsymbol{x}$). With such a regularization penalty, the auto-encoder is called **contractive**[1] because the mapping from input $\boldsymbol{x}$ to representation $\boldsymbol{h}$ is encouraged to be contractive, i.e., to have small derivatives in all directions. Note that a sparsity regularization indirectly leads to a contractive mapping as well, when the non-linearity used happens to have a zero derivative at $h_i = 0$ (which is the case for the sigmoid non-linearity).

- **Robustness to injected noise or missing information**: if noise is injected in inputs or hidden units, or if some inputs are missing, while the neural network is asked to *reconstruct the clean and complete input*, then it cannot simply learn the identity function. It has to capture the structure of the data distribution in order to optimally perform this reconstruction. Such auto-encoders are called *denoising auto-encoders* and are discussed in more detail in Section 15.9.

## 15.2 Denoising Auto-encoders

There is a tight connection between the denoising auto-encoders and the contractive auto-encoders: it can be shown (Alain and Bengio, 2013) that in the limit of small Gaussian injected input noise, the denoising reconstruction error is equivalent to a contractive penalty on the reconstruction function that maps $\boldsymbol{x}$ to $\boldsymbol{r} = g(f(\boldsymbol{x}))$. In other words, since both $\boldsymbol{x}$ and

[1] A function ( ) is contractive if ( ) ( ) for nearby and , or equivalently if its derivative ( ) 1.

$\boldsymbol{x}+\boldsymbol{\epsilon}$ (where $\boldsymbol{\epsilon}$ is some small noise vector) must yield the same target output $\boldsymbol{x}$, the reconstruction function is encouraged to be insensitive to changes in all directions $\epsilon$. The only thing that prevents reconstruction $\boldsymbol{r}$ from simply being a constant (completely insensitive to the input $\boldsymbol{x}$), is that one also has to reconstruct correctly for different training examples $\boldsymbol{x}$. However, the auto-encoder can learn to be approximately constant around training examples $\boldsymbol{x}$ while producing a different answer for different training examples. As discussed in Section 17.4, if the examples are near a low-dimensional manifold, this encourages the representation to vary only on the manifold and be locally constant in directions orthogonal to the manifold, i.e., the representation locally captures a (not necessarily Euclidean, not necessarily orthogonal) coordinate system for the manifold. In addition to the denoising auto-encoder, the *variational auto-encoder* (Section 20.9.3) and the *generative stochastic networks* (Section 20.12) also involve the injection of noise, but typically in the representation-space itself, thus introducing the notion of $\boldsymbol{h}$ as a *latent variable*.

- **Pressure of a Prior on the Representation:** an interesting way to generalize the notion of regularization applied to the representation is to introduce in the cost function for the auto-encoder a log-prior term

$$-\log P(\boldsymbol{h})$$

which captures the assumption that we would like to find a representation that has a simple distribution (if $P(\boldsymbol{h})$ has a simple form, such as a factorized distribution[2]), or at least one that is simpler than the original data distribution. Among all the encoding functions $f$, we would like to pick one that

1. can be inverted (easily), and this is achieved by minimizing some reconstruction loss, and
2. yields representations $\boldsymbol{h}$ whose distribution is "simpler", i.e., can be captured with less capacity than the original training distribution itself.

The sparse variants described above clearly fall in that framework. The variational auto-encoder (Section 20.9.3) provides a clean mathematical framework for justifying the above pressure of a top-level prior when the objective is to model the data generating distribution.

From the point of view of regularization (Chapter 7), adding the $-\log P(\boldsymbol{h})$ term to the objective function (e.g. for encouraging sparsity) or adding a contractive penalty do not fit the traditional view of a prior on the parameters. Instead,

all the sparse priors we have described correspond to a factorized distribution

the prior on the latent variables acts like a *data-dependent prior*, in the sense that it depends on the particular values $\boldsymbol{h}$ that are going to be sampled (usually from a posterior or an encoder), based on the input example $\boldsymbol{x}$. Of course, indirectly, this is also a regularization on the parameters, but one that depends on the particular data distribution.

## 15.3 Representational Power, Layer Size and Depth

Autoencoders are often trained with only a single layer encoder and a single layer decoder. However, this is not a requirement, and using deep encoders and decoders offers many advantages.

Recall from Sec. 6.6 that there are many advantages to depth in a feed-forward network. Because auto-encoders are feed-forward networks, these advantages also apply to auto-encoders. Moreover, the encoder is itself a feed-forward network as is the decoder, so each of these components of the auto-encoder can individually benefit from depth.

One major advantage of non-trivial depth is that the universal approximator theorem guarantees that a feedforward neural network with at least one hidden layer can represent an approximation of any function (within a broad class) to an arbitrary degree of accuracy, provided that it has enough hidden units. This means that an autoencoder with a single hidden layer is able to represent the identity function along the domain of the data arbitrarily well. However, the mapping from input to code is shallow. This means that we are not able to enforce arbitrary constraints, such as that the code should be sparse. A deep autoencoder, with at least one additional hidden layer inside the encoder itself, can approximate any mapping from input to code arbitrarily well, given enough hidden units.

The above viewpoint also motivates overcomplete autoencoders, that is, autoencoders with very wide layers, in order to achieve a rich family of possible functions.

Depth can exponentially reduce the computational cost of evaluating a representation of some functions, and can also exponentially decrease the amount of training data needed to learn some functions.

Experimentally, deep auto-encoders yield much better compression than corresponding shallow or linear auto-encoders (Hinton and Salakhutdinov, 2006).

A common strategy for training a deep autoencoder is to greedily pre-train the deep architecture by training a stack of shallow auto-encoders, so we often encounter shallow auto-encoders, even when the ultimate goal is to train a deep auto-encoder.

## 15.4 Reconstruction Distribution

The above "parts" (encoder function $f$, decoder function $g$, reconstruction loss $L$) make sense when the loss $L$ is simply the squared reconstruction error, but there are many cases where this is not appropriate, e.g., when $\boldsymbol{x}$ is a vector of discrete variables or when $P(\boldsymbol{x} \mid \boldsymbol{h})$ is not well approximated by a Gaussian distribution [3]. Just like in the case of other types of neural networks (starting with the feedforward neural networks, Section 6.3.2), it is convenient to define the loss $L$ as a negative log-likelihood over some target random variables. This probabilistic interpretation is particularly important for the discussion in Sections 20.9.3, 20.11 and 20.12 about generative extensions of auto-encoders and stochastic recurrent networks, where the output of the auto-encoder is interpreted as a probability distribution $P(\boldsymbol{x} \mid \boldsymbol{h})$, for reconstructing $\boldsymbol{x}$, given hidden units $\boldsymbol{h}$. This distribution captures not just the expected reconstruction but also the *uncertainty* about the original $\boldsymbol{x}$ (which gave rise to $\boldsymbol{h}$, either deterministically or stochastically, given $\boldsymbol{h}$). In the simplest and most ordinary cases, this distribution factorizes, i.e., $P(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_i P(x_i \mid \boldsymbol{h})$. This covers the usual cases of $x_i \mid \boldsymbol{h}$ being Gaussian (for unbounded real values) and $x_i | \boldsymbol{h}$ having a Bernoulli distribution (for binary values $x_i$), but one can readily generalize this to other distributions, such as mixtures (see Sections 3.10.6 and 6.3.2).

Thus we can generalize the notion of *decoding function* $g(\boldsymbol{h})$ to *decoding distribution* $P(\boldsymbol{x} \mid \boldsymbol{h})$. Similarly, we can generalize the notion of *encoding function* $f(\boldsymbol{x})$ to *encoding distribution* $Q(\boldsymbol{h} \mid \boldsymbol{x})$, as illustrated in Figure 15.3. We use this to capture the fact that noise is injected at the level of the representation $\boldsymbol{h}$, now considered like a latent variable. This generalization is crucial in the development of the variational auto-encoder (Section 20.9.3) and the generalized stochastic networks (Section 20.12).

We also find a stochastic encoder and a stochastic decoder in the RBM, described in Section 20.2. In that case, the encoding distribution $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$ "match", in the sense that $Q(\boldsymbol{h} \mid \boldsymbol{x}) = P(\boldsymbol{h} \mid \boldsymbol{x})$, i.e., there is a unique joint distribution which has both $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$ as conditionals. This is not true in general for two independently parametrized conditionals like $Q(\boldsymbol{h} \mid \boldsymbol{x})$ and $P(\boldsymbol{x} \mid \boldsymbol{h})$, although the work on generative stochastic networks (Alain *et al.*, 2015) shows that learning will tend to make them compatible asymptotically (with enough capacity and examples).

---

See the link between squared error and normal density in Sections 5.6 and 6.3.2

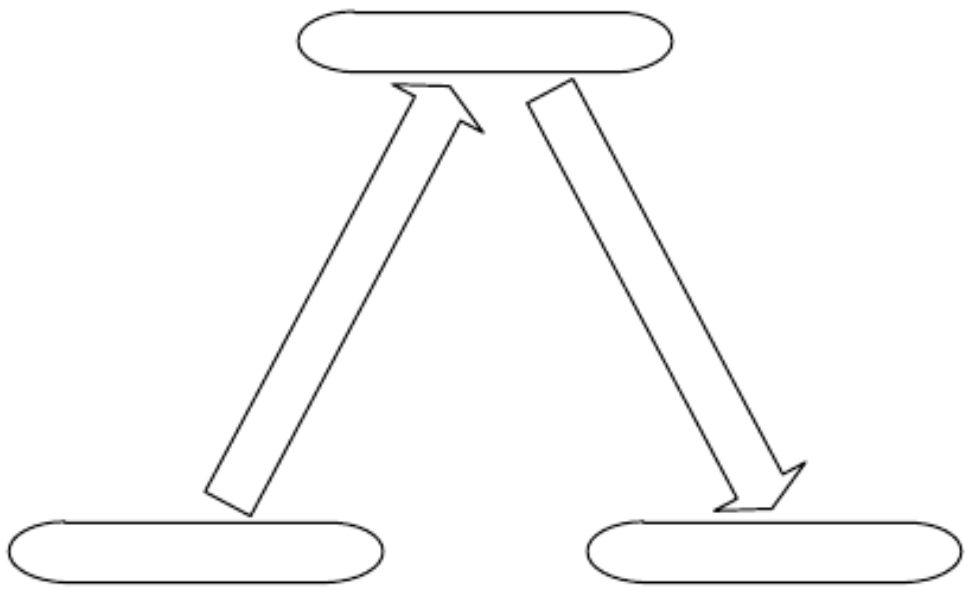

Figure 15.3: Basic scheme of a stochastic auto-encoder, in which both the encoder and the decoder are not simple functions but instead involve some noise injection, meaning that their output can be seen as sampled from a distribution, $Q(\boldsymbol{h} \mid \boldsymbol{x})$ for the encoder and $P(\boldsymbol{x} \mid \boldsymbol{h})$ for the decoder. RBMs are a special case where $P = Q$ (in the sense of a unique joint corresponding to both conditinals) but in general these two distributions are not necessarily conditional distributions compatible with a unique joint distribution $P(\boldsymbol{x}, \boldsymbol{h})$.

## 15.5 Linear Factor Models

Now that we have introduced the notion of a probabilistic decoder, let us focus on a very special case where the latent variable $\boldsymbol{h}$ generates $\boldsymbol{x}$ via a linear transformation plus noise, i.e., classical linear factor models, which do not necessarily have a corresponding parametric encoder.

The idea of discovering explanatory factors that have a simple joint distribution among themselves is old, e.g., see Factor Analysis (see below), and has been explored first in the context where the relationship between factors and data is linear, i.e., we assume that the data was generated as follows. First, sample the real-valued factors,

$$\boldsymbol{h} \sim P(\boldsymbol{h}), \tag{15.2}$$

and then sample the real-valued observable variables given the factors:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{15.3}$$

where the noise is typically Gaussian and diagonal (independent across dimensions). This is illustrated in Figure 15.4.

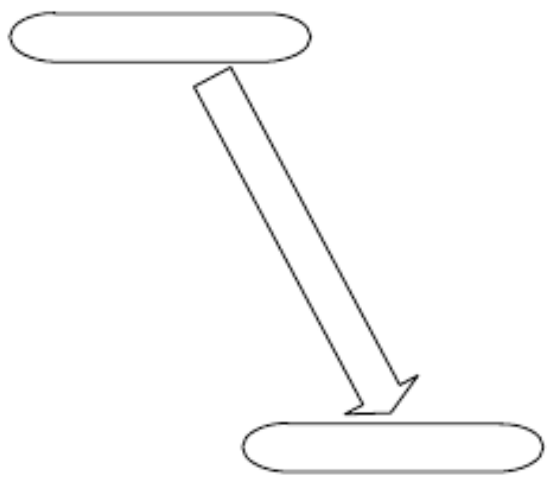

Figure 15.4: Basic scheme of a linear factors model, in which we assume that an observed data vector $\boldsymbol{x}$ is obtained by a linear combination of latent factors $\boldsymbol{h}$, plus some noise. Different models, such as probabilistic PCA, factor analysis or ICA, make different choices about the form of the noise and of the prior $P(\boldsymbol{h})$.

## 15.6 Probabilistic PCA and Factor Analysis

Probabilistic PCA (Principal Components Analysis), factor analysis and other linear factor models are special cases of the above equations (15.2 and 15.3) and only differ in the choices made for the prior (over latent, not parameters) and noise distributions.

In factor analysis (Bartholomew, 1987; Basilevsky, 1994), the latent variable prior is just the unit variance Gaussian

$$\boldsymbol{h} \sim \mathcal{N}(0, \boldsymbol{I})$$

while the observed variables $x_i$ are assumed to be *conditionally independent*, given $\boldsymbol{h}$, i.e., the noise is assumed to be coming from a diagonal covariance Gaussian distribution, with covariance matrix $\boldsymbol{\psi} = \mathrm{diag}(\boldsymbol{\sigma}^2)$, with $\boldsymbol{\sigma}^2 = (\sigma_1^2, \sigma_2^2, \ldots)$ a vector of per-variable variances.

The role of the latent variables is thus to *capture the dependencies* between the different observed variables $x_i$. Indeed, it can easily be shown that $\boldsymbol{x}$ is just a Gaussian-distribution (multivariate normal) random variable, with

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \boldsymbol{\psi})$$

where we see that the weights $\boldsymbol{W}$ induce a dependency between two variables $x_i$ and $x_j$ through a kind of auto-encoder path, whereby $x_i$ influences $\hat{\boldsymbol{h}}_k = \boldsymbol{W}_k\boldsymbol{x}$ via $w_{ki}$ (for every $k$) and $\hat{\boldsymbol{h}}_k$ influences $x_j$ via $w_{kj}$.

In order to cast PCA in a probabilistic framework, we can make a slight modification to the factor analysis model, making the conditional variances $\sigma_i$

equal to each other. In that case the covariance of $\boldsymbol{x}$ is just $\boldsymbol{W}\boldsymbol{W}^\top + \sigma^2 \boldsymbol{I}$, where $\sigma^2$ is now a scalar, i.e.,

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^\top + \sigma^2 \boldsymbol{I})$$

or equivalently

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \sigma \boldsymbol{z}$$

where $\boldsymbol{z} \sim \mathcal{N}(0, \boldsymbol{I})$ is white noise. Tipping and Bishop (1999) then show an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$ and $\sigma^2$.

What the probabilistic PCA model is basically saying is that the covariance is mostly captured by the latent variables $\boldsymbol{h}$, up to some small residual *reconstruction error* $\sigma^2$. As shown by Tipping and Bishop (1999), probabilistic PCA becomes PCA as $\sigma \to 0$. In that case, the conditional expected value of $\boldsymbol{h}$ given $\boldsymbol{x}$ becomes an orthogonal projection onto the space spanned by the $d$ columns of $\boldsymbol{W}$, like in PCA. See Section 17.1 for a discussion of the "inference" mechanism associated with PCA (probabilistic or not), i.e., recovering the expected value of the latent factors $h_i$ given the observed input $\boldsymbol{x}$. That section also explains the very insightful *geometric and manifold interpretation* of PCA.

However, as $\sigma \to 0$, the density model becomes very sharp around these $d$ dimensions spanned the columns of $\boldsymbol{W}$, as discussed in Section 17.1, which would not make it a very faithful model of the data, in general (not just because the data may live on a higher-dimensional manifold, but more importantly because the real data manifold may not be a flat hyperplane - see Chapter 17 for more).

### 15.6.1 ICA

Independent Component Analysis (ICA) is among the oldest representation learning algorithms (Herault and Ans, 1984; Jutten and Herault, 1991; Comon, 1994; Hyvärinen, 1999; Hyvärinen *et al.*, 2001). It is an approach to modeling linear factors that seeks non-Gaussian projections of the data. Like probabilistic PCA and factor analysis, it also fits the linear factor model of Eqs. 15.2 and 15.3. What is particular about ICA is that unlike PCA and factor analysis it *does not assume that the latent variable prior is Gaussian.* It only assumes that it is *factorized*, i.e.,

$$P(\boldsymbol{h}) = \prod_i P(h_i). \tag{15.4}$$

Since there is no parametric assumption behind the prior, we are really in front of a so-called *semi-parametric model*, with parts of the model being parametric ($P(\boldsymbol{x} \mid \boldsymbol{h})$) and parts being non-specified or non-parametric ($P(\boldsymbol{h})$). In fact, this typically yields to *non-Gaussian* priors: if the priors were Gaussian, then one could not distinguish between the factors $\boldsymbol{h}$ and a rotation of $\boldsymbol{h}$. Indeed, note

that if

$$\boldsymbol{h} = \boldsymbol{U}\boldsymbol{z}$$

with $\boldsymbol{U}$ an orthonormal (rotation) square matrix, i.e.,

$$\boldsymbol{z} = \boldsymbol{U}^\top \boldsymbol{h},$$

then, although $\boldsymbol{h}$ might have a Normal$(0, \boldsymbol{I})$ distribution, the $\boldsymbol{z}$ *also have a unit covariance*, i.e., they are uncorrelated:

$$Var[\boldsymbol{z}] = \ [\boldsymbol{z}\boldsymbol{z}^\top] = \ [\boldsymbol{U}^\top \boldsymbol{h}\boldsymbol{h}^\top \boldsymbol{U}] = \boldsymbol{U}^\top Var[\boldsymbol{h}]\boldsymbol{U} = \boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}.$$

In other words, imposing independence among Gaussian factors does not allow one to disentangle them, and we could as well recover any linear rotation of these factors. It means that, given the observed $\boldsymbol{x}$, even though we might assume the right generative model, PCA cannot recover the original generative factors. However, if we assume that the latent variables are *non-Gaussian*, then we can recover them, and this is what ICA is trying to achieve. In fact, under these generative model assumptions, the true underlying factors can be recovered (Comon, 1994). In fact, many ICA algorithms are looking for projections of the data $\boldsymbol{s} = \boldsymbol{V}\boldsymbol{x}$ such that they are *maximally non-Gaussian*. An intuitive explanation for these approaches is that although the true latent variables $\boldsymbol{h}$ may be non-Gaussian, almost any linear combination of them will look more Gaussian, because of the central limit theorem. Since linear combinations of the $x_i$'s are also linear combinations of the $h_j$'s, to recover the $h_j$'s we just need to find the linear combinations that are maximally non-Gaussian (while keeping these different projections orthogonal to each other).

There is an interesting connection between ICA and sparsity, since the dominant form of non-Gaussianity in real data is due to sparsity, i.e., concentration of probability at or near 0. Non-Gaussian distributions typically have more mass around zero, although you can also get non-Gaussianity by increasing skewness, asymmetry, or kurtosis.

Like PCA can be generalized to non-linear auto-encoders described later in this chapter, ICA can be generalized to a non-linear generative model, e.g., $\boldsymbol{x} = f(\boldsymbol{h}) +$noise. See Hyvärinen and Pajunen (1999) for the initial work on non-linear ICA and its successful use with ensemble learning by Roberts and Everson (2001); Lappalainen *et al.* (2000).

### 15.6.2 Sparse Coding as a Generative Model

One particularly interesting form of non-Gaussianity arises with distributions that are sparse. These typically have not just a peak at 0 but also a fat tail[4]. Like the

with probability going to 0 as the values increase in magnitude at a rate that is slower than the Gaussian, i.e., less than quadratic in the log-domain.

other linear factor models (Eq. 15.3), sparse coding corresponds to a linear factor model, but one with a "sparse" latent variable $\boldsymbol{h}$, i.e., $P(\boldsymbol{h})$ puts high probability at or around 0. Unlike with ICA (previous section), the latent variable prior is parametric. For example the factorized Laplace density prior is

$$P(\boldsymbol{h}) = \prod_i P(h_i) = \prod_i \frac{\lambda}{2} e^{-\lambda|h_i|} \tag{15.5}$$

and the factorized Student-t prior is

$$P(\boldsymbol{h}) = \prod_i P(h_i) \propto \prod_i \frac{1}{1 + \frac{h_i}{\nu}}^{\nu} . \tag{15.6}$$

Both of these densities have a strong preference for near-zero values but, unlike the Gaussian, accomodate large values. In the standard sparse coding models, the reconstruction noise is assumed to be Gaussian, so that the corresponding reconstruction error is the squared error.

Regarding sparsity, note that the actual value $h_i = 0$ has zero measure under both densities, meaning that the posterior distribution $P(\boldsymbol{h} \mid \boldsymbol{x})$ will not generate values $\boldsymbol{h} = 0$. However, sparse coding is normally considered under a maximum a posteriori (MAP) inference framework, in which the inferred values of $\boldsymbol{h}$ are those that maximize the posterior, and these tend to often be zero if the prior is sufficiently concentrated around 0. The inferred values are those defined in Eq. 15.1, reproduced here,

$$\boldsymbol{h} = f(\boldsymbol{x}) = \arg\min_{\boldsymbol{h}} L(g(\boldsymbol{h}), \boldsymbol{x})) + \lambda\Omega(\boldsymbol{h})$$

where $L(g(\boldsymbol{h}), \boldsymbol{x})$ is interpreted as $-\log P(\boldsymbol{x} \mid g(\boldsymbol{h}))$ and $\Omega(\boldsymbol{h})$ as $-\log P(\boldsymbol{h})$. This MAP inference view of sparse coding and an interesting probabilistic interpretation of sparse coding are further discussed in Section 19.3.

To relate the generative model of sparse coding to ICA, note how the prior imposes not just sparsity but also independence of the latent variables $h_i$ under $P(\boldsymbol{h})$, which may help to separate different explanatory factors, unlike PCA, factor analysis or probabilistic PCA, because these rely on a Gaussian prior, which yields a factorized prior under any rotation of the factors, multiplication by an orthonormal matrix, as demonstrated in Section 15.6.1.

See Section 17.2 about the manifold interpretation of sparse coding.

TODO: relate to and point to Spike-and-slab sparse coding (Goodfellow *et al.*, 2012) (section?)

## 15.7 Reconstruction Error as Log-Likelihood

Although traditional auto-encoders (like traditional neural networks) were introduced with an associated training loss, just like for neural networks, that training loss can generally be given a probabilistic interpretation as a conditional log-likelihood of the original input $\boldsymbol{x}$, given the reprensentation $\boldsymbol{h}$.

We have already covered negative log-likelihood as a loss function in general for feedforward neural networks in Section 6.3.2. Like prediction error for regular feedforward neural networks, reconstruction error for auto-encoders does not have to be squared error. When we view the loss as negative log-likelihood, we interpret the reconstruction error as

$$L = -\log P(\boldsymbol{x} \mid \boldsymbol{h})$$

where $\boldsymbol{h}$ is the representation, which may generally be obtained through an encoder taking $\boldsymbol{x}$ as input.

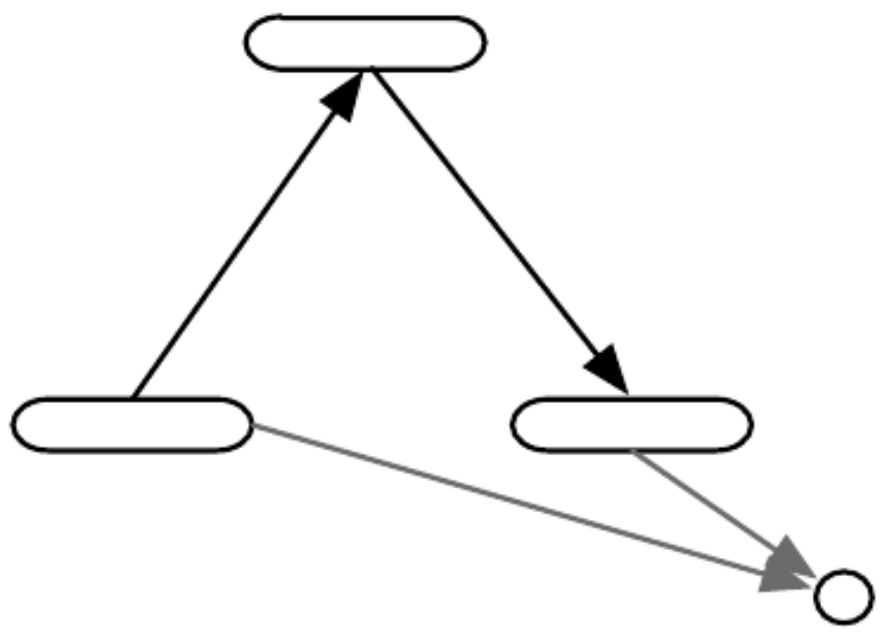

Figure 15.5: The computational graph of an auto-encoder, which is trained to maximize the probability assigned by the decoder $g$ to the data point $\boldsymbol{x}$, given the output of the encoder $\boldsymbol{h} = f(\boldsymbol{x})$. The training objective is thus $L = -\log P(\boldsymbol{x} \mid g(f(\boldsymbol{x})))$, which ends up being squared reconstruction error if we choose a Gaussian reconstruction distribution with mean $g(f(\boldsymbol{x}))$, and cross-entropy if we choose a factorized Bernoulli reconstruction distribution with means $g(f(\boldsymbol{x}))$.

An advantage of this view is that it immediately tells us what kind of loss function one should use depending on the nature of the input. If the input is real-valued and unbounded, then squared error is a reasonable choice of reconstruction

error, and corresponds to $P(\boldsymbol{x} \mid \boldsymbol{h})$ being Normal. If the input is a vector of bits, then cross-entropy is a more reasonable choice, and corresponds to $P(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_i P(x_i \mid \boldsymbol{h})$ with $x_i \mid \boldsymbol{h}$ being Bernoulli-distributed. We then view the decoder $g(\boldsymbol{h})$ as computing the *parameters* of the reconstruction distribution, i.e., $P(\boldsymbol{x} \mid \boldsymbol{h}) = P(\boldsymbol{x} \mid g(\boldsymbol{h}))$.

Another advantage of this view is that we can think about the training of the decoder as estimating the conditional distribution $P(\boldsymbol{x} \mid \boldsymbol{h})$, which comes handy in the probabilistic interpretation of denoising auto-encoders, allowing us to talk about the distribution $P(\boldsymbol{x})$ explicitly or implicitly represented by the auto-encoder (see Sections 15.9, 20.9.3 and 20.11 for more details). In the same spirit, we can rethink the notion of encoder from a simple function to a conditional distribution $Q(\boldsymbol{h} \mid \boldsymbol{x})$, with a special case being when $Q(\boldsymbol{h} \mid \boldsymbol{x})$ is a Dirac at some particular value. Equivalently, thinking about the encoder as a distribution corresponds to *injecting noise* inside the auto-encoder. This view is developed further in Sections 20.9.3 and 20.12.

## 15.8 Sparse Representations

Sparse auto-encoders are auto-encoders which learn a sparse representation, i.e., one whose elements are often either zero or close to zero. Sparse coding was introduced in Section 15.6.2 as a linear factor model in which the prior $P(\boldsymbol{h})$ on the representation $\boldsymbol{h} = f(\boldsymbol{x})$ encourages values at or near 0. In Section 15.8.1, we see how ordinary auto-encoders can be prevented from learning a useless identity transformation by using a sparsity penalty rather than a bottleneck. The main difference between a sparse auto-encoder and sparse coding is that sparse coding has no explicit parametric encoder, whereas sparse auto-encoders have one. The "encoder" of sparse coding is the algorithm that performs the approximate inference, i.e., looks for

$$h^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{h}} \log P(\boldsymbol{h} \mid \boldsymbol{x}) = \arg\min_{\boldsymbol{h}} \frac{||\boldsymbol{x} - (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h})||^2}{\sigma^2} - \log P(\boldsymbol{h}) \tag{15.7}$$

where $\sigma^2$ is a reconstruction variance parameter (which should equal the average squared reconstruction error[5]), and $P(\boldsymbol{h})$ is a "sparse" prior that puts more probability mass around $\boldsymbol{h} = 0$, such as the Laplacian prior, with factorized marginals

$$P(h_i) = \frac{\lambda}{2} e^{\lambda |h_i|} \tag{15.8}$$

but can be lumped into the regularizer which controls the strength of the sparsity prior, defined in Eq. 15.8, for example.

or the Student-t prior, with factorized marginals

$$P(h_i) \propto \frac{1}{(1 + \frac{h_i}{\nu})^{\frac{\nu}{}}}. \tag{15.9}$$

The advantages of such a non-parametric encoder and the sparse coding approach over sparse auto-encoders are that

1. it can in principle minimize the combination of reconstruction error and log-prior better than any parametric encoder,

2. it performs what is called *explaining away* (see Figure 13.8), i.e., it allows to "choose" some "explanations" (hidden factors) and inhibits the others.

The disadvantages are that

1. computing time for encoding the given input $\boldsymbol{x}$, i.e., performing inference (computing the representation $\boldsymbol{h}$ that goes with the given $\boldsymbol{x}$) can be substantially larger than with a parametric encoder (because an optimization must be performed *for each example* $\boldsymbol{x}$), and

2. the resulting encoder function could be non-smooth and possibly too non-linear (with two nearby $\boldsymbol{x}$'s being associated with very different $\boldsymbol{h}$'s), potentially making it more difficult for the downstream layers to properly generalize.

In Section 15.8.2, we describe PSD (Predictive Sparse Decomposition), which combines a non-parametric encoder (as in sparse coding, with the representation obtained via an optimization) and a parametric encoder (like in the sparse auto-encoder). Section 15.9 introduces the Denoising Auto-Encoder (DAE), which puts pressure on the representation by requiring it to extract information about the underlying distribution and where it concentrates, so as to be able to denoise a corrupted input. Section 15.10 describes the Contractive Auto-Encoder (CAE), which optimizes an explicit regularization penalty that aims at making the representation as insensitive as possible to the input, while keeping the information sufficient to reconstruct the training examples.

### 15.8.1 Sparse Auto-Encoders

A sparse auto-encoder is simply an auto-encoder whose training criterion involves a sparsity penalty $\Omega(\boldsymbol{h})$ in addition to the reconstruction error:

$$L = -\log P(\boldsymbol{x} \mid g(\boldsymbol{h})) + \Omega(\boldsymbol{h}) \tag{15.10}$$

where $g(\boldsymbol{h})$ is the decoder output and typically we have $\boldsymbol{h} = f(\boldsymbol{x})$, the encoder output.

We can think of that penalty $\Omega(\boldsymbol{h})$ simply as a regularizer or as a log-prior on the representations $\boldsymbol{h}$. For example, the sparsity penalty corresponding to the Laplace prior ($\frac{\lambda}{2} e^{-\lambda|h_i|}$) is the absolute value sparsity penalty (see also Eq. 15.8 above):

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|$$

$$-\log P(\boldsymbol{h}) = \sum_i \log \frac{\lambda}{2} + \lambda |h_i| = \text{const} + \Omega(\boldsymbol{h}) \tag{15.11}$$

where the constant term depends only of $\lambda$ and not $\boldsymbol{h}$ (which we typically ignore in the training criterion because we consider $\lambda$ as a hyperparameter rather than a parameter). Similarly (as per Eq. 15.9), the sparsity penalty corresponding to the Student-t prior (Olshausen and Field, 1997) is

$$\Omega(\boldsymbol{h}) = \sum_i \frac{\nu + 1}{2} \log(1 + \frac{h_i^2}{\nu}) \tag{15.12}$$

where $\nu$ is considered to be a hyperparameter.

The early work on sparse auto-encoders (Ranzato *et al.*, 2007a, 2008) considered various forms of sparsity and proposed a connection between sparsity regularization and the partition function gradient in energy-based models (see Section TODO). The idea is that a regularizer such as sparsity makes it difficult for an auto-encoder to achieve zero reconstruction error everywhere. If we consider reconstruction error as a proxy for energy (unnormalized log-probability of the data), then minimizing the training set reconstruction error forces the energy to be low on training examples, while the regularizer prevents it from being low everywhere. The same role is played by the gradient of the partition function in energy-based models such as the RBM (Section TODO).

However, the sparsity penalty of sparse auto-encoders does not need to have a probabilistic interpretation. For example, Goodfellow *et al.* (2009) successfully used the following sparsity penalty, which does not try to bring $h_i$ all the way down to 0, but only towards some low target value such as $\rho = 0.05$.

$$\Omega(\boldsymbol{h}) = \sum_i \rho \log h_i + (1 - \rho) \log(1 - h_i) \tag{15.13}$$

where $0 < h_i < 1$, usually with $h_i = \text{sigmoid}(a_i)$. This is just the cross-entropy between the Bernoulli distributions with probability $p = h_i$ and the target Bernoulli distribution with probability $p = \rho$.

One way to achieve *actual zeros* in $\boldsymbol{h}$ for sparse (and denoising) auto-encoders was introduced in Glorot *et al.* (2011c). The idea is to use a half-rectifier (a.k.a. simply as "rectifier") or ReLU (Rectified Linear Unit, introduced in Glorot *et al.* (2011b) for deep supervised networks and earlier in Nair and Hinton (2010a) in the context of RBMs) as the output non-linearity of the encoder. With a prior that actually pushes the representations to zero (like the absolute value penalty), one can thus indirectly control the average number of zeros in the representation. ReLUs were first successfully used for *deep feedforward networks* in Glorot *et al.* (2011a), achieving for the first time the ability to *train fairly deep supervised networks without the need for unsupervised pre-training*, and this turned out to be an important component in the 2012 object recognition breakthrough with deep convolutional networks (Krizhevsky *et al.*, 2012b).

Interestingly, the "regularizer" used in sparse auto-encoders does not conform to the classical interpretation of regularizers as priors on the parameters. That classical interpretation of the regularizer comes from the MAP (Maximum A Posteriori) point estimation (see Section 5.5.1) of parameters associated with the Bayesian view of parameters as random variables and considering the joint distribution of data $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$ (see Section 5.7):

$$\arg\max_{\boldsymbol{\theta}} P(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \left(\log P(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log P(\boldsymbol{\theta})\right)$$

where the first term on the right is the usual data log-likelihood term and the second term, the log-prior over parameters, incorporates the preference over particular values of $\boldsymbol{\theta}$.

With regularized auto-encoders such as sparse auto-encoders and contractive auto-encoders, instead, the regularizer corresponds to a *log-prior over the representation, or over latent variables.* In the case of sparse auto-encoders, predictive sparse decomposition and contractive auto-encoders, the regularizer specifies a *preference over functions* of the data, rather than over parameters. This makes such a regularizer *data-dependent*, unlike the classical parameter log-prior. Specifically, in the case of the sparse auto-encoder, it says that we prefer an encoder whose output produces values closer to 0. Indirectly (when we marginalize over the training distribution), this is also indicating a preference over parameters, of course.

### 15.8.2 Predictive Sparse Decomposition

TODO: we have too many forward refs to this section. There are 150 lines about PSD in this section and at least 20 lines of forward references to this section in this chapter, some of which are just 100 lines away. Predictive sparse decomposition (PSD) is a variant that combines sparse coding and a parametric

encoder (Kavukcuoglu *et al.*, 2008b), i.e., it has both a parametric encoder and iterative inference. It has been applied to unsupervised feature learning for object recognition in images and video (Kavukcuoglu *et al.*, 2009, 2010b; Jarrett *et al.*, 2009a; Farabet *et al.*, 2011), as well as for audio (Henaff *et al.*, 2011). The representation is considered to be a free variable (possibly a latent variable if we choose a probabilistic interpretation) and the training criterion combines a sparse coding criterion with a term that encourages the optimized sparse representation $\boldsymbol{h}$ (after inference) to be close to the output of the encoder $f(\boldsymbol{x})$:

$$L = \arg\min_{\boldsymbol{h}} \left( ||\boldsymbol{x} - g(\boldsymbol{h})||^2 + \lambda |\boldsymbol{h}|_1 + \gamma ||\boldsymbol{h} - f(\boldsymbol{x})||^2 \right) \tag{15.14}$$

where $f$ is the encoder and $g$ is the decoder. Like in sparse coding, for each example $\boldsymbol{x}$ an iterative optimization is performed in order to obtain a representation $\boldsymbol{h}$. However, because the iterations can be initialized from the output of the encoder, i.e., with $\boldsymbol{h} = f(\boldsymbol{x})$, only a few steps (e.g. 10) are necessary to obtain good results. Simple gradient descent on $\boldsymbol{h}$ has been used by the authors. After $\boldsymbol{h}$ is settled, both $g$ and $f$ are updated towards minimizing the above criterion. The first two terms are the same as in L1 sparse coding while the third one encourages $f$ to predict the outcome of the sparse coding optimization, making it a better choice for the initialization of the iterative optimization. Hence $f$ can be used as a parametric approximation to the non-parametric encoder implicitly defined by sparse coding. It is one of the first instances of *learned approximate inference* (see also Sec. 19.8). Note that this is different from separately doing sparse coding (i.e., training $g$) and then training an approximate inference mechanism $f$, since both the encoder and decoder are trained together to be "compatible" with each other. Hence the decoder will be learned in such a way that inference will tend to find solutions that can be well approximated by the approximate inference. TODO: this is probably too much forward reference, when we bring these things in we can remind people that they resemble PSD, but it doesn't really help the reader to say that the thing we are describing now is similar to things they haven't seen yet A similar example is the variational auto-encoder, in which the encoder acts as approximate inference for the decoder, and both are trained jointly (Section 20.9.3). See also Section 20.9.4 for a probabilistic interpretation of PSD in terms of a variational lower bound on the log-likelihood.

In practical applications of PSD, the iterative optimization is only used during training, and $f$ is used to compute the learned features. It makes computation fast at recognition time and also makes it easy to use the trained features $f$ as initialization (unsupervised pre-training) for the lower layers of a deep net. Like other unsupervised feature learning schemes, PSD can be stacked greedily, e.g., training a second PSD on top of the features extracted by the first one, etc.

## 15.9 Denoising Auto-Encoders

The Denoising Auto-Encoder (DAE) was first proposed (Vincent *et al.*, 2008, 2010) as a means of forcing an auto-encoder to learn to capture the data distribution without an explicit constraint on either the dimension or the sparsity of the learned representation. It was motivated by the idea that in order to fully capture a complex distribution, an auto-encoder needs to have at least as many hidden units as needed by the complexity of that distribution. Hence its dimensionality should not be restricted to the input dimension.

The principle of the denoising auto-encoder is deceptively simple and illustrated in Figure 15.6: the encoder sees as input a corrupted version of the input, but the decoder tries to reconstruct the clean uncorrupted input.

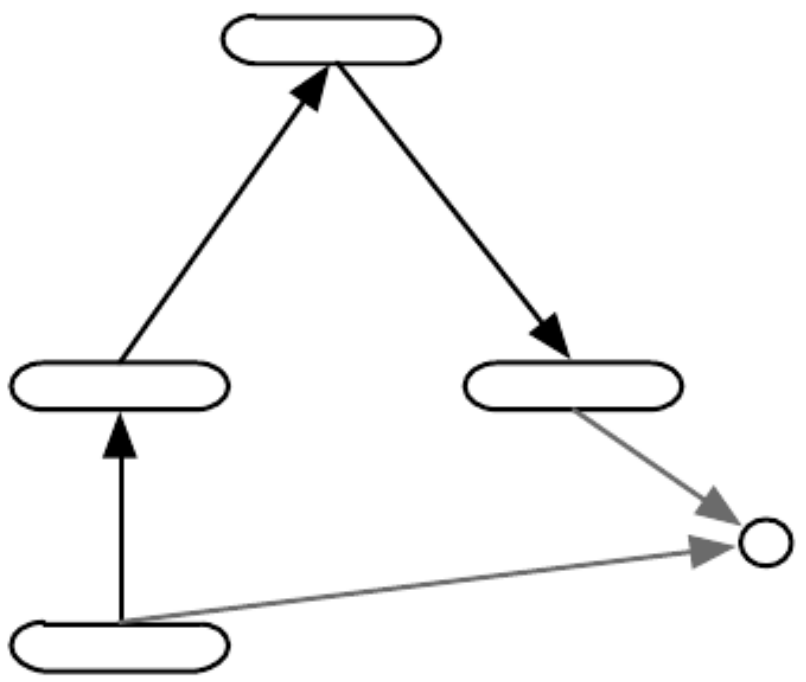

Figure 15.6: The computational graph of a denoising auto-encoder, which is trained to reconstruct the clean data point $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, i.e., to minimize the loss $L = -\log P(\boldsymbol{x} \mid g(f(\tilde{\boldsymbol{x}})))$, where $\tilde{\boldsymbol{x}}$ is a corrupted version of the data example $\boldsymbol{x}$, obtained through a given corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.

Mathematically, and following the notations used in this chapter, this can be formalized as follows. We introduce a corruption process $C(\tilde{\mathbf{x}} \mid \mathbf{x})$ which represents a conditional distribution over corrupted samples $\tilde{\mathbf{x}}$, given a data sample $\mathbf{x}$. The auto-encoder then learns a *reconstruction distribution* $P(\mathbf{x} \mid \tilde{\mathbf{x}})$ estimated from training pairs $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$, as follows:

1. Sample a training example $\mathbf{x} = \boldsymbol{x}$ from the data generating distribution (the training set).

2. Sample a corrupted version $\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}$ from the conditional distribution $C(\tilde{\mathbf{x}} \mid$

$\mathbf{x} = \boldsymbol{x}$).

3. Use $(\boldsymbol{x}, \tilde{\boldsymbol{x}})$ as a training example for estimating the auto-encoder reconstruction distribution $P(\boldsymbol{x} \mid \tilde{\boldsymbol{x}}) = P(\boldsymbol{x} \mid g(\boldsymbol{h}))$ with $\boldsymbol{h}$ the output of encoder $f(\tilde{\boldsymbol{x}})$ and $g(\boldsymbol{h})$ the output of the decoder.

Typically we can simply perform gradient-based approximate minimization (such as minibatch gradient descent) on the negative log-likelihood $-\log P(\boldsymbol{x} \mid \boldsymbol{h})$, i.e., the denoising reconstruction error, using back-propagation to compute gradients, just like for regular feedforward neural networks (the only difference being the corruption of the input and the choice of target output).

We can view this training objective as performing stochastic gradient descent on the denoising reconstruction error, but where the "noise" now has two sources:

1. the choice of training sample $\boldsymbol{x}$ from the data set, and

2. the random corruption applied to $\boldsymbol{x}$ to obtain $\tilde{\boldsymbol{x}}$.

We can therefore consider that the DAE is performing stochastic gradient descent on the following expectation:

$$-E_{\boldsymbol{x} \sim Q(\ )} E_{\tilde{\ } \sim C(\ \mid \boldsymbol{x})} \log P(\boldsymbol{x} \mid g(f(\tilde{\boldsymbol{x}})))$$

where $Q(\mathbf{x})$ is the training distribution.

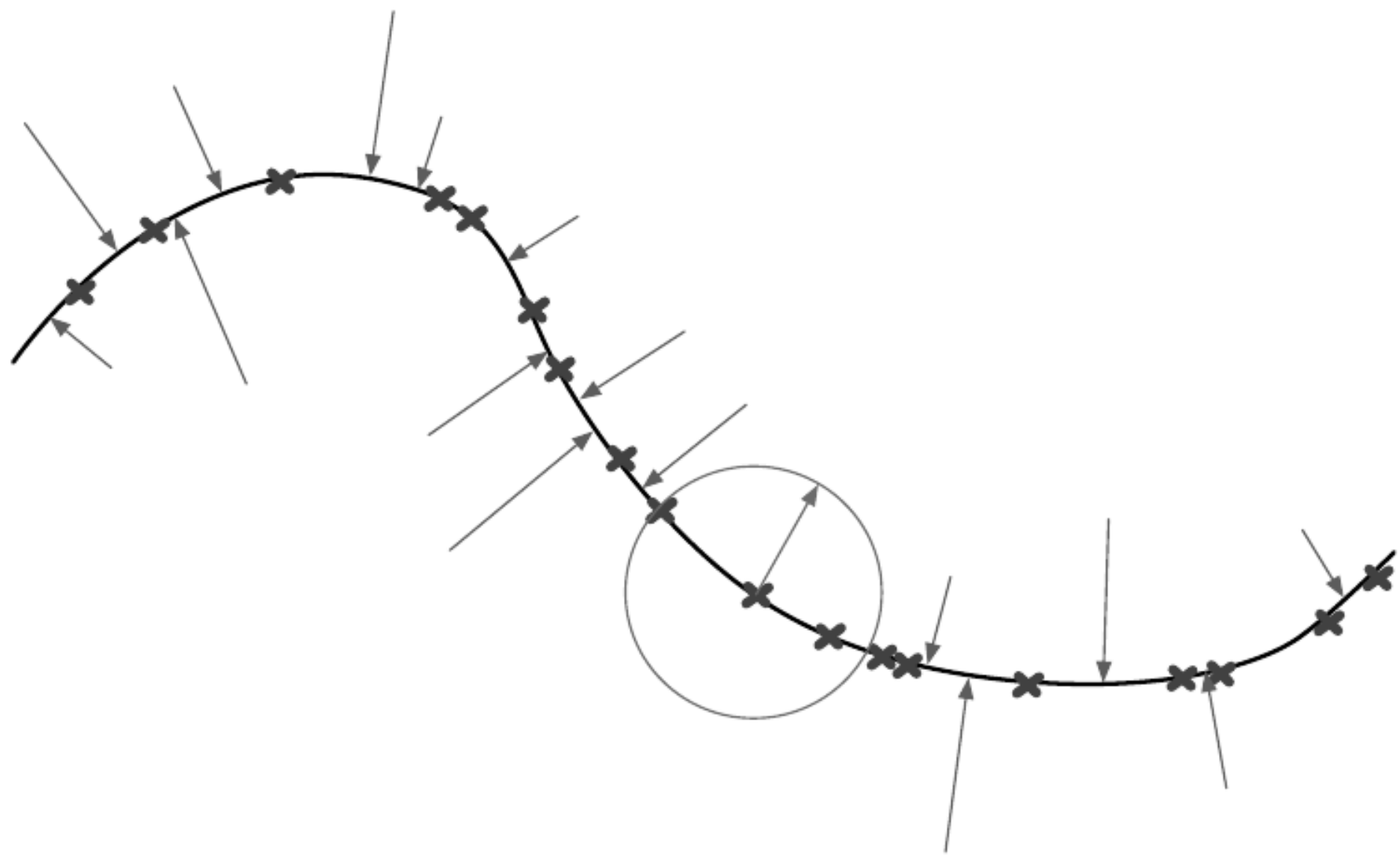

Figure 15.7: A denoising auto-encoder is trained to reconstruct the clean data point $\boldsymbol{x}$ from it corrupted version $\tilde{\boldsymbol{x}}$. In the figure, we illustrate the corruption process $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ by a grey circle of equiprobable corruptions, and grey arrow for the corruption process) acting on examples $\boldsymbol{x}$ (red crosses) lying near a low-dimensional manifold near which probability concentrates. When the denoising auto-encoder is trained to minimize the average of squared errors $||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||$ , the reconstruction $g(f(\tilde{\boldsymbol{x}}))$ estimates $[\mathbf{x} \mid \tilde{\boldsymbol{x}}]$, which approximately points orthogonally towards the manifold, since it estimates the center of mass of the clean points $\boldsymbol{x}$ which could have given rise to $\tilde{\boldsymbol{x}}$. The auto-encoder thus learns a vector field $g(f(\boldsymbol{x})) - \boldsymbol{x}$ (the green arrows) and it turns out that this vector field estimates the gradient field ——— (up to a multiplicative factor that is the average root mean square reconstruction error), where $Q$ is the unknown data generating distribution.

### 15.9.1 Learning a Vector Field that Estimates a Gradient Field

As illustrated in Figure 15.7, a very important property of DAEs is that their training criterion makes the auto-encoder learn a vector field $(g(f(\boldsymbol{x})) - \boldsymbol{x})$ that estimates the gradient field (or *score*) $\frac{\partial \log Q(\boldsymbol{x})}{\partial x}$, as per Eq. 15.15. A first result in this direction was proven by Vincent (2011a), showing that minimizing squared reconstruction error in a denoising auto-encoder with Gaussian noise was related to *score matching* (Hyvärinen, 2005a), making the denoising criterion a regularized form of score matching called *denoising score matching* (Kingma and LeCun, 2010a). Score matching is an alternative to maximum likelihood and provides a consistent estimator. It is discussed further in Section 18.4. The denoising version

is discussed in Section 18.5.

The connection between denoising auto-encoders and score matching was first made (Vincent, 2011a) in the case where the denoising auto-encoder has a particular parametrization (one hidden layer, sigmoid activation functions on hidden units, linear reconstruction), in which case the denoising criterion actually corresponds to a regularized form of score matching on a Gaussian RBM (with binomial hidden units and Gaussian visible units). The connection between ordinary auto-encoders and Gaussian RBMs had previously been made by Bengio and Delalleau (2009), which showed that contrastive divergence training of RBMs was related to an associated auto-encoder gradient, and later by Swersky (2010), which showed that non-denoising reconstruction error corresponded to score matching plus a regularizer.

The fact that the denoising criterion yields an estimator of the score for general encoder/decoder parametrizations has been proven (Alain and Bengio, 2012, 2013) in the case where the corruption and the reconstruction distributions are Gaussian (and of course $\boldsymbol{x}$ is continuous-valued), i.e., with the squared error denoising error

$$||g(f(\tilde{\boldsymbol{x}})) - \boldsymbol{x}||^2$$

and corruption

$$C(\tilde{\mathbf{x}} = \tilde{\boldsymbol{x}}|\boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}; \mu = \boldsymbol{x}, \Sigma = \sigma^2 I)$$

with noise variance $\sigma^2$.

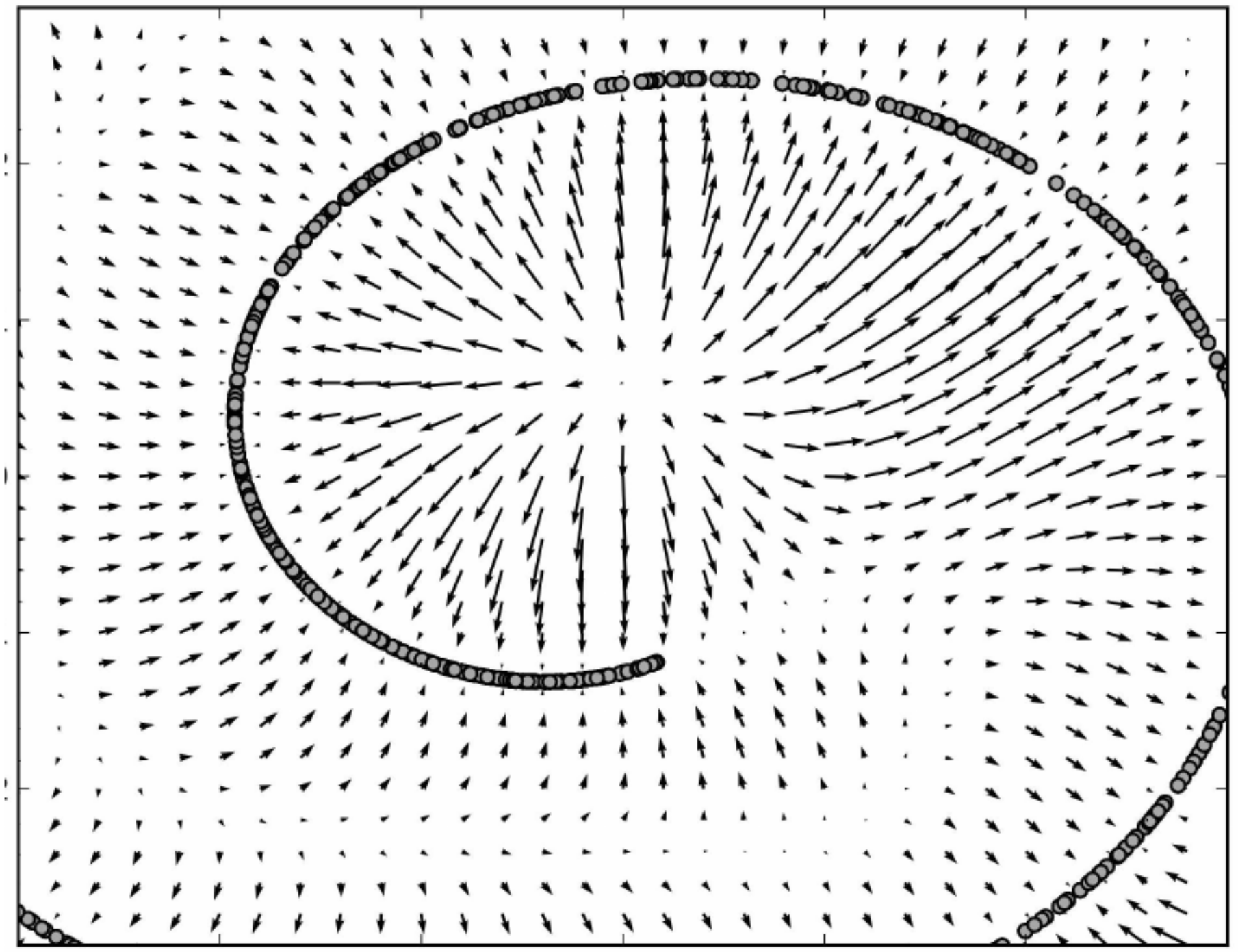

Figure 15.8: Vector field learned by a denoising auto-encoder around a 1-D curved manifold near which the data (orange circles) concentrates in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the auto-encoder and points towards higher probability according to the implicitly estimated probability distribution. Note that the vector field has zeros at both peaks of the estimated density function (on the data manifolds) and at troughs (local minima) of that density function, e.g., on the curve that separates different arms of the spiral or in the middle of it.

More precisely, the main theorem states that $\frac{g(f(\boldsymbol{x}))-\boldsymbol{x}}{\sigma}$ is a consistent estimator of $\frac{\partial \log Q(\boldsymbol{x})}{\partial \boldsymbol{x}}$, where $Q(\boldsymbol{x})$ is the data generating distribution,

$$\frac{g(f(\boldsymbol{x})) - \boldsymbol{x}}{\sigma^2} \rightarrow \frac{\partial \log Q(\boldsymbol{x})}{\partial \boldsymbol{x}}, \tag{15.15}$$

so long as $f$ and $g$ have sufficient capacity to represent the true score (and assuming that the expected training criterion can be minimized, as usual when proving consistency associated with a training objective).

Note that in general, there is no guarantee that the reconstruction $g(f(\boldsymbol{x}))$ minus the input $\boldsymbol{x}$ corresponds to the gradient of something (the estimated score should be the gradient of the estimated log-density with respect to the input

$\boldsymbol{x}$). That is why the early results (Vincent, 2011a) are specialized to particular parametrizations where $g(f(\boldsymbol{x})) - \boldsymbol{x}$ is the derivative of something. See a more general treatment by Kamyshanska and Memisevic (2015).

Although it was intuitively appealing that in order to denoise correctly one must capture the training distribution, the above consistency result makes it mathematically very clear in what sense the DAE is capturing the input distribution: it is estimating the gradient of its energy function (i.e., of its log-density), i.e., learning to point towards more probable (lower energy) configurations. Figure 15.8 (see details of experiment in Alain and Bengio (2013)) illustrates this. Note how the norm of reconstruction error (i.e. the norm of the vectors shown in the figure) is related to but *different* from the energy (unnormalized log-density) associated with the estimated model. The energy should be low only where the probability is high. The reconstruction error (norm of the estimated score vector) is low where probability is near a peak of probability (or a trough of energy), but it can also be low at *maxima* of energy (minima of probability).

Section 20.11 continues the discussion of the relationship between denoising auto-encoders and probabilistic modeling by showing how one can *generate* from the distribution implicitly estimated by a denoising auto-encoder. Whereas (Alain and Bengio, 2013) generalized the score estimation result of Vincent (2011a) to arbitrary parametrizations, the result from Bengio *et al.* (2013b), discussed in Section 20.11, provides a probabilistic – and in fact generative – interpretation to every denoising auto-encoder.

## 15.10 Contractive Auto-Encoders

The Contractive Auto-Encoder or CAE (Rifai *et al.*, 2011a,c) introduces an explicit regularizer on the code $\boldsymbol{h} = f(\boldsymbol{x})$, encouraging the derivatives of $f$ to be as small as possible:

$$\Omega(\boldsymbol{h}) = \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 \tag{15.16}$$

which is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function. Whereas the denoising auto-encoder learns to contract the reconstruction function (the composition of the encoder and decoder), the CAE learns to specifically contract the encoder. See Figure 17.13 for a view of how contraction near the data points makes the auto-encoder capture the manifold structure.

If it weren't for the opposing force of reconstruction error, which attempts to make the code $\boldsymbol{h}$ keep all the information necessary to *reconstruct training examples*, the CAE penalty would yield a code $\boldsymbol{h}$ that is constant and does not

depend on the input $\boldsymbol{x}$. The compromise between these two forces yields an auto-encoder whose derivatives $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ are tiny in most directions, except those that are needed to reconstruct training examples, i.e., the directions that are tangent to the manifold near which data concentrate. Indeed, in order to distinguish (and thus, reconstruct correctly) two nearby examples on the manifold, one must assign them a different code, i.e., $f(\boldsymbol{x})$ must vary as $\boldsymbol{x}$ moves from one to the other, i.e., in the direction of a tangent to the manifold.

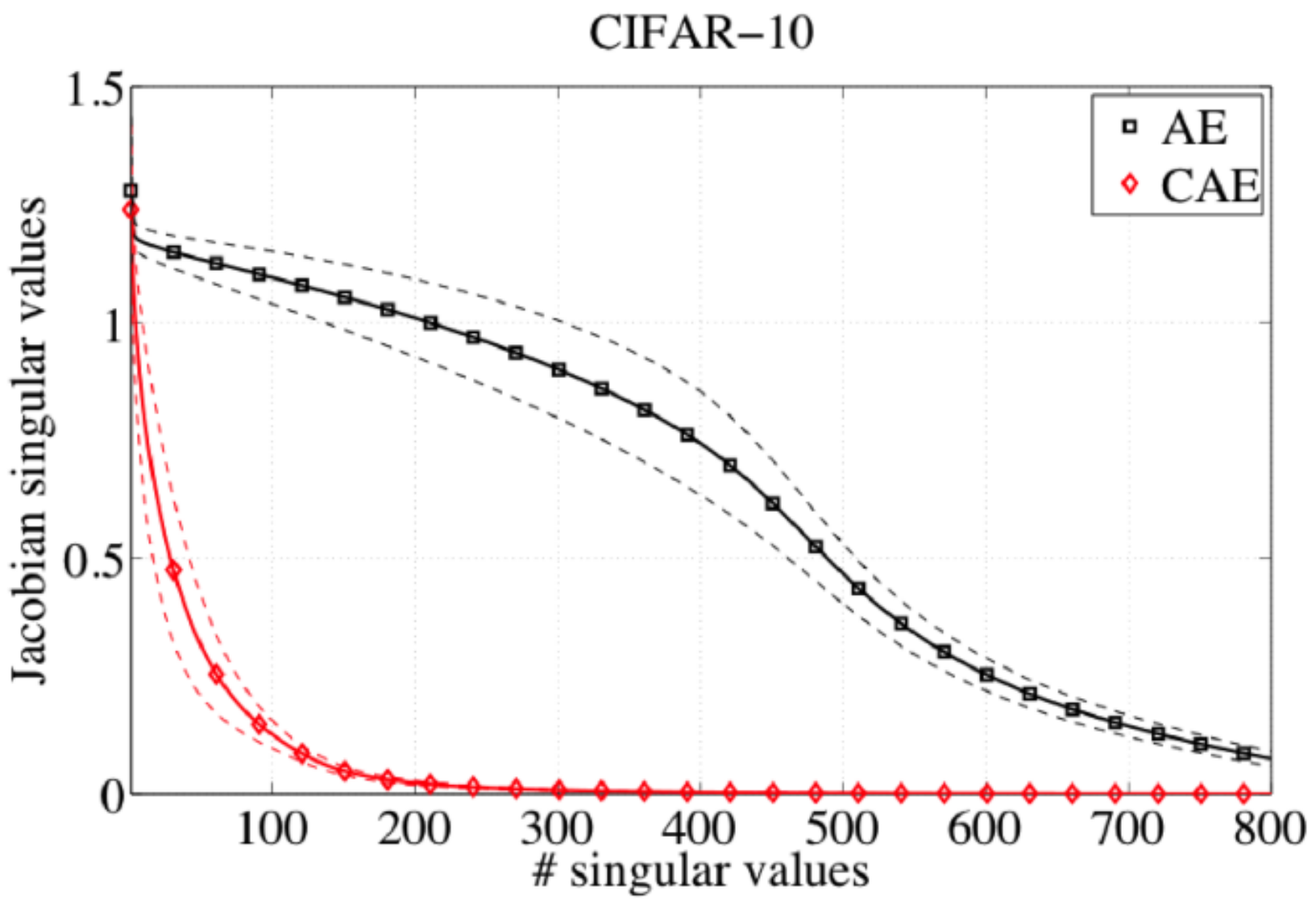


Figure 15.9: Average (over test examples) of the singular value spectrum of the Jacobian matrix —— for the encoder $f$ learned by a regular auto-encoder (AE) versus a contractive auto-encoder (CAE). This illustrates how the contractive regularizer yields a smaller set of directions in input space (those corresponding to large singular value of the Jacobian) which provoke a response in the representation $\boldsymbol{h}$ while the representation remains almost insensitive for most directions of change in the input.

What is interesting is that this penalty forces more strongly the representation to be invariant in directions orthogonal to the manifold. This can be seen clearly by comparing the singular value spectrum of the Jacobian $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ for different auto-encoders, as shown in Figure 15.9. We see that the CAE manages to concentrate the sensitivity of the representation in fewer dimensions than a regular (or sparse) auto-encoder. Figure 17.3 illustrates tangent vectors obtained by a CAE on the MNIST digits dataset, showing that the leading tangent vectors correspond to small deformations such as translation. More impressively, Figure 15.10 shows tangent vectors learned on 32×32 color (RGB) CIFAR-10 images by a CAE,

compared to the tangent vectors by a non-distributed representation learner (a mixture of local PCAs).

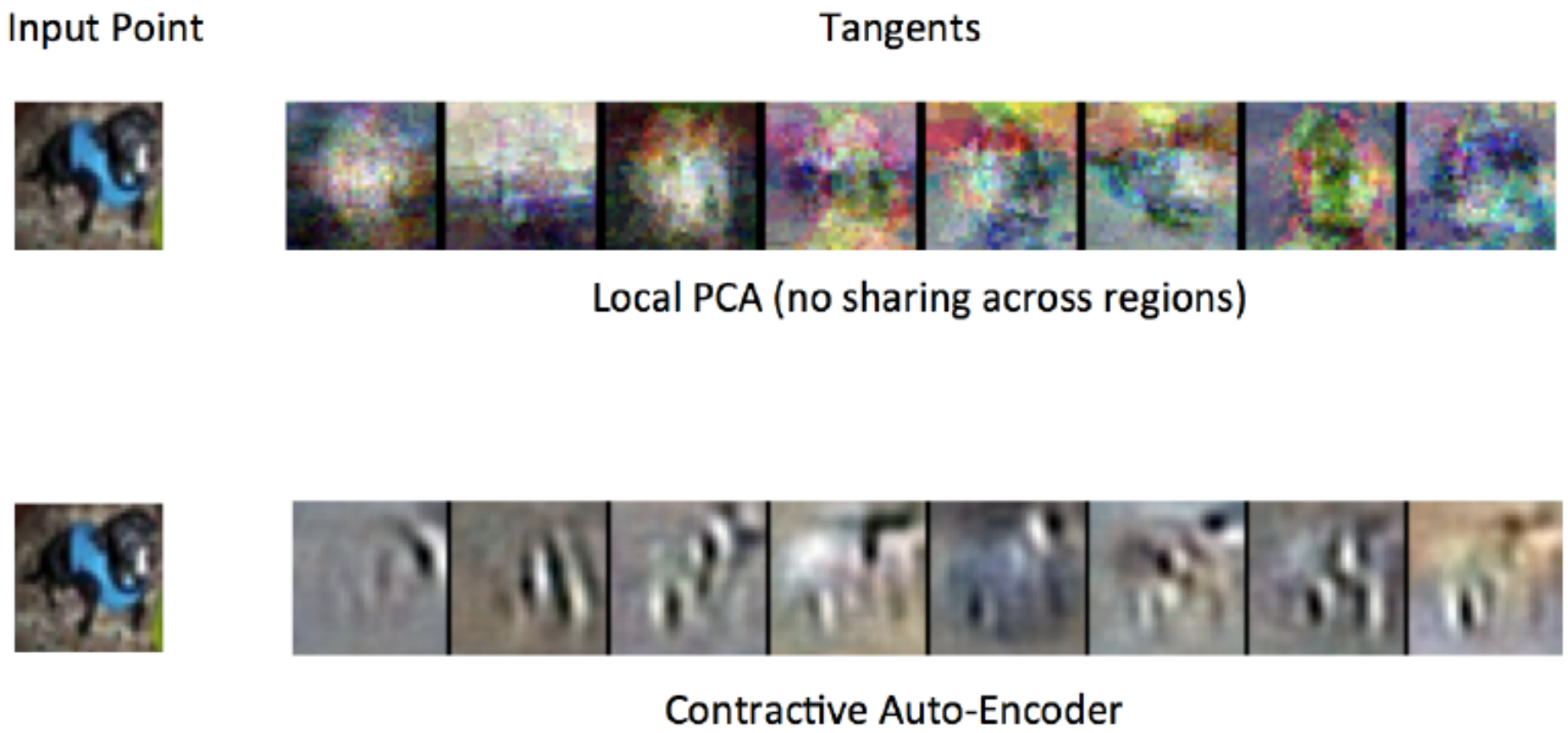


Figure 15.10: Illustration of tangent vectors (bottom) of the manifold estimated by a contractive auto-encoder (CAE), at some input point (left, CIFAR-10 image of a dog). See also Fig. 17.3. Each image on the right corresponds to a tangent vector, either estimated by a local PCA (equivalent to a Gaussian mixture), top, or by a CAE (bottom). The tangent vectors are estimated by the leading singular vectors of the Jacobian matrix — of the input-to-code mappiing. Although both local PCA and CAE can capture local tangents that are different in different points, the local PCA does not have enough training data to meaningful capture good tangent directions, whereas the CAE does (because it exploits parameter sharing across different locations that share a subset of active hidden units). The CAE tangent directions typically correspond to moving or changing parts of the object (such as the head or legs), which corresponds to plausible changes in the input image.

One practical issue with the CAE regularization criterion is that although it is cheap to compute in the case of a single hidden layer auto-encoder, it becomes much more expensive in the case of deeper auto-encoders. The strategy followed by Rifai *et al.* (2011a) is to separately pre-train each single-layer auto-encoder stacked to form a deeper auto-encoder. However, a deeper encoder could be advantageous in spite of the computational overhead, as argued by Schulz and Behnke (2012).

Another practical issue is that the contraction penalty on the encoder $f$ could yield useless results if the decoder $g$ would exactly compensate (e.g. by being scaled up by exactly the same amount as $f$ is scaled down). In Rifai *et al.* (2011a), this is compensated by tying the weights of $f$ and $g$, both being of the form of an affine transformation followed by a non-linearity (e.g. sigmoid), i.e., the weights of $g$ and the transpose of the weights of $f$.

# Chapter 16

# Structured Probabilistic Models for Deep Learning

Deep learning draws upon many modeling formalisms that researchers can use to guide their design efforts and describe their algorithms. One of these formalisms is the idea of *structured probabilistic models*. We have already discussed structured probabilistic models briefly in Sec. 3.14. That brief presentation was sufficient to understand how to use structured probabilistic models as a language to describe some of the algorithms in part II of this book. Now, in part III, structured probabilistic models are a key ingredient of many of the most important research topics in deep learning. In order to prepare to discuss these research ideas, this chapter describes structured probabilistic models in much greater detail. This chapter is intended to be self-contained; the reader does not need to review the earlier introduction before continuing with this chapter.

A structured probabilistic model is a way of describing a probability distribution, using a graph to describe which random variables in the probability distribution interact with each other directly. Here we use "graph" in the graph theory sense–a set of vertices connected to one another by a set of edges. Because the structure of the model is defined by a graph, these models are often also referred to as *graphical models*.

The graphical models research community is large and has developed many different models, training algorithms, and inference algorithms. In this chapter, we provide basic background on some of the most central ideas of graphical models, with an emphasis on the concepts that have proven most useful to the deep learning research community. If you already have a strong background in graphical models, you may wish to skip most of this chapter. However, even a graphical model expert may benefit from reading the final section of this chapter, section 16.6, in which we highlight some of the unique ways that graphical models are used for deep

learning algorithms. Deep learning practitioners tend to use very different model structures, learning algorithms, and inference procedures than are commonly used by the rest of the graphical models research community. In this chapter, we identify these differences in preferences and explain the reasons for them.

In this chapter we first describe the challenges of building large-scale probabilistic models in section 16.1. Next, we describe how to use a graph to describe the structure of a probability distribution in section 16.2. We then revisit the challenges we described in section 16.1 and show how the structured approach to probabilistic modeling can overcome these challenges in section 16.3. One of the major difficulties in graphical modeling is understanding which variables need to be able to interact directly, i.e., which graph structures are most suitable for a given problem. We outline two approaches to resolving this difficulty by learning about the dependencies in section 16.4. Finally, we close with a discussion of the unique emphasis that deep learning practitioners place on specific approaches to graphical modeling in section 16.6.

## 16.1 The Challenge of Unstructured Modeling

The goal of deep learning is to scale machine learning to the kinds of challenges needed to solve artificial intelligence. This means being able to understand high-dimensional data with rich structure. For example, we would like AI algorithms to be able to understand natural images[1], audio waveforms representing speech, and documents containing multiple words and punctuation characters.

Classification algortihms can take such a rich high-dimensional input and summarize it with a categorical label—what object is in a photo, what word is spoken in a recording, what topic a document is about. The process of classification discards most of the information in the input and produces on a single output (or a probability distribution over values of that single output). The classifier is also often able to ignore many parts of the input. For example, when recognizing an object in a photo, it is usually possible to ignore the background of the photo.

It is possible to ask probabilistic models to do many other tasks. These tasks are often more expensive than classification. Some of them require producing multiple output values. Most require a complete understanding of the entire structure of the input, with no option to ignore sections of it. These tasks include

- Density estimation: given an input $\boldsymbol{x}$, the machine learning system returns an estimate of $p(\boldsymbol{x})$. This requires only a single output, but it does require a complete understanding of the entire input. If even one element of the vector is unusual, the system must assign it a low probability.

[1] A *natural image* is an image that might captured by a camera in a reasonably ordinary environment, as opposed to synthetically rendered images, screenshots of web pages, etc.

- Denoising: given a damaged or incorrectly observed input $\tilde{\boldsymbol{x}}$, the machine learning system returns an estimate of the original or correct $\boldsymbol{x}$. For example, the machine learning system might be asked to remove dust or scratches from an old photograph. This requires multiple outputs (every element of the estimated clean example $\boldsymbol{x}$) and an understanding of the entire input (since even one damaged area will still reveal the final estimate as being damaged).

- Missing value imputation: given the observations of some elements of $\boldsymbol{x}$, the model is asked to return estimates of or a probability distribution over some or all of the unobserved elements of $\boldsymbol{x}$. This requires multiple outputs, and because the model could be asked to restore any of the elements of $\boldsymbol{x}$, it must understand the entire input.

- Sampling: the model generates new samples from the distribution $p(\boldsymbol{x})$. Applications include speech synthesis, i.e. producing new waveforms thatsound like natural human speech. This requires multiple output values and a good model of the entire input. If the samples have even one element drawn from the wrong distribution, then the sampling process is wrong.

For an example of the sampling tasks on small natural images, see Fig. 16.1.

Modeling a rich distribution over thousands or millions of random variables is a challenging task, both computationally and statistically. Suppose we only wanted to model binary variables. This is the simplest possible case, and yet already it seems overwhelming. For a small, $32 \times 32$ pixel color (RGB) image, there are $2^{3072}$ possible binary images of this form. This number is over $10^{800}$ times larger than the estimated number of atoms in the universe.

In general, if we wish to model a distribution over a random vector $\mathbf{x}$ containing $n$ discrete variables capable of taking on $k$ values each, then the naive approach of representing $P(\mathbf{x})$ by storing a lookup table with one probability value per possible outcome requires $k^n$ parameters!

This is not feasible for several reasons:

- **Memory: the cost of storing the representation** : For all but very small values of $n$ and $k$, representing the distribution as a table will require too many values to store.

- **Statistical efficiency**: As the number of parameters in a model increases, so does the amount of training examples needed to choose the values of those parameters using a statistical estimator. Because the table-based model has an astronomical number of parameters, it will require an astronomically large training set to fit accurately. Any such model will overfit the training set very badly.

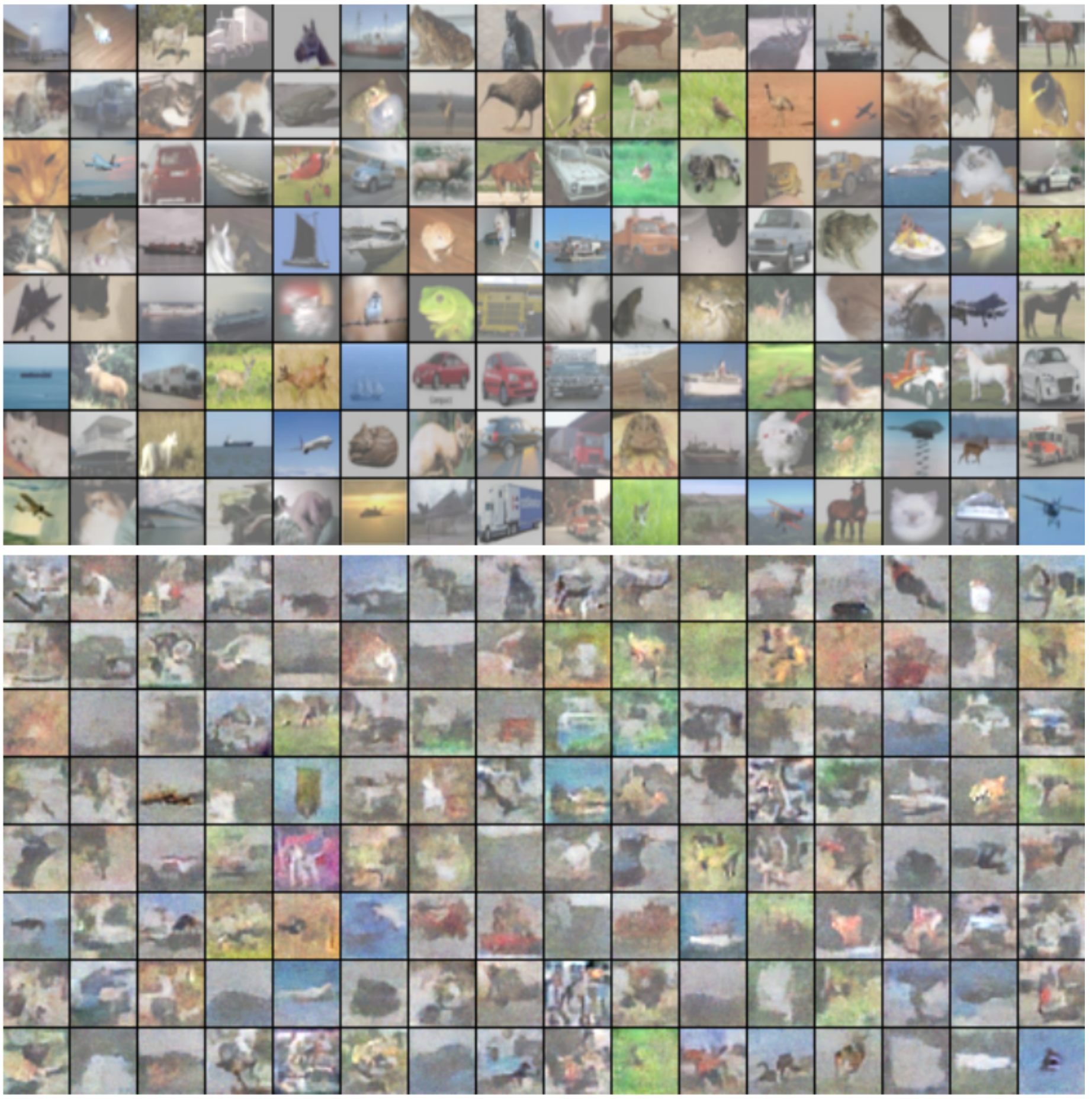

Figure 16.1: Probabilistic modeling of natural images. *Top:* Example $32 \times 32$ pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). *Bottom:* Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training example that is closest to it in Euclidean space. This comparison allows us to see that the model is truly synthesizing new images, rather than memorizing the training data. Contrast of both sets of images has been adjusted for display. Figure reproduced with permission from (Courville *et al.*, 2011).

- **Runtime: the cost of inference**: Suppose we want to perform an *inference* task where we use our model of the joint distribution $P(\mathbf{x})$ to compute some other distribution, such as the marginal distribution $P(\mathrm{x}_1)$ or the conditional distribution $P(\mathrm{x}_2 \mid \mathrm{x}_1)$. Computing these distributions will require summing across the entire table, so the runtime of these operations is as high as the intractable memory cost of storing the model.

- **Runtime: the cost of sampling**: Likewise, suppose we want to draw a sample from the model. The naive way to do this is to sample some value $u \sim U(0, 1)$, then iterate through the table adding up the probability values until they exceed $u$ and return the outcome whose probability value was added last. This requires reading through the whole table in the worst case, so it has the same exponential cost as the other operations.

The problem with the table-based approach is that we are explicitly modeling every possible kind of interaction between every possible subset of variables. The probability distributions we encounter in real tasks are much simpler than this. Usually, most variables influence each other only indirectly.

For example, consider modeling the finishing times of a team in a relay race. Suppose the team consists of three runners, Alice, Bob, and Carol. At the start of the race, Alice carries a baton and begins running around a track. After completing her lap around the track, she hands the baton to Bob. Bob then runs his own lap and hands the baton to Carol, who runs the final lap. We can model each of their finishing times as a continuous random variable. Alice's finishing time does not depend on anyone else's, since she goes first. Bob's finishing time depends on Alice's, because Bob does not have the opportunity to start his lap until Alice has completed hers. If Alice finishes faster, Bob will finish faster, all else being equal. Finally, Carol's finishing time depends on both her teammates. If Alice is slow, Bob will probably finish late too, and Carol will have quite a late starting time and thus is likely to have a late finishing time as well. However, Carol's finishing time depends only *indirectly* on Alice's finishing time via Bob's. If we already know Bob's finishing time, we won't be able to estimate Carol's finishing time better by finding out what Alice's finishing time was. This means we can model the relay race using only two interactions: Alice's effect on Bob, and Bob's effect on Carol. We can omit the third, indirect interaction between Alice and Carol from our model.

Structured probabilistic models provide a formal framework for modeling only direct interactions between random variables. This allows the models to have significantly fewer parameters which can in turn be estimated reliably from less data. These smaller models also have dramatically reduced computation cost in terms of storing the model, performing inference in the model, and drawing samples from the model.

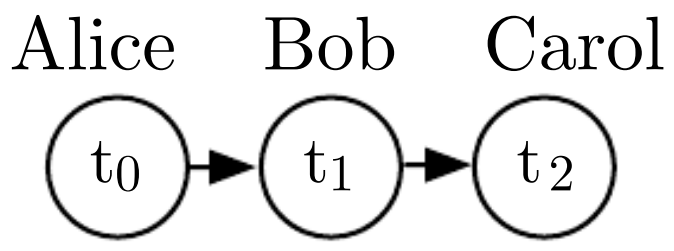


Figure 16.2: A directed graphical model depicting the relay race example. Alice's finishing time $t_0$ influences Bob's finishing time $t_1$, because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob's finishing time $t_1$ influences Carol's finishing time $t_2$.

## 16.2 Using Graphs to Describe Model Structure

Structured probabilistic models use graphs (in the graph theory sense of "nodes" or "vertices" connected by edges) to represent interactions between random variables. Each node represents a random variable. Each edge represents a direct interaction. These direct interactions imply other, indirect interactions, but only the direct interactions need to be explicitly modeled.

There is more than one way to describe the interactions in a probability distribution using a graph. In the following sections we describe some of the most popular and useful approaches.

### 16.2.1 Directed Models

One kind of structured probabilistic model is the *directed graphical model* otherwise known as the *belief network* or *Bayesian network* [2] (Pearl, 1985).

Directed graphical models are called "directed" because their edges are directed, that is, they point from one vertex to another. This direction is represented in the drawing with an arrow. The direction of the arrow indicates which variable's probability distribution is defined in terms of the other's. Drawing an arrow from a to b means that we define the probability distribution over b via a conditional distribution, with a as one of the variables on the right side of the conditioning bar. In other words, the distribution over b depends on the value of a.

Let's continue with the relay race example from Section 16.1. Suppose we name Alice's finishing time $t_0$, Bob's finishing time $t_1$, and Carol's finishing time $t_2$. As we saw earlier, our estimate of $t_1$ depends on $t_0$. Our estimate of $t_2$ depends directly on $t_1$ but only indirectly on $t_0$. We can draw this relationship in a directed graphical model, illustrated in Fig. 16.2.

[2] Judea Pearl suggested using the term Bayes Network when one wishes to "emphasize the judgmental" nature of the values computed by the network, i.e. to highlight that they usually represent degrees of belief rather than frequencies of events.

Formally, a directed graphical model defined on variables $\mathbf{x}$ is defined by a directed acyclic graph $\mathcal{G}$ whose vertices are the random variables in the model, and a set of *local conditional probability distributions* $p(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i))$ where $Pa_{\mathcal{G}}(\mathrm{x}_i)$ gives the parents of $\mathrm{x}_i$ in $\mathcal{G}$. The probability distribution over $\mathbf{x}$ is given by

$$p(\mathbf{x}) = \Pi_i p(\mathrm{x}_i \mid Pa_{\mathcal{G}}(\mathrm{x}_i)).$$

In our relay race example, this means that, using the graph drawn in Fig. 16.2,

$$p(\mathrm{t}_0, \mathrm{t}_1, \mathrm{t}_2) = p(\mathrm{t}_0)p(\mathrm{t}_1 \mid \mathrm{t}_0)p(\mathrm{t}_2 \mid \mathrm{t}_1).$$

This is our first time seeing a structured probabilistic model in action. We can examine the cost of using it, in order to observe how structured modeling has many advantages relative to unstructured modeling.

Suppose we represented time by discretizing time ranging from minute 0 to minute 10 into 6 second chunks. This would make $\mathrm{t}_0$, $\mathrm{t}_1$, and $\mathrm{t}_2$ each be discrete variables with 100 possible values. If we attempted to represent $p(\mathrm{t}_0, \mathrm{t}_1, \mathrm{t}_2)$ with a table, it would need to store 999,999 values (100 values of $\mathrm{t}_0 \times 100$ values of $\mathrm{t}_1 \times$ 100 values of $\mathrm{t}_2$, minus 1, since the probability of one of the configurations is made redundant by the constraint that the sum of the probabilities be 1). If instead, we only make a table for each of the conditional probability distributions, then the distribution over $\mathrm{t}_0$ requires 99 values, the table defining $\mathrm{t}_1$ given $\mathrm{t}_0$ requires 9900 values, and so does the table defining $\mathrm{t}_2$ and $\mathrm{t}_1$. This comes to a total of 19,899 values. This means that using the directed graphical model reduced our number of parameters by a factor of more than 50!

In general, to model $n$ discrete variables each having $k$ values, the cost of the single table approach scales like $O(k^n)$, as we've observed before. Now suppose we build a directed graphical model over these variables. If $m$ is the maximum number of variables appearing (on either side of the conditioning bar) in a single conditional probability distribution, then the cost of the tables for the directed model scales like $O(k^m)$. As long as we can design a model such that $m << n$, we get very dramatic savings.

In other words, so long as each variable has few parents in the graph, the distribution can be represented with very few parameters. Some restrictions on the graph structure (e.g. it is a tree) can also guarantee that operations like computing marginal or conditional distributions over subsets of variables are efficient.

It's important to realize what kinds of information can be encoded in the graph, and what can't be. The graph just encodes simplifying assumptions about which variables are conditionally independent from each other. It's also possible to make other kinds of simplifying assumptions. For example, suppose we assume Bob always runs the same regardless of how Alice performed. (In reality, Alice's

performance probably influences Bob's performance–depending on Bob's personality, if Alice runs especially fast in a given race, this might encourage Bob to push hard and match her exceptional performance, or it might make him overconfident and lazy). Then the only effect Alice has on Bob's finishing time is that we must add Alice's finishing time to the total amount of time we think Bob needs to run. This observation allows us to define a model with $O(k)$ parameters instead of $O(k^2)$. However, note that $\mathrm{t}_0$ and $\mathrm{t}_1$ are still directly dependent with this assumption, because $\mathrm{t}_1$ represents the absolute time at which Bob finishes, not the total time he himself spends running. This means our graph must still contain an arrow from $\mathrm{t}_0$ to $\mathrm{t}_1$. The assumption that Bob's personal running time is independent from all other factors cannot be encoded in a graph over $\mathrm{t}_0$, $\mathrm{t}_1$, and $\mathrm{t}_2$. Instead, we encode this information in the definition of the conditional distribution itself. The conditional distribution is no longer a $k \times k - 1$ element table indexed by $\mathrm{t}_0$ and $\mathrm{t}_1$ but is now a slightly more complicated formula using only $k - 1$ parameters. The directed graphical model syntax does not place any constraint on how we define our conditional distributions. It only defines which variables they are allowed to take in as arguments.

### 16.2.2 Undirected Models

Directed graphical models give us one language for describing structured probabilistic models. Another popular language is that of *undirected models*, otherwise known as *Markov random fields* (MRFs) or *Markov networks* (Kindermann, 1980). As their name implies, undirected models use graphs whose edges are undirected.

Directed models are most naturally applicable to situations where there is a clear reason to draw each arrow in one particular direction. Often these are situations where we understand the causality, and the causality only flows in one direction. One such situation is the relay race example. Earlier runners affects the finishing times of later runners; later runners do not affect the finishing times of earlier runners.

Not all situations we might want to model have such a clear direction to their interactions. When the interactions seem to have no intrinsic direction, or to operate in both directions, it may be more appropriate to use an undirected model.

As an example of such a situation, suppose we want to model a distribution over three binary variables: whether or not you are sick, whether or not your coworker is sick, and whether or not your roommate is sick. As in the relay race example, we can make simplifying assumptions about the kinds of interactions that take place. Assuming that your coworker and your roommate do not know each other, it is very unlikely that one of them will give the other a disease such as a cold directly. This event can be seen as so rare that it is acceptable not to model

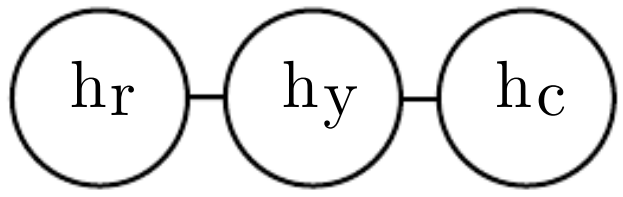


Figure 16.3: An undirected graph representing how your roommate's health $h_r$, your health $h_y$, and your work colleague's health $h_c$ affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague don't know each other, they can only infect each other indirectly via you.

it. However, it is reasonably likely that either of them could give you a cold, and that you could pass it on to the other. We can model the indirect transmission of a cold from your coworker to your roommate by modeling the transmission of the cold from your coworker to you and the transmission of the cold from you to your roommate.

In this case, it's just as easy for you to cause your roommate to get sick as it is for your roommate to make you sick, so there is not a clean, uni-directional narrative on which to base the model. This motivates using an undirected model. As with directed models, if two nodes in an undirected model are connected by an edge, then the random variables corresponding to those nodes interact with each other directly. Unlike directed models, the edge in an undirected model has no arrow, and is not associated with a conditional probability distribution.

Let's call the random variable representing your health $h_y$, the random variable representing your roommate's health $h_r$, and the random variable representing your colleague's health $h_c$. See Fig. 16.3 for a drawing of the graph representing this scenario.

Formally, an undirected graphical model is a structured probabilistic model defined on an undirected graph $\mathcal{G}$. For each clique $\mathcal{C}$ in the graph[3], a *factor* $\phi(\mathcal{C})$ (also called a *clique potential*) measures the affinity of the variables in that clique for being in each of their possible joint states. The factors are constrained to be non-negative. Together they define an *unnormalized probability distribution*

$$\tilde{p}(\mathbf{x}) = \Pi_{\mathcal{C}\in\mathcal{G}}\phi(\mathcal{C}).$$

The unnormalized probability distribution is efficient to work with so long as all the cliques are small. It encodes the idea that states with higher affinity are more likely. However, unlike in a Bayesian network, there is little structure to the definition of the cliques, so there is nothing to guarantee that multiplying them together will yield a valid probability distribution. See Fig. 16.4 for an example

[3] A clique of the graph is a subset of nodes that are all connected to each other by an edge of the graph.

of reading factorization information from an undirected graph.

Our example of the cold spreading between you, your roommate, and your colleague contains two cliques. One clique contains $\mathrm{h}_y$ and $\mathrm{h}_c$. The factor for this clique can be defined by a table, and might have values resembling these:

| | $\mathrm{h}_y = 0$ | $\mathrm{h}_y = 1$ |
|---|---|---|
| $\mathrm{h}_c = 0$ | 2 | 1 |
| $\mathrm{h}_c = 1$ | 1 | 10 |

A state of 1 indicates good health, while a state of 0 indicates poor health (having been infected with a cold). Both of you are usually healthy, so the corresponding state has the highest affinity. The state where only one of you is sick has the lowest affinity, because this is a rare state. The state where both of you are sick (because one of you has infected the other) is a higher affinity state, though still not as common as the state where both are healthy.

To complete the model, we would need to also define a similar factor for the clique containing $\mathrm{h}_y$ and $\mathrm{h}_r$.

### 16.2.3 The Partition Function

While the unnormalized probability distribution is guaranteed to be non-negative everywhere, it is not guaranteed to sum or integrate to 1. To obtain a valid probability distribution, we must use the corresponding normalized probability distribution[4]:

$$p(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x})$$

where $Z$ is the value that results in the probability distribution summing or integrating to 1:

$$Z = \int \tilde{p}(\mathbf{x})d\mathbf{x}.$$

You can think of $Z$ as a constant when the $\phi$ functions are held constant. Note that if the $\phi$ functions have parameters, then $Z$ is a function of those parameters. It is common in the literature to write $Z$ with its arguments omitted to save space. $Z$ is known as the *partition function*, a term borrowed from statistical physics.

Since $Z$ is an integral or sum over all possible joint assignments of the state $\mathbf{x}$ it is often intractable to compute. In order to be able to obtain the normalized probability distribution of an undirected model, the model structure and the definitions of the $\phi$ functions must be conducive to computing $Z$ efficiently.

[4] A distribution defined by normalizing a product of clique potentials is also called a *Gibbs distribution*.

In the context of deep learning, $Z$ is usually intractable, and we must resort to approximations. Such approximate algorithms are the topic of Chapter 18.

One important consideration to keep in mind when designing undirected models is that it is possible for $Z$ not to exist. This happens if some of the variables in the model are continuous and the integral of $\tilde{p}$ over their domain diverges. For example, suppose we want to model a single scalar variable $\mathrm{x} \in \mathbb{R}$ with a single clique potential $\phi(x) = x^2$. In this case,

$$Z = \int x^2 dx.$$

Since this integral diverges, there is no probability distribution corresponding to this choice of $\phi(x)$. Sometimes the choice of some parameter of the $\phi$ functions determines whether the probability distribution is defined. For example, for $\phi(x; \beta) = \exp\left(-\beta x^2\right)$, the $\beta$ parameter determines whether $Z$ exists. Positive $\beta$ results in a Gaussian distribution over x but all other values of $\beta$ make $\phi$ impossible to normalize.

One key difference between directed modeling and undirected modeling is that directed models are defined directly in terms of probability distributions from the start, while undirected models are defined more loosely by $\phi$ functions that are then converted into probability distributions. This changes the intuitions one must develop in order to work with these models. One key idea to keep in mind while working with undirected models is that the domain of each of the variables has dramatic effect on the kind of probability distribution that a given set of $\phi$ functions corresponds to. For example, consider an $n$-dimensional vector-valued random variable $\mathbf{x}$ and an undirected model parameterized by a vector of biases $\boldsymbol{b}$. Suppose we have one clique for each element of $\mathbf{x}$, $\phi_i(\mathrm{x}_i) = \exp(b_i \mathrm{x}_i)$. What kind of probability distribution does this result in? The answer is that we don't have enough information, because we have not yet specified the domain of $\mathbf{x}$. If $\mathbf{x} \in \mathbb{R}^n$, then the integral defining $Z$ diverges and no probability distribution exists. If $\mathbf{x} \in \{0, 1\}^n$, then $p(\mathbf{x})$ factorizes into $n$ independent distributions, with $p(\mathrm{x}_i = 1) = \mathrm{sigmoid}\,(b_i)$. If the domain of $\mathbf{x}$ is the set of elementary basis vectors $(\{[1, 0, \ldots, 0], [0, 1, \ldots, 0], \ldots, [0, 0, \ldots, 1]\}\,)$ then $p(\mathbf{x}) = \mathrm{softmax}\,(b)$, so a large value of $b_i$ actually reduces $p(\mathrm{x}_j = 1)$ for $j \neq i$. Often, it is possible to leverage the effect of a carefully chosen domain of a variable in order to obtain complicated behavior from a relatively simple set of $\phi$ functions. We'll explore a practical application of this idea later, in Chapter 20.7.

### 16.2.4 Energy-Based Models

Many interesting theoretical results about undirected models depend on the assumption that $\forall \mathbf{x}, \tilde{p}(\mathbf{x}) > 0$. A convenient way to enforce this to use an *energy-*

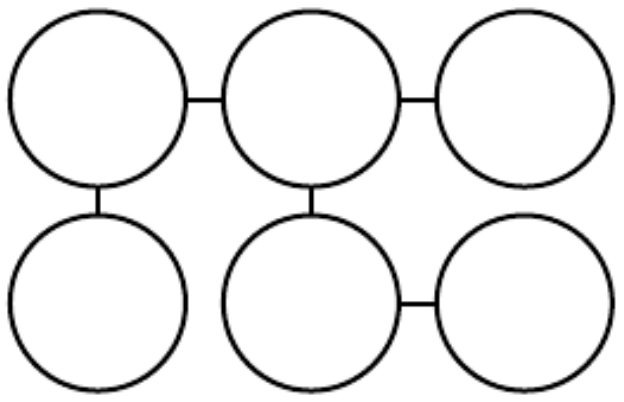

Figure 16.4: This graph implies that $p(A, B, C, D, E, F)$ can be written as $\frac{1}{Z}\phi_{A,B}(A, B)\phi_{B,C}(B, C)\phi_{A,D}(A, D)\phi_{B,E}(B, E)\phi_{E,F}(E, F)$ for an appropriate choice of the $\phi$ functions.

*based model* (EBM) where

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \tag{16.1}$$

and $E(\mathbf{x})$ is known as the *energy function*. Because $\exp(z)$ is positive for all $z$, this guarantees that no energy function will result in a probability of zero for any state $\mathbf{x}$ Being completely free to choose the energy function makes learning simpler. If we learned the clique potentials directly, we would need to use constrained optimization, and we would need to arbitrarily impose some specific minimal probability value. By learning the energy function, we can use unconstrained optimization[5], and the probabilities in the model can approach arbitrarily close to zero but never reach it.

Any distribution of the form given by equation 16.1 is an example of a *Boltzmann distribution*. For this reason, many energy-based models are called *Boltzmann machines*. There is no accepted guideline for when to call a model an energy-based model and when to call it a Boltzmann machines. The term Boltzmann machine was first introduced to describe a model with exclusively binary variables, but today many models such as the mean-covariance restricted Boltzmann machine incorporate real-valued variables as well.

Cliques in an undirected graph correspond to factors of the unnormalized probability function. Because $\exp(a)\exp(b) = \exp(a + b)$, this means that different cliques in the undirected graph correspond to the different terms of the energy function. In other words, an energy-based model is just a special kind of Markov network: the exponentiation makes each term in the energy function correspond to a factor for a different clique. See Fig. 16.5 for an example of how to read the form of the energy function from an undirected graph structure.

One part of the definition of an energy-based model serves no functional purpose from a machine learning point of view: the $-$ sign in Eq. 16.1. This $-$ sign could be incorporated into the definition of $E$, or for many functions $E$ the learning algorithm could simply learn parameters with opposite sign. The $-$ sign

[5] For some models, we may still need to use constrained optimization to make sure exists.

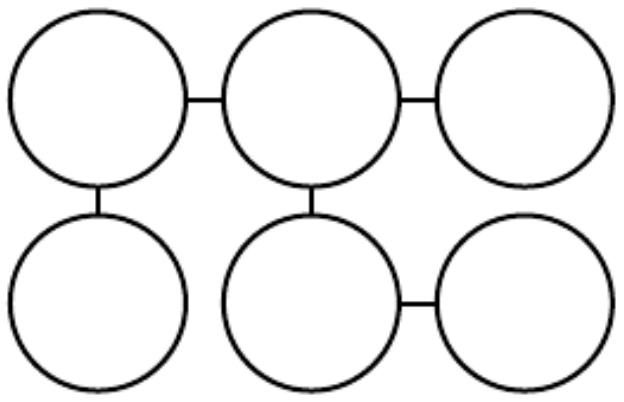

Figure 16.5: This graph implies that $E(\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f})$ can be written as $E_{\mathrm{a},\mathrm{b}}(\mathrm{a},\mathrm{b}) + E_{\mathrm{b},\mathrm{c}}(\mathrm{b},\mathrm{c}) + E_{\mathrm{a},\mathrm{d}}(\mathrm{a},\mathrm{d}) + E_{\mathrm{b},\mathrm{e}}(\mathrm{b},\mathrm{e}) + E_{\mathrm{e},\mathrm{f}}(\mathrm{e},\mathrm{f})$ for an appropriate choice of the per-clique energy functions. Note that we can obtain the $\phi$ functions in Fig. 16.4 by setting each $\phi$ to the exp of the corresponding negative energy, e.g., $\phi_{\mathrm{a},\mathrm{b}}(\mathrm{a},\mathrm{b}) = \exp(-E(\mathrm{a},\mathrm{b}))$.

is present primarily to preserve compatibility between the machine learning literature and the physics literature. Many advances in probabilistic modeling were originally developed by statistical physicists, for whom $E$ refers to actual, physical energy and does not have arbitrary sign. Terminology such as "energy" and "partition function" remains associated with these techniques, even though their mathematical applicability is broader than the physics context in which they were developed. Some machine learning researchers (e.g., Smolensky (1986), who referred to negative energy as *harmony*) have chosen to emit the negation, but this is not the standard convention.

### 16.2.5 Separation and D-Separation

The edges in a graphical model tell us which variables directly interact. We often need to know which variables *indirectly* interact. Some of these indirect interactions can be enabled or disabled by observing other variables. More formally, we would like to know which subsets of variables are conditionally independent from each other, given the values of other subsets of variables.

Identifying the conditional independences in a graph is very simple in the case of undirected models. In this case, conditional independence implied by the graph is called *separation*. We say that a set of variables $\mathbb{A}$ is *separated* from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$. If two variables a and b are connected by a path involving only unobserved variables, then those variables are not separated. If no path exists between them, or all paths contain an observed variable, then they are separated. We refer to paths involving only unobserved variables as "active" and paths including an observed variable as "inactive."

When we draw a graph, we can indicate observed variables by shading them in. See Fig. 16.6 for a depiction of how active and inactive paths in an undirected look when drawn in this way. See Fig. 16.7 for an example of reading separation

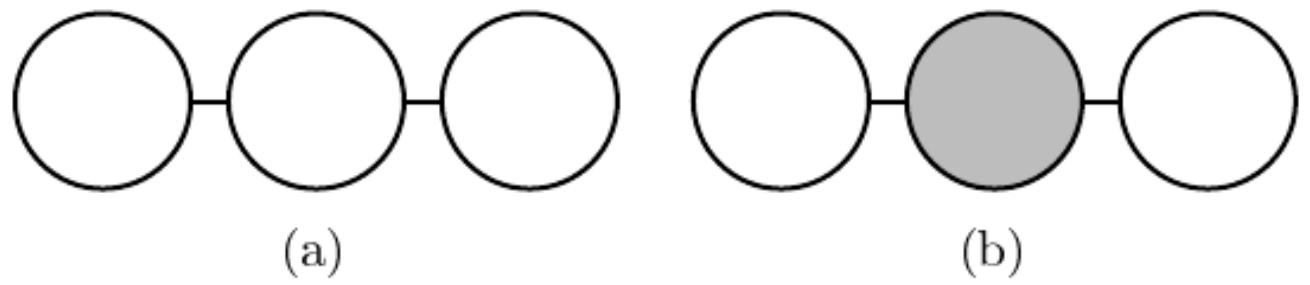


Figure 16.6: a) The path between random variable a and random variable b through s is active, because s is not observed. This means that a and b are not separated. b) Here s is shaded in, to indicate that it is observed. Because the only path between $ra$ and b is through s, and that path is inactive, we can conclude that a and b are separated given s.

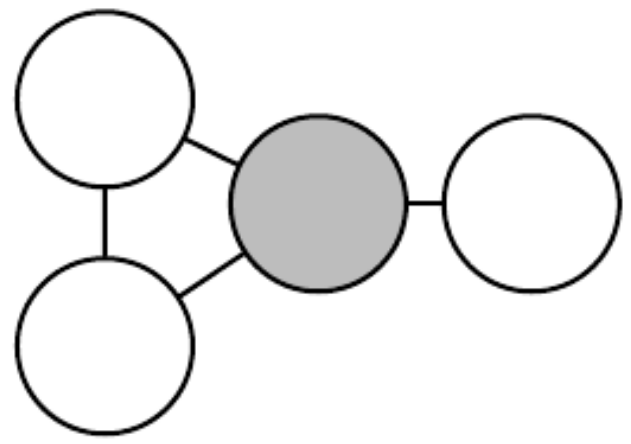

Figure 16.7: An example of reading separation properties from an undirected graph. Here b is shaded to indicate that it is observed. Because observing b blocks the only path from a to c, we say that a and c are separated from each other given b. The observation of b also blocks one path between a and d, but there is a second, active path between them. Therefore, a and d are not separated given b.

from an undirected graph.

Similar concepts apply to directed models, except that in the context of directed models, these concepts are referred to as *d-separation*. The "d" stands for "dependence." D-separation for directed graphs is defined the same as separation for undirected graphs: We say that a set of variables $\mathbb{A}$ is d-separated from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$.

As with undirected models, we can examine the independences implied by the graph by looking at what active paths exist in the graph. As before, two variables are dependent if there is an active path between them, and d-separated if no such path exists. In directed nets, determining whether a path is active is somewhat more complicated. See Fig. 16.8 for a guide to identifying active paths in a directed model. See Fig. 16.9 for an example of reading some properties from a graph.

It is important to remember that separation and d-separation tell us only about those conditional independences *that are implied by the graph.* There is no requirement that the graph imply all independences that are present. In particular, it is always legitimate to use the complete graph (the graph with all possible edges)

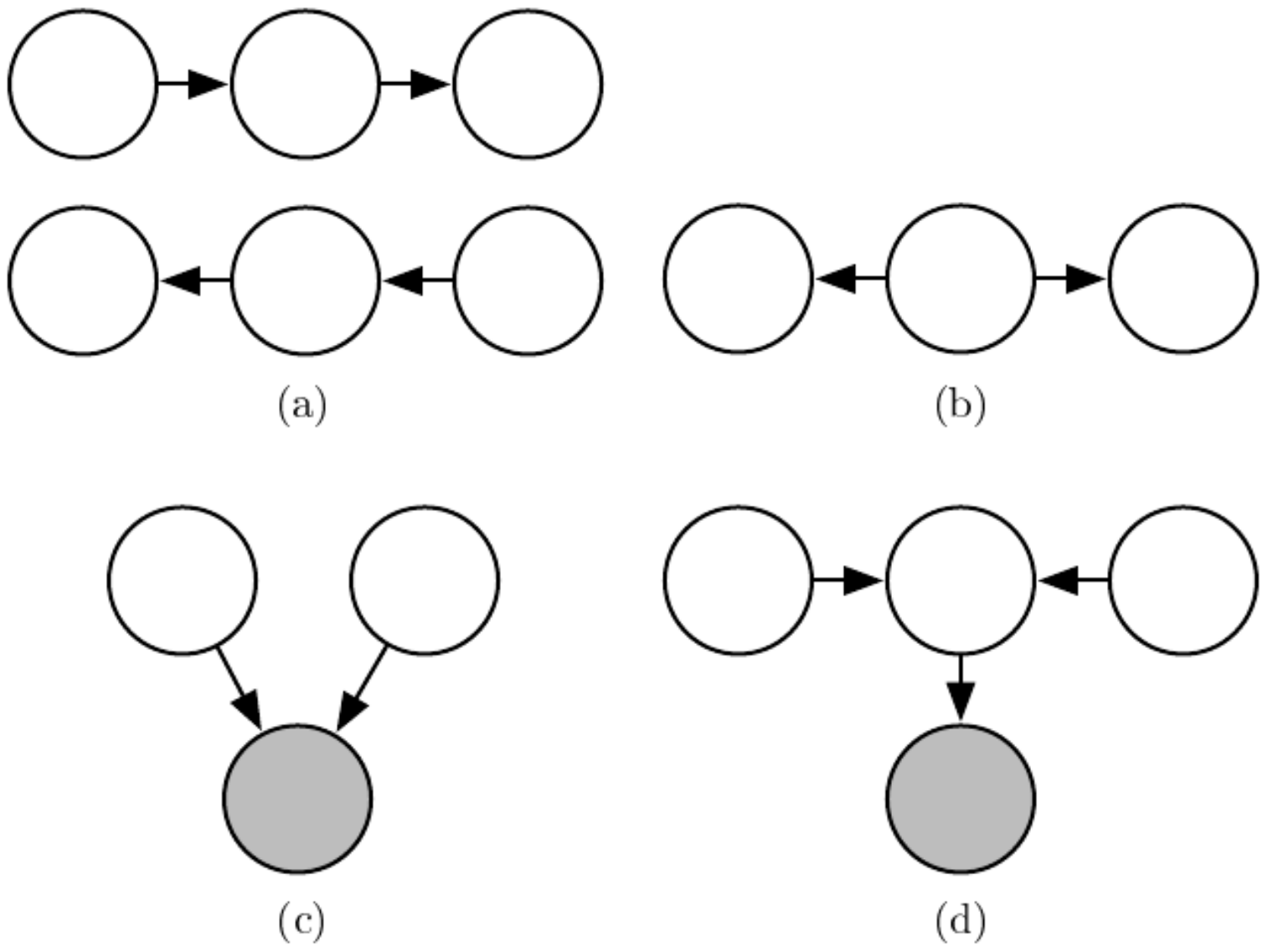


Figure 16.8: All of the kinds of active paths of length two that can exist between random variables a and $rb$. a) Any path with arrows proceeding directly from a to b or vice versa. This kind of path becomes blocked if s is observed. We have already seen this kind of path in the relay race example. b) a and b are connected by a *common cause* s. For example, suppose s is a variable indicating whether or not there is a hurricane and a and b measure the wind speed at two different nearby weather monitoring outposts. If we observe very high winds at station a, we might expect to also see high winds at b. This kind of path can be blocked by observing s. If we already know there is a hurricane, we expect to see high winds at b, regardless of what is observed at a. A lower than expected wind at a (for a hurricane) would not change our expectation of winds at b (knowing there is a hurricane). However, if s is not observed, then a and b are dependent, i.e., the path is inactive. c) a and b are both parents of s. This is called a *V-structure* or *the collider case*, and it causes a and a to be related by the *explaining away effect*. In this case, the path is actually active when s is observed. For example, suppose s is a variable indicating that your colleague is not at work. The variable a represents her being sick, while b represents her being on vacation. If you observe that she is not at work, you can presume she is probably sick or on vacation, but it's not especially likely that both have happened at the same time. If you find out that she is on vacation, this fact is sufficient to *explain* her absence, and you can infer that she is probably not also sick. d) The explaining away effect happens even if any descendant of s is observed! For example, suppose that c is a variable representing whether you have received a report from your colleague. If you notice that you have not received the report, this increases your estimate of the probability that she is not at work today, which in turn makes it more likely that she is either sick or on vacation. The only way to block a path through a V-structure is to observe none of the descendants of the shared child.

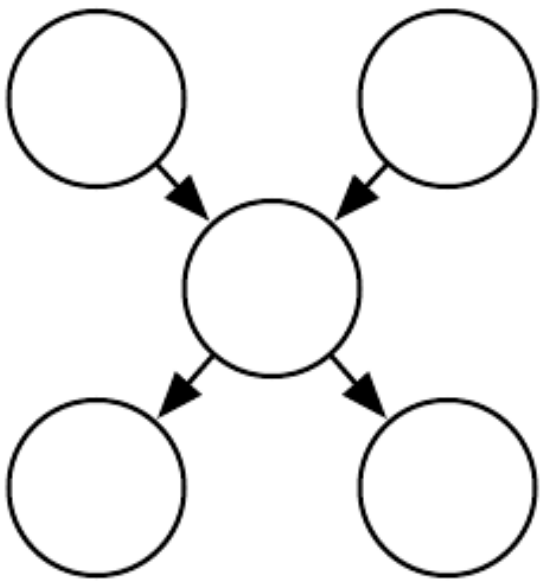

Figure 16.9

From this graph, we can read out several d-separation properties. Examples include:

- a and b are d-separated given the empty set.
- a and e are d-separated given c.
- d and e are d-separated given c.

We can also see that some variables are no longer d-separated when we observe some variables:

- a and b are not d-separated given c.
- a and b are not d-separated given d.

to represent any distribution. In fact, some distributions contain independences that are not possible to represent with existing graphical notation. *Context-specific independences* are independences that are present dependent on the value of some variables in the network. For example, consider a model of three binary variables, a, b, and c. Suppose that when a is 0, b and c are independent, but when a is 1, b is deterministically equal to c. Encoding the behavior when $a = 1$ requires an edge connecting b and c. The graph then fails to indicate that b and c are independent when $a = 0$.

In general, a graph will never imply that an independence exists when it does not. However, a graph may fail to encode an independence.

### 16.2.6 Converting between Undirected and Directed Graphs

In common parlance, we often refer to certain model classes as being undirected or directed. For example, we typically refer to RBMs as undirected and sparse coding as directed. This way of speaking can be somewhat misleading, because no probabilistic model is inherently directed or undirected. Instead, some models are most easily *described* using a directed graph, or most easily described using an undirected graph.

Every probability distribution can be represented by either a directed model or by an undirected model. In the worst case, one can always represent any distribution by using a "complete graph." In the case of a directed model, the complete graph is any directed acyclic graph where we impose some ordering on the random variables, and each variable has all other variables that precede it in the ordering as its ancestors in the graph. For an undirected model, the complete graph is simply a graph containing a single clique encompassing all of the variables.

Of course, the utility of a graphical model is that the graph implies that some variables do not interact directly. The complete graph is not very useful because it does not imply any independences. TODO figure complete graph

When we represent a probability distribution with a graph, we want to choose a graph that implies as many independences as possible, without implying any independences that do not actually exist.

From this point of view, some distributions can be represented more efficiently using directed models, while other distributions can be represented more efficiently using undirected models. In other words, directed models can encode some independences that undirected models cannot encode, and vice versa.

Directed models are able to use one specific kind of substructure that undirected models cannot represent perfectly. This substructure is called an *immorality*. The structure occurs when two random variables a and b are both parents of a third random variable c, and there is no edge directly connecting a and b in either direction. (The name "immorality" may seem strange; it was coined in

the graphical models literature as a joke about unmarried parents) To convert a directed model with graph $\mathcal{D}$ into an undirected model, we need to create a new graph $\mathcal{U}$. For every pair of variables x and y, we add an undirected edge connecting x and y to $\mathcal{U}$ if there is a directed edge (in either direction) connecting x and y in $\mathcal{D}$ or if x and y are both parents in $\mathcal{D}$ of a third variable z. The resulting $\mathcal{U}$ is known as a *moralized graph*. See Fig. 16.10 for examples of converting directed models to undirected models via moralization.

Likewise, undirected models can include substructures that no directed model can represent perfectly. Specifically, a directed graph $\mathcal{D}$ cannot capture all of the conditional independences implied by an undirected graph $\mathcal{U}$ if $\mathcal{U}$ contains a *loop* of length greater than three, unless that loop also contains a *chord*. A loop is a sequence of variables connected by undirected edges, with the last variable in the sequence connected back to the first variable in the sequence. A chord is a connection between any two non-consecutive variables in this sequence. If $\mathcal{U}$ has loops of length four or greater and does not have chords for these loops, we must add the chords before we can convert it to a directed model. Adding these chords discards some of the independence information that was encoded in $\mathcal{U}$. The graph formed by adding chords to $\mathcal{U}$ is known as a *chordal* or *triangulated* graph, because all the loops can now be described in terms of smaller, triangular loops. To build a directed graph $\mathcal{D}$ from the chordal graph, we need to also assign directions to the edges. When doing so, we must not create a directed cycle in $\mathcal{D}$, or the result does not define a valid directed probabilistic model. One way to assign directions to the edges in $\mathcal{D}$ is to impose an ordering on the random variables, then point each edge from the node that comes earlier in the ordering to the node that comes later in the ordering. TODO point to fig

IG HERE

TODO: started this above, need to scrap some some BNs encode independences that MNs can't encode, and vice versa example of BN that an MN can't encode: A and B are parents of C A is d-separated from B given the empty set The Markov net requires a clique over A, B, and C in order to capture the active path from A to B when C is observed This clique means that the graph cannot imply A is separated from B given the empty set example of a MN that a BN can't encode: A, B, C, D connected in a loop BN cannot have both A d-sep D given B, C and B d-sep C given A, D

In many cases, we may want to convert an undirected model to a directed model, or vice versa. To do so, we choose the graph in the new format that implies as many independences as possible, while not implying any independences that were not implied by the original graph.

To convert a directed model $\mathcal{D}$ to an undirected model $\mathcal{U}$, we re

TODO: conversion between directed and undirected models

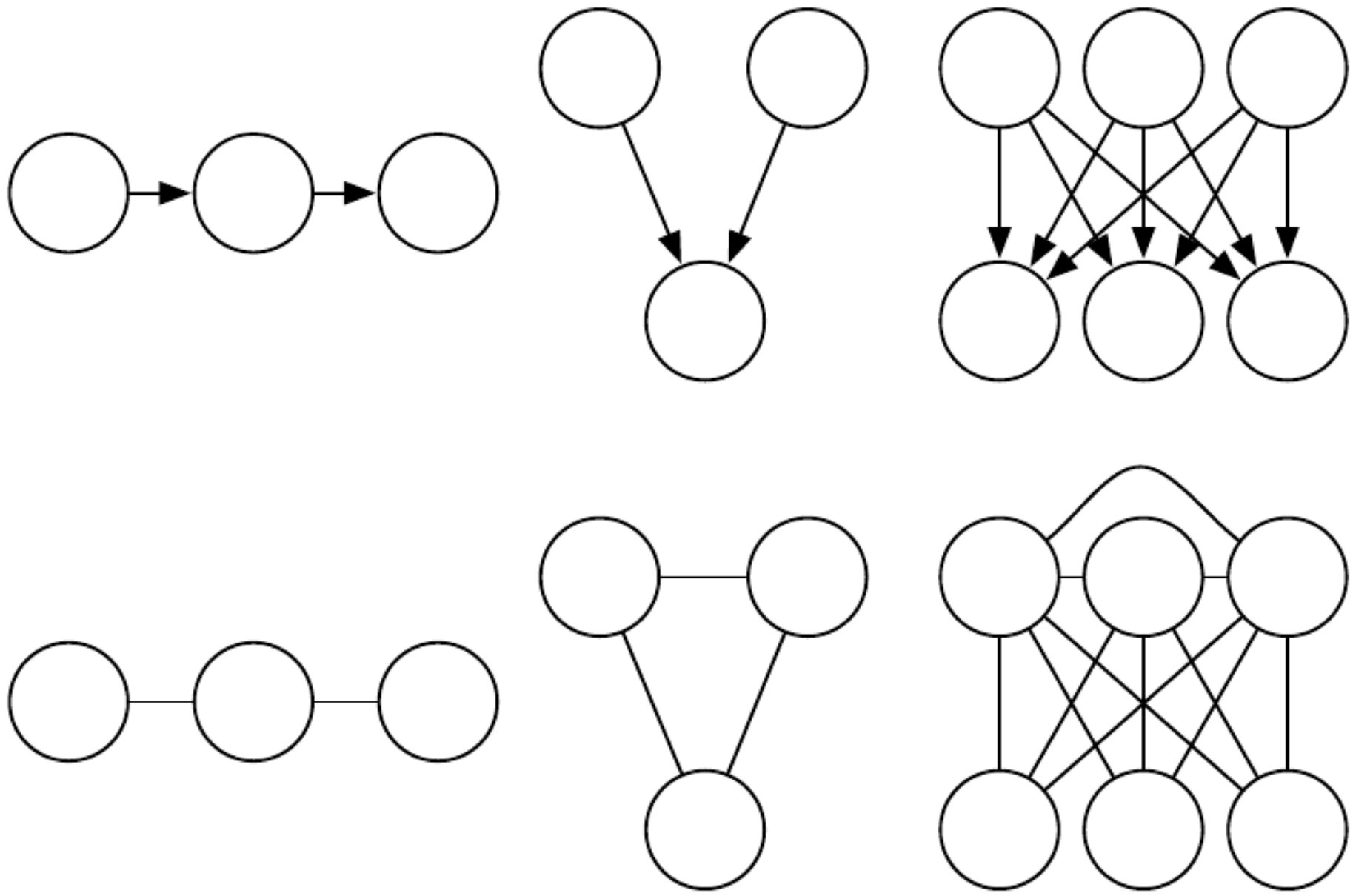

Figure 16.10: Examples of converting directed models to undirected models by constructing moralized graphs. *Left)* This simple chain can be converted to a moralized graph merely by replacing its directed edges with undirected edges. The resulting undirected model implies exactly the same set of independences and conditional independences. *Center)* This graph is the simplest directed model that cannot be converted to an undirected model without losing some independences. This graph consists entirely of a single immorality. Because a and b are parents of c, they are connected by an active path when c is observed. To capture this dependence, the undirected model must include a clique encompassing all three variables. This clique fails to encode the fact that $a \perp b$. *Right)* In general, moralization may add many edges to the graph, thus losing many implied independences. For example, this sparse coding graph requires adding moralizing edges between every pair of hidden units, thus introducing a quadratic number of new direct dependences.

### 16.2.7 Marginalizing Variables out of a Graph

TODO: marginalizing variables out of a graph

### 16.2.8 Factor Graphs

*Factor graphs* are another way of drawing undirected models that resolve an ambiguity in the graphical representation of standard undirected model syntax. In an undirected model, the scope of every $\phi$ function must be a subset of some clique in the graph. However, it is not necessary that there exist any $\phi$ whose scope contains the entirety of every clique. Factor graphs explicitly represent the scope of each $\phi$ function. Specifically, a factor graph is a graphical representation of an undirected model that consists of a bipartite undirected graph. Some of the nodes are drawn as circles. These nodes correspond to random variables as in a standard undirected model. The rest of the nodes are drawn as squares. These nodes correspond to the factors $\phi$ of the unnormalized probability distribution. Variables and factors may be connected with undirected edges. A variable and a factor are connected in the graph if and only if the variable is one of the arguments to the factor in the unnormalized probability distribution. No factor may be connected to another factor in the graph, nor can a variable be connected to a variable. See Fig. 16.11 for an example of how factor graphs can resolve ambiguity in the interpretation of undirected networks.

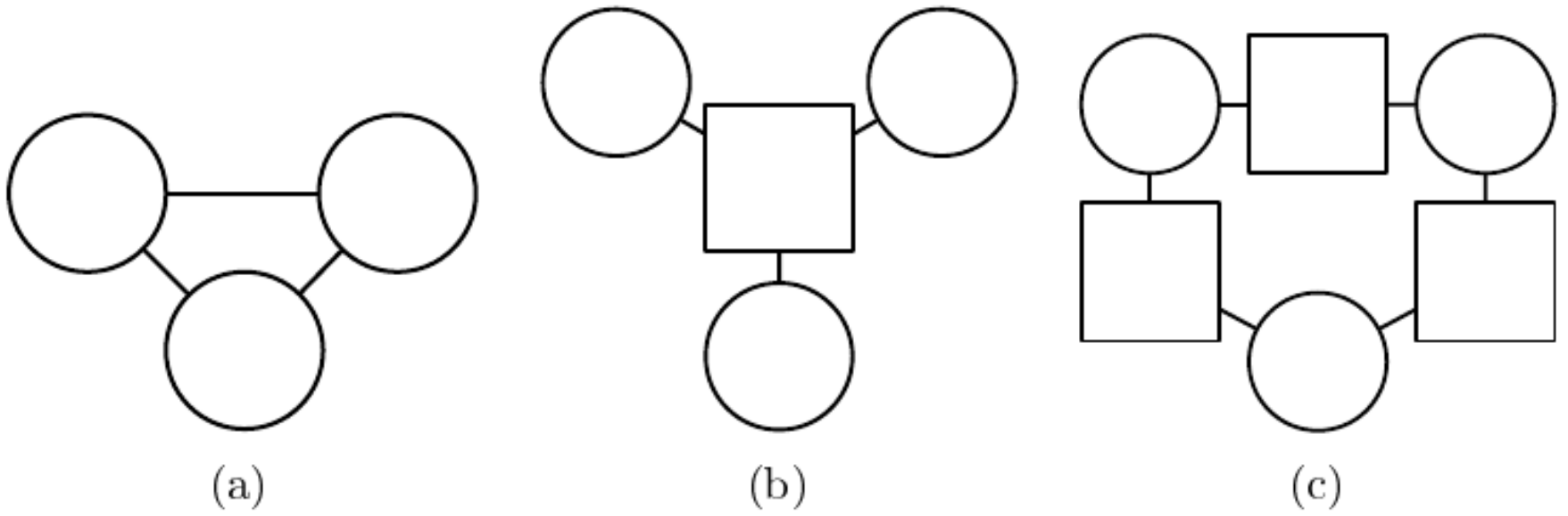


Figure 16.11: An example of how a factor graph can resolve ambiguity in the interpretation of undirected networks. a) An undirected network with a clique involving three variables a, b, and c. b) A factor graph corresponding to the same undirected model. This factor graph has one factor over all three variables. c) Another valid factor graph for the same undirected model. This factor graph has three factors, each over only two variables. Note that representation, inference, and learning are all asymptotically cheaper in (c) compared to (b), even though both require the same undirected graph to represent.

## 16.3 Advantages of Structured Modeling

TODO– note that we have already shown that some things are cheaper in the sections where we introduce the modeling syntax

TODO: revisit each of the three challenges from sec:unstructured TODO: hammer point that graphical models convey information by leaving edges out TODO: need to show reduced cost of sampling, but first reader needs to know about ancestral and gibbs sampling.... TODO: benefit of separating representation from learning and inference

## 16.4 Learning about Dependencies

We describe here two types of random variables: observed or "visible" variables $\mathbf{v}$ and latent or "hidden" variables $\mathbf{h}$. The observed variables $\mathbf{v}$ correspond to the variables actually provided in the data set during training. $\mathbf{h}$ consists of variables that are introduced to the model in order to help it explain the structure in $\mathbf{v}$. Generally the exact semantics of $\mathbf{h}$ depend on the model parameters and are created by the learning algorithm. The motivation for this is twofold.

### 16.4.1 Latent Variables Versus Structure Learning

Often the different elements of $\mathbf{v}$ are highly dependent on each other. A good model of $\mathbf{v}$ which did not contain any latent variables would need to have very large numbers of parents per node in a Bayesian network or very large cliques in a Markov network. Just representing these higher order interactions is costly–both in a computational sense, because the number of parameters that must be stored in memory scales exponentially with the number of members in a clique, but also in a statistical sense, because this exponential number of parameters requires a wealth of data to estimate accurately.

There is also the problem of learning which variables need to be in such large cliques. An entire field of machine learning called *structure learning* is devoted to this problem . For a good reference on structure learning, see (Koller and Friedman, 2009). Most structure learning techniques are a form of greedy search. A structure is proposed, a model with that structure is trained, then given a score. The score rewards high training set accuracy and penalizes model complexity. Candidate structures with a small number of edges added or removed are then proposed as the next step of the search, and the search proceeds to a new structure that is expected to increase the score.

Using latent variables instead of adaptive structure avoids the need to perform discrete searches and multiple rounds of training. A fixed structure over visible and hidden variables can use direct interactions between visible and hidden units

to impose indirect interactions between visible units. Using simple parameter learning techniques we can learn a model with a fixed structure that imputes the right structure on the marginal $p(\mathrm{v})$.

### 16.4.2 Latent Variables for Feature Learning

Another advantage of using latent variables is that they often develop useful semantics.

As discussed in section 3.10.6, the mixture of Gaussians model learns a latent variable that corresponds to which category of examples the input was drawn from. This means that the latent variable in a mixture of Gaussians model can be used to do classification.

In Chapter 13 we saw how simple probabilistic models like sparse coding learn latent variables that can be used as input features for a classifier, or as coordinates along a manifold. Other models can be used in this same way, but deeper models and models with different kids of interactions can create even richer descriptions of the input. Most of the approaches mentioned in sec. 16.4.2 accomplish feature learning by learning latent variables. Often, given some model of $\mathbf{v}$ and $\mathbf{h}$, it turns out that $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ TODO: uh-oh, is there a collision between set notation and expectation notation? or argmax $p(\boldsymbol{h}, \boldsymbol{v})$ is a good feature mapping for $\boldsymbol{v}$.

TODO: appropriate links to Monte Carlo methods chapter spun off from here

## 16.5 Inference and Approximate Inference over Latent Variables

As soon as we introduce latent variables in a graphical model, this raises the question: how to choose values of the latent variables $\boldsymbol{h}$ given values of the visible variables $\boldsymbol{x}$? This is what we call *inference*, in particular inference over the latent variables. The general question of inference is to guess some variables given others.

TODO: inference has definitely been introduced above... TODO: mention loopy BP, show how it is very expensive for DBMs

TODO: briefly explain what variational inference is and reference approximate inference chapter

### 16.5.1 Reparametrization Trick

Sometimes, in order to estimate the stochastic gradient of an expected loss over some random variable $\mathbf{h}$, with respect to parameters that influence $\mathbf{h}$, we would like to compute gradients through $\mathbf{h}$, i.e., on the parameters that influenced the probability distribution from which $\mathbf{h}$ was sampled. If $\mathbf{h}$ is continuous-valued, this

is generally possible by using the *reparametrization trick*, i.e., rewriting

$$\mathbf{h} \sim p(\mathbf{h} \mid \theta) \tag{16.2}$$

as

$$\mathbf{h} = f(\theta, \eta) \tag{16.3}$$

where $\eta$ is some independent noise source of the appropriate dimension with density $p(\eta)$, and $f$ is a continuous (differentiable almost everywhere) function. The reparametrization trick is the idea that if the random variable to be integrated over is continuous, we can *back-propagate* through the process that gave rise to it in order to figure how to change that process.

For example, let us suppose we want to estimate the expected gradient

$$\frac{\partial}{\partial \theta} \int L(\mathbf{h}) p(\mathbf{h} \mid \theta) d\mathbf{h} \tag{16.4}$$

where the parameters $\theta$ influences the random variable $\mathbf{h}$ which in turn influence our loss $L$. A very efficient (Kingma and Welling, 2014b; Rezende *et al.*, 2014) way to achieve[6] this is to perform the reparametrization in Eq. 16.3 and the corresponding change of variable in the integral of Eq. 16.4, integrating over $\eta$ rather than $\mathbf{h}$:

$$\frac{\partial}{\partial \theta} \int L(f(\theta, \eta)) p(eta) d\eta. \tag{16.5}$$

We can now more easily enter the derivative in the integral, getting

$$g = \int \frac{\partial L(f(\theta, \eta))}{\partial \theta} p(eta) d\eta.$$

Finally, we get a stochastic gradient estimator

$$\hat{g} = \frac{\partial L(f(\theta, \eta))}{\partial \theta}$$

where we sampled $\eta \sim p(\eta)$ and $E[\hat{g}] = g$.

This trick was first introduced to machine learning by Williams (1992) in the context of back-propagation. The idea of back-propagating through a sampling process was independently re-discovered and re-popularized in the context of deep generative models more recently by Bengio (2013b); Bengio *et al.* (2013a); Kingma (2013). The underlying principle was actually discovered much earlier in the information theory community (Price, 1958; Bonnet, 1964). The theorems from Price (1958); Bonnet (1964) were first used in a variational context by Opper and

[6] compared to approaches that do not back-propagate through the generation of

Archambeau (2009); Challis and Barber (2012) and for efficient Bayesian inference by Salimans and Knowles (2013). It was further developed and analyzed by Kingma and Welling (2014b) and was used to train generative stochastic networks (GSNs) (Bengio *et al.*, 2014a,b), described in Section 20.12, which can be viewed as recurrent networks with noise injected both in input and hidden units (with each time step corresponding to one step of a generative Markov chain). The reparametrization trick was also used to estimate the parameter gradient in variational auto-encoders (Kingma and Welling, 2014a; Rezende *et al.*, 2014; Kingma *et al.*, 2014), which are described in Section 20.9.3.

## 16.6 The Deep Learning Approach to Structured Probabilistic Models

Deep learning practictioners generally use the same basic computational tools as other machine learning practitioners who work with structured probabilistic models. However, in the context of deep learning, we usually make different design decisions about how to combine these tools, resulting in overall algorithms and models that have a very different flavor from more traditional graphical models.

The most striking difference between the deep learning style of graphical model design and the traditional style of graphical model design is that the deep learning style heavily emphasizes the use of latent variables. Deep learning models typically have more latent variables than observed variables. Moreover, the practitioner typically does not intend for the latent variables to take on any specific semantics ahead of time— the training algorithm is free to invent the concepts it needs to model a particular dataset. The latent variables are usually not very easy for a human to interpret after the fact, though visualization techniques may allow some rough characterization of what they represent. Complicated non-linear interactions between variables are accomplished via indirect connections that flow through multiple latent variables. By contrast, traditional graphical models usually contain variables that are at least occasionally observed, even if many of the variables are missing at random from some training examples. Complicated non-linear interactions between variables are modeled by using higher-order terms, with structure learning algorithms used to prune connections and control model capacity. When latent variables are used, they are often designed with some specific semantics in mind—the topic of a document, the intelligence of a student, the disease causing a patient's symptoms, etc. These models are often much more interpretable by human practitioners and often have more theoretical guarantees, yet are less able to scale to complex problems and are not reuseable in as many different contexts as deep models.

Another obvious difference is the kind of graph structure typically used in

the deep learning approach. This is tightly linked with the choice of inference algorithm. Traditional approaches to graphical models typically aim to maintain the tractability of exact inference. When this constraint is too limiting, a popular exact inference algorithm is loopy belief propagation. Both of these approaches often work well with very sparsely connected graphs. By comparison, very few interesting deep models admit exact inference, and loopy belief propagation is almost never used for deep learning. Most deep models are designed to make Gibbs sampling or variational inference algorithms, rather than loopy belief propagation, efficient. Another consideration is that deep learning models contain a very large number of latent variables, making efficient numerical code essential. As a result of these design constraints, most deep learning models are organized into regular repeating patterns of units grouped into layers, but neighboring layers may be fully connected to each other. When sparse connections are used, they usually follow a regular pattern, such as the block connections used in convolutional models.

Finally, the deep learning approach to graphical modeling is characterized by a marked tolerance of the unknown. Rather than simplifying the model until all quantities we might want can be computed exactly, we increase the power of the model until it is just barely possible to train or use. We often use models whose marginal distributions cannot be computed, and are satisfied simply to draw approximate samples from these models. We often train models with an intractable objective function that we cannot even approximate in a reasonable amount of time, but we are still able to approximately train the model if we can efficiently obtain an estimate of the gradient of such a function. The deep learning approach is often to figure out what the minimum amount of information we absolutely need is, and then to figure out how to get a reasonable approximation of that information as quickly as possible.

### 16.6.1 Example: The Restricted Boltzmann Machine

TODO: rework this section. Add pointer to Chapter 20.2. TODO what do we want to exemplify here?

The *restricted Boltzmann machine* (RBM) (Smolensky, 1986) or *harmonium* is an example of a model that TODO what do we want to exemplify here?

It is an energy-based model with binary visible and hidden units. Its energy function is

$$E(v, h) = -b^\top v - c^\top h - v^\top W h$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. The model is depicted graphically in Fig. 16.12. As this figure makes clear, an important aspect of this model is that there are no direct interactions between any two

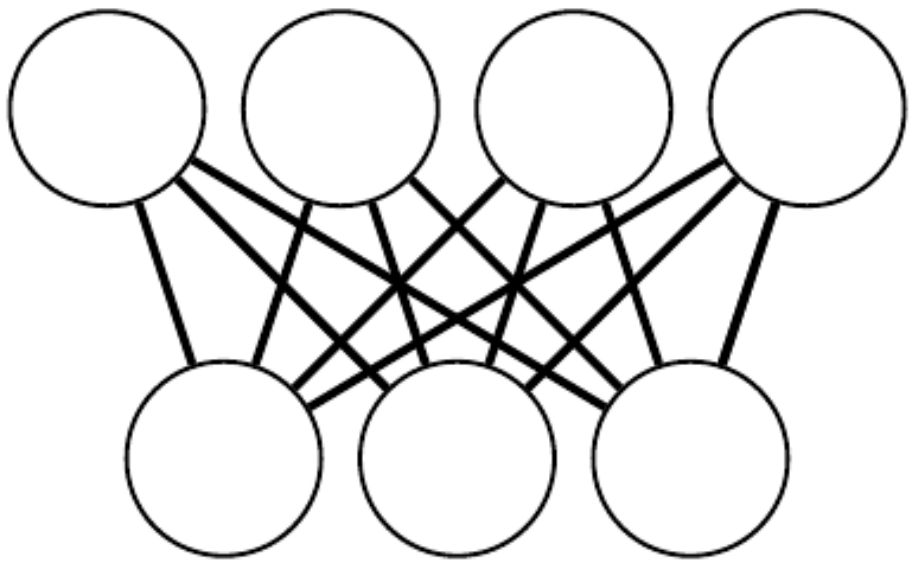

Figure 16.12: An example RBM drawn as a Markov network

visible units or between any two hidden units (hence the "restricted," a general Boltzmann machine may have arbitrary connections).

The restrictions on the RBM structure yield the nice properties

$$p(\mathbf{h} \mid \mathbf{v}) = \Pi_i p(\mathrm{h}_i \mid \mathbf{v})$$

and

$$p(\mathbf{v} \mid \mathbf{h}) = \Pi_i p(\mathrm{v}_i \mid \mathbf{h}).$$

The individual conditionals are simple to compute as well, for example

$$p(\mathrm{h}_i = 1 \mid \mathbf{v}) = \sigma\left(\mathbf{v}^\top \boldsymbol{W}_{:,i} + b_i\right).$$

Together these properties allow for efficient block Gibbs sampling, alternating between sampling all of $\mathbf{h}$ simultaneously and sampling all of $\mathbf{v}$ simultaneously.

Since the energy function itself is just a linear function of the parameters, it is easy to take the needed derivatives. For example,

$$\frac{\partial}{\partial \boldsymbol{W}_{i,j}} \mathbb{E}\ ,\ \ E(\mathbf{v}, \mathbf{h}) = -\mathrm{v}_i \mathrm{h}_j\,.$$

These two properties–efficient Gibbs sampling and efficient derivatives– make it possible to train the RBM with stochastic approximations to $\nabla_\theta \log Z$.

# Chapter 17

# The Manifold Perspective on Representation Learning

*Manifold learning* is an approach to machine learning that is capitalizing on the *manifold hypothesis* (Cayton, 2005; Narayanan and Mitter, 2010): *the data generating distribution is assumed to concentrate near regions of low dimensionality.* The notion of manifold in mathematics refers to continuous spaces that locally resemble Euclidean space, and the term we should be using is really *submanifold*, which corresponds to a subset which has a manifold structure. The use of the term manifold in machine learning is much looser than its use in mathematics, though:

- the data may not be strictly on the manifold, but only near it,
- the dimensionality may not be the same everywhere,
- the notion actually referred to in machine learning naturally extends to discrete spaces.

Indeed, although the very notions of a manifold or submanifold are defined for continuous spaces, the more general notion of *probability concentration* applies equally well to discrete data. It is a kind of informal *prior* assumption about the data generating distribution that seems particularly well-suited for AI tasks such as those involving images, video, speech, music, text, etc. In all of these cases the natural data has the property that *randomly choosing configurations of the observed variables according to a factored distribution (e.g. uniformly) are very unlikely to generate the kind of observations we want to model.* What is the probability of generating a natural looking image by choosing pixel intensities independently of each other? What is the probability of generating a meaningful natural language paragraph by independently choosing each character in a

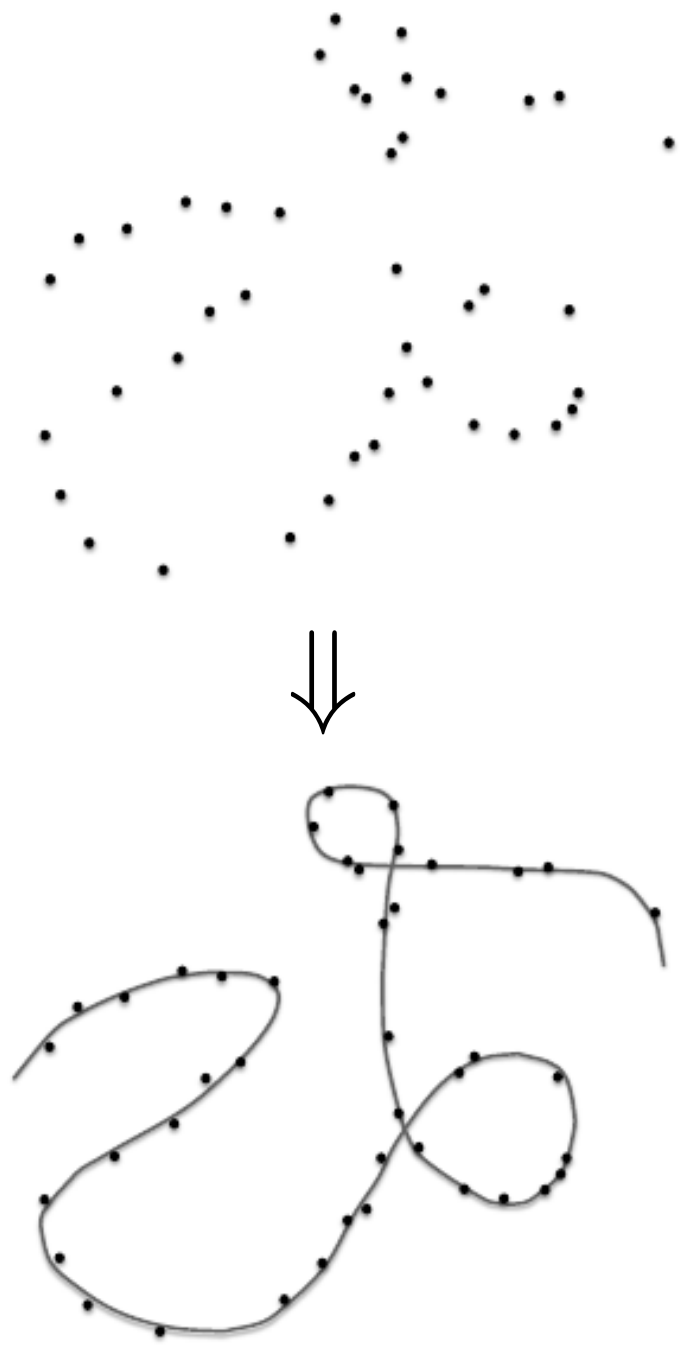

Figure 17.1: Top: data sampled from a distribution in a high-dimensional space (one 2 dimensions shown for illustration) that is actually concentrated near a one-dimensional manifold, which here is like a twisted string. Bottom: the underlying manifold that the learner should infer.

string? Doing a thought experiment should give a clear answer: an exponentially tiny probability. This is because the probability distribution of interest concentrates in a tiny volume of the total space of configurations. That means that to the first degree, the problem of characterizing the data generating distribution can be reduced to a binary classification problem: *is this configuration probable or not?*. Is this a grammatically and semantically plausible sentence in English? Is this a natural-looking image? Answering these questions tells us much more about the nature of natural language or text than the additional information one would have by being able to assign a precise probability to each possible sequence of characters or set of pixels. Hence, simply characterizing *where* probability concentrates is a fundamental importance, and this is what manifold learning algorithms attempt to do. Because it is a *where* question, it is more about *geometry* than about probability distributions, although we find both views useful

when designing learning algorithms for AI tasks.

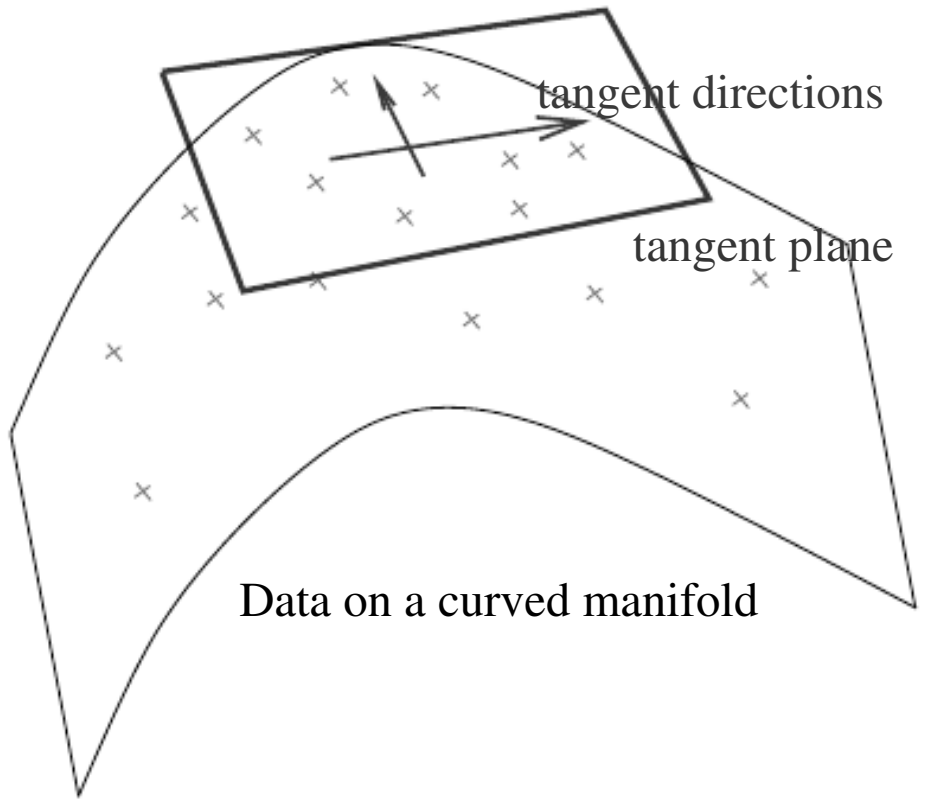


Figure 17.2: A two-dimensional manifold near which training examples are concentrated, along with a tangent plane and its associated tangent directions, forming a basis that specify the directions of small moves one can make to stay on the manifold.

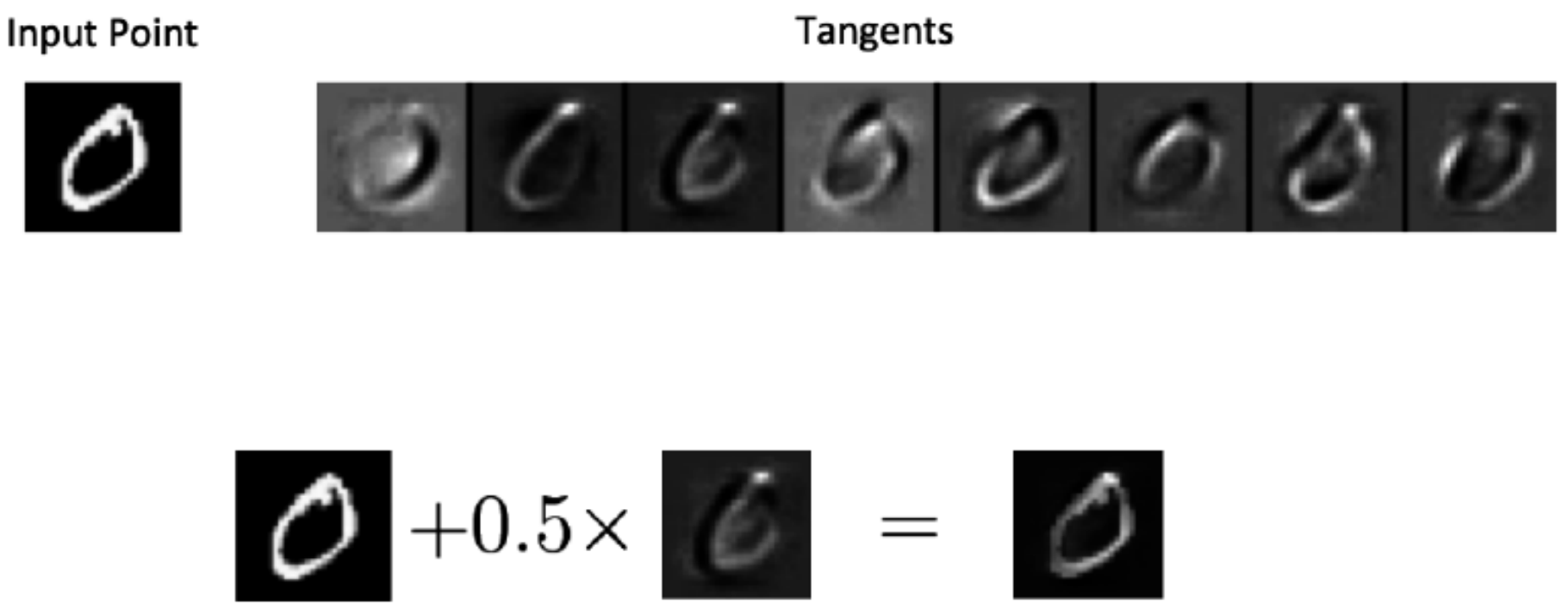


Figure 17.3: Illustration of tangent vectors of the manifold estimated by a contractive auto-encoder (CAE), at some input point (top left, image of a zero). Each image on the top right corresponds to a tangent vector. They are obtained by picking the dominant singular vectors (with largest singular value) of the Jacobian $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ (see Section 15.10). Taking the original image plus a small quantity of any of these tangent vectors yields another plausible image, as illustrated in the bottom. The leading tangent vectors seem to correspond to small deformations, such as translation, or shifting ink around locally in the original image. Reproduced with permission from the authors of Rifai *et al.* (2011a).

In addition to the property of probability concentration, there is another one

that characterizes the manifold hypothesis: *when a configuration is probable it is generally surrounded (at least in some directions) by other probable configurations.* If a configuration of pixels looks like a natural image, then there are tiny changes one can make to the image (like translating everything by 0.1 pixel to the left) which yield another natural-looking image. The number of independent ways (each characterized by a number indicating how much or whether we do it) by which a probable configuration can be locally transformed into another probable configuration indicates the local *dimension of the manifold.* Whereas maximum likelihood procedures tend to concentrate probability mass on the training examples (which can each become a local maximum of probability when the model overfits), the manifold hypothesis suggests that good solutions instead concentrate probability along ridges of high probability (or their high-dimensional generalization) that connect nearby examples to each other. This is illustrated in Figure 17.1.

What is most commonly learned to characterize a manifold is a *representation* of the data points on (or near, i.e. projected on) the manifold. Such a representation for a particular example is also called its *embedding.* It is typically given by a low-dimensional vector, with less dimensions than the "ambient" space of which the manifold is a low-dimensional subset. Some algorithms (non-parametric manifold learning algorithms, discussed below) directly learn an embedding for each training example, while others learn a more general mapping, sometimes called an encoder, or representation function, that maps any point in the ambient space (the input space) to its embedding.

Another important characterization of a manifold is the set of its *tangent planes.* At a point $x$ on a $d$-dimensional manifold, the tangent plane is given by $d$ basis vectors that span the local directions of variation allowed on the manifold. As illustrated in Figure 17.2, these local directions specify how one can change $x$ infinitesimally while staying on the manifold.

Manifold learning has mostly focused on unsupervised learning procedures that attempt to capture these manifolds. Most of the initial machine learning research on learning non-linear manifolds has focused on *non-parametric* methods based on the *nearest-neighbor graph.* This graph has one node per training example and edges connecting near neighbors. Basically, these methods (Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004; Hinton and Roweis, 2003; van der Maaten and Hinton, 2008a) associate each of these nodes with a tangent plane that spans the directions of variations associated with the difference vectors between the example and its neighbors, as illustrated in Figure 17.4.

A global coordinate system can then be obtained through an optimization or

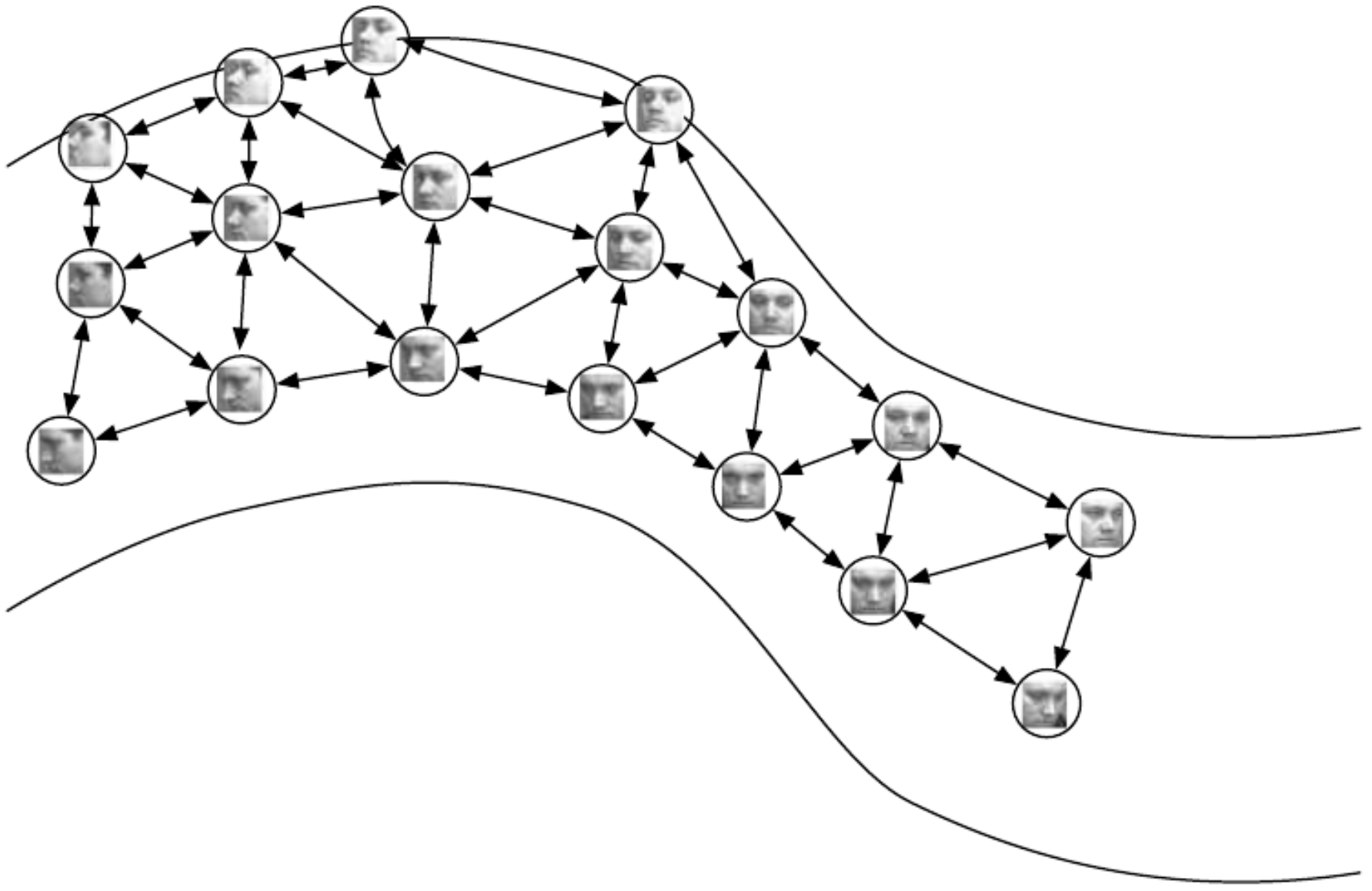

Figure 17.4: Non-parametric manifold learning procedures build a nearest neighbor graph whose nodes are training examples and arcs connect nearest neighbors. Various procedures can thus obtain the tangent plane associated with a neighborhood of the graph, and a coordinate system that associates each training example with a real-valued vector position, or *embedding*. It is possible to generalize such a representation to new examples by a form of interpolation. So long as the number of examples is large enough to cover the curvature and twists of the manifold, these approaches work well. Images from the QMUL Multiview Face Dataset (Gong *et al.*, 2000).

solving a linear system. Figure 17.5 illustrates how a manifold can be tiled by a large number of locally linear Gaussian-like patches (or "pancakes", because the Gaussians are flat in the tangent directions).

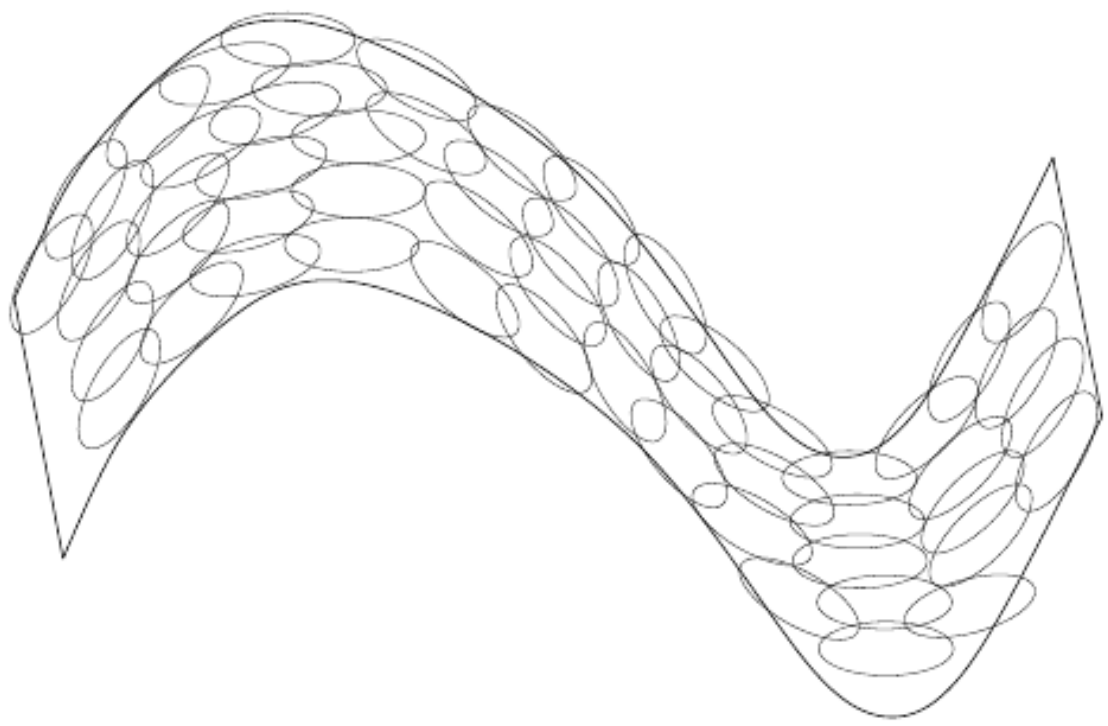

Figure 17.5: If the tangent plane at each location is known, then they can be tiled to form a global coordinate system or a density function. In the figure, each local patch can be thought of as a local Euclidean coordinate system or as a locally flat Gaussian, or *"pancake"*, with a very small variance in the directions orthogonal to the pancake and a very large variance in the directions defining the coordinate system on the pancake. The average of all these Gaussians would provide an estimated density function, as in the Manifold Parzen algorithm (Vincent and Bengio, 2003) or its non-local neural-net based variant (Bengio *et al.*, 2006c).

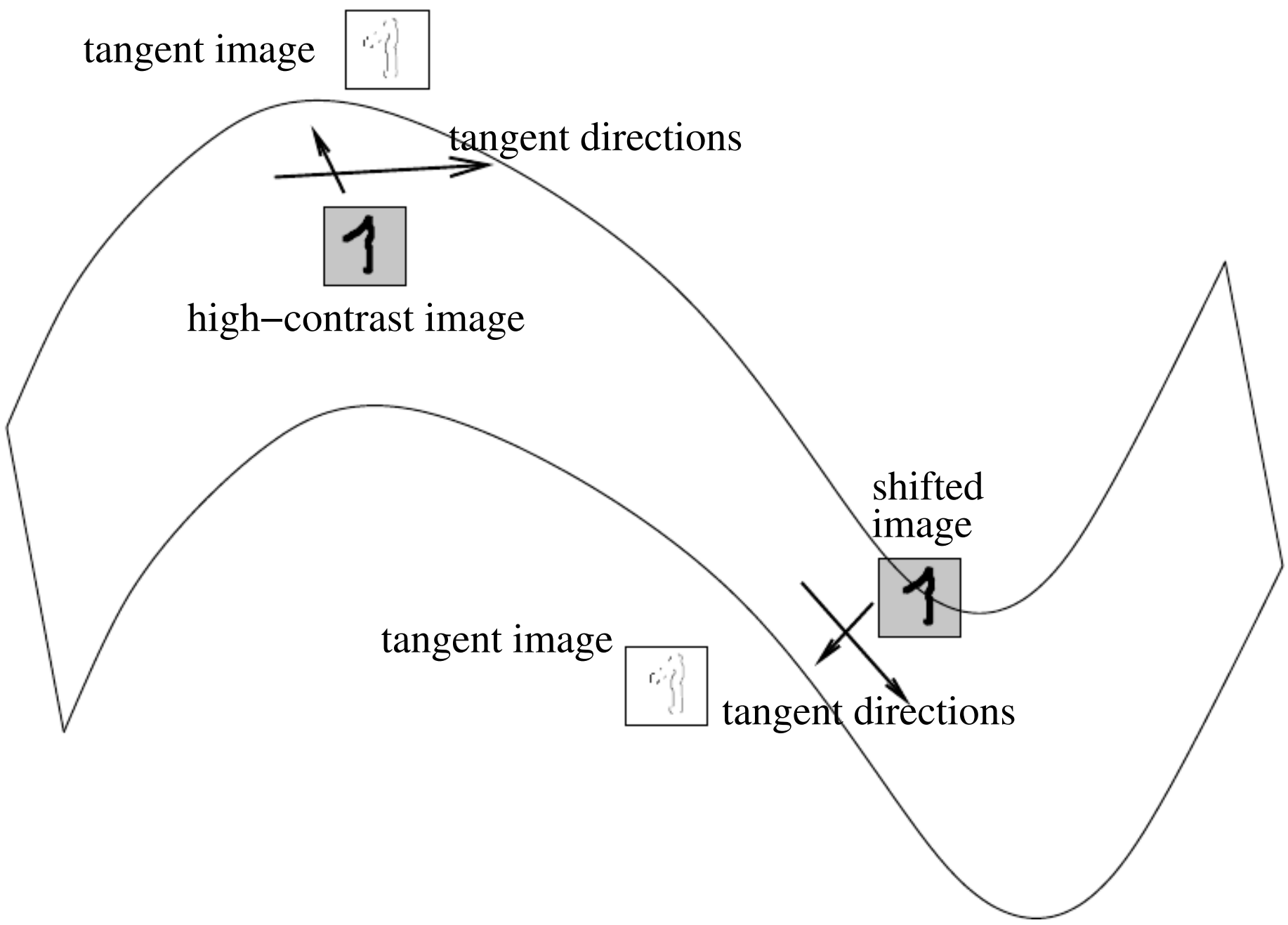


Figure 17.6: When the data are images, the tangent vectors can also be visualized like images. Here we show the tangent vector associated with translation: it corresponds to the difference between an image and a slightly translated version. This basically extracts

However, there is a fundamental difficulty with such non-parametric neighborhood-based approaches to manifold learning, raised in Bengio and Monperrus (2005): if the manifolds are not very smooth (they have many ups and downs and twists), one may need a very large number of training examples to cover each one of these variations, with no chance to generalize to unseen variations. Indeed, these methods can only generalize the shape of the manifold by interpolating between neighboring examples. Unfortunately, the manifolds of interest in AI have many ups and downs and twists and strong curvature, as illustrated in Figure 17.6. This motivates the use of distributed representations and deep learning for capturing manifold structure, which is the subject of this chapter.

Figure 17.7: Training examples of a face dataset – the QMUL Multiview Face Dataset (Gong *et al.*, 2000) – for which the subjects were asked to move in such a way as to cover the two-dimensional manifold corresponding to two angles of rotation. We would like learning algorithms to be able to discover and disentangle such factors. Figure 17.8 illustrates such a feat.

The hope of many manifold learning algorithms, including those based on deep learning and auto-encoders, is that one learns an *explicit or implicit coordinate system* for the leading factors of variation that explain most of the structure in the unknown data generating distribution. An example of an explicit coordinate system is one where the dimensions of the representation (e.g., the outputs of the encoder, i.e., of the hidden units that compute the "code" associated with the input) are directly the coordinates that map the unknown manifold. Training examples of a face dataset in which the images have been arranged visually on a 2-D manifold are shown in Figure 17.7, with the images laid down so that each of the two axes corresponds to one of the two angles of rotation of the face.

However, the objective is to *discover* such manifolds, and Figure 17.8 illustrates the images *generated* by a variational auto-encoder (Kingma and Welling, 2014a) when the two-dimensional auto-encoder code (representation) is varied

on the 2-D plane. Note how the algorithm actually discovered two independent factors of variation: angle of rotation and emotional expression.

Another kind of interesting illustration of manifold learning involves the discovery of *distributed representations for words*. Neural language models were initiated with the work of Bengio *et al.* (2001c, 2003b), in which a neural network is trained to predict the next word in a sequence of natural language text, given the previous words, and where each word is represented by a real-valued vector, called *embedding* or *neural word embedding*.

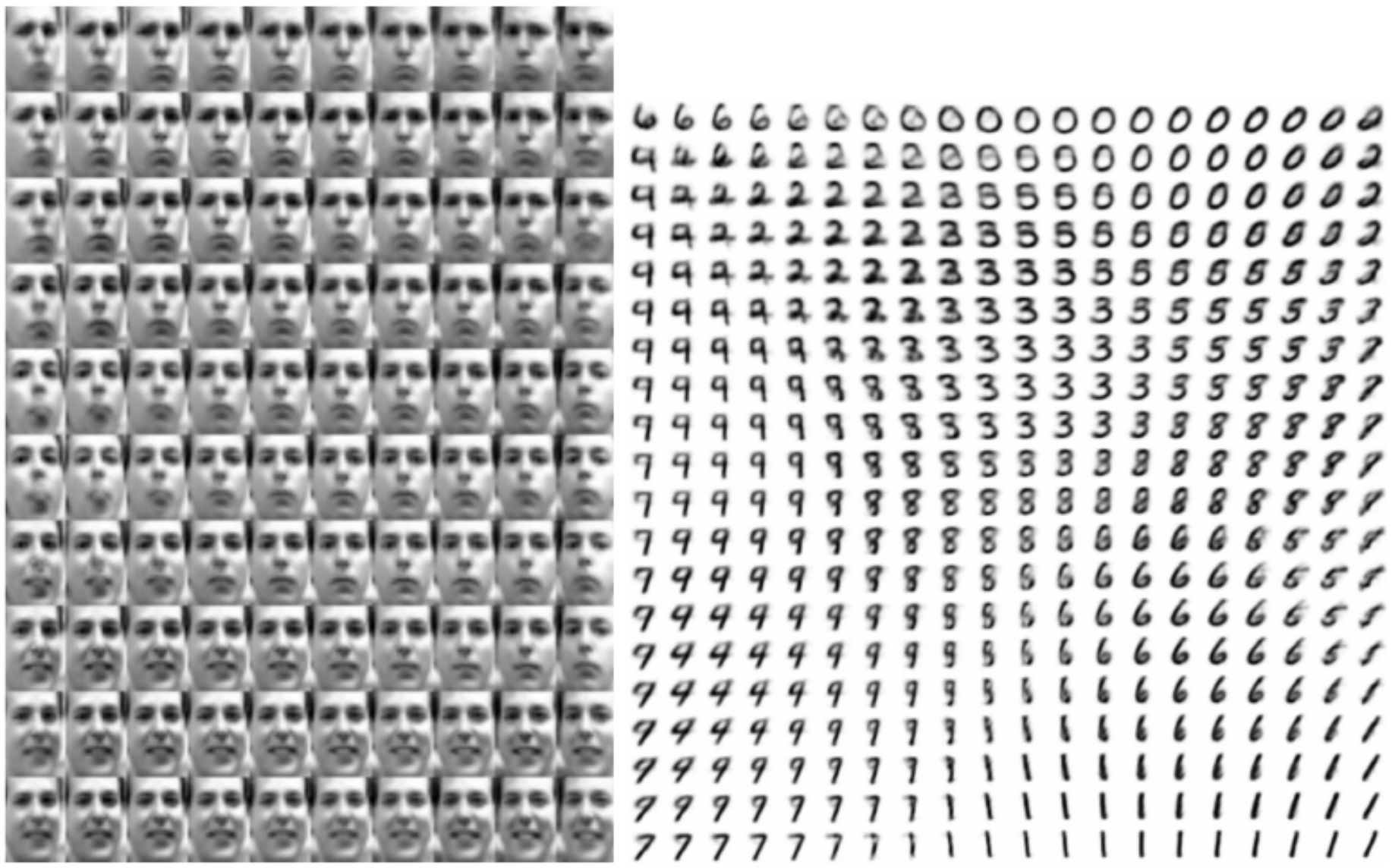

Figure 17.8: Two-dimensional representation space (for easier visualization), i.e., a Euclidean coordinate system for Frey faces (left) and MNIST digits (right), learned by a variational auto-encoder (Kingma and Welling, 2014a). Figures reproduced with permission from the authors. The images shown are not examples from the training set but images $\boldsymbol{x}$ actually generated by the model $P(\boldsymbol{x} \mid \boldsymbol{h})$, simply by changing the 2-D "code" $\boldsymbol{h}$ (each image corresponds to a different choice of "code" $\boldsymbol{h}$ on a 2-D uniform grid). On the left, one dimension that has been discovered (horizontal) mostly corresponds to a rotation of the face, while the other (vertical) corresponds to the emotional expression. The decoder deterministically maps codes (here two numbers) to images. The encoder maps images to codes (and adds noise, during training).

Figure 17.9 shows such neural word embeddings reduced to two dimensions (originally 50 or 100) using the t-SNE non-linear dimensionality reduction algorithm (van der Maaten and Hinton, 2008a). The figures zooms into different areas of the word-space and illustrates that words that are semantically and syntacti-

cally close end up having nearby embeddings.

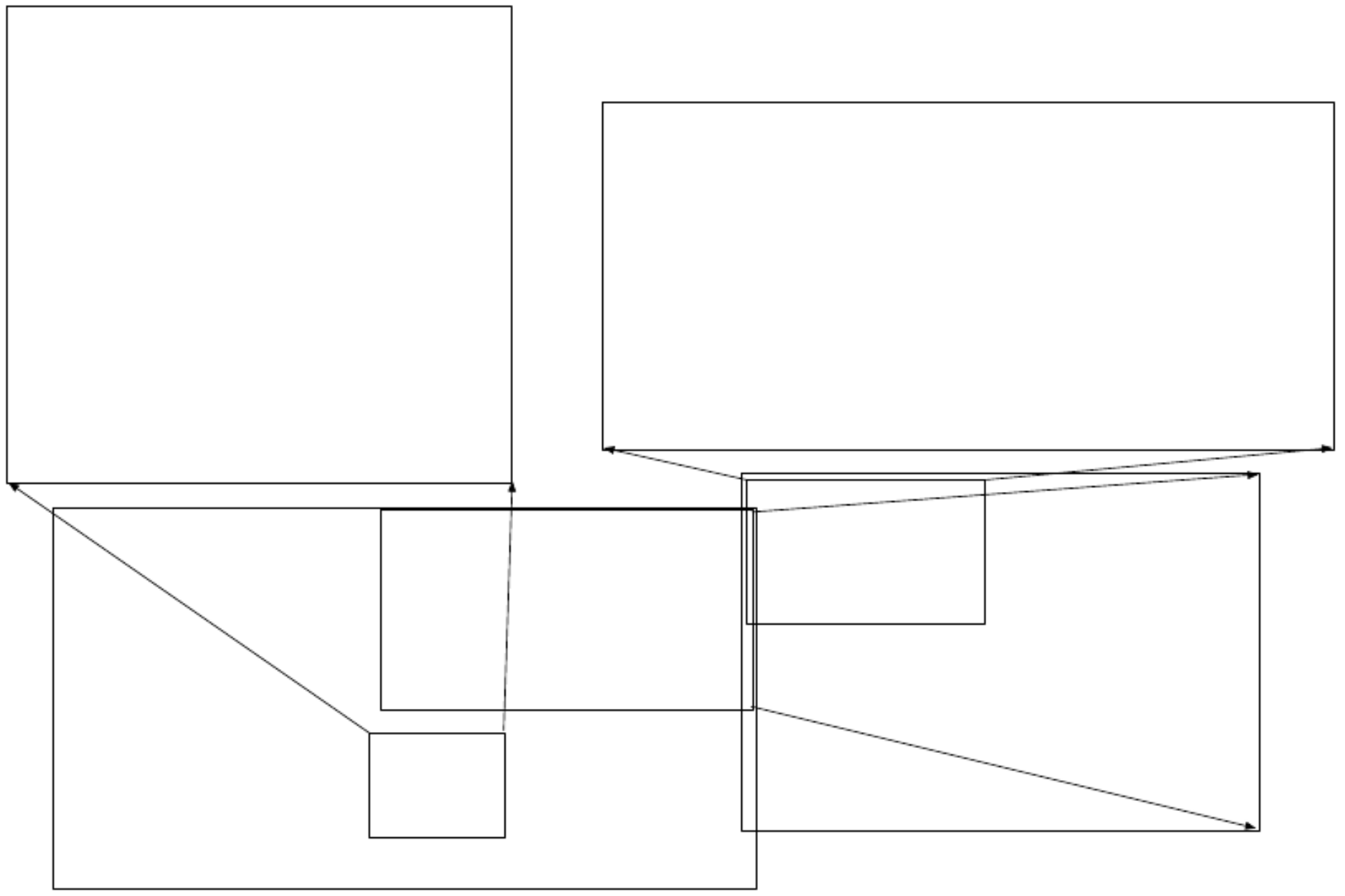

Figure 17.9: Two-dimensional representation space (for easier visualization), of English words, learned by a neural language model as in Bengio *et al.* (2001c, 2003b), with t-SNE for the non-linear dimensionality reduction from 100 to 2. Different regions are zoomed to better see the details. At the global level one can identify big clusters corresponding to part-of-speech, while locally one sees mostly semantic similarity explaining the neighborhood structure.

## 17.1 Manifold Interpretation of PCA and Linear Auto-Encoders

The above view of probabilistic PCA as a thin "pancake" of high probability is related to the manifold interpretation of PCA and linear auto-encoders, in which we are looking for projections of $\boldsymbol{x}$ into a subspace that preserves as much information as possible about $\boldsymbol{x}$. This is illustrated in Figure 17.10. Let the encoder be

$$\boldsymbol{h} = f(\boldsymbol{x}) = \boldsymbol{W}^\top(\boldsymbol{x} - \boldsymbol{\mu})$$

computing such a projection, a low-dimensional representation of $h$. With the auto-encoder view, we have a decoder computing the reconstruction

$$\hat{\boldsymbol{x}} = g(\boldsymbol{h}) = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}.$$

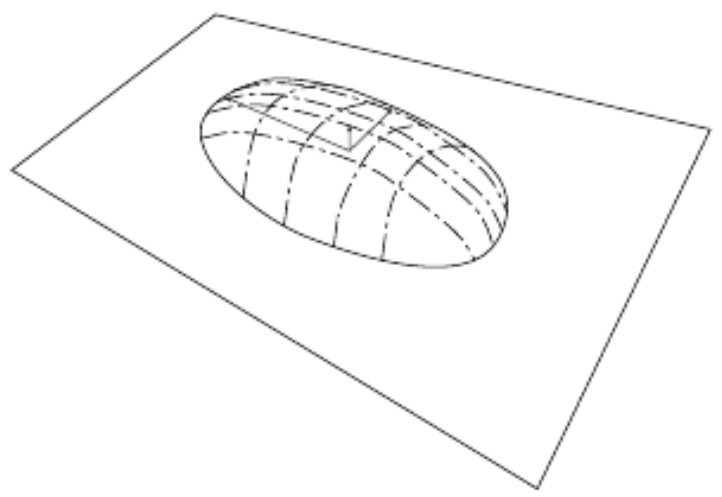

Figure 17.10: Flat Gaussian capturing probability concentration near a low-dimensional manifold. The figure shows the upper half of the "pancake" above the "manifold plane" which goes through its middle. The variance in the direction orthogonal to the manifold is very small (upward red arrow) and can be considered like "noise", where the other variances are large (larger red arrows) and correspond to "signal", and a coordinate system for the reduced-dimension data.

It turns out that the choices of linear encoder and decoder that minimize reconstruction error

$$\mathbb{E}[||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2]$$

correspond to $\boldsymbol{V} = \boldsymbol{W}$, $\boldsymbol{\mu} = \boldsymbol{b} = \mathbb{E}[\boldsymbol{x}]$ and the rows of $\boldsymbol{W}$ form an orthonormal basis which spans the same subspace as the principal eigenvectors of the covariance matrix

$$\boldsymbol{C} = \mathbb{E}[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^\top].$$

In the case of PCA, the rows of $\boldsymbol{W}$ are these eigenvectors, ordered by the magnitude of the corresponding eigenvalues (which are all real and non-negative). This is illustrated in Figure 17.11.

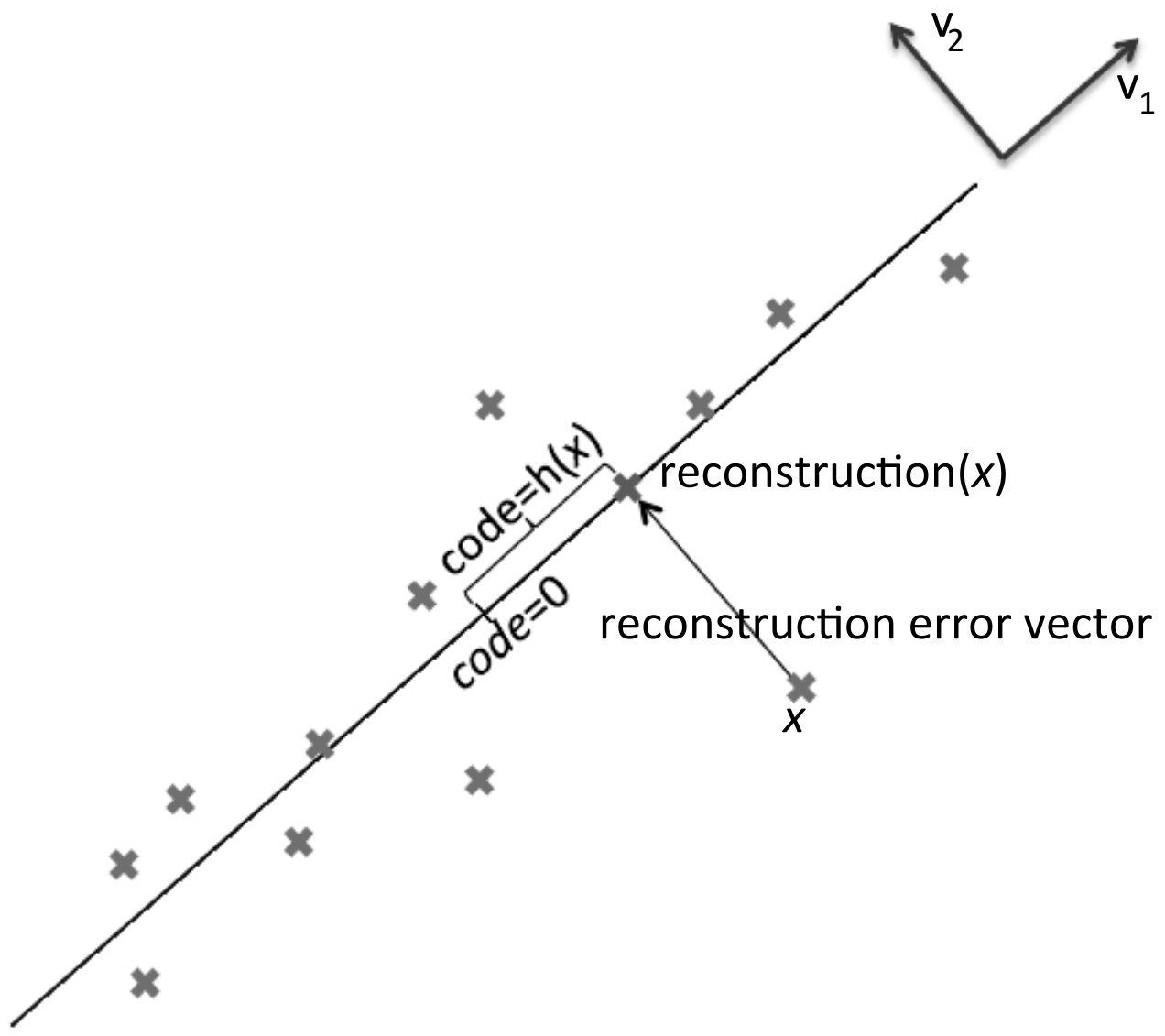


Figure 17.11: Manifold view of PCA and linear auto-encoders. The data distribution is concentrated near a manifold aligned with the leading eigenvectors (here, this is just $\boldsymbol{v}_1$) of the data covariance matrix. The other eigenvectors (here, just $\boldsymbol{v}_2$) are orthogonal to the manifold. A data point (in red, $\boldsymbol{x}$) is encoded into a lower-dimensional representation or code $\boldsymbol{h}$ (here the scalar which indicates the position on the manifold, starting from $\boldsymbol{h} = 0$). The decoder (transpose of the encoder) maps $\boldsymbol{h}$ to the data space, and corresponds to a point lying exactly on the manifold (green cross), the orthogonal projection of $\boldsymbol{x}$ on the manifold. The optimal encoder and decoder minimize the sum of reconstruction errors (difference vector between $\boldsymbol{x}$ and its reconstruction).

One can also show that eigenvalue $\lambda_i$ of $\boldsymbol{C}$ corresponds to the variance of $\boldsymbol{x}$ in the direction of eigenvector $\boldsymbol{v}_i$. If $\boldsymbol{x} \in \mathbb{R}^D$ and $\boldsymbol{h} \in \mathbb{R}^d$ with $d < D$, then the optimal reconstruction error (choosing $\boldsymbol{\mu}$, $\boldsymbol{b}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ as above) is

$$\min \mathbb{E}[||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2] = \sum_{i=d+1}^{D} \lambda_i.$$

Hence, if the covariance has rank $d$, the eigenvalues $\lambda_{d+1}$ to $\lambda_D$ are 0 and reconstruction error is 0.

Furthermore, one can also show that the above solution can be obtained by maximizing the variances of the elements of $\boldsymbol{h}$, under orthonormal $\boldsymbol{W}$, instead of minimizing reconstruction error.

## 17.2 Manifold Interpretation of Sparse Coding

Sparse coding was introduced in Section 15.6.2 a linear factors generative model. It also has an interesting *manifold learning interpretation.* The codes $\boldsymbol{h}$ inferred with the above equation do not fill the space in which $\boldsymbol{h}$ lives. Instead, probability mass is concentrated on axis-aligned subspaces: sets of values of $\boldsymbol{h}$ for which most of the axes are set at 0. We can thus decompose $\boldsymbol{h}$ into two pieces of information:

- A binary pattern $\boldsymbol{\beta}$ which specifies which $h_i$ are non-zero, with $N_a = \sum_i \beta_i$ the number of "active" (non-zero) dimensions.
- A variable-length real-valued vector $\boldsymbol{\alpha} \in \mathbb{R}^{N_a}$ which specifies the coordinates for each of the active dimensions.

The pattern $\boldsymbol{\beta}$ can be viewed as specifying an $N_a$-dimensional region in input space (the set of $\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$ where $h_i = 0$ if $b_i = 0$). That region is actually a linear manifold, an $N_a$-dimensional hyperplane. All those hyperplanes go through a "center" $\boldsymbol{x} = \boldsymbol{b}$. The vector $\boldsymbol{\alpha}$ then specifies a Euclidean coordinate on that hyperplane.

Because the prior $P(\boldsymbol{h})$ is concentrated around 0, the probability mass of $P(\boldsymbol{x})$ is concentrated on the regions of these hyperplanes near $\boldsymbol{x} = \boldsymbol{b}$. Depending on the amount of reconstruction error (output variance for $P(\boldsymbol{x} \mid g(\boldsymbol{h}))$), there is also probability mass bleeding around these hyperplanes and making them look more like pancakes. Each of these hyperplane-aligned manifolds and the associated distribution is just like the ones we associate to probabilistic PCA and factor analysis. The crucial difference is that instead of one hyperplane, we have $2^d$ hyperplanes if $\boldsymbol{h} \in \mathbb{R}^d$. Due to the sparsity prior, however, most of these flat Gaussians are unlikely: only the ones corresponding to a small $N_a$ (with only a few of the axes being active) are likely. For example, if we were to restrict ourselves to only those values of $\boldsymbol{b}$ for which $N_a = k$, then one would have $\binom{d}{k}$ Gaussians. With this exponentially large number of Gaussians, the interesting thing to observe is that the sparse coding model only has a number of parameters linear in the number of dimensions of $\boldsymbol{h}$. This property is shared with other distributed representation learning algorithms described in this chapter, such as the regularized auto-encoders.

## 17.3 The Entropy Bias from Maximum Likelihood

TODO: how the log-likelihood criterion forces a learner that is not able to generalize perfectly to yield an estimator that is much smoother than the target distribution. Phrase it in terms of entropy, not smoothness.

## 17.4 Manifold Learning via Regularized Auto-Encoders

Auto-encoders have been described in Section 15. What is their connection to manifold learning? This is what we discuss here.

We denote $f$ the encoder function, with $\boldsymbol{h} = f(\boldsymbol{x})$ the representation of $\boldsymbol{x}$, and $g$ the decoding function, with $\hat{\boldsymbol{x}} = g(\boldsymbol{h})$ the reconstruction of $\boldsymbol{x}$, although in some cases the encoder is a conditional distribution $q(\boldsymbol{h} \mid \boldsymbol{x})$ and the decoder is a conditional distribution $P(\boldsymbol{x} \mid \boldsymbol{h})$.

What all auto-encoders have in common, when they are prevented from simply learning the identity function for all possible input $\boldsymbol{x}$, is that training them involves a compromise between two "forces":

1. Learning a representation $\boldsymbol{h}$ of training examples $\boldsymbol{x}$ such that $\boldsymbol{x}$ can be approximately recovered from $\boldsymbol{h}$ through a decoder. Note that this needs not be true for any $\boldsymbol{x}$, only for those that are probable under the data generating distribution.

2. Some constraint or regularization is imposed, either on the code $\boldsymbol{h}$ or on the composition of the encoder/decoder, so as to make the transformed data somehow simpler or to prevent the auto-encoder from achieving perfect reconstruction everywhere. We can think of these constraints or regularization as a preference for solutions in which the representation is as simple as possible, e.g., factorized or as constant as possible, in as many directions as possible. In the case of the bottleneck auto-encoder a fixed number of representation dimensions is allowed, that is smaller than the dimension of $\boldsymbol{x}$. In the case of sparse auto-encoders (Section 15.8) the representation elements $h_i$ are pushed towards 0. In the case of denoising auto-encoders (Section 15.9), the encoder/decoder function is encouraged to be contractive (have small derivatives). In the case of the contractive auto-encoder (Section 15.10), the encoder function alone is encouraged to be contractive, while the decoder function is tied (by symmetric weights) to the encoder function. In the case of the variational auto-encoder (Section 20.9.3), a prior $\log P(\boldsymbol{h})$ is imposed on $\boldsymbol{h}$ to make its distribution factorize and concentrate as much as possible. Note how in the limit, for all of these cases, the regularization prefers representations that are insensitive to the input.

Clearly, the second type of force alone would not make any sense (as would any regularizer, in general). How can these two forces (reconstruction error on one hand, and "simplicity" of the representation on the other hand) be reconciled? The solution of the optimization problem is that *only the variations that are needed to distinguish training examples need to be represented.* If the data generating distribution concentrates near a low-dimensional manifold, this yields

Figure 17.12: A regularized auto-encoder or a bottleneck auto-encoder has to reconcile two forces: reconstruction error (which forces it to keep enough information to distinguish training examples from each other), and a regularizer or constraint that aims at reducing its representational ability, to make it as insensitive as possible to the input in as many directions as possible. The solution is for the learned representation to be sensitive to changes along the manifold (green arrow going to the right, tangent to the manifold) but invariant to changes orthogonal to the manifold (blue arrow going down). This yields to *contraction* of the representation in the directions orthogonal to the manifold.

representations that implicitly capture a local coordinate for this manifold: only the variations tangent to the manifold around $\boldsymbol{x}$ need to correspond to changes in $\boldsymbol{h} = f(\boldsymbol{x})$. Hence the encoder learns a mapping from the embedding space $\boldsymbol{x}$ to a representation space, a mapping that is only sensitive to changes along the manifold directions, but that is insensitive to changes orthogonal to the manifold. This idea is illustrated in Figure 17.12. A one-dimensional example is illustrated in Figure 17.13, showing that by making the auto-encoder contractive around the data points (and the reconstruction point towards the nearest data point), we recover the manifold structure (of a set of 0-dimensional manifolds in a 1-dimensional embedding space, in the figure).

## 17.5 Tangent Distance, Tangent-Prop, and Manifold Tangent Classifier

One of the early attempts to take advantage of the manifold hypothesis is the Tangent Distance algorithm (Simard *et al.*, 1993, 1998). It is a non-parametric nearest-neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates. It is assumed that we are trying to classify examples

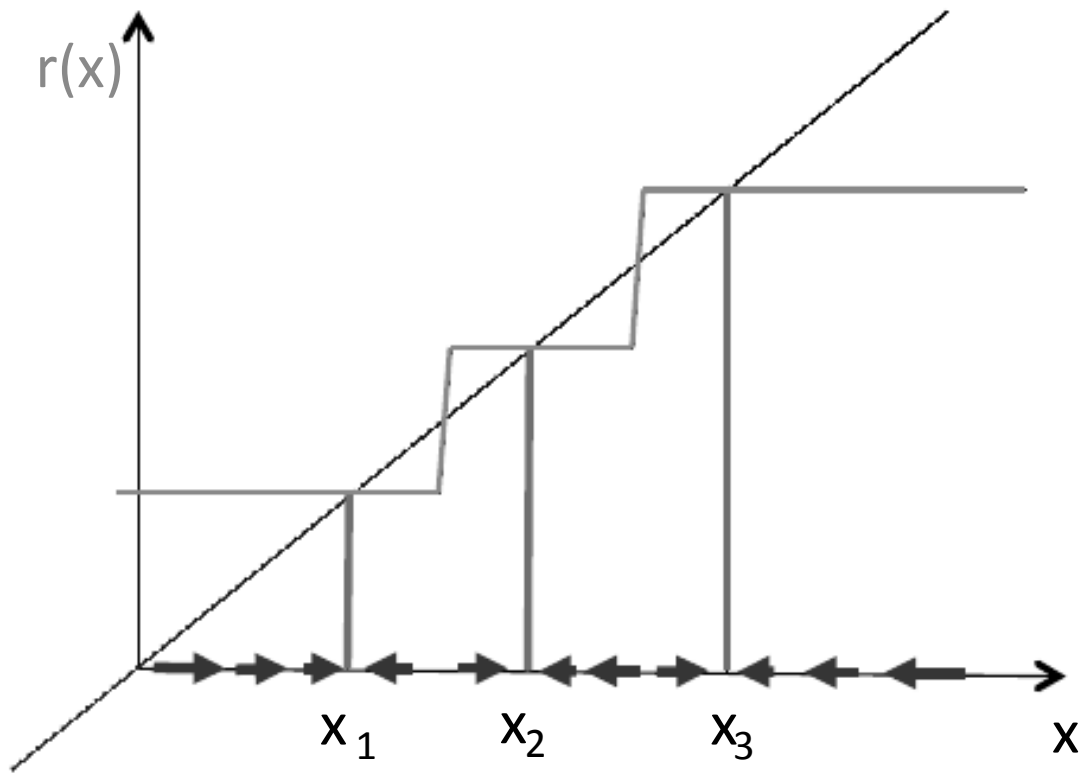


Figure 17.13: If the auto-encoder learns to be contractive around the data points, with the reconstruction pointing towards the nearest data points, it captures the manifold structure of the data. This is a 1-dimensional version of Figure 17.12. The denoising auto-encoder explicitly tries to make the derivative of the reconstruction function $r(\boldsymbol{x})$ small around the data points. The contractive auto-encoder does the same thing for the encoder. Although the derivative of $r(\boldsymbol{x})$ is asked to be small around the data points, it can be large between the data points (e.g. in the regions between manifolds), and it has to be large there so as to reconcile reconstruction error ($r(\boldsymbol{x}) \approx \boldsymbol{x}$ for data points $\boldsymbol{x}$) and contraction (small derivatives of $r(\boldsymbol{x})$ near data points).

and that examples on the same manifold share the same category. Since the classifier should be invariant to the local factors of variation that correspond to movement on the manifold, it would make sense to use as nearest-neighbor distance between points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ the distance between the manifolds $M_1$ and $M_2$ to which they respectively belong. Although that may be computationally difficult (it would require an optimization, to find the nearest pair of points on $M_1$ and $M_2$), a cheap alternative that makes sense locally is to approximate $M_i$ by its tangent plane at $\boldsymbol{x}_i$ and measure the distance between the two tangents, or between a tangent plane and a point. That can be achieved by solving a low-dimensional linear system (in the dimension of the manifolds). Of course, this algorithm requires one to specify the tangent vectors at any point

In a related spirit, the Tangent-Prop algorithm (Simard *et al.*, 1992) proposes to train a neural net classifier with an extra penalty to make the output $f(x)$ of the neural net locally invariant to known factors of variation. These factors of variation correspond to movement of the manifold near which examples of the same class concentrate. Local invariance is achieved by requiring $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ to be orthogonal to the known manifold tangent vectors $\boldsymbol{v}_i$ at $\boldsymbol{x}$, or equivalently that

the directional derivative of $f$ at $\boldsymbol{x}$ in the directions $\boldsymbol{v}_i$ be small:

$$\text{regularizer} = \lambda \sum_i \left( \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \cdot \boldsymbol{v}_i \right)^2 . \tag{17.1}$$

Like for tangent distance, the tangent vectors are derived a priori, e.g., from the formal knowledge of the effect of transformations such as translation, rotation, and scaling in images. Tangent-Prop has been used not just for supervised learning (Simard *et al.*, 1992) but also in the context of reinforcement learning (Thrun, 1995).

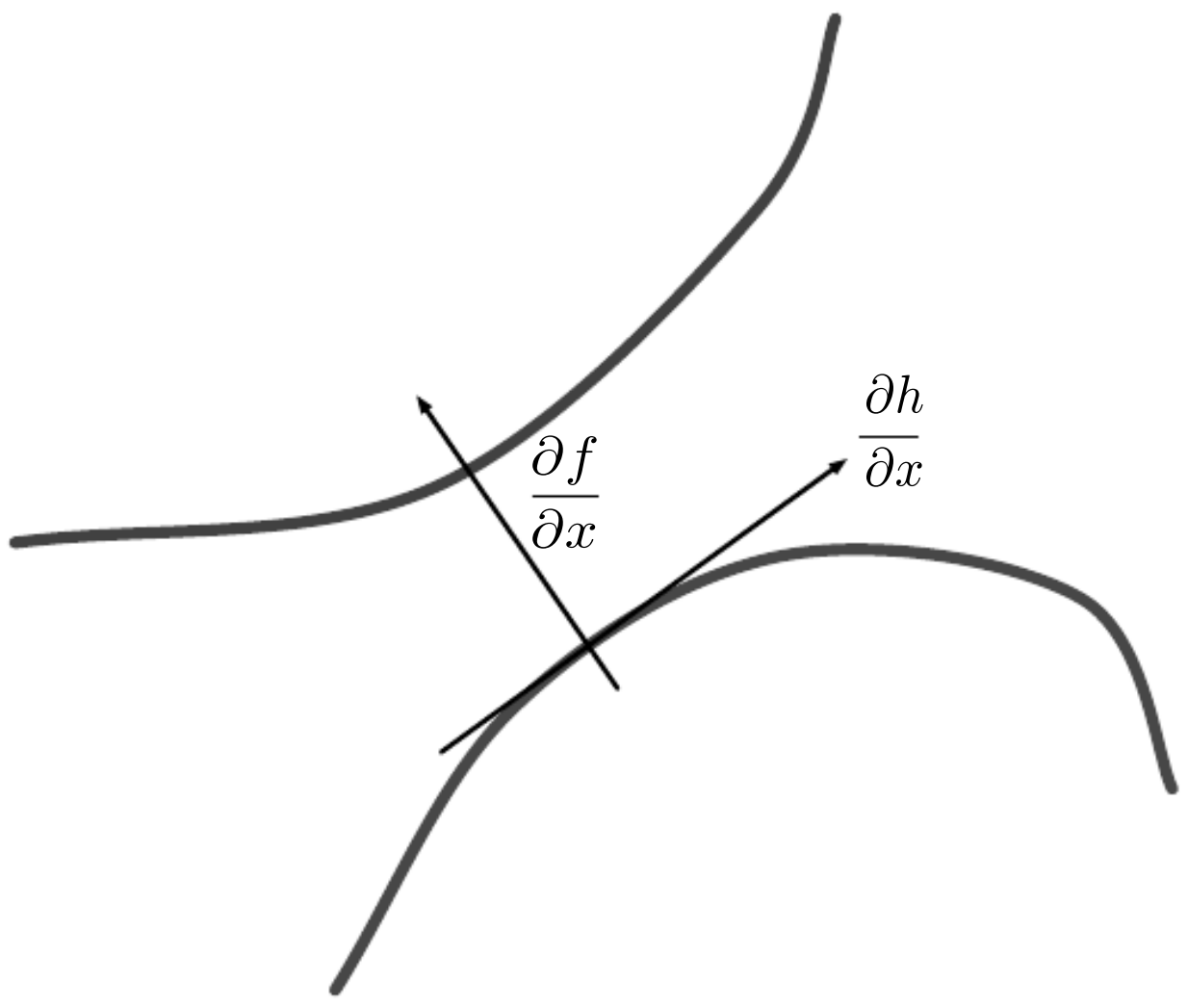


Figure 17.14: Illustration of the main idea of the tangent-prop algorithm (Simard *et al.*, 1992) and manifold tangent classifier (Rifai *et al.*, 2011d), which both regularize the classifier output function $f(\boldsymbol{x})$ (e.g. estimating conditional class probabilities given the input) so as to make it invariant to the local directions of variations $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ (manifold tangent directions). This can be achieved by penalizing the magnitude of the dot product of all the rows of $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ (the tangent directions) with all the rows of $\frac{\partial f}{\partial \boldsymbol{x}}$ (the directions of sensitivity of each output to the input). In the case of the tangent-prop algorithm, the tangent directions are given a priori, whereas in the case of the manifold tangent classifier, they are learned, with $\boldsymbol{h}(\boldsymbol{x})$ being the learned representation of the input $\boldsymbol{x}$. The figure illustrates two manifolds, one per class, and we see that the classifier output increases the most as we move from one manifold to the other, in input space.

A more recent paper introduces the Manifold Tangent Classifier (Rifai *et al.*, 2011d), which eliminates the need to know the tangent vectors a priori, and

instead uses a contractive auto-encoder to estimate them at any point. As we have seen in the previous section and Figure 15.9, auto-encoders in general, and contractive auto-encoders especially well, learn a representation $\boldsymbol{h}$ that is most sensitive to the factors of variation present in the data $\boldsymbol{x}$, so that the leading singular vectors of $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ correspond to the estimated tangent vectors. As illustrated in Figure 15.10, these estimated tangent vectors go beyond the classical invariants that arise out of the geometry of images (such as translation, rotation and scaling) and include factors that must be learned because they are object-specific (such as adding or moving body parts). The algorithm proposed with the manifold tangent classifier is therefore simple: (1) use a regularized auto-encoder such as the contractive auto-encoder to learn the manifold structure by unsupervised learning (2) use these tangents to regularize a neural net classifier as in Tangent-Prop (Eq. 17.1). TODO Tangent Prop or Tangent-Prop?

# Chapter 18

# Confronting the Partition Function

TODO– make sure the book explains asymptotic consistency somewhere, add links to it here

In Section 13.2.2 we saw that many probabilistic models (commonly known as undirected graphical models) are defined by an unnormalized probability distribution $\tilde{p}(\mathbf{x};\theta)$ or energy function (Section 13.2.4)

$$E(\mathbf{x}) = -\log \tilde{p}(\mathbf{x}). \tag{18.1}$$

Because the analytic formulation of the model is via this energy function or unnormalized probability, the complete formulation of the probability function or probability density requires a normalization constant called partition function $Z(\theta)$ such that

$$p(\mathbf{x};\theta) = \frac{1}{Z(\theta)}\tilde{p}(\mathbf{x};\theta)$$

is a valid, normalized probability distribution. The partition function is an integral or sum over the unnormalized probability of all states. This operation is intractable for many interesting models.

As we will see in chapter 20, many deep learning models are designed to have a tractable normalizing constant, or are designed to be used in ways that do not involve computing $p(\mathbf{x})$ at all. However, other models directly confront the challenge of intractable partition functions. In this chapter, we describe techniques used for training and evaluating models that have intractable partition functions.

## 18.1 The Log-Likelihood Gradient of Energy-Based Models

What makes learning by maximum likelihood particularly difficult is that the *partition function depends on the parameters*, so that the log-likelihood gradient has a term corresponding to the gradient of the partition function:

$$\frac{\partial \log p(\mathbf{x};\theta)}{\partial \theta} = -\frac{\partial E(\mathbf{x})}{\partial \theta} - \frac{\log Z(\theta)}{\partial \theta}. \tag{18.2}$$

In the case where the energy function is analytically tractable (e.g., RBMs), the difficult part is estimating the the gradient of the partition function. Unsurprisingly, since computing $Z$ itself is intractable, we find that computing its gradient is also intractable, but the good news is that it corresponds to an expectation over the model distribution, which can be estimated by Monte-Carlo methods.

Though the gradient of the log partition function is intractable to evaluate accurately, it is straightforward to analyze algebraically. The derivatives we need for learning are of the form

$$\frac{\partial}{\partial \theta} \log p(\mathbf{x})$$

where $\theta$ is one of the parameters of $p(\mathbf{x})$. These derivatives are given simply by

$$\frac{\partial}{\partial \theta} \log p(\mathbf{x}) = \frac{\partial}{\partial \theta} \left( \log \tilde{p}(\mathbf{x}) - \log Z \right).$$

In this chapter, we are primarily concerned with the estimation of the term on the right:

$$\begin{aligned}
&\frac{\partial}{\partial \theta} \log Z \\
&= \frac{\frac{\partial}{\partial \theta} Z}{Z} \\
&= \frac{\frac{\partial}{\partial \theta} \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})}{Z} \\
&= \frac{\sum_{\mathbf{x}} \frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x})}{Z}.
\end{aligned}$$

For models that guarantee $p(\mathbf{x}) > 0$ for all $\mathbf{x}$, we can substitute $\exp\left(\log \tilde{p}(\mathbf{x})\right)$ for $\tilde{p}(\mathbf{x})$:

$$\begin{aligned}
&= \frac{\sum_{\mathbf{x}} \frac{\partial}{\partial \theta} \exp\left(\log \tilde{p}(\mathbf{x})\right)}{Z} \\
&= \frac{{}_{\mathbf{x}} \exp\left(\log \tilde{p}(\mathbf{x})\right) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x})}{Z}
\end{aligned}$$



$\sum$

$$= \frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x})}{Z}$$

$$= \sum_{\mathbf{x}} p(\mathbf{x}) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x})$$

$$= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x}).$$

This derivation made use of summation over discrete $\boldsymbol{x}$, but a similar result applies using integration over continuous $\boldsymbol{x}$. In the continuous version of the derivation, we use Leibniz's rule for differentiation under the integral sign to obtain the identity

$$\frac{\partial}{\partial \theta} \int \tilde{p}(\mathbf{x}) d\boldsymbol{x} = \int \frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x}) d\boldsymbol{x}.$$

This identity is only applicable under certain regularity conditions on $\tilde{p}$ and $\frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x})$[1]. Fortunately, most machine learning models of interest have these properties.

This identity

$$\frac{\partial}{\partial \theta} \log Z = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x}) \tag{18.3}$$

is the basis for a variety of Monte Carlo methods for approximately maximizing the likelihood of models with intractable partition functions.

Putting this result together with Eq. 18.2, we obtain the following well-known decomposition of the gradient in terms of the gradient of the energy function on the observed $\mathbf{x}$ and in average over the model distribution:

$$\frac{\partial - \log p(\mathbf{x}; \theta)}{\partial \theta} = \frac{\partial E(\mathbf{x})}{\partial \theta} - \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \frac{\partial}{\partial \theta} E(\mathbf{x}). \tag{18.4}$$

The first term is called the *positive phase* contribution to the gradient and it corresponds to pushing the energy down on the "positive" examples and reinforcing the interactions that are observed between random variables when $\mathbf{x}$ is observed, while the second term is called the *negative phase* contribution to the gradient and it corresponds to pushing the energy up everywhere else, with proportionally more push where the model currently puts more probability mass. When a minimum of the negative log-likelihood is found, the two terms must of course cancel each other, and the only thing that prevents the model from putting probability mass in exactly the same way as the training distribution is that it may be regularized or have some constraints, e.g. be parametric.

[1] In measure theoretic terms, the conditions are: (i) $\tilde{}$ must be a Lebesgue-integrable function of for every value of ; (ii) $\frac{\partial}{\partial \theta} \tilde{}(\ )$ must exist for all and almost all ; (iii) There exists an integrable function $(\ )$ that bounds $\frac{\partial}{\partial \theta} \tilde{}(\ )$ (i.e. such that $\frac{\partial}{\partial \theta} \tilde{}(\ )$ $(\ )$ for all and almost all ).

## 18.2 Stochastic Maximum Likelihood and Contrastive Divergence

The naive way of implementing equation 18.3 is to compute it by burning in a set of Markov chains from a random initialization every time the gradient is needed. When learning is performed using stochastic gradient descent, this means the chains must be burned in once per gradient step. This approach leads to the training procedure presented in Algorithm 18.1. The high cost of burning in the Markov chains in the inner loop makes this procedure computationally infeasible, but this procedure is the starting point that other more practical algorithms aim to approximate.

**Algorithm 18.1** A naive MCMC algorithm for maximizing the log-likelihood with an intractable partition function using gradient ascent.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow burn in. Perhaps 100 to train an RBM on a small image patch.
**while** Not converged **do**
  Sample a minibatch of $m$ examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$
  Initialize a set of $m$ samples $\{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(m)}\}$ to random values (e.g., from a uniform or normal distribution, or possibly a distribution with marginals matched to the model's marginals)
  **for** $i = 1$ to $k$ **do**
    **for** $j = 1$ to $m$ **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow$ gibbs_update($\tilde{\mathbf{x}}^{(j)}$)
    **end for**
  **end for**
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}$
**end while**

We can view the MCMC approach to maximum likelihood as trying to achieve balance between two forces, one pushing up on the model distribution where the data occurs, and another pushing down on the model distribution where the model samples occur. Fig. 18.1 illustrates this process. The two forces correspond to maximizing $\log \tilde{p}$ and minimizing $\log Z$. In this chapter, we assume the positive phase is tractable and may be performed exactly, but other chapters, especially chapter 19 deal with intractable positive phases. In this chapter, we present several approximations to the negative phase. Each of these approximations can

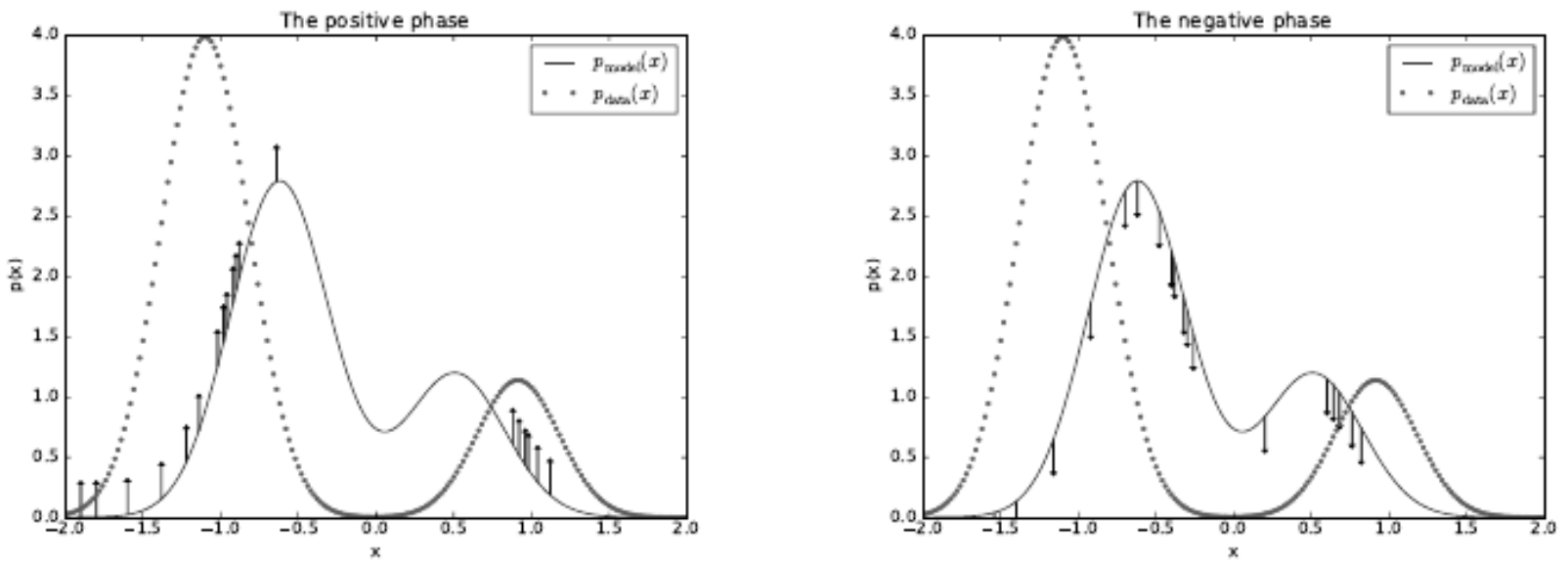


Figure 18.1: The view of Algorithm 18.1 as having a "positive phase" and "negative phase". Left) In the positive phase, we sample points from the data distribution, and push up on their unnormalized probability. This means points that are likely in the data get pushed up on more. Right) In the negative phase, we sample points from the model distribution, and push down on their unnormalized probability. This counteracts the positive phase's tendency to just add a large constant to the unnormalized probability everywhere. When the data distribution and the model distribution are equal, the positive phase has the same chance to push up at a point as the negative phase has to push down. At this point, there is no longer any gradient (in expectation) and training must terminate.

be understood as making the negative phase computationally cheaper but also making it push down in the wrong locations.

Because the negative phase involves drawing samples from the model's distribution, we can think of it as finding points that the model believes in strongly. Because the negative phase acts to reduce the probability of those points, they are generally considered to represent the model's incorrect beliefs about the world. They are frequently referred to in the literature as "hallucinations" or "fantasy particles." In fact, the negative phase has been proposed as a possible explanation for dreaming in humans and other animals (Crick and Mitchison, 1983), the idea being that the brain maintains a probabilistic model of the world and follows the gradient of $\log \tilde{p}$ while experiencing real events while awake and follows the negative gradient of $\log \tilde{p}$ to minimize $\log Z$ while sleeping and experiencing events sampled from the current model. This view explains much of the language used to describe algorithms with a positive and negative phase, but it has not been proven to be correct with neuroscientific experiments. In machine learning models, it is usually necessary to use the positive and negative phase simultaneously, rather than in separate time periods of wakefulness and REM sleep. As we will see in chapter 19.8, other machine learning algorithms draw samples from the model distribution for other purposes and such algorithms could also provide an account for the function of dream sleep.

Given this understanding of the role of the positive and negative phase of learning, we can attempt to design a less expensive alternative to Algorithm 18.1. The main cost of the naive MCMC algorithm is the cost of burning in the Markov chains from a random initialization at each step. A natural solution is to initialize the Markov chains from a distribution that is very close to the model distribution, so that the burn in operation does not take as many steps.

The *contrastive divergence* (CD, or CD-$k$ to indicate CD with $k$ Gibbs steps) algorithm initializes the Markov chain at each step with samples from the data distribution (Hinton, 2000). This approach is presented as Algorithm 18.2. Obtaining samples from the data distribution is free, because they are already available in the data set. Initially, the data distribution is not close to the model distribution, so the negative phase is not very accurate. Fortunately, the positive phase can still accurately increase the model's probability of the data. After the positive phase has had some time to act, the model distribution is closer to the data distribution, and the negative phase starts to become accurate.

**Algorithm 18.2** The contrastive divergence algorithm, using gradient ascent as the optimization procedure.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{x};\theta)$ to mix when initializedfrom $p_{\text{data}}$. Perhaps 1-20 to train an RBM on a small image patch.
**while** Not converged **do**
  Sample a minibatch of $m$ examples $\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)};\boldsymbol{\theta})$
  **for** $i = 1$ to $m$ **do**
    $\tilde{\mathbf{x}}^{(i)} \leftarrow \mathbf{x}^{(i)}$
  **end for**
  **for** $i = 1$ to $k$ **do**
    **for** $j = 1$ to $m$ **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs_update}(\tilde{\mathbf{x}}^{(j)})$
    **end for**
  **end for**
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)};\boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon\mathbf{g}$
**end while**

Of course, CD is still an approximation to the correct negative phase. The main way that CD qualitatively fails to implement the correct negative phase is that it fails to suppress "spurious modes" — regions of high probability that are far from actual training examples. Fig. 18.2 illustrates why this happens.

Essentially, it is because modes in the model distribution that are far from the data distribution will not be visited by Markov chains initialized at training points, unless $k$ is very large.

Carreira-Perpiñan and Hinton (2005) showed experimentally that the CD estimator is biased for RBMs and fully visible Boltzmann machines, in that it converges to different points than the maximum likelihood estimator. They argue that because the bias is small, CD could be used as an inexpensive way to initialize a model that could later be fine-tuned via more expensive MCMC methods. Bengio and Delalleau (2009) showed that CD can be interpreted as discarding the smallest terms of the correct MCMC update gradient, which explains the bias.

CD is useful for training shallow models like RBMs. These can in turn be stacked to initialize deeper models like DBNs or DBMs. However, CD does not provide much help for training deeper models directly. This is because it is difficult to obtain samples of the hidden units given samples of the visible units. Since the hidden units are not included in the data, initializing from training points cannot solve the problem. Even if we initialize the visible units from the data, we will still need to burn in a Markov chain sampling from the distribution over the hidden units conditioned on those visible samples. Most of the approximate inference techniques described in chapter 19 for approximately marginalizing out the hidden units cannot be used to solve this problem. This is because all of the approximate marginalization methods based on giving a lower bound on $\tilde{p}$ would give a lower bound on $\log Z$. We need to minimize $\log Z$, and minimizing a lower bound is not a useful operation.

The CD algorithm can be thought of as penalizing the model for having a Markov chain that changes the input rapidly when the input comes from the data. This means training with CD somewhat resembles autoencoder training. Even though CD is more biased than some of the other training methods, it can be useful for pre-training shallow models that will later be stacked. This is because the earliest models in the stack are encouraged to copy more information up to their latent variables, thereby making it available to the later models. This should be thought of more of as an often-exploitable side effect of CD training rather than a principled design advantage.

Sutskever and Tieleman (2010) showed that the CD update direction is not the gradient of any function. This allows for situations where CD could cycle forever, but in practice this is not a serious problem.

A different strategy that resolves many of the problems with CD is to initialize the Markov chains at each gradient step with their states from the previous gradient step. This approach was first discovered under the name *stochastic maximum likelihood* (SML) in the applied mathematics and statistics community (Younes, 1998) and later independently rediscovered under the name *persistent contrastive*

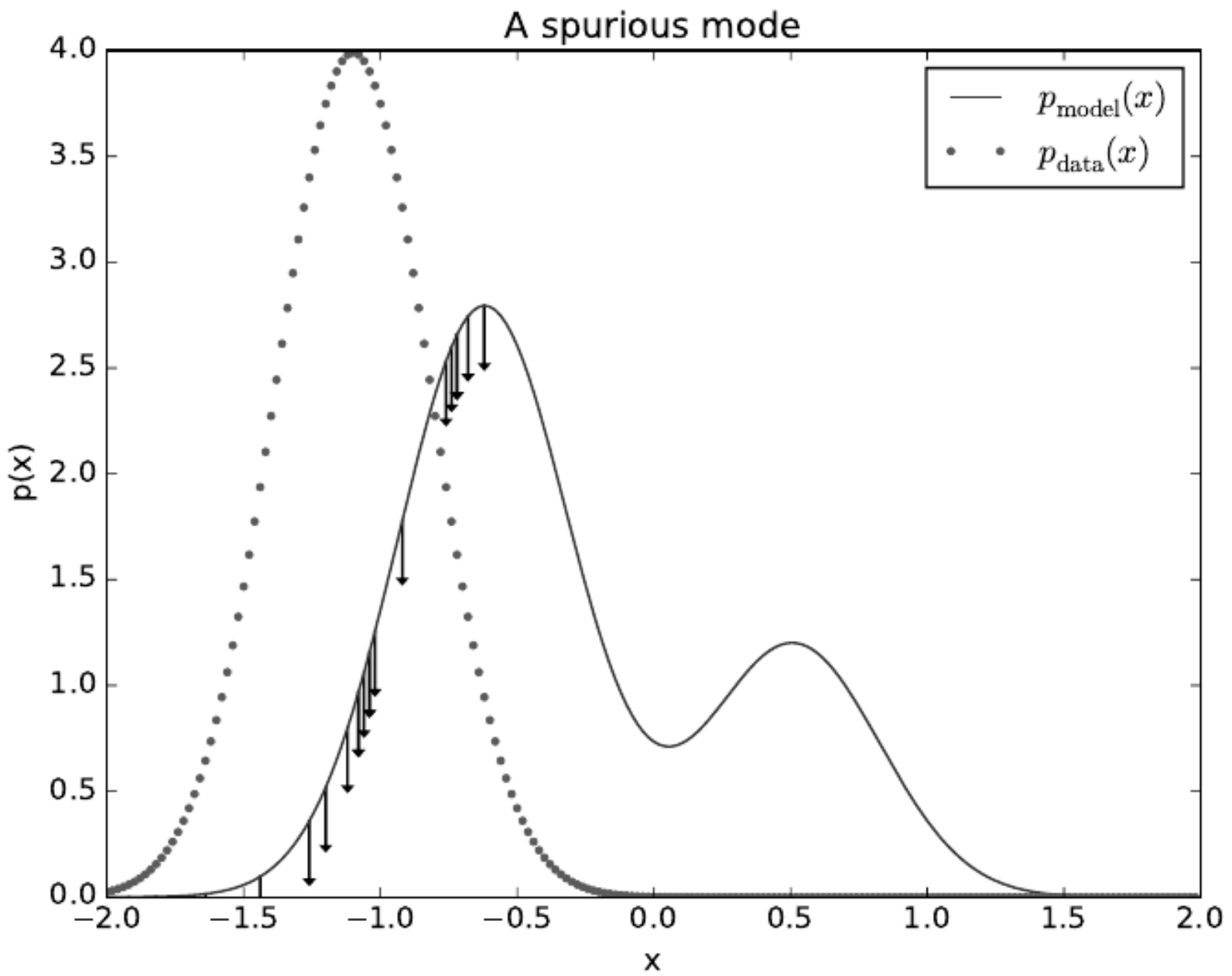


Figure 18.2: An illustration of how the negative phase of contrastive divergence (Algorithm 18.2) can fail to suppress spurious modes. A spurious mode is a mode that is present in the model distribution but absent in the data distribution. Because contrastive divergence initializes its Markov chains from data points and runs the Markov chain for only a few steps, it is unlikely to visit modes in the model that are far from the data points. This means that when sampling from the model, we will sometimes get samples that do not resemble the data. It also means that due to wasting some of its probability mass on these modes, the model will struggle to place high probability mass on the correct modes. Note that this figure uses a somewhat simplified concept of distance–the spurious mode is far from the correct mode along the number line in $\mathbb{R}$. This corresponds to a Markov chain based on making local moves with a single $x$ variable in $\mathbb{R}$. For most deep probabilistic models, the Markov chains are based on Gibbs sampling and can make non-local moves of individual variables but cannot move all of the variables simultaneously. For these problems, it is usually better to consider the edit distance between modes, rather than the Euclidean distance. However, edit distance in a high dimensional space is difficult to depict in a 2-D plot.

*divergence* (PCD, or PCD-$k$ to indicate the use of $k$ Gibbs steps per update) in the deep learning community (Tieleman, 2008). See Algorithm 18.3. The basic idea of this approach is that, so long as the steps taken by the stochastic gradient algorithm are small, then the model from the previous step will be similar to the model from the current step. It follows that the samples from the previous model's distribution will be very close to being fair samples from the current model's distribution, so a Markov chain initialized with these samples will not require much time to mix.

Because each Markov chain is continually updated throughout the learning process, rather than restarted at each gradient step, the chains are free to wander far enough to find all of the model's modes. SML is thus considerably more resistant to forming models with spurious modes than CD is. Moreover, because it is possible to store the state of all of the sampled variables, whether visible or latent, SML provides an initialization point for both the hidden and visible units. CD is only able to provide an initialization for the visible units, and therefore requires burn-in for deep models. SML is able to train deep models efficiently. Marlin *et al.* (2010) compared SML to many of the other criteria presented in this chapter. They found that SML results in the best test set log-likelihood for an RBM, and if the RBM's hidden units are used as features for an SVM classifier, SML results in the best classification accuracy.

SML is vulnerable to becoming inaccurate if $k$ is too small or $\epsilon$ is too large — in other words, if the stochastic gradient algorithm can move the model faster than the Markov chain can mix between steps. There is no known way to test formally whether the chain is successfully mixing between steps. Subjectively, if the learning rate is too high for the number of Gibbs steps, the human operator will be able to observe that there is much more variance in the negative phase samples across gradient steps rather than across different Markov chains. For example, a model trained on MNIST might sample exclusively 7s on one step. The learning process will then push down strongly on the mode corresponding to 7s, and the model might sample exclusively 9s on the next step.

Care must be taken when evaluating the samples from a model trained with SML. It is necessary to draw the samples starting from a fresh Markov chain initialized from a random starting point after the model is done training. The samples present in the persistent negative chains used for training have been influenced by several recent versions of the model, and thus can make the model appear to have greater capacity than it actually does.

Berglund and Raiko (2013) performed experiments to examine the bias and variance in the estimate of the gradient provided by CD and SML. CD proves to have low variance than the estimator based on exact sampling. SML has higher variance. The cause of CD's low variance is its use of the same training points

**Algorithm 18.3** The stochastic maximum likelihood / persistent contrastive divergence algorithm using gradient ascent as the optimization procedure.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{x}; \boldsymbol{\theta} + \epsilon \mathbf{g})$ toburn in, starting from samples from $p(\mathbf{x}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch, or 5-50 for a morecomplicated model like a DBM.
Initialize a set of $m$ samples $\{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(m)}\}$ to random values (e.g., from a uniform or normal distribution, or possibly a distribution with marginals matched to the model's marginals)
**while** Not converged **do**
  Sample a minibatch of $m$ examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training set.
  $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$
  **for** $i = 1$ to $k$ **do**
    **for** $j = 1$ to $m$ **do**
      $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs_update}(\tilde{\mathbf{x}}^{(j)})$
    **end for**
  **end for**
  $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta})$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}$
**end while**

in both the positive and negative phase. If the negative phase is initialized from different training points, the variance rises above that of the estimator based on exact sampling.

TODO– FPCD? TODO– Rates-FPCD?

TODO– mention that all these things can be coupled with enhanced samplers, which I believe are mentioned in the intro to graphical models chapter

One key benefit to the MCMC-based methods described in this section is that they provide an estimate of the gradient of $\log Z$, and thus we can essentially decompose the problem into the $\log \tilde{p}$ contribution and the $\log Z$ contribution. We can then use any other method to tackle $\log \tilde{p}(\mathbf{x})$, and just add our negative phase gradient onto the other method's gradient. In particular, this means that our positive phase can make use of methods that provide only a lower bound on $\tilde{p}$. Most of the other methods of dealing with $\log Z$ presented in this chapter are incompatible with bound-based positive phase methods.

## 18.3 Pseudolikelihood

Monte Carlo approximations to the partition function and its gradient directly confront the partition function. Other approaches sidestep the issue, by training the model without computing the partition function. Most of these approaches are based on the observation that it is easy to compute ratios of probabilities in an unnormalized probabilistic model. This is because the partition function appears in both the numerator and the denominator of the ratio and cancels out:

$$\frac{p(\mathbf{x})}{p(\mathbf{y})} = \frac{\frac{1}{Z}\tilde{p}(\mathbf{x})}{\frac{1}{Z}\tilde{p}(\mathbf{y})} = \frac{\tilde{p}(\mathbf{x})}{\tilde{p}(\mathbf{y})}.$$

The pseudolikelihood is based on the observation that conditional probabilities take this ratio-based form, and thus can be computed without knowledge of the partition function. Suppose that we partition $\mathbf{x}$ into $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, where $\mathbf{a}$ contains the variables we want to find the conditional distribution over, $\mathbf{b}$ contains the variables we want to condition on, and $\mathbf{c}$ contains the variables that are not part of our query.

$$p(\mathbf{a} \mid \mathbf{b}) = \frac{p(\mathbf{a}, p(\mathbf{b})}{p(\mathbf{b})} = \frac{p(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a},\mathbf{c}} p(\mathbf{a}, \mathbf{b}, \mathbf{c})} = \frac{\tilde{p}(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a},\mathbf{c}} \tilde{p}(\mathbf{a}, \mathbf{b}, \mathbf{c})}.$$

This quantity requires marginalizing out $\mathbf{a}$, which can be a very efficient operation provided that $\mathbf{a}$ and $\mathbf{c}$ do not contain very many variables. In the extreme case, $\mathbf{a}$ can be a single variable and $\mathbf{c}$ can be empty, making this operation require only as many evaluations of $\tilde{p}$ as there are values of a single random variable.

Unfortunately, in order to compute the log-likelihood, we need to marginalize out large sets of variables. If there are $n$ variables total, we must marginalize a set of size $n-1$. By the chain rule of probability,

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2 \mid x_1) + \cdots + p(x_n \mid \mathbf{x}_{1:n-1}).$$

In this case, we have made $\mathbf{a}$ maximally small, but $\mathbf{c}$ can be as large as $\mathbf{x}_{2:n}$. What if we simply move $\mathbf{c}$ into $\mathbf{b}$ to reduce the computational cost? This yields the *pseudolikelihood* (Besag, 1975) objective function:

$$\sum_{i=1}^{n} \log p(x_i \mid \mathbf{x}_{-i}).$$

If each random variable has $k$ different values, this requires only $k \times n$ evaluations of $\tilde{p}$ to compute, as opposed to the $k^n$ evaluations needed to compute the partition function.

This may look like an unprincipled hack, but it can be proven that estimation by maximizing the log pseudolikelihood is asymptotically consistent (Mase, 1995). Of course, in the case of datasets that do not approach the large sample limit, pseudolikelihood may display different behavior from the maximum likelihood estimator.

It is possible to trade computational complexity for deviation from maximum likelihood behavior by using the *generalized pseudolikelihood* estimator (Huang and Ogata, 2002). The generalized pseudolikelihood estimator uses $m$ different sets $S^{(i)}, i = 1, \ldots, m$ of indices variables that appear together on the left side of the conditioning bar. In the extreme case of $m = 1$ and $S^{(1)} = 1, \ldots, n$ the generalized pseudolikelihood recovers the log-likelihood. In the extreme case of $m = n$ and $S^{(i)} = \{i\}$, the generalized pseudolikelihood recovers the pseudolikelihood. The generalized pseudolikelihood objective function is given by

$$\sum_{i=1}^{m} \log p(\mathbf{x}_{S^{(i)}} \mid \mathbf{x}_{-S^{(i)}}).$$

The performance of pseudolikelihood-based approaches depends largely on how the model will be used. Pseudolikelihood tends to perform poorly on tasks that require a good model of the full joint $p(\mathbf{x})$, such as density estimation and sampling. However, it can perform better than maximum likelihood for tasks that require only the conditional distributions used during training, such as filling in small amounts of missing values. Generalized pseudolikelihood techniques are especially powerful if the data has regular structure that allows the $S$ index sets to be designed to capture the most important correlations while leaving out groups of variables that only have negligible correlation. For example, in natural images, pixels that are widely separated in space also have weak correlation, so the generalized pseudolikelihood can be applied with each $S$ set being a small, spatially localized window.

One weakness of the pseudolikelihood estimator is that it cannot be used with other approximations that provide only a lower bound on $\tilde{p}(\mathbf{x})$, such as variational inference, which will be covered in chapter 19.6. This is because $\tilde{p}$ appears in the denominator. A lower bound on the denominator provides only an upper bound on the expression as a whole, and there is no benefit to maximizing an upper bound. This makes it difficult to apply pseudolikelihood approaches to deep models such as deep Boltzmann machines, since variational methods are one of the dominant approaches to approximately marginalizing out the many layers of hidden variables that interact with each other. However, pseudolikelihood is still useful for deep learning, because it can be used to train single layer models, or deep models using approximate inference methods that are not based on lower bounds.

Pseudolikelihood has a much greater cost per gradient step than SML, due its explicit computation of all of the conditionals. However, generalized pseudolikelihood and similar criteria can still perform well if only one randomly selected conditional is computed per example (Goodfellow *et al.*, 2013b), thereby bringing the computational cost down to match that of SML.

Though the pseudolikelihood estimator does not explicitly minimize $\log Z$, it can still be thought of as having something resembling a negative phase. The denominators of each conditional distribution result in the learning algorithm suppressing the probability of all states that have only one variable differing from a training example.

## 18.4 Score Matching and Ratio Matching

Score matching (Hyvärinen, 2005b) provides another consistent means of training a model without estimating $Z$ or its derivatives. The strategy used by score matching is to minimize the expected squared difference between the derivatives of the model's log pdf with respect to the input and the derivatives of the data's log pdf with respect to the input:

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\mathbf{x}}||\nabla_{\mathbf{x}} \log p_{\text{model}}(\mathbf{x};\boldsymbol{\theta}) - \nabla_{\mathbf{x}} \log p_{\text{data}}(\mathbf{x})||_2^2.$$

Because the $\nabla_{\mathbf{x}} Z = 0$, this objective function avoids the difficulties associated with differentiating the partition function. However, it appears to have another difficult: it requires knowledge of the true distribution generating the training data, $p_{\text{data}}$. Fortunately, minimizing $J(\boldsymbol{\theta})$ turns out to be equivalent to minimizing

$$\tilde{J}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} \quad \frac{\partial^2}{\partial x_j^2} \log p_{\text{model}}(\mathbf{x};\boldsymbol{\theta}) + \frac{1}{2}\left(\frac{\partial}{\partial x_i} \log p_{\text{model}}(\mathbf{x};\boldsymbol{\theta})\right)^2\Bigg)$$

where $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ is the training set and $n$ is the dimensionality of $\mathbf{x}$.

Because score matching requires taking derivatives with respect to $\mathbf{x}$, it is not applicable to models of discrete data. However, the latent variables in the model may be discrete.

Like the pseudolikelihood, score matching only works when we are able to evaluate $\log \tilde{p}(\mathbf{x})$ and its derivatives directly. It is not compatible with methods that only provide a lower bound on $\log \tilde{p}(\mathbf{x})$, because we are not able to conclude anything about the relationship between the derivatives and second derivatives of the lower bound, and the relationship of the true derivatives and second derivatives needed for score matching. This means that score matching cannot be applied to estimating models with complicated interactions between the hidden units, such

as sparse coding models or deep Boltzmann machines. Score matching can be used to pretrain the first hidden layer of a larger model. Score matching has not been applied as a pretraining strategy for the deeper layers of a larger model, because the hidden layers of such models usually contain some discrete variables.

While score matching does not explicitly have a negative phase, it can be viewed as a version of contrastive divergence using a specific kind of Markov chain (Hyvärinen, 2007a). The Markov chain in this case is not Gibbs sampling, but rather a different approach that makes local moves guided by the gradient. Score matching is equivalent to CD with this type of Markov chain when the size of the local moves approaches zero.

Lyu (2009) generalized score matching to the discrete case (but made an error in their derivation that was corrected by Marlin *et al.* (2010)). Marlin *et al.* (2010) found that *generalized score matching* (GSM) does not work in high dimensional discrete spaces where the observed probability of many events is 0.

A more successful approach to extending the basic ideas of score matching to discrete data is *ratio matching* (Hyvärinen, 2007b). Ratio matching applies specifically to binary data. Ratio matching consists of minimizing the following objective function:

$$J^{(\mathrm{RM})}(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(\frac{1}{1+\frac{p_{\mathrm{model}}\left(\mathbf{x}^{(i)};\boldsymbol{\theta}\right)}{p_{\mathrm{model}}\left(f(\mathbf{x}^{(i)},j);\boldsymbol{\theta}\right)}}\right)^2$$

where $f(\mathbf{x}, j)$ return $\mathbf{x}$ with the bit at position $j$ flipped. Ratio matching avoids the partition function using the same trick as the pseudolikelihood estimator: in a ratio of two probabilities, the partition function cancels out. Marlin *et al.* (2010) found that ratio matching outperforms SML, pseudolikelihood, and GSM in terms of the ability of models trained with ratio matching to denoise test set images.

Like the pseudolikelihood estimator, ratio matching requires $n$ evaluations of $\tilde{p}$ per data point, making its computational cost per update roughly $n$ times higher than that of SML.

Like the pseudolikelihood estimator, ratio matching can be thought of as pushing down on all fantasy states that have only one variable different from a training example. Since ratio matching applies specifically to binary data, this means that it acts on all fantasy states within Hamming distance 1 of the data.

Ratio matching can also be useful as the basis for dealing with high-dimensional sparse data, such as word count vectors. This kind of data poses a challenge for MCMC-based methods because the data is extremely expensive to represent in dense format, yet the MCMC sampler does not yield sparse values until the model

has learned to represent the sparsity in the data distribution. Dauphin and Bengio (2013) overcame this issue by designing an unbiased stochastic approximation to ratio matching. The approximation evaluates only a randomly selected subset of the terms of the objective, and does not require the model to generate complete fantasy samples.

## 18.5 Denoising Score Matching

In some cases we may wish to regularize score matching, by fitting a distribution

$$p_{\text{smoothed}}(\mathbf{x}) = \int p_{\text{data}}(\mathbf{x} + \mathbf{y}) q(y \mid \mathbf{x}) dy$$

rather than the true $p_{\text{data}}$. This is especially useful because in practice we usually do not have access to the true $p_{\text{data}}$ but rather only an empirical distribution defined by samples from it. Any consistent estimator will, given enough capacity, make $p_{\text{model}}$ into a set of Dirac distributions centered on the training points. Smoothing by $q$ helps to reduce this problem, at the loss of the asymptotic consistency property. Kingma and LeCun (2010b) introduced a procedure for performing regularized score matching with the smoothing distribution $q$ being normally distributed noise.

Surprisingly, some denoising autoencoder training algorithms correspond to training energy-based models with denoising score matching (Vincent, 2011b). The denoising autoencoder variant of the algorithm is significantly less computationally expensive than score matching. Swersky *et al.* (2011) showed how to derive the denoising autoencoder for any energy-based model of real data. This approach is known as *denoising score matching* (SMD).

## 18.6 Noise-Contrastive Estimation

Most techniques for estimating models with intractable partition functions do not provide an estimate of the partition function. SML and CD estimate only the gradient of the log partition function, rather than the partition function itself. Score matching and pseudolikelihood avoid computing quantities related to the partition function altogether.

*Noise-contrastive estimation (NCE)* (Gutmann and Hyvarinen, 2010) takes a different strategy. In this approach, the probability distribution by the model is represented explicitly as

$$\log p_{\text{model}}(\mathbf{x}) = \log \tilde{p}_{\text{model}}(\mathbf{x}; \theta) + c,$$

where $c$ is explicitly introduced as an approximation of $-\log Z(\theta)$. Rather than estimating only $\theta$, the noise contrastive estimation procedure treats $c$ as just another parameter and estimates $\theta$ and $c$ simultaneously, using the same algorithm for both. The resulting thus may not correspond exactly to a valid probability distribution, but will become closer and closer to being valid as the estimate of $c$ improves.[2]

Such an approach would not be possible using maximum likelihood as the criterion for the estimator. The maximum likelihood criterion would choose to set $c$ arbitrarily high, rather than setting $c$ to create a valid probability distribution.

NCE works by reducing the unsupervised learning problem of estimating $p(\mathbf{x})$ to a supervised learning problem. This supervised learning problem is constructed in such a way that maximum likelihood estimation in this supervised learning problem defines an asymptotically consistent estimator of the original problem.

Specifically, we introduce a second distribution, the *noise distribution* $p_{\text{noise}}(\mathbf{x})$. The noise distribution should be tractable to evaluate and to sample from. We can now construct a model over both $\mathbf{x}$ and a new, binary class variable $y$. In the new joint model, we specify that

$$p_{\text{joint_model}}(y = 1) = \frac{1}{2},$$

$$p_{\text{joint_model}}(\mathbf{x} \mid y = 1) = p_{\text{model}}(\mathbf{x}),$$

and

$$p_{\text{joint_model}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x}).$$

In other words, $y$ is a switch variable that determines whether we will generate $\mathbf{x}$ from the model or from the noise distribution.

We can construct a similar joint model of training data. In this case, the switch variable determines whether we draw $\mathbf{x}$ from the *data* or from the noise distribution. Formally, $p_{\text{train}}(y = 1) = \frac{1}{2}$, $p_{\text{train}}(\mathbf{x} \mid y = 1) = p_{\text{data}}(\mathbf{x})$, and $p_{\text{train}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x})$.

We can now just use standard maximum likelihood learning on the *supervised* learning problem of fitting $p_{\text{joint_model}}$ to $p_{\text{train}}$:

$$\theta, c = \arg\max_{\theta, c} \mathbb{E}_{\mathbf{x}, y \sim p_{\text{train}}} \log p_{\text{joint_model}}(y \mid \mathbf{x}).$$

It turns out that $p_{\text{joint_model}}$ is essentially a logistic regression model applied to the difference in log probabilities of the model and the noise distribution:

$$p_{\text{joint_model}}(y = 1 \mid \mathbf{x}) = \frac{p_{\text{model}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})}$$

[2] NCE is also applicable to problems with a tractable partition function, where there is no need to introduce the extra parameter . However, it has generated the most interest as a means of estimating models with difficult partition functions.

$$
\begin{aligned}
&= \frac{1}{1 + \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}} \\
&= \frac{1}{1 + \exp\left(\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right)} \\
&= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right) \\
&= \sigma\left(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})\right).
\end{aligned}
$$

NCE is thus simple to apply so long as $\log \tilde{p}_{\text{model}}$ is easy to backpropagate through, and, as specified above, noise is easy to evaluate (in order to evaluate $p_{\text{joint_model}}$) and sample from (in order to generate the training data).

NCE is most successful when applied to problems with few random variables, but can work well even if those random variables can take on a high number of values. For example, it has been successfully applied to modeling the conditional distribution over a word given the context of the word (Mnih and Kavukcuoglu, 2013). Though the word may be drawn from a large vocabulary, there is only one word.

When NCE is applied to problems with many random variables, it becomes less efficient. The logistic regression classifier can reject a noise sample by identifying any one variable whose value is unlikely. This means that learning slows down greatly after $p_{\text{model}}$ has learned the basic marginal statistics. Imagine learning a model of images of faces, using unstructured Gaussian noise as $p_{\text{noise}}$. If $p_{\text{model}}$ learns about eyes, it can reject almost all unstructured noise samples without having learned anything other facial features, such as mouths.

The constraint that $p_{\text{noise}}$ must be easy to evaluate and easy to sample from can be overly restrictive. When $p_{\text{noise}}$ is simple, most samples are likely to be too obviously distinct from the data to force $p_{\text{model}}$ to improve noticeably.

Like score matching and pseudolikelihood, NCE does not work if only a lower bound on $\tilde{p}$ is available. Such a lower bound could be used to construct a lower bound on $p_{\text{joint_model}}(y = 1 \mid \mathbf{x})$, but it can only be used to construct an upper bound on $p_{\text{joint_model}}(y = 0 \mid \mathbf{x})$, which appears in half the terms of the NCE objective. Likewise, a lower bound on $p_{\text{noise}}$is not useful, because it provides only an upper bound on $p_{\text{joint_model}}(y = 1 \mid \mathbf{x})$.

TODO– put herding in this chapter? and if not, where to put it?

TODO– cite the Bregman divergence paper?

## 18.7 Estimating the Partition Function

While much of this chapter is dedicated to describing methods for working around the unknown and intractable partition function $Z(\theta)$ associated with an undi-

rected graphical model; in this section we will discuss several methods for directly estimating the partition function.

Estimating the partition function can be important because we require it if we wish to compute the normalized likelihood of data. This is often important in *evaluating* the model, monitoring training performance, and comparing models to each other.

For example, imagine we have two models: $\mathcal{M}_A : p_A(\mathbf{x};\theta_A) = \frac{1}{Z_A}\tilde{p}_A(\mathbf{x};\theta_A)$ and $\mathcal{M}_B : p_B(\mathbf{x};\theta_B) = \frac{1}{Z_B}\tilde{p}_B(\mathbf{x};\theta_B)$. A common way to compare the models is to evaluate the likelihood of an i.i.d. test dataset of size $N_{\text{test}}$: $D_{\text{test}} = \{\mathbf{x}_i^{(t)}\}^{N_{\text{test}}}$ under both models. If $\prod_t p_A(\mathbf{x}^{(t)}\theta_A) > \prod_t p_B(\mathbf{x}^{(t)}\theta_B)$ or equivalently if $\sum_t \ln p_A(\mathbf{x}^{(t)}\theta_A) - \sum_t \ln p_B(\mathbf{x}^{(t)}\theta_B) > 0$, then we say that $\mathcal{M}_A$ is a better model than $\mathcal{M}_B$ (or, at least, it is a better model of the test set). More specifically, to say that $\mathcal{M}_A$ is better than $\mathcal{M}_B$, we need that:

$$\sum_t \ln p_A(\mathbf{x}^{(t)}\theta_A) - \sum_t \ln p_B(\mathbf{x}^{(t)}\theta_B) > 0$$

$$\sum_t \Big(\ln \tilde{p}_A(\mathbf{x}^{(t)}\theta_A) - \ln Z(\theta_A)\Big) - \sum_t \Big(\ln \tilde{p}_B(\mathbf{x}^{(t)}\theta_B) - \ln Z(\theta_B)\Big) > 0$$

$$\sum_t \Big(\ln \tilde{p}_A(\mathbf{x}^{(t)}\theta_A) - \ln \tilde{p}_B(\mathbf{x}^{(t)};\theta_B)\Big) - N_{test}\ln Z(\theta_A) + N_{test}\ln Z(\theta_B) > 0$$

$$\sum_t \ \ln \frac{\tilde{p}_A(\mathbf{x}^{(t)};\theta_A)}{\tilde{p}_B(\mathbf{x}^{(t)};\theta_B)}\Bigg) - N_{test}\ln \frac{Z(\theta_A)}{Z(\theta_B)} > 0.$$

TODO: too much repetition of "to know" TODO: be more specific about what it means to "compare", does this mean to take the ratio of two likelihoods? In order to compare two models we need to compare not only their unnormalized probabilities, but also their partition functions. It is interesting to note that, in order to compare these models, we do not actually need to know the value of their partition function. We need only know their ratio. That is, we need to know their relative value, up to some shared constant. If, however, we wanted to know the actual probability of the test data under either $\mathcal{M}_A$ or $\mathcal{M}_B$, we would need to know the actual value of the partition functions. That said, if we knew the ratio of two partition functions, $R = \frac{Z(\theta_B)}{Z(\theta_A)}$, and we knew the actual value of just one of the two, say $Z(\theta_A)$, we can compute the value of the other:

$$Z(\theta_B) = R \times Z(\theta_A) = \frac{Z(\theta_B)}{Z(\theta_A)} Z(\theta_A)$$

We can make use of this observation to estimate the partition functions of undirected graphical models.

For a given probability distribution, say $p_1(\mathbf{x})$, the partition function is defined as

$$Z_1 = \int \tilde{p}_1(\mathbf{x})\ d\mathbf{x} \tag{18.5}$$

where the integral is over the domain of $\mathbf{x}$. Of course, in the case of discrete $\mathbf{x}$, we replace the integral with a sum. For convenience, we have suppressed the dependency of both the partition functions and the unnormalized distributions on the model parameters.

A simple way to estimate the partition function is to use a Monte Carlo method such as simple importance sampling. Here we consider a proposal distribution, say $p_0(\mathbf{x})$, from which we can *sample* and *evaluate* both its partition function $Z_0$, and its unnormalized distribution $\tilde{p}_0(\mathbf{x})$.

$$\begin{aligned} Z_1 &= \int \tilde{p}_1(\mathbf{x})\ d\mathbf{x} \\ &= \int \frac{p_0(\mathbf{x})}{p_0(\mathbf{x})} \tilde{p}_1(\mathbf{x})\ d\mathbf{x} \\ &= Z_0 \int p_0(\mathbf{x}) \frac{\tilde{p}_1(\mathbf{x})}{\tilde{p}_0(\mathbf{x})}\ d\mathbf{x} \\ \frac{Z_1}{Z_0} &\approx \sum_{k=1}^{K} \frac{\tilde{p}(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t.} : \mathbf{x}^{(k)} \sim p_0 \end{aligned} \tag{18.6}$$

In the last line, we make a Monte Carlo approximation of the integral using samples drawn from $p_0(\mathbf{x})$ and then weigh each sample with the ratio of the unnormalized $\tilde{p}_1$ and the proposal $p_0$ each evaluated at that sample.

If the distribution $p_0$ is close to $p_1$, this can be an effective way of estimating the partition function (Minka, 2005). Unfortunately, most of the time $p_1$ is both complicated, i.e. multimodal, and defined over a high dimensional space. It is difficult to find a tractable $p_0$ that is simple enough to evaluate while still being close enough to $p_1$ to result in a high quality approximation. If $p_0$ and $p_1$ are not close, most samples from $p_0$ will have low probability under $p_1$ and therefore make (relatively) negligible contribution to the sum in Eq. 18.6. Having few samples with significant weights in this sum will result in an estimator with high variance, i.e. a poor quality estimator.

TODO: quantify this

We now turn to two related strategies developed to cope with the challenging task of estimating partition functions for complex distributions over high-dimensional spaces: annealed importance sampling and Bennett's ratio acceptance method. Both start with the simple importance sampling strategy introduced above and both attempt to overcome the problem of the proposal $p_0$ being

too far from $p_1$ by introducing intermediate distributions that attempt to *bridge the gap* between $p_0$ and $p_1$.

### 18.7.1 Annealed Importance Sampling

TODO– describe how this is the main way of evaluating $p(\mathbf{x})$ when you want to get test set likelihoods but can't be used for training TODO– also mention Guillaume's "tracking the partition function" paper?

In situations where $D_{\mathrm{KL}}(p_0|p_1)$ is large (i.e., where there is little overlap between $p_0$ and $p_1$), AIS attempts to bridge the gap by introducing *intermediate distributions*. Consider a sequence of distributions $p_{\eta_0}, \ldots, p_{\eta_n}$, with $0 = \eta_0 < \eta_1 < \cdots < \eta_{n-1} < \eta_n = 1$ so that the first and last distributions in the sequence are $p_0$ and $p_1$ respectively. We can now write the ratio $\frac{Z_1}{Z_0}$ as

$$\begin{aligned}
\frac{Z_1}{Z_0} &= \frac{Z_1}{Z_0}\frac{Z_{\eta_1}}{Z_{\eta_1}} \cdots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-1}}} \\
&= \frac{Z_{\eta_1}}{Z_0}\frac{Z_{\eta_2}}{Z_{\eta_1}} \cdots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-2}}}\frac{Z_1}{Z_{\eta_{n-1}}} \\
&= \prod_{j=0}^{n-1} \frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}
\end{aligned} \tag{18.7}$$

Provided the distributions $p_{\eta_j}$ and $p_{\eta_{j+1}}$, for all $0 \leq j \leq n-1$, are sufficiently close, we can reliably estimate each of the factors $\frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}$ using simple importance sampling and then use these to obtain an estimate of $\frac{Z_1}{Z_0}$.

Where do these intermediate distributions come from? Just as the original proposal distribution $p_0$ is a design choice, so is the sequence of distributions $p_{\eta_1} \ldots p_{\eta_{n-1}}$. That is, it can be specifically constructed to suit the problem domain. One general-purpose and popular choice for the intermediate distributions is to use the weighted geometric average of the target distribution $p_1$ and the starting proposal distribution (for which the partition function is known) $p_0$:

$$p_{\eta_j} \propto p_1^{\eta_j} p_0^{1-\eta_j} \tag{18.8}$$

In order to sample from these intermediate distributions, we define a series of Markov chain transition functions $T_{\eta_j}(\mathbf{x}', \mathbf{x})$ that define the probability distribution of transitioning from $\mathbf{x}'$ to $\mathbf{x}$. $T_{\eta_j}(\mathbf{x}', \mathbf{x})$ is defined to leave $p_{\eta_j}(\mathbf{x})$ invariant:

$$p_{\eta_j}(\mathbf{x}) = \int p_{\eta_j}(\mathbf{x}')T_{\eta_j}(\mathbf{x}', \mathbf{x})\, d\mathbf{x}' \tag{18.9}$$

These transitions may be constructed as any Markov chain Monte Carlo method (e.g.. Metropolis-Hastings, Gibbs), including methods involving multiple scans or other iterations.

The AIS sampling strategy is then to generate samples from $p_0$ and then use the transition operators to sequentially generate samples from the intermediate distributions until we arrive at samples from the target distribution $p_1$:

- for $k = 1 \dots K$
  - Sample $\boldsymbol{x}_{\eta_1}^{(k)} \sim p_0(\mathbf{x})$
  - Sample $vx_{\eta_2}^{(k)} \sim T_{\eta\,1}(\boldsymbol{x}_{\eta_1}^{(k)}, \mathbf{x})$
  - $\dots$
  - Sample $\boldsymbol{x}_{\eta_n\,1}^{(k)} \sim T_{\eta_n\,2}(\boldsymbol{x}_{\eta_n\,2}^{(k)}, \mathbf{x})$
  - Sample $\boldsymbol{x}_{\eta_n}^{(k)} \sim T_{\eta_n\,1}(\boldsymbol{x}_{\eta_n\,1}^{(k)}, \mathbf{x})$
- end

For sample $k$, we can derive the importance weight by chaining together the importance weights for the jumps between the intermediate distributions given in Eq. 18.7.

$$w^{(k)} = \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta\,1}^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})} \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta\,2}^{(k)})}{\tilde{p}_1(\mathbf{x}_{\eta_1}^{(k)})} \dots \frac{\tilde{p}_1(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_n\,1}(\mathbf{x}_{\eta_n\,1}^{(k)})} \tag{18.10}$$

To avoid computational issues such as overflow, it is probably best to do the computation in log space, i.e. $\ln w^{(k)} = \ln \tilde{p}_{\eta\,1}(\mathbf{x}) - \ln \tilde{p}_0(\mathbf{x}) + \dots$.

With the sampling procedure thus define and the importance weights given in Eq. 18.10, the estimate of the ratio of partition functions is given by:

$$\frac{Z_1}{Z_0} \approx \frac{1}{K} \sum_{k=1}^{K} w^{(k)} \tag{18.11}$$

In order to verify that this procedure defines a valid importance sampling scheme, we can show that the AIS procedure corresponds to simple importance sampling on an extended state space with points sampled over the product space: $[\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_n\,1}, \mathbf{x}_1]$ Neal (2001).

We define the distribution over the extended space as:

$$\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_n\,1}, \mathbf{x}_1) = \tilde{p}_1(\mathbf{x}_1)\tilde{T}_{\eta_n\,1}(\mathbf{x}_1, \mathbf{x}_{\eta_n\,1})\tilde{T}_{\eta_n\,2}(\mathbf{x}_{\eta_n\,1}, \mathbf{x}_{\eta_n\,2}) \dots \tilde{T}_{\eta_1}(\mathbf{x}_{\eta_2}, \mathbf{x}_{\eta_1}) \tag{18.12}$$

where $\tilde{T}_a$ is the reverse of the transition operator defined by $T_a$ (via an application of Bayes' rule):

$$\tilde{T}_a(\mathbf{x}, \mathbf{x}') = \frac{p_a(\mathbf{x}')}{p_a(\mathbf{x})} T_a(\mathbf{x}', \mathbf{x}) = \frac{\tilde{p}_a(\mathbf{x}')}{\tilde{p}_a(\mathbf{x})} T_a(\mathbf{x}', \mathbf{x}). \tag{18.13}$$

Plugging the above into the expression for the joint distribution on the extended state space given in Eq. 18.12, we get:

$$\begin{aligned}
&\tilde{p}(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{x}_0) \\
&= \tilde{p}(\mathbf{x}_1) \frac{\tilde{p}_2(\mathbf{x}_2)}{\tilde{p}_1(\mathbf{x}_1)} T_1(\mathbf{x}_2, \mathbf{x}_1) \frac{\tilde{p}_3(\mathbf{x}_3)}{\tilde{p}_2(\mathbf{x}_2)} T_2(\mathbf{x}_3, \mathbf{x}_2) \ldots \frac{\tilde{p}_n(\mathbf{x}_n)}{\tilde{p}_{n-1}(\mathbf{x}_{n-1})} T_{n-1}(\mathbf{x}_n, \mathbf{x}_{n-1}) \\
&= \frac{\tilde{p}_1(\mathbf{x}_1)}{\tilde{p}_2(\mathbf{x}_2)} T_1(\mathbf{x}_2, \mathbf{x}_1) \frac{\tilde{p}_3(\mathbf{x}_3)}{\tilde{p}_2(\mathbf{x}_2)} T_2(\mathbf{x}_3, \mathbf{x}_2) \ldots \frac{\tilde{p}_n(\mathbf{x}_n)}{\tilde{p}_{n-1}(\mathbf{x}_{n-1})} T_{n-1}(\mathbf{x}_n, \mathbf{x}_{n-1}) \tilde{p}_n(\mathbf{x}_n)
\end{aligned} \tag{18.14}$$

If we now consider the sampling scheme given above as a means of generating samples from a proposal distribution $q$ over the extended state, with its distribution given by:

$$q(\mathbf{x}_{\eta_1}, \ldots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) = p_0(\mathbf{x}_{\eta_1}) T_{\eta_1}(\mathbf{x}_{\eta_1}, \mathbf{x}_{\eta_2}) \ldots T_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \tag{18.15}$$

We have a joint distribution on the extended space given by Eq. 18.14. Taking $q(\mathbf{x}_{\eta_1}, \ldots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)$ as the proposal distribution on the extended state space from which we will draw samples, it remains to determine the importance weights:

$$w^{(k)} = \frac{\tilde{p}(\mathbf{x}_{\eta_1}, \ldots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)}{q(\mathbf{x}_{\eta_1}, \ldots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)} = \frac{\tilde{p}_1(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}^{(k)})} \cdots \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\mathbf{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})} \tag{18.16}$$

These weights are the same as proposed for AIS. Thus we can interpret AIS as simple importance sampling applied to an extended state and its validity follows immediately from the validity of importance sampling.

Annealed importance sampling (AIS) was first discovered by Jarzynski (1997) and then again, independently, by Neal (2001). It is currently the most common way of estimating the partition function for undirected probabilistic models. The reasons for this may have more to do with the publication of an influential paper Salakhutdinov and Murray (2008) describing its application to estimating the partition function of restricted Boltzmann machines and deep belief networks than with any inherent advantage the method has over the other method described below.

A discussion of the properties of the AIS estimator (e.g.. its variance and efficiency) can be found in Neal (2001).

### 18.7.2 Bridge Sampling

Bridge sampling Bennett (1976) is another method that, like AIS, addresses the shortcomings of importance sampling; however it does so in a different but related manner. Rather than chaining together a series of intermediate distributions, bridge sampling relies on a single distribution $p_*$, known as the bridge, to interpolate between a distribution with known partition function, $p_0$, and a distribution $p_1$ for which we are trying to estimate the partition function $Z_1$.

Bridge sampling estimates the ratio $Z_1/Z_0$ as the ratio of the expected importance weights between $\tilde{p}_0$ and $\tilde{p}_*$ and between $\tilde{p}_1$ and $\tilde{p}_*$:

$$\frac{Z_1}{Z_0} \approx \sum_{k=1}^{K} \frac{\tilde{p}_*(x_0^{(k)})}{\tilde{p}_0(x_0^{(k)})} \Bigg/ \sum_{k=1}^{K} \frac{\tilde{p}_*(x_1^{(k)})}{\tilde{p}_1(x_1^{(k)})} \tag{18.17}$$

If the bridge distribution $p_*$ is chosen carefully to have a large overlap of support with both $p_0$ and $p_1$, then bridge sampling can allow the distance between two distributions (or more formally, $D_{\mathrm{KL}}(p_0|p_1)$) to be much larger than with standard importance sampling.

It can be shown than the optimal bridging distribution is given by $p_*^{(opt)}(x) \propto \frac{\tilde{p}_0(x)\tilde{p}_1(x)}{r\tilde{p}_0(x)+\tilde{p}_1(x)}$ where $r = Z_1/Z_0$.

This appears to be an unworkable solution as it would seem to require the very quantity we are trying to estimate, i.e. $Z_1/Z_0$. However, it is possible to start with a coarse estimate of $r$ and use the resulting bridge distribution to refine our estimate recursively Neal (2005).

TODO: illustration of the bridge distribution

### 18.7.3 Extensions

**Linked importance sampling** Both AIS and bridge sampling have their advantages. If $D_{\mathrm{KL}}(p_0|p_1)$ is not too large (i.e. if $p_0$ and $p_1$ are sufficiently close) bridge sampling can be a more effective means of estimating the ratio of partition functions than AIS. If, however, the two distributions are too far apart for a single distribution $p_*$ to bridge the gap then one can at least use AIS with potential many intermediate distributions to span the distance between $p_0$ and $p_1$. Neal (2005) showed how his linked importance sampling method leveraged the power of the bridge sampling strategy to bridge the intermediate distributions used in AIS to significantly improve the overall partition function estimates.

**Tracking the partition function while training** Using a combination of bridge sampling, AIS and parallel tempering, Desjardins *et al.* (2011) devised a scheme to track the partition function of an RBM throughout the training

process. The strategy is based on the maintenance of independent estimates of the partition functions of the RBM at every temperature operating in the parallel tempering scheme. The authors combined bridge sampling estimates of the ratios of partition functions of neighboring chains (i.e. from parallel tempering) with AIS estimates across time to come up with a low variance estimate of the partition functions at every iteration of learning.

# Chapter 19

# Approximate inference

TODO: somewhere in this chapter, point out that variational inference implicitly defines a recurrent net, stochastic approximate inference implicitly defines a stochastic recurrent net

Misplaced TODO: discussion of the different directions of the KL divergence, and the effects on ignoring / preserving modes

Many probabilistic models are difficult to train because it is difficult to perform inference in them. In the context of deep learning, we usually have a set of visible variables $\boldsymbol{v}$ and a set of latent variables $\boldsymbol{h}$. The challenge of inference usually refers to the difficult problem of computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ or taking expectations with respect to it. Such operations are often necessary for tasks like maximum likelihood learning.

Many simple graphical models with only one hidden layer, such as restricted Boltzmann machines and probabilistic PCA are defined in a way that makes inference operations like computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ or taking expectations with respect to it simple. Unfortunately, most graphical models with multiple layers of hidden variables, such as deep belief networks and deep Boltzmann machines have intractable posterior distributions. Exact inference requires an exponential amount of time in these models. Even some models with only a single layer, such as sparse coding, have this problem.

Intractable inference problems usually arise from interactions between latent variables in a structured graphical model. See Fig. 19.1 for some examples. These interactions may be due to direct interactions in undirected models or "explaining away" interactions between mutual ancestors of the same visible unit in directed models.

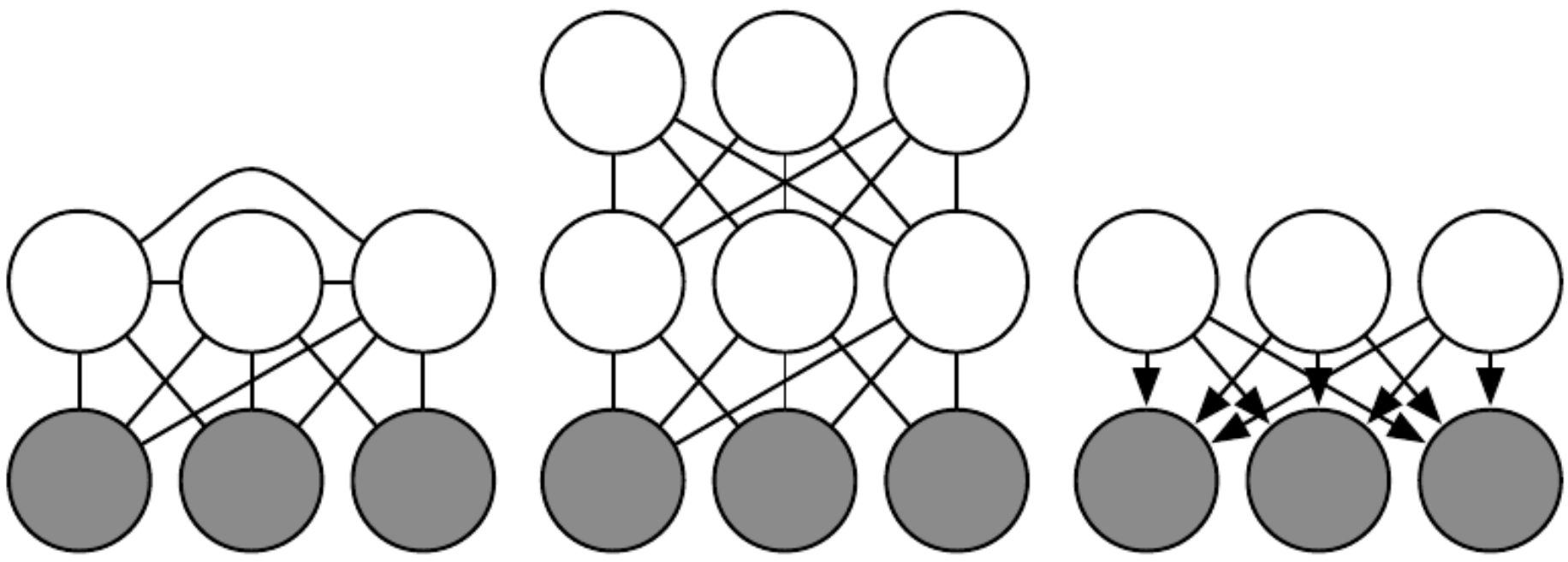

Figure 19.1: Intractable inference problems are usually the result of interactions between latent variables in a structured graphical model. These can be due to direct edges, or due to paths that are activated when the child of a V-structure is observed. Left) A semi-restricted Boltzmann machine with connections between hidden units. These direct connections between latent variables make the posterior distribution complicated. Center) A deep Boltzmann machine, organized into layers of variables without intra-layer connections, still has an intractable posterior distribution due to the connections between layers. Right) This directed model has interactions between latent variables when the visible variables are observed, because ever two latent variables are co-parents. Note that it is still possible to have these graph structures yet have tractable inference. For example, probabilistic PCA has the graph structure shown in the right, yet simple inference due to special properties of the specific conditional distributions it uses (linear-Gaussian conditionals with mutually orthogonal basis vectors). MISPLACED TODO–make sure probabilistic PCA is at least defined somewhere in the book

## 19.1 Inference as Optimization

Many approaches to confronting the problem of difficult inference make use of the observation that exact inference can be described as an optimization problem.

Specifically, assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We would like to compute the log probability of the observed data, $\log p(\boldsymbol{v}; \boldsymbol{\theta})$. Sometimes it is too difficult to compute $\log p(\boldsymbol{v}; \boldsymbol{\theta})$ if it is costly to marginalize out $\boldsymbol{h}$. Instead, we can compute a lower bound on it. This bound is called the *evidence lower bound* (ELBO). Other names for this lower bound include the negative *variational free energy* and the negative *Helmholtz free energy*. Specifically, this lower bound is defined as TODO– figure out why I was making q be bm in some but not all places

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D \quad (q(\boldsymbol{h}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$$

where $q$ is an arbitrary probability distribution over $\boldsymbol{h}$.

TODO: the below equations framebust It is straightforward to see that this is a lower bound on $\log p(\boldsymbol{v})$:

$$\begin{aligned}
\ln p(\boldsymbol{v}) = \ln p(\boldsymbol{v}) + \quad & q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \\
= \quad & q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} + q(\boldsymbol{h} \mid \boldsymbol{v}) \log p(\boldsymbol{v}) \\
= \quad & q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - \ln p(\boldsymbol{v}) \\
= \quad & q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{p(\boldsymbol{v}) q(\boldsymbol{h} \mid \boldsymbol{v})} \\
= \quad & q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{v}, \boldsymbol{h})}{q(\boldsymbol{h} \mid \boldsymbol{v})} - q(\boldsymbol{h} \mid \boldsymbol{v}) \ln \frac{p(\boldsymbol{h} \mid \boldsymbol{v})}{q(\boldsymbol{h} \mid \boldsymbol{v})} \\
= \mathcal{L}(q) + \mathrm{KL}(q \| p)
\end{aligned}$$

Because the difference $\log p(\boldsymbol{v})$ and $\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q})$ is given by the KL-divergence and because the KL-divergence is always non-negative, we can see that $\mathcal{L}$ always has at most the same value as the desired log probability, and is equal to it if and only if $q$ is the same distribution as $p(\boldsymbol{h} \mid \boldsymbol{v})$.

Surprisingly, $\mathcal{L}$ can be considerably easier to compute for some distributions $q$. Simple algebra shows that we can rearrange $\mathcal{L}$ into a much more convenient form:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \log p(\boldsymbol{v}; \boldsymbol{\theta}) - D \quad (q(\boldsymbol{h}) \| p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta}))$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \qquad \frac{\log q(\boldsymbol{h})}{\log p(\boldsymbol{h} \mid \boldsymbol{v})}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \qquad \frac{\log q(\boldsymbol{h})}{\log \text{——}}$$

$$= \log p(\boldsymbol{v}; \boldsymbol{\theta}) - \qquad [\log q(\boldsymbol{h}) - \log p(\boldsymbol{h}, \boldsymbol{v}) + \log p(\boldsymbol{v})]$$

$$= \; - \qquad [\log q(\boldsymbol{h}) - \log p(\boldsymbol{h}, \boldsymbol{v})] \, .$$

This yields the more canonical definition of the evidence lower bound,

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\theta}, \boldsymbol{q}) = \qquad [\log p(\boldsymbol{h}, \boldsymbol{v})] + H(q). \tag{19.1}$$

The first term of $\mathcal{L}$ is known as the *energy term*. The second term is known as the *entropy term*. For an appropriate choice of $q$, both terms can be easy to compute. The only question is how close to $p(\boldsymbol{h} \mid \boldsymbol{v})$ the distribution $q$ will be. This determines how good of an approximation $\mathcal{L}$ will be for $\log p(\boldsymbol{v})$.

We can thus think of inference as the procedure for finding the $q$ that maximizes $\mathcal{L}$. Exact inference maximizes $\mathcal{L}$ perfectly. Throughout this chapter, we will show how many forms of approximate inference are possible. No matter what choice of $q$ we use, $\mathcal{L}$ will give us a lower bound on the likelihood. We can get tighter or looser bounds that are cheaper or more expensive to compute depending on how we choose to approach this optimization problem. We can obtain a poorly matched $q$ but reduce the computational cost by using an imperfect optimization procedure, or by using a perfect optimization procedure over a restricted family of $q$ distributions.

## 19.2 Expectation Maximization

*Expectation maximization* (EM) is a popular training algorithm for models with latent variables. It consists of alternating between two steps until convergence:

- The *E-step* (Expectation step): Set $q(\boldsymbol{h} \quad) = p(\boldsymbol{h} \quad \mid \boldsymbol{v} \; i); \boldsymbol{\theta})$ for all indices $i$ of the training examples $\boldsymbol{v} \quad$ we want to train on (both batch and minibatch variants are valid). By this we mean $q$ is defined in terms of the *current* value of $\boldsymbol{\theta}$; if we vary $\theta$ then $p(\boldsymbol{h} \mid \boldsymbol{v}; \boldsymbol{\theta})$ will change but $q(\boldsymbol{h})$ will not.

- The M-step (Maximization step): Completely or partially maximize $\quad \mathcal{L}(\boldsymbol{v} \; i), \boldsymbol{\theta}, q)$ with respect to $\boldsymbol{\theta}$ using your optimization algorithm of choice.

This can be viewed as a coordinate ascent algorithm to maximize $\mathcal{L}$. On one step, we maximize $\mathcal{L}$ with respect to $q$, and on the other, we maximize $\mathcal{L}$ with respect to $\boldsymbol{\theta}$.

Stochastic gradient ascent on latent variable models can be seen as a special case of the EM algorithm where the M step consists of taking a single gradient step. Other variants of the EM algorithm can make much larger steps. For some model families, the M step can even be performed analytically, jumping all the way to the optimal solution given the current $q$.

Even though the E-step involves exact inference, we can think of the EM algorithm as using approximate inference in some sense. Specifically, the M-step assumes that the same value of $q$ can be used for all values of $\boldsymbol{\theta}$. This will introduce a gap between $\mathcal{L}$ and the true $\log p(v)$ as the M-step moves further and further. Fortunately, the E-step reduces the gap to zero again as we enter the loop for the next time.

The EM algorithm is a workhorse of classical machine learning, and it can be considered to be used in deep learning in the sense that stochastic gradient ascent can be seen as EM with a very simple and small M step. However, because $\mathcal{L}$ can not be analytically maximized for many interesting deep models, the more general EM framework as a whole is typically not explored in the deep learning research community.

TODO–cite the emview paper

## 19.3 MAP Inference: Sparse Coding as a Probabilistic Model

TODO synch up with other sections on sparse coding

Many versions of sparse coding can be cast as probabilistic models. For example, suppose we encode visible data $\boldsymbol{v} \in$ with latent variables $\boldsymbol{h} \in$ . We can use a prior to encourage our latent code variables to be sparse:

$$p(h) = TODO.$$

We can define the visible units to be Gaussian with an affine transformation from the code to the mean of the Gaussian:

$$\boldsymbol{v} \sim \mathcal{N}(\boldsymbol{v} \mid \boldsymbol{\mu} + \boldsymbol{W}\boldsymbol{h}, \boldsymbol{\beta} \quad )$$

where $\boldsymbol{\beta}$ is a diagonal precision matrix to maintain tractability.

Computing $p(\boldsymbol{h} \mid \boldsymbol{v})$ is difficult. TODO explain why

One operation that we can do is perform *maximum a posteriori* (MAP) inference, which means solving the following optimization problem:

$$\boldsymbol{h} \quad = \arg\max p(\boldsymbol{h} \mid \boldsymbol{v}).$$

This yields the familiar optimization problem

TODO synch with other sparse coding sections, make sure the other sections talk about using gradient descent, feature sign, ISTA, etc.

This shows that the popular feature extraction strategy for sparse coding can be justified as having a probabilistic interpretation–it may be MAP inference in this probabilistic model ( there are other probabilistic models that yield the same optimization problem, so we cannot positively identify this specific model from the feature extraction process ).

Excitingly, MAP inference of $\boldsymbol{h}$ given $\boldsymbol{v}$ also has an interpretation in terms of maximizing the evidence lower bound. Specifically, MAP inference maximizes $\mathcal{L}$ with respect to $q$ under the constraint that $q$ take the form form of a Dirac distribution. During learning of sparse coding, we alternate between using convex optimization to extract the codes, and using convex optimization to update $\boldsymbol{W}$ to achieve the optimal reconstruction given the codes. This turns out to be equivalent to maximizing $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ for the $q$ that was obtained from MAP inference. The learning algorithm can be thought of as EM restricted to using a Dirac posterior. In other words, rather than performing learning exactly using standard inference, we learn to maximize a bound on the true likelihood, using exact MAP inference.

## 19.4 Sequence Modeling with Graphical Models

This section regards probabilistic approaches to sequential data modeling which have often been viewed as in competition with RNNs, although RNNs can be seen as a particular form of *dynamic Bayesian networks*[1], as directed graphical models with deterministic latent variables[2].

### 19.4.1 Efficient Inference by Dynamic Programming

Many temporal modeling approaches can be cast in the following framework, which also includes hybrids of neural networks with HMMs and conditional random fields (CRFs), first introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c) and later developed and applied with great success in Graves *et al.* (2006); Graves (2012) with the Connectionist Temporal Classification (CTC) approach, as well as in Do and Artières (2010) and other more recent work (Farabet *et al.*, 2013b; Deng *et al.*, 2014). These ideas have been rediscovered in a simplified form (limiting

[1]Dynamic Bayesian networks or dynamic probabilistic networks are directed graphical models for sequential data, with shared parameters across time (Dean and Kanazawa, 1989; Kanazawa *et al.*, 1995)

[2]Latent variables are random variables that are not directly observed, although they can depend on some that are, and here the dependency is so strong that the latent variables are functions of observed variables

the input-output relationship to a linear one) as CRFs (Lafferty *et al.*, 2001), i.e., undirected graphical models whose parameters are linear functions of input variables. In section 19.5 we consider in more detail the neural network hybrids and the "graph transformer" generalizations of the ideas presented below.

All these approaches (with or without neural nets in the middle) concern the case where we have an input sequence (discrete or continuous-valued) $\{\boldsymbol{x}\}$ and a symbolic output sequence $\{y\}$ (typically of the same length, although shorter output sequences can be handled by introducing "empty string" values in the output). Generalizations to non-sequential output structure have been introduced more recently (e.g. to condition the Markov Random Fields sometimes used to model structural dependencies in images (Stewart *et al.*, 2007)), at the loss of exact inference (the dynamic programming methods described below).

Optionally, one also considers a latent variable sequence $\{s\}$ that is also discrete and inference needs to be done over $\{s\}$, either via marginalization (summing over all possible values of the state sequence) or maximization, i.e., picking exactly or approximately the so-called MAP sequence, the one with the largest probability, given the input. If the state variables $s$ and the target variables $y$ have a 1-D Markov structure to their dependency, then computing likelihood, partition function and MAP values can all be done efficiently by exploiting dynamic programming to factorize the computation. On the other hand, if the state or output sequence dependencies are captured by an RNN, then there is no finite-order Markov property and no efficient and exact inference is generally possible. However, many reasonable approximations have been used in the past, such as with variants of the beam search algorithm (Lowerre, 1976). The idea of beam search is that one maintains a set of promising candidate paths that end at some time step $t$ For each additional time step, one considers extensions to $t+1$ of each of these paths and then prunes those with the worse overall cumulative score (up to $t+1$). The beam size is the number of candidates that are kept. See Section 19.5.1 for more details on beam search.

The application of the principle of dynamic programming in these setups is the same as what is used in the Forward-Backward algorithm (detailed more around Eq. 19.5), for graphical models and HMMs (detailed more in Section 19.4.2) and the Viterbi algorithm detailed below (Eq. 19.7). For both of these algorithms, we are trying to sum (Forward-Backward algorithm) or maximize (Viterbi algorithm) over paths the probability or score associated with each path.

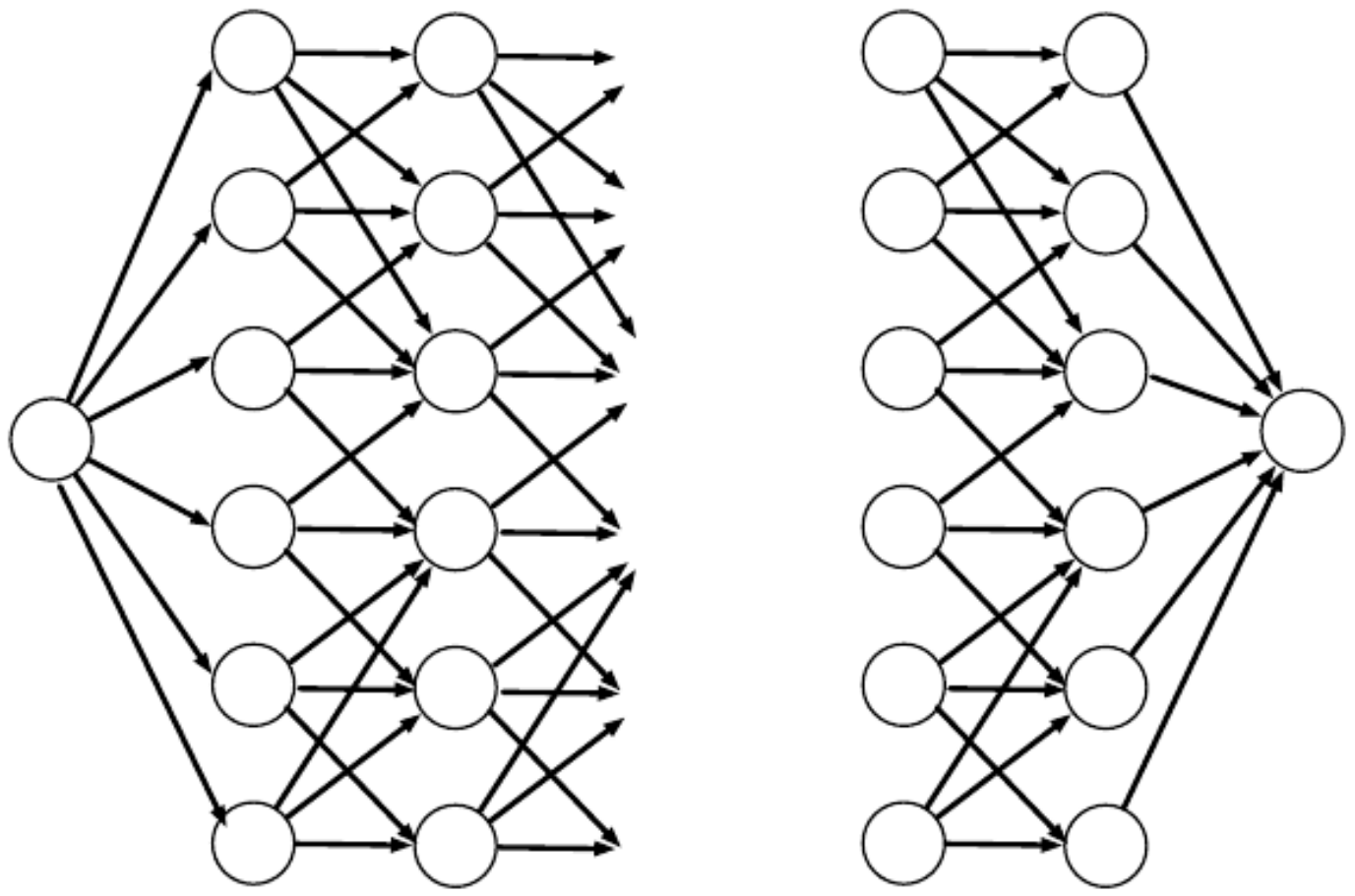

Figure 19.2: Example of a temporally structured output graph, as can be found in CRFs, HMMs and neural net hybrids. Each node corresponds to a particular **value** of an output random variable at a particular point in the output sequence (contrast with a graphical model representation, where each node corresponds to a random variable). A path from the source node to the sink node (e.g. red bold arrows) corresponds to an interpretation of the input as a sequence of output labels. The dynamic programming recursions that are used for computing likelihood (or conditional likelihood) or performing MAP inference (finding the best path) involve sums or maximizations over sub-paths ending at one of the particular interior nodes.

Let $\mathcal{G}$ be a directed acyclic graph whose paths correspond to the sequences that can be selected (for MAP) or summed over (marginalized for computing a likelihood), as illustrated in Fig. 19.2. In the above example, let $z$ represent the choice variable (e.g., $s$ and $y$ in the above example), and each arc with score $a$ corresponds to a particular value of $z$ in its Markov context. In the language of undirected graphical models, if $a$ is the score associated with an arc from the node for $z = j$ to the one for $z = i$, then $a$ is minus the energy of a term of the energy function associated with the event 1 and the associated information from the input $\boldsymbol{x}$ (e.g. some value of $x$ ).

Hidden Markov models are based on the notion of Markov chain, which is covered in much more detail in Section 17.1. A Markov chain is a sequence of random variables z , . . . z . For our purposes the main property of a Markov chain chain of *order 1* is that the current value of z contains enough information about the previous values $z$ , . . . $z$ in order to predict the distribution of the next random variable, $z$ . In our context, we can make the $z$'s conditioned on

some $\boldsymbol{x}$. The order 1 Markov property then means that

$$P(z \mid z \quad , z \quad , \ldots , z \ , \boldsymbol{x}) = P(z \mid z \quad , \boldsymbol{x}),$$

where $\boldsymbol{x}$ is the conditioning information (the input sequence). When we consider a path in that space, i.e. a sequence of values, we draw a graph with a node for each discrete value of $z$, and if it is possible to transition from $z \quad = j$ to $z = i$ we draw an arc between these two nodes. Hence, the total number of nodes in the graph would be equal to $Tn$ (the length of the sequence times the number of values of $z$). The number of arcs in the graph could be up to $Tn$ . This extreme case occurs if every value of $z$ can follow every value of $z \quad$ . In practice the connectivity is often much smaller because not all transitions are typically feasible. A score $a$ is computed for each arc (which may include some component that only depends on the source or only on the destination node), as a function of the conditioning information $\boldsymbol{x}$. The inference or marginalization problems involve performing the following computations.

For the **marginalization** task, we want to compute the sum over all complete paths (e.g. from source to sink) of the product along the path of the exponentiated scores associated with the arcs on that path:

$$m(\mathcal{G}) = \qquad e \tag{19.2}$$

where the product is over all the arcs on a path (with score $a$), and the sum is over all the paths associated with complete sequences (from beginning to end of a sequence). $m(\mathcal{G})$ may correspond to a likelihood, numerator or denominator of a probability. For example,

$$P(\{z\} \in \quad \mid \boldsymbol{x}) = \frac{m(\mathcal{G} \quad )}{m(\mathcal{G})} \tag{19.3}$$

where $\mathcal{G} \quad$ is the subgraph of $\mathcal{G}$ which is restricted to sequences that are compatible with some target answer .

For the **inference** task, we want to compute

$$\pi(\mathcal{G}) \quad = \quad \arg\max \qquad e \quad = \arg\max \qquad a$$

$$v(\mathcal{G}) \quad = \quad \max \qquad a$$

where $\pi(\mathcal{G})$ is the most probable path and $v(\mathcal{G})$ is its log-score or value, and again the set of paths considered includes all of those starting at the beginning and ending at the end the sequence.

The principle of dynamic programming is to recursively compute intermediate quantities that can be reused efficiently so as to avoid actually going through an exponential number of computations, e.g., though the exponential number of paths to consider in the above sums or maxima. Note how it is already at play in the underlying efficiency of back-propagation (or back-propagation through time), where gradients w.r.t. intermediate layers or time steps or nodes in a flow graph can be computed based on previously computed gradients (for later layers, time steps or nodes). Here it can be achieved by considering to restrictions of the graph to those paths that end at a node $n$, which we denote $\mathcal{G}$ . $\mathcal{G}$ indicates the additional restriction to subsequences that are compatible with the target sequence $Y$, i.e., with the beginning of the sequence $Y$.

We can thus perform marginalization efficiently as follows, using a generalization of the Forward-Backward algorithm for HMMs. This is a generalization of the so-called Forward-Backward algorithm for HMMs.

$$m(\mathcal{G}) = \qquad m(\mathcal{G} \quad ) \tag{19.4}$$

where final($\mathcal{G}$) is the set of final nodes in the graph $\mathcal{G}$. We can recursively compute the node-restricted sum via the identity

$$m(G\ ) = \qquad m(\mathcal{G} \quad )e \tag{19.5}$$

where pred($n$) is the set of predecessors of node $n$ in the graph and $a$ is the log-score associated with the arc from $m$ to $n$. It is easy to see that expanding the above recursion recovers the result of Eq. 19.2.

Similarly, we can perform efficient MAP inference (also known as Viterbi decoding) as follows.

$$v(\mathcal{G}) = \quad \max \quad v(G \quad ) \tag{19.6}$$

and

$$v(\mathcal{G} \quad ) = \quad \max \quad v(\mathcal{G} \quad ) + a \quad . \tag{19.7}$$

To obtain the corresponding path, it is enough to keep track of the argmax associated with each of the above maximizations and trace back $\pi(\mathcal{G})$ starting from the nodes in final($\mathcal{G}$). For example, the last element of $\pi(\mathcal{G})$ is

$$n \quad \leftarrow \arg\max v(\mathcal{G} \quad )$$

and (recursively) the argmax node before $n$ along the selected path is a new $n$ ,

$$n \leftarrow \arg\max v(\mathcal{G}) + a \quad ,$$

etc. Keeping track of these $n$ along the way gives the selected path. Proving that these recursive computations yield the desired results is straightforward and left as an exercise.

### 19.4.2 HMMs

Hidden Markov Models (HMMs) are probabilistic models of sequences that were introduced in the 60's (Baum and Petrie, 1966) along with the E-M algorithm (Section 19.2). They are very commonly used to model sequential structure, in particular having been since the mid 80's and until recently the technological core of speech recognition systems (Rabiner and Juang, 1986; Rabiner, 1989). Just like RNNs, HMMs are dynamic Bayes nets (Koller and Friedman, 2009), i.e., the same parameters and graphical model structure are used for every time step. Compared to RNNs, what is particular to HMMs is that the latent variable associated with each time step (called the *state*) is discrete, with a separate set of parameters associated with each state value. We consider here the most common form of HMM, in which the Markov chain is of order 1, i.e., the state $s$ at time $t$, given the previous states, only depends on the previous state $s$ :

$$P(s \mid s, s, \ldots, s) = P(s \mid s),$$

which we call the *transition or state-to-state distribution*. Generalizing to higher-order Markov chains is straightforward: for example, order-2 Markov chains can be mapped to order-1 Markov chains by considering as order-1 "states" all the pairs $(s = i, s = j)$.

Given the state value, a generative probabilistic model of the visible variable $\boldsymbol{x}$ is defined, that specifies how each observation $\boldsymbol{x}$ in a sequence $(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x})$ can be generated, via a model $P(\boldsymbol{x} \mid s)$. Two kinds of parameters are distinguished: those that define the transition distribution, which can be given by a matrix

$$A = P(s = i \mid s = j),$$

and those that define the output model $P(\boldsymbol{x} \mid s)$. For example, if the data are discrete and $\boldsymbol{x}$ is a symbol $x$ , then another matrix can be used to define the output (or emission) model:

$$B = P(x = k \mid s = i).$$

Another common parametrization for $P(\boldsymbol{x} \mid s = i)$, in the case of continuous vector-valued $\boldsymbol{x}$, is the Gaussian mixture model, where we have a different mixture (with its own means, covariances and component probabilities) for each state $s = i$. Alternatively, the means and covariances (or just variances) can be shared across states, and only the component probabilities are state-specific.

The overall likelihood of an observed sequence is thus

$$P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x}) = \qquad P(\boldsymbol{x} \mid s)P(s \mid s \quad). \tag{19.8}$$

In the language established earlier in Section 19.4.1, we have a graph $\mathcal{G}$ with one node $n$ per time step $t$ and state value $i$, i.e., for $s = i$, and one arc between each node $n$ (for ) and its predecessors $m$ for (when the transition probability is non-zero, i.e., $P(s = i \mid s = j) \neq 0$). Following Eq. 19.8, the log-score $a$ for the transition between $m$ and $n$ would then be

$$a = \log P(x \mid s = i) + \log P(s = i \mid s = j).$$

As explained in Section 19.4.1, this view gives us a dynamic programming algorithm for computing the likelihood (or the conditional likelihood given some constraints on the set of allowed paths), called the forward-backward or sum-product algorithm, in time $O(kNT)$ where $T$ is the sequence length, $N$ is the number of states and $k$ the average in-degree of each node.

Although the likelihood is tractable and could be maximized by a gradient-based optimization method, HMMs are typically trained by the E-M algorithm (Section 19.2), which has been shown to converge rapidly (approaching the rate of Newton-like methods) in some conditions (if we view the HMM as a big mixture, then the condition is for the final mixture components to be well-separated, i.e., have little overlap) (Xu and Jordan, 1996).

At test time, the sequence of states that maximizes the joint likelihood

$$P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x}, s, s, \ldots, s)$$

can also be obtained using a dynamic programming algorithm (called the Viterbi algorithm). This is a form of *inference* (see Section 16.5) that is called MAP (Maximum A Posteriori) inference because we want to find the most probable value of the unobserved state variables given the observed inputs. Using the same definitions as above (from Section 19.4.1) for the nodes and log-score of the graph $\mathcal{G}$ in which we search for the optimal path, the Viterbi algorithm corresponds to the recursion defined by Eq. 19.7.

If the HMM is structured in such a way that states have a meaning associated with labels of interest, then from the MAP sequence one can read off the associated

labels. When the number of states is very large (which happens for example with large-vocabulary speech recognition based on $n$-gram language models), even the efficient Viterbi algorithm becomes too expensive. In such cases only approximate search is feasible. A common family of search algorithms for HMMs is the *beam search* algorithm (Lowerre, 1976) (Section 19.5.1).

More details about speech recognition are given in Section 12.3. An HMM can be used to associate a sequence of labels $(y, y, \ldots, y)$ with the input $(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x})$, where the output sequence is typically shorter than the input sequence, i.e., $N < T$. Knowledge of $(y, y, \ldots, y)$ constrains the set of compatible state sequences $(s, s, \ldots, s)$, and the generative conditional likelihood

$$P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x} \mid y, y, \ldots, y) = P(\boldsymbol{x} \mid s) P(s \mid s). \tag{19.9}$$

can be computed using the same forward-backward technique. This enables us to maximize its logarithm during training, as discussed above.

Various discriminative alternatives to the generative likelihood of Eq. 19.9 have been proposed (Brown, 1987; Bahl *et al.*, 1987; Nadas *et al.*, 1988; Juang and Katagiri, 1992; Bengio *et al.*, 1992a; Bengio, 1993; Leprieur and Haffner, 1995; Bengio, 1999a), the simplest of which is simply $P(y, y, \ldots, y \mid \boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x})$, which is obtained from Eq. 19.9 by Bayes rule, i.e., involving a normalization over all sequences, i.e., the unconstrained likelihood of Eq. 19.8:

$$P(y, y, \ldots, y \mid \boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x}) = \frac{P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x} \mid y, y, \ldots, y) P(y, y, \ldots, y)}{P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x})}.$$

Both the numerator and denominator can be formulated in the framework of the previous section (Eqs. 19.3-19.5), where for the numerator we merge (add) the log-scores coming from the structured output output model $P(y, y, \ldots, y)$ and from the input likelihood model $P(\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x} \mid y, y, \ldots, y)$. Again, each node of the graph corresponds to a state of the HMM at a particular time step $t$ (which may or may not emit the next output symbol $y$), associated with an input vector $\boldsymbol{x}$. Instead of making the relationship to the input the result of a simple parametric form (Gaussian or multinomial, typically), the scores can be computed by a neural network (or any other parametrized differential function). This gives rise to discriminative hybrids of search or graphical models with neural networks, discussed below, Section 19.5.

### 19.4.3 CRFs

Whereas HMMs are typically trained to maximize the probability of an input sequence given a target sequence and correspond to a directed graphical model,

Conditional Random Fields (CRFs) (Lafferty *et al.*, 2001) are *undirected* graphical models that are trained to maximize the joint probability of the target variables, given input variables, $P(\quad | \quad)$. CRFs are special cases of the graph transformer model introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c), where neural nets are replaced by affine transformations and there is a single graph involved. A *graph transformer* is an computational module that transforms a weighted (with a scalar on each arc) directed acyclic graph into another one. Examples of graph transformers are illustrated in Fig. 19.3. In this context, graph transformers can be seen as transforming the set of weights in their input graph into a set of weights on their output graph, typically so that we can compute derivatives of the output graph weights with respect to input graph weights, i.e., we can back-propagated costs through the graph transformer. Section 19.5 below provides more discussion and examples.

Many applications of CRFs involve sequences and the discussion here will be focused on this type of application, although applications to images (e.g. for image segmentation) are also common. Compared to other graphical models, another characteristic of CRFs is that there are no latent variables. The general equation for the probability distribution modeled by a CRF is basically the same as for fully visible (not latent variable) undirected graphical models, also known as Markov Random Fields (MRFs, see Section 16.2.2), *except* that the "potentials" (terms of the energy function) are parametrized functions of the input variables, and the likelihood of interest is the posterior probability $P(\quad | \quad)$.

As in many other MRFs, CRFs often have a particular connectivity structure in their graph, which allows one to perform learning or inference more efficiently. In particular, when dealing with sequences, the energy function typically only has terms that relate neighboring elements of the sequence of target variables. For example, the target variables could form a homogeneous[3] Markov chain of order $k$ (given the input variables). A typical linear CRF example with binary outputs would have the following structure:

$$P(\quad = \boldsymbol{y} \mid \quad) = \frac{1}{Z}\exp \qquad y \ (b + \qquad w\, x \ ) + \qquad y\, y \quad (u \ + \qquad v \ \ x \ ) \tag{19.10}$$

where $Z$ is the normalization constant, which is the sum over all $\boldsymbol{y}$ sequences of the numerator. In that case, the score marginalization framework of Section 19.4.1 and coming from Bottou *et al.* (1997); LeCun *et al.* (1998c) can be applied by making terms in the above exponential correspond to scores associated with nodes $t$ of a graph $\mathcal{G}$. If there were more than two output classes, more nodes per time step would be required but the principle would remain the same. A more general

[3] meaning that the same parameters are used for every time step

formulation for Markov chains of order $d$ is the following:

$$P(\quad = \boldsymbol{y} \mid \quad) = \frac{1}{Z} \exp \qquad f\ (y\ , y\quad , \ldots , y\quad , x\ ) \tag{19.11}$$

where $f$ computes a potential of the energy function, a parametrized function of both the past target values (up to $y$ ) and of the current input value $x$ . For example, as discussed below $f$ could be the output of an arbitrary parametrized computation, such as a neural network.

Although $Z$ looks intractable, because of the Markov property of the model (order 1, in the example), it is again possible to exploit dynamic programming to compute $Z$ efficiently, as per Eqs. 19.3-19.5). Again, the idea is to compute the sub-sum for sequences of length $t \leq T$ (where $T$ is the length of a target sequence $\boldsymbol{y}$), ending in each of the possible state values at $t$, e.g., $y = 1$ and $y = 0$ in the above example. For higher order Markov chains (say order $d$ instead of 1) and a larger number of state values (say $N$ instead of 2), the required sub-sums to keep track of are for each element in the cross-product of $d-1$ state values, i.e., $N$ . For each of these elements, the new sub-sums for sequences of length $t+1$ (for each of the $N$ values at $t+1$ and corresponding $N$ past values for the past $d-2$ time steps) can be obtained by only considering the sub-sums for the $N$ joint state values for the last $d-1$ time steps before $t+1$.

Following Eq. 19.7, the same kind of decomposition can be performed to efficiently find the MAP configuration of $y$'s given $\boldsymbol{x}$, where instead of products (sums inside the exponential) and sums (for the outer sum of these exponentials, over different paths) we respectively have sums (corresponding to adding the sums inside the exponential) and maxima (across the different competing "previous-state" choices).

## 19.5 Combining Neural Networks and Search

The idea of combining neural networks with HMMs or related search or alignment-based components (such as graph transformers) for speech and handwriting recognition dates from the early days of research on multi-layer neural networks (Bourlard and Wellekens, 1990; Bottou *et al.*, 1990; Bengio, 1991; Bottou, 1991; Haffner *et al.*, 1991; Bengio *et al.*, 1992a; Matan *et al.*, 1992; Bourlard and Morgan, 1993; Bengio *et al.*, 1995; Bengio and Frasconi, 1996; Baldi and Brunak, 1998) – and see more references in Bengio (1999b). See also 12.5 for combining recurrent and other deep learners with generative models such as CRFs, GSNs or RBMs.

The principle of efficient marginalization and inference for temporally structured outputs by exploiting dynamic programming (Sec. 19.4.1) can be applied

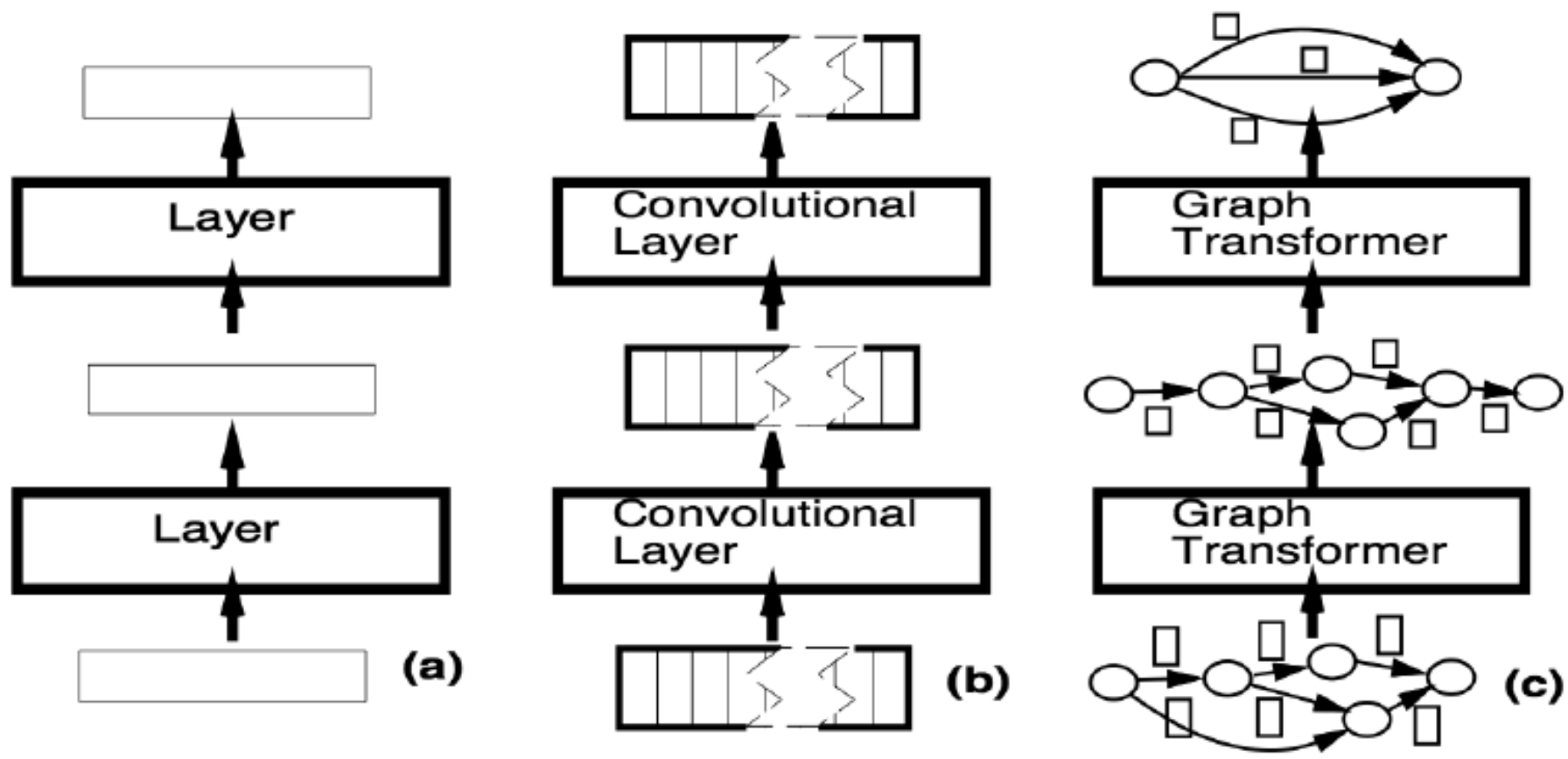


Figure 19.3: Illustration of the stacking of graph transformers (right, c) as a generalization of the stacking of convolutional layers (middle, b) or of regular feedforward layers that transform fixed-size vectors (left, a). Figure reproduced with permission from the authors of Bottou *et al.* (1997). Quoting from that paper, (c) shows that "multilayer graph transformer networks are composed of trainable modules that operate on and produce graphs whose arcs carry numerical information".

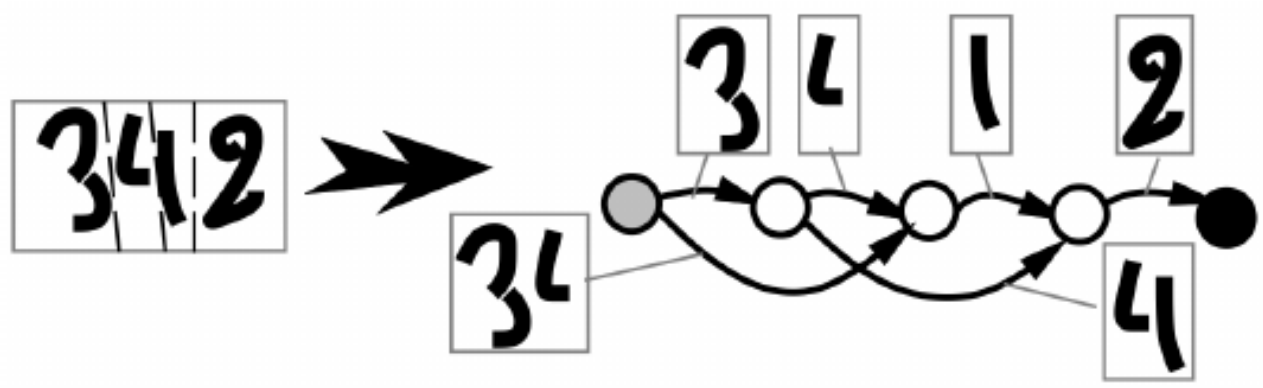

Figure 19.4: Illustration of the input and output of a simple graph transformer that maps a singleton graph corresponding to an input image to a graph representing hypothesized segmentation hypotheses. Reproduced with permission from the authors of Bottou *et al.* (1997).

not just when the log-scores of Eqs. 19.2 and 19.4 are parameters or linear functions of the input. This principle can also applied when the log-scores are *learned non-linear functions* of the input, including functions represented by a neural network. Bottou *et al.* (1997) and LeCun *et al.* (1998c) introduced this idea and the powerful idea of *learned graph transformers*, illustrated in Fig. 19.3. In this context, a graph transformer is a machine that can *map a directed acyclic graph* $\mathcal{G}$ *to another graph* $\mathcal{G}$ . Both input and output graphs have paths that represent hypotheses about the observed data.

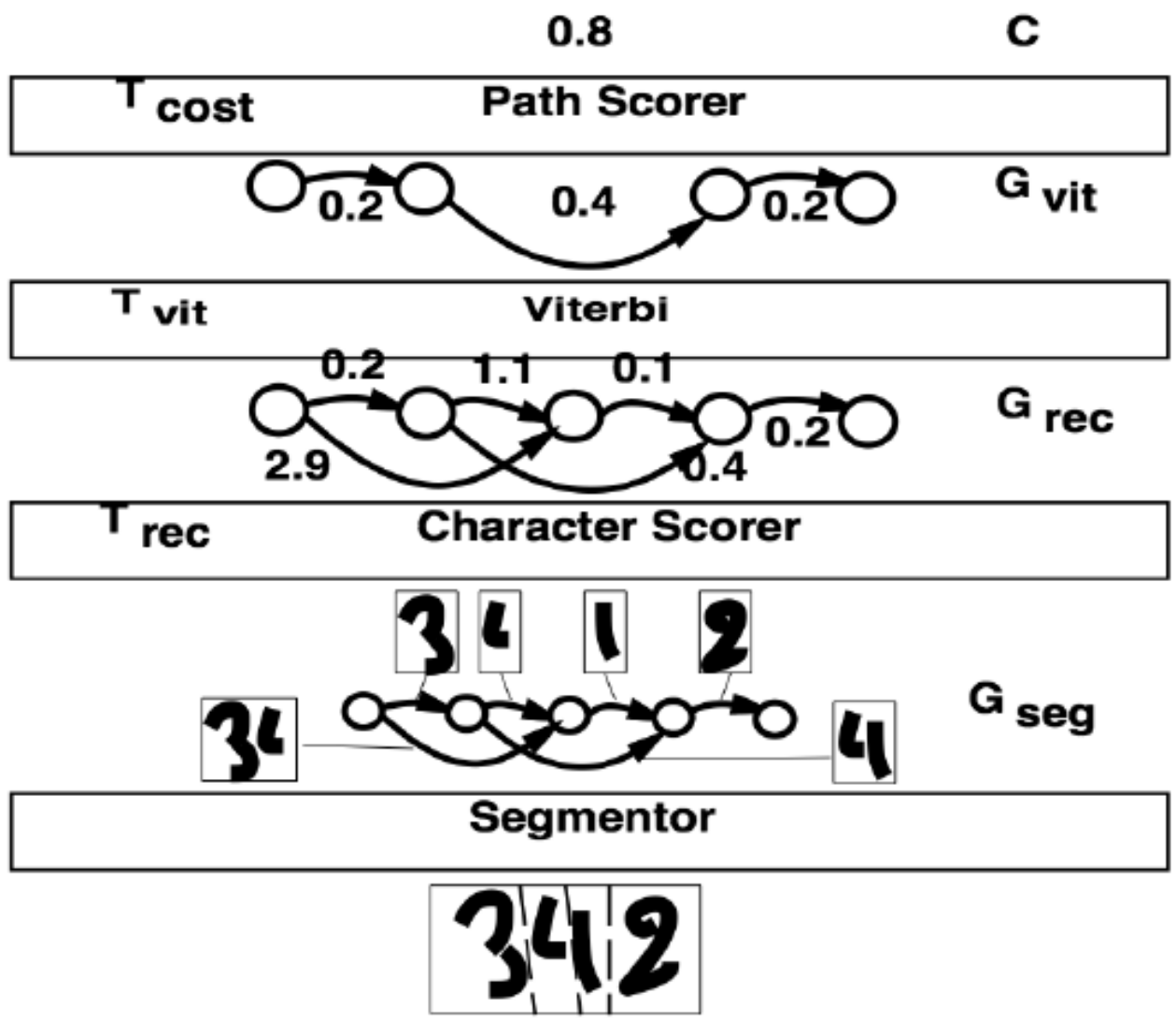


Figure 19.5: Illustration of the graph transformer network that has been used for finding the best segmentation of a handwritten word, for handwriting recognition. Reproduced with permission from Bottou *et al.* (1997).

For example, a segmentation graph transformer takes a singleton input graph (the image $\boldsymbol{x}$) and outputs a graph representing segmentation hypotheses (regarding sequences of segments that could each contain a character in the image). This process is illustrated in Fig. 19.4. Such a graph transformer could be used as one layer of a *graph transformer network* for handwriting recognition or document analysis for reading amounts on checks, as illustrated respectively in Fig.s 19.5 and 19.6.

For example, after the segmentation graph transformer, a recognition graph transformer could expand each node of the segmentation graph into a subgraph whose arcs correspond to different interpretations of the segment (which character

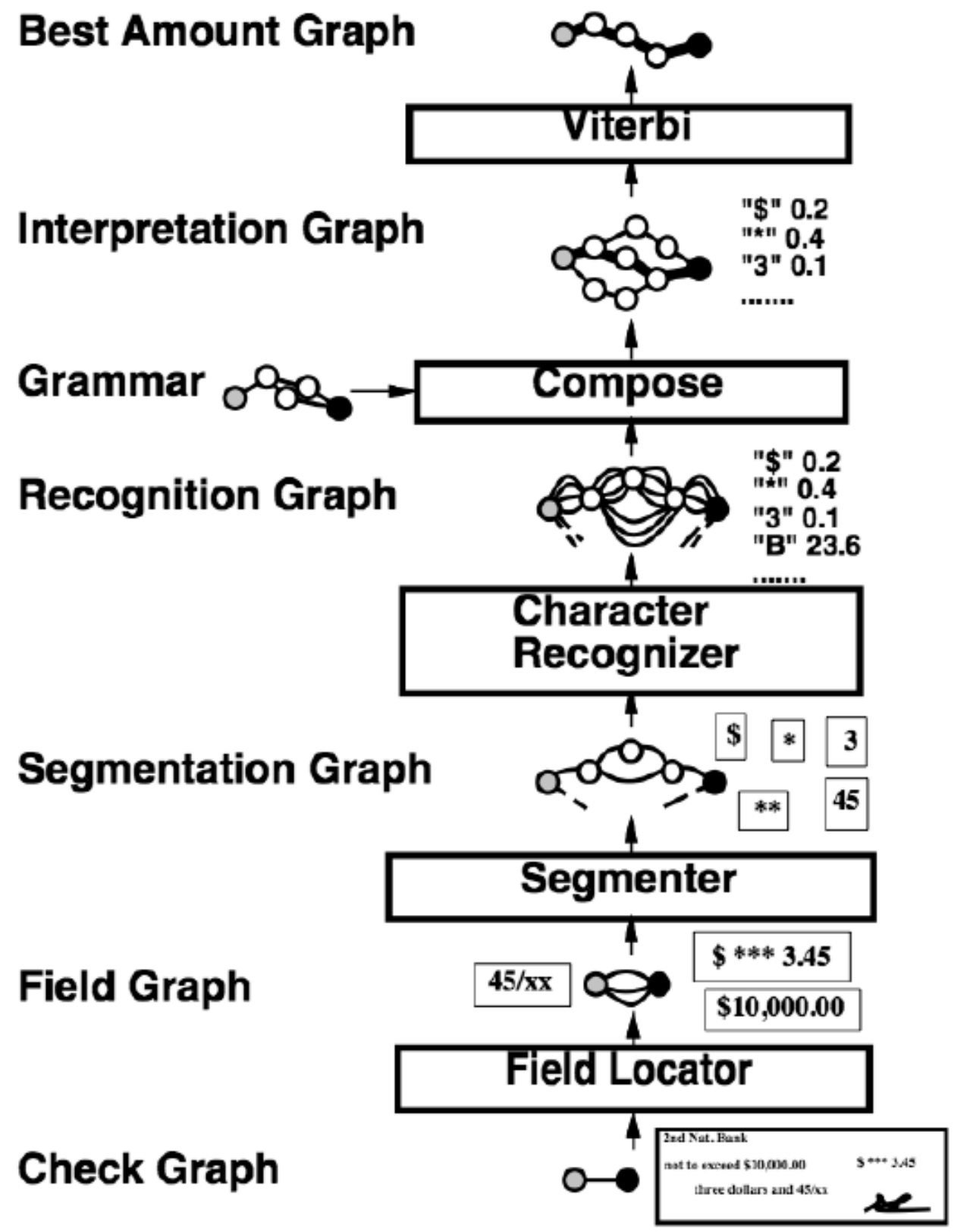


Figure 19.6: Illustration of the graph transformer network that has been used for reading amounts on checks, starting from the single graph containing the image of the graph to the recognized sequences of characters corresponding to the amount on the graph, with currency and other recognized marks. Note how the grammar graph transformer composes the grammar graph (allowed sequences of characters) and the recognition graph (with character hypotheses associated with specific input segments, on the arcs) into an interpretation graph that only contains the recognition graph paths that are compatible with the grammar. Reproduced with permission from Bottou *et al.* (1997).

is present in the segment?). Then, a dictionary graph transformer takes the recognition graph and expands it further by considering only the sequences of characters that are compatible with sequences of words in the language of interest. Finally, a language-model graph transformer expands sequences of word hypotheses so as to include multiple words in the state (context) and weigh the arcs according to the language model next-word log-probabilities.

Each of these transformations is parametrized and takes real-valued scores on the arcs of the input graph into real-valued scores on the arcs of the output graph. These transformations can be parametrized and learned by gradient-based optimization over the whole series of graph transformers.

### 19.5.1 Approximate Search

Unfortunately, as in the above example, when the number of nodes of the graph becomes very large (e.g., considering all previous $n$ words to condition the log-probability of the next one, for $n$ large), even dynamic programming (whose computation scales with the number of arcs) is too slow for practical applications such as speech recognition or machine translation. A common example is when a recurrent neural network is used to compute the arcs log-score, e.g., as in neural language models (Section 12.4). Since the prediction at step $t$ depends on all $t-1$ previous choices, the number of states (nodes of the search graph $\mathcal{G}$) grows exponentially with the length of the sequence. In that case, one has to resort to *approximate search*.

#### Beam Search

In the case of sequential structures as discussed in this chapter, a common family of approximate search algorithms is the *beam search* (Lowerre, 1976). To perform beam search with *beam width* $k$, we do the following:

- Break the nodes of the graph into $g$ groups containing only "comparable nodes", e.g., the group of nodes $n$ for which the maximum length of the paths ending at $n$ is exactly $t$.
- Process these groups of nodes sequentially, keeping only at each step $t$ a selected subset of the nodes (the "beam"), chosen based on the subset . Each node in is associated with a score $\hat{v}(\mathcal{G})$ that represents an approximation (a lower bound) on the maximum total log-score of the path ending at the node, $v(\mathcal{G})$ (defined in Eq. 19.7, Viterbi decoding).
- is obtained by following all the arcs from the nodes in , and sorting all the resulting group $t$ nodes $n$ according to their estimated (lower bound)

score

$$\hat{v}(\mathcal{G}\ \ ) = \qquad \max \qquad \hat{v}(\mathcal{G}\ \ ) + a \quad ,$$

while keeping track of the argmax in order to trace back the estimated best path. Only the $k$ nodes with the highest log-score are kept and stored in .

- The estimated best final node can be read off from $\max \quad \hat{v}(\mathcal{G}\ \ )$ and the estimated best path from the associated argmax choices made along the way, just like in the Viterbi algorithm.

One problem with beam search is that the beam often ends up lacking in diversity, making the approximation poor. For example, imagine that we have two "types" of solutions, but that each type has exponentially many variants (as a function of $t$), due, e.g., to small independent variations in ways in which the type can be expressed at each time step $t$. Then, even though the two types may have close best log-score up to time $t$, the beam could be dominated by the one that wins slightly, eliminating the other type from the search, although later time steps might reveal that the second type was actually the best one.

## 19.6 Variational Inference and Learning

One common difficulty in probabilistic modeling is that the posterior distribution $p(\boldsymbol{h} \mid \boldsymbol{v})$ is infeasible to compute for many models with hidden variables $\boldsymbol{h}$ and visible variables $\boldsymbol{v}$. Expectations with respect to this distribution may also be intractable.

Consider as an example the *binary sparse coding* model. In this model, the input $\boldsymbol{v} \in$ is formed by adding Gaussian noise to the sum of $m$ different components which can each be present or absent. Each component is switched on or off by the corresponding hidden unit in $\boldsymbol{h} \in \{0, 1\}$ :

$$p(h \ \ = 1) = \sigma(b\ )$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{v} \mid \boldsymbol{W}\boldsymbol{h}, \beta \quad )$$

where $\boldsymbol{b}$ is a learn-able set of biases, $\boldsymbol{W}$ is a learn-able weight matrix, and $\boldsymbol{\beta}$ is a learn-able, diagonal precision matrix.

Training this model with maximum likelihood requires taking the derivative with respect to the parameters. Consider the derivative with respect to one of the biases:

$$\frac{\partial}{\partial b} \log p(\boldsymbol{v})$$

$$
\begin{aligned}
&= \frac{-p(\boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{- \quad p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{- \quad p(\boldsymbol{h}) p(\boldsymbol{v} \mid \boldsymbol{h})}{p(\boldsymbol{v})} \\
&= \frac{p(\boldsymbol{v} \mid \boldsymbol{h}) - p(\boldsymbol{h})}{p(\boldsymbol{v})} \\
&= p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{-p(\boldsymbol{h})}{p(\boldsymbol{h})} \\
&= p(\boldsymbol{h} \mid \boldsymbol{v}) \frac{-p(\boldsymbol{h})}{p(\boldsymbol{h})} \\
&= \frac{\partial}{\partial b} \log p(\boldsymbol{h}).
\end{aligned}
$$

This requires computing expectations with respect to $p(\boldsymbol{h} \mid \boldsymbol{v})$. Unfortunately, $p(\boldsymbol{h} \mid \boldsymbol{v})$ is a complicated distribution. See Fig. 19.7 for the graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$ and $p(\boldsymbol{h} \mid \boldsymbol{v})$. The posterior distribution corresponds to the complete graph over the hidden units, so variable elimination algorithms do not help us to compute the required expectations any faster than brute force.

One solution to this problem is to use *variational methods*. Variational methods involve using a simple distribution $q(\boldsymbol{h})$ to approximate the true, complicated posterior $p(\boldsymbol{h} \mid \boldsymbol{v})$. The name "variational" derives from their frequent use of a branch of mathematics called *calculus of variations*. However, not all variational methods use calculus of variations.

TODO variational inference involves maximization of a BOUND TODO variational inference also usually involves a restriction on the function family TODO

### 19.6.1 Discrete Latent Variables

TODO– BSC example

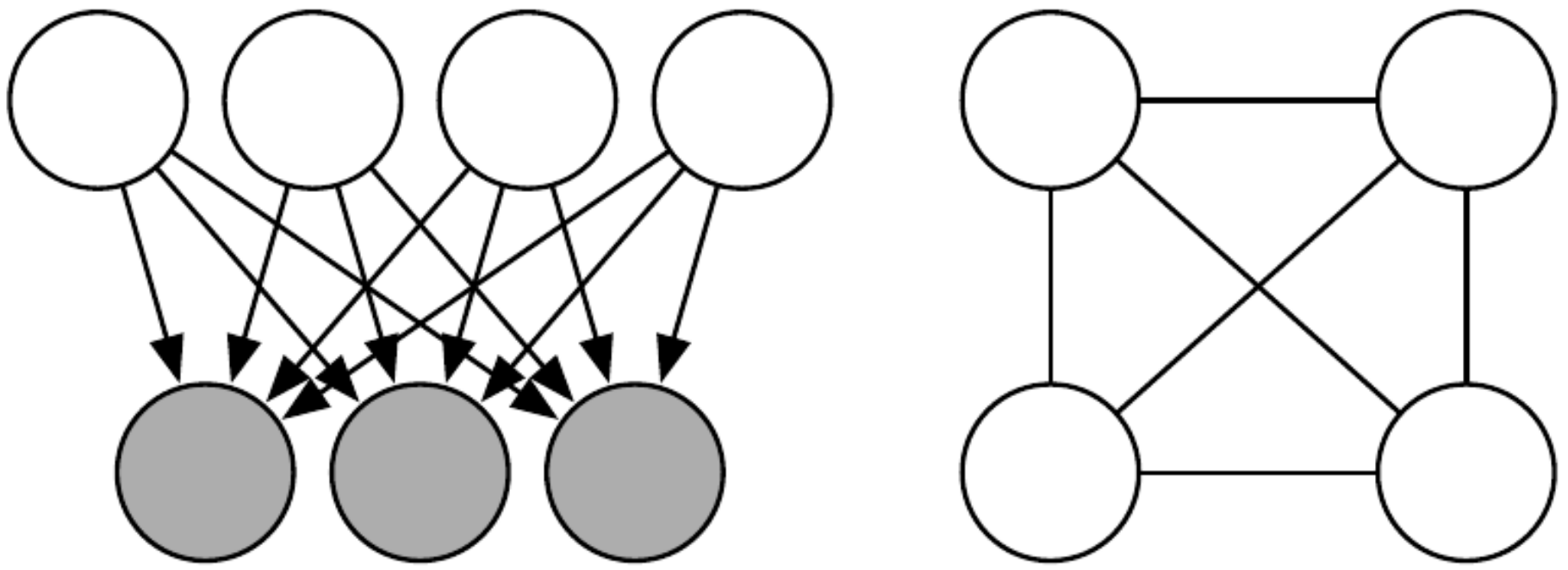

Figure 19.7: The graph structure of a binary sparse coding model with four hidden units. Left) The graph structure of $p(\boldsymbol{h}, \boldsymbol{v})$. Note that the edges are directed, and that every two hidden units co-parents of every visible unit. Right) The graph structure of $p(\boldsymbol{h} \mid \boldsymbol{v})$. In order to account for the active paths between co-parents, the posterior distribution needs an edge between all of the hidden units.

### 19.6.2 Calculus of Variations

Many machine learning techniques are based on minimizing a function $J(\boldsymbol{\theta})$ by finding the input vector $\boldsymbol{\theta} \in$ for which it takes on its minimal value. This can be accomplished with multivariate calculus and linear algebra, by solving for the critical points where $\nabla \ J(\boldsymbol{\theta}) = 0$. In some cases, we actually want to solve for a function $f(\boldsymbol{x})$, such as when we want to find the probability density function over some random variable. This is what *calculus of variations* enables us to do.

A function of a function $f$ is known as a *functional* $J[f]$. Much as we can take partial derivatives of a function with respect to elements of its vector-valued argument, we can take *functional derivatives*, also known as *variational derivatives* of a functional $J[f]$ with respect to respect to individual values of the function $f(\boldsymbol{x})$. The functional derivative of the functional $J$ with respect to the value of the function $f$ at point $\boldsymbol{x}$ is denoted $\text{——} J$.

A complete formal development of functional derivatives is beyond the scope of this book. For our purposes, it is sufficient to state that for differentiable functions $f(\boldsymbol{x})$ and differentiable functions $g(y, \boldsymbol{x})$ with continuous derivatives, that

$$\frac{\delta}{\delta f(\boldsymbol{x})} \quad g\left(f(\boldsymbol{x}), \boldsymbol{x}\right) d\boldsymbol{x} = \frac{\partial}{\partial y} g(f(\boldsymbol{x}), \boldsymbol{x}). \tag{19.12}$$

To gain some intuition for this identity, one can think of $f(\boldsymbol{x})$ as being a vector with uncountably many elements, indexed by a real vector $\boldsymbol{x}$. In this (somewhat incomplete view), the identity providing the functional derivatives is the same as

we would obtain for a vector $\boldsymbol{\theta} \in$ indexed by positive integers:

$$\frac{\partial}{\partial\theta} \quad g(\theta\ , j) = \frac{\partial}{\partial\theta} g(\theta\ , i).$$

Many results in other machine learning publications are presented using the more general *Euler-Lagrange equation* which allows $g$ to depend on the derivatives of $f$ as well as the value of $f$, but we do not need this fully general form for the results presented in this book.

To optimize a function with respect to a vector, we take the gradient of the function with respect to the vector and solve for the point where every element of the gradient is equal to zero. Likewise, we can optimize a functional by solving for the function where the functional derivative at every point is equal to zero.

As an example of how this process works, consider the problem of finding the probability distribution function over $x \in$ that has maximal Shannon entropy. Recall that the entropy of a probability distribution $p(x)$ is defined as

$$H[p] = - \quad \log p(x).$$

For continuous values, the expectation is an integral:

$$H[p] = - \quad p(x) \log p(x) dx.$$

We cannot simply maximize $H(x)$ with respect to the function $p(x)$, because the result might not be a probability distribution. Instead, we need to use Lagrange multipliers, to add a constraint that $p(x)$ integrate to 1. Also, the entropy increases without bound as the variance increases, so we can only search for the distribution with maximal entropy for fixed variance $\sigma$ . Finally, the problem is underdetermined because the distribution can be shifted arbitrarily without changing the entropy. To impose a unique solution, we add a constraint that the mean of the distribution be $\mu$. The Lagrangian functional for this optimization problem is

$$\mathcal{L}[p] = \lambda \quad p(x)dx - 1 \ + \lambda \ ( \ [x] - \mu) + \lambda \quad [(x - \mu) \ ] - \sigma \quad + H[p]$$

$$= \quad \lambda \ p(x) + \lambda \ p(x)x + \lambda \ p(x)(x - \mu) \ - p(x) \log p(x) \ dx - \lambda \ - \mu\lambda \ - \sigma \ \lambda \ .$$

To minimize the Lagrangian with respect to $p$, we set the functional derivatives equal to 0:

$$\forall x, \frac{\delta}{\delta p(x)} \mathcal{L} = \lambda \ + \lambda \ x + \lambda \ (x - \mu) \ - 1 - \log p(x) = 0.$$

This condition now tells us the functional form of $p(x)$. By algebraically rearranging the equation, we obtain

$$p(x) = \exp\ -\lambda\ - \lambda\ x + \lambda\ (x - \mu)\ + 1\ \ .$$

We never assumed directly that $p(x)$ would take this functional form; we obtained the expression itself by analytically minimizing a functional. To finish the minimization problem, we must choose the $\lambda$ values to ensure that all of our constraints are satisfied. We are free to choose any $\lambda$ values, because the gradient of the Lagrangian with respect to the $\lambda$ variables is zero so long as the constraints are satisfied. To satisfy all of the constraints, we may set $\lambda\ = \log \sigma\sqrt{2\pi}$, $\lambda\ = 0$, and $\lambda\ = -$— to obtain

$$p(x) = \mathcal{N}(x \mid \mu, \sigma\ \ ).$$

This is one reason for using the normal distribution when we do not know the true distribution. Because the normal distribution has the maximum entropy, we impose the least possible amount of structure by making this assumption.

What about the probability distribution function that *minimizes* the entropy? It turns out that there is no specific function that achieves minimal entropy. As functions place more mass on $x = \mu \pm \sigma$ and less on all other values of $x$, they lose entropy. However, any function placing exactly zero mass on all but two points does not integrate to one, and is not a valid probability distribution. There thus is no single minimal entropy probability distribution function, much as there is no single minimal positive real number.

### 19.6.3 Continuous Latent Variables

TODO: Gaussian example from IG's thesis? TODO: S3C example

## 19.7 Stochastic Inference

TODO: Charlie Tang's SFNNs? Is there anything else where sampling-based inference actually gets used?

## 19.8 Learned Approximate Inference

TODO: wake-sleep algorithm

In chapter 18.2 we saw that one possible explanation for the role of dream sleep in human beings and animals is that dreams could provide the negative phase samples that Monte Carlo training algorithms use to approximate the negative gradient of the log partition function of undirected models. Another possible

explanation for biological dreaming is that it is providing samples from $p(\boldsymbol{h}, \boldsymbol{v})$ which can be used to train an inference network to predict $\boldsymbol{h}$ given $\boldsymbol{v}$. In some senses, this explanation is more satisfying than the partition function explanation. Monte Carlo algorithms generally do not perform well if they are run using only the positive phase of the gradient for several steps then with only the negative phase of the gradient for several steps. Human beings and animals are usually awake for several consecutive hours then asleep for several consecutive hours, and it is not readily apparent how this schedule could support Monte Carlo training of an undirected model. Learning algorithms based on maximizing $\mathcal{L}$ can be run with prolonged periods of improving $q$ and prolonged periods of improving $\boldsymbol{\theta}$, however. If the role of biological dreaming is to train networks for predicting $q$, then this explains how animals are able to remain awake for several hours (the longer they are awake, the greater the gap between $\mathcal{L}$ and $\log p(v)$, but $\mathcal{L}$ will remain a lower bound) and to remain asleep for several hours (the generative model itself is not modified during sleep) without damaging their internal models. Of course, these ideas are purely speculative, and there is no hard evidence to suggest that dreaming accomplishes either of these goals. Dreaming may also serve reinforcement learning rather than probabilistic modeling, by sampling synthetic experiences from the animal's transition model, on which to train the animal's policy. Or sleep may serve some other purpose not yet anticipated by the machine learning community.

TODO: DARN and NVIL? TODO: fast DBM inference

# Chapter 20

# Deep Generative Models

In this chapter, we present several of the specific kinds of generative models that can be built and trained using the techniques presented in chapters 16, 18 and 19. All of these models represent probability distributions over multiple variables in some way. Some allow the probability distribution function to be evaluated explicitly. Others do not allow the evaluation of the probability distribution function, but support operations that implicitly require knowledge of it, such as sampling. Some of these models are structured probabilistic models described in terms of graphs and factors, as described in chapter 16. Others can not easily be described in terms of factors, but represent probability distributions nonetheless.

## 20.1 Boltzmann Machines

Boltzmann machines were originally introduced in Ackley *et al.* (1985) as a general "connectionist" approach to learning arbitrary probability distributions over binary vectors. Boltzmann Machines form the basis of a large number of popular variants. Indeed, these variants have long ago surpassed the popularity of the original and most general incarnation of the Boltzmann machine. In this section we briefly introduce the general Boltzmann machine and discuss the issues that come up when trying to train and perform inference in the model.

We define our Boltzmann machine over a $d$-dimensional binary random vector $\mathbf{x} \in \{0,1\}^d$. The Boltzmann machine is an energy-based model[1], meaning we define the joint probability distribution over the model variable using an energy function.

$$P(\boldsymbol{x}) = \frac{\exp\left(-E(\boldsymbol{x})\right)}{Z}. \tag{20.1}$$

Where $E(\boldsymbol{x})$ is the energy function and $Z$ is the partition function, that ensures

[1]For a general discussion of energy based models see Sec. 16.2.4

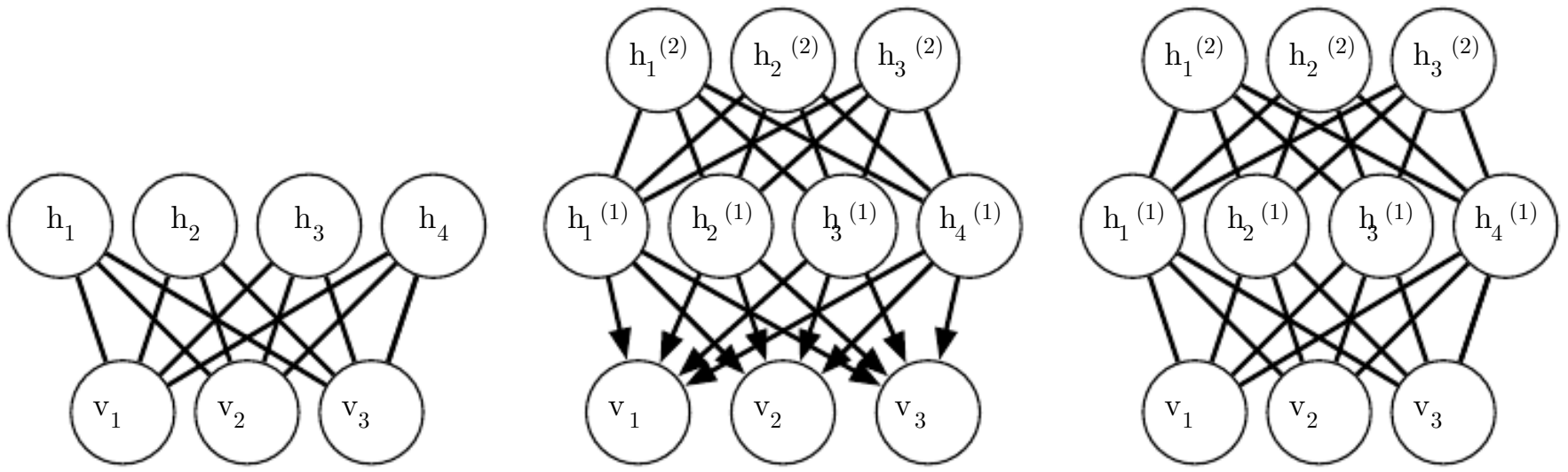


Figure 20.1: Examples of models that may be built with restricted Boltzmann machines. *a)* The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph. There are no connections among the visible units, nor any connections among the hidden units. Typically every visible unit is connected to every hidden unit but it is possible to construct sparsely connected RBMs such as convolutional RBMs. *b)* A deep belief network is a hybrid graphical model involving both directed and undirected connections. Like an RBM, it has no intra-layer connections. However, a DBN has multiple hidden layers, and thus there are connections between hidden units that are in separate layers. All of the local conditional probability distributions needed by the deep belief network are copied directly from the local conditional probability distributions of its constituent RBMs. Note that we could also represent the deep belief network with a completely undirected graph, but it would need intra-layer connections to capture the dependencies between parents. *c)* A deep Boltzmann machine is an undirected graphical model with several layers of latent variables. Like RBMs and DBNs, DBMs lack intra-layer connections. DBMs are less closely tied to RBMs than DBNs are. When initializing a DBM from a stack of RBMs, it is necessary to modify the RBM parameters slightly. Some kinds of DBMs may be trained without first training a set of RBMs.

that the $P(\boldsymbol{x}) = 1$. The energy function of the Boltzmann machine is given by:

$$E(\boldsymbol{x}) = -\boldsymbol{x} \ \ \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b} \ \ \boldsymbol{x}, \tag{20.2}$$

where $\boldsymbol{U}$ is the "weight" matrix of model parameters and $\boldsymbol{b}$ are the offsets for each $\boldsymbol{x}$.

In the general setting of the Boltzmann machine, we could consider that the goal is that we are given a set of observations, each of which are $d$-dimensional and that we are to use the joint probability distribution given in Eq. 20.1 describes the joint probability distribution over the observed variables (also called *visible units*). While this scenario is certainly viable, it does limit the kinds of interactions between the observed variables to those described by the weight matrix. Specifically it limits the model to 2nd-order interactions.

In the spirit of the "connectionist" approach to density modeling that originally inspired the Boltzmann machine, it is interesting to consider the case where not all the variables are observed. In this case, the non-observed variables, or *latent* variables can act similarly to hidden units in a multi-layer perceptron and model higher-order interactions among the visible units.

Formally, we decompose the units into two subsets: the visible units $\boldsymbol{x}_v$ and the latent (or hidden) units $\boldsymbol{x}_h$. Without loss of generality, we can re-express the energy function decomposing $\boldsymbol{x}$ into subsets $\boldsymbol{x}_v$ and $\boldsymbol{x}_h$:

$$E(\boldsymbol{x}_v, \boldsymbol{x}_h) = -\boldsymbol{x}_v \, \boldsymbol{R}\boldsymbol{x}_v - \boldsymbol{x}_v \, \boldsymbol{W}\boldsymbol{x}_h - \boldsymbol{x}_h \, \boldsymbol{S}\boldsymbol{x}_h - \boldsymbol{b} \ \ \boldsymbol{x}_v - \boldsymbol{c} \ \ \boldsymbol{x}_h, \tag{20.3}$$

**Boltzmann Machine Learning** As a probabilistic model, it is natural to consider maximum likelihood as the learning paradigm for Boltzmann machines. According to the ML paradigm, we are interested in choosing the parameters that (locally) maximize the probability of the visible units over a dataset.

Consider a dataset of $n$ examples $\boldsymbol{X}_v = [\boldsymbol{x}_v \ \ , \ldots, \boldsymbol{x}_v^t \ , \ldots, \boldsymbol{x}_v^n]$. Our goal is to maximize the likelihood of this dataset under the Boltzmann machine. Assuming the data is i.i.d, this amounts to maximizing the following:

$$\ell(\boldsymbol{\theta}) = \log P(\boldsymbol{X}_v) = \quad {}^{n} \quad \log P(\boldsymbol{x}_v^t \ ). \tag{20.4}$$
$$t$$

Of course, our Boltzmann machine does not explicitly parametrize a distribution over the visible units as $P(\boldsymbol{x}_v^t \ )$, instead, as given in Eq. 20.3, it is parametrized via an energy function to joint probability distribution over $\boldsymbol{x}_v$ and $\boldsymbol{x}_h$, the hidden units. In order to recover $P(\boldsymbol{x}_v^t \ )$, we need to *marginalize out* the influence of $\boldsymbol{x}_h$.

$$P(\boldsymbol{x}_v^t \ ) = \quad P(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ ) = \quad \frac{1}{Z} \exp \ \ -E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ ) \ \ . \tag{20.5}$$

Combining Eqs. 20.4 and 20.5 gives us our objective function we wish to maximize. Unfortunately, because $Z$ is a function of the model parameters, maximizing likelihood function is not amenable to analytically solution. Instead we will do as we do for the vast majority of deep learning models, we will follow the gradient of our objective function. The Boltzmann machine likelihood gradient is given by:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \quad {}^{n} \quad \frac{1}{Z} \log \quad \exp \quad -E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ ) \tag{20.6}$$

$$= \quad {}^{n} \quad \frac{\partial}{\partial \boldsymbol{\theta}} \quad \log \quad \exp \quad -E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ ) \quad - \frac{\partial Z}{\partial \boldsymbol{\theta}} \tag{20.7}$$

$$= \quad {}^{n} \quad \frac{\exp \quad -E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ )}{\exp \quad -E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ )} \frac{\partial}{\partial \boldsymbol{\theta}} E(\boldsymbol{x}_v^t \ , \boldsymbol{x}_h^t \ ) \quad - \frac{\partial Z}{\partial \boldsymbol{\theta}} \tag{20.8}$$

## 20.2 Restricted Boltzmann Machines

Restricted Boltzmann machines are some of the most common building blocks of deep probabilistic models. They are undirected probabilistic graphical models containing a layer of observable variables and a single layer of latent variables. RBMs may be stacked (one on top of the other) to form deeper models. See Fig. 20.1 for some examples. In particular, Fig. 20.1a shows the graph structure of an RBM itself. It is a bipartite graph: with no connections permitted between any variables in the observed layer or between any units in the latent layer.

TODO– review and pointers to other sections of the book This should be the main place where they are described in detail, earlier they are just an example of undirected models or an example of a feature learning algorithm.

TODO: please use lower-case letter names for scalars, none of this $D$ and $N$ stuff. do we even use these variable names anywhere, or do we just define them and never refer back to them? if they are never used, delete them, don't make the reader hold variables in their head for no payoff

We begin with the binary version of the restricted Boltzmann machine, but as we see later there are extensions to other types of visible and hidden units.

More formally, we will consider the observed layer to consist of a set of $D$ binary random variables which we refer to collectively with the vector $\mathbf{v}$, where the $i$th element, i.e. $v_i$ is a binary random variable. We will refer to the latent or hidden layer of $N$ binary random variables collectively as $\boldsymbol{h}$, with the $j$th random elements as $h_j$.

Like the general Boltzmann machine, the restricted Boltzmann machine is an energy-based model with the joint probability distribution specified by its energy

function:

$$P(\mathbf{v} = \boldsymbol{v}, \mathbf{h} = \boldsymbol{h}) = \frac{1}{Z} \exp\left\{-E(\boldsymbol{v}, \boldsymbol{h})\right\}.$$

Where $E(\boldsymbol{v}, \boldsymbol{h})$ is the energy function that parametrizes the relationship between the visible and hidden variables:

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}\ \ \boldsymbol{v} - \boldsymbol{c}\ \ \boldsymbol{h} - \boldsymbol{v}\ \ \boldsymbol{W}\boldsymbol{h}, \tag{20.9}$$

and the $Z$ is the normalizing constant known as the partition function:

$$Z = \qquad \exp\left\{-E(\boldsymbol{v}, \boldsymbol{h})\right\}.$$

For many undirected models, it is apparent from the definition of the partition function $Z$ that the naive method of computing $Z$ (exhaustively summing over all states) would be computationally intractable. However, it is still possible that a more cleverly designed algorithm could exploit regularities in the probability distribution to compute $Z$ faster than the naive algorithm, so the exponential cost of the naive algorithm is not a guarantee of the partiion function's intractability. In the case of restricted Boltzmann machines, there is actually a hardness result, proven by Long and Servedio (2010).

### 20.2.1 Conditional Distributions

The intractable partition function $Z$, implies that the joint probability distribution is also intractable (in the sense that the normalized probability of a given joint configuration of $[\boldsymbol{v}, \boldsymbol{h}]$ is generally not available). However, due the bipartite graph structure, the restricted Boltzmann machine has the very special property that its conditional distributions $P(\mathbf{h} \mid \mathbf{v})$ and $P(\mathbf{v} \mid \mathbf{h})$ are factorial and relatively simple to compute and sample from. Indeed, it is this property that has made the RBM a relatively popular model for a wide range of applications including image modeling (TODO CITE), speech processing (TODO CITE) and natural language processing (TODO CITE).

Deriving the conditional distributions from the joint distribution is straight-

forward.

$$
\begin{aligned}
p(\boldsymbol{h} \mid \boldsymbol{v}) &= \frac{p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{p(\boldsymbol{h}, \boldsymbol{v})}{p(\boldsymbol{v})} \\
&= \frac{1}{p(\boldsymbol{v})} \frac{1}{Z} \exp \quad \boldsymbol{b} \ \boldsymbol{v} + \boldsymbol{c} \ \ \boldsymbol{h} + \boldsymbol{v} \ \ \boldsymbol{W} \boldsymbol{h} \\
&= \frac{1}{Z} \exp \quad \boldsymbol{c} \ \boldsymbol{h} + \boldsymbol{v} \ \ \boldsymbol{W} \boldsymbol{h} \\
&= \frac{1}{Z} \exp \quad {}_{j}^{n} \quad c_j h_j + {}_{j}^{n} \quad \boldsymbol{v} \ \ \boldsymbol{W}_{,j} \boldsymbol{h}_j \\
&= \frac{1}{Z} \quad {}_{j}^{n} \quad \exp \quad c_j h_j + \boldsymbol{v} \ \ \boldsymbol{W}_{,j} \boldsymbol{h}_j
\end{aligned}
$$

Since we are conditioning on the visible units $\mathbf{v}$, we can treat these as constants w.r.t. the distribution $p(\mathbf{h} \mid \mathbf{v})$. The factorial nature of the conditional $p(\mathbf{h} \mid \mathbf{v})$ follows immediately from our ability to wright the joint probability over the vector $\boldsymbol{h}$ as the product of (unnormalized) distributions over the individual elements, $h_j$. It is now a simple matter of normalizing the distributions over the individual binary $h_j$.

$$
\begin{aligned}
P(h_j = 1 \mid \boldsymbol{v}) &= \frac{\tilde{P}(h_j = 1 \mid \boldsymbol{v})}{\tilde{P}(h_j = 0 \mid \boldsymbol{v}) + \tilde{P}(h_j = 1 \mid \boldsymbol{v})} \\
&= \frac{\exp \quad c_j + \boldsymbol{v} \ \ \boldsymbol{W}_{,j}}{\exp\{0\} + \exp\{c_j + \boldsymbol{v} \ \ \boldsymbol{W}_{,j}\}} \\
&= \text{sigmoid} \quad c_j + \boldsymbol{v} \ \ \boldsymbol{W}_{,j} \quad . \qquad (20.10)
\end{aligned}
$$

We can now express the full conditional over the hidden layer as the factorial distribution:

$$
P(\boldsymbol{h} \mid \boldsymbol{v}) = {}_{j}^{n} \quad \text{sigmoid} \quad c_j + \boldsymbol{v} \ \ \boldsymbol{W}_{,j} \quad . \qquad (20.11)
$$

A similar derivation will show that the other condition of interest to us, $P(\boldsymbol{v} \mid \boldsymbol{h})$, is also a factorial distribution:

$$
P(\boldsymbol{v} \mid \boldsymbol{h}) = {}_{i}^{d} \quad \text{sigmoid}\,(b_i + \boldsymbol{W}_{i,} \ \boldsymbol{h})\,. \qquad (20.12)
$$

### 20.2.2 RBM Gibbs Sampling

The factorial nature of these conditions is a very useful property of the RBM, and allows us to efficiently draw samples from the joint distribution via a block Gibbs sampling strategy (see section 17.1 for a more complete discussion of Gibbs sampling methods).

Block Gibbs sampling simply refers to the situation where in each step of Gibbs sampling, multiple variables (or a "block" of variables) are sampled jointly. In the case of the RBM, each iteration of block Gibbs sampling consists of two steps. **Step 1:** Sample $\boldsymbol{h}^{l} \sim P(\boldsymbol{h} \mid \boldsymbol{v}^{l})$. Due to the factorial nature of the conditionals, we can simultaneously and independently sample from all the elements of $\boldsymbol{h}^{l}$ given $\boldsymbol{v}^{l}$. **Step 2:** Sample $\boldsymbol{v}^{l} \sim P(\boldsymbol{v} \mid \boldsymbol{h}^{l})$. Again, the factorial nature of the conditional $P(\boldsymbol{v} \mid \boldsymbol{h}^{l})$ allows us can simultaneously and independently sample from all the elements of $\boldsymbol{v}^{l}$ given $\boldsymbol{h}^{l}$.

## 20.3 Training Restricted Boltzmann Machines

Despite the simplicity of the RBM conditionals, training these models is not without its complications. As a probabilistic model, a sensible inductive principle for estimating the model parameters is maximum likelihood – though other possibilities are certainly possible Marlin *et al.* (2010) and will be discussed later in Sec. 20.3.3. In the following we derive the maximum likelihood gradient with respect to the model parameters.

Let us consider that we have a batch (or minibatch) of $n$ examples taken from an i.i.d dataset (independently and identically distributed examples) $\{\boldsymbol{v}, \ldots, \boldsymbol{v}^{t}, \ldots, \boldsymbol{v}^{n}\}$. The log-likelihood under the RBM with parameters $\boldsymbol{b}$ (visible unit biases), $\boldsymbol{c}$ (hidden unit biases) and $\boldsymbol{W}$ (interaction weights) is given by:

$$
\begin{aligned}
\ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) &= \sum_{t}^{n} \log P(\boldsymbol{v}^{t}) \\
&= \sum_{t}^{n} \log \sum_{\boldsymbol{h}} P(\boldsymbol{v}^{t}, \boldsymbol{h}) \\
&= \sum_{t}^{n} \log \sum_{\boldsymbol{h}} \exp\left(-E(\boldsymbol{v}^{t}, \boldsymbol{h})\right) - n \log Z \\
&= \sum_{t}^{n} \log \sum_{\boldsymbol{h}} \exp\left(-E(\boldsymbol{v}^{t}, \boldsymbol{h})\right) - n \log \sum_{\boldsymbol{v}, \boldsymbol{h}} \exp\left\{-E(\boldsymbol{v}, \boldsymbol{h})\right\},
\end{aligned}
\tag{20.13}
$$

In the last line of the equation above, we have used the definition of the partition function.

To maximize the likelihood of the data under the restricted Boltzmann machine, we consider the gradient of the likelihood with respect to the model parameters, which we will refer to collectively as $\boldsymbol{\theta} = \{\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{W}\}$:

$$\begin{aligned}
\nabla \ \ell(\boldsymbol{\theta}) &= \nabla \ {}^{n}_{t} \log \ \exp \ -E(\boldsymbol{v}^{t}, \boldsymbol{h}) \ - n \frac{\partial}{\partial \theta} \log \ \exp\{-E(\boldsymbol{v}, \boldsymbol{h})\} \\
&= {}^{n}_{t} \frac{\exp \ -E(\boldsymbol{v}^{t}, \boldsymbol{h}) \ \nabla \ -E(\boldsymbol{v}^{t}, \boldsymbol{h})}{\exp \ -E(\boldsymbol{v}^{t}, \boldsymbol{h})} - n \frac{\exp\{-E(\boldsymbol{v}, \boldsymbol{h})\} \nabla \ -E(\boldsymbol{v}, \boldsymbol{h})}{\exp\{-E(\boldsymbol{v}, \boldsymbol{h})\}} \\
&= {}^{n}_{t} P \ \nabla \ -E(\boldsymbol{v}^{t}, \boldsymbol{h}) - n \ P \ [\nabla \ -E(\boldsymbol{v}, \boldsymbol{h})] \qquad (20.14)
\end{aligned}$$

As we can see from Eq. 20.14, the gradient of the log likelihood is specified as the difference between two expectations of the gradient of the energy function, The first expectation (the *data term*) is with respect to the product of the empirical distribution over the data, $P(v) = 1/n \ {}^{n}_{t} \ \delta(x - \boldsymbol{v}^{t})$ [2] and the conditional distribution $P(\boldsymbol{h} \mid \boldsymbol{v}^{t})$. The second expectation (the *model term*) is with respect to the joint model distribution $P(\boldsymbol{v}, \boldsymbol{h})$.

This difference between a data-driven term and a model-driven term is not unique to RBMs, as discussed in some detail in Sec. 18.2, this is a general feature of the maximum likelihood gradient for all undirected models.

We can complete the derivation of log-likelihood gradient by expanding the term: $\nabla \ -E(\boldsymbol{v}, \boldsymbol{h})$. We will consider first the gradient of the negative energy function of $\boldsymbol{W}$.

$$\begin{aligned}
nabla \ \ -E(\boldsymbol{v}, \boldsymbol{h}) &= \frac{\partial}{\partial \boldsymbol{W}} \ \boldsymbol{b} \ \ \boldsymbol{v} + \boldsymbol{c} \ \ \boldsymbol{h} + \boldsymbol{v} \ \ \boldsymbol{W}\boldsymbol{h} \qquad (20.15) \\
&= \boldsymbol{h}\boldsymbol{v} \qquad (20.16)
\end{aligned}$$

The gradients with respect to $\boldsymbol{b}$ and $\boldsymbol{c}$ are similarly derived:

$$\nabla \ -E(\boldsymbol{v}, \boldsymbol{h}) = \boldsymbol{v}, \nabla \ -E(\boldsymbol{v}, \boldsymbol{h}) = \boldsymbol{h} \qquad (20.17)$$

[2] As discussed in Sec. 3.10.5, we use the term empirical distribution to refer to a mixture over delta functions placed on training examples

Putting it all together we can the following equations for the gradients with respect to the RBM parameters and given $n$ training examples:

$$\nabla \quad \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \quad {}^{n} \quad \hat{\boldsymbol{h}}^{t} \ \boldsymbol{v}^{t} \quad - N \ {}_{P} \ , \quad \boldsymbol{h}\boldsymbol{v}$$

$$\nabla \ \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \quad {}^{n} \quad \boldsymbol{v}^{t} \ - n \ {}_{P} \ , \quad [\boldsymbol{v}]$$

$$\nabla \ \ell(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}) = \quad {}^{n} \quad \hat{\boldsymbol{h}}^{t} \ - n \ {}_{P} \ , \quad [\boldsymbol{h}]$$

where we have defined $\hat{\boldsymbol{h}}^{t}$ as

$$\boldsymbol{h}^{t} = \hat{} \quad {}_{P} \quad [\boldsymbol{h}] = \text{sigmoid} \ \boldsymbol{c} + \boldsymbol{v}^{t} \ \boldsymbol{W} \ . \tag{20.18}$$

While we are able to write down these expressions for the log-likelihood gradient, unfortunately, in most situations of interest, we are not able to use them directly to calculate gradients. The problem is the expectations over the joint model distribution $P(\boldsymbol{v}, \boldsymbol{h})$. While we have conditional distributions $P(\boldsymbol{v} \mid \boldsymbol{h})$ and $P(\boldsymbol{h} \mid \boldsymbol{v})$ that are easy to work with, the RBM joint distribution is not amenable to analytic evaluation of the expectation ${}_{P} \ , \quad [f(\boldsymbol{v}, \boldsymbol{h})]$.

This is bad news—it implies that in most cases it is impractical to compute the exact log-likelihood gradient. Fortunately, as discussed in Sec. 18.2, there are two widely used approximation strategies that have been applied to the training of RBM with some degree of success: contrastive divergence and stochastic maximum likelihood.

In the following sections we discuss two different strategies to approximate this gradient that have been applied to training the RBM. However, before getting into the actual training algorithms, it is worth considering what general approaches are available to us in approximating the log-likelihood gradient. As we mentioned, our problem stems from the expectation over the joint distribution $P(\boldsymbol{v}, \boldsymbol{h})$, but we know that we have access to factorial conditionals and that we can use these as the basis of a Gibbs sampling procedure to recover samples form the joint distribution (as discussed in Sec. 20.2.2). Thus, we can imagine using, for example, $T$ MCMC samples from $P(\boldsymbol{v}, \boldsymbol{h})$ to form a Monte Carlo estimate of the expectations over the joint distribution:

$${}_{P} \ , \quad [f(\boldsymbol{v}, \boldsymbol{h})] \approx \frac{1}{T} \quad {}^{T} \quad f(\boldsymbol{v}^{t}, \boldsymbol{h}^{t}). \tag{20.19}$$

There is a problem with this strategy that has to do with the initialization of the MCMC chain. MCMC chains typically require a burn-in period, where the chain

### 20.3.1 Contrastive Divergence Training of the RBM

As discussed in a more general context in Sec. 18.2, Contrastive divergence (CD) seeks to approximate the expectation over the joint distribution with samples drawn from short Gibbs sampling chains. CD deals with the typical requirement for an extended burn-in sample sequence by initializing these chains at the data points used in the data-dependent, conditional term. The result is a biased approximation of the log-likelihood gradient (Carreira-Perpiñan and Hinton, 2005; Bengio and Delalleau, 2009; Fischer and Igel, 2011), that never-the-less has been empirically shown to be effective. The constrastive divergence algorithm, as applied to RBMs, is given in Algorithm 20.1.

**Algorithm 20.1** The contrastive divergence algorithm, using gradient ascent as the optimization procedure.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow a Markov chain of $p(\boldsymbol{v};\theta)$ to mix when initializedfrom $p_{\text{data}}$. Perhaps 1-20 to train an RBM on a small image patch.
**while** Not converged **do**
  Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{v}^{1}, \ldots, \boldsymbol{v}^{m}\}$.
  $\Delta \leftarrow \frac{1}{m} \sum_{t}^{m} \boldsymbol{v}^{t} \hat{\boldsymbol{h}}^{t}$
  $\Delta \leftarrow \frac{1}{m} \sum_{t}^{m} \boldsymbol{v}^{t}$
  $\Delta \leftarrow \frac{1}{m} \sum_{t}^{m} \hat{\boldsymbol{h}}^{t}$
  **for** $t = 1$ to $m$ **do**
    $\tilde{\boldsymbol{v}}^{t} \leftarrow \boldsymbol{v}^{t}$
  **end for**
  **for** $l = 1$ to $k$ **do**
    **for** $t = 1$ to $m$ **do**
      $\tilde{\boldsymbol{h}}^{t}$ sampled from $\prod_{j}^{n} \text{sigmoid}\left(c_j + \tilde{\boldsymbol{v}}^{t} \boldsymbol{W}_{,j}\right)$.
      $\tilde{\boldsymbol{v}}^{t}$ sampled from $\prod_{i}^{d} \text{sigmoid}\left(b_i + \boldsymbol{W}_{i,} \tilde{\boldsymbol{h}}^{t}\right)$.
    **end for**
  **end for**
  $\bar{\boldsymbol{h}}^{t} \leftarrow \text{sigmoid}\left(\boldsymbol{c} + \tilde{\boldsymbol{v}}^{t} \boldsymbol{W}\right)$
  $\Delta \leftarrow \Delta - \frac{1}{m} \sum_{t}^{m} \tilde{\boldsymbol{v}}^{t} \bar{\boldsymbol{h}}^{t}$
  $\Delta \leftarrow \Delta - \frac{1}{m} \sum_{t}^{m} \tilde{\boldsymbol{v}}^{t}$
  $\Delta \leftarrow \Delta - \frac{1}{m} \sum_{t}^{m} \bar{\boldsymbol{h}}^{t}$
  $\boldsymbol{W} \leftarrow \boldsymbol{W} + \epsilon \Delta$
  $\boldsymbol{b} \leftarrow \boldsymbol{b} + \epsilon \Delta$
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \epsilon \Delta$
**end while**

### 20.3.2 Stochastic Maximum Likelihood for the RBM

While contrastive divergence has been the most popular method of training RBMs, the *stochastic maximum likelihood* (SML) algorithm (Younes, 1998; Tieleman, 2008) is known to be a competitive alternative – especially if we are interested in recovering the best possible generative model (i.e. achieving the highest possible test set likelihood). As with CD, the general SML algorithm is described in Sec. 18.2. Here we are concerned with how to apply the algorithm to training an RBM.

In comparison to the CD algorithm, SML uses an alternative solution to the problem of how to approximate the partition function's contribution to the log-likelihood gradient. Instead of initializing the $k$-step MCMC chain with the current example from the training set, in SML we initialize the MCMC chain for training iteration $s$ with the last state of the MCMC chain from the last training iteration ($s-1$). Assuming that the gradient updates to the model parameters do not significantly change the model, the MCMC state of the last iteration should be close to the equilibrium distribution at iteration $s$ – minimizing the number of "burn-in" MCMC steps needed to reach equilibrium at the current iteration. As with CD, in practice we often use just one Gibbs step between learning iterations. Algorithm 20.2 describes the SML algorithm as applied to RBMs.

TODO: include experimental examples, i.e. an RBM trained with CD on MNIST

### 20.3.3 Other Inductive Principles

TODO:Other inductive principles have been used to train RBMs. In this section we briefly discuss these.

## 20.4 Deep Belief Networks

*Deep belief networks* (DBNs) were one of the first successful non-convolutional architectures. The introduction of deep belief networks in 2006 began the current deep learning renaissance. Prior to the introduction of deep belief networks, deep models were considered too difficult to optimize, due to the vanishing and exploding gradient problems and the existence of plateaus, negative curvature, and suboptimal local minima that can arise in neural network objective functions. Kernel machines with convex objective functions dominated the research landscape. Deep belief networks demonstrated that deep architectures can be successful, by outperforming kernelized support vector machines on the MNIST dataset (Hinton *et al.*, 2006). Today, deep belief networks have mostly fallen out of favor and are rarely used, even compared to other unsupervised or generative

**Algorithm 20.2** The stochastic maximum likelihood / persistent contrastive divergence algorithm for training an RBM.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow a Markov chain of $p(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta} + \epsilon\Delta\ )$ toburn in, starting from samples from $p(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch.
Initialize a set of $m$ samples $\{\tilde{\boldsymbol{v}}\ \ , \ldots, \tilde{\boldsymbol{v}}^{m}\ \}$ to random values (e.g., from a uniform or normal distribution, or possibly a distribution with marginals matched to the model's marginals)
**while** Not converged **do**
  Sample a minibatch of $m$ examples $\{\boldsymbol{v}\ \ , \ldots, \boldsymbol{v}^{m}\ \}$ from the training set.
  $\Delta\ \ \leftarrow \frac{}{m}\ \ \sum_{t}^{m}\ \hat{\boldsymbol{h}}^{t}\ \boldsymbol{v}^{t}$
  $\Delta\ \leftarrow \frac{}{m}\ \ \sum_{t}^{m}\ \boldsymbol{v}^{t}$
  $\Delta\ \leftarrow \frac{}{m}\ \ \sum_{t}^{m}\ \hat{\boldsymbol{h}}^{t}$
  **for** $l = 1$ to $k$ **do**
    **for** $t = 1$ to $m$ **do**
      $\tilde{\boldsymbol{h}}^{t}$ sampled from $\sum_{j}^{n}\ \text{sigmoid}\ \ c_j + \tilde{\boldsymbol{v}}^{t}\ \ \boldsymbol{W}_{,j}$ .
      $\tilde{\boldsymbol{v}}^{t}$ sampled from $\sum_{i}^{d}\ \text{sigmoid}\ \ b_i + \boldsymbol{W}_{i,}\tilde{\boldsymbol{h}}^{t}$ .
    **end for**
  **end for**
  $\Delta\ \ \leftarrow \Delta\ \ - \frac{}{m}\ \ \sum_{t}^{m}\ \tilde{\boldsymbol{v}}^{t}\ \tilde{\boldsymbol{h}}^{t}$
  $\Delta\ \leftarrow \Delta\ - \frac{}{m}\ \ \sum_{t}^{m}\ \tilde{\boldsymbol{v}}^{t}$
  $\Delta\ \leftarrow \Delta\ - \frac{}{m}\ \ \sum_{t}^{m}\ \tilde{\boldsymbol{h}}^{t}$
  $\boldsymbol{W} \leftarrow \boldsymbol{W} + \epsilon\Delta$
  $\boldsymbol{b} \leftarrow \boldsymbol{b} + \epsilon\Delta$
  $\boldsymbol{c} \leftarrow \boldsymbol{c} + \epsilon\Delta$
**end while**

learning algorithms, but they are still deservedly recognized for their important role in deep learning history.

Deep belief networks are generative models with several layers of latent variables. The latent variables are typically binary, and the visible units may be binary or real. There are no intra-layer connections. Usually, every unit in each layer is connected to every unit in each neighboring layer, though it is possible to construct more sparsely connected DBNs. The connections between the top two layers are undirected. The connections between all other layers are directed, with the arrows pointed toward the layer that is closest to the data. See Fig. 20.1b for an example.

A DBN with $L$ hidden layers contains $L$ weight matrices: $W\ \ , \ldots, W^{L}$ . It

also contains $L+1$ bias vectors: $b \quad , \ldots, b^L$ with $b \quad$ providing the biases for the visible layer. The probability distribution represented by the DBN is given by

$$p(\boldsymbol{h}^{\ L}, \boldsymbol{h}^{\ L} \quad) \propto \exp \ \boldsymbol{b}^{\ L} \ \boldsymbol{h}^{\ L} + \boldsymbol{b}^{\ L} \quad \boldsymbol{h}^{\ L} \quad + \boldsymbol{h}^{\ L} \quad \boldsymbol{W}^{\ L} \boldsymbol{h}^{\ L} \quad ,$$

$$p(h_i^{\ l} \ = 1 \mid \boldsymbol{h}^{\ l} \quad) = \sigma \ \ b_i^{\ l} \ + \boldsymbol{W}_{,i}^{\ l} \quad \boldsymbol{h}^{\ l} \quad \forall i, \forall l \in 1, \ldots, L-2,$$

$$p(v_i = 1 \mid \boldsymbol{h} \quad) = \sigma \ \ b_i \quad + \boldsymbol{W}_{,i} \quad \boldsymbol{h} \quad \forall i.$$

In the cause of real-valued visible units, substitute

$$\boldsymbol{v} \sim \mathcal{N} \ \ \boldsymbol{v} \mid \boldsymbol{b} \quad + \boldsymbol{W} \quad \boldsymbol{h} \quad , \boldsymbol{\beta}$$

with $\boldsymbol{\beta}$ diagonal for tractability. Generalizations to other exponential family visible units are straightforward, at least in theory. Note that a DBN with only one hidden layer is just an RBM.

To generate a sample from a DBN, we first run several steps of Gibbs sampling on the top two hidden layers. This stage is essentially drawing a sample from the RBM defined by the top two hidden layers. We can then use a single pass of ancestral sampling through the rest of the model to draw a sample from the visible units.

Inference in a deep belief network is intractable due to the explaining away effect within each directed layer, and due to the interaction between the two final hidden layers. Evaluating or maximizing the standard evidence lower bound on the log-likelihood is also intractable, because the evidence lower bound takes the expectation of cliques whose size is equal to the network width.

Evaluating or maximizing the log-likelihood requires not just confronting the problem of intractable inference to marginalize out the latent variables, but also the problem of an intractable partition function within the undirected model of the last two layers.

As a hybrid of directed and undirected models, deep belief networks encounter many of the difficulties associated with both families of models. Because deep belief networks are partially undirected, they require Markov chains for sampling and have an intractable partition function. Because they are directed and generally consist of binary random variables, their evidence lower bound is intractable.

TODO–training procedure TODO–discriminative fine-tuning TODO–view of MLP as variational inference with very loose bound comment on how this does not capture intra-layer explaining away interactions comment on how this does not capture inter-layer feedback interactions TODO–quantitative analysis with AIS TODO–wake sleep?

The term "deep belief network" is commonly used incorrectly to refer to any kind of deep neural network, even networks without latent variable semantics. The

term "deep belief network" should refer specifically to models with undirected connections in the deepest layer and directed connections pointing downward between all other pairs of sequential layers.

The term "deep belief network" may also cause some confusion because the term "belief network" is sometimes used to refer to purely directed models, while deep belief networks contain an undirected layer. Deep belief networks also share the acronym DBN with dynamic Bayesian networks, which are Bayesian networks for representing Markov chains.

## 20.5 Deep Boltzmann Machines

A *deep Boltzmann machine* (DBM) is another kind of deep, generative model (Salakhutdinov and Hinton, 2009a). Unlike the deep belief network (DBN), it is an entirely undirected model. Unlike the RBM, the DBM has several layers of latent variables (RBMs have just one). But like the RBM, within each layer, each of the variables are mutually independent, conditioned on the variables in the neighboring layers. See Fig. 20.2 for the graph structure.

Like RBMs and DBNs, DBMs typically contain only binary units – as we assume in our development of the model – but it may sometimes contain real-valued visible units.

A DBM is an energy-based model, meaning that the the joint probability distribution over the model variables is parametrized by an energy function $E$. In the case of a deep Boltzmann machine with one visible layer, $\boldsymbol{v}$, and three hidden layers, $\boldsymbol{h}\quad,\boldsymbol{h}\quad and\boldsymbol{h}\quad$, the joint probability is given by:

$$P\quad \boldsymbol{v},\boldsymbol{h}\quad,\boldsymbol{h}\quad,\boldsymbol{h}\quad = \frac{1}{Z(\boldsymbol{\theta})}\exp\quad -E(\boldsymbol{v},\boldsymbol{h}\quad,\boldsymbol{h}\quad,\boldsymbol{h}\quad;\boldsymbol{\theta})\quad . \tag{20.20}$$

The DBM energy function is:

$$E(\boldsymbol{v},\boldsymbol{h}\quad,\boldsymbol{h}\quad,\boldsymbol{h}\quad;\boldsymbol{\theta}) = -\boldsymbol{v}\quad \boldsymbol{W}\quad \boldsymbol{h}\quad - \boldsymbol{h}\quad \boldsymbol{W}\quad \boldsymbol{h}\quad - \boldsymbol{h}\quad \boldsymbol{W}\quad \boldsymbol{h}\quad . \tag{20.21}$$

In comparison to the RBM energy function (Eq. 20.9), the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($W\quad$ and $W\quad$). As we will see, these connections have significant consequences for both the model behavior as well as how we go about performing inference in the model.

In comparison to fully connected Boltzmann machines (with every unit connected to every other unit), the DBM offers some similar advantages as offered by the RBM. Specifically, as illustrated in Fig. (TODO: include figure), the DBM layers can be organized into a bipartiite graph, with odd layers on one side and

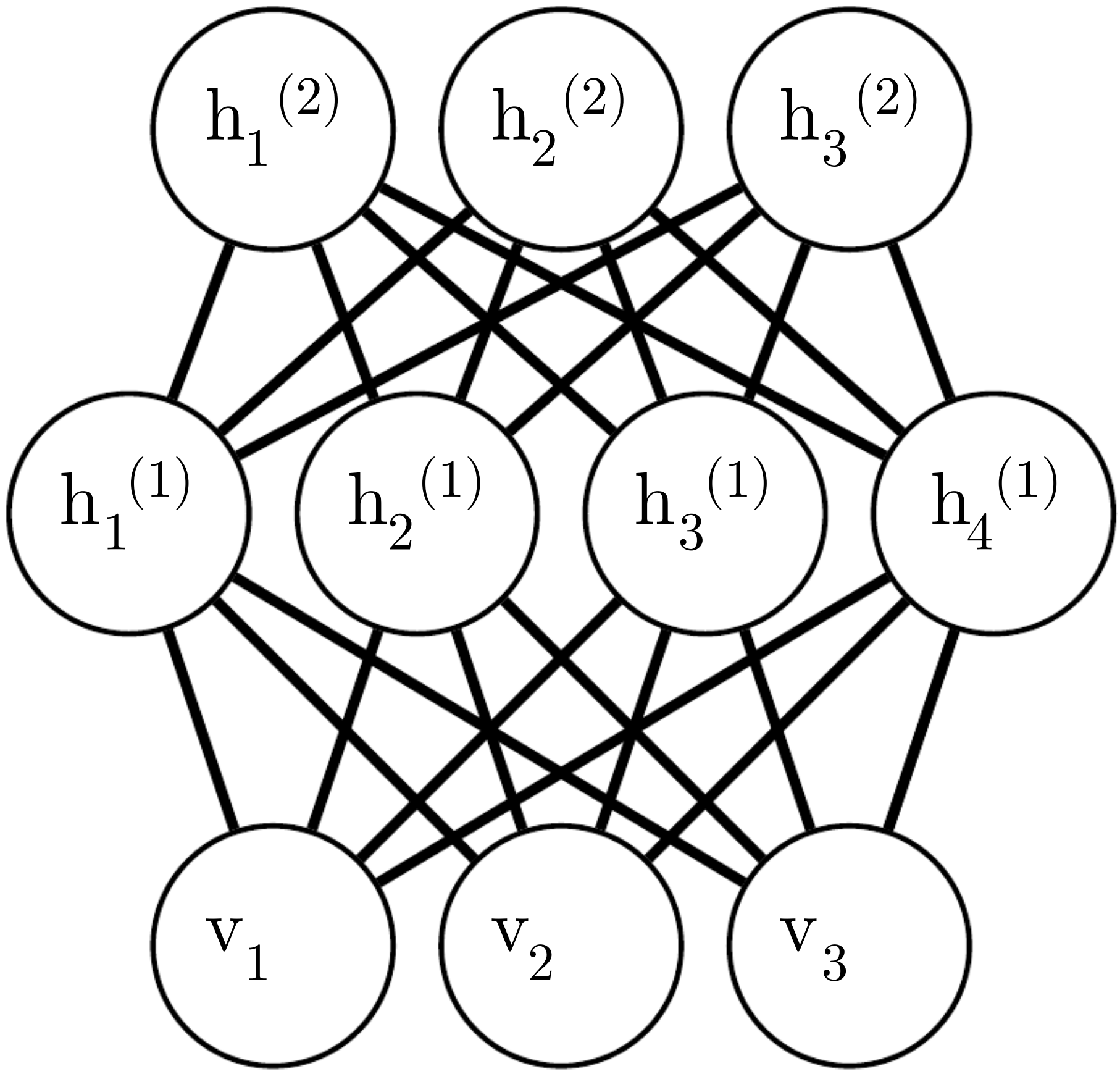


Figure 20.2: The graphical model for a deep Boltzmann machine.

even layers on the other. This immediately implies that when we condition on the variables in the even layer, the variables in the odd layers become conditionally independent. Of course, when we condition on the variables in the odd layers, the variables in the even layers also become conditionally independent.

We show this explicitly for the conditional distribution $P(\boldsymbol{h} \quad = 1 \mid \boldsymbol{v}, \boldsymbol{h} \quad)$, in the case of a DBM with two hidden layers (of course, this result generalizes to a DBM with any number of layers).

$$
\begin{aligned}
P(\boldsymbol{h} \quad \mid \boldsymbol{v}, \boldsymbol{h} \quad) &= \frac{P(\boldsymbol{h} \quad, \boldsymbol{v}, \boldsymbol{h} \quad)}{P(\boldsymbol{v}, \boldsymbol{h} \quad)} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad \boldsymbol{h} \quad + \boldsymbol{h} \quad \boldsymbol{W} \quad \boldsymbol{h}}{\cdots \quad \exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad \boldsymbol{h} \quad + \boldsymbol{h} \quad \boldsymbol{W} \quad \boldsymbol{h}} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad \boldsymbol{h} \quad + \boldsymbol{h} \quad \boldsymbol{W} \quad \boldsymbol{h}}{\cdots \quad \exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad \boldsymbol{h} \quad + \boldsymbol{h} \quad \boldsymbol{W} \quad \boldsymbol{h}} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}}{\cdots \quad \exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}}{\cdots \quad \exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}}{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}} \\
&= \frac{\exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad h \quad + h \quad \boldsymbol{W} \quad \boldsymbol{h}}{1 + \exp \quad \boldsymbol{v} \quad \boldsymbol{W} \quad + \boldsymbol{W} \quad \boldsymbol{h}} \\
&= P(h \quad \mid \boldsymbol{v}, \boldsymbol{h} \quad). \qquad (20.22)
\end{aligned}
$$

From the above we can conclude that the conditional distribution for any layer of the DBM conditioned on the neighboring layers, is fractorial (i.e. all variables in the layer are conditionally independent). Further, we've shown that

this conditional distribution is given by a logistic sigmoid function:

$$P(h_j = 1 \mid \boldsymbol{v}, \boldsymbol{h}) = \frac{\exp \boldsymbol{v} \boldsymbol{W}_{,j} + \boldsymbol{W}_{j,} \boldsymbol{h}}{1 + \exp \boldsymbol{v} \boldsymbol{W}_{,j} + \boldsymbol{W}_{j,} \boldsymbol{h}}$$
$$= \frac{1}{1 + \exp -\boldsymbol{v} \boldsymbol{W}_{,j} - \boldsymbol{W}_{j,} \boldsymbol{h}}$$
$$= \text{sigmoid} \boldsymbol{v} \boldsymbol{W}_{,j} + \boldsymbol{W}_{j,} \boldsymbol{h} . \tag{20.23}$$

For the two layer DBM, the conditional distributions of the remaining two layers $(\boldsymbol{v}, \boldsymbol{h})$ also factorize. That is $P(\boldsymbol{v} \mid \boldsymbol{h}) = {}_i^d P(v_i \mid h)$, where

$$P(v_i = 1 \mid h) = \text{sigmoid} \boldsymbol{W}_{i,} h . \tag{20.24}$$

Also, $P(h \mid h) = {}_k^m P(h_k \mid h)$, where

$$P(h_k = 1 \mid h) = \text{sigmoid} h \boldsymbol{W}_{,k} . \tag{20.25}$$

### 20.5.1 Interesting Properties

TODO: comparison to DBNs TODO: comparison to neuroscience (local learning) "most biologically plausible" TODO: description of easy mean field TODO: description of sampling, comparison to general Boltzmann machines,DBNs

### 20.5.2 DBM Mean Field Inference

For the two hidden layer DBM, the conditional distributions, $P(\boldsymbol{v} \mid \boldsymbol{h})$, $P(\boldsymbol{h} \mid \boldsymbol{v}, \boldsymbol{h})$, and $P(\boldsymbol{h} \mid \boldsymbol{h})$are factorial, however the posterior distribution over all the hidden units given the visible unit, i.e. $P(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})$, can be complicated. This is, of course, due to the interaction weights $\boldsymbol{W}$ between $\boldsymbol{h}$ and $\boldsymbol{h}$ which render these variables mutually dependent, given an observed $\boldsymbol{v}$.

So, like the DBN we are left to seek out methods to approximate the DBM posterior distribution. However, unlike the DBN, the DBM posterior distribution over their hidden units – while complicated – is easy to approximate with a *variational* approximation (as discussed in Sec. 19.1), specifically a mean field approximation. The mean field approximation is a simple form of variational inference, where we restrict the approximating distribution to fully factorial distributions. In the context of DBMs, the mean field equations capture the bidirectional interactions between layers. In this section we derive the iterative approximate inference procedure originally introduced in Salakhutdinov and Hinton (2009a)

In variational approximations to inference, we approach the task of approximating a particular target distribution – in our case, the posterior distribution over the hidden units given the visible units – by some reasonably simple family of distributions. In the case of the mean field approximation, the approximating family is the set of distributions where the hidden units are conditionally independent.

Let $Q(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})$ be the approximation of $P(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})$. The mean field assumption implies that

$$Q(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v}) = \prod_{j}^{n} Q(h_j \mid \boldsymbol{v}) \prod_{k}^{m} Q(h_k \mid \boldsymbol{v}). \tag{20.26}$$

The mean field approximation attempts to find *for every observation* a member of this family of distributions that "best fits" the true posterior $P(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})$. By best fit, we specifically mean that we wish to find the approximation $Q$ that minimizes the KL-divergence with $P$, i.e. $\mathrm{KL}(Q\|P)$ where:

$$\mathrm{KL}(Q\|P) = \sum Q(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v}) \log \frac{Q(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})}{P(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v})} \tag{20.27}$$

In general, we do not have to provide a parametric form of the approximating distribution beyond enforcing the independence assumptions. The variational approximation procedure is generally able to recover a functional form of the approximate distribution. However, in the case of a mean field assumption on binary hidden units (the case we are considering here) there is no loss of generality by fixing a parametrization of the model in advance.

We parametrize $Q$ as a product of Bernoulli distributions, that is we consider the probability of each element of $\boldsymbol{h}$ to be associated with a parameter. Specifically, for each $j \in \{1, \ldots, n\}$, $\hat{h}_j = P(h_j = 1)$, where $\hat{h}_j \in [0, 1]$ and for each $k \in \{1, \ldots, m\}$, $\hat{h}_k = P(h_k = 1)$, where $\hat{h}_k \in [0, 1]$. Thus we have the following approximation to the posterior:

$$\begin{aligned} Q(\boldsymbol{h}, \boldsymbol{h} \mid \boldsymbol{v}) &= \prod_{j}^{n} Q(h_j \mid \boldsymbol{v}) \prod_{k}^{m} Q(h_k \mid \boldsymbol{v}) \\ &= \prod_{j}^{n} (\hat{h}_j)^{h} (1 - \hat{h}_j)^{h} \times \prod_{k}^{m} (\hat{h}_k)^{h} (1 - \hat{h}_k)^{h} \end{aligned} \tag{20.28}$$

Of course, for DBMs with more layers the approximate posterior parametrization can be extended in the obvious way.

Now that we have specified our family of approximating distributions $Q$. It remains to specify a procedure for choosing the member of this family that best fits $P$. One way to do this is to explicitly minimize $\mathrm{KL}(Q\|P)$ with respect to the variational parameters of $Q$. We will approach the selection of $Q$ from a slightly different, but entirely equivalent, path. Rather than minimize $\mathrm{KL}(Q\|P)$, we will maximize the variational lower bound (or evidence lower bound: see Sec. 19.1), which in the context of the 2-hidden-layer deep Boltzmann machine is given by:

$$\begin{aligned}\mathcal{L}(Q) &= \sum_{\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)}} Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v}) \log \frac{P(\boldsymbol{v},\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)};\boldsymbol{\theta})}{q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})} \\ &= -\sum_{\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)}} Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v}) E(\boldsymbol{v},\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)};\boldsymbol{\theta}) - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q), \quad (20.29)\end{aligned}$$

where $Z(\boldsymbol{\theta})$ is the DBM partition function and $\mathcal{H}(Q)$ is the entropy of the mean field distribution.

We wish to maximize the variational lower bound in Eq. 20.29 with respect to the mean field parameters of $Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})$. Substituting Eq. 20.28 for $Q(\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)} \mid \boldsymbol{v})$ in the variational lower bound, we get:

$$\mathcal{L}(q) = \sum_{i,j} v_i W_{ij} \hat{h}_j^{(1)} + \sum_{j,k} \hat{h}_j^{(1)} W_{jk}^{(2)} \hat{h}_k^{(2)} - \ln Z(\theta) + \mathcal{H}(q). \quad (20.30)$$

We maximize the above expression (Eq. 20.30) by taking deriatives with respect to the variational parameters and solving for the system of fixed point equations:

$$\frac{\partial}{\partial \hat{h}_j^{(1)}} \mathcal{L}(q) = 0 \quad \forall j \in \{1,\ldots,n\}, \qquad \frac{\partial}{\partial \hat{h}_k^{(2)}} \mathcal{L}(q) = 0 \quad \forall k \in \{1,\ldots,m\}$$

The gradient with respect to, for example, $\hat{h}_j^{(1)}$ is reasonable straightforward

to evaluate:

$$\begin{aligned}
\frac{\partial}{\partial \hat{h}_j}\mathcal{L}(q) &= \frac{\partial}{\partial \hat{h}_j}\left( \sum_i \sum_j v_i W_{ij} \hat{h}_j + \sum_j \sum_k \hat{h}_j W_{jk} \hat{h}_k - \ln Z(\theta) + \mathcal{H}(q) \right) \\
&= \frac{\partial}{\partial \hat{h}_j}\left( \sum_i \sum_j v_i W_{ij} \hat{h}_j + \sum_j \sum_k \hat{h}_j W_{jk} \hat{h}_k - \ln Z(\theta) \right) \\
&\quad - \sum_j \left( \hat{h}_j \ln \hat{h}_j + (1-\hat{h}_j)\ln(1-\hat{h}_j) \right) \\
&\quad - \sum_k \left( \hat{h}_k \ln \hat{h}_k + (1-\hat{h}_k)\ln(1-\hat{h}_k) \right) \\
&= \sum_i v_i W_{ij} + \sum_k W_{jk} \hat{h}_k - \ln \frac{\hat{h}_j}{1-\hat{h}_j},
\end{aligned}$$

where in the second line, we have just expanded the terms involved in the entropy $\mathcal{H}(q)$. Setting this derivative to zero and solving for $\hat{h}_j$, we have

$$\frac{\partial}{\partial \hat{h}_j}\mathcal{L}(q) = 0 = \sum_i v_i W_{ij} + \sum_k W_{jk} \hat{h}_k - \ln \frac{\hat{h}_j}{1-\hat{h}_j}$$

$$\hat{h}_j = \text{sigmoid}\left( \sum_i v_i W_{ij} + \sum_k W_{jk} \hat{h}_k \right)$$

A similar derivation leads to the other set of equations for the second hidden layer variational parameters. Putting these together, we have the following system of equations:

$$\hat{h}_j = \text{sigmoid}\left( \sum_i v_i W_{ij} + \sum_k W_{jk} \hat{h}_k \right), \quad \forall j \tag{20.31}$$

$$\hat{h}_k = \text{sigmoid}\left( \sum_j W_{jk} \hat{h}_j \right), \quad \forall k \tag{20.32}$$

At a fixed point of this system of equations, we have a local maximum of our variational lower bound $\mathcal{L}(q)$. Thus they define a iterative algorithm where we intersperse updates of $\hat{h}_j$ (using Eq. 20.31) and updates of $\hat{h}_k$ (using Eq. 20.32). So variational inference in the two hidden layer deep Boltzmann machine amounts to iterating these update equations for $\hat{h}_j$ and $\hat{h}_k$ until convergence. In practice, $\approx 10$ iterations is usually sufficient. Extending approximate variational inference to deeper DBMs is straightforward.

### 20.5.3 DBM Parameter Learning

Because a deep Boltzmann machine contains restricted Boltzmann machines as components, the hardness results for computing the partition function and sampling that apply to restricted Boltzmann machines also apply to deep Boltzmann machines. This means that evaluating the probability mass function of a Boltzmann machine requires approximate methods such as annealed importance sampling. Likewise, training the model requires approximations to the gradient of the log partition function. See chapter 18 for a general description of these methods.

The posterior distribution over the hidden units in a deep Boltzmann machine is intractable, due to the interactions between different hidden layers. This means that we must use approximate inference during learning. The standard approach is to use stochastic gradient ascent on the mean field lower bound, as described in chapter 19. Mean field is incompatible with most of the methods for approximating the gradients of the log partition function described in chapter 18. Moreover, it has been observed that for contrastive divergence to work well, it is important that the samples from the posterior (e.g., for the 2 hidden layer DBM: $P(\boldsymbol{h} \quad , \boldsymbol{h} \quad | \ \boldsymbol{v})$) be exact (Salakhutdinov and Hinton, 2009b). In the case of the DBM, the intractability of the posterior means that we would have to run a Gibbs sampler until the samples converged to samples from the true posterior (i.e. until they "burned in"). Thus for the DBM, CD offers no speedup relative to naive MCMC methods. Instead, DBMs are usually trained using a variant of stochastic maximum likelihood. The negative phase samples can be generated simply by running a Gibbs sampling chain that alternates between sampling the odd-numbered layers and sampling the even-numbered layers.

Learning in the DBM can equivalently be considered as performing a variational form of the Expectation Maximization (EM) algorithm. Specifically, consider the variational lower bound for the two-layer DBM (making the dependency on the model parameters explicit):

$$\mathcal{L}(Q, \boldsymbol{\theta}) = \sum_{i} \sum_{j} v_i W_{ij} \hat{h}_j + \sum_{j} \sum_{k} \hat{h}_j W_{jk} \hat{h}_k - \ln Z(\theta) + \mathcal{H}(Q).$$

This expression lower bounds the likelihood $P(\boldsymbol{v} \mid \boldsymbol{\theta})$. So by maximizing this bound we hope to improve the likelihood. Thus we can think of $\mathcal{L}(Q, \boldsymbol{\theta})$ as a surrogate objective function for the DBM. From this perspective it is natural to consider a 2-step optimization procedure. In the first step (the E-step or expectation step), we optimize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to the variational parameters. In the case of the two-layer DBM this amounts to solving for $\hat{\boldsymbol{h}} \quad$ and $\hat{\boldsymbol{h}} \quad$ via the iterative scheme introduced above. Then in the second step (the M-step or maximization step), we optimize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$.

Note that maximizing the variational lower bound with respect to the parameters does not guarantee that we improve the true likelihood $P(\boldsymbol{v} \mid \boldsymbol{\theta})$ on every step. [3] That said, in practice we often find that we are able to make progress in training DBMs by maximizing the lower bound $\mathcal{L}(Q, \boldsymbol{\theta})$.

Unlike the standard M-step we typically have as part of the EM algorithm, our M-step will not actually maximize $\mathcal{L}(Q, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ (holding $Q$ fixed). The presence of the partition function makes it impractical to solve the system of equations $\nabla \mathcal{L}(Q, \boldsymbol{\theta}) = \mathbf{0}$ for $\boldsymbol{\theta}$. Instead we will be content to make incremental progress toward this maximum by taking a small step in the direction of the gradient $\nabla \mathcal{L}(Q, \boldsymbol{\theta})$. In the case of the 2-hidden layer DBM, this is given by:

$$\begin{aligned}
\nabla \mathcal{L}(Q, \boldsymbol{\theta}) &= \frac{\partial}{\partial \theta} \sum_i \sum_j v_i W_{ij} \hat{h}_j + \sum_j \sum_k \hat{h}_j W_{jk} \hat{h}_k - \ln Z(\theta) + \mathcal{H}(Q) \\
&= \frac{\partial}{\partial \theta} \sum_i \sum_j v_i W_{ij} \hat{h}_j + \sum_j \sum_k \hat{h}_j W_{jk} \hat{h}_k - \frac{\partial}{\partial \theta} \ln Z(\theta)
\end{aligned} \tag{20.33}$$

The first term in Eq. 20.33 is straightforward, once the values of $\hat{\boldsymbol{h}}$ and $\hat{\boldsymbol{h}}$ have been computed in the E-step. Our use of variation approximate inference has rendered learning in the DBM as analogous to training in the RBM where the likelihood gradient (Eq. 20.14) is also composed of a analytically tractable term and a term involving the gradient of the partition function:

$$-n \, \mathbb{E}_{P(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{h})} \nabla - E(\boldsymbol{v}, \boldsymbol{h}, \boldsymbol{h}) \tag{20.34}$$

Similar to the RBM case, the partition function's contribution to the gradient of the variational lower bound is intractable. We approximate it using a variational version of stochastic maximum likelihood [4] (VSML) algorithm. The non-variational version of stochastic maximum likelihood algorithm is discussed in Sec. 18.2 and is applied to RBMs in Sec. 20.3.2.

Unlike in the RBM, the interaction between the hidden units of the DBM precludes a direct application of the contrastive divergence training algorithm. Specifically the issue is that, in the positive phase, in order to get samples from

[3] In standard EM we do have just a guarantee. The difference is that in the case of standard EM we assume the true posterior is tractable and therefore we can set

. Under these conditions the lower bound is tight, i.e.

[4] Salakhutdinov and Hinton (2009a) refer to this algorithm as persistent contrastive divergence. We prefer to distinguish the variational version of the algorithm as applied to DBMs from the original stochastic maximum likelihood algorithm that directly (though stochastically) maximizes the likelihood rather than a lower bound on the likelihood as we are doing here.

the posterior $P(\boldsymbol{h} \mid \boldsymbol{v})$, one may have to wait a significant amount of time for the samples to "burn-in". The necessity for this burn-in renders CD an impractical algorithm for training CD. As far as we known, variants of CD that make use of the variational approximation for the positive phase gradient approximation have not been unexplored.

Variational stochastic maximum likelihood as applied to the DBM is given in Algorithm 20.3. Recall that we have included the offset parameters in the weight matrices $\boldsymbol{W}$ and $\boldsymbol{W}$. Note that the Gibbs sampling in the negative phase of the stochastic maximum likelihood algorithm can be divided into two blocks of updates, one including all odd layers (including the visible layer) and the other including all even layers. Due to the DBM connection pattern, given the even layers, the distribution over the odd layers is factorial and thus can be sampled simultaneously and independently as a block. Likewise given the odd layers, the even layers can be sampled simultaneously and independently as a block.

### 20.5.4 Practical Training Strategies

Unfortunately, training a DBM using stochastic maximum likelihood (as described above) from a random initialization usually results in failure. In some cases, the model fails to learn to represent the distribution adequately. In other cases, the DBM may represent the distribution well, but with no higher likelihood than could be obtained with just an RBM. Note that a DBM with very small weights in all but the first layer represents approximately the same distribution as an RBM.

It is not clear exactly why this happens. When DBMs are initialized from a pretrained configuration, training usually succeeds. See section 20.5.4 for details. One possibility is that it is difficult to coordinate the learning rate of the stochastic gradient algorithm with the number of Gibbs steps used in the negative phase of stochastic maximum likelihood. SML relies on the learning rate being small enough relative to the number of Gibbs steps that the Gibbs chain can mix again after each update to the model parameters. The distribution represented by the model can change very rapidly during the earlier parts of training, and this may make it difficult for the negative chains employed by SML to fully mix. As described in section 20.5.5, *multi-prediction deep Boltzmann machines* avoid the potential inaccuracy of SML by training with a different objective function that is less principled but easier to compute. Another possible explanation for the failure of joint training with mean field and SML is that the Hessian matrix could be poorly conditioned. This perspective motivates *centered deep Boltzmann machines*, presented in section 20.5.6, which modify the model family in order to obtain a better conditioned Hessian matrix.

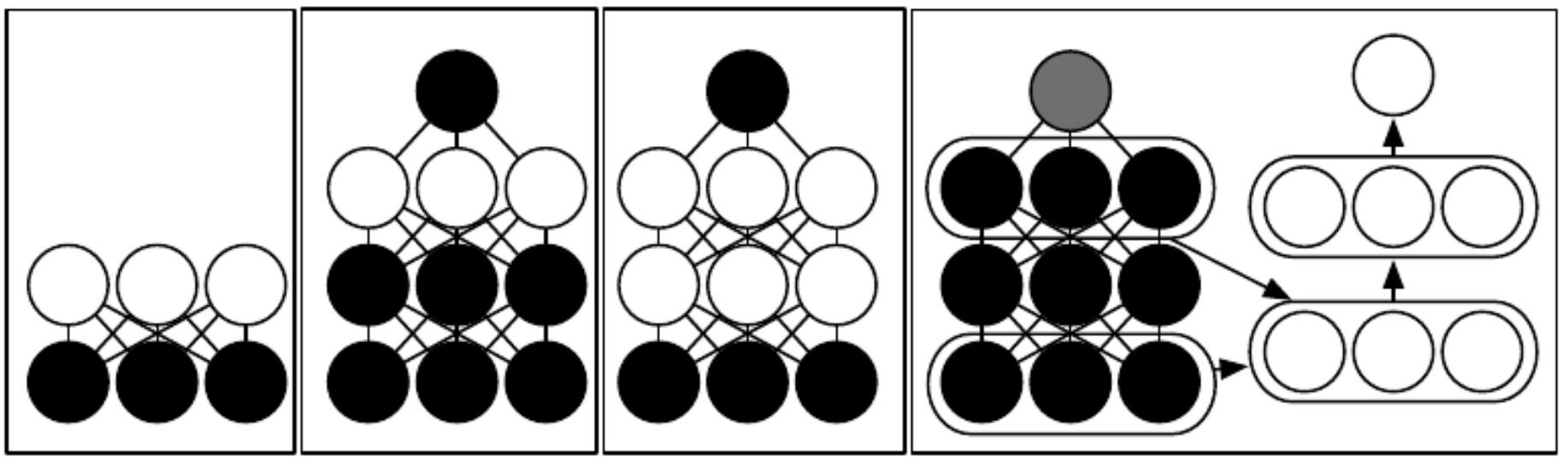

Figure 20.3: The deep Boltzmann machine training procedure used to obtain the state of the art classification accuracy on the MNIST dataset (Srivastava *et al.*, 2014; Salakhutdinov and Hinton, 2009a). TODO: this is not state of the art anymore, just best DBM result a) Train an RBM by using CD to approximately maximize $logP(\boldsymbol{v})$. b) Train a second RBM that models $\boldsymbol{h}$ and y by using CD-$k$ to approximately maximize $\log P(\boldsymbol{h}$ ,y) where $\boldsymbol{h}$ is drawn from the first RBM's posterior conditioned on the data. Increase $k$ from 1 to 20 during learning. c) Combine the two RBMs into a DBM. Train it to approximately maximize $logP(\mathbf{v}$,y ) using stochastic maximum likelihood with $k = 5$. d) Delete y from the model. Define a new set of features $\boldsymbol{h}$ and $\boldsymbol{h}$ 2) that are obtained by running mean field inference in the model lacking y. Use these features as input to an MLP whose structure is the same as an additional pass of mean field, with an additional output layer for the estimate of y. Initialize the MLP's weights to be the same as the DBM's weights. Train the MLP to approximately maximize $\log P(\text{y} \mid \mathbf{v})$ using stochastic gradient descent and dropout. Figure reprinted from (Goodfellow *et al.*, 2013b).

### Layerwise Pretraining

The original and most popular method for overcoming the joint training problem of DBMs is greedy layerwise pretraining. In this method, each layer of the DBM is trained in isolation as an RBM. The first layer is trained to model the input data. Each subsequent RBM is trained to model samples from the previous RBM's posterior distribution. After all of the RBMs have been trained in this way, they can be combined to form a DBM. The DBM may then be trained with PCD. Typically PCD training will only make a small change in the model's parameters and its performance as measured by the log likelihood it assigns to the data, or its ability to classify inputs.

Note that this greedy layerwise training procedure is not just coordinate ascent. It bears some passing resemblance to coordinate ascent because we optimize one subset of the parameters at each step. However, in the case of the greedy layerwise training procedure, we actually use a different objective function at each step.

TODO: details of combining stacked RBMs into a DBM TODO: partial mean field negative phase

### 20.5.5 Multi-Prediction Deep Boltzmann Machines

TODO– cite stoyanov TODO

### 20.5.6 Centered Deep Boltzmann Machines

TODO

This chapter has described the tools needed to fit a very broad class of probabilistic models. Which tool to use depends on which aspects of the log-likelihood are problematic.

For the simplest distributions $p$, the log-likelihood is tractable, and the model can be fit with a straightforward application of maximum likelihood estimation and gradient ascent as described in chapter

In this chapter, I've shown what to do in two different difficult cases. If $Z$ is intractable, then one may still use maximum likelihood estimation via the sampling approximation techniques described in section 18.2. If $p(h \mid v)$ is intractable, one may still train the model using the negative variational free energy rather than the likelihood, as described in 19.6.

It is also possible that *both* of these difficulties will arise. An example of this occurs with the *deep Boltzmann machine* (Salakhutdinov and Hinton, 2009b), which is essential a sequence of RBMs composed together. The model is depicted graphically in Fig. 20.1c.

This model still has the same problem with computing the partition function as the simpler RBM does. It has also discarded the restricted structure that made $P(h \mid v)$ easy to represent in the RBM. The typical way to train the DBM is to minimize the variational free energy rather than maximize the likelihood. Of course, the variational free energy still depends on the partition function, so it is necessary to use sampling techniques to approximate its gradient.

TODO: k-NADE

## 20.6 Boltzmann Machines for Real-Valued Data

While Boltzmann machines were originally developed for use with binary data, many applications such as image and audio modeling seem to require the ability to represent probability distributions over real values. In some cases, it is possible to treat real-valued data in the interval [0, 1] as representing the expectation of a binary variable (TODO cite some examples). However, this is not a particularly theoretically satisfying approach.

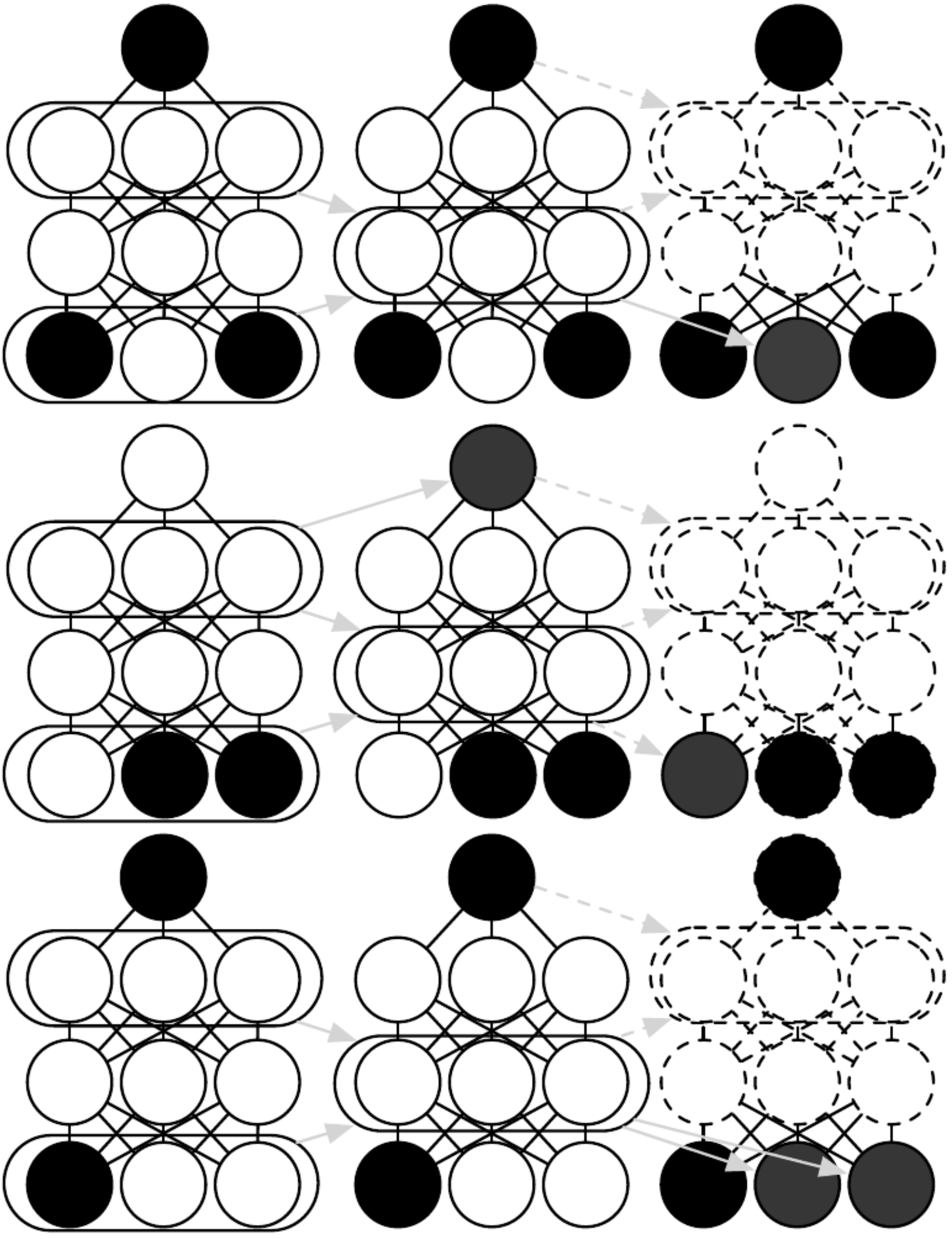

Figure 20.4: TODO caption and label, reference from text Figure reprinted from (Goodfellow *et al.*, 2013b).

### 20.6.1 Gaussian-Bernoulli RBMs

TODO– cite exponential family harmoniums? TODO– multiple ways of parametrizing them (citations?)

### 20.6.2 mcRBMs

TODO–mcRBMs [5] TODO–HMC

### 20.6.3 mPoT Model

TODO–mPoT

### 20.6.4 Spike and Slab Restricted Boltzmann Machines

Spike and slab restricted Boltzmann machines (Courville *et al.*, 2011) or ssRBMs provide another means of modeling the covariance structure of real-valued data. Compared to mcRBMs, ssRBMs have the advantage of requiring neither matrix inversion nor Hamiltonian Monte Carlo methods.

The spike and slab RBM has two sets of hidden units: the *spike* units $\mathbf{h}$ which are binary, and the slab units $\mathbf{s}$ which are real-valued. The mean of the visible units conditioned on the hidden units is given by $(\boldsymbol{h} \odot \boldsymbol{s})\boldsymbol{W}$ . In other words, each column $\boldsymbol{W}_{,i}$ defines a component that can be appear in the input. The corresponding spike variable $\mathrm{h}_i$ determines whether that component is present at all. The corresponding slab variable $\mathrm{s}_i$ determines the brightness of that component, if it is present. When a spike variable is active, the corresponding slab variable adds variance to the input along the axis defined by $\boldsymbol{W}_{,i}$. This allows us to model the covariance of the inputs. Fortunately, contrastive divergence and persistent contrastive divergence with Gibbs sampling are still applicable. There is no need to invert any matrix.

Gating by the spike variables means that the true marginal distribution over $\mathbf{h} \odot \mathbf{s}$ is sparse. This is different from sparse coding, where samples from the model "almost never" (in the measure theoretic sense) contain zeros in the code, and MAP inference is required to impose sparsity.

The primary disadvantage of the spike and slab restricted Boltzmann machine is that some settings of the parameters can correspond to a covariance matrix that is not positive definite. Such a covariance matrix places more unnormalized probability on values that are farther from the mean, causing the integral over all possible outcomes to diverge. Generally this issue can be avoided with simple heuristic tricks. There is not yet any theoretically satisfying solution. Using

[5] The term "mcRBM" is pronounced by saying the name of the letters M-C-R-B-M; the "mc" is not pronounced like the "Mc" in "McDonald's."

constrained optimization to explicitly avoid the regions where the probability is undefined is difficult to do without being overly conservative and also preventing the model from accessing high-performing regions of parameter space.

Qualitatively, convolutional variants of the ssRBM produce excellent samples of natural images. Some examples are shown in Fig. 16.1.

The ssRBM allows for several extensions. Including higher-order interactions and average-pooling of the slab variables (Courville *et al.*, 2014) enables the model to learn excellent features for a classifier when labeled data is scarce. Adding a term to the energy function that prevents the partition function from becoming undefined results in a sparse coding model, spike and slab sparse coding (Goodfellow *et al.*, 2013c), also known as S3C.

## 20.7 Convolutional Boltzmann Machines

As seen in chapter 9, extremely high dimensional inputs such as images place great strain on the computation, memory, and statistical requirements of machine learning models. Replacing matrix multiplication by discrete convolution with a small kernel is the standard way of solving these problems for inputs that have translation invariant spatial or temporal structure. Desjardins and Bengio (2008) showed that this approach works well when applied to RBMs.

Deep convolutional networks usually require a pooling operation so that the spatial size of each successive layer decreases. Feedforward convolutional networks often use a pooling function such as the maximum of the elements to be pooled. It is unclear how to generalize this to the setting of energy-based models. We could introduce a binary pooling unit $\mathrm{p}$ over $n$ binary detector units $\mathbf{d}$ and enforce $p = \max_i \mathrm{d}_i$ by setting the energy function to be $\infty$ whenever that constraint is violated. This does not scale well though, as it requires evaluating $2^n$ different energy configurations to compute the normalization constant. For a small $3 \times 3$ pooling region this requires $2 \quad = 512$ energy function evaluations per pooling unit!

Lee *et al.* (2009) developed a solution to this problem called *probabilistic max pooling* (not to be confused with "stochastic pooling," which is a technique for implicitly constructing ensembles of convolutional feedforward networks). The strategy behind probabilistic max pooling is to constrain the detector units so at most one may be active at a time. This means there are only $n + 1$ total states (one state for each of the $n$ detector units being on, and an additional state corresponding to all of the detector units being off). The pooling unit is on if and only if one of the detector units is on. The state with all units off is assigned energy zero. We can think of this as describing a model with a single variable that has $n + 1$ states, or equivalently as model that has $n + 1$ variables that assigns energy $\infty$ to all but $n + 1$ joint assignments of variables.

While efficient, probabilistic max pooling does force the detector units to be mutually exclusive, which may be a useful regularizing constraint in some contexts or a harmful limit on model capacity in other contexts. It also does not support overlapping pooling regions. Overlapping pool regions are usually required to obtain the best performance from feedforward convolutional networks, so this constraint probably greatly reduces the performance of convolutional Boltzmann machines.

Lee *et al.* (2009) demonstrated that probabilistic max pooling could be used to build convolutional deep Boltzmann machines[6]. This model is able to perform operations such as filling in missing portions of its input. However, it has not proven especially useful as a pretraining strategy for supervised learning, performing similarly to shallow baseline models introduced by Pinto *et al.* (2008).

TODO: comment on partition function changing when you change the image size, boundary issues

## 20.8 Other Boltzmann Machines

TODO–Conditional Boltzmann machine TODO-RNN-RBM TODO–discriminative Boltzmann machine TODO–Heng's class relevant and irrelevant Boltzmann machines TODO– Honglak's recent work

## 20.9 Directed Generative Nets

So far in this chapter we have focused on undirected generative models, in the deep learning context these are almost always parametrized via an energy function $E$ and possess an intractable partition function $Z$. The exception being the deep belief net which can be characterized as a hybrid directed / undirected model.

As discussed in Chapter 16, directed graphical models make up a second prominent class of graphical models. While directed graphical models have been the very popular within the greater Machine Learning community, within the smaller Deep Learning community they have until recently been overshadowed by undirected models such as the RBM.

In this section we will consider some of the standard directed graphical models that have traditionally associated with the deep learning community [7].

[6]The publication describes the model as a "deep belief network" but because it can be described as a purely undirected model with tractable layer-wise mean field fixed point updates, it best fits the definition of a deep Boltzmann machine.

[7]The list of directed graphical models that we cover here is inevitably going to be incomplete. The choice of models we include has more to do their prominence within the Deep Learning

TODO: sigmoid belief nets –> for fully observed sigmoid nets, coordinate with sec:autoregressive-nets which comes next

TODO: refer back to DBN TODO: sparse coding (maybe drop this from the list, covered elsewhere) TODO: deconvolutional nets? (AC votes for dropping this) TODO: refer back to S3C and BSC (binary sparse coding) TODO: NADE will be in RNN chapter, refer back to it here make sure k-NADE and multi-NADE are mentioned somewhere

TODO: refer to DARN and NVIL?

TODO: Stochastic Feedforward nets

### 20.9.1 Sigmoid Belief Nets

Sigmoid Belief Nets were originally conceived in response to the Neal (1992)

is one of the first

### 20.9.2 Differentiable Generator Nets

TODO describe how VAEs and GANs both use the same kind of generator net cite the generating chairs paper to show how this generator net can be trained with a procedure that isn't explicitly unsupervised cite both Kevin and Zoubin's version of training a generator net with MMD

### 20.9.3 Variational Autoencoders

The variational autoencoder is model

$$\mathcal{L} \tag{20.35}$$

TODO

### 20.9.4 Variational Interpretation of PSD

TODO, develop the explanation of Sec. 9.1 of Bengio *et al.* (2013c).

### 20.9.5 Generative Adversarial Networks

TODO: do we want to still use the capital value function here? Should we say it's a functional? Note that the G is OK because it's a distribution

Generative adversarial networks (TODO cite) are another kind of generative model based on differentiable mappings from input noise to samples that resemble the data. In this sense, they closely resemble variational autoencoders. However,

community and the perceived impact that they have had on the community.

the training procedure is different, and generative model is not necessarily coupled with an inference network. It is theoretically possible to train an inference network using a strategy similar to the wake-sleep algorithm, but there is no need to infer posterior variables during training.

Generative adversarial networks are based on game theory. A *generator network* is trained to map input noise $\boldsymbol{z}$ to samples $\boldsymbol{x}$. This function $g(\boldsymbol{z})$ defines the generative model. The distribution $p(\boldsymbol{z})$ is not learned; it is simply fixed to some distribution at the start of training (usually a very unstructured distribution such as a normal or uniform distribution). We can think of $g(\boldsymbol{z})$ as defining a conditional distribution

$$p(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x} \mid g \quad \boldsymbol{z}), \frac{1}{\beta}\boldsymbol{I} \quad ,$$

but in all learning rules we take limit as $\beta \to \infty$ so we can treat $g(\boldsymbol{z})$ itself as a sample and ignore the parametrization of the output distribution.

The generator $g$ is pitted against an adversary: a discriminator network $d$. The discriminator network receives data or samples $\boldsymbol{x}$ as input and outputs its estimate of the probability that $\boldsymbol{x}$ was sampled from the data rather than the model. During training, $d$ tries to maximize and $g$ tries to minimize a value function measuring the log probability of $d$ being correct:

$$g \quad = \arg\min_{g} \max_{d} V(g, d)$$

where

$$v(g, d) = \qquad {}_{p\ \text{data}} \log d(\boldsymbol{x}) + \qquad {}_{p_{\text{model}}} \log\left(1 - d(\boldsymbol{x})\right).$$

The optimization of $g$ can be done simply by backpropagating through $d$ then $g$, so the learning process requires neither approximate inference nor approximation of a partition function gradient. In the case where $\max_d v(g, d)$ is convex (such as the case where optimization is performed directly in the space of probability density functions) then the procedure is guaranteed to converge and is asymptotically consistent. In practice, the procedure can be difficult to make work, because it can be difficult to keep $d$ optimized well enough to provide a good estimate of how to update $g$ at all times.

### 20.9.6 Convolutional Generative Networks

TODO– discuss convolutional generator nets (was GANs paper the first?) and be sure to cover "unpooling" include the unpooling technique from generatoring chairs paper, and include others if there are relevant others

## 20.10 Auto-Regressive Networks

Auto-regressive networks are similar to recurrent networks in the sense that we also decompose a joint probability over the observed variables as a product of conditionals of the form $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t \quad}, \ldots, \boldsymbol{x} \ )$ but we drop the form of parameter sharing that makes these conditionals all share the same parametrization across time. This makes sense when the variables are *not* elements of a translation-equivariant sequence (see Section 9.2 for more on equivariance), but instead form an arbitrary tuple without any particular ordering that would correspond to a translation-equivariant form of relationship between variables at position $k$ and variables at position $k$ . Such models have been called *fully-visible Bayes networks* (FVBNs) and used successfully in many forms, first with logistic regression for each conditional distribution (Frey, 1998) and then with neural networks (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described in Section 20.10.3 below, we can re-introduce a form of parameter sharing that is different from the one found in recurrent networks, but that brings both a statistical advantage (less parameters) and a computational advantage (less computation). Although we drop the sharing over time, as we see below in Section 20.10.2, using a deep learning concept of *reuse of features*, we can *share* features that have been computed for predicting $\boldsymbol{x}_{t \ k}$ with the sub-network that predicts $\boldsymbol{x}_t$.

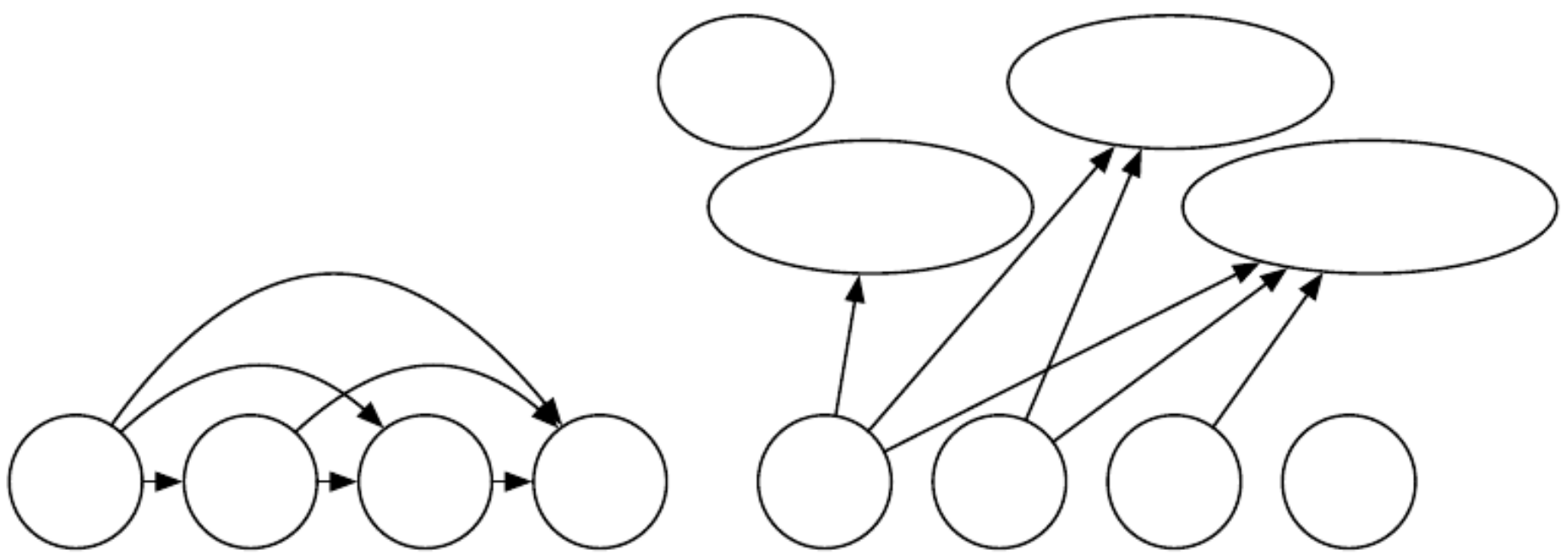

Figure 20.5: A fully visible belief network predicts the $i$-th variable from the $i-1$ previous ones. Left: corresponding graphical model (which is the same as that of a recurrent network). Right: corresponding computational graph, in the case of the logistic FVBN, where each prediction has the form of a logistic regression, with $i$ free parameters (for the $i-1$ weights associated with $i-1$ inputs, and an offset parameter).

### 20.10.1 Logistic Auto-Regressive Networks

Let us first consider the simplest auto-regressive network, without hidden units, and hence no sharing at all. Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ is parametrized as a linear model (linear regression for real-valued data, logistic regression for binary data, softmax regression for discrete data). This model was introduced by Frey (1998) and has $O(T^2)$ parameters when there are $T$ variables to be modeled. It is illustrated in Fig. 20.5, showing both the graphical model (left) and the computational graph (right).

A clear disadvantage of the logistic auto-regressive network is that one cannot easily increase its capacity in order to capture more complex data distributions. It defines a parametric family of fixed capacity, like the linear regression, the logistic regression, or the Gaussian distribution. In fact, if the variables are continuous, one gets a linear auto-regressive model, which is thus another way to formulate a Gaussian distribution, i.e., only capturing pairwise interactions between the observed variables.

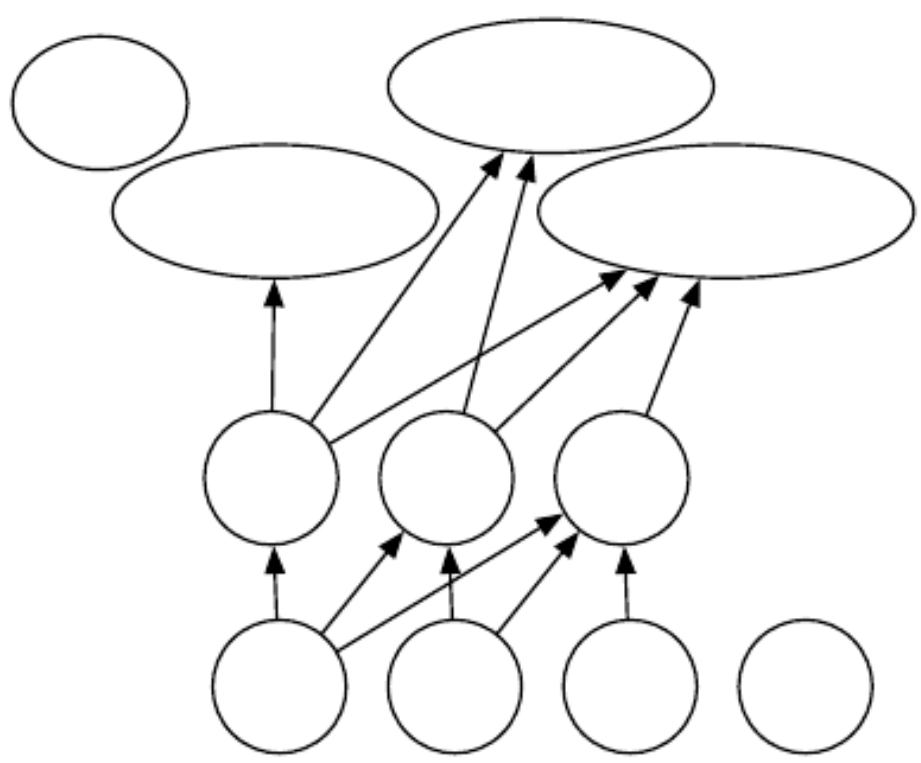

Figure 20.6: A neural auto-regressive network predicts the $i$-th variable $\boldsymbol{x}_i$ from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted $\boldsymbol{h}_i$) that are functions of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ can be reused in predicting all of the subsequent variables $\boldsymbol{x}_{i+1}, \boldsymbol{x}_{i+2}, \ldots$.

### 20.10.2 Neural Auto-Regressive Networks

Neural Auto-Regressive Networks have the same left-to-right graphical model as logistic auto-regressive networks (Fig. 20.5, left) but a different parametrization that is at once more powerful (allowing to extend the capacity as needed and approximate any joint distribution) and can improve generalization by introducing a parameter sharing and feature sharing principle common to deep learning in general. The first paper on neural auto-regressive networks by Bengio and Bengio

(2000b) (see also Bengio and Bengio (2000a) for the more extensive journal version) were motivated by the objective to avoid the curse of dimensionality arising out of traditional non-parametric graphical models, sharing the same structure as Fig. 20.5 (left). In the non-parametric discrete distribution models, each conditional distribution is represented by a table of probabilities, with one entry and one parameter for each possible configuration of the variables involved. By using a neural network instead, two advantages are obtained:

1. The parametrization of each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t \quad}, \ldots, \boldsymbol{x}\ )$ by a neural network with $(t-1) \times k$ inputs and $k$ outputs (if the variables are discrete and take $k$ values, encoded one-hot) allows one to estimate the conditional probability without requiring an exponential number of parameters (and examples), yet still allowing to capture high-order dependencies between the random variables.

2. Instead of having a different neural network for the prediction of each $\boldsymbol{x}_t$, a *left-to-right* connectivity illustrated in Fig. 20.6 allows one to merge all the neural networks into one. Equivalently, it means that the hidden layer features computed for predicting $\boldsymbol{x}_t$ can be reused for predicting $\boldsymbol{x}_{t\ k}$ $(k > 0)$. The hidden units are thus organized in *groups* that have the particularity that all the units in the $t$-th group only depend on the input values $\boldsymbol{x}\ , \ldots, \boldsymbol{x}_t$. In fact the parameters used to compute these hidden units are jointly optimized to help the prediction of all the variables $\boldsymbol{x}_{t \quad}, \boldsymbol{x}_{t \quad}, \ldots$. This is an instance of the *reuse principle* that makes *multi-task learning* and *transfer learning* successful with neural networks and deep learning in general (See Sections 7.12 and 14.2).

Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t \quad}, \ldots, \boldsymbol{x}\ )$ can represent a conditional distribution by having outputs of the neural network predict *parameters* of the conditional distribution of $\boldsymbol{x}_t$, as discussed in Section 6.3.2. Although the original neural auto-regressive networks were initially evaluated in the context of purely discrete multivariate data (with a sigmoid output for a Bernoulli variable or softmax output for a multinoulli variable) it is natural to extend such models to continuous variables or joint distributions involving both discrete and continuous variables. For example, Uria *et al.* (2013) developed an extension to real-valued variables called RNADE.

### 20.10.3 NADE

NADE (Larochelle and Murray, 2011) is a very successful recent form of neural auto-regressive network. The connectivity is the same as for the original neural auto-regressive network of Bengio and Bengio (2000b) but NADE introduces a parameter sharing scheme. The parameters of the hidden units of different groups

$j$ are shared. The weights $W_{jki}$ from the $i$-th input $\boldsymbol{x}_i$ to the $k$-th element of the $j$-th group of hidden unit $h_{jk}$ ($j \geq i$) are shared:

$$W_{jki} = W_{ki}.$$

Larochelle and Murray (2011) chose this sharing scheme so that forward propagation in a NADE model loosely resembles the computations performed in mean-field inference to fill in missing inputs in an RBM. This mean-field inference corresponds to running a recurrent network with shared weights and the first step of that inference is the same as in NADE. The only difference is that with the proposed NADE, the output weights are not forced to be simply transpose values of the input weights (they are not tied). One could imagine actually extending this procedure to not just one time step of the mean-field recurrent inference but to $k$ steps, as in Raiko *et al.* (2014).

Although the neural auto-regressive networks and NADE were originally proposed to deal with discrete distributions, they can in principle be generalized to continuous ones by replacing the conditional discrete probability distributions (for $P(\boldsymbol{x}_j \mid \boldsymbol{x}_j \quad , \ldots, \boldsymbol{x} \ )$) by continuous ones and following general practice to predict continuous random variables with neural networks (see Section 6.3.2) using the log-likelihood framework. A fairly generic way of parametrizing a continuous density is as a Gaussian mixture. RNADE uses this parameterization to extend NADE to real values. Stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means and the conditional variances. The gradient on the mean is divided by the conditional variance, so the gradient can become large when the variances become small. Uria *et al.* (2013) have used a heuristic to rescale the gradient on the component means by the associated standard deviation which seems to have helped optimizing RNADE.

Another very interesting extension of the neural auto-regressive architectures gets rid of the need to choose an arbitrary *order* for the observed variables (Murray and Larochelle, 2014). In auto-regressive networks, the idea is to train the network to be able to cope with any order by randomly sampling orders and providing the information to hidden units specifying which of the inputs are observed (on the right side of the conditioning bar) and which are to be predicted and are thus considered missing (on the left side of the conditioning bar). This is nice because it allows one to use a trained auto-regressive network to *perform any inference* (i.e. predict or sample from the probability distribution over any subset of variables given any subset) extremely efficiently. Finally, since many orders of variables are possible ($n!$ for $n$ variables) and each order $o$ of variables yields a different $p(\mathbf{x} \mid o)$, we can form an ensemble of models for many values of $o$:

$$p_{\text{ensemble}}(\mathbf{x}) = \frac{1}{k} \sum_{i}^{k} p(\mathbf{x} \mid o).$$

This ensemble model usually generalizes better and assigns higher probability to the test set than an individual model defined by a single ordering does.

In the same paper, the authors propose to consider deep versions of the architecture, but unfortunately that immediately makes computation as expensive as in the original neural auto-regressive neural network (Bengio and Bengio, 2000b). The first layer and the output layer can still be computed in $O(nh)$ multiply-add operations, as in the regular NADE, where $h$ is the number of hidden units (the size of the groups $h_i$, in Fig. 20.6), whereas it is $O(n\ h)$ in Bengio and Bengio (2000b). However, for the other hidden layers, the computation is $O(n\ h\ )$ if every "previous" group at layer $l$ participates in predicting the "next" group at layer $l+1$, assuming $n$ groups of $h$ hidden units at each layer. Making the $i$-th group at layer $l+1$ only depend on the $i$-th group, as in Murray and Larochelle (2014) at layer $l$ reduces it to $O(nh\ )$, which is still $h$ times worse than the regular NADE.

## 20.11 A Generative View of Autoencoders

Many kinds of autoencoders can be viewed as probabilistic models. Different autoencoders can be interpreted as probabilistic models in different ways.

One of the first probabilistic interpretations of autoencoders was the view denoising autoencoders as energy-based models trained using regularized score matching. See Sections 13.8.1 and 18.5 for details. Since the early work (Vincent, 2011a) made the connection with Gaussian RBMs, this gave denoising autoencoders with a particular parametrization a generative interpretation (they could be sampled from using the MCMC sampling techniques for Gaussian RBMs) .

The next milestone in connecting auto-encoders with a generative interpretation came with the work of Rifai *et al.* (2012). It relied on the view of contractive auto-encoders as estimators of the *tangent of the manifold* near which probability concentrates, discussed in Section 13.9 (see also Figures 13.9, 15.3). In this context, Rifai *et al.* (2012) demonstrated experimentally that good samples could be obtained from a trained contractive auto-encoder by alternating encoding, decoding, and adding noise in a particular way.

As discussed in Section 13.8.1, the application of the encoder/decoder pair moves the input configuration towards a more probable one. This can be exploited to actually sample from the estimated distribution. If you consider most Monte-Carlo Markov Chain (MCMC) algorithms, they have two elements:

1. move from lower probability configurations towards higher probability configurations, and

2. inject randomness so that the chain moves around (and does not stay stuck at some peak of probability, or mode) and has a chance to visit every config-

uration in the whole space, with a relative frequency equal to its probability under the underlying model.

So conceptually all one needs to do is to perform encode-decode operations (go towards more probable configurations) as well as inject noise (to move around the probable configurations), as hinted at in (Mesnil *et al.*, 2012; Rifai *et al.*, 2012).

### 20.11.1 Markov Chain Associated with any Denoising Auto-Encoder

The above discussion left open the question of what noise to inject and where, in order to obtain a Markov chain that would generate from the distribution estimated by the auto-encoder. Bengio *et al.* (2013b) showed how to construct such a Markov chain for *generalized denoising autoencoders*. Generalized denoising autoencoders are specified by a denoising distribution for sampling an estimate of the clean input given the corrupted input.

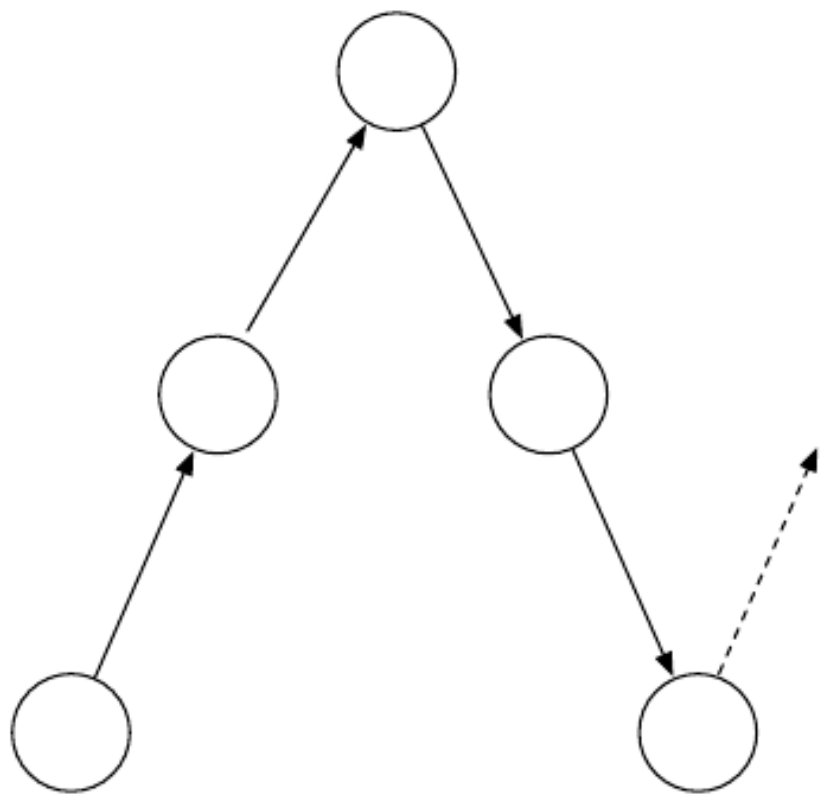

Figure 20.7: Each step of the Markov chain associated with a trained denoising auto-encoder, that generates the samples from the probabilistic model implicitly trained by the denoising reconstruction criterion. Each step consists in (a) injecting corruption $C$ in state $\boldsymbol{x}$, yielding $\tilde{\boldsymbol{x}}$, (b) encoding it with $f$, yielding $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$, (c) decoding the result with $g$, yielding parameters $\omega$ for the reconstruction distribution, and (d) given $\omega$, sampling a new state from the reconstruction distribution $P(\mathbf{x} \mid \omega = g(f(\tilde{\boldsymbol{x}})))$. In the typical squared reconstruction error case, $g(\boldsymbol{h}) = \hat{\boldsymbol{x}}$, which estimates $E[\boldsymbol{x} \mid \tilde{\boldsymbol{x}}]$, corruption consists in adding Gaussian noise and sampling from $P(\mathbf{x} \mid \omega)$ consists in adding another Gaussian noise to the reconstruction $\hat{\boldsymbol{x}}$. The latter noise level should correspond to the mean squared error of reconstructions, whereas the injected noise is a hyperparameter that controls the mixing speed as well as the extent to which the estimator *smoothes* the empirical distribution (Vincent, 2011b). In the figure, only the $C$ and $P$ conditionals are stochastic steps ($f$ and $g$ are deterministic computations), although noise can also be injected inside the auto-encoder, as in generative stochastic networks (Bengio *et al.*, 2014b)

Each step of the Markov chain that generates from the estimated distribution consists of the following sub-steps, illustrated in Figure 20.7:

1. starting from the previous state $\boldsymbol{x}$, inject corruption noise, sampling $\tilde{\boldsymbol{x}}$ from $C(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$.

2. Encode $\tilde{\boldsymbol{x}}$ into $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$.

3. Decode $\boldsymbol{h}$ to obtain the parameters $\omega = g(\boldsymbol{h})$ of $P(\mathbf{x} \mid \omega = g(\boldsymbol{h})) = P(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.

4. Sample the next state $\boldsymbol{x}$ from $P(\mathbf{x} \mid \omega = g(\boldsymbol{h})) = P(\mathbf{x} \mid \tilde{\boldsymbol{x}})$.

The theorem states that if the auto-encoder $P(\mathbf{x} \mid \tilde{\mathbf{x}})$ forms a consistent estimator of corresponding true conditional distribution, then the stationary distribution of

the above Markov chain forms a consistent estimator (albeit an implicit one) of the data generating distribution of $\mathbf{x}$.

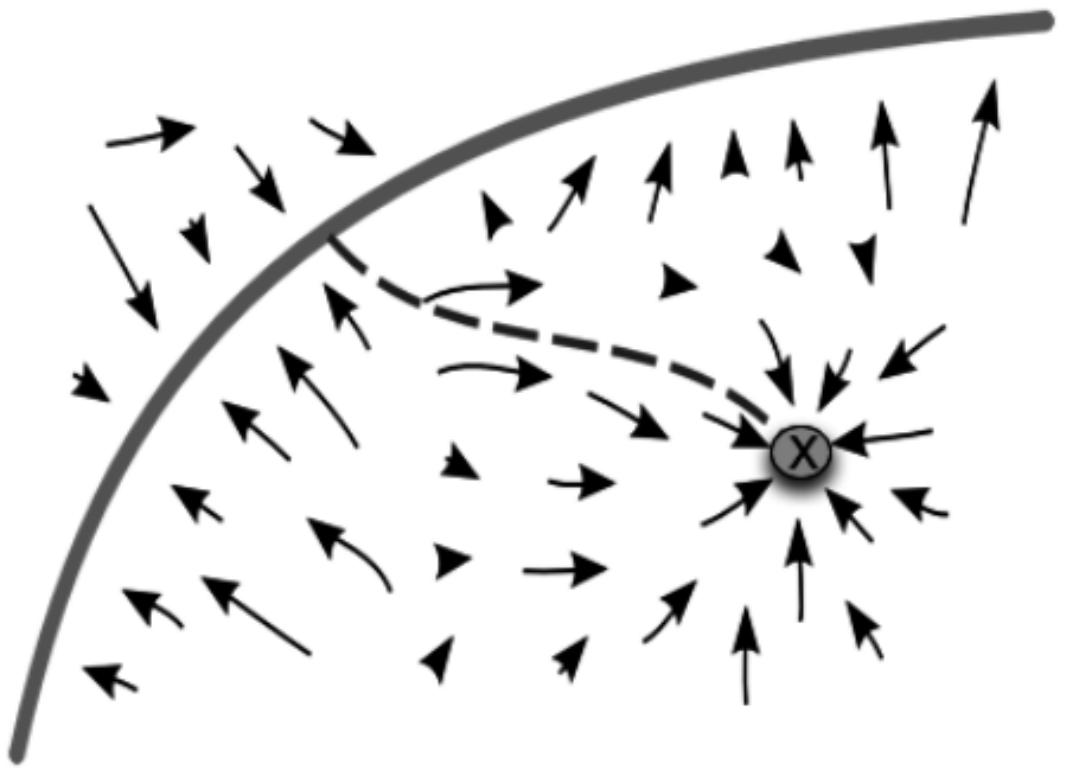

Figure 20.8: Illustration of the effect of the walk-back training procedure, used for denoising auto-encoders or GSNs in general. The objective is to remove spurious modes faster by letting the Markov chain go towards them (along the red path, starting on the purple data manifold and following the arrows plus noise), and then punishing the Markov chain for this behavior (i.e., walking back to the right place) by telling the chain to return towards the data manifold (reconstruct the original data).

### 20.11.2 Clamping and Conditional Sampling

Similarly to Boltzmann machines, denoising auto-encoders and GSNs can be used to sample from a conditional distribution $P(\mathbf{x}_f \mid \mathbf{x}_o)$, simply by clamping the *observed* units $\mathbf{x}_f$ and only resampling the *free* units $\mathbf{x}_o$ given $\mathbf{x}_f$ and the sampled latent variables (if any). This has been introduced by Bengio *et al.* (2014b).

However, note that Proposition 1 of that paper is missing a condition: the transition operator (defined by the stochastic mapping going from one state of the chain to the next) should satisfy *detailed balance*, described in Section 17.1.1.

An experiment in clamping half of the pixels (the right part of the image) and running the Markov chain on the other half is shown in Figure 20.9.

Figure 20.9: Illustration of clamping the right half of the image and running the Markov by resampling only the left half at each step. These samples come from a GSN trained to reconstruct MNIST digits at each time step, i.e., using the walkback procedure.

### 20.11.3 Walk-Back Training Procedure

The walk-back training procedure was proposed by Bengio *et al.* (2013b) as a way to speed-up the convergence of generative training of denoising auto-encoders. Instead of performing a one-step encode-decode reconstruction, this procedure consists in alternative multiple stochastic encode-decode steps (as in the generative Markov chain) initialized at a training example (just like with the contrastive divergence algorithm, described in Sections 18.2) and 20.3.1) and penalizing the last probabilistic reconstructions (or all of the reconstructions along the way).

It was shown in that paper that training with $k$ steps is equivalent (in the sense of achieving the same stationary distribution) as training with one step, but practically has the advantage that spurious modes farther from the data can be removed more efficiently, as illustrated in Figure 20.8.

Figure 20.10 illustrates the application of the walk-back procedure in a generative stochastic network, which is described in the next section.

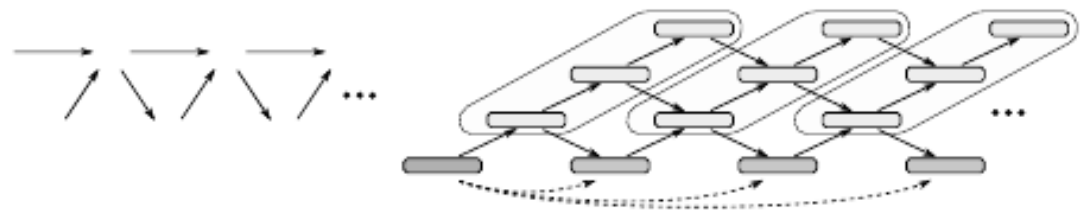

Figure 20.10: Left: graphical model of the generative Markov chain associated with a generative stochastic network (GSN). Right specific case where the latent variable is formed of several layers, each connected to the one above and the one below, making the generative process very similar to Gibbs sampling in a deep Boltzmann machine (Salakhutdinov and Hinton, 2009b). The walk-back training procedure is used, i.e., at every step the reconstruction probability distribution is pushed towards generating the training example (which also initializes the chain).

## 20.12 Generative Stochastic Networks

Generative stochastic networks (Bengio *et al.*, 2014b) or GSNs are generalizations of denoising auto-encoders that include latent variables in the generative Markov chain, in addition to the visible variables (usually denoted $\boldsymbol{x}$). The generative Markov chain looks like the one in Figure 20.11. An example of a GSN structured like a deep Boltzmann machine and trained by the walk-back procedure is shown in Figure 20.10.

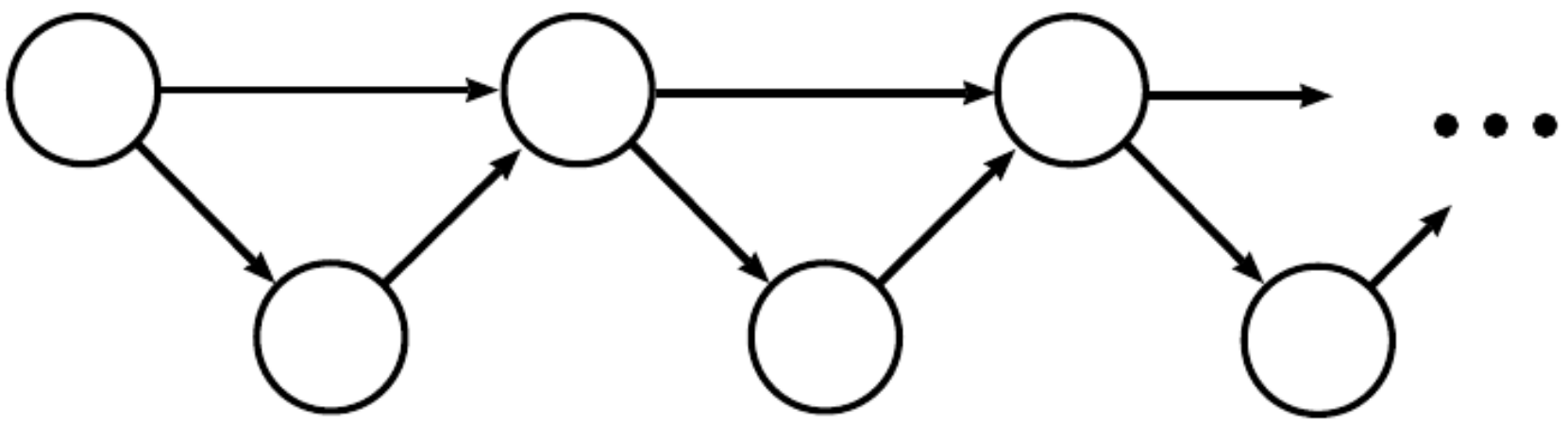

Figure 20.11: Markov chain of a GSN (Generative Stochastic Network) with latent variables with $H$ and visible variable $X$, i.e., an unfolding of the generative process with $X$ and $H$ at step $k$ of the chain. TODO: please use h and x etc. throughout the GSN section

A GSN is parametrized by two conditional probability distributions which specify one step of the Markov chain:

1. $P(X_k \mid H_k)$ tells how to generate the next visible variable given the current latent state. Such a "reconstruction distribution" is also found in denoising auto-encoders, RBMs, DBNs and DBMs.

2. $P(H_k \mid H_{k} \quad , X_{k} \quad )$ tells how to update the latent state variable, given the previous latent state and visible variable.

Denoising auto-encoders and GSNs differ from classical probabilistic models (directed or undirected) in that it parametrizes the generative process itself rather than the mathematical specification of the joint distribution of visible and latent variables. Instead, the latter is defined *implicitly*, *if it exists*, as the stationary distribution of the generative Markov chain. The conditions for existence of the stationary distribution are mild (basically, the chain mixes) but can be violated by some choices of the transition distributions (for example, if they were deterministic).

One could imagine different training criteria for GSNs. The one proposed and evaluated by Bengio *et al.* (2014b) is simply reconstruction log-probability on the visible units, just like for denoising auto-encoders. This is achieved by clamping $X = x$ to the observed example and maximizing the probability of generating $x$ at some subsequent time steps, i.e., maximizing $\log P(X_k = x \mid H_k)$, where $H_k$ is sampled from the chain, given $X = x$. In order to estimate the gradient of $\log P(X_k = x \mid H_k)$ with respect to the other pieces of the model, Bengio *et al.* (2014b) use the reparametrization trick, introduced in Section 16.5.1.

The *walk-back training* protocol (described in Section 20.11.3 was used (Bengio *et al.*, 2014b) to improve training convergence of GSNS.

### 20.12.1 Discriminant GSNs

Whereas the original formulation of GSNs (Bengio *et al.*, 2014b) was meant for unsupervised learning and implicitly modeling $P(\mathbf{x})$ for observed data $\mathbf{x}$, it is possible to modify the framework to optimize $P(\mathbf{y} \mid \boldsymbol{x})$.

For example, Zhou and Troyanskaya (2014) generalize GSNs in this way, by only back-propagating the reconstruction log-probability over the output variables, keeping the input variables fixed. They applied this successfully to model *sequences* (protein secondary structure) and introduced a (one-dimensional) *convolutional* structure in the transition operator of the Markov chain. Keep in mind that, for each step of the Markov chain, one generates a new sequence for each layer, and that sequence is the input for computing other layer values (say the one below and the one above) at the next time step, as illustrated in Figure 20.12.

Hence the Markov chain is really over the *output variable* (and associated higher-level hidden layers), and the input sequence only serves to condition that chain, with back-propagation allowing to learn how the input sequence can condition the output distribution implicitly represented by the Markov chain. It is therefore a case of using the GSN in the context of *structured outputs*, where $P(\mathbf{y} \mid \mathbf{x})$ does not have a simple parametric form but instead the components of $\mathbf{y}$ are statistically dependent of each other, given $\mathbf{x}$, in complicated ways.

Zöhrer and Pernkopf (2014) considered a hybrid model that combines a supervised objective (as in the above work) and an unsupervised objective (as in the

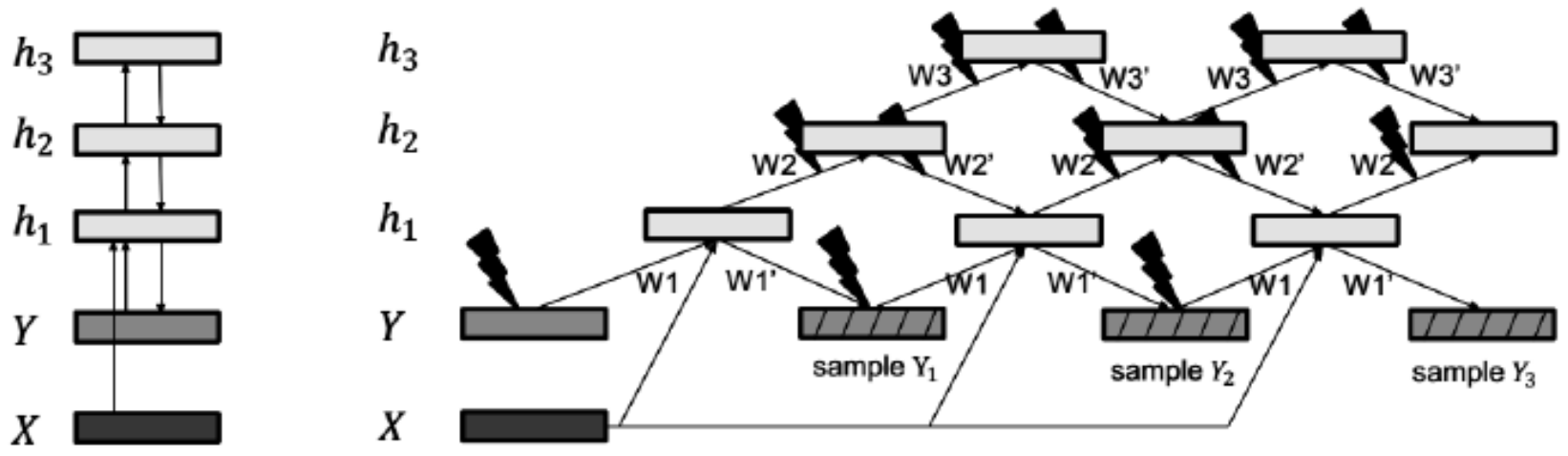


Figure 20.12: Markov chain arising out of a discriminant GSN, i.e., where a GSN is used as a structured output model over a variable y, conditioned on an input $X$. Reproduced with permission from Zhou and Troyanskaya (2014). The structure is as in a GSN (over the output) but with computations being conditioned on the input $X$ at each step.

original GSN work), by simply adding (with a different weight) the supervised and unsupervised costs i.e., the reconstruction log-probabilities of $\mathbf{y}$ and $\mathbf{x}$ respectively. Such a hybrid criterion had previously been introduced for RBMs by Larochelle and Bengio (2008a). They show improved classification performance using this scheme.

## 20.13 Methodological Notes

Researchers studying generative models often need to compare one generative model to another, usually in order to demonstrate that a newly invented generative model is better at capturing some distribution than the pre-existing models.

This can be a difficult and subtle task. In many cases, we can not actually evaluate the log probability of the data under the model, but only an approximation. In these cases, it's important to think and communicate clearly about exactly what is being measured. For example, suppose we can evaluate a stochastic estimate of the log-likelihood for model A, and a deterministic lower bound on the log-likelihood for model B. If model A gets a higher score than model B, which is better? If we care about determining which model has a better internal representation of the distribution, we actually cannot tell, unless we have some way of determining how loose the bound for model B is. However, if we care about how well we can use the model in practice, for example to perform anomaly detection, then it is fair to say that model A is better based on a criterion specific to the practical task of interest, e.g., based on ranking test examples and ranking criterian such as precision and recall.

Another subtletly of evaluating generative models is that the evaluation metrics are often hard research problems in and of themselves. It can be very difficult to establish that models are being compared fairly. For example, suppose we use

AIS to estimate $\log Z$ in order to compute $\log \tilde{p}(\boldsymbol{x}) - \log Z$ for a new model we have just invented. A computationally economical implementation of AIS may fail to find several modes of the model distribution and underestimate $Z$, which will result in us overestimating $\log p(\boldsymbol{x})$. It can thus be difficult to tell whether a good likelihood estimate is due to a good model or a bad AIS implementation.

Other fields of machine learning usually allow for some variation in the preprocessing of the data. For example, when comparing the accuracy of object recognition algorithms, it is usually acceptable to preprocess the input images slightly differently for each algorithm based on what kind of input requirements it has. Generative modeling is different because changes in preprocessing, even very small and subtle ones, are completely unacceptable. Any change to the input data changes the distribution to be captured and fundamentally alters the task. For example, multiplying the input by 0.1 will artificially increase likelihood by 10.

Issues with preprocessing commonly arise when benchmarking generative models on the MNIST dataset, one of the more popular generative modeling benchmarks. MNIST consists of grayscale images. Some models treat MNIST images as points in a real vector space, while others treat them as binary. Yet others treat the grayscale values as probabilities for a binary samples. It is essential to compare real-valued models only to other real-valued models and binary-valued models only to other binary-valued models. Otherwise the likelihoods measured are not on the same space. (For the binary-valued models, the log-likelihood can be at most 0., while for real-valued models it can be arbitrarily high, since it is the measurement of a density) Among binary models, it is important to compare models using exactly the same kind of binarization. For example, we might binarize a gray pixel to 0 or 1 by thresholding at 0.5, or by drawing a random sample whose probability of being 1 is given by the gray pixel intensity. If we use the random binarization, we might binarize the whole dataset once, or we might draw a different random example for each step of training and then draw multiple samples for evaluation. Each of these three schemes yields wildly different likelihood numbers, and when comparing different models it is important that both models use the same binarization scheme for training and for evaluation. In fact, researchers who apply a single random binarization step share a file containing the results of the random binarization, so that there is no difference in results based on different outcomes of the binarization step.

Finally, in some cases the likelihood seems not to measure any attribute of the model that we really care about. For example, real-valued models of MNIST can obtain arbitrarily high likelihood by assigning arbitrarily low variance to background pixels that never change. Models and algorithms that detect these constant features can reap unlimited rewards, even though this is not a very useful thing to do. This strongly suggests a need for developing other ways of evaluating

generative models.

Although this is still an open question, this might be achieved by converting the problem into a classification task. For example, we have seen that the NCE method (Noise Contrastive Estimation, Section 18.6) compares the density of the training data according to a learned unnormalized model with its density under a background model. However, generative models do not always provide us with an energy function (equivalently, an unnormalized density), e.g., deep Boltzmann machines, generative stochastic networks, most denoising auto-encoders (that are not guaranteed to correspond to an energy function), deep Belief networks, etc. Therefore, it would be interesting to consider a classification task in which one tries to distinguish the training examples from the generated examples. This is precisely what is achieved by the discriminator network of generative adversarial networks (Section 20.9.5). However, it would require an expensive operation (training a discriminator) each time one would have to evaluate performance

**Algorithm 20.3** The variational stochastic maximum likelihood algorithm for training a 2 hidden-layer DBM.

Set $\epsilon$, the step size, to a small positive number
Set $k$, the number of Gibbs steps, high enough to allow a Markov chain of $p(\boldsymbol{v}, \boldsymbol{h} \quad , \boldsymbol{h} \quad ; \boldsymbol{\theta} + \epsilon \Delta \quad )$ toburn in, starting from samples from $p(\boldsymbol{v}, \boldsymbol{h} \quad , \boldsymbol{h} \quad ; \boldsymbol{\theta})$.
Initialize three random matrices, $\tilde{\boldsymbol{V}}$, $\tilde{\boldsymbol{H}} \quad$ and $\tilde{\boldsymbol{H}} \quad$ each with $m$ columns set to random values (e.g., from bernoulli distributions, possibly with marginals matched to the model's marginals).
**while** Not converged (learning loop) **do**
  Sample a minibatch of $m$ examples from the training data and arrange them as the columns of a matrix $\boldsymbol{V} = \quad \boldsymbol{v} \quad , \ldots , \boldsymbol{v} \quad ^{m}$ from the training set.
  **while** Not converged (Mean-field inference loop) **do**
    $\tilde{\boldsymbol{H}} \quad \leftarrow \text{sigmoid} \quad \boldsymbol{V} \quad \boldsymbol{W} \quad + \tilde{\boldsymbol{H}} \quad \quad \boldsymbol{W} \quad \quad .$
    $\tilde{\boldsymbol{H}} \quad \leftarrow \text{sigmoid} \quad \tilde{\boldsymbol{H}} \quad \boldsymbol{W} \quad .$
  **end while**
  $\Delta \quad \leftarrow \frac{}{m} \boldsymbol{V} \hat{\boldsymbol{H}}$
  $\Delta \quad \leftarrow \frac{}{m} \hat{\boldsymbol{H}} \quad \hat{\boldsymbol{H}}$
  **for** $l = 1$ to $k$ (Gibbs sampling) **do**
    Gibbs block 1:
    $\tilde{\boldsymbol{V}}$sampled from $\quad {}^{m}_{i} \quad {}^{d}_{a} \quad \text{sigmoid} \quad \boldsymbol{W}_{a,} \quad \tilde{\boldsymbol{H}} \quad _{,i} \quad .$
    $\tilde{\boldsymbol{H}} \quad$ sampled from $\quad {}^{m}_{i} \quad {}^{m}_{b} \quad \text{sigmoid} \quad \tilde{\boldsymbol{H}} \quad _{,i} \quad \boldsymbol{W} \quad _{,b} \quad .$
    Gibbs block 2:
    $\tilde{\boldsymbol{H}} \quad$ sampled from $\quad {}^{m}_{i} \quad {}^{n}_{j} \quad \text{sigmoid} \quad \tilde{\boldsymbol{V}}_{,i} \boldsymbol{W} \quad _{,j} \quad + \boldsymbol{W} \quad _{j,} \quad \tilde{\boldsymbol{H}} \quad _{,i} \quad .$
  **end for**
  $\Delta \quad \leftarrow \Delta \quad \quad - \frac{}{m} \boldsymbol{V} \hat{\boldsymbol{H}}$
  $\Delta \quad \leftarrow \Delta \quad \quad - \frac{}{m} \hat{\boldsymbol{H}} \quad \hat{\boldsymbol{H}}$
  $\boldsymbol{W} \quad \leftarrow \boldsymbol{W} \quad + \epsilon \Delta$
  $\boldsymbol{W} \quad \leftarrow \boldsymbol{W} \quad + \epsilon \Delta$
**end while**

# Bibliography


Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, **9**, 147–169. 582

Alain, G. and Bengio, Y. (2012). What regularized auto-encoders learn from the data generating distribution. Technical Report Arxiv report 1211.4246, Université de Montréal. 452

Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In *ICLR'2013*. also arXiv report 1211.4246. 435, 452, 454

Alain, G., Bengio, Y., Yao, L., Éric Thibodeau-Laufer, Yosinski, J., and Vincent, P. (2015). GSNs: Generative stochastic networks. arXiv:1503.05571. 439

Anderson, E. (1935). The Irises of the Gaspe Peninsula. *Bulletin of the American Iris Society*, **59**, 2–5. 19

Ba, J., Mnih, V., and Kavukcuoglu, K. (2014). Multiple object recognition with visual attention. *arXiv:1412.7755*. 192

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. Technical report, arXiv:1409.0473. 23, 94, 346, 361, 407, 416, 417

Bahl, L. R., Brown, P., de Souza, P. V., and Mercer, R. L. (1987). Speech recognition with continuous-parameter hidden Markov models. *Computer, Speech and Language*, **2**, 219–234. 65, 569

Baldi, P. and Brunak, S. (1998). *Bioinformatics, the Machine Learning Approach*. MIT Press. 571

Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, **2**, 53–58. 248

Baldi, P., Brunak, S., Frasconi, P., Soda, G., and Pollastri, G. (1999). Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, **15**(11), 937–946. 344

Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, **5**. 23

Barron, A. E. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Trans. on Information Theory*, **39**, 930–945. 193

Bartholomew, D. J. (1987). *Latent variable models and factor analysis*. Oxford University Press. 440

Basilevsky, A. (1994). *Statistical Factor Analysis and Related Methods: Theory and Applications*. Wiley. 440

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. 78, 187, 387

Basu, S. and Christensen, J. (2013). Teaching classification boundaries to humans. In *AAAI'2013*. 293

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Ann. Math. Stat.*, **37**, 1559–1563. 567

Baxter, J. (1995). Learning internal representations. In *Proceedings of the 8th International Conference on Computational Learning Theory (COLT'95)*, pages 311–320, Santa Cruz, California. ACM Press. 237

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2015). Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*. 185

Bayer, J. and Osendorfer, C. (2014). Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*. 234

Becker, S. and Hinton, G. (1992). A self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, **355**, 161–163. 485

Behnke, S. (2001). Learning iterative image reconstruction in the neural abstraction pyramid. *Int. J. Computational Intelligence and Applications*, **1**(4), 427–438. 449

Beiu, V., Quintana, J. M., and Avedillo, M. J. (2003). Vlsi implementations of threshold logic-a comprehensive survey. *Neural Networks, IEEE Transactions on*, **14**(5), 1217–1243. 391

Belkin, M. and Niyogi, P. (2002). Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS'01*, Cambridge, MA. MIT Press. 472

Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, **15**(6), 1373–1396. 153, 490

Bengio, S. and Bengio, Y. (2000a). Taking on the curse of dimensionality in joint distributions using neural networks. *IEEE Transactions on Neural Networks, special issue on Data Mining and Knowledge Discovery*, **11**(3), 550–557. 615

Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. Technical report, arXiv:1506.03099. 333

Bengio, Y. (1991). *Artificial Neural Networks and their Application to Sequence Recognition*. Ph.D. thesis, McGill University, (Computer Science), Montreal, Canada. 352, 571

Bengio, Y. (1993). A connectionist approach to speech recognition. *International Journal on Pattern Recognition and Artificial Intelligence*, **7**(4), 647–668. 569

Bengio, Y. (1999a). Markovian models for sequential data. *Neural Computing Surveys*, **2**, 129–162. 569

Bengio, Y. (1999b). Markovian models for sequential data. *Neural Computing Surveys*, **2**, 129–162. 571

Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural Computation*, **12**(8), 1889–1900. 378

Bengio, Y. (2002). New distributed probabilistic language models. Technical Report 1215, Dept. IRO, Université de Montréal. 409

Bengio, Y. (2009). *Learning deep architectures for AI*. Now Publishers. 149, 195

Bengio, Y. (2013a). Deep learning of representations: looking forward. In *Statistical Language and Speech Processing*, volume 7978 of *Lecture Notes in Computer Science*, pages 1–37. Springer, also in arXiv at http://arxiv.org/abs/1305.0445. 389

Bengio, Y. (2013b). Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal. 525

Bengio, Y. and Bengio, S. (2000b). Modeling high-dimensional discrete data with multi-layer neural networks. In *NIPS'99*, pages 400–406. MIT Press. 613, 614, 615, 617

Bengio, Y. and Delalleau, O. (2009). Justifying and generalizing contrastive divergence. *Neural Computation*, **21**(6), 1601–1621. 452, 539, 591

Bengio, Y. and Frasconi, P. (1996). Input/Output HMMs for sequence processing. *IEEE Transactions on Neural Networks*, **7**(5), 1231–1249. 571

Bengio, Y. and Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. In *NIPS'03*, Cambridge, MA. MIT Press, Cambridge. 114

Bengio, Y. and LeCun, Y. (2007a). Scaling learning algorithms towards AI. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*. MIT Press. 17, 197

Bengio, Y. and LeCun, Y. (2007b). Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. 149

Bengio, Y. and Monperrus, M. (2005). Non-local manifold tangent learning. In *NIPS'04*, pages 129–136. MIT Press. 151, 492, 493

Bengio, Y. and Sénécal, J.-S. (2003). Quick training of probabilistic neural nets by importance sampling. In *Proceedings of AISTATS 2003*. 412

Bengio, Y. and Sénécal, J.-S. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Trans. Neural Networks*, **19**(4), 713–722. 412

Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1991). Phonetically motivated acoustic parameters for continuous speech recognition using artificial neural networks. In *Proceedings of EuroSpeech'91*. 24, 398

Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1992a). Global optimization of a neural network-hidden Markov model hybrid. *IEEE Transactions on Neural Networks*, **3**(2), 252–259. 569, 571

Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1992b). Neural network - gaussian mixture hybrid for speech recognition or density estimation. In *NIPS 4*, pages 175–182. Morgan Kaufmann. 398

Bengio, Y., Frasconi, P., and Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1195, San Francisco. IEEE Press. (invited paper). 255, 361

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Tr. Neural Nets*. 255, 256, 349, 357, 361, 362

Bengio, Y., LeCun, Y., Nohl, C., and Burges, C. (1995). Lerec: A NN/HMM hybrid for on-line handwriting recognition. *Neural Computation*, **7**(6), 1289–1303. 571

Bengio, Y., Latendresse, S., and Dugas, C. (1999). Gradient-based learning of hyper-parameters. Learning Conference, Snowbird. 378

Bengio, Y., Ducharme, R., and Vincent, P. (2001a). A neural probabilistic language model. In *NIPS'00*, pages 932–938. MIT Press. 16, 388, 408

Bengio, Y., Ducharme, R., and Vincent, P. (2001b). A neural probabilistic language model. In *NIPS'2000*, pages 932–938. 401, 403, 414, 424

Bengio, Y., Ducharme, R., and Vincent, P. (2001c). A neural probabilistic language model. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *NIPS'2000*, pages 932–938. MIT Press. 494, 495

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003a). A neural probabilistic language model. *JMLR*, **3**, 1137–1155. 402, 408, 414

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003b). A neural probabilistic language model. *Journal of Machine Learning Research*, **3**, 1137–1155. 494, 495

Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., and Marcotte, P. (2006a). Convex neural networks. In *NIPS'2005*, pages 123–130. 232

Bengio, Y., Delalleau, O., and Le Roux, N. (2006b). The curse of highly variable functions for local kernel machines. In *NIPS'2005*. 149

Bengio, Y., Larochelle, H., and Vincent, P. (2006c). Non-local manifold Parzen windows. In *NIPS'2005*. MIT Press. 151, 492

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007a). Greedy layer-wise training of deep networks. In *NIPS'2006*. 12, 16, 459, 461

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007b). Greedy layer-wise training of deep networks. In *NIPS 19*, pages 153–160. MIT Press. 195, 288

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *ICML'09*. 292, 293

Bengio, Y., Léonard, N., and Courville, A. (2013a). Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv:1308.3432. 189, 192, 389, 525

Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013b). Generalized denoising auto-encoders as generative models. In *NIPS'2013*. 454, 618, 621

Bengio, Y., Courville, A., and Vincent, P. (2013c). Representation learning: A review and new perspectives. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, **35**(8), 1798–1828. 484, 611

Bengio, Y., Thibodeau-Laufer, E., Alain, G., and Yosinski, J. (2014a). Deep generative stochastic networks trainable by backprop. Technical Report arXiv:1306.1091. 526

Bengio, Y., Thibodeau-Laufer, E., Alain, G., and Yosinski, J. (2014b). Deep generative stochastic networks trainable by backprop. In *ICML'2014*. 526, 619, 620, 622, 623

Bennett, C. (1976). Efficient estimation of free energy differences from Monte Carlo data. *Journal of Computational Physics*, **22**(2), 245–268. 555

Berger, A. L., Della Pietra, V. J., and Della Pietra, S. A. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics*, **22**, 39–71. 415

Berglund, M. and Raiko, T. (2013). Stochastic gradient estimate variance in contrastive divergence and persistent contrastive divergence. *CoRR*, **abs/1312.6002**. 541

Bergstra, J. (2011). *Incorporating Complex Cells into Neural Networks for Pattern Classification*. Ph.D. thesis, Université de Montréal. 229, 434

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Machine Learning Res.*, **13**, 281–305. 376, 377

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010a). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. 78, 387

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010b). Theano: a CPU and GPU math expression compiler. In *Proc. SciPy*. 187, 188

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *NIPS'2011*. 378

Bertsekas, D. P. (2004). *Nonlinear programming*. Athena Scientific, 2 edition. 259

Bertsekas, D. P. and Tsitsiklis, J. (1996). *Neuro-Dynamic Programming*. Athena Scientific. 99

Besag, J. (1975). Statistical analysis of non-lattice data. *The Statistician*, **24**(3), 179–195. 543

Bishop, C. M. (1994). Mixture density networks. 172

Bishop, C. M. (1995a). Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN'95*, volume 1, page 141–148. 216, 225

Bishop, C. M. (1995b). Training with noise is equivalent to Tikhonov regularization. *Neural Computation*, **7**(1), 108–116. 216

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. 92, 139

Blum, A. L. and Rivest, R. L. (1992). Training a 3-node neural network is np-complete. 257

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1989). Learnability and the vapnik–chervonenkis dimension. *Journal of the ACM*, **36**(4), 929—865. 106

Bonnet, G. (1964). Transformations des signaux aléatoires à travers les systèmes non linéaires sans mémoire. *Annales des Télécommunications*, **19**(9–10), 203–220. 189, 525

Bordes, A., Weston, J., Collobert, R., and Bengio, Y. (2011). Learning structured embeddings of knowledge bases. In *AAAI 2011*. 426

Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2012). Joint learning of words and meaning representations for open-text semantic parsing. *AISTATS'2012*. 349, 426, 427

Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2013a). A semantic matching energy function for learning with multi-relational data. *Machine Learning: Special Issue on Learning Semantics*. 426

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. (2013b). Translating embeddings for modeling multi-relational data. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2787–2795. Curran Associates, Inc. 426

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, New York, NY, USA. ACM. 16, 135, 149, 165

Bottou, L. (1991). *Une approche théorique de l'apprentissage connexioniste; applications à la reconnaissance de la parole*. Ph.D. thesis, Université de Paris XI. 571

Bottou, L. (1998). Online algorithms and stochastic approximations. In D. Saad, editor, *Online Learning in Neural Networks*. Cambridge University Press, Cambridge, UK. 260

Bottou, L. (2011). From machine learning to machine reasoning. Technical report, arXiv.1102.1808. 348, 349

Bottou, L. (2015). Multilayer neural networks. Deep Learning Summer School. 382

Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In *NIPS'2008*. 245, 259, 261

Bottou, L., Fogelman-Soulié, F., Blanchet, P., and Lienard, J. S. (1990). Speaker independent isolated digit recognition: multilayer perceptrons vs dynamic time warping. *Neural Networks*, **3**, 453–465. 571

Bottou, L., Bengio, Y., and LeCun, Y. (1997). Global training of document processing systems using graph transformer networks. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'97)*, pages 490–494, Puerto Rico. IEEE. 562, 570, 572, 573, 574

Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *ICML'12*. 418

Boureau, Y., Ponce, J., and LeCun, Y. (2010). A theoretical analysis of feature pooling in vision algorithms. In *Proc. International Conference on Machine learning (ICML'10)*. 306

Boureau, Y., Le Roux, N., Bach, F., Ponce, J., and LeCun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. In *Proc. International Conference on Computer Vision (ICCV'11)*. IEEE. 306

Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, **59**, 291–294. 431

Bourlard, H. and Morgan, N. (1993). *Connectionist Speech Recognition. A Hybrid Approach*, volume 247 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Boston. 571

Bourlard, H. and Wellekens, C. (1989). Speech pattern discrimination and multi-layered perceptrons. *Computer Speech and Language*, **3**, 1–19. 398

Bourlard, H. and Wellekens, C. (1990). Links between hidden Markov models and multilayer perceptrons. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **12**, 1167–1178. 571

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA. 88

Brady, M. L., Raghavan, R., and Slawny, J. (1989). Back-propagation fails to separate where perceptrons succeed. *IEEE Transactions on Circuits and Systems*, **36**, 665–674. 247

Brand, M. (2003). Charting a manifold. In *NIPS'2002*, pages 961–968. MIT Press. 153, 490

Breiman, L. (1994). Bagging predictors. *Machine Learning*, **24**(2), 123–140. 230

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA. 138

Bridle, J. S. (1990). Alphanets: a recurrent 'neural' network architecture with a hidden Markov model interpretation. *Speech Communication*, **9**(1), 83–92. 168

Briggman, K., Denk, W., Seung, S., Helmstaedter, M. N., and Turaga, S. C. (2009). Maximin affinity learning of image segmentation. In *NIPS'2009*, pages 1865–1873. 315

Brown, P. (1987). *The Acoustic-Modeling problem in Automatic Speech Recognition*. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University. 569

Brown, P. F., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, **16**(2), 79–85. 19

Brown, P. F., Pietra, V. J. D., DeSouza, P. V., Lai, J. C., and Mercer, R. L. (1992). Class-based $n$-gram models of natural language. *Computational Linguistics*, **18**, 467–479. 405

Bryson, A. and Ho, Y. (1969). *Applied optimal control: optimization, estimation, and control*. Blaisdell Pub. Co. 200

Bryson, Jr., A. E. and Denham, W. F. (1961). A steepest-ascent method for solving optimum programming problems. Technical Report BR-1303, Raytheon Company, Missle and Space Division. 200

Buchberger, B., Collins, G. E., Loos, R., and Albrecht, R. (1983). *Computer Algebra*. Springer-Verlag. 188

Buciluǎ, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM. 388

Cai, M., Shi, Y., and Liu, J. (2013). Deep maxout neural networks for speech recognition. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 291–296. IEEE. 198

Carreira-Perpiñan, M. A. and Hinton, G. E. (2005). On contrastive divergence learning. In R. G. Cowell and Z. Ghahramani, editors, *AISTATS'2005*, pages 33–40. Society for Artificial Intelligence and Statistics. 539, 591

Caruana, R. (1993). Multitask connectionist learning. In *Proc. 1993 Connectionist Models Summer School*, pages 372–379. 235

Cauchy, A. (1847a). Méthode générale pour la résolution de systèmes d'équations simultanées. In *Compte rendu des séances de l'académie des sciences*, pages 536–538. 79

Cauchy, L. A. (1847b). Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu à l'Académie des Sciences*. 200

Cayton, L. (2005). Algorithms for manifold learning. Technical Report CS2008-0923, UCSD. 153, 487

Challis, E. and Barber, D. (2012). Affine independent variational inference. In *NIPS'2012*. 526

Chapelle, O., Weston, J., and Schölkopf, B. (2003). Cluster kernels for semi-supervised learning. In *NIPS'02*, pages 585–592, Cambridge, MA. MIT Press. 472

Chapelle, O., Schölkopf, B., and Zien, A., editors (2006). *Semi-Supervised Learning*. MIT Press, Cambridge, MA. 472

Chellapilla, K., Puri, S., and Simard, P. (2006). High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France). Université de Rennes 1, Suvisoft. http://www.suvisoft.com. 21, 24, 386

Chen, S. F. and Goodman, J. T. (1999). An empirical study of smoothing techniques for language modeling. *Computer, Speech and Language*, **13**(4), 359–393. 402, 404, 415

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014a). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM. 391

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., *et al.* (2014b). Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 609–622. IEEE. 391

Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. (2014). Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 388

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014a). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*. 340, 345, 357, 416

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014b). On the properties of neural machine translation: Encoder-decoder approaches. *ArXiv e-prints*, **abs/1409.1259**. 357

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks. 247, 249, 464

Chorowski, J., Bahdanau, D., Cho, K., and Bengio, Y. (2014). End-to-end continuous speech recognition using attention-based recurrent nn: First results. arXiv:1412.1602. 399

Chrupala, G., Kadar, A., and Alishahi, A. (2015). Learning language through pictures. arXiv 1506.03694. 357

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. NIPS'2014 Deep Learning workshop, arXiv 1412.3555. 357, 399

Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2015). Gated feedback recurrent neural networks. In *ICML'15*. 357

Ciresan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012). Multi-column deep neural network for traffic sign classification. *Neural Networks*, **32**, 333–338. 22, 195

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep big simple neural nets for handwritten digit recognition. *Neural Computation*, **22**, 1–14. 21, 24, 386

Coates, A. and Ng, A. Y. (2011). The importance of encoding versus training with sparse coding and vector quantization. In *ICML'2011*. 24

Coates, A., Lee, H., and Ng, A. Y. (2011). An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*. 395

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28 (3), pages 1337–1345. JMLR Workshop and Conference Proceedings. 21, 24, 319, 388

Collobert, R. (2004). *Large Scale Machine Learning*. Ph.D. thesis, Université de Paris VI, LIP6. 165

Collobert, R. (2011). Deep learning for efficient discriminative parsing. In *AISTATS'2011*. 95

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML'2008*. 413

Collobert, R., Bengio, S., and Bengio, Y. (2001). A parallel mixture of SVMs for very large scale problems. Technical Report IDIAP-RR-01-12, IDIAP. 390

Collobert, R., Bengio, S., and Bengio, Y. (2002). Parallel mixture of SVMs for very large scale problems. *Neural Computation*, **14**(5), 1105–1114. 390

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011a). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, **12**, 2493–2537. 293

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011b). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*. 387

Comon, P. (1994). Independent component analysis - a new concept? *Signal Processing*, **36**, 287–314. 441, 442

Cortes, C. and Vapnik, V. (1995). Support vector networks. *Machine Learning*, **20**, 273–297. 16, 135, 149

Couprie, C., Farabet, C., Najman, L., and LeCun, Y. (2013). Indoor semantic segmentation using depth information. In *International Conference on Learning Representations (ICLR2013)*. 22, 195

Courbariaux, M., Bengio, Y., and David, J.-P. (2015). Low precision arithmetic for deep learning. In *Arxiv:1412.7024, ICLR'2015 Workshop*. 392

Courville, A., Bergstra, J., and Bengio, Y. (2011). Unsupervised models of images by spike-and-slab RBMs. In *ICML'11*. 506, 608

Courville, A., Desjardins, G., Bergstra, J., and Bengio, Y. (2014). The spike-and-slab RBM and extensions to discrete and sparse data distributions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **36**(9), 1874–1887. 609

Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory, 2nd Edition*. Wiley-Interscience. 56

Cox, D. and Pinto, N. (2011). Beyond simple features: A large-scale feature search approach to unconstrained face recognition. In *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*, pages 8–15. IEEE. 318

Cramér, H. (1946). *Mathematical methods of statistics*. Princeton University Press. 127, 261

Crick, F. H. C. and Mitchison, G. (1983). The function of dream sleep. *Nature*, **304**, 111–114. 537

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, **2**, 303–314. 192, 481

Dahl, G. E., Ranzato, M., Mohamed, A., and Hinton, G. E. (2010). Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS'2010*. 22

Dahl, G. E., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, **20**(1), 33–42. 398

Dahl, G. E., Jaitly, N., and Salakhutdinov, R. (2014). Multi-task neural networks for QSAR predictions. arXiv:1406.1231. 23

Dauphin, Y. and Bengio, Y. (2013). Stochastic ratio matching of RBMs for sparse high-dimensional inputs. In *NIPS26*. NIPS Foundation. 546

Dauphin, Y., Glorot, X., and Bengio, Y. (2011). Large-scale learning of embeddings with reconstruction sampling. In *ICML'2011*. 412

Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS'2014*. 247, 248, 249, 461, 464

Davis, A., Rubinstein, M., Wadhwa, N., Mysore, G., Durand, F., and Freeman, W. T. (2014). The visual microphone: Passive recovery of sound from video. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, **33**(4), 79:1–79:10. 392

Dayan, P. (1990). Reinforcement comparison. In *Connectionist Models: Proceedings of the 1990 Connectionist Summer School*, San Mateo, CA. 192

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS'2012*. 388

Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, **5**(3), 142–150. 562

Delalleau, O. and Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *NIPS*. 17, 194, 481, 482

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*. 19, 143

Deng, J., Berg, A. C., Li, K., and Fei-Fei, L. (2010a). What does classifying more than 10,000 image categories tell us? In *Proceedings of the 11th European Conference on Computer Vision: Part V*, ECCV'10, pages 71–84, Berlin, Heidelberg. Springer-Verlag. 19

Deng, J., Ding, N., Jia, Y., Frome, A., Murphy, K., Bengio, S., Li, Y., Neven, H., and Adam, H. (2014). Large-scale object classification using label relation graphs. In *ECCV'2014*, pages 48–64. 562

Deng, L. and Yu, D. (2014). Deep learning – methods and applications. *Foundations and Trends in Signal Processing*. 399

Deng, L., Seltzer, M., Yu, D., Acero, A., Mohamed, A., and Hinton, G. (2010b). Binary coding of speech spectrograms using a deep auto-encoder. In *Interspeech 2010*, Makuhari, Chiba, Japan. 22

Desjardins, G. and Bengio, Y. (2008). Empirical evaluation of convolutional RBMs for vision. Technical Report 1327, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal. 609

Desjardins, G., Courville, A., and Bengio, Y. (2011). On tracking the partition function. In *NIPS'2011*. 555

Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *Proc. ACL'2014*. 415

DiCarlo, J. J. (2013). Mechanisms underlying visual object recognition: Humans vs. neurons vs. machines. NIPS Tutorial. 23, 321

Do, T.-M.-T. and Artières, T. (2010). Neural conditional random fields. In *International Conference on Artificial Intelligence and Statistics*, pages 177–184. 562

Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., and Darrell, T. (2014). Long-term recurrent convolutional networks for visual recognition and description. arXiv:1411.4389. 95

Donoho, D. L. and Grimes, C. (2003). Hessian eigenmaps: new locally linear embedding techniques for high-dimensional data. Technical Report 2003-08, Dept. Statistics, Stanford University. 153, 490

Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, **1**, 75–80. 256, 349

Dreyfus, S. E. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, **5(1)**, 30–45. 200

Dreyfus, S. E. (1973). The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, **18(4)**, 383–385. 200

Dudik, M., Langford, J., and Li, L. (2011). Doubly robust policy evaluation and learning. In *Proceedings of the 28th International Conference on Machine learning*, ICML '11. 424

Dugas, C., Bengio, Y., Bélisle, F., and Nadeau, C. (2001). Incorporating second-order functional knowledge for better option pricing. In *NIPS'00*, pages 472–478. MIT Press. 65, 165

El Hihi, S. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS 8*. MIT Press. 347, 365, 366

ElHihi, S. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS'1995*. 353

Elkahky, A. M., Song, Y., and He, X. (2015). A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*, pages 278–288. 423

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, **48**, 781–799. 293

Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Proceedings of AISTATS'2009*. 195

Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *J. Machine Learning Res.* 460, 461, 462, 463

Fang, H., Gupta, S., Iandola, F., Srivastava, R., Deng, L., Dollár, P., Gao, J., He, X., Mitchell, M., Platt, J. C., Zitnick, C. L., and Zweig, G. (2015). From captions to visual concepts and back. arXiv:1411.4952. 95

Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., and Talay, S. (2011). Large-scale FPGA-based convolutional networks. In R. Bekkerman, M. Bilenko, and J. Langford, editors, *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press. 448

Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013a). Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 22, 195

Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013b). Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**(8), 1915–1929. 315, 562

Fei-Fei, L., Fergus, R., and Perona, P. (2006). One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **28**(4), 594–611. 468

Fischer, A. and Igel, C. (2011). Bounding the bias of contrastive divergence learning. *Neural Computation*, **23**(3), 664–73. 591

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, 179–188. 19, 98

Frasconi, P., Gori, M., and Sperduti, A. (1997). On the efficient classification of data structures by neural networks. In *Proc. Int. Joint Conf. on Artificial Intelligence*. 348, 349

Frasconi, P., Gori, M., and Sperduti, A. (1998). A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, **9**(5), 768–786. 348, 349

Freund, Y. and Schapire, R. E. (1996a). Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of Thirteenth International Conference*, pages 148–156, USA. ACM. 232

Freund, Y. and Schapire, R. E. (1996b). Game theory, on-line prediction and boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, pages 325–332. 232

Frey, B. J. (1998). *Graphical models for machine learning and digital communication*. MIT Press. 613, 614

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, **20**, 121–136. 459

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, **36**, 193–202. 14, 21, 24, 322

Gal, Y. and Ghahramani, Z. (2015). Bayesian convolutional neural networks with bernoulli approximate variational inference. *arXiv preprint arXiv:1506.02158*. 234

Gallinari, P., LeCun, Y., Thiria, S., and Fogelman-Soulie, F. (1987). Memoires associatives distribuees. In *Proceedings of COGNITIVA 87*, Paris, La Villette. 449

Garcia-Duran, A., Bordes, A., Usunier, N., and Grandvalet, Y. (2015). Combining two and three-way embeddings models for link prediction in knowledge bases. *arXiv preprint arXiv:1506.00999*. 426

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., and Pallett, D. S. (1993). Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon Technical Report N*, **93**, 27403. 398

Garson, J. (1900). The metric system of identification of criminals, as used in in great britain and ireland. *The Journal of the Anthropological Institute of Great Britain and Ireland*, (2), 177–227. 19

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural computation*, **12**(10), 2451–2471. 358

Glorot, X. and Bengio, Y. (2010a). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*. 164, 382

Glorot, X. and Bengio, Y. (2010b). Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256. 284

Glorot, X., Bordes, A., and Bengio, Y. (2011a). Deep sparse rectifier neural networks. In *AISTATS'2011*. 14, 165, 447

Glorot, X., Bordes, A., and Bengio, Y. (2011b). Deep sparse rectifier neural networks. In *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*. 198

Glorot, X., Bordes, A., and Bengio, Y. (2011c). Domain adaptation for large-scale sentiment classification: A deep learning approach. In *ICML'2011*. 447, 467

Gong, S., McKenna, S., and Psarrou, A. (2000). *Dynamic Vision: From Images to Face Recognition*. Imperial College Press. 491, 493

Goodfellow, I., Le, Q., Saxe, A., and Ng, A. (2009). Measuring invariances in deep networks. In *NIPS'2009*, pages 646–654. 229, 434, 447

Goodfellow, I., Koenig, N., Muja, M., Pantofaru, C., Sorokin, A., and Takayama, L. (2010). Help me help you: Interfaces for personal robots. In *Proc. of Human Robot Interaction (HRI)*, Osaka, Japan. ACM Press, ACM Press. 93

Goodfellow, I., Courville, A., and Bengio, Y. (2012). Large-scale feature learning with spike-and-slab sparse coding. In *ICML'2012*. 443

Goodfellow, I. J. (2010). Technical report: Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal. 313

Goodfellow, I. J., Courville, A., and Bengio, Y. (2011). Spike-and-slab sparse coding for unsupervised feature discovery. In *NIPS Workshop on Challenges in Learning Hierarchical Models*. 195, 467

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013a). Maxout networks. In S. Dasgupta and D. McAllester, editors, *ICML'13*, pages 1319–1327. 198, 234, 320, 395

Goodfellow, I. J., Mirza, M., Courville, A., and Bengio, Y. (2013b). Multi-prediction deep Boltzmann machines. In *NIPS26*. NIPS Foundation. 94, 545, 605, 607

Goodfellow, I. J., Courville, A., and Bengio, Y. (2013c). Scaling up spike-and-slab models for unsupervised feature learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**(8), 1902–1914. 609

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2014a). An empirical investigation of catastrophic forgeting in gradient-based neural networks. In *ICLR'2014*. 199

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014b). Explaining and harnessing adversarial examples. *CoRR*, **abs/1412.6572**. 238

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014c). Generative adversarial networks. In *NIPS'2014*. 189

Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014d). Multi-digit number recognition from Street View imagery using deep convolutional neural networks. In *International Conference on Learning Representations*. 22, 94, 195, 340, 390

Goodfellow, I. J., Vinyals, O., and Saxe, A. M. (2015). Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*. 247, 249

Goodman, J. (2001). Classes for fast maximum entropy training. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Utah. 409

Gori, M. and Tesi, A. (1992). On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-14**(1), 76–86. 247

Gosset, W. S. (1908). The probable error of a mean. *Biometrika*, **6**(1), 1–25. Originally published under the pseudonym "Student". 19

Gouws, S., Bengio, Y., and Corrado, G. (2014). Bilbowa: Fast bilingual distributed representations without word alignments. Technical report, arXiv:1410.2455. 417, 470

Graf, H. P. and Jackel, L. D. (1989). Analog electronic neural network circuits. *Circuits and Devices Magazine, IEEE*, **5**(4), 44–49. 391

Graves, A. (2011a). Practical variational inference for neural networks. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc. 216

Graves, A. (2011b). Practical variational inference for neural networks. In *NIPS'2011*. 218

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Springer. 328, 344, 356, 357, 399, 562

Graves, A. (2013). Generating sequences with recurrent neural networks. Technical report, arXiv:1308.0850. 173, 356, 361, 364

Graves, A. and Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In *ICML'2014*. 356

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, **18**(5), 602–610. 344

Graves, A. and Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS'2008*, pages 545–552. 344

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *ICML'2006*, pages 369–376, Pittsburgh, USA. 399, 562

Graves, A., Liwicki, M., Bunke, H., Schmidhuber, J., and Fernández, S. (2008). Unconstrained on-line handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *NIPS'2007*, pages 577–584. 344

Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2009). A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **31**(5), 855–868. 356

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *ICASSP'2013*, pages 6645–6649. 344, 346, 347, 356, 357, 399

Graves, A., Wayne, G., and Danihelka, I. (2014a). Neural Turing machines. arXiv:1410.5401. 23

Graves, A., Wayne, G., and Danihelka, I. (2014b). Neural turing machines. *arXiv preprint arXiv:1410.5401*. 360

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). LSTM: a search space odyssey. *arXiv preprint arXiv:1503.04069*. 358

Gregor, K. and LeCun, Y. (2010). Emergence of complex-like cells in a temporal product network with local receptive fields. Technical report, arXiv:1006.0448. 311

Gülçehre, Ç. and Bengio, Y. (2013). Knowledge matters: Importance of prior information for optimization. In *International Conference on Learning Representations (ICLR'2013)*. 22

Guo, H. and Gelfand, S. B. (1992). Classification trees with neural network feature extraction. *Neural Networks, IEEE Transactions on*, **3**(6), 923–933. 390

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. *CoRR*, **abs/1502.02551**. 392

Gutmann, M. and Hyvarinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*. 547

Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Han, J., Muller, U., and LeCun, Y. (2007). Online learning for offroad robots: Spatial label propagation to learn long-range traversability. In *Proceedings of Robotics: Science and Systems*, Atlanta, GA, USA. 393

Haffner, P., Franzini, M., and Waibel, A. (1991). Integrating time alignment and neural networks for high performance continuous speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 105–108, Toronto. 571

Hajnal, A., Maass, W., Pudlak, P., Szegedy, M., and Turan, G. (1993). Threshold circuits of bounded depth. *J. Comput. System. Sci.*, **46**, 129–154. 194

Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California. ACM Press. 194, 481

Håstad, J. and Goldmann, M. (1991). On the power of small-depth threshold circuits. *Computational Complexity*, **1**, 113–129. 194, 481

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The elements of statistical learning: data mining, inference and prediction*. Springer Series in Statistics. Springer Verlag. 139

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *arXiv preprint arXiv:1502.01852*. 164, 199

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York. 15

Henaff, M., Jarrett, K., Kavukcuoglu, K., and LeCun, Y. (2011). Unsupervised learning of sparse features for scalable audio classification. In *ISMIR'11*. 448

Herault, J. and Ans, B. (1984). Circuits neuronaux à synapses modifiables: Décodage de messages composites par apprentissage non supervisé. *Comptes Rendus de l'Académie des Sciences*, **299(III-13)**, 525—528. 441

Hinton, G. (2012). Neural networks for machine learning. Coursera, video lectures. 265

Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012a). Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, **29**(6), 82–97. 22, 399

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*. 389

Hinton, G. E. (2000). Training products of experts by minimizing contrastive divergence. Technical Report GCNU TR 2000-004, Gatsby Unit, University College London. 538

Hinton, G. E. and Roweis, S. (2003). Stochastic neighbor embedding. In *NIPS'2002*. 490

Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, **313**, 504–507. 419, 461

Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786), 504–507. 437, 459, 460

Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In *NIPS'1993*. 431

Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, **18**, 1527–1554. 12, 16, 24, 136, 459, 460, 461, 592

Hinton, G. E., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012b). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, **29**(6), 82–97. 94

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012c). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580. 213

Hinton, G. E., Vinyals, O., and Dean, J. (2014). Dark knowledge. Invited talk at the BayLearn Bay Area Machine Learning Symposium. 389

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, T.U. Münich. 255, 349, 361

Hochreiter, S. and Schmidhuber, J. (1995). Simplifying neural nets by discovering flat minima. In *Advances in Neural Information Processing Systems 7*, pages 529–536. MIT Press. 220

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, **9**(8), 1735–1780. 23, 356, 357

Hochreiter, S., Informatik, F. F., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2000). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In J. Kolen and S. Kremer, editors, *Field Guide to Dynamical Recurrent Networks*. IEEE Press. 357

Holi, J. L. and Hwang, J.-N. (1993). Finite precision error analysis of neural network hardware implementations. *Computers, IEEE Transactions on*, **42**(3), 281–290. 391

Holt, J. L. and Baker, T. E. (1991). Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 121–126. IEEE. 391

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, **2**, 359–366. 192, 481

Hornik, K., Stinchcombe, M., and White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, **3**(5), 551–560. 193

Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. 2

Huang, F. and Ogata, Y. (2002). Generalized pseudo-likelihood estimates for markov random fields on lattice. *Annals of the Institute of Statistical Mathematics*, **54**(1), 1–18. 544

Huang, P.-S., He, X., Gao, J., Deng, L., Acero, A., and Heck, L. (2013). Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2333–2338. ACM. 423

Hubel, D. and Wiesel, T. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, **195**, 215–243. 319

Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, **148**, 574–591. 319

Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, **160**, 106–154. 319

Hutter, F., Hoos, H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *LION-5*. Extended version as UBC Tech report TR-2010-10. 378

Hyotyniemi, H. (1996). Turing machines are recurrent neural networks. In *STeP'96*, pages 13–24. 330

Hyvärinen, A. (1999). Survey on independent component analysis. *Neural Computing Surveys*, **2**, 94–128. 441

Hyvärinen, A. (2005a). Estimation of non-normalized statistical models using score matching. *J. Machine Learning Res.*, **6**. 452

Hyvärinen, A. (2005b). Estimation of non-normalized statistical models using score matching. *Journal of Machine Learning Research*, **6**, 695–709. 545

Hyvärinen, A. (2007a). Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables. *IEEE Transactions on Neural Networks*, **18**, 1529–1531. 546

Hyvärinen, A. (2007b). Some extensions of score matching. *Computational Statistics and Data Analysis*, **51**, 2499–2512. 546

Hyvärinen, A. and Pajunen, P. (1999). Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, **12**(3), 429–439. 442

Hyvärinen, A., Karhunen, J., and Oja, E. (2001). *Independent Component Analysis*. Wiley-Interscience. 441

Hyvärinen, A., Hurri, J., and Hoyer, P. O. (2009). *Natural Image Statistics: A probabilistic approach to early computational vision*. Springer-Verlag. 325

Inayoshi, H. and Kurita, T. (2005). Improved generalization by adding both auto-association and hidden-layer noise to neural-network-based-classifiers. *Machine Learning for Signal Processing*, pages 141—-146. 449

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. 22, 93, 369

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, **1**(4), 295–307. 264

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixture of local experts. *Neural Computation*, **3**, 79–87. 172, 390

Jaeger, H. (2003). Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems 15*. 349, 350, 351

Jaeger, H. (2007a). Discovering multiscale dynamical features with hierarchical echo state networks. Technical report, Jacobs University. 347

Jaeger, H. (2007b). Echo state network. *Scholarpedia*, **2**(9), 2330. 349, 350

Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, **304**(5667), 78–80. 24, 349, 350

Jaeger, H., Lukosevicius, M., Popovici, D., and Siewert, U. (2007). Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, **20**(3), 335–352. 354

Jain, V., Murray, J. F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K. L., Helmstaedter, M. N., Denk, W., and Seung, H. S. (2007). Supervised learning of image restoration with convolutional networks. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE. 315

Janzing, D., Peters, J., Sgouritsa, E., Zhang, K., Mooij, J. M., and Schölkopf, B. (2012). On causal and anticausal learning. In *ICML'2012*, pages 1255–1262. 473, 475

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009a). What is the best multi-stage architecture for object recognition? In *ICCV'09*. 14, 165, 448

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009b). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE. 21, 24, 197, 318, 319

Jarzynski, C. (1997). Nonequilibrium equality for free energy differences. *Phys. Rev. Lett.*, **78**, 2690–2693. 554

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. 48

Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2014). On using very large target vocabulary for neural machine translation. arXiv:1412.2007. 340, 416

Jelinek, F. and Mercer, R. L. (1980). Interpolated estimation of markov source parameters from sparse data. In E. S. Gelsema and L. N. Kanal, editors, *Pattern Recognition in Practice*. North-Holland, Amsterdam. 402, 415

Jia, Y., Huang, C., and Darrell, T. (2012). Beyond spatial pyramids: Receptive field learning for pooled image features. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3370–3377. IEEE. 306

Jim, K.-C., Giles, C. L., and Horne, B. G. (1996). An analysis of noise in recurrent neural networks: convergence and generalization. *IEEE Transactions on Neural Networks*, **7**(6), 1424–1438. 216, 218

Jordan, M. I. (1998). *Learning in Graphical Models*. Kluwer, Dordrecht, Netherlands. 16

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015a). An empirical evaluation of recurrent network architectures. In *ICML'2015*. 358

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015b). An empirical exploration of recurrent network architectures. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2342–2350. 287, 357

Juang, B. H. and Katagiri, S. (1992). Discriminative learning for minimum error classification. *IEEE Transactions on Signal Processing*, **40**(12), 3043–3054. 569

Judd, J. S. (1989). *Neural Network Design and the Complexity of Learning*. MIT press. 257

Jutten, C. and Herault, J. (1991). Blind separation of sources, part I: an adaptive algorithm based on neuromimetic architecture. *Signal Processing*, **24**, 1–10. 441

Kahou, S. E., Pal, C., Bouthillier, X., Froumenty, P., Gülçehre, c., Memisevic, R., Vincent, P., Courville, A., Bengio, Y., Ferrari, R. C., Mirza, M., Jean, S., Carrier, P.-L., Dauphin, Y., Boulanger-Lewandowski, N., Aggarwal, A., Zumer, J., Lamblin, P., Raymond, J.-P., Desjardins, G., Pascanu, R., Warde-Farley, D., Torabi, A., Sharma, A., Bengio, E., Côté, M., Konda, K. R., and Wu, Z. (2013). Combining modality specific deep neural networks for emotion recognition in video. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*. 195

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *EMNLP'2013*. 340, 416

Kamyshanska, H. and Memisevic, R. (2015). The potential energy of an autoencoder. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 454

Kanazawa, K., Koller, D., and Russell, S. (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *Proc. UAI'1995*, pages 346–351. 562

Karpathy, A. and Li, F.-F. (2015). Deep visual-semantic alignments for generating image descriptions. In *CVPR'2015*. arXiv:1412.2306. 95

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In *CVPR*. 19

Karush, W. (1939). *Minima of Functions of Several Variables with Inequalities as Side Constraints*. Master's thesis, Dept. ¯of Mathematics, Univ. ¯of Chicago. 90

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **ASSP-35**(3), 400–401. 402, 415

Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2008a). Fast inference in sparse coding algorithms with applications to object recognition. CBLL-TR-2008-12-01, NYU. 434

Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2008b). Fast inference in sparse coding algorithms with applications to object recognition. Technical report, Computational and Biological Learning Lab, Courant Institute, NYU. Tech Report CBLL-TR-2008-12-01. 448

Kavukcuoglu, K., Ranzato, M.-A., Fergus, R., and LeCun, Y. (2009). Learning invariant features through topographic filter maps. In *CVPR'2009*. 448

Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., and LeCun, Y. (2010a). Learning convolutional feature hierarchies for visual recognition. In *Advances in Neural Information Processing Systems 23 (NIPS'10)*, pages 1090–1098. 319

Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., and LeCun, Y. (2010b). Learning convolutional feature hierarchies for visual recognition. In *NIPS'2010*. 448

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *ARS Journal*, **30**(10), 947–954. 200

Khan, F., Zhu, X., and Mutlu, B. (2011). How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems 24 (NIPS'11)*, pages 1449–1457. 293

Kim, S. K., McAfee, L. C., McMahon, P. L., and Olukotun, K. (2009). A highly scalable restricted Boltzmann machine FPGA implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372. IEEE. 391

Kindermann, R. (1980). *Markov Random Fields and Their Applications (Contemporary Mathematics ; V. 1)*. American Mathematical Society. 510

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 267

Kingma, D. and LeCun, Y. (2010a). Regularized estimation of image statistics by score matching. In *NIPS'2010*. 452

Kingma, D. and LeCun, Y. (2010b). Regularized estimation of image statistics by score matching. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1126–1134. 547

Kingma, D., Rezende, D., Mohamed, S., and Welling, M. (2014). Semi-supervised learning with deep generative models. In *NIPS'2014*. 370, 526

Kingma, D. P. (2013). Fast gradient-based inference with continuous latent variable models in auxiliary form. Technical report, arxiv:1306.0733. 189, 525

Kingma, D. P. and Welling, M. (2014a). Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 189, 493, 494, 526

Kingma, D. P. and Welling, M. (2014b). Efficient gradient-based inference through transformations between bayes nets and neural nets. Technical report, arxiv:1402.0480. 189, 525, 526

Kirkpatrick, S., Jr., C. D. G., , and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, **220**, 671–680. 291

Kiros, R., Salakhutdinov, R., and Zemel, R. (2014a). Multimodal neural language models. In *ICML'2014*. 95

Kiros, R., Salakhutdinov, R., and Zemel, R. (2014b). Unifying visual-semantic embeddings with multimodal neural language models. *arXiv:*`1411.2539 [cs.LG]`. 95, 356

Klementiev, A., Titov, I., and Bhattarai, B. (2012). Inducing crosslingual distributed representations of words. In *Proceedings of COLING 2012*. 417, 470

Knowles-Barley, S., Jones, T. R., Morgan, J., Lee, D., Kasthuri, N., Lichtman, J. W., and Pfister, H. (2014). Deep learning for the connectome. *GPU Technology Conference*. 23

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press. 523, 529, 567

Konig, Y., Bourlard, H., and Morgan, N. (1996). REMAP: Recursive estimation and maximization of A posteriori probabilities – application to transition-based connectionist speech recognition. In *NIPS'95*. MIT Press, Cambridge, MA. 398

Koren, Y. (2009). 1 the bellkor solution to the netflix grand prize. 232, 422

Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork RNN. In *ICML'2014*. 347, 366

Kočiský, T., Hermann, K. M., and Blunsom, P. (2014). Learning Bilingual Word Representations by Marginalizing Alignments. In *Proceedings of ACL*. 417

Krause, O., Fischer, A., Glasmachers, T., and Igel, C. (2013). Approximation properties of DBNs with binary hidden units and real-valued visible units. In *ICML'2013*. 481

Krizhevsky, A. (2010). Convolutional deep belief networks on CIFAR-10. Technical report, University of Toronto. Unpublished Manuscript: http://www.cs.utoronto.ca/ kriz/conv-cifar10-aug2010.pdf. 387

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto. 19, 506

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*. 21, 24, 93, 326, 393, 397

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012b). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*. 22, 195, 447

Krueger, K. A. and Dayan, P. (2009). Flexible shaping: how learning in small steps helps. *Cognition*, **110**, 380–394. 293

Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif. University of California Press. 90

Kumar, A., Irsoy, O., Su, J., Bradbury, J., English, R., Pierce, B., Ondruska, P., Iyyer, M., Gulrajani, I., and Socher, R. (2015). Ask me anything: Dynamic memory networks for natural language processing. *arXiv:1506.07285*. 427

Kumar, M. P., Packer, B., and Koller, D. (2010). Self-paced learning for latent variable models. In *NIPS'2010*. 293

Lafferty, J., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In C. E. Brodley and A. P. Danyluk, editors, *ICML 2001*. Morgan Kaufmann. 563, 570

Lang, K. J. and Hinton, G. E. (1988). The development of the time-delay neural network architecture for speech recognition. Technical Report CMU-CS-88-152, Carnegie-Mellon University. 322, 327, 352

Langford, J. and Zhang, T. (2008). The epoch-greedy algorithm for contextual multi-armed bandits. In *NIPS'2008*, pages 1096—1103. 423

Lappalainen, H., Giannakopoulos, X., Honkela, A., and Karhunen, J. (2000). Nonlinear independent component analysis using ensemble learning: Experiments and discussion. In *Proc. ICA*. Citeseer. 442

Larochelle, H. and Bengio, Y. (2008a). Classification using discriminative restricted Boltzmann machines. In *ICML'2008*. 229, 434, 624

Larochelle, H. and Bengio, Y. (2008b). Classification using discriminative restricted Boltzmann machines. In *ICML'08*, pages 536–543. ACM. 472

Larochelle, H. and Murray, I. (2011). The Neural Autoregressive Distribution Estimator. In *AISTATS'2011*. 613, 615, 616

Larochelle, H., Erhan, D., and Bengio, Y. (2008). Zero-data learning of new tasks. In *AAAI Conference on Artificial Intelligence*. 468

Lasserre, J. A., Bishop, C. M., and Minka, T. P. (2006). Principled hybrids of generative and discriminative models. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'06)*, pages 87–94, Washington, DC, USA. IEEE Computer Society. 227, 472

Le, Q., Ngiam, J., Chen, Z., hao Chia, D. J., Koh, P. W., and Ng, A. (2010). Tiled convolutional neural networks. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23 (NIPS'10)*, pages 1279–1287. 311

Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. (2011). On optimization methods for deep learning. In *Proc. ICML'2011*. ACM. 276

Le, Q., Ranzato, M., Monga, R., Devin, M., Corrado, G., Chen, K., Dean, J., and Ng, A. (2012). Building high-level features using large scale unsupervised learning. In *ICML'2012*. 21, 24

Le Roux, N. and Bengio, Y. (2010). Deep belief networks are compact universal approximators. *Neural Computation*, **22**(8), 2192–2207. 481

LeCun, Y. (1985). Une procédure d'apprentissage pour Réseau à seuil assymétrique. In *Cognitiva 85: A la Frontière de l'Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences*, pages 599–604, Paris 1985. CESTA, Paris. 200

LeCun, Y. (1987). *Modèles connexionistes de l'apprentissage*. Ph.D. thesis, Université de Paris VI. 16, 431, 449

LeCun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., and Hubbard, W. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, **27**(11), 41–46. 322

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998a). Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag. 372

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998b). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324. 14, 24, 397

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998c). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324. 16, 19, 399, 562, 570, 573

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient based learning applied to document recognition. *Proc. IEEE*. 326

LeCun, Y., Kavukcuoglu, K., and Farabet, C. (2010). Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 253–256. IEEE. 326

L'Ecuyer, P. (1994). Efficiency improvement and variance reduction. In *Proceedings of the 1994 Winter Simulation Conference*, pages 122—132. 191

Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. (2014). Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*. 290

Lee, D.-H., Zhang, S., Fischer, A., and Bengio, Y. (2015). Difference target propagation. In A. Appice, P. P. Rodrigues, V. Santos Costa, C. Soares, J. Gama, and A. Jorge, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 9284 of *Lecture Notes in Computer Science*, pages 498–515. Springer International Publishing.

Lee, H., Ekanadham, C., and Ng, A. (2008). Sparse deep belief net model for visual area V2. In *NIPS'07*. 229, 434

Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In L. Bottou and M. Littman, editors, *ICML 2009*. ACM, Montreal, Canada. 319, 609, 610

Lee, Y. J. and Grauman, K. (2011). Learning the easy things first: self-paced visual category discovery. In *CVPR'2011*. 293

Leibniz, G. W. (1676). Memoir using the chain rule. (Cited in TMME 7:2&3 p 321-332, 2010). 200

Lenat, D. B. and Guha, R. V. (1989). *Building large knowledge-based systems; representation and inference in the Cyc project*. Addison-Wesley Longman Publishing Co., Inc. 2

Leprieur, H. and Haffner, P. (1995). Discriminant learning with minimum memory loss for improved non-vocabulary rejection. In *EUROSPEECH'95*, Madrid, Spain. 569

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, **6**, 861—867. 193, 194

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathematics*, **II**(2), 164–168. 273

L'Hôpital, G. F. A. (1696). *Analyse des infiniment petits, pour l'intelligence des lignes courbes*. Paris: L'Imprimerie Royale. 200

Lin, T., Horne, B. G., Tino, P., and Giles, C. L. (1996). Learning long-term dependencies is not as difficult with NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, **7**(6), 1329–1338. 352, 353

Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. (2015). Learning entity and relation embeddings for knowledge graph completion. In *Proc. AAAI'15*. 426

Linde, N. (1992). The machine that changed the world, episode 3. Documentary miniseries. 2

Lindsey, C. and Lindblad, T. (1994). Review of hardware neural networks: a user's perspective. In *Proc. Third Workshop on Neural Networks: From Biology to High Energy Physics*, pages 195—202, Isola d'Elba, Italy. 391

Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, **16**(2), 146–160. 200

Long, P. M. and Servedio, R. A. (2010). Restricted Boltzmann machines are hard to approximately evaluate or simulate. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*. 586

Lovelace, A. (1842). Notes upon L. F. Menabrea's "Sketch of the Analytical Engine invented by Charles Babbage". 1

Lowerre, B. (1976). *The Harpy Speech Recognition System*. Ph.D. thesis. 563, 569, 575

Lu, T., Pál, D., and Pál, M. (2010). Contextual multi-armed bandits. In *International Conference on Artificial Intelligence and Statistics*, pages 485–492. 423

Luenberger, D. G. (1984). *Linear and Nonlinear Programming*. Addison Wesley. 277

Lukoševičius, M. and Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, **3**(3), 127–149. 350

Luo, H., Carrier, P.-L., Courville, A., and Bengio, Y. (2013). Texture modeling with convolutional spike-and-slab RBMs and deep extensions. In *AISTATS'2013*. 95

Lyness, J. N. and Moler, C. B. (1967). Numerical differentiation of analytic functions. *SIAM J.Numer. Anal.*, **4**, 202—210. 185

Lyu, S. (2009). Interpretation and generalization of score matching. In *UAI'09*. 546

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing*. 164, 199

Maass, W. (1992). Bounds for the computational power and learning complexity of analog neural nets (extended abstract). In *Proc. of the 25th ACM Symp. Theory of Computing*, pages 335–344. 194

Maass, W., Schnitger, G., and Sontag, E. D. (1994). A comparison of the computational power of sigmoid and boolean threshold circuits. *Theoretical Advances in Neural Computation and Learning*, pages 127–151. 194

Maass, W., Natschlaeger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, **14**(11), 2531–2560. 349, 350

MacKay, D. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. 56

Maclaurin, D., Duvenaud, D., and Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. *arXiv preprint arXiv:1502.03492*. 378

Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., and Yuille, A. L. (2015). Deep captioning with multimodal recurrent neural networks. In *ICLR'2015*. arXiv:1410.1090. 95

Marcotte, P. and Savard, G. (1992). Novel approaches to the discrimination problem. *Zeitschrift für Operations Research (Theory)*, **36**, 517–545. 241

Marlin, B., Swersky, K., Chen, B., and de Freitas, N. (2010). Inductive principles for restricted Boltzmann machine learning. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, volume 9, pages 509–516. 541, 546, 588

Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics*, **11**(2), 431–441. 273

Marr, D. and Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, **194**. 322

Martens, J. (2010). Deep learning via Hessian-free optimization. In L. Bottou and M. Littman, editors, *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*, pages 735–742. ACM. 285

Martens, J. and Medabalimi, V. (2014). On the expressive efficiency of sum product networks. *arXiv:1411.7717*. 194, 482

Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In *Proc. ICML'2011*. ACM. 362

Martens, J., Chattopadhyay, A., Pitassi, T., and Zemel, R. (2013). On the representational efficiency of restricted Boltzmann machines. In *NIPS'2013*. 193

Mase, S. (1995). Consistency of the maximum pseudo-likelihood estimator of continuous state space Gibbsian processes. *The Annals of Applied Probability*, **5**(3), pp. 603–612. 544

Matan, O., Burges, C. J. C., LeCun, Y., and Denker, J. S. (1992). Multi-digit recognition using a space displacement neural network. In *NIPS'91*, pages 488–495, San Mateo CA. Morgan Kaufmann. 571

McCullagh, P. and Nelder, J. (1989). *Generalized Linear Models*. Chapman and Hall, London. 167

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, **5**, 115–133. 13

Mead, C. and Ismail, M. (2012). *Analog VLSI implementation of neural systems*, volume 80. Springer Science & Business Media. 391

Mesnil, G., Dauphin, Y., Glorot, X., Rifai, S., Bengio, Y., Goodfellow, I., Lavoie, E., Muller, X., Desjardins, G., Warde-Farley, D., Vincent, P., Courville, A., and Bergstra, J. (2011). Unsupervised and transfer learning challenge: a deep learning approach. In *JMLR W&CP: Proc. Unsupervised and Transfer Learning*, volume 7. 195, 467

Mesnil, G., Rifai, S., Dauphin, Y., Bengio, Y., and Vincent, P. (2012). Surfing on the manifold. Learning Workshop, Snowbird. 618

Miikkulainen, R. and Dyer, M. G. (1991). Natural language processing with modular PDP networks and distributed lexicon. *Cognitive Science*, **15**, 343–399. 402

Mikolov, T. (2012). *Statistical Language Models based on Neural Networks*. Ph.D. thesis, Brno University of Technology. 173, 364

Mikolov, T., Deoras, A., Kombrink, S., Burget, L., and Cernocky, J. (2011a). Empirical evaluation and combination of advanced language modeling techniques. In *Proc. 12th annual conference of the international speech communication association (INTERSPEECH 2011)*. 414

Mikolov, T., Deoras, A., Povey, D., Burget, L., and Cernocky, J. (2011b). Strategies for training large scale neural network language models. In *Proc. ASRU'2011*. 293, 414

Mikolov, T., Le, Q. V., and Sutskever, I. (2013). Exploiting similarities among languages for machine translation. Technical report, arXiv:1309.4168. 470

Minka, T. (2005). Divergence measures and message passing. *Microsoft Research Cambridge UK Tech Rep MSRTR2005173*, **72**(TR-2005-173). 551

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons*. MIT Press, Cambridge. 13

Misra, J. and Saha, I. (2010). Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, **74**(1), 239–255. 391

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York. 92

Mnih, A. and Gregor, K. (2014). Neural variational inference and learning in belief networks. In *ICML'2014*. 192

Mnih, A. and Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2265–2273. Curran Associates, Inc. 413, 549

Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. In *ICML'2012*, pages 1751–1758. 413

Mnih, V. and Hinton, G. (2010). Learning to detect roads in high-resolution aerial images. In *Proceedings of the 11th European Conference on Computer Vision (ECCV)*. 95

Mnih, V., Heess, N., Graves, A., and kavukcuoglu, k. (2014). Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *NIPS'2014*, pages 2204–2212. 192

Mobahi, H. and Fisher III, J. W. (2015). A theoretical analysis of optimization by gaussian continuation. In *AAAI'2015*. 291

Mohamed, A., Dahl, G., and Hinton, G. (2012). Acoustic modeling using deep belief networks. *IEEE Trans. on Audio, Speech and Language Processing*, **20**(1), 14–22. 398

Montúfar, G. (2014). Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, **26**. 481

Montúfar, G. and Ay, N. (2011). Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. *Neural Computation*, **23**(5), 1306–1319. 481

Montufar, G. and Morton, J. (2014). When does a mixture of products contain a product of mixtures? *SIAM Journal on Discrete Mathematics*, **29**(1), 321–347. 480

Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *NIPS'2014*. 17, 194, 479, 482, 483

Mor-Yosef, S., Samueloff, A., Modan, B., Navot, D., and Schenker, J. G. (1990). Ranking the risk factors for cesarean: logistic regression analysis of a nationwide study. *Obstet Gynecol*, **75**(6), 944–7. 2

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *AISTATS'2005*. 409, 411

Mozer, M. C. (1992). The induction of multiscale temporal structure. In *NIPS'91*, pages 275–282, San Mateo, CA. Morgan Kaufmann. 353, 354, 366

Murphy, K. P. (2012). *Machine Learning: a Probabilistic Perspective*. MIT Press, Cambridge, MA, USA. 92, 139

Murray, B. U. I. and Larochelle, H. (2014). A deep and tractable density estimator. In *ICML'2014*. 173, 616, 617

Nadas, A., Nahamoo, D., and Picheny, M. A. (1988). On a model-robust training method for speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **ASSP-36**(9), 1432–1436. 569

Nair, V. and Hinton, G. (2010a). Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*. 165

Nair, V. and Hinton, G. E. (2010b). Rectified linear units improve restricted Boltzmann machines. In L. Bottou and M. Littman, editors, *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*, pages 807–814. ACM. 14

Narayanan, H. and Mitter, S. (2010). Sample complexity of testing the manifold hypothesis. In *NIPS'2010*. 153, 487

Navigli, R. and Velardi, P. (2005). Structural semantic interconnections: a knowledge-based approach to word sense disambiguation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, **27**(7), 1075—1086. 427

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial Intelligence*, **56**, 71–113. 611

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics. Springer. 235

Neal, R. M. (2001). Annealed importance sampling. *Statistics and Computing*, **11**(2), 125–139. 553, 554

Neal, R. M. (2005). Estimating ratios of normalizing constants using linked importance sampling. 555

Nedic, A. and Bertsekas, D. (2000). Convergence rate of incremental subgradient algorithms. In *Stochastic Optimization: Algorithms and Applications*, pages 263–304. Kluwer. 261

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence . *Doklady AN SSSR (translated as Soviet. Math. Docl.)*, **269**, 543–547. 260, 263

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS. 19

Ney, H. and Kneser, R. (1993). Improved clustering techniques for class-based statistical language modelling. In *European Conference on Speech Communication and Technology (Eurospeech)*, pages 973–976, Berlin. 405

Ng, A. (2015). Advice for applying machine learning. https://see.stanford.edu/materials/aimlcs229/ML-advice.pdf. 367

Niesler, T. R., Whittaker, E. W. D., and Woodland, P. C. (1998). Comparison of part-of-speech and automatically derived category-based language models for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 177–180. 405

Ning, F., Delhomme, D., LeCun, Y., Piano, F., Bottou, L., and Barbano, P. E. (2005). Toward automatic phenotyping of developing embryos from videos. *Image Processing, IEEE Transactions on*, **14**(9), 1360–1371. 315

Niranjan, M. and Fallside, F. (1990). Neural networks and radial basis functions in classifying static speech patterns. *Computer Speech and Language*, **4**, 275–289. 165

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer. 85, 90

Norouzi, M. and Fleet, D. J. (2011). Minimal loss hashing for compact binary codes. In *ICML'2011*. 420

Nowlan, S. J. (1990). Competing experts: An experimental investigation of associative mixture models. Technical Report CRG-TR-90-5, University of Toronto. 390

Olshausen, B. and Field, D. J. (2005). How close are we to understanding V1? *Neural Computation*, **17**, 1665–1699. 14

Olshausen, B. A. and Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, **381**, 607–609. 229, 325, 434, 486

Olshausen, B. A. and Field, D. J. (1997). Sparse coding with an overcomplete basis set: a strategy employed by V1? *Vision Research*, **37**, 3311–3325. 446

Opper, M. and Archambeau, C. (2009). The variational gaussian approximation revisited. *Neural computation*, **21**(3), 786–792. 189, 525

Paccanaro, A. and Hinton, G. E. (2000). Extracting distributed representations of concepts and relations from positive and negative propositions. In *International Joint Conference on Neural Networks (IJCNN)*, Como, Italy. IEEE, New York. 426

Parker, D. B. (1985). Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT. 200

Pascanu, R. (2014). *On recurrent and deep networks*. Ph.D. thesis, Université de Montréal. 251, 252

Pascanu, R. and Bengio, Y. (2012). On the difficulty of training recurrent neural networks. Technical Report arXiv:1211.5063, Universite de Montreal. 173

Pascanu, R., Mikolov, T., and Bengio, Y. (2013a). On the difficulty of training recurrent neural networks. In *ICML'2013*. 173, 257, 349, 354, 363, 364, 365, 366

Pascanu, R., Montufar, G., and Bengio, Y. (2013b). On the number of inference regions of deep feed forward networks with piece-wise linear activations. Technical report, U. Montreal, arXiv:1312.6098. 194

Pascanu, R., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014a). How to construct deep recurrent neural networks. In *ICLR'2014*. 17, 234, 346, 347, 348, 356, 399, 482, 483

Pascanu, R., Montufar, G., and Bengio, Y. (2014b). On the number of inference regions of deep feed forward networks with piece-wise linear activations. In *ICLR'2014*. 480

Pearl, J. (1985). Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine*, pages 329–334. 508

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. 49

Petersen, K. B. and Pedersen, M. S. (2006). The matrix cookbook. Version 20051003. 28

Peterson, G. B. (2004). A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, **82**(3), 317–328. 293

Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., and Culurciello, E. (2012). Neuflow: dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 1044–1047. IEEE. 391

Pinheiro, P. H. O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In *ICML'2014*. 315

Pinheiro, P. H. O. and Collobert, R. (2015). From image-level to pixel-level labeling with convolutional networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. 315

Pinto, N., Cox, D. D., and DiCarlo, J. J. (2008). Why is real-world visual object recognition hard? *PLoS Comput Biol*, **4**. 396, 610

Pinto, N., Stone, Z., Zickler, T., and Cox, D. (2011). Scaling up biologically-inspired computer vision: A case study in unconstrained face recognition on facebook. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 35–42. IEEE. 318

Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, **46**(1), 77–105. 348

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, **4**(5), 1–17. 261

Poole, B., Sohl-Dickstein, J., and Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. *CoRR*, **abs/1406.1831**. 215

Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In *UAI'2011*, Barcelona, Spain. 194, 482

Poundstone, W. (2005). *Fortune's Formula: The untold story of the scientific betting system that beat the casinos and Wall Street*. Macmillan. 57

Powell, M. (1987). Radial basis functions for multivariable interpolation: A review. 165

Presley, R. K. and Haggard, R. L. (1994). A fixed point implementation of the backpropagation learning algorithm. In *Southeastcon'94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE*, pages 136–138. IEEE. 391

Price, R. (1958). A useful theorem for nonlinear devices having gaussian inputs. *IEEE Transactions on Information Theory*, **4**(2), 69–72. 189, 525

Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., and Fried, I. (2005). Invariant visual representation by single neurons in the human brain. *Nature*, **435**(7045), 1102–1107. 320

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, **77**(2), 257–286. 398, 567

Rabiner, L. R. and Juang, B. H. (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 257–285. 567

Raiko, T., Yao, L., Cho, K., and Bengio, Y. (2014). Iterative neural autoregressive distribution estimator (NADE-k). Technical report, arXiv:1406.1485. 616

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In L. Bottou and M. Littman, editors, *ICML 2009*, pages 873–880, New York, NY, USA. ACM. 24, 386

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science 120, Springer. 185

Ramsey, F. P. (1926). Truth and probability. In R. B. Braithwaite, editor, *The Foundations of Mathematics and other Logical Essays*, chapter 7, pages 156–198. McMaster University Archive for the History of Economic Thought. 50

Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In *NIPS'2006*. 12, 16, 446, 459, 460, 461

Ranzato, M., Huang, F., Boureau, Y., and LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press. 319

Ranzato, M., Boureau, Y., and LeCun, Y. (2008). Sparse feature learning for deep belief networks. In *NIPS'2007*. 446

Rao, C. (1945). Information and the accuracy attainable in the estimation of statistical parameters. *Bulletin of the Calcutta Mathematical Society*, **37**, 81–89. 127, 261

Rasmus, A., Valpola, H., Honkala, M., Berglund, M., and Raiko, T. (2015). Semi-supervised learning with ladder network. *arXiv preprint arXiv:1507.02672*. 370

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS'2011*. 388

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *ICML'2014*. 189, 525, 526

Richard Socher, Milind Ganjoo, C. D. M. and Ng, A. Y. (2013). Zero-shot learning through cross-modal transfer. In *27th Annual Conference on Neural Information Processing Systems (NIPS 2013)*. 468

Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011a). Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML'2011*. 454, 456, 490

Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., and Glorot, X. (2011b). Higher order contractive auto-encoder. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*. 435

Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., and Glorot, X. (2011c). Higher order contractive auto-encoder. In *ECML PKDD*. 454

Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., and Muller, X. (2011d). The manifold tangent classifier. In *NIPS'2011*. 501, 502

Rifai, S., Bengio, Y., Dauphin, Y., and Vincent, P. (2012). A generative process for sampling contractive auto-encoders. In *ICML'2012*. 617, 618

Ringach, D. and Shapley, R. (2004). Reverse correlation in neurophysiology. *Cognitive Science*, **28**(2), 147–166. 323

Roberts, S. and Everson, R. (2001). *Independent component analysis: principles and practice*. Cambridge University Press. 442

Robinson, A. J. and Fallside, F. (1991). A recurrent error propagation network speech recognition system. *Computer Speech and Language*, **5**(3), 259–274. 24, 398

Rockafellar, R. T. (1997). Convex analysis. princeton landmarks in mathematics. 88

Romero, A., Ballas, N., Ebrahimi Kahou, S., Chassang, A., Gatta, C., and Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *ICLR'2015, arXiv:1412.6550*. 290

Rosen, J. B. (1960). The gradient projection method for nonlinear programming. part i. linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, **8**(1), pp. 181–217. 88

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**, 386–408. 12, 13, 24

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, New York. 13, 24

Roweis, S. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, **290**(5500). 153, 154, 490

Rumelhart, D., Hinton, G., and Williams, R. (1986a). Learning representations by back-propagating errors. *Nature*, **323**, 533–536. 12, 16, 22, 200, 401, 424

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge. 19, 24, 200

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986c). Learning representations by back-propagating errors. *Nature*, **323**, 533–536. 159, 327

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986d). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge. 15, 200

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986e). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge. 159

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014a). ImageNet Large Scale Visual Recognition Challenge. 19

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., *et al.* (2014b). Imagenet large scale visual recognition challenge. *arXiv preprint arXiv:1409.0575*. 25

Rust, N., Schwartz, O., Movshon, J. A., and Simoncelli, E. (2005). Spatiotemporal elements of macaque V1 receptive fields. *Neuron*, **46**(6), 945–956. 322

Sainath, T., rahman Mohamed, A., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for LVCSR. In *ICASSP 2013*. 399

Salakhutdinov, R. and Hinton, G. (2009a). Deep Boltzmann machines. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 5, pages 448–455. 21, 24, 460, 595, 598, 603, 605

Salakhutdinov, R. and Hinton, G. (2009b). Deep Boltzmann machines. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, volume 8. 602, 606, 622

Salakhutdinov, R. and Hinton, G. E. (2007). Semantic hashing. In *SIGIR'2007*. 419, 420

Salakhutdinov, R. and Hinton, G. E. (2008). Using deep belief nets to learn covariance kernels for Gaussian processes. In *NIPS'07*, pages 1249–1256, Cambridge, MA. MIT Press. 473

Salakhutdinov, R. and Mnih, A. (2008). Probabilistic matrix factorization. In *NIPS'2008*. 422

Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *ICML 2008*, volume 25, pages 872–879. ACM. 554

Salakhutdinov, R., Mnih, A., and Hinton, G. (2007). Restricted Boltzmann machines for collaborative filtering. In *ICML*. 422

Salimans, T. and Knowles, D. A. (2013). Fixed-form variational posterior approximation through stochastic linear regression. *Bayesian Analysis*, **8**(4), 837–882. 526

Sanger, T. D. (1994). Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE Transactions on Robotics and Automation*, **10**(3). 293

Saul, L. K., Jaakkola, T., and Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research*, **4**, 61–76. 24

Savich, A. W., Moussa, M., and Areibi, S. (2007). The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *Neural Networks, IEEE Transactions on*, **18**(1), 240–252. 391

Saxe, A. M., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., and Ng, A. (2011). On random weights and unsupervised feature learning. In *Proc. ICML'2011*. ACM. 318

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *ICLR*. 247, 248, 284

Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, **4**(2), 234–242. 347

Schmidhuber, J. (1996). Sequential neural text compression. *IEEE Transactions on Neural Networks*, **7**(1), 142–146. 402

Schölkopf, B. and Smola, A. (2002). *Learning with kernels*. MIT Press. 149

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, **10**, 1299–1319. 153, 490

Schölkopf, B., Burges, C. J. C., and Smola, A. J. (1999). *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA. 16, 165, 196

Schulz, H. and Behnke, S. (2012). Learning two-layer contractive encodings. In *ICANN'2012*, pages 620–628. 456

Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, **45**(11), 2673–2681. 344

Schwenk, H. (2007). Continuous space language models. *Computer speech and language*, **21**, 492–518. 402, 408

Schwenk, H. (2010). Continuous space language models for statistical machine translation. *The Prague Bulletin of Mathematical Linguistics*, **93**, 137–146. 402, 414

Schwenk, H. (2014). Cleaned subset of wmt '14 dataset. 19

Schwenk, H. and Bengio, Y. (1998). Training methods for adaptive boosting of neural networks. In *NIPS'97*, pages 647–653. MIT Press. 232

Schwenk, H. and Gauvain, J.-L. (2002a). Connectionist language modeling for large vocabulary continuous speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 1, pages 765–768. 402

Schwenk, H. and Gauvain, J.-L. (2002b). Connectionist language modeling for large vocabulary continuous speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 765–768, Orlando, Florida. 408

Schwenk, H. and Gauvain, J.-L. (2005). Building continuous space language models for transcribing european languages. In *Interspeech*, pages 737–740. 402

Schwenk, H., Costa-jussà, M. R., and Fonollosa, J. A. R. (2006). Continuous space language models for the iwslt 2006 task. In *International Workshop on Spoken Language Translation*, pages 166–173. 402, 414

Seide, F., Li, G., and Yu, D. (2011). Conversational speech transcription using context-dependent deep neural networks. In *Interspeech 2011*, pages 437–440. 22

Sermanet, P., Chintala, S., and LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. *CoRR*, **abs/1204.3968**. 397

Sermanet, P., Kavukcuoglu, K., Chintala, S., and LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'13)*. IEEE. 22, 195

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, **27**(3), 379—423. 57

Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, **37**(1), 10–21. 57

Shilov, G. (1977). *Linear Algebra*. Dover Books on Mathematics Series. Dover Publications. 28

Siegelmann, H. (1995). Computation beyond the Turing limit. *Science*, **268**(5210), 545–548. 330

Siegelmann, H. and Sontag, E. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, **4**(6), 77–80. 330

Siegelmann, H. T. and Sontag, E. D. (1995). On the computational power of neural nets. *Journal of Computer and Systems Sciences*, **50**(1), 132–150. 256, 330, 331

Simard, D., Steinkraus, P. Y., and Platt, J. C. (2003). Best practices for convolutional neural networks. In *ICDAR'2003*. 326

Simard, P. and Graf, H. P. (1994). Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, pages 232–239. 391

Simard, P., Victorri, B., LeCun, Y., and Denker, J. (1992). Tangent prop - A formalism for specifying selected invariances in an adaptive network. In *NIPS'1991*. 501, 502

Simard, P. Y., LeCun, Y., and Denker, J. (1993). Efficient pattern recognition using a new transformation distance. In *NIPS'92*. 500

Simard, P. Y., LeCun, Y. A., Denker, J. S., and Victorri, B. (1998). Transformation invariance in pattern recognition — tangent distance and tangent propagation. *Lecture Notes in Computer Science*, **1524**. 500

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *ICLR*. 288

Sjöberg, J. and Ljung, L. (1995). Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control*, **62**(6), 1391–1407. 225

Skinner, B. F. (1958). Reinforcement today. *American Psychologist*, **13**, 94–99. 293

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 6, pages 194–281. MIT Press, Cambridge. 515, 527

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *NIPS'2012*. 378

Socher, R., Huang, E. H., Pennington, J., Ng, A. Y., and Manning, C. D. (2011a). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS'2011*. 348, 349

Socher, R., Manning, C., and Ng, A. Y. (2011b). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the Twenty-Eighth International Conference on Machine Learning (ICML'2011)*. 348

Socher, R., Pennington, J., Huang, E. H., Ng, A. Y., and Manning, C. D. (2011c). Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP'2011*. 348

Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP'2013*. 348, 349

Solla, S. A., Levin, E., and Fleisher, M. (1988). Accelerated learning in layered neural networks. *Complex Systems*, **2**, 625–639. 169

Solomonoff, R. J. (1989). A system for incremental learning based on algorithmic probability. 293

Sontag, E. D. and Sussman, H. J. (1989). Backpropagation can give rise to spurious local minima even for networks without hidden layers. *Complex Systems*, **3**, 91–106. 247

Sordoni, A., Bengio, Y., Vahabi, H., Lioma, C., Simonsen, J., and Nie, J.-Y. (2015). A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *Proc. of CIKM*. 420

Spall, J. C. (1992). Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, **37**, 332–341. 185

Spitkovsky, V. I., Alshawi, H., and Jurafsky, D. (2010). From baby steps to leapfrog: how "less is more" in unsupervised dependency parsing. In *HLT'10*. 293

Squire, W. and Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, **40**(1), 110—112. 382

Srivastava, N. and Salakhutdinov, R. (2012). Multimodal learning with deep Boltzmann machines. In *NIPS'2012*. 471

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15**, 1929–1958. 232, 234, 235, 605

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. *arXiv:1505.00387*. 290

Steinkrau, D., Simard, P. Y., and Buck, I. (2005). Using gpus for machine learning algorithms. *2013 12th International Conference on Document Analysis and Recognition*, **0**, 1115–1119. 386

Stewart, L., He, X., and Zemel, R. S. (2007). Learning flexible features for conditional random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **30**(8), 1415–1426. 563

Supancic, J. and Ramanan, D. (2013). Self-paced learning for long-term tracking. In *CVPR'2013*. 293

Sussillo, D. (2014). Random walks: Training very deep nonlinear feed-forward networks with smart initialization. *CoRR*, **abs/1412.6558**. 284, 286

Sutskever, I. (2012). *Training Recurrent Neural Networks*. Ph.D. thesis, Department of computer science, University of Toronto. 352, 362

Sutskever, I. and Tieleman, T. (2010). On the Convergence Properties of Contrastive Divergence. In Y. W. Teh and M. Titterington, editors, *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9, pages 789–795. 539

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *ICML*. 263, 352, 362

Sutskever, I., Vinyals, O., and Le, Q. V. (2014a). Sequence to sequence learning with neural networks. Technical report, arXiv:1409.3215. 23, 94, 356, 357

Sutskever, I., Vinyals, O., and Le, Q. V. (2014b). Sequence to sequence learning with neural networks. In *NIPS'2014*. 340, 345, 416

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press. 99

Sutton, R. S., Mcallester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *NIPS'1999*, pages 1057––1063. MIT Press. 192

Swersky, K. (2010). *Inductive Principles for Learning Restricted Boltzmann Machines*. Master's thesis, University of British Columbia. 452

Swersky, K., Ranzato, M., Buchman, D., Marlin, B., and de Freitas, N. (2011). On autoencoders and score matching for energy based models. In *ICML'2011*. ACM. 547

Swersky, K., Snoek, J., and Adams, R. P. (2014). Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*. 379

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014a). Going deeper with convolutions. Technical report, arXiv:1409.4842. 21, 22, 24, 195, 238, 290, 307

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2014b). Intriguing properties of neural networks. *ICLR*, **abs/1312.6199**. 237

Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *CVPR'2014*. 93

Tang, Y. and Eliasmith, C. (2010). Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning, June 21-24, 2010, Haifa, Israel*. 215

Taylor, G. and Hinton, G. (2009). Factored conditional restricted Boltzmann machines for modeling motion style. In L. Bottou and M. Littman, editors, *ICML 2009*, pages 1025–1032. ACM. 418

Taylor, G., Hinton, G. E., and Roweis, S. (2007). Modeling human motion using binary latent variables. In *NIPS'06*, pages 1345–1352. MIT Press, Cambridge, MA. 418, 419

Tenenbaum, J., de Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, **290**(5500), 2319–2323. 153, 154, 462, 463, 490

Thompson, J., Jain, A., LeCun, Y., and Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In *NIPS'2014*. 315

Thrun, S. (1995). Learning to play the game of chess. In *NIPS'1994*. 501

Tibshirani, R. J. (1995). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society B*, **58**, 267–288. 210

Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *ICML 2008*, pages 1064–1071. ACM. 541, 592

Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal components analysis. *Journal of the Royal Statistical Society B*, **61**(3), 611–622. 440, 441

Tom Schaul, Ioannis Antonoglou, D. S. (2014). Unit tests for stochastic optimization. In *International Conference on Learning Representations*. 269

Torabi, A., Pal, C., Larochelle, H., and Courville, A. (2015). Using descriptive video services to create a large data source for video annotation research. *arXiv preprint arXiv: 1503.01070*. 143

Torralba, A., Fergus, R., and Weiss, Y. (2008). Small codes and large databases for recognition. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'08)*, pages 1–8. 419, 420

Tu, K. and Honavar, V. (2011). On the utility of curricula in unsupervised learning of probabilistic grammars. In *IJCAI'2011*. 293

Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., and Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Computation*, **22**(2), 511–538. 315

Töscher, A., Jahrer, M., and Bell, R. M. (2009). The bigchaos solution to the netflix grand prize. 422

Uria, B., Murray, I., and Larochelle, H. (2013). Rnade: The real-valued neural autoregressive density-estimator. In *NIPS'2013*. 615, 616

van den Oörd, A., Dieleman, S., and Schrauwen, B. (2013). Deep content-based music recommendation. In *NIPS'2013*. 423

van der Maaten, L. and Hinton, G. E. (2008a). Visualizing data using t-SNE. *J. Machine Learning Res.*, **9**. 402, 462, 490, 494

van der Maaten, L. and Hinton, G. E. (2008b). Visualizing data using t-SNE. *Journal of Machine Learning Research*, **9**, 2579–2605. 463

Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*. 385, 392

Vapnik, V. N. (1982). *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, Berlin. 106

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer, New York. 106

Vapnik, V. N. and Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, **16**, 264–280. 106

Vincent, P. (2011a). A connection between score matching and denoising autoencoders. *Neural Computation*, **23**(7). 452, 454, 617

Vincent, P. (2011b). A connection between score matching and denoising autoencoders. *Neural Computation*, **23**(7), 1661–1674. 547, 619

Vincent, P. and Bengio, Y. (2003). Manifold Parzen windows. In *NIPS'2002*. MIT Press. 492

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *ICML 2008*. 215, 449

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Machine Learning Res.*, **11**. 449

Vinyals, O., Kaiser, L., Koo, T., Petrov, S., Sutskever, I., and Hinton, G. (2014a). Grammar as a foreign language. Technical report, arXiv:1412.7449. 356

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2014b). Show and tell: a neural image caption generator. arXiv 1411.4555. 356

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: a neural image caption generator. In *CVPR'2015*. arXiv:1411.4555. 95

Viola, P. and Jones, M. (2001). Robust real-time object detection. In *International Journal of Computer Vision*. 390

Von Melchner, L., Pallas, S. L., and Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. *Nature*, **404**(6780), 871–876. 14

Waibel, A., Hanazawa, T., Hinton, G. E., Shikano, K., and Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **37**, 328–339. 327, 393, 398

Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML'2013*. 235

Wang, S. and Manning, C. (2013). Fast dropout training. In *ICML'2013*. 235

Wang, Z., Zhang, J., Feng, J., and Chen, Z. (2014a). Knowledge graph and text jointly embedding. In *Proc. EMNLP'2014*. 426

Wang, Z., Zhang, J., Feng, J., and Chen, Z. (2014b). Knowledge graph embedding by translating on hyperplanes. In *Proc. AAAI'2014*. 426

Warde-Farley, D., Goodfellow, I. J., Lamblin, P., Desjardins, G., Bastien, F., and Bengio, Y. (2011). pylearn2. `http://deeplearning.net/software/pylearn2`. 387

Wawrzynek, J., Asanovic, K., Kingsbury, B., Johnson, D., Beck, J., and Morgan, N. (1996). Spert-ii: A vector microprocessor system. *Computer*, **29**(3), 79–86. 391

Weaver, L. and Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In *Proc. UAI'2001*, pages 538–545. 192

Weinberger, K. Q. and Saul, L. K. (2004). Unsupervised learning of image manifolds by semidefinite programming. In *CVPR'2004*, pages 988–995. 153, 490

Weiss, Y., Torralba, A., and Fergus, R. (2008). Spectral hashing. In *NIPS*, pages 1753–1760. 420

Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770. 200

Weston, J., Ratle, F., and Collobert, R. (2008). Deep learning via semi-supervised embedding. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *ICML 2008*, pages 1168–1175, New York, NY, USA. ACM. 472

Weston, J., Bengio, S., and Usunier, N. (2010). Large scale image annotation: learning to rank with joint word-image embeddings. *Machine Learning*, **81**(1), 21–35. 349

Weston, J., Chopra, S., and Bordes, A. (2014). Memory networks. *arXiv preprint arXiv:1410.3916*. 360, 427

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, volume 4, pages 96–104. IRE, New York. 13, 19, 21, 24

Wikipedia (2015). List of animals by number of neurons — wikipedia, the free encyclopedia. [Online; accessed 4-March-2015]. 21, 24

Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In *NIPS'95*, pages 514–520. MIT Press, Cambridge, MA. 196

Williams, R. J. (1992). Simple statistical gradient-following algorithms connectionist reinforcement learning. *Machine Learning*, **8**, 229–256. 189, 190, 361, 525

Wilson, J. R. (1984). Variance reduction techniques for digital simulation. *American Journal of Mathematical and Management Sciences*, **4**(3), 277—312. 191

Wolpert, D. and MacReady, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, **1**, 67–82. 257

Wolpert, D. H. (1996). The lack of a priori distinction between learning algorithms. *Neural Computation*, **8**(7), 1341–1390. 110

Wu, R., Yan, S., Shan, Y., Dang, Q., and Sun, G. (2015). Deep image: Scaling up image recognition. arXiv:1501.02876. 22, 388

Wu, Z. (1997). Global continuation for distance geometry problems. *SIAM Journal of Optimization*, **7**, 814–836. 291

Xiong, H. Y., Barash, Y., and Frey, B. J. (2011). Bayesian prediction of tissue-regulated splicing using RNA sequence and cellular context. *Bioinformatics*, **27**(18), 2554–2562. 235

Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015a). Show, attend and tell: Neural image caption generation with visual attention. In *ICML'2015*. 95, 192

Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015b). Show, attend and tell: Neural image caption generation with visual attention. arXiv:1502.03044. 356

Xu, L. and Jordan, M. I. (1996). On convergence properties of the EM algorithm for gaussian mixtures. *Neural Computation*, **8**, 129–151. 568

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *NIPS'2014*. 289

Younes, L. (1998). On the convergence of Markovian stochastic algorithms with rapidly decreasing ergodicity rates. In *Stochastics and Stochastics Models*, pages 177–228. 539, 592

Yu, D., Wang, S., and Deng, L. (2010). Sequential labeling using deep-structured conditional random fields. *IEEE Journal of Selected Topics in Signal Processing*. 288

Zaremba, W. and Sutskever, I. (2014). Learning to execute. arXiv 1410.4615. 293

Zaremba, W. and Sutskever, I. (2015). Reinforcement learning neural turing machines. *arXiv:1505.00521*. 361

Zaslavsky, T. (1975). *Facing Up to Arrangements: Face-Count Formulas for Partitions of Space by Hyperplanes*. Number no. 154 in Memoirs of the American Mathematical Society. American Mathematical Society. 480

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *ECCV'14*. 6

Zhou, J. and Troyanskaya, O. G. (2014). Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. In *ICML'2014*. 623, 624

Zöhrer, M. and Pernkopf, F. (2014). General stochastic networks for classification. In *NIPS'2014*. 623

# Index


$L^p$ norm, 35
$k$-means, 317, 473
$k$-nearest neighbors,
boldindex135, 473
$n$-gram,
boldindex400
Freebase, 423
GeneOntology, 423
Wikibase, 423
WordNet, 423
0-1 loss,
boldindex97, 222, 238

, 318

Absolute value rectification, 163
Active constraint, 90
Adagrad, 262
ADALINE, *see* Adaptive Linear Element
Adaptive Linear Element, 13, 21, 24
Adversarial example, 235
Affine, 102
AIS, *see* annealed importance sampling
Almost everywhere, 68
Ancestral sampling, 526
ANN, *see* Artificial neural network
Annealed importance sampling, 548, 598
Approximate inference, 521
Artificial intelligence, 1
Artificial neural network, *see* Neural network
Asymptotically unbiased, 115
Audio, 313
Autoencoder, 4
Automatic differentiation, 183
Back-propagation, 173
Back-Propagation Through Time, 333
Bagging, 227
Batch gradient descent, 256
Bayes error,
boldindex109
Bayes' rule, 67
Bayesian hyperparameter optimization,
377
Bayesian network, *see* directed graphical model
Bayesian probability, 50
Bayesian statistics,
boldindex126
Beam Search, 570
Beam search, 559
Belief network, *see* directed graphical model
Bernoulli distribution, 59
Bias, 115
Bigram, 400
Binary relation, 422
Boltzmann distribution, 511
Boltzmann machine, 511
Boltzmann Machines, 578
BPTT, *see* Back-Propagation Through Time

CAE, *see* contractive autoencoder
Calculus of variations, 574
Categorical distribution, see multinoulli distribution60
CD, *see* contrastive divergence
Centering trick (DBM), 602
Central limit theorem, 62
Chain rule of probability, 54

Chess, 2
Chord, 517
Chordal graph, 517
Class-based language models, 403
Classical dynamical system, 327
Classical regularization, 202
Classification, 93
Cliffs, 248
Clipping the gradient, 362
Clique potential, *see* factor (graphical model)
CNN, *see* convolutional neural network
Collaborative Filtering, 419
Collider, *see* explaining away
Color images, 313
Computer vision, 390
Concept drift, 464
Conditional computation, *see* dynamic structure
Conditional independence, vi, 54
Conditional probability, 53
Connectionism, 15, 382
Connectionist temporal classification, 558
Consistency, 122
Constrained optimization, 88
Content-based addressing, 359
Content-Based Recommender Systems, 420
Context-specific independence, 516
Contextual Bandits, 420
Continuation methods, 289
Contractive auto-encoder, 498
Contractive autoencoder, 451
Contractive autoencoders, 432
Contrast, 391
Contrastive divergence, 535, 598, 601
Convolution, 292, 605
Convolutional network, 14
Convolutional neural network, 225, boldindex292
Coordinate descent, 279, 601
Correlation, 56
Cost function, *see* objective function
Covariance, vi, 55
Covariance matrix, 56
Cross entropy, boldindex59, 164
Cross-correlation, 294
Cross-validation, 113
CTC, *see* connectionist temporal classification
Curriculum learning, 291
curse of dimensionality, 144
Cyc, 2

D-separation, 513
DAE, *see* denoising autoencoder
Data generating distribution, boldindex103, 123
Data generating process, 103
Data parallelism, 385
Dataset, 97
Dataset augmentation, 391, 395
DBM, *see* deep Boltzmann machine
Decision tree, boldindex137
Decision trees, 473
Decoder, 4
Deep belief network, 24, 457, 553, 579, 588, 606
Deep Blue, 2
Deep Boltzmann machine, 21, 24, 457, 553, 579, 591, 601, 606
Deep learning, 1, 5
Denoising autoencoder, 445
Denoising autoencoders, 187
Denoising score matching, 543
Density estimation, 96
Derivative, vi, 79
Design matrix, boldindex99
Detector layer, 302
Deterministic gradient descent, 256
Diagonal matrix, 37
Dirac delta function, 63
Directed graphical model, 69, 505
Directional derivative, 83
Distributed Representation, 472
Distributed representation, 15
domain adaptation, 462
Dot product, 31

Double exponential distribution, *see* Laplace distribution
Doubly block circulant matrix, 296
Dream sleep, 534, 577
DropConnect, 233
Dropout, 187, *230*, 372, 373, 601
Dynamic structure, 387

E-step, 556
Early stopping, 218, 219, 221-223
EBM, *see* energy-based model
Echo state network, 21, 24, 349
Effective number of parameters, 206
Efficiency, 126
Eigendecomposition, 38
Eigenvalue, 38
Eigenvector, 38
ELBO, *see* evidence lower bound
Element-wise product, *see* Hadamard product, *see* Hadamard product
EM, *see* expectation maximization
Embedding, 487
Empirical distribution, 63
Empirical risk, 238
Empirical risk minimization, 238
Encoder, 4
Energy function, 511
Energy-based model, 511, 591
Ensemble methods, 227
Epoch, 218
Equality constraint, 89
Equivariance, 299
Error function, *see* objective function
ESN, *see* echo state network
Euclidean norm, 35
Euler-Lagrange equation, 575
Evidence lower bound, 555-558, 590
Example, 97
Excess error, 256
Expectation, 55
Expectation maximization, 556
Expected value, *see* expectation
Explaining away, 514
Exploitation, 421
Exploration, 421
Exponential distribution, boldindex62

Factor (graphical model), 508
Factor analysis, 437
Factor graph, 519
Factors of variation, 4
Feature, 97
Feedforward deep network, 156
Fine-tuning, 286
Finite differences, 380
Forward-Backward algorithm, 559
Fourier transform, 313, 315
Fovea, 319
Freebase, 423
Frequentist probability, 50
Frequentist statistics, boldindex126
Fully-visible Bayes network, 609
Functional derivatives, 574
FVBN, *see* fully-visible Bayes network

Gabor function, 320
Gaussian distribution, see Normal distribution
Gaussian kernel, 134
Gaussian mixture, 64
GCN, *see* Global contrast normalization
Generalization, 102
Generalized Lagrange function, *see* Generalized Lagrangian
Generalized Lagrangian, 89
Generative adversarial networks, 187
Gibbs distribution, 509
Gibbs sampling, 527
Global contrast normalization, 392
GPU, *see* Graphics processing unit
Gradient, 82
Gradient clipping, 362
Gradient clipping, 250
Gradient descent, 83
Graph, v
Graph Transformer, 567
Graph transformer, 566
Graphical model, *see* structured probabilistic model
Graphics processing unit, 383

Greedy algorithm, 286
Greedy layer-wise unsupervised pre-training, 455
Greedy supervised pre-training, 286
Grid search, 374

Hadamard product, v, 31
Hard tanh, 163
Harmonium, see Restricted Boltzmann machine524
Harmony theory, 512
Helmholtz free energy, *see* evidence lower bound
Hessian matrix, vi, 84, 268
Hidden layer, 6
Hidden Markov model, 563
HMM, *see* hidden Markov model
Hyperbolic tangent, 162
Hyperparameter optimization, 374
Hyperparameters, 112, 372
Hypothesis space, 104, 110

i.i.d assumptions, 235
i.i.d., 114
i.i.d. assumptions, 103
Identity matrix, 32
Immorality, 516
Independence, vi, 54
Independent and identically distributed, 114
Independent component analysis, 438
Inequality constraint, 89
Inference, 504, 521, 553, 555-557, 572, 574, 576, 577
Information retrieval, 417
Initialization, 280
Integral, vi
Invariance, 302
Isomap, 459

Jacobian matrix, vi, 68, 83
Joint probability, 52

Karush-Kuhn-Tucker conditions, 90
Karush-Kuhn-Tucker, 88
Kernel (convolution), 293, 294
Kernel machine, 473
Kernel trick, 134
KKT, *see* Karush-Kuhn-Tucker
KKT conditions, *see* Karush-Kuhn-Tucker conditions
KL divergence, *see* Kllback-Leibler divergence57
Knowledge base, 2
Kullback-Leibler divergence, vi, boldindex57

Lagrange multipliers, 88, 90, 575
Lagrangian, *see* Gneralized Lagrangian89
Laplace distribution, boldindex63
Latent variable, 64, 433
LCN, *see* local contrast normalization
Leaky units, 352
Learning Relations, 422
Line search, 83
Linear combination, 33
Linear convergence, 256
Linear dependence, 34
Linear factor models, 436
Linear regression, boldindex100, 102, 133
Link Prediction, 423
Liquid state machine, 349
Local conditional probability distribution, 506
Local contrast normalization, 393
Logistic regression, 2, 134
Logistic sigmoid, 7, 65
Long short-term memory, 354
Loop, 517
Loss function, *see* objective function
LSTM, 22, *see* lng short-term memory354

M-step, 556
Machine learning, 2
Main diagonal, 30
Manifold, 151
Manifold hypothesis, 484
Manifold hypothesis, 152
Manifold learning, 151, 484
Manifold Tangent Classifier, 498
MAP inference, 558
Marginal probability, 53

Markov chain, 526, 560
Markov network, see undirected model507
Markov property, 560
Markov random field, see undirected model507
Matrix, iv, v, 29
Matrix inverse, 32
Matrix product, 30
Max pooling, 302
Maximum likelihood, boldindex123
Maxout, 163
Mean field, 598, 601
Mean squared error, 101
Measure theory, 67
Measure zero, 67
Memory network, 359
Memory Networks, 357
Method of steepest descent, *see* gradient descent
Minibatch, 143, 241
Missing inputs, 93
Mixing (Markov chain), 528
Mixture distribution, 64
Mixture of experts, 388, 473
MLP, *see* multilayer perception
MNIST, 18, 19, 601
Model averaging, 227
Model compression, 386
Model identifiability, 245
Model parallelism, 385
Moore-Penrose Pseudoinverse, 41
Moore-Penrose pseudoinverse, 212
Moralized graph, 517
MP-DBM, *see* multi-prediction DBM
MRF (Markov Random Field), see undirected model507
MSE, see mean squared error101
Multi-modal learning, 468
Multi-prediction DBM, 600, 602
Multi-task learning, 233, 464
Multilayer perception, 5
Multilayer perceptron, 24, boldindex156
Multinomial distribution, 60
Multinoulli distribution, 60
Naive Bayes, 2, 71
Nat, 57
natural image, 501
Nearest neighbor regression, boldindex107
Negative definite, 84
Negative phase, 532, 534
Neocognitron, 14, 21, 24
Nesterov momentum, 260
Netflix Grand Prize, 230
Neural network, 12
Neural Turing machine, 359
Neuroscience, 13
Noise-contrastive estimation, 544
Non-parametric model, boldindex107
Norm, vii, 35
Normal distribution, 60, 62
Normal equations, boldindex101, 102, 104, 206
Normalized initialization, 282
Numerical differentiation, 183, *see* finite differences

Object detection, 390
Object recognition, 390
Objective function, 79
Offset, 158
One-shot learning, 466
Orthodox statistics, *see* frequentist statistics
Orthogonal matrix, 37
Orthogonality, 37

Parallel distributed processing, 15
Parameter initialization, 280
Parameter sharing, 297
Parameter tying , Parameter sharing224
Parametric model, boldindex107
Partial derivative, 82
Partition function, 509, 530, 598
PCA, *see* principal components analysis
PCD, *see* stochastic maximum likelihood
Perceptron, 13, 24
Perplexity, 125

Persistent contrastive divergence, *see* stochastic maximum likelihood
Point Estimator, 114
Policy, 421
Pooling, 292, 605
Positive definite, 84
Positive phase, 532, 534
Pre-training, 285, 455
Precision (of a normal distribution), 60, 62
Predictive sparse decomposition, 317, 431, 445
Preprocessing, 390
Primary visual cortex, 317
Principal components analysis, 43, 437, 553
Principle components analysis, 139-141, 153
Prior probability distribution, boldindex126
Probabilistic max pooling, 605
Probability density function, 52
Probability distribution, 51
Probability function estimation, 96
Probability mass function, 51
Product rule of probability, *see* chain rule of probability
PSD, *see* predictive sparse decomposition
Pseudolikelihood, 539

Quadrature pair, 323
Quasi-Newton condition, 275
Quasi-Newton methods, 274

Radial basis function, 163
Random search, 375
Random variable, 51
Ratio matching, 543
RBF, 163
RBM, *see* restricted Boltzmann machine
Receptive field, 298
Recommender Systems, 418
Rectified linear unit, 162
Rectifier, 162
Recurrent network, 24
Recurrent neural network, 329
Recursive neural networks, 327
Regression, 94
Regularization, boldindex111, 111, 199, 372
Reinforcement Learning, 99, 420
Reinforcement learning, 187
ReLU, 162
Representation learning, 2
Restricted Boltzmann machine, 524, 553, 579, 581, 601, 602, 604, 605
Ridge regression, see weight decay203
Risk, 238

Saddle points, 246
Sample mean, 116
Scalar, iv, v, 28
Score matching, 542
Secant condition, 275
Second derivative, 83
Second derivative test, 84
Self-information, 57
Semantic hashing, 417
Semi-supervised learning, 469
Separable convolution, 315
Separation (probabilistic modeling), 512
Set, v
SGD, *see* stochastic gradient descent
Shannon entropy, vi, 57, 575
Shortlist, 406
Sigmoid, vii, *see* logistic sigmoid, 162
Sigmoid belief network, 24
Simple cell, 318
Singular value, *see* singular value decompositio
Singular value decomposition, 40, 140, 141
Singular vector, *see* singular value decompositi
SML, *see* stochastic maximum likelihood
Softmax, 162, 166
Softplus, vii, 65, 163
Spam detection, 2
Sparse coding, 431, 439, 553
Sparse initialization, 283
Sparse representations, 226, 442
Spearmint, 377
spectral radius, 349

Speech recognition, 395
Sphering, *see* Whitening, 393
Spike and slab restricted Boltzmann machine, 604
Square matrix, 34
ssRBM, *see* spike and slab restricted Boltzmann machine
Standard deviation, 55
Standard error, 118
Statistic, 114
Statistical learning theory, 103
Steepest descent, *see* gradient descent
Stochastic gradient descent, 13, boldindex143, 241, boldindex257, 601
Stochastic maximum likelihood, 536, 598, 601
Stochastic pooling, 233
Structure learning, 520
Structured output, 94
Structured probabilistic model, 69, 500
Student-t, 431
Sum rule of probability, 53
Sum-product network, 479
Supervised learning, boldindex98
Support vector machine, 134
Surrogate loss function, 239
SVD, *see* singular value decomposition
Symbolic differentiation, 184
Symmetric matrix, 37, 40

t-SNE, 459
Tangent Distance, 497
Tangent plane, 487
Tangent-Prop, 498
Tanh, 162
TDNN, *see* time-delay neural network
Teacher forcing, 331
Tensor, iv, v, 30
Test set, 103
Tikhonov regularization, *see* weight decay
Tiled convolution, 309
Time-delay neural network, 320, 326
Time-delay neural networks, 352
Toeplitz matrix, 296
Trace operator, 42
Training error, 103
Transcription, 94
Transfer learning, 462
Translation, 94
Transpose, v, 30
Triangle inequality, 35
Triangulated graph, *see* chordal graph
Trigram, 400

Unbiased, 115
Undirected graphical model, 69
Undirected model, 507
Uniform distribution, 52
Unigram, 400
Unit norm, 37
Unit vector, 37
Universal approximation theorem, 190
Universal approximator, 478
Unnormalized probability distribution, 508
Unsupervised learning, boldindex98, 138
Unsupervised pre-training, 455

V-structure, *see* explaining away
V1, 317
Variance, vi, 55
Variational autoencoder, 187
Variational derivatives, *see* functional derivatives
Variational free energy, *see* evidence lower bound
Vector, iv, v, 29
Visible layer, 6
Viterbi algorithm, 559
Viterbi decoding, 562
Volumetric data, 313

Weight decay, 110, boldindex203, 373
Weight space symmetry, 245
Weights, 13, 100
Whitening, 393
Wikibase, 423
Word-Sense Disambiguation, 424

Zero-data learning, 466
Zero-shot learning, 466