

Deep Learning

Yoshua Bengio
Ian J. Goodfellow
Aaron Courville

March 30, 2015

Table of Contents

Acknowledgments	1
Notation	2
1 Introduction	4
1.1 Who Should Read This Book?	10
1.2 Historical Trends in Deep Learning	12
I Applied math and machine learning basics	18
2 Linear Algebra	20
2.1 Scalars, Vectors, Matrices and Tensors	20
2.2 Multiplying Matrices and Vectors	22
2.3 Identity and Inverse Matrices	24
2.4 Linear Dependence, Span, and Rank	25
2.5 Norms	26
2.6 Special Kinds of Matrices and Vectors	28
2.7 Eigendecomposition	29
2.8 Singular Value Decomposition	30
2.9 The Trace Operator	31
2.10 Determinant	31
2.11 Example: Principal Components Analysis	32
3 Probability and Information Theory	35
3.1 Why Probability?	35
3.2 Random Variables	37
3.3 Probability Distributions	37
3.3.1 Discrete Variables and Probability Mass Functions	38
3.3.2 Continuous Variables and Probability Density Functions	38
3.4 Marginal Probability	39
3.5 Conditional Probability	39
3.6 The Chain Rule of Conditional Probabilities	40
3.7 Independence and Conditional Independence	40

3.8	Expectation, Variance, and Covariance	41
3.9	Information Theory	42
3.10	Common Probability Distributions	44
3.10.1	Bernoulli Distribution	44
3.10.2	Multinoulli Distribution	44
3.10.3	Gaussian Distribution	45
3.10.4	Dirac Distribution	47
3.10.5	Mixtures of Distributions and Gaussian Mixture	48
3.11	Useful Properties of Common Functions	48
3.12	Bayes' Rule	51
3.13	Technical Details of Continuous Variables	51
3.14	Example: Naive Bayes	52
4	Numerical Computation	56
4.1	Overflow and Underflow	56
4.2	Poor Conditioning	57
4.3	Gradient-Based Optimization	58
4.4	Constrained Optimization	65
4.5	Example: Linear Least Squares	68
5	Machine Learning Basics	70
5.1	Learning Algorithms	70
5.1.1	The Task, T	70
5.1.2	The Performance Measure, P	72
5.1.3	The Experience, E	73
5.2	Example: Linear Regression	74
5.3	Generalization, Capacity, Overfitting and Underfitting	76
5.3.1	Generalization	76
5.3.2	Capacity	77
5.3.3	Occam's Razor, Underfitting and Overfitting	78
5.4	Estimating and Monitoring Generalization Error	81
5.5	Estimators, Bias, and Variance	83
5.5.1	Point Estimation	83
5.5.2	Bias	84
5.5.3	Variance	85
5.5.4	Trading off Bias and Variance and the Mean Squared Error	85
5.5.5	Consistency	86
5.6	Maximum Likelihood Estimation	87
5.6.1	Properties of Maximum Likelihood	87
5.6.2	Regularized Likelihood	87
5.7	Bayesian Statistics	87
5.8	Supervised Learning	88
5.8.1	Estimating Conditional Expectation by Minimizing Squared Error	88

5.8.2	Estimating Probabilities or Conditional Probabilities by Maximum Likelihood	89
5.9	Unsupervised Learning	90
5.9.1	Principal Components Analysis	91
5.10	Weakly Supervised Learning	93
5.11	The Smoothness Prior, Local Generalization and Non-Parametric Models	93
5.12	Manifold Learning and the Curse of Dimensionality	97
5.13	Challenges of High-Dimensional Distributions	100
II	Modern practical deep networks	102
6	Feedforward Deep Networks	104
6.1	Formalizing and Generalizing Neural Networks	104
6.2	Parametrizing a Learned Predictor	108
6.2.1	Family of Functions	108
6.2.2	Loss Function and Conditional Log-Likelihood	109
6.2.3	Training Criterion and Regularizer	115
6.2.4	Optimization Procedure	116
6.3	Flow Graphs and Back-Propagation	117
6.3.1	Chain Rule	118
6.3.2	Back-Propagation in an MLP	119
6.3.3	Back-Propagation in a General Flow Graph	120
6.4	Universal Approximation Properties and Depth	126
6.5	Feature / Representation Learning	128
6.6	Piecewise Linear Hidden Units	129
6.7	Historical Notes	130
7	Regularization	131
7.1	Classical Regularization: Parameter Norm Penalty	132
7.1.1	L^2 Parameter Regularization	133
7.1.2	L^1 Regularization	135
7.1.3	L^∞ Regularization	138
7.2	Classical Regularization as Constrained Optimization	138
7.3	Regularization from a Bayesian Perspective	139
7.4	Regularization and Under-Constrained Problems	139
7.5	Dataset Augmentation	141
7.6	Classical Regularization as Noise Robustness	142
7.7	Bagging and Other Ensemble Methods	142
7.8	Early Stopping as a Form of Regularization	143
7.9	Parameter Sharing	150
7.10	Sparse Representations	151
7.11	Dropout	151
7.12	Multi-Task Learning	154

8 Optimization for Training Deep Models	156
8.1 Optimization for Model Training	156
8.1.1 Empirical Risk Minimization	156
8.1.2 Surrogate Loss Functions	157
8.1.3 Generalization	157
8.1.4 Batches and Minibatches	158
8.1.5 Data Parallelism	158
8.2 Challenges in Optimization	158
8.2.1 Local Minima	158
8.2.2 Ill-Conditioning	158
8.2.3 Plateaus, Saddle Points, and Other Flat Regions	158
8.2.4 Cliffs and Exploding Gradients	158
8.2.5 Vanishing and Exploding Gradients - An Introduction to the Issue of Learning Long-Term Dependencies	161
8.3 Optimization Algorithms	164
8.3.1 Gradient Descent	164
8.3.2 Stochastic Gradient Descent	165
8.3.3 Momentum	166
8.3.4 Adagrad	167
8.3.5 RMSprop	167
8.3.6 Adadelta	168
8.3.7 No Pesky Learning Rates	168
8.4 Approximate Natural Gradient and Second-Order Methods	168
8.5 Conjugate Gradients	168
8.6 BFGS	168
8.6.1 New	168
8.6.2 Optimization Strategies and Meta-Algorithms	168
8.6.3 Coordinate Descent	168
8.6.4 Greedy Supervised Pre-training	169
8.7 Hints and Curriculum Learning	169
9 Convolutional Networks	173
9.1 The Convolution Operation	173
9.2 Motivation	175
9.3 Pooling	178
9.4 Variants of the Basic Convolution Function	183
9.5 Data Types	188
9.6 Efficient Convolution Algorithms	190
9.7 Deep Learning History	190
10 Sequence Modeling: Recurrent and Recursive Nets	191
10.1 Unfolding Flow Graphs and Sharing Parameters	191
10.2 Recurrent Neural Networks	193
10.2.1 Computing the Gradient in a Recurrent Neural Network	195

10.2.2	Recurrent Networks as Generative Directed Acyclic Models	197
10.2.3	RNNs to Represent Conditional Probability Distributions	199
10.3	Bidirectional RNNs	201
10.4	Recursive Neural Networks	204
10.5	Auto-Regressive Networks	205
10.5.1	Logistic Auto-Regressive Networks	206
10.5.2	Neural Auto-Regressive Networks	207
10.5.3	NADE	209
10.6	Facing the Challenge of Long-Term Dependencies	210
10.6.1	Echo State Networks: Choosing Weights to Make Dynamics Barely Contractive	210
10.6.2	Combining Short and Long Paths in the Unfolded Flow Graph	212
10.6.3	Leaky Units and a Hierarchy of Different Time Scales	213
10.6.4	The Long-Short-Term-Memory Architecture and Other Gated RNNs	214
10.6.5	Deep RNNs	217
10.6.6	Better Optimization	218
10.6.7	Clipping Gradients	219
10.6.8	Regularizing to Encourage Information Flow	220
10.6.9	Organizing the State at Multiple Time Scales	221
10.7	Handling Temporal Dependencies with N-Grams, HMMs, CRFs and Other Graphical Models	222
10.7.1	N-grams	222
10.7.2	Efficient Marginalization and Inference for Temporally Structured Outputs by Dynamic Programming	223
10.7.3	HMMs	227
10.7.4	CRFs	229
10.8	Combining Neural Networks and Search	231
10.8.1	Approximate Search	233
11	Large scale deep learning	236
11.1	Fast CPU Implementations	236
11.2	GPU Implementations	236
11.3	Asynchronous Parallel Implementations	236
11.4	Model Compression	236
11.5	Dynamically Structured Nets	237
12	Practical methodology	238
12.1	When to Gather More Data, Control Capacity, or Change Algorithms	238
12.2	Machine Learning Methodology 101	238
12.3	Manual Hyperparameter Tuning	238
12.4	Hyper-parameter Optimization Algorithms	238
12.5	Tricks of the Trade for Deep Learning	240
12.5.1	Debugging Back-Prop	240

12.5.2	Automatic Differentiation and Symbolic Manipulations of Flow Graphs	240
12.5.3	Momentum and Other Averaging Techniques as Cheap Second Order Methods	240
13	Applications	241
13.1	Computer Vision	241
13.1.1	Preprocessing	242
13.1.2	Convolutional Nets	247
13.2	Speech Recognition	247
13.3	Natural Language Processing and Neural Language Models	248
13.3.1	The Basics of Neural Language Models	248
13.3.2	The Problem With N-Grams	248
13.3.3	How Neural Language Models can Generalize Better	250
13.3.4	Neural Machine Translation	252
13.3.5	High-Dimensional Outputs	252
13.3.6	Combining Neural Language Models with N-Grams	252
13.4	Structured Outputs	252
13.5	Other Applications	252
III	Deep learning research	253
14	Structured Probabilistic Models: A Deep Learning Perspective	255
14.1	The Challenge of Unstructured Modeling	256
14.2	Using Graphs to Describe Model Structure	259
14.2.1	Directed Models	259
14.2.2	Undirected Models	261
14.2.3	The Partition Function	263
14.2.4	Energy-Based Models	265
14.2.5	Separation and D-Separation	266
14.2.6	Converting Between Undirected and Directed Graphs	267
14.2.7	Marginalizing Variables out of a Graph	272
14.2.8	Factor Graphs	272
14.3	Advantages of Structured Modeling	272
14.4	Learning About Dependencies	273
14.4.1	Latent Variables Versus Structure Learning	273
14.4.2	Latent Variables for Feature Learning	274
14.5	Inference and Approximate Inference Over Latent Variables	274
14.5.1	Reparametrization Trick	274
14.6	The Deep Learning Approach to Structured Probabilistic Modeling	276
14.6.1	Example: The Restricted Boltzmann Machine	277

15 Monte Carlo Methods	279
15.1 Markov Chain Monte Carlo Methods	279
15.1.1 Markov Chain Theory	280
16 Linear Factor Models and Auto-Encoders	282
16.1 Regularized Auto-Encoders	283
16.2 Representational Power, Layer Size and Depth	287
16.3 Reconstruction Distribution	288
16.4 Linear Factor Models	289
16.5 Probabilistic PCA and Factor Analysis	289
16.5.1 ICA	291
16.5.2 Sparse Coding as a Generative Model	292
16.6 Probabilistic Interpretation of Reconstruction Error as Log-Likelihood	293
16.7 Sparse Representations	295
16.7.1 Sparse Auto-Encoders	296
16.7.2 Predictive Sparse Decomposition	298
16.8 Denoising Auto-Encoders	298
16.8.1 Learning a Vector Field that Estimates a Gradient Field	301
16.9 Contractive Auto-Encoders	304
17 Representation Learning	307
17.1 Greedy Layerwise Unsupervised Pre-Training	308
17.1.1 Why Does Unsupervised Pre-Training Work?	311
17.2 Transfer Learning and Domain Adaptation	315
17.3 Semi-Supervised Learning	322
17.4 Causality, Semi-Supervised Learning and Disentangling the Underlying Factors	323
17.5 Assumption of Underlying Factors and Distributed Representation	325
17.6 Exponential Gain in Representational Efficiency from Distributed Representations	329
17.7 Exponential Gain in Representational Efficiency from Depth	330
17.8 Priors Regarding The Underlying Factors	333
18 The Manifold Perspective on Representation Learning	336
18.1 Manifold Interpretation of PCA and Linear Auto-Encoders	344
18.2 Manifold Interpretation of Sparse Coding	347
18.3 Manifold Learning via Regularized Auto-Encoders	347
18.4 Tangent Distance, Tangent-Prop, and Manifold Tangent Classifier	348
19 Confronting the Partition Function	352
19.1 Estimating the Partition Function	352
19.1.1 Annealed Importance Sampling	354
19.1.2 Bridge Sampling	357
19.1.3 Extensions	357

19.2	Stochastic Maximum Likelihood and Contrastive Divergence	358
19.3	Pseudolikelihood	365
19.4	Score Matching and Ratio Matching	367
19.5	Denoising Score Matching	369
19.6	Noise-Contrastive Estimation	370
20	Approximate inference	372
20.1	Inference as Optimization	372
20.2	Expectation Maximization	375
20.3	MAP Inference: Sparse Coding as a Probabilistic Model	375
20.4	Variational Inference and Learning	376
20.4.1	Discrete Latent Variables	378
20.4.2	Calculus of Variations	378
20.4.3	Continuous Latent Variables	380
20.5	Stochastic Inference	380
20.6	Learned Approximate Inference	380
21	Deep Generative Models	382
21.1	Restricted Boltzmann Machines	382
21.1.1	Conditional Distributions	384
21.1.2	RBM Gibbs Sampling	385
21.2	Training Restricted Boltzmann Machines	385
21.2.1	Contrastive Divergence Training of the RBM	388
21.2.2	Stochastic Maximum Likelihood (Persistent Contrastive Divergence) for the RBM	388
21.2.3	Other Inductive Principles	388
21.3	Deep Belief Networks	389
21.4	Deep Boltzmann Machines	391
21.4.1	Interesting Properties	395
21.4.2	Mean Field Inference in the DBM	395
21.4.3	Variational Expectation Maximization	396
21.4.4	Variational Learning With SML	396
21.4.5	Layerwise Pretraining	397
21.4.6	Multi-Prediction Deep Boltzmann Machines	398
21.4.7	Centered Deep Boltzmann Machines	398
21.5	Boltzmann Machines for Real-Valued Data	400
21.5.1	Gaussian-Bernoulli RBMs	400
21.5.2	mcRBMs	400
21.5.3	mPoT Model	400
21.5.4	Spike and Slab Restricted Boltzmann Machines	401
21.6	Convolutional Boltzmann Machines	401
21.7	Other Boltzmann Machines	402
21.8	Directed Generative Nets	403
21.8.1	Sigmoid Belief Nets	403

21.8.2	Variational Autoencoders	403
21.8.3	Variational Interpretation of PSD	403
21.8.4	Generative Adversarial Networks	403
21.9	A Generative View of Autoencoders	404
21.9.1	Markov Chain Associated with any Denoising Auto-Encoder . .	405
21.9.2	Clamping and Conditional Sampling	407
21.9.3	Walk-Back Training Procedure	408
21.10	Generative Stochastic Networks	409
21.10.1	Discriminant GSNs	410
21.11	Methodological Notes	411
Bibliography		413
Index		437

Acknowledgments

We would like to thank the following people who commented our proposal for the book and helped plan its contents and organization: Hugo Larochelle, Guillaume Alain, Kyunghyun Cho, Caglar Gulcehre (TODO diacritics), Razvan Pascanu, David Krueger and Thomas Rohée.

We would like to thank the following people who offered feedback on the content of the book itself:

In many chapters: Julian Serban, Laurent Dinh, Guillaume Alain, Ilya Sutskever, Vincent Vanhoucke, David Warde-Farley, Jurgen Van Gael, Dustin Webb, Johannes Roith, Ion Androutsopoulos, Paweł Chilinski, Halis Sak, Grigory Sapunov, Ion Androutsopoulos.

Introduction: Johannes Roith, Eric Morris, Samira Ebrahimi, Ozan Çaglayan, Martín Abadi.

Math background chapters:

Linear algebra: Pierre Luc Carrier, Li Yao, Thomas Rohée, Colby Toland, Amjad Almahairi, Sergey Oreshkov,

Probability: Rasmus Antti, Stephan Gouws, Vincent Dumoulin, Artem Oboturov, Li Yao, John Philip Anderson.

Numerical: Meire Fortunato,

Optimization: Marcel Ackermann

ML: Dzmitry Bahdanau Kelvin Xu

MLPs:

Convolutional nets: Mehdi Mirza, Caglar Gulcehre.

Unsupervised: Kelvin Xu

Partition function: Sam Bowman.

Graphical models: Kelvin Xu

RNNs: Kelvin Xu Dmitriy Serdyuk Dongyu Shi

We also want to thank David Warde-Farley, Matthew D. Zeiler, Rob Fergus, Chris Olah, Jason Yosinski, Nicolas Chapados and James Bergstra for contributing images or figures (as noted in the captions).

TODO— this section is just notes, write it up in nice presentation form.

Notation

Mathematical Objects

a	A scalar (integer or real) value with the name “a”
\mathbf{a}	A vector with the name “a”
\mathbf{A}	A matrix with the name “A”
TODO	TODO— higher order tensors
\mathbb{A}	A set with the name “A”
\mathbb{R}	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
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”
\mathcal{G}	A graph with the name “G”

Indexing

a_i	Element i of vector \mathbf{a} , with indexing starting at 1
$A_{i,j}$	Element i, j of matrix \mathbf{A}
$\mathbf{A}_{i,:}$	Row i of matrix \mathbf{A}
$\mathbf{A}_{:,i}$	Column i of matrix \mathbf{A}
TODO	TODO— higher order tensors
a_i	Element i of the random vector \mathbf{a}
$\mathbf{x}^{(t)}$	usually the t -th example (input) from a dataset, with $y^{(t)}$ the associated target, for supervised learning
\mathbf{X}	The matrix of input examples, with one row per example $\mathbf{x}^{(t)}$.

Linear Algebra Operations

\mathbf{A}^\top Transpose of matrix \mathbf{A}

$\mathbf{A} \odot \mathbf{B}$ Element-wise (Hadamard) product of \mathbf{A} and \mathbf{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

$\nabla_{\mathbf{x}} y$ Gradient of y with respect to \mathbf{x}

$\nabla_{\mathbf{x}} y$ Matrix derivatives of y with respect to \mathbf{x}

$\int f(\mathbf{x}) d\mathbf{x}$ Definite integral over the entire domain of \mathbf{x}

$\int_{\mathbb{S}} f(\mathbf{x}) d\mathbf{x}$ Definite integral with respect to \mathbf{x} over the set \mathbb{S}

Miscellaneous

$f \circ g$ Composition of the functions f and g

$\log x$ Natural logarithm of x

Probability and Information Theory

$a \perp b$ The random variables a and b are independent.

$a \perp b | c$ The random variables a and b are conditionally independent given c .

$\mathbb{E}_{x \sim P}[f(x)]$ or $\mathbb{E}f(x)$ Expectation of $f(x)$ with respect to $P(x)$

$Var(f(x))$ Variance of $f(x)$ under $P(x)$

$Cov(f(x), g(x))$ Covariance of $f(x)$ and $g(x)$ under $P(x, y)$

$D_{KL}(P \| Q)$ Kullback-Leibler divergence of P and Q

TODO—norms TODO—entropy TODO—Jacobian and Hessian TODO—Specify that unless otherwise clear from context, functions applied to vectors and matrices are applied elementwise.

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 concept, 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 com-

puter. 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 such as chess that are among the most difficult mental undertakings for a human being are among the easiest for a computer. A person's everyday life requires an immense amount of knowledge about the world, and 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 Cyc¹. Cyc (Lenat and Guha, 1989) 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*² 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

¹<http://www.amazon.com/Building-Large-Knowledge-Based-Systems-Representation/dp/0201517523>

²Logistic regression was developed in statistics to generalize linear regression to the prediction of the conditional probability of categorical variables. It can be viewed as a multi-layer neural network with no hidden layer, trained for classification of labels y given inputs \mathbf{x} with the conditional log-likelihood criterion $-\log P(y|\mathbf{x})$. Note how very similar algorithms, such as logistic regression, have been developed in parallel in the machine learning community and in the statistics community, often not using the same language (Breiman, 2001).

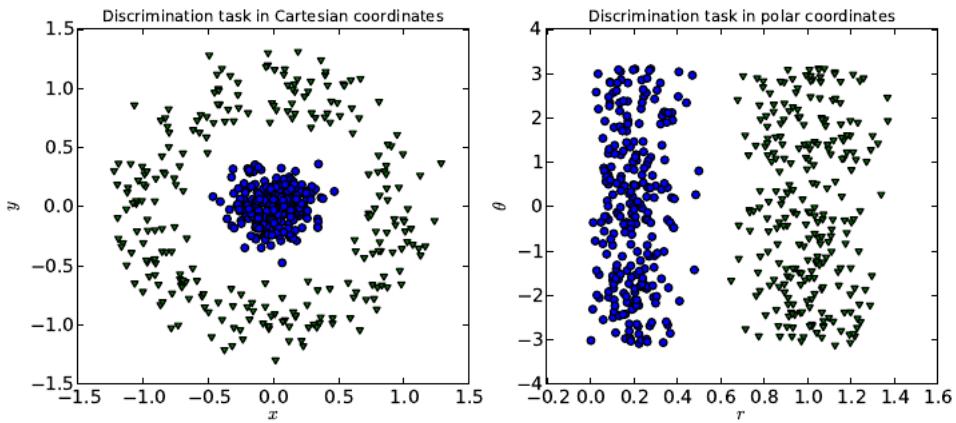


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data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

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 voxels³ 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

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

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 often 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 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 and 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 *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 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.

“Depth” is not a mathematically rigorous term in this context; there is no formal definition of deep learning and no generally accepted convention for measuring the depth of a particular model. All approaches to deep learning share the idea of nested representations of data, but different approaches view depth in different ways. For some approaches, the depth of the system is the depth of the flowchart describing the computations needed to produce the final representation. The depth corresponds roughly to the number of times we update the representation (and of course, what one person considers to be a single complex update, another person may consider to be multiple simple updates, so even two people using this same basic approach to defining depth may not agree on the exact number of layers present in a model). Other approaches consider depth to be 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.

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

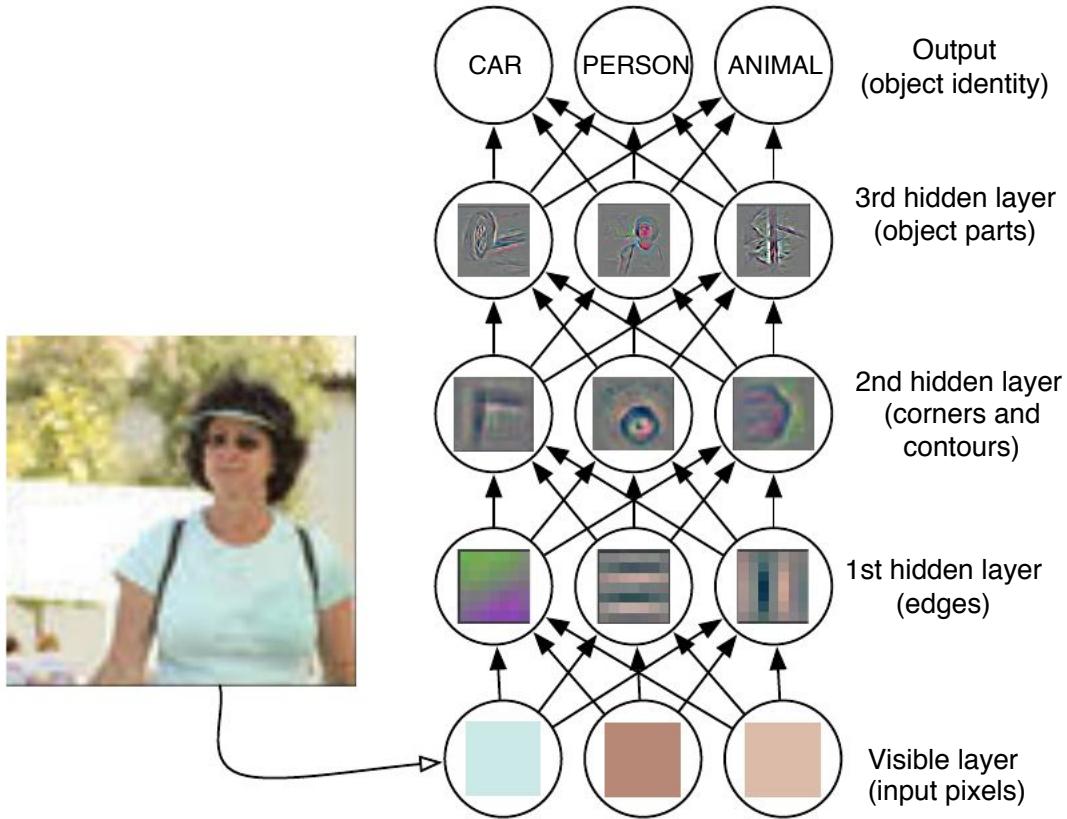


Figure 1.2: Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating 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 \(2014\)](#).

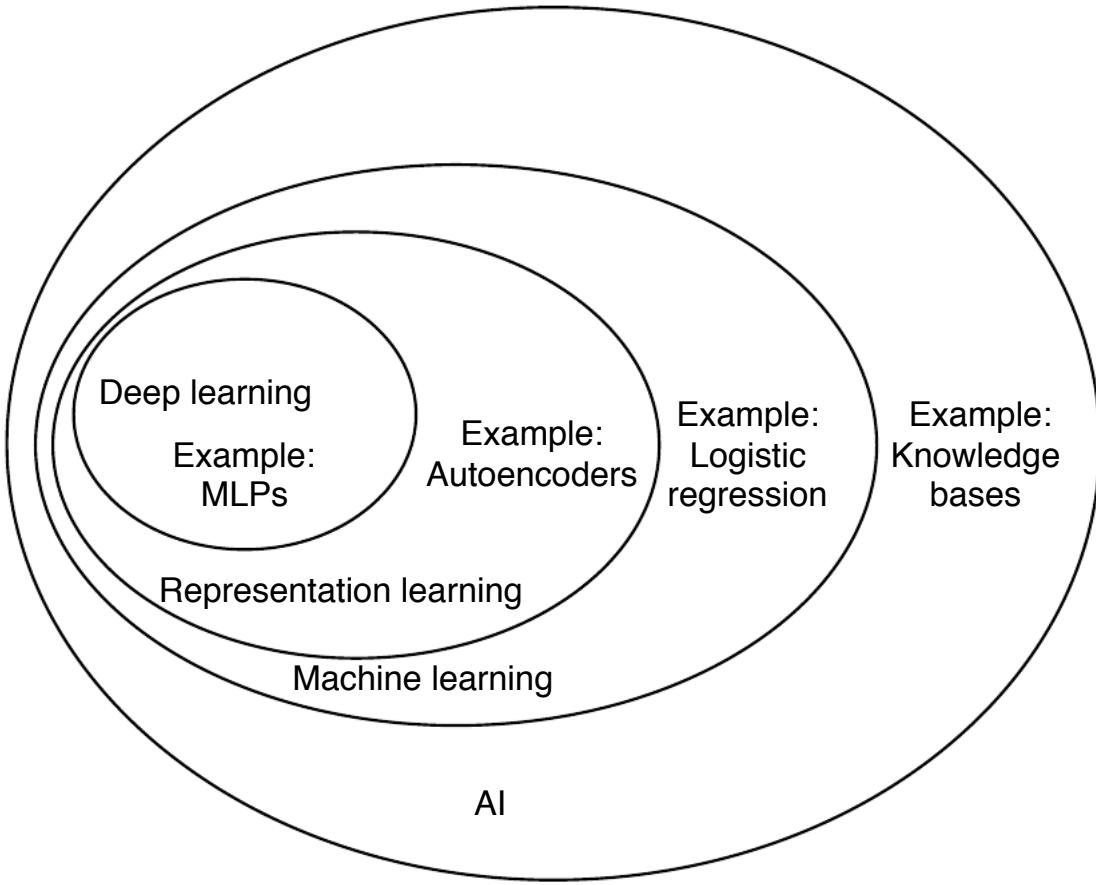


Figure 1.3: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

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, with each concept defined in relation to simpler concepts. Fig. 1.3 illustrates the relationship between these different AI disciplines. Fig. 1.4 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

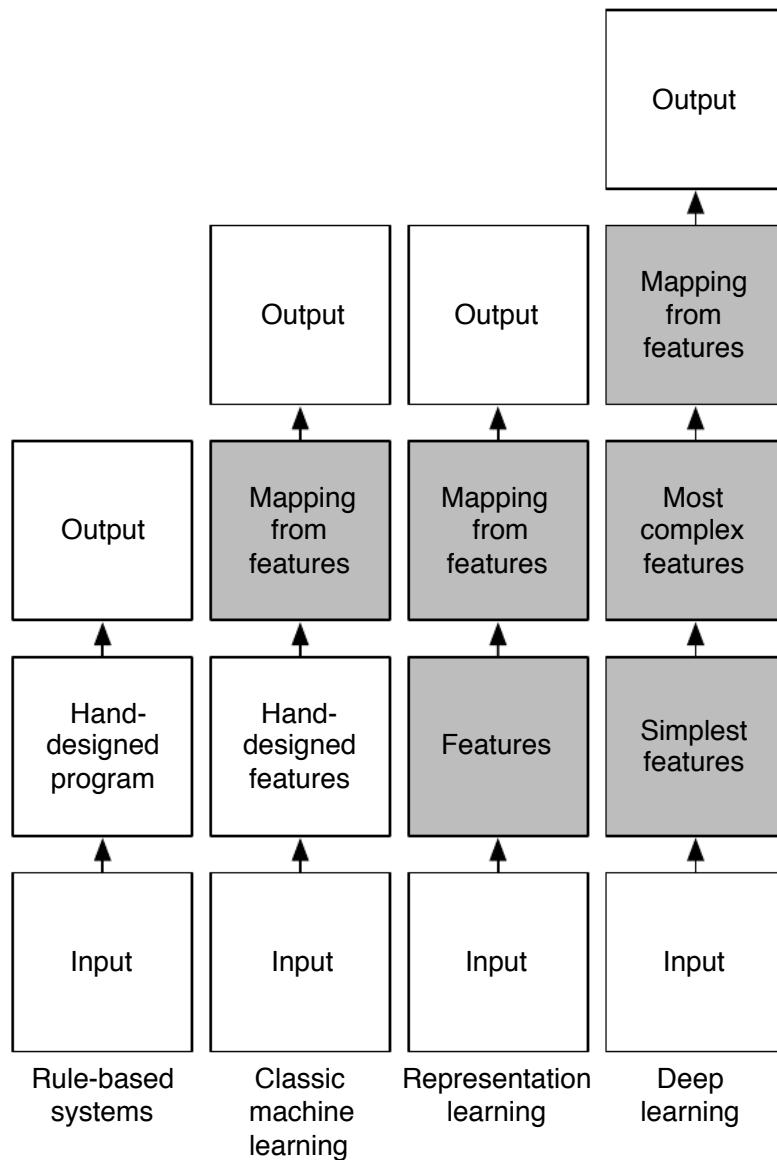


Figure 1.4: Flow-charts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.

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 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

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. Previous names include “artificial neural networks,” “parallel distributed processing,” and “connectionism.”

One important perspective in the history of deep learning is the idea that artificial intelligence should draw inspiration from the brain (whether the human brain or the brains of animals). This perspective gave rise to the “neural network” terminology. Unfortunately, we know extremely little about the brain. The brain contains billions of neurons with tens of thousands of connections between neurons. We are not yet able to accurately record the individual activities of more than a handful of neurons simultaneously. Consequently, we do not have the right kind of data to reverse engineer the algorithms used by the brain. Deep learning algorithms resemble the brain insofar as both the brain and deep learning models involve a very large number of computation units that are not especially intelligent in isolation but become intelligent when they interact with each other. Beyond that, it is difficult to say how similar the two are, and unlikely that they have many similarities, and our knowledge of the brain does not give very specific guidance for improving deep learning. For these reasons, modern terminology no longer emphasizes the biological inspiration of deep learning algorithms. Deep learning has now drawn considerably useful insights from many fields other than neuroscience, including structured probabilistic models and manifold learning, and the

modern terminology aims to avoid implying that only one field has inspired the current algorithms.

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 and the size of the model increases. In the age of “Big Data” we now have large enough training sets to make deep learning algorithms consistently perform well (see Fig. 1.5), and fast enough CPUs or GPUs and enough memory to train very large models (See Fig. 1.6 and Fig. 1.7). The algorithms reaching human performance on complex tasks today are very similar to the algorithms that struggled to solve toy problems in the 1980s—the most important difference is that today we can provide these algorithms with the resources they need to succeed.

The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. These models took a vector of n input values \mathbf{x} and computed a simple function $f(\mathbf{x}) = \sum_{i=1}^n w_i x_i$ using a vector of learned “weights” \mathbf{w} . The Perceptron (Rosenblatt, 1958, 1962) could recognize two different categories of inputs by testing whether $f(\mathbf{x})$ is positive or negative. The Adaptive Linear Element (ADALINE) simply returned the value of $f(\mathbf{x})$ itself to predict a real number (Widrow and Hoff, 1960).

These simple learning algorithms greatly affected the modern landscape of machine learning. ADALINE can be seen as using the stochastic gradient descent algorithm, which is still in use in state of the art deep learning algorithms today with only slight modification, to train a linear regression model, which is still used in cases where we prefer speed or interpretability of the model over the ability to fit complex training sets, or where we have too little training data or too noisy of a relationship between inputs and outputs to fit a more complicated model.

Unfortunately, the limitations of these linear models lead to a backlash against biologically inspired machine learning in general (Minsky and Papert, 1969) and other approaches dominated AI until the early 1980s. In the mid-1980’s, the back-propagation algorithm enabled the extension of biologically-inspired machine learning approaches to more complex models that incorporated non-linear behavior via the introduction of hidden layers (Rumelhart *et al.*, 1986a; LeCun, 1987). Neural networks became popular again 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 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 that were more amenable to theoretical analysis. Kernel machines (Boser *et al.*, 1992; Cortes and Vapnik, 1995; Schölkopf *et al.*, 1999) and graphical models (Jordan, 1998) became the main focus of academic study, while hand-designing domain-specific features became the typical approach to practical applications. During this time, neural networks continued to obtain impressive performance on some tasks (LeCun *et al.*,

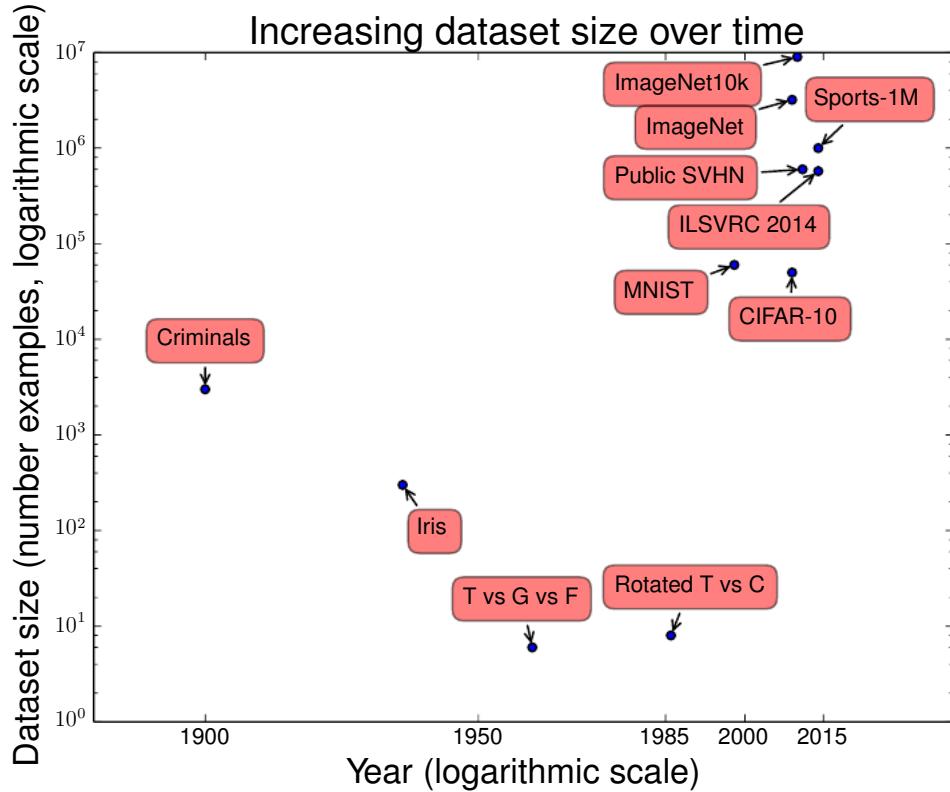


Figure 1.5: Dataset sizes have increased greatly over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through 1980s, the pioneers of biologically-inspired machine 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 of scans of handwritten numbers (LeCun *et al.*, 1998a). 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, 2010; Russakovsky *et al.*, 2014), and the Sports-1M dataset (Karpathy *et al.*, 2014). Deep learning methods so far require large, labeled datasets to succeed. 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.

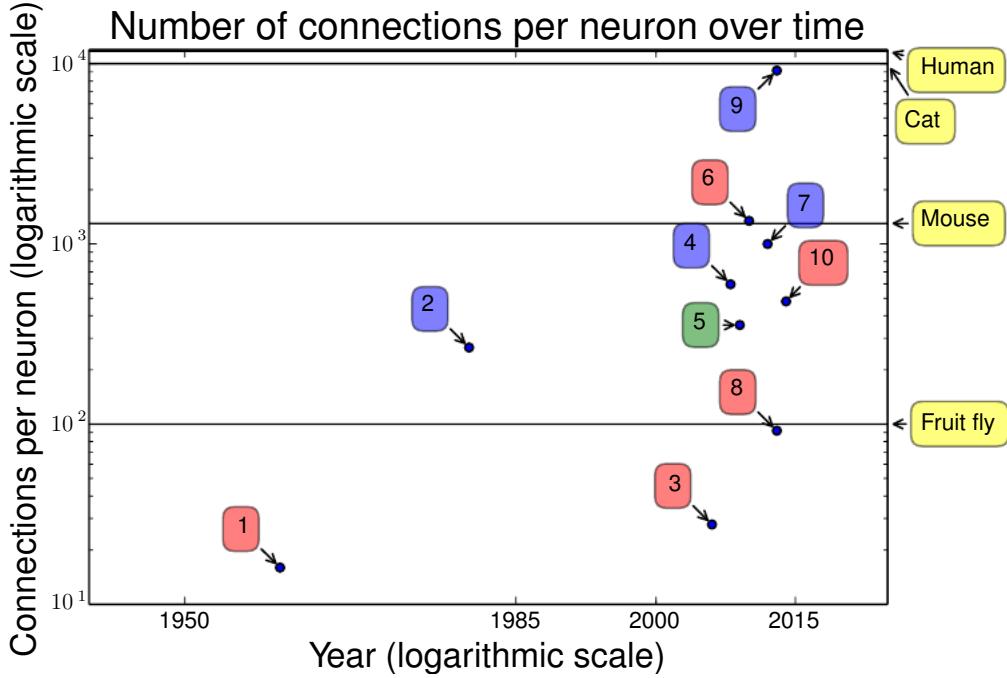


Figure 1.6: Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, 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., 2009a](#))
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., 2014](#))

1998a; Bengio *et al.*, 2001a).

Research groups led by Geoffrey Hinton at University of Toronto, Yoshua Bengio at University of Montreal, and Yann LeCun at New York University re-popularized neural networks re-branded as “deep learning” beginning in 2006. At this time, it was believed that the primary difficulty in using deep learning was optimizing the non-convex functions involved in neural network training. Until approximately 2012, most research in deep learning focused on using unsupervised learning to “pretrain” each layer of the network in isolation, so that the final supervised training stage would not need to modify the network’s parameters greatly (Hinton *et al.*, 2006; Bengio *et al.*, 2007; Ranzato *et al.*, 2007).

Since 2012, the greatest successes in deep learning have come not from this layerwise pretraining scheme but simply from applying traditional supervised learning techniques to large models on large datasets. TODO something about piecewise linear units and dropout

TODO: scan through this section above, find terms that haven’t been introduced well, like convexity, unsupervised pretraining, etc., and introduce them appropriately

TODO: get this idea in somewhere Deep architectures had been proposed before (Fukushima, 1980; LeCun *et al.*, 1989; Schmidhuber, 1992; Utgoff and Stracuzzi, 2002) but without major success to jointly train a deep neural network with many layers except to some extent in the case of convolutional architectures (LeCun *et al.*, 1989, 1998c)

TODO neocognitron

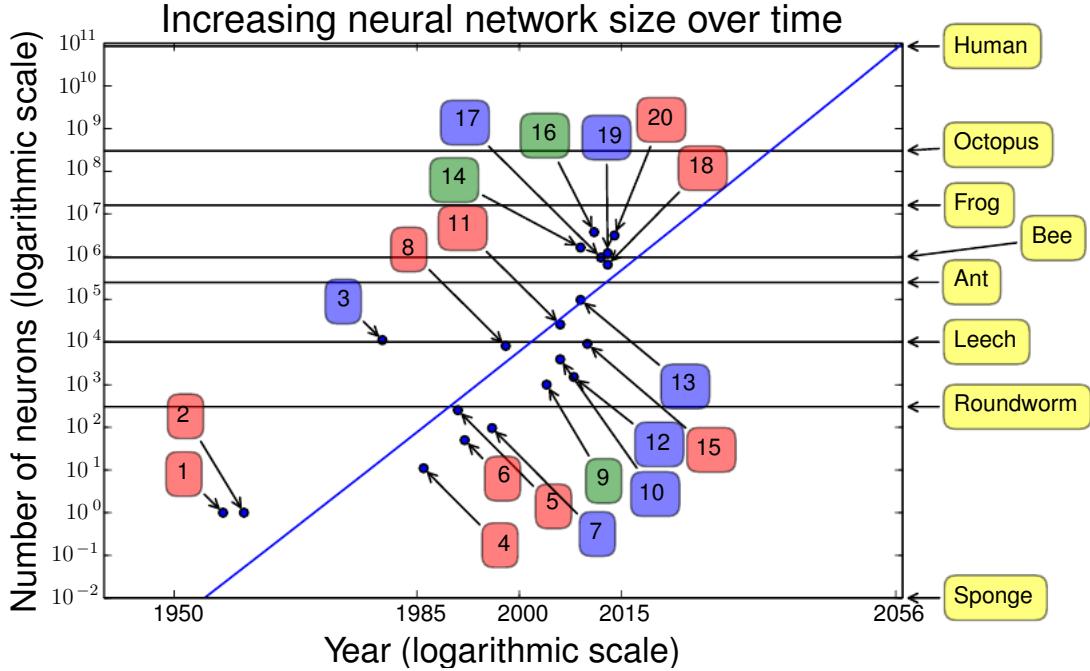


Figure 1.7: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. This growth is driven primarily by faster computers with larger memory, but also by the availability of larger datasets. These larger networks are able to achieve higher accuracy on more complex tasks. This trend looks set to continue for many years. Unless new technologies allow faster scaling, artificial neural networks will not have the same number of neurons as the human brain until 2056. Real biological neurons are likely to represent more complicated functions than current artificial neurons, so biological neural networks may be even larger than this plot portrays. 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.*, 2009a](#))
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.*, 2014](#))

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 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 deep learning and working with 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 get by, but we highly recommend that you also consult another resource focused exclusively on teaching linear algebra, such as ([Shilov, 1977](#)).

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 \mathbf{x} . The elements of the vector are identified by writing its name in italic typeface, with a subscript. The first element of \mathbf{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:

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

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 \mathbf{x}_S . We use the $-$ sign to index the complement of a set. For example \mathbf{x}_{-1} is the vector containing all elements of \mathbf{x} except for x_1 , and \mathbf{x}_{-S} is the vector containing all of the elements of \mathbf{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 \mathbf{A} . If a real-valued matrix \mathbf{A} has a height of m and a width of n , then we say that $\mathbf{A} \in \mathbb{R}^{m \times n}$. 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 \mathbf{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, $\mathbf{A}_{:,i}$ denotes the horizontal cross section of \mathbf{A} with vertical coordinate i . This is known as the i -th *row* of \mathbf{A} . Likewise, $\mathbf{A}_{i,:}$ is the i -th *column* of \mathbf{A} . When we need to explicitly identify the elements of a matrix, we write them as an array enclosed in square brackets:

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

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, $f(\mathbf{A})_{i,j}$ gives element (i,j) of the matrix computed by applying the function f to \mathbf{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*.

By convention, we consider that adding (or subtracting) a scalar and a vector yields a vector with the additive operation performed on each element. The same thing happens

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

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

with a matrix or a tensor. This is called a *broadcasting* operation in Python’s `numpy` library.

One important operation on matrices is the *transpose*. The transpose of a matrix is the mirror image of the matrix across a diagonal line 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 A as A^\top , and it is defined such that

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

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. $\mathbf{x} = [x_1, x_2, x_3]^\top$.

We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements: $C = A + B$ 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: $D = a \cdot B + c$ 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 A and B is a third matrix C . In order for this product to be defined, A must have the same number of columns as B has rows. If A is of shape $m \times n$ and B is of shape $n \times p$, then C is of shape $m \times p$. We can write the matrix product just by placing two or more matrices together, e.g.

$$C = AB.$$

The product operation is defined by

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

$$\sum_{22}$$

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 book¹ as $\mathbf{A} \odot \mathbf{B}$.

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

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

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

It is also associative:

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

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

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

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

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 can also multiply matrices and vectors by scalars. To multiply by a scalar, we just multiply every element of the matrix or vector by that scalar:

$$s\mathbf{A} = \begin{bmatrix} sa_{1,1} & \dots & sa_{1,n} \\ \vdots & \ddots & \vdots \\ sa_{m,1} & \dots & sa_{m,n} \end{bmatrix}$$

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

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

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

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

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

...

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

¹The 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 \mathbf{I}_3 .

or, even more explicitly, as:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\dots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m. \end{aligned}$$

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 \mathbf{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 \mathbf{I}_n . Formally,

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

The structure of the identity matrix is simple: all of the entries along the *main diagonal* (where the row coordinate is the same as the column coordinate) are 1, while all of the other entries are zero. See Fig. 2.2 for an example.

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

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

We can now solve equation 2.1 by the following steps:

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{A}^{-1} \mathbf{A}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{I}_n \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}. \end{aligned}$$

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

When \mathbf{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 \mathbf{b} . However, \mathbf{A}^{-1} is primarily useful as a theoretical tool, and should not actually be used in practice for most software applications. Because \mathbf{A}^{-1} can only be represented with limited precision on a digital computer, algorithms that make use of the value of \mathbf{b} can usually obtain more accurate estimates of \mathbf{x} .

2.4 Linear Dependence, Span, and Rank

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

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

is also a solution for any real α .

To analyze how many solutions the equation has, we can think of the columns of \mathbf{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 \mathbf{b} . In this view, each element of \mathbf{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 :

$$\mathbf{Ax} = \sum_i x_i \mathbf{A}_{:,i}.$$

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

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

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

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

In order for the system $\mathbf{Ax} = \mathbf{b}$ to have a solution for all values of $\mathbf{b} \in \mathbb{R}^m$, we therefore require that the column space of \mathbf{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 \mathbf{b} that has no solution. This implies immediately that \mathbf{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 3×2 matrix. The target \mathbf{b} is 3-D, but \mathbf{x} is only 2-D, so modifying the value of \mathbf{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 \mathbf{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 2×2 matrix where both of the columns are equal to each other. This has the same column space as a 2×1 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 have at least m linearly independent columns. This condition is both necessary and sufficient for equation 2.1 to have a solution for every value of \mathbf{b} .

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 \mathbf{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 \mathbf{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:

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

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

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:

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

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:

- $f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0}$

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

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 \mathbf{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 $\mathbf{x}^\top \mathbf{x}$.

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 \mathbf{x} each depend only on the corresponding element of \mathbf{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

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

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 \mathbf{x} moves away from 0 by ϵ , the L^1 norm increases by ϵ .

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 α does not change the number of nonzero entries.

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

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

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,

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

where θ is the angle between \mathbf{x} and \mathbf{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 \mathbf{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 book², we write $\text{diag}(\mathbf{v})$ to denote a square diagonal matrix whose diagonal entries are given by the entries of the vector \mathbf{v} . Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient. To compute $\text{diag}(\mathbf{v})\mathbf{x}$, we only need to scale each element x_i by v_i . In other words, $\text{diag}(\mathbf{v})\mathbf{x} = \mathbf{v} \odot \mathbf{x}$. Inverting a diagonal matrix is also efficient. The inverse exists only if every diagonal entry is nonzero, and in that case, $\text{diag}(\mathbf{v})^{-1} = \text{diag}([1/v_1, \dots, 1/v_n]^\top)$. 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.

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

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

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 \mathbf{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*:

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

A vector \mathbf{x} and a vector \mathbf{y} are *orthogonal* to each other if $\mathbf{x}^\top \mathbf{y} = 0$. 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:

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

This implies that

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

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.

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

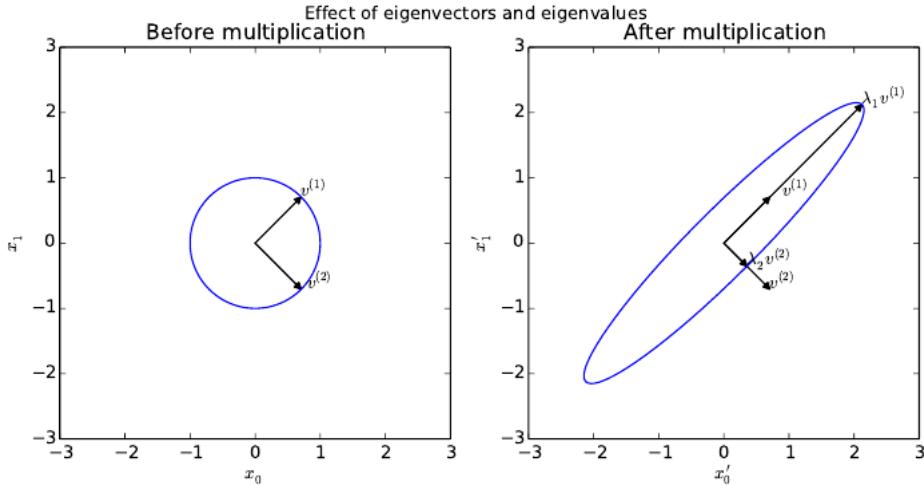


Figure 2.3: An example of the effect of eigenvectors and eigenvalues. Here, we have a matrix \mathbf{A} with two orthonormal eigenvectors, $\mathbf{v}^{(1)}$ with eigenvalue λ_1 and $\mathbf{v}^{(2)}$ with eigenvalue λ_2 . *Left)* We plot the set of all unit vectors $\mathbf{u} \in \mathbb{R}^2$ as a blue circle. *Right)* We plot the set of all points \mathbf{Au} . By observing the way that \mathbf{A} distorts the unit circle, we can see that it scales space in direction $\mathbf{v}^{(i)}$ by λ_i .

2.7 Eigendecomposition

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

$$\mathbf{Av} = \lambda\mathbf{v}.$$

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

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

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

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

has the desired eigenvalues and eigenvectors. If we make \mathbf{V} an orthogonal matrix, then we can think of \mathbf{A} as scaling space by λ_i in direction $\mathbf{v}^{(i)}$. 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.

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:

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

where \mathbf{Q} is an orthogonal matrix composed of eigenvectors of \mathbf{A} , and Λ is a diagonal matrix, with $\lambda_{i,i}$ being the eigenvalue corresponding to $\mathbf{Q}_{:,i}$.

While any real symmetric matrix \mathbf{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 \mathbf{Q} using those eigenvectors instead. By convention, we usually sort the entries of Λ 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 $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A} \mathbf{x}$ subject to $\|\mathbf{x}\|_2 = 1$. Whenever \mathbf{x} is equal to an eigenvector of \mathbf{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 $\forall \mathbf{x}, \mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0$. Positive definite matrices additionally guarantee that $\mathbf{x}^\top \mathbf{A} \mathbf{x} = 0 \Rightarrow \mathbf{x} = \mathbf{0}$.

2.8 Singular Value Decomposition

The singular value decomposition is a general purpose matrix factorization method that decomposes an $n \times m$ matrix \mathbf{X} into three distinct matrices: $\mathbf{X} = \mathbf{U}\Sigma\mathbf{W}^\top$, where Σ is a rectangular diagonal matrix³, \mathbf{U} is an $n \times n$ -dimensional matrix whose columns are mutually orthonormal and similarly \mathbf{W} is an $m \times m$ -dimensional matrix whose columns are mutually orthonormal. The elements along the diagonal of Σ , σ_i for $i \in \{1, \dots, \min\{n, m\}\}$, are referred to as the singular values and the n columns of

³ A rectangular $n \times m$ -dimensional diagonal matrix can be seen as consisting of two sub-matrices, one square diagonal matrix of size $c \times c$ where $c = \min\{n, m\}$ and a matrix of zeros to fill out the rest of the rectangular matrix.

\mathbf{U} and the m columns of \mathbf{W} are commonly referred to as the *left-singular vectors* and *right-singular vectors* of \mathbf{X} respectively.

The singular value decomposition has many useful properties, however for our purposes, we will concentrate on their relation to the eigen-decomposition. Specifically, the left-singular vectors of \mathbf{X} (the columns $\mathbf{U}_{:,i}$ for $i \in \{1, \dots, n\}$) are the eigenvectors of $\mathbf{X}\mathbf{X}^\top$. The right-singular vectors of \mathbf{X} (the columns $\mathbf{W}_{:,j}$ for $j \in \{1, \dots, m\}$) are the eigenvectors of $\mathbf{X}^\top\mathbf{X}$. Finally, the non-zero singular values of \mathbf{X} (σ_i : for all i such that $\sigma_i \neq 0$) are the square roots of the non-zero eigenvalues for both $\mathbf{X}\mathbf{X}^\top$ and $\mathbf{X}^\top\mathbf{X}$.

2.9 The Trace Operator

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

$$\text{Tr}(\mathbf{A}) = \sum_i a_{i,i}.$$

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:

$$\|\mathbf{A}\|_F = \sqrt{\text{Tr}(\mathbf{A}^\top\mathbf{A})}.$$

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:

$$\text{Tr}(\mathbf{A}) = \text{Tr}(\mathbf{A}^\top).$$

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

$$\text{Tr}(\mathbf{ABC}) = \text{Tr}(\mathbf{BCA}) = \text{Tr}(\mathbf{CAB})$$

or more generally,

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

2.10 Determinant

The determinant of a square matrix, denoted $\det(\mathbf{A})$, 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.

2.11 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 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 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 $\mathbf{x}^{(i)} \in \mathbb{R}^n$ we will find a corresponding code vector $\mathbf{c}^{(i)} \in \mathbb{R}^l$. If l is smaller than n , it will take less memory to store the code points than the original data. We can use matrix multiplication to map the code back into \mathbb{R}^n . Let the reconstruction $r(\mathbf{c}) = \mathbf{D}\mathbf{c}$, where $\mathbf{D} \in \mathbb{R}^{n \times l}$ is the matrix defining the decoding.

To simplify the computation of the optimal encoding, we constrain the columns of \mathbf{D} to be orthogonal to each other. (Note that \mathbf{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 $\mathbf{D}_{:,i}$ if we decrease c_i proportionally for all points. To give the problem a unique solution, we constrain all of the columns of \mathbf{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 \mathbf{c}^* for each input point \mathbf{x} . One way to do this is to minimize the distance between the input point \mathbf{x} and its reconstruction, $r(\mathbf{c})$. We can measure this distance using a norm. In the principal components algorithm, we use the L^2 norm:

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

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 \mathbf{c} . This is because the L^2 norm is non-negative and the squaring operation is monotonically increasing for non-negative arguments.

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

The function being minimized simplifies to

$$(\mathbf{x} - r(\mathbf{c}))^\top (\mathbf{x} - r(\mathbf{c}))$$

(by the definition of the L^2 norm)

$$= \mathbf{x}^\top \mathbf{x} - \mathbf{x}^\top r(\mathbf{c}) - r(\mathbf{c})^\top \mathbf{x} + r(\mathbf{c})^\top r(\mathbf{c})$$

(by the distributive property)

$$= \mathbf{x}^\top \mathbf{x} - 2\mathbf{x}^\top r(\mathbf{c}) + r(\mathbf{c})^\top r(\mathbf{c})$$

(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 \mathbf{c} :

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

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

$$\begin{aligned} \mathbf{c}^* &= \arg \min_{\mathbf{c}} -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{D}^\top \mathbf{D}\mathbf{c} \\ &= \arg \min_{\mathbf{c}} -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{I}_l \mathbf{c} \\ & \quad (\text{by the orthogonality and unit norm constraints on } \mathbf{D}) \\ &= \arg \min_{\mathbf{c}} -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c} \end{aligned}$$

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

$$\begin{aligned} \nabla_{\mathbf{c}}(-2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c}) &= \mathbf{0} \\ -2\mathbf{D}^\top \mathbf{x} + 2\mathbf{c} &= \mathbf{0} \\ \mathbf{c} &= \mathbf{D}^\top \mathbf{x}. \end{aligned} \tag{2.2}$$

This is good news: we can optimally encode \mathbf{x} just using a vector-product operation.

Next, we need to choose the encoding matrix \mathbf{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 \mathbf{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:

$$\mathbf{D}^* = \arg \min_{\mathbf{D}} \sqrt{\sum_{i,j} (x_j^{(i)} - r_j^{(i)})^2} \text{ subject to } \mathbf{D}^\top \mathbf{D} = \mathbf{I}_l \tag{2.3}$$

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

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

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

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

$$= \arg \min_{\mathbf{d}} \text{Tr} \left(\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top \right)^\top \left(\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top \right) \text{ subject to } \mathbf{d}^\top \mathbf{d} = 1$$

(by the alternate definition of the Frobenius norm)

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

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

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

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

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

(because the trace is invariant to transpose)

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

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

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

(due to the constraint)

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

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

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

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

In the general case, where $l > 1$, \mathbf{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 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 argu-

ments presented here are summarized from or inspired by (Pearl, 1988). Much earlier work in probability and engineering introduced and developed the underlying fundamental notions, such as the notion of exchangeability (de Finetti, 1937), Cox’s theorem as the foundations of Bayesian inference (Cox, 1946), and the theory of stochastic processes (Doob, 1953).

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.

One source of uncertainty is incomplete observability. When we cannot observe something, we are uncertain about its true nature. In machine learning, it is often the case that we can observe a large amount of data, but there is not a data instance for every situation we care about. We are also generally not able to observe directly what process generates the data. Since we are uncertain about what process generates the data, we are also uncertain about what happens in the situations for which we have not observed data points. Lack of observability can also give rise to apparent stochasticity. Deterministic systems can appear stochastic when we cannot observe all of the variables that drive the behavior of the system. For example, consider a game of Russian roulette. The outcome is deterministic if you know which chamber of the revolver is loaded. If you do not know this important information, then it is a game of chance. In many cases, we are able to observe some quantity, but our measurement is itself uncertain. For example, laser range finders may have several centimeters of random error.

Uncertainty can also arise from the simplifications we make in order to model real-world processes. For example, if we discretize space, then we immediately become uncertain about the precise position of objects: each object could be anywhere within the discrete cell that we know it occupies.

Conceivably, the universe itself could have stochastic dynamics, but we make no claim on this subject.

In many cases, it is more practical to use a simple but uncertain rule rather than a complex but certain one, even if 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, and 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 an uppercase script letter, 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 x can take on. 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; $P(x)$ is usually not the same as $P(y)$.

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 $x = x$, with a probability of 1 indicating that $x = x$ is certain and a probability of 0 indicating that $x = x$ is impossible. Sometimes to disambiguate which PMF to use, we write the name of the random variable explicitly: $P(x = x)$. Sometimes we define a variable first, then use \sim notation to specify which distribution it follows later: $x \sim P(x)$.

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*. $P(x = x, y = y)$ denotes the probability that $x = x$ and $y = y$ 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 f must meet the following properties:

- The domain of f must be the set of all possible states of x .
- The range of f must be a subset of the real interval $[0, 1]$. (No state can be more likely than a guaranteed event or less likely than an impossible event.)
- $\sum_{x \in x} f(x) = 1$. (f 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

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

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 $\sum_i P(x = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1$, 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. A probability density function must satisfy the following properties:

- It must map from the domain of the random variable whose distribution it describes to the real numbers.

- $\forall x, p(x) \geq 0$. Note that we do not require $p(x) \leq 1$.
- $\int p(x)dx = 1$.

A probability density function does not give the probability of a specific state directly, instead the probability of landing inside an infinitesimal region with volume δx 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 $\int_{[a,b]} p(x)dx$.

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*.

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

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

The name “marginal probability” comes from the process of computing marginal probabilities on paper. When the values of $P(x, y)$ 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 $y = y$ given $x = x$ as $P(y = y \mid x = x)$. This conditional probability can be computed with the formula

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

Note that this is only defined when $P(x = x) > 0$. 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:

$$P(x^{(1)}, \dots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^n P(x^{(i)} \mid x^{(1)}, \dots, x^{(i-1)}).$$

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

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

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:

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

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 :

$$\forall x \in \mathcal{X}, y \in \mathcal{Y}, p(x = x, y = y) = p(x = x)p(y = y).$$

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 :

$$\forall x \in \mathbf{x}, y \in \mathbf{y}, z \in \mathbf{z}, p(\mathbf{x} = x, \mathbf{y} = y | \mathbf{z} = z) = p(\mathbf{x} = x | \mathbf{z} = z)p(\mathbf{y} = y | \mathbf{z} = z).$$

We can denote independence and conditional independence with compact notation: $\mathbf{x} \perp \mathbf{y}$ means that x and y are independent, while $\mathbf{x} \perp \mathbf{y} | \mathbf{z}$ 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 $P(x)$ 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. $\mathbb{E}_x[f(x)]$. If it is clear which random variable the expectation is over, we may omit the subscript entirely, e.g. $\mathbb{E}[f(x)]$. Likewise, when there is no ambiguity, we may omit the square brackets.

Expectations are linear, for example, $\mathbb{E}[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}[f(x)] + \beta \mathbb{E}[g(x)]$.

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

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

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:

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

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 concepts of covariance and dependence are conceptually related, but are in fact distinct concepts. Two random variables have non-zero covariance only if they

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 $\mathbb{E}[x] + s(x - \mathbb{E}[x])$. Clearly, x and y are not independent, because y only has two possible values given x . However, $\text{Cov}(x, y) = 0$.

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

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

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

3.9 Information Theory

Information theory is a branch of applied mathematics that revolves around quantifying how much information is present in a signal. 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.
- Unlikely events should have high information content
- If one event is half as likely as another, then learning about the former event should convey twice as much information as the latter.

In order to satisfy all three of these properties, we define the *self-information* of an event $\mathbf{x} = x$ to be $I(x) = -\log P(x)$. 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.

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:

$$H(\mathbf{x}) = \mathbb{E}_{x \sim P}[I(x)].$$

¹Shannon 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](#)).

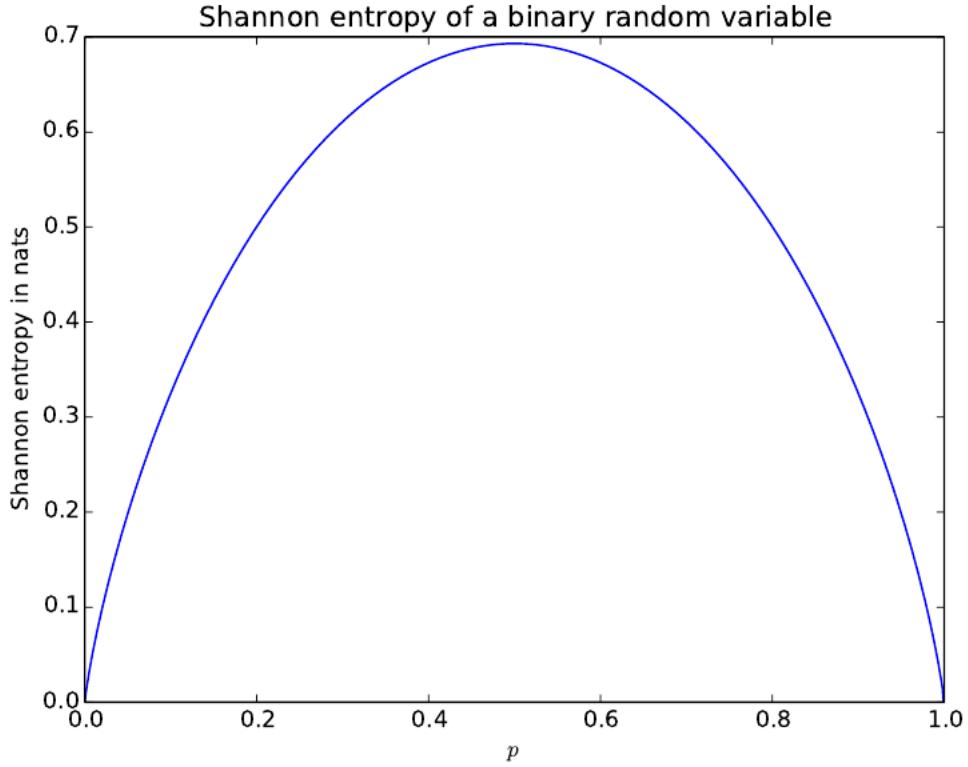


Figure 3.1: This plot shows how distributions that are closer to deterministic have low Shannon entropy while distributions that are close to uniform have high Shannon entropy. On the horizontal axis, we plot p , the probability of a binary random variable being equal to 1. When p is near 0, the distribution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

In other words, the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. Distributions that are nearly deterministic have low entropy; distributions that are closer to uniform have high entropy. See Fig. 3.1 for a demonstration.

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

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

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. $D_{\text{KL}}(P\|Q) \neq D_{\text{KL}}(Q\|P)$ for some P and Q .

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 $\lim_{x \rightarrow 0} x \log x = 0$.

Many information theoretic concepts can be interpreted in terms of the number of bits required to send certain messages. While these interpretations are valuable in other fields, they have not usually been of much practical importance in the context of deep learning.

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:

$$\begin{aligned} P(x = 1) &= \phi \\ P(x = 0) &= 1 - \phi \\ P(x = x) &= \phi^x (1 - \phi)^{1-x} \\ \mathbb{E}_x[x] &= \phi \\ \text{Var}_x(x) &= \phi(1 - \phi) \\ H(x) &= (\phi - 1) \log(1 - \phi) - \phi \log \phi. \end{aligned}$$

3.10.2 Multinoulli Distribution

The *Multinoulli* distribution is a distribution over a single discrete variable with k different states, where k is finite². The Multinoulli distribution is parametrized by a vector $\mathbf{p} \in [0, 1]^{k-1}$, where p_i gives the probability of the i -th state. The final, k -th state’s probability is given by $1 - \mathbf{1}^\top \mathbf{p}$. Note that we must constrain $\mathbf{1}^\top \mathbf{p} \leq 1$. Multinoulli distributions are often used to refer to distributions over categories of objects, so we do

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 $\{0, \dots, k\}^n$ 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.

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*:

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

See Fig. 3.2 for a schematic.

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

Note that when we evaluate the PDF, we need to square and invert σ . 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:

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

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 Σ :

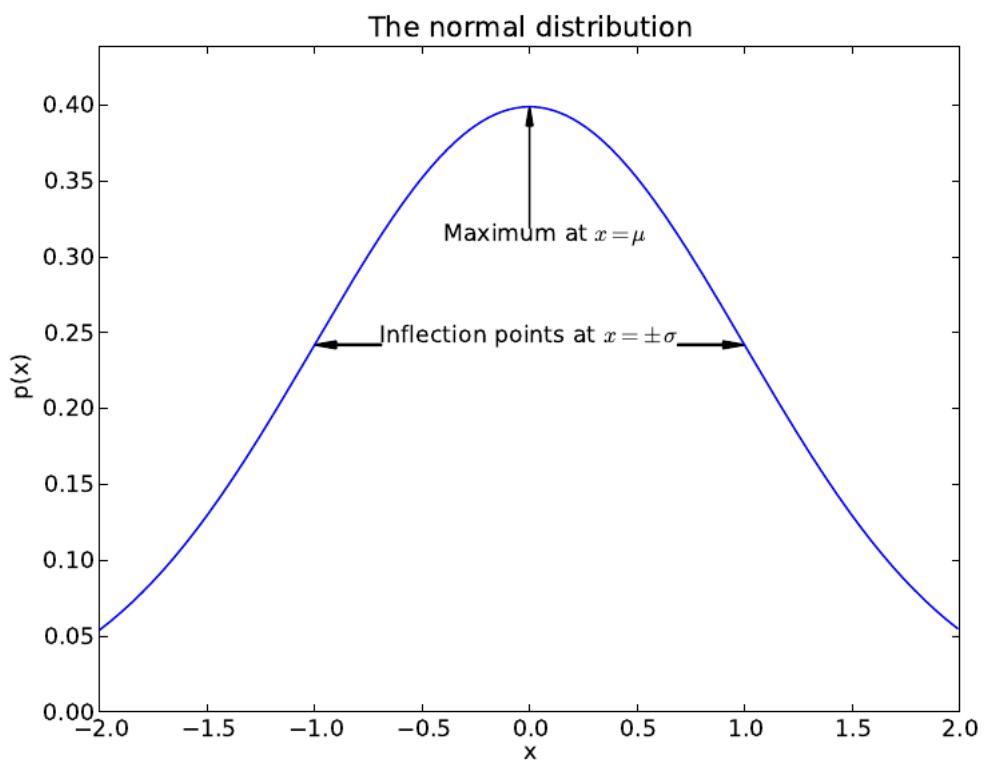


Figure 3.2: *The normal distribution*: The normal distribution $\mathcal{N}(x | \mu, \sigma^2)$ exhibits a classic “bell curve” shape, with the x coordinate of its central peak given by μ , and the width of its peak controlled by σ . In this example, we depict the *standard normal distribution*, with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

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

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* $\boldsymbol{\beta}$:

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

3.10.4 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 δ 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*,

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

which puts probability mass $\frac{1}{n}$ on each of the n points x_1, \dots, 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.

3.10.5 Mixtures of Distributions and Gaussian Mixture

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:

$$P(\mathbf{x}) = \sum_i P(\mathbf{c} = i)P(\mathbf{x} | \mathbf{c} = i)$$

where $P(\mathbf{c})$ is the Multinoulli distribution over component identities. In chapter 14, we explore the art of building complex probability distributions from simple ones in more detail. Note that we can think of the variable \mathbf{c} as a non-observed (or latent) random variable that is related to \mathbf{x} through their joint distribution $P(\mathbf{x}, \mathbf{c}) = P(\mathbf{x}|\mathbf{c})P(\mathbf{c})$. Latent variables are discussed further in Section 14.4.2.

A very powerful and common type of mixture model is the *Gaussian mixture* model, in which the components $P(\mathbf{x} | \mathbf{c} = i)$ are Gaussians, each with its mean μ_i and covariance Σ_i . Some mixtures can have more constraints, for example, the covariances could be shared across components, i.e., $\Sigma_i = \Sigma_j = \Sigma$, 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 ϕ parameter of a Bernoulli distribution because its range is $(0, 1)$, which lies within the valid range of values for the ϕ parameter. See Fig. 3.3 for a graph of the sigmoid function.

Another commonly encountered function is the *softplus* function:

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

The softplus function can be useful for producing the β or σ parameter of a normal distribution because its range is \mathbb{R}^+ . It also arises commonly when manipulating expressions involving sigmoids, as it is the primitive of the sigmoid, i.e., the integral from

The logistic sigmoid function

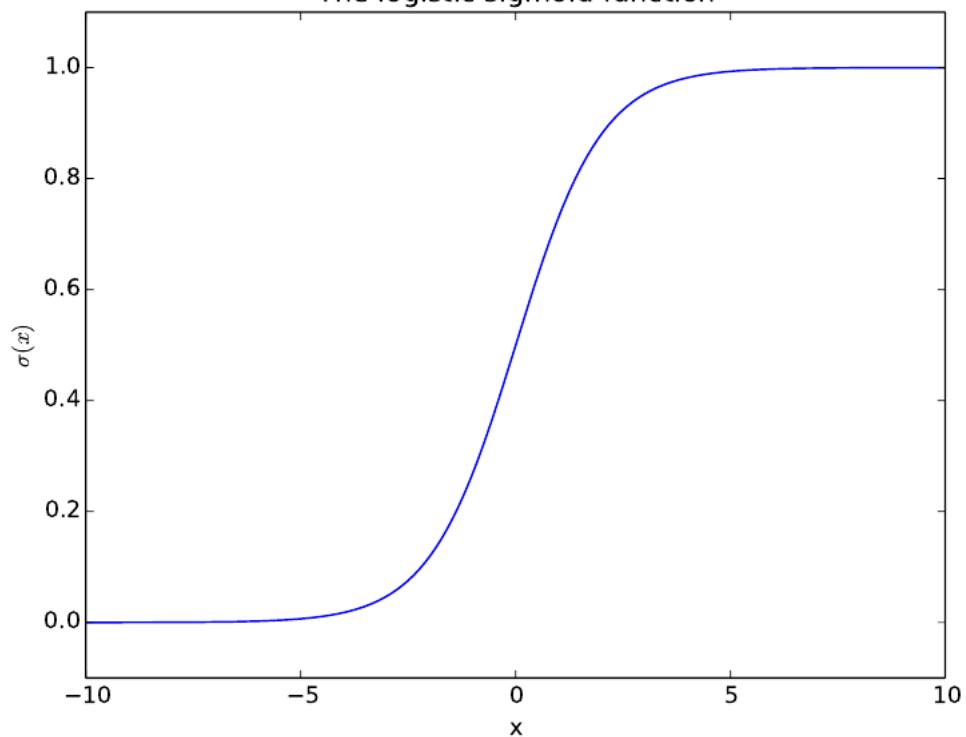


Figure 3.3: The logistic sigmoid function.

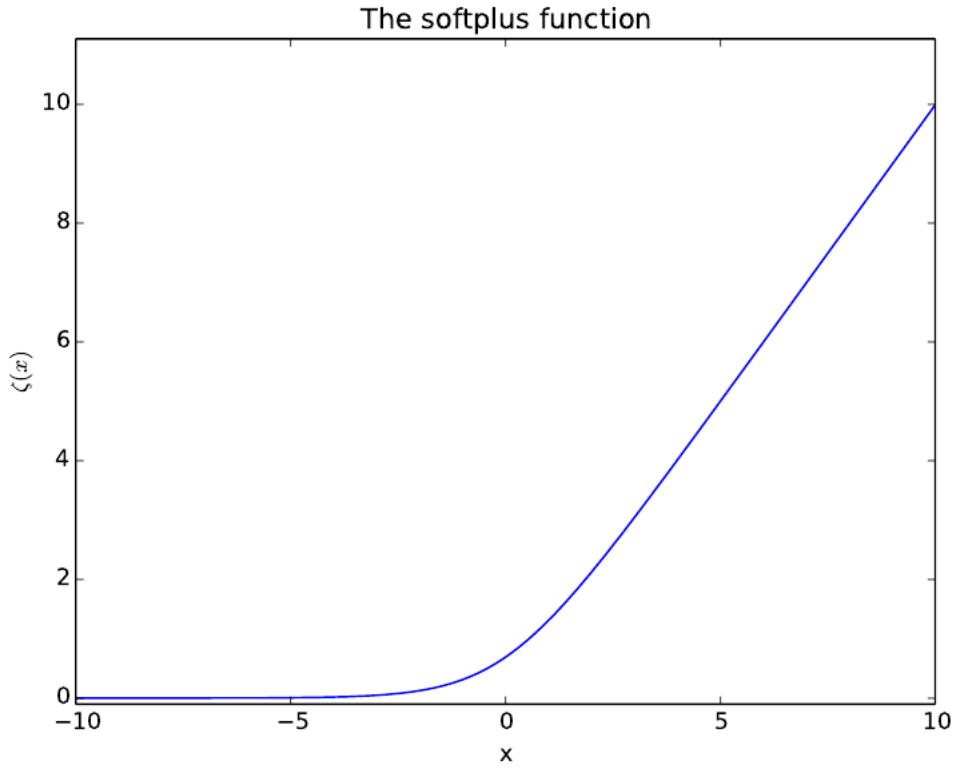


Figure 3.4: The softplus function.

$-\infty$ to x of the sigmoid. 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.

The following properties are all useful to have memorized:

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

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

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

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

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

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

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

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

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 $P(y | x)$ and need to know $P(x | y)$. Fortunately, if we also know $P(x)$, we can compute the desired quantity using *Bayes’ rule*:

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

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

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 \mathbf{x} lying in some set S is given by the integral of $p(\mathbf{x})$ 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 numbers³. 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.

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

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

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, \mathbf{x} and \mathbf{y} , such that $\mathbf{y} = g(\mathbf{x})$. You might think that $p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y}))$. This is actually not the case.

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

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

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 $g(\mathbf{x})$. Recall that the probability of \mathbf{x} lying in an infinitesimally small region with volume $d\mathbf{x}$ is given by $p_x(\mathbf{x})d\mathbf{x}$. Since g can expand or contract space, the infinitesimal volume surrounding \mathbf{x} in \mathbf{x} space may have different volume in \mathbf{y} space. To correct the problem, we need to preserve the property

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

Solving from this, we obtain

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

or equivalently

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x})) \left| \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \right|. \quad (3.1)$$

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 Example: Naive Bayes

We now know enough probability theory that we can derive some simple machine learning tools.

The *Naive Bayes* model is a simple probabilistic model that is often used to recognize patterns. The model consists of one random variable c representing a category, and a set of random variables $\mathbb{F} = \{f^{(1)}, \dots, f^{(n)}\}$ 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 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 $f^{(1)}$ 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 $f^{(2)} \in \mathbb{R}$ 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:

$$P(c, f^{(1)}, \dots, f^{(n)}) = P(c) \prod_i P(f^{(i)} | c).$$

These assumptions are very strong and unlikely to be true in practice, hence the name “naive”. Surprisingly, Naive Bayes often produces good predictions in practice, 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 $P(c)$ a Bernoulli distribution, with $P(c = c_1) = \phi^c$. We can also make $P(f^{(1)} | c)$ a Bernoulli distribution, with

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

In other words, the Bernoulli parameter changes depending on the value of c . Finally, we need to choose the distribution over $f^{(2)}$. Since $f^{(2)}$ is real-valued, a normal distribution is a good choice. Because $f^{(2)}$ 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 $f^{(1)}$, we need to use different parameters for different values of c , to represent that patients with the flu have different temperatures than patients without it:

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

Now we are ready to determine how likely a patient is to have the flu. To do this, we want to compute $P(c | \mathbb{F})$, but we know $P(c)$ and $P(\mathbb{F} | c)$. 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:

$$P(c | \mathbb{F}) = \frac{P(c)P(\mathbb{F} | c)}{P(\mathbb{F})}. \quad (3.2)$$

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

$$\begin{aligned} P(\mathbb{F}) &= \sum_{c \in \mathcal{C}} P(c = c, \mathbb{F}) \text{ (by the sum rule)} \\ &= \sum_{c \in \mathcal{C}} P(c = c)P(\mathbb{F} | c = c) \text{ (by the chain rule).} \end{aligned}$$

Substituting this result back into equation 3.2, we obtain

$$\begin{aligned} P(c | \mathbb{F}) &= \frac{P(c)P(\mathbb{F} | c)}{\sum_{c \in \mathcal{C}} P(c)P(\mathbb{F} | c = c)} \\ &= \frac{P(c)\prod_i P(f^{(i)} | c)}{\sum_{c \in \mathcal{C}} P(c)\prod_i P(f^{(i)} | c = c)} \end{aligned}$$

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:

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

where

$$g(c) = P(c = c)P(f^{(1)} = f_1 | c = c)P(f^{(2)} = f_2 | c = c).$$

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

$$\begin{aligned} P(c = 1 | f^{(1)} = f_1, f^{(2)} = f_2) &= \frac{g(1)}{g(0) + g(1)} \\ &= \frac{1}{1 + \frac{g(0)}{g(1)}} \\ &= \frac{1}{1 + \exp(\log g(0) - \log g(1))} \\ &= \sigma(\log g(1) - \log g(0)). \end{aligned} \tag{3.3}$$

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

$$\begin{aligned} \log g(i) &= \log \phi^{(c)i}(1 - \phi^{(c)})^{1-i}\phi_1^{(f)f_1}(1 - \phi_1^{(f)})^{1-f_1} \sqrt{\frac{1}{2\pi\sigma_i^2}} \exp\left(-\frac{1}{2\sigma_i^2}(f_2 - \mu_i)^2\right) \\ &= i \log \phi^{(c)} + (1-i) \log(1 - \phi^{(c)}) + f_1 \log \phi_i^{(f)} + (1-f_1) \log(1 - \phi_i^{(f)}) - \frac{1}{2} \log \frac{1}{2\pi\sigma_i^2} - \frac{1}{2\sigma_i^2}(f_2 - \mu_i)^2. \end{aligned}$$

Substituting this back into equation 3.3, we obtain

$$P(c = c | f^{(1)} = f_1, f^{(2)} = f_2) =$$

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

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 μ_1 , the average temperature of a flu patient.

Chapter 4

Numerical Computation

Machine learning algorithms usually require a high amount of numerical computation. This refers to algorithms that solve mathematical problems by methods that iteratively update estimates of the solution, rather than analytically deriving 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. If the result is later used as the denominator of a fraction, the computer encounters a divide-by-zero error rather than performing a valid floating point operation.

Another highly damaging form of numerical error is *overflow*. Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$. No more valid arithmetic can be done at this point.

For an example of the need to design software implementations to deal with overflow and underflow, consider the softmax function:

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

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 $\text{softmax}(\mathbf{z})$ where $\mathbf{z} = \mathbf{x} - \max_i x_i$. 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 $\log \text{softmax}(\mathbf{x})$ 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 softmax in a numerically stable way. The log 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.*, 2010; 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 perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

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

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

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.

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 $f(\mathbf{x})$ by altering \mathbf{x} . We usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\mathbf{x})$.

The function we want to minimize or maximize is called the *objective function*. When we are minimizing it, we may also call it the *cost function*, *loss function*, or *error function*.

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

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

The *derivative* of a function $y = f(x)$ from \mathbb{R} to \mathbb{R} , denoted as $f'(x)$ or $\frac{dy}{dx}$, gives the slope of $f(x)$ at the point x . It quantifies *how a small change in x gets scaled in order to obtain a corresponding change in y* . 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 $f(x - \epsilon \text{ sign}(f'(x)))$ is less than $f(x)$ for small enough ϵ . We can thus reduce $f(x)$ by moving x in small steps with opposite sign of the derivative. This technique is called gradient descent (Cauchy, 1847). 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 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. The figure caption also raises the question of whether local minima or saddle points and plateaus are more to blame for the difficulties one may encounter in training deep networks, a question that is discussed further in Chapter 8, in particular Section 8.2.3.

We often minimize functions that have multiple inputs: $f : \mathbb{R}^n \rightarrow \mathbb{R}$. 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 $\frac{\partial}{\partial x_i} f(\mathbf{x})$ measures how f changes as only the variable x_i increases at

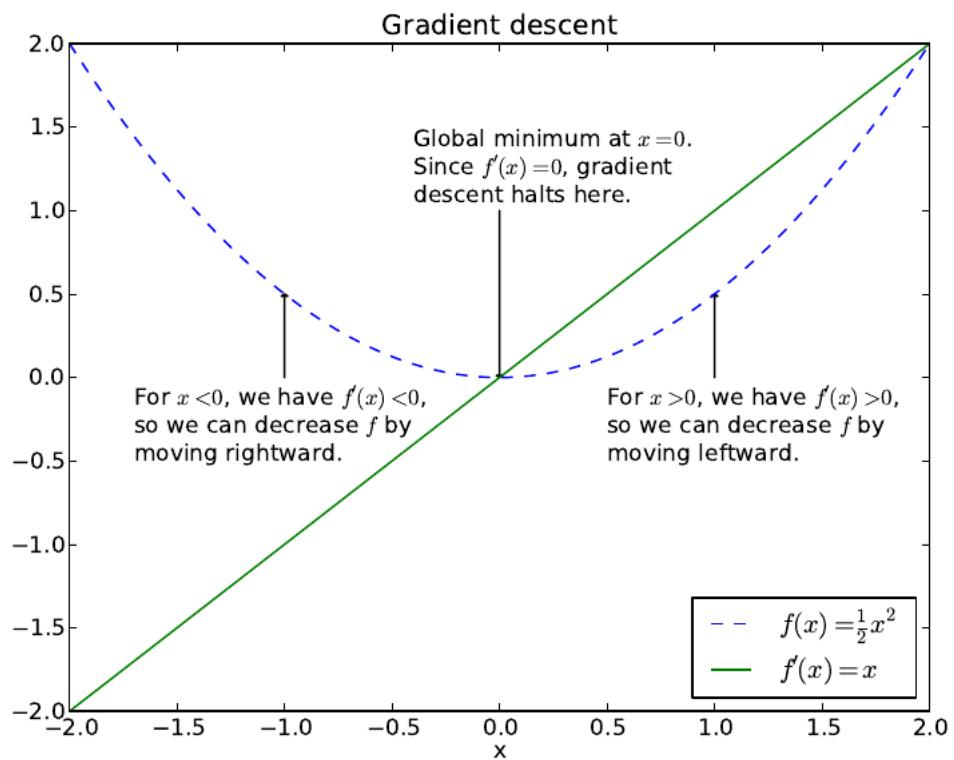


Figure 4.1: An illustration of how the derivatives of a function can be used to follow the function downhill to a minimum. This technique is called *gradient descent*.

Types of critical points

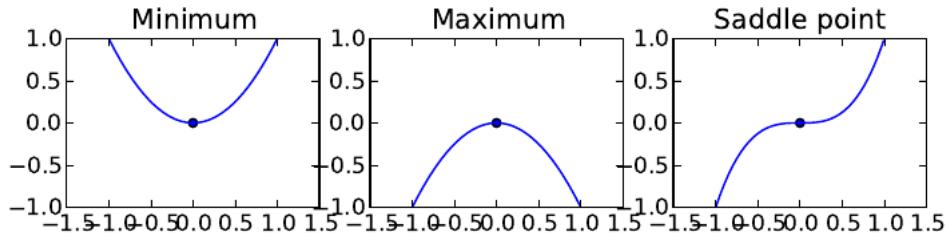


Figure 4.2: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors 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.

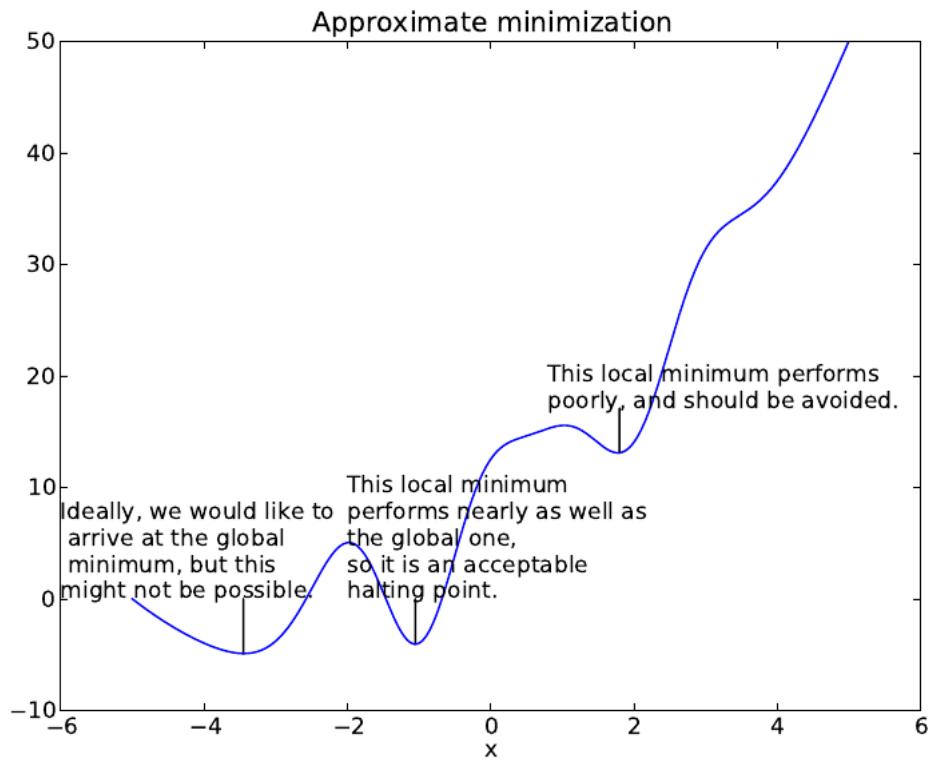


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function. Optimizing MLPs was believed to suffer from the presence of many local minima, but this idea is questioned in recent work (Dauphin *et al.*, 2014), with saddle points being considered as the more serious issue.

point \mathbf{x} . The *gradient* of f is the vector containing all of the partial derivatives, denoted $\nabla_{\mathbf{x}} f(\mathbf{x})$. 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 \mathbf{u} (a unit vector) is the slope of the function f in direction u . In other words, the derivative of the function $f(\mathbf{x} + \alpha \mathbf{u})$ with respect to α , evaluated at $\alpha = 0$. Using the chain rule, we can see that this $\mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$.

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

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

where θ is the angle between \mathbf{u} and the gradient. Substituting in $\|\mathbf{u}\|_2 = 1$ and ignoring factors that don't depend on \mathbf{u} , this simplifies to $\min_{\mathbf{u}} \cos \theta$. This is minimized when \mathbf{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

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

where ϵ is the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}))$ for several values of ϵ 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 $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$ for \mathbf{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 $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, then the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{n \times m}$ of \mathbf{f} is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a *second derivative*. For example, for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the derivative with respect to x_i of the derivative of f with respect to x_j is denoted as $\frac{\partial^2}{\partial x_i \partial x_j} f$. In a single dimension, we can denote $\frac{d^2}{dx^2} f$ 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 ϵ . 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 $\mathbf{H}(f)(\mathbf{x})$ is defined such that

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

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:

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

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 $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$, 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 definite¹, 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 definite², the point is a local maximum. In multiple dimensions, it is 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 \mathbf{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

¹all its eigenvalues are positive

²all its eigenvalues are negative

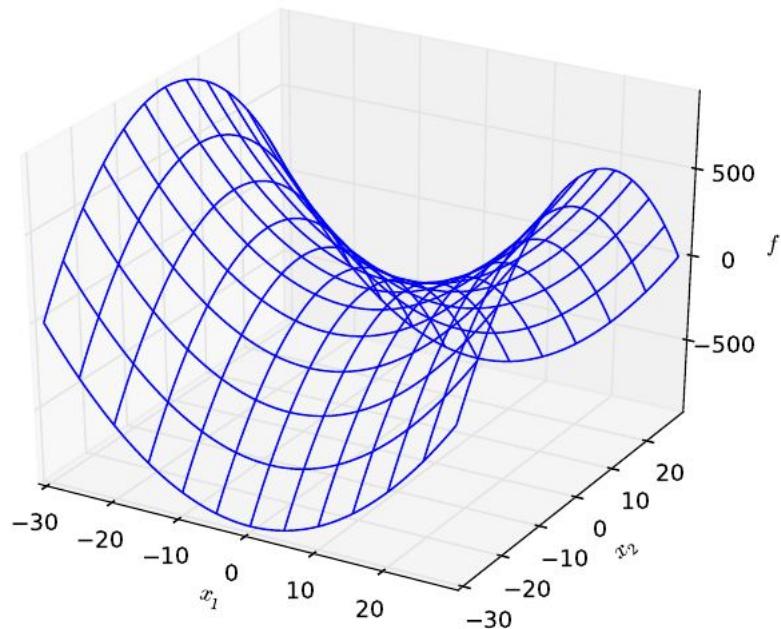


Figure 4.4: A saddle point containing both positive and negative curvature. The function in this example is $f(\mathbf{x}) = x_1^2 - x_2^2$. Along axis corresponding to x_1 , the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to x_2 , 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. See chapter 8.2.3 for a longer discussion of saddle points in deep nets.

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 $f(\mathbf{x})$ near some point \mathbf{x}_0 :

$$f(\mathbf{x}) = f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^\top \nabla_{\mathbf{x}} f(\mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top H(f)(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0).$$

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

$$\mathbf{x}^* = \mathbf{x}_0 - H(f)(\mathbf{x}_0)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0).$$

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 that gradient descent may be fortunate enough to avoid.

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 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 \(2004\)](#).

4.4 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(\mathbf{x})$ over all possible values of \mathbf{x} . Instead we may wish to find the maximal or minimal value of $f(\mathbf{x})$ for

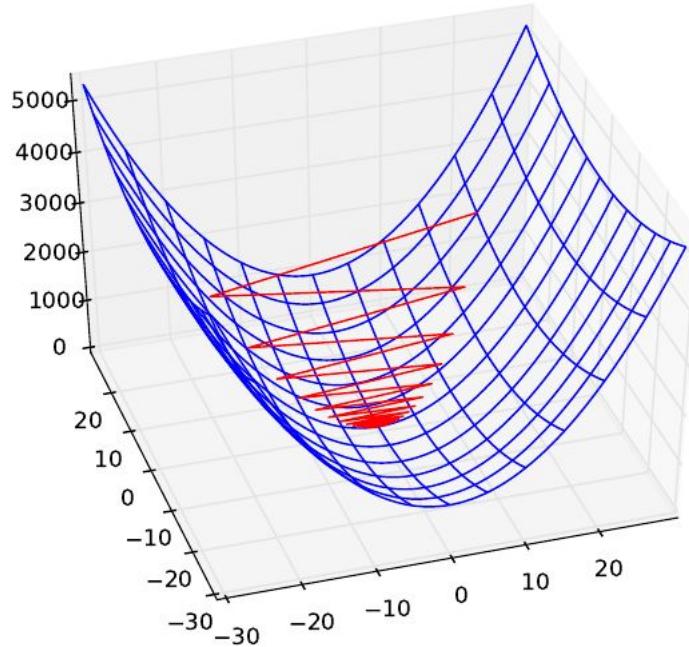


Figure 4.5: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent on a quadratic function whose Hessian matrix has condition number 5 (curvature is 5 times larger in one direction than in some other direction). The red lines indicate 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. Note that some recent results suggest that the above picture is not representative for deep highly non-linear networks. See Section 8.2.4 for more on this subject.

values of \mathbf{x} in some set S . This is known as *constrained optimization*. Points \mathbf{x} that lie within the set S are called *feasible* points in constrained optimization terminology.

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 ϵ , 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 ϵ that yield new \mathbf{x} points that are feasible, or we can project each point on the line back into the constraint region.

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 $f(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^2$ with \mathbf{x} constrained to have exactly unit L^2 norm, we can instead minimize $g(\theta) = f([\cos \theta, \sin \theta]^T)$ with respect to θ , 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_i and n functions h_j so that $S = \{\mathbf{x} \mid \forall i, g_i(\mathbf{x}) = 0 \text{ and } \forall j, h_j(\mathbf{x}) \leq 0\}$. The equations involving g_i are called the *equality constraints* and the inequalities involving h_j are called *inequality constraints*.

We introduce new variables λ_i and α_j for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

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

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 $f(\mathbf{x})$ is not permitted to have value ∞ , then

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

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

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

This follows because any time the constraints are satisfied,

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

while any time a constraint is violated,

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

³The KKT approach generalizes the method of *Lagrange multipliers* which only allows equality constraints

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 $-f(\mathbf{x})$, which leads to this optimization problem:

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

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

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

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 λ_i .

The inequality constraints are particularly interesting. We say that a constraint $h_i(\mathbf{x})$ is *active* if $h_i(\mathbf{x}^*) = 0$. 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 $\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ will have $\alpha_i = 0$. We can thus observe that at the solution, $\boldsymbol{\alpha} \circ \mathbf{h}(\mathbf{x}) = \mathbf{0}$. In other words, for all i , we know that at least one of the constraints $\alpha_i \geq 0$ and $h_i(\mathbf{x}) \leq 0$ 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 \mathbf{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 \mathbf{x} and the KKT multipliers are satisfied, and $\boldsymbol{\alpha} \circ \mathbf{h}(\mathbf{x}) = \mathbf{0}$ are called the *Karush-Kuhn-Tucker* (KKT) conditions TODO cite Karush, cite Kuhn and Tucker. 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 \mathbf{x} that minimizes

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

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:

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

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

Algorithm 4.1 An algorithm to minimize $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2$ with respect to \mathbf{x} using gradient descent.

Set ϵ , the step size, and δ , the tolerance, to small, positive numbers.

```

while  $\|\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}\|_2 > \delta$  do
     $\mathbf{x} \leftarrow \mathbf{x} - \epsilon (\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b})$ 
end while
```

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 $\mathbf{x}^\top \mathbf{x} \leq 1$. To do so, we introduce the Lagrangian

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

We can now solve the problem

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda).$$

The solution to the unconstrained least squares problem is given by $\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$. 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 \mathbf{x} , we obtain the equation

$$\mathbf{Ax} - \mathbf{b} + 2\lambda \mathbf{x} = 0.$$

This tells us that the solution will take the form

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

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

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

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

Chapter 5

Machine Learning Basics

TODO: no free lunch theorem

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. If you are already familiar with machine learning basics, feel free to skip to the end of this chapter, starting at Section 5.11. These sections cover basic machine learning notions having to do with smoothness, local vs non-local generalization and the curse of dimensionality, as these are not always discussed in basic courses in machine learning.

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.

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.

Many kinds of tasks can be solved with machine learning. This book describes these kinds of tasks:

- *Classification*: In this type of task, the algorithm is asked to output a function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$. Here $f(\mathbf{x})$ can be interpreted as an estimate of the category that \mathbf{x} belongs to. There are other variants of the classification task, for example, where f outputs a probability distribution over classes. An example of this type of 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 recognize objects and deliver them to people on command (Goodfellow *et al.*, 2010). Object recognition is also commercially interesting to search within a set of photos, to recognize faces in order to identify a person or unlock your phone, or for web sites that use targeted advertising based on the content of user-uploaded photos. Modern object recognition is best accomplished with deep learning (Krizhevsky *et al.*, 2012a; Zeiler and Fergus, 2014; Sermanet *et al.*, 2014).
- *Classification with missing inputs* : This is similar to classification, except rather than providing a single classification function, the algorithm must learn a set of functions. Each function corresponds to classifying \mathbf{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.
- *Regression* : In this type of task, the algorithm is asked to output a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. This type of task is similar to classification, except the machine learning system predicts a real-valued output instead of a category code. 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 used for algorithmic trading.
- *Transcription* : This type of task is similar to classification, except that the output is a sequence of symbols, rather than a category code (which can be thought of as a single symbol). For example, in speech recognition and in machine translation, the target output is a sequence of words. Such tasks fall under the more general umbrella of *structured output* tasks, where the target output is a complex compositional object involving many random variables (such as the individual words in the translated sentence) that have a non-trivial joint distribution, given the input.
- *Density estimation*: In this type of task, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{\text{model}}(\mathbf{x})$ can be interpreted as a probability density function 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. This can be used for anomaly detection (see below) but it can also be used in principle (at the price of some computation) to answer all questions of the form: what values are probable (or what are the expected values or other

such statistic) for some subset of the variables in \mathbf{x} , given another subset of these variables? This *generalizes all of the tasks presented here and many more* but may involve intractable computations called *inference*, discussed in Chapter 20. Density estimation is a form of *unsupervised learning*, and many other forms of unsupervised learning are possible, which can solve all or a subset of such tasks, provided appropriate inference is computationally feasible.

- *Anomaly detection:* In this type of task, the machine learning algorithm is to identify which events are normal and which events are unusual. This is closely related to the density estimation task, since unusual events are less likely. 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 presumably 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 unusual 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).
- *Imputation of missing values:* In this type of task, the machine learning algorithm is given a new example $\mathbf{x} \in \mathbb{R}^n$, but with some entries x_i of \mathbf{x} missing. The algorithm must provide a prediction of the values of the missing entries. This task is closely related to density estimation, because it can be solved by learning $p_{\text{model}}(\mathbf{x})$ then conditioning on the observed entries of \mathbf{x} . TODO: example application

Of course, many other tasks and types of tasks are possible.

5.1.2 The Performance Measure, P

In order to evaluate the performance 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. This is simply the proportion of examples for which the model produces the correct output. 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. We discuss this idea further in section 5.3.

In machine learning, the performance measure is sometimes called the *loss* which represent the cost associated with a particular event (such as a classification). The objective of learning is then to minimize the loss. For classification tasks, a common choice for the loss function is the *error* – which is simply the proportion of examples for which the model produces an incorrect output. Minimizing the error is, of course, equivalent to maximizing the accuracy.

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. This arises frequently in 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.

Finally, in machine learning, our goal is to adapt the model parameters to optimize the performance measure; however in many cases the relevant performance measure is not amenable to direct application in the optimization process. In order for a performance measure to be optimized directly, we almost always require a smooth signal – such as a gradient – to let us know how to move the model parameters in order to improve the performance measure. Many natural performance measures lack this property.

For example, consider again the use of classification accuracy as a measure of performance. Very small (infinitesimal) changes in the model parameters are unlikely to cause a change in the classification of any example. As a consequence, there is no local signal to tell us how to update the model parameters to improve model performance.

As a result, in place of the natural loss functions, we often use *surrogate* performance measures (also called *surrogate loss functions*) that are amenable for direct use as the *objective function* optimized with respect to the model parameters.

5.1.3 The Experience, E

In this book, the experience E usually consists of allowing the algorithm to observe a *dataset* containing several *examples*. This is a relatively simple kind of experience; other approaches to machine learning involve providing the algorithm with richer and more interactive experiences. For example, in *active learning* the algorithm may request specific additions to the dataset, and in *reinforcement learning* the algorithm continually interacts with an environment.

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. A famous example of a dataset that can be represented with a design matrix is the Iris dataset (Fisher, 1936). This dataset contains measurements of different parts of iris plants. The features measured are the sepal length, sepal width, petal length, and petal width. In total, the dataset describes 150 different individual iris plants. This means we can represent the dataset with a design matrix $\mathbf{X} \in \mathbb{R}^{150 \times 4}$, 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 width and height, 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 many cases, the example contains a *label* 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 \mathbf{X} , we also provide a vector of labels \mathbf{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 string of characters. These cases require more sophisticated data structures.

5.2 Example: Linear Regression

Let's begin with an example of a simple machine learning algorithm. One of the simplest machine learning algorithms is *linear regression*. As the name implies, this learning algorithm solves a regression problem. In other words, the goal is to build a system that can take a vector $\mathbf{x} \in \mathbb{R}^n$ 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

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

where $\mathbf{w} \in \mathbb{R}^n$ 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 \mathbf{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 \mathbf{x} by outputting $\hat{y} = \mathbf{w}^\top \mathbf{x}$. 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 $\mathbf{X}^{(\text{test})}$ and the vector of regression targets as $\mathbf{y}^{(\text{test})}$.

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

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

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

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

so the error increases whenever the Euclidean distance between the predictions and the targets increases. We will justify the use of this performance measure more formally in section 5.6.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights \mathbf{w} in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$. One intuitive way of doing this is just to minimize the mean squared error on the training set, $\text{MSE}_{\text{train}}$. We will justify this intuition later, in section 5.6.

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

$$\begin{aligned} \nabla_{\mathbf{w}} \text{MSE}_{\text{train}} &= 0 \\ \Rightarrow \nabla_{\mathbf{w}} \frac{1}{m} \|\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\|_2^2 &= 0 \\ \Rightarrow \frac{1}{m} \nabla_{\mathbf{w}} \|\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 &= 0 \end{aligned}$$

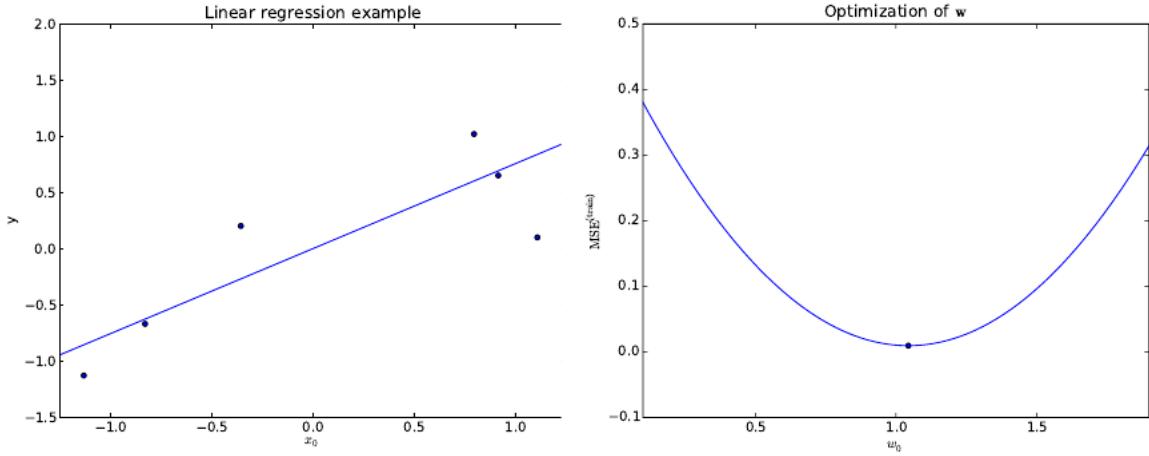


Figure 5.1: Consider this example linear regression problem, with a training set consisting of 5 data points, each containing one feature. This means that the weight vector \mathbf{w} contains only a single parameter to learn, w_0 . (*Left*) Observe that linear regression learns to set w_0 such that the line $y = w_0 x$ comes as close as possible to passing through all the training points. (*Right*) The plotted point indicates the value of w_0 found by the normal equations, which we can see minimizes the mean squared error on the training set.

$$\begin{aligned}
 & \Rightarrow \nabla_{\mathbf{w}} (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})})^\top (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}) = 0 \\
 & \Rightarrow \nabla_{\mathbf{w}} (\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})\top} \mathbf{y}^{(\text{train})}) = 0 \\
 & \Rightarrow 2\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} = 0 \\
 & \Rightarrow \mathbf{w} = (\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})})^{-1} \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})}
 \end{aligned} \tag{5.1}$$

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.

This 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

5.3.1 Generalization

In our example machine learning algorithm, linear regression, we fit the model by minimizing the squared error on the training set. However, what we actually care about is the performance of the model on new, previously unseen examples, such as the test

set. The expected error over all examples is known as the *generalization error*, and it is the quantity that machine learning algorithms aim to minimize. Generalization is the central objective of learning algorithms because for most problems of interest, the number of possible input configurations is huge, so that *new examples are almost surely going to be different from any of the training examples*. The learner has to generalize from the training examples to new cases.

Formally, generalization performance is typically defined as the *expected value of the chosen performance measure*, taken *over the probability distribution of interest*. For example, in the above regression example, we care about the expected squared prediction error. This probability distribution over which generalization performance is to be measured would be the probability distribution from which future test cases are supposed to arise. Hopefully, this is also the probability distribution from which training cases are obtained. Otherwise, it will be difficult to obtain theoretical guarantees about generalization. However, if some test examples from the target distribution are available, they could be used to assess future generalization performance.

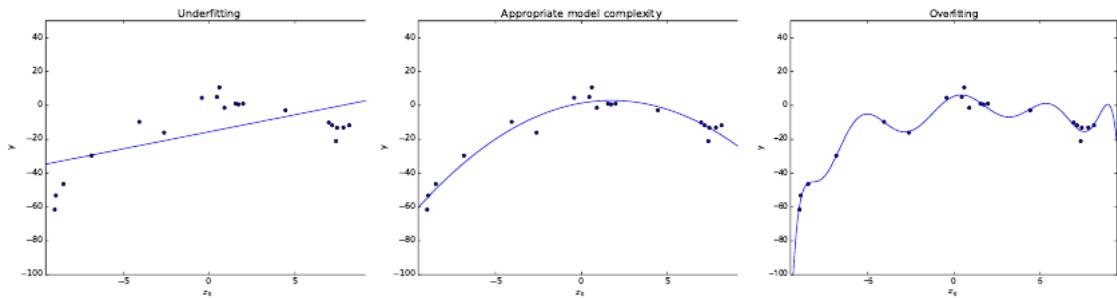


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by adding a small amount of noise to a quadratic function. (*Left*) A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (*Center*) 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 10 fit to the data suffers from overfitting. It is tightly fit to noise in the data and has inappropriately steep slope at the edges of the data.

5.3.2 Capacity

Capacity is a general term designating the ability of the learner to discover a function taken from a more or less rich family of function. For example, Figure 5.2 illustrates the use of three families of functions over a scalar input x : the linear (actually, affine) predictor (left of the figure),

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

the quadratic predictor (middle of the figure),

$$\hat{y} = b + w_1x + w_2x^2,$$

and the degree-10 polynomial predictor (right of the figure),

$$\hat{y} = b + \sum_{i=1}^{10} w_i x^i.$$

The latter family is clearly richer, allowing to capture more complex functions, but as discussed next, a richer family of functions or a larger capacity can yield to the problem of *overfitting* and poor generalization.

Many ways of defining the capacity of a model exist. One of the most popular is the *Vapnik-Chervonenkis dimension* or VC dimension (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). The VC dimension is defined for binary classifiers. The basic idea is to measure the size of the largest dataset on which the model is at least capable of representing a function that correctly classifies all training examples. Of course, it is easy to classify an infinitely large dataset correctly if all of the examples belong to the same class. We therefore require that the model be able to classify all of these training examples correctly regardless of the label that is assigned to them. Formally, the VC dimension of a binary classifier with decision function $f(\mathbf{x}; \boldsymbol{\theta})$ is the largest number m such that

$$\min_{\mathbf{X} \in \mathbb{R}^{m \times n}} \max_{y \in \{0,1\}^m} \min_{\boldsymbol{\theta}} \sum_{i=1}^m |f(\mathbf{X}[i,:]; \boldsymbol{\theta}) - y_i| = 0.$$

As discussed in many parts of this book, there are many ways to change the capacity of a learner. For example, with deep neural networks (Chapter 6), one can change the number of hidden units of each layer, the number of training iterations, or use regularization techniques (Chapter 7) to *reduce capacity*. In general, regularization is equivalent to *imposing a preference* over the set of functions that a learner can obtain as a solution. Such preferences can be thought of as a *prior probability distribution* over the space of functions or parameters that the learner can access. This point of view is particularly important according to the Bayesian statistics approach to machine learning, introduced in Section 5.7.

5.3.3 Occam’s Razor, Underfitting and Overfitting

A fundamental element of machine learning is the trade-off between capacity and generalization, and it has been the subject of a vast and mathematically grounded literature, also known as *statistical learning theory*, with some of the best known contributions coming from Vapnik and his collaborators, starting with the Vapnik-Chervonenkis dimension or VC dimension (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995), and with the many contributions reported at the COLT (COmputational Learning Theory) conference. This trade-off is related to a much older idea, *Occam’s razor* (c. 1287-1347), or principle of parsimony, which states that among competing hypotheses (here, read functions that could explain the observed data), one should choose the “simpler” one. Simplicity here is just another word for the opposite of “richness of the family of functions”, or capacity.

Machine learning algorithms can fail in two ways: *underfitting* and *overfitting*. **Underfitting** is simple to understand: it happens when the learner cannot find a solution that fits the observed data well enough. If we try to fit with a linear regression (x, y) pairs for which y is actually a quadratic function of x , then we are bound to have underfitting. There will be important aspects of the data that our learner cannot capture. If the learner needs a richer model in order to capture the data, it is underfitting. Sometimes, underfitting is not just due to the limitation imposed on the family of functions but also due to limitations on computational resources. For example, deep neural networks or recurrent neural networks can be very difficult to optimize (and in theory finding the global optimum of the training objective can be NP-hard). With a limited computational budget (e.g., a limited number of training iterations of an iterative optimization procedure), the learner may not be able to fit the training data well enough. It could also be that the optimization procedure easily gets stuck in local minima of the objective function and vastly more expensive optimization procedures are needed to really extract all the juice from the data and the theoretical richness of the family of functions in which the learner searches for a solution.

Overfitting is more subtle and not so easy to wrap one's mind around. What would be wrong with picking a very rich family of functions that is large enough to always fit our training data? We could certainly fit the training data all right, but *we might lose the ability to generalize well*. Why? The fundamental reason is that if the family of functions accessible to the learner (with its limitations on computational resources) is too large, then this family of functions may actually *contain many functions which all fit the training data well*. Without sufficient data, we would therefore not be able to distinguish which one was most appropriate, and the learner would make an arbitrary choice among these apparently good solutions. Yet, these functions are all different from each other, yielding different answers in at least some cases. Even if one of these functions was the correct one, we might pick the wrong one, and the expected generalization error would thus be at least proportional to the average discrepancy between these functions, which is related to the *variance of the estimator*, discussed below (Section 5.5). Occam's razor suggests to pick the family of functions just large enough to leave only one choice that fits well the data. Statistical learning theory has quantified this notion more precisely, and shown a mathematical relationship between capacity (e.g. measured by the VC-dimension) and the difference between generalization error and training error (also known as optimism). Although generalization error cannot generally be characterized exactly because it depends on the specifics of the training distribution, error bounds can be proven, 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).

Since training error decreases as a function of capacity, and adding a decreasing function (training error) and an increasing function (the difference between generalization and training error) yields a U-shaped function (first decreasing, then increasing), it means that *generalization error is a U-shaped function of capacity*. This is illustrated in Figure 5.3, showing the underfitting regime (left) and overfitting regime (right).

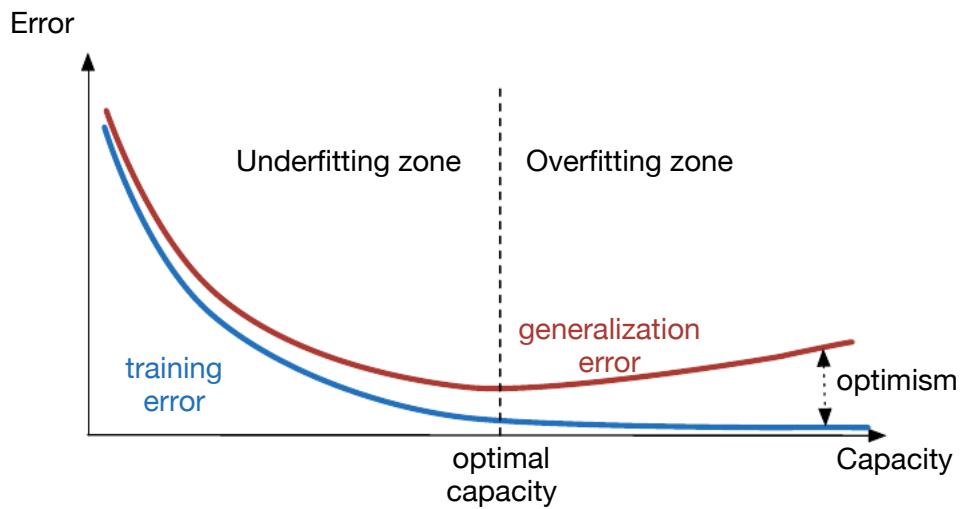


Figure 5.3: Typical relationship between capacity (horizontal axis) and both training (bottom curve, blue) and generalization (or test) error (top curve, red). Capacity is basically the number of examples that the learner can always fit. As we increase it, training error can be reduced, but the optimism (difference between training and generalization error, called “optimism”) increases. At some point, the increase in optimism is larger than the decrease in training error (typically when the training error is low and cannot go much lower), and we enter the *overfitting regime*, where capacity is too large, above the *optimal capacity*. Before reaching optimal capacity, we are in the *underfitting regime*.

How do we know if we are underfitting or overfitting? If by increasing capacity we decrease generalization error, then we are underfitting, otherwise we are overfitting. The capacity associated with the transition from underfitting to overfitting is the *optimal capacity*.

An interesting question is how one can select optimal capacity, and how it is expected to vary. We can use a *validation set* on monitor generalization error as a function of capacity and thus estimate optimal capacity empirically. Methods such as *cross-validation* (TODO) allow one to estimate generalization error even when the dataset is small, by considering many possible splits of the data into training and validation examples. What does theory tell us about the optimal capacity? Because theory predicts that optimism roughly increases with the ratio of capacity and number of training examples (Vapnik, 1995), it means that *optimal capacity should increase with the number of training examples*, as illustrated in Figure 5.4. This also agrees with intuition: with more training examples, we can “afford” a more complex model. If one only sees a single (x_1, y_1) example, a very reasonable prediction for a function $y = f(x)$ is the constant function $f(x) = y_1$. If one sees two examples (x_1, y_1) and (x_2, y_2) , a very reasonable prediction is the straight line that passes through these two points, i.e., a linear function of x . As we see more examples, it makes sense to consider higher order polynomials, but certainly not of higher order than the number of training examples.

With this notion in mind, let us consider an example. We can fit a polynomial of degree n to a dataset by using linear regression on a transformed space. Instead of fitting a linear function to $\mathbf{x} = [x_1]^\top$, we fit it to $\mathbf{x}' = [1, x_1, x_1^2, \dots, x_1^n]^\top$. For an example of how lower order polynomials are more prone to underfitting and higher order polynomials are more prone to overfitting, see Fig. 5.2.

Generally speaking, underfitting can be caused in two ways:

1. *The model complexity (richness of the family of functions) is too limited for the function being approximated.* This is the situation illustrated in Fig. 5.2 and is most commonly how machine learning practitioners perceive underfitting.
2. *The optimization procedure fails to find a solution that well approximates the target function despite the model having sufficient capacity for the task.* While in practice it is often difficult to decouple the two effects, this cause for underfitting is thought to be fairly common with deep learning methods. This is one reason why there is so much emphasis on models and methods that promote more effective optimization. We discuss these in Chapter 8.

5.4 Estimating and Monitoring Generalization Error

TODO

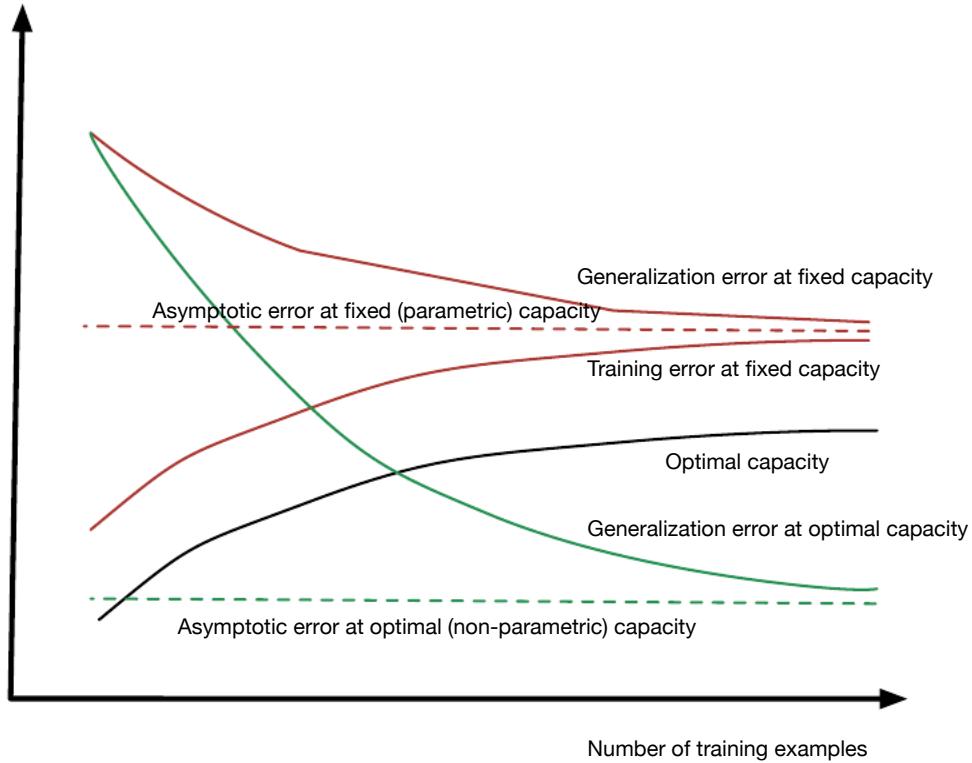


Figure 5.4: We consider here both the classical parametric case (red curves) where capacity is fixed (but we see the effect of increasing number of training examples), e.g., a logistic regression or neural network with fixed number of hidden units, and the non-parametric case where capacity (e.g., number of hidden units, number of mixture components, etc.) is allowed to increase optimally (e.g. by cross-validation) as the number of training examples increases. So, with more training examples, optimal capacity (bold black) increases (we can afford a bigger and more flexible model), and the associated generalization error (green bold) would decrease, eventually reaching the (non-parametric) asymptotic error (green dashed line, the best possible performance or Bayes error). If capacity was fixed (parametric setting), increasing the number of training examples would also decrease generalization error (top red curve), but not as fast (we are underfitting, capacity is too small), and training error would slowly increase (bottom red curve), so that both would meet at an asymptotic value (dashed red line) corresponding to the best achievable solution in the chosen fixed-size parametric class of learned functions.

5.5 Estimators, Bias, and Variance

TODO TODO: discuss sample variance and covariance using sample mean versus known true mean

As a discipline, machine learning borrows many useful concepts from statistics. Foundational concepts such as parameter estimation, bias and variance are all borrowed from statistics. In this section we briefly review these concepts and relate them to the machine learning concepts that we have already discussed, such as generalization.

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 function.

In order to distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter θ by $\hat{\theta}$.

In the remainder of this section we will be assuming the orthodox perspective on statistics. That is, we assume that the true parameter value θ is fixed but unknown, while the point estimate $\hat{\theta}$ is a function of the data and is therefore a random variable.

Let x_1, \dots, x_n be n independent and identically distributed (IID) data points. A **point estimator** is any function of the data:

$$\hat{\theta}_n = g(x_1, \dots, x_n). \quad (5.2)$$

In other words, any statistic¹ 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 $g(x_1, \dots, x_n)$ should correspond to that of the 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.

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) y given an input vector x (also called the covariates). We consider that there is a function $f(x)$ that describes the relationship between y and x . For example, we may assume that $y = f(x) + \epsilon$.

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.

¹A statistic is a function of the data, typically of the whole training set, such as the mean.

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 x and target y .

In the following we will review the most commonly studied properties of point estimators and discuss what they tell us about point estimators.

As $\hat{\theta}$ 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:

$$\text{bias}(\hat{\theta}_n) = \mathbb{E}(\hat{\theta}_n) - \theta \quad (5.3)$$

where the expectation is over the data (seen as samples from a random variable). An estimator $\hat{\theta}_n$ is said to be *unbiased* if $\text{bias}(\hat{\theta}_n) = 0$, i.e., if $\mathbb{E}(\hat{\theta}_n) = \theta$. An estimator $\hat{\theta}_n$ is said to be *asymptotically unbiased* if $\lim_{n \rightarrow \infty} \text{bias}(\hat{\theta}_n) = 0$, i.e., if $\lim_{n \rightarrow \infty} \mathbb{E}(\hat{\theta}_n) = \theta$.

Example: Bernoulli distribution Consider a data samples $\mathbf{x} \in \{0, 1\}^n$ that are independently and identically distributed according to a Bernoulli distribution ($x_i \sim \text{Bernoulli}(\theta)$, where $i \in [1, n]$). The Bernoulli *p.m.f.* (probability mass function, or probability function) is given by $P(x_i; \theta) = \theta^{x_i}(1 - \theta)^{(1-x_i)}$.

We are interested in knowing if the estimator $\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n x_i$ is biased.

$$\begin{aligned} \text{bias}(\hat{\theta}) &= \mathbb{E}[\hat{\theta}_n] - \theta \\ &= \mathbb{E}\left[\frac{1}{n} \sum_{i=1}^n x_i\right] - \theta \\ &= \frac{1}{n} \sum_{i=1}^n \mathbb{E}[x_i] - \theta \\ &= \frac{1}{n} \sum_{i=1}^n \sum_{x_i=0}^1 \left(x_i \theta^{x_i} (1 - \theta)^{(1-x_i)}\right) - \theta \\ &= \frac{1}{n} \sum_{i=1}^n (\theta) - \theta \\ &= \theta - \theta = 0 \end{aligned}$$

Since $\text{bias}(\hat{\theta}) = 0$, we say that our estimator $\hat{\theta}$ is unbiased.

TODO: another example Unbiased estimators are clearly desirable, though 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

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*.

$$\text{Var}[\hat{\theta}] = \mathbb{E}[\hat{\theta}^2] - \mathbb{E}[\hat{\theta}]^2 \quad (5.4)$$

We can also define the standard error (se) of the estimator as

$$\text{se}(\hat{\theta}) = \sqrt{\text{Var}[\hat{\theta}]} \quad (5.5)$$

Example: Bernoulli distribution Let’s once again consider a dataset $\mathbf{x} \in \{0, 1\}^n$ that are independently and identically distributed according to the Bernoulli distribution, where $P(x_i; \theta) = \theta^{x_i}(1 - \theta)^{(1-x_i)}$. This time we are interested in computing the variance of the estimator $\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n x_i$.

$$\begin{aligned} \text{Var}(\hat{\theta}_n) &= \text{Var} \left(\frac{1}{n} \sum_{i=1}^n x_i \right) \\ &= \frac{1}{n^2} \sum_{i=1}^n \text{Var}(x_i) \\ &= \frac{1}{n^2} \sum_{i=1}^n \theta(1 - \theta) \\ &= \frac{1}{n^2} n\theta(1 - \theta) \\ &= \frac{1}{n} \theta(1 - \theta) \end{aligned}$$

TODO: another example

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 less 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, discussed (TODO where?) Alternatively, we can also compare the *mean squared error* (MSE) of the estimates:

$$\begin{aligned} \text{MSE} &= \mathbb{E}[\hat{\theta}_n - \theta]^2 \\ &= \text{Bias}(\hat{\theta}_n)^2 + \text{Var}[\hat{\theta}_n] \end{aligned} \quad (5.6)$$

The MSE measures the overall expected deviation – in a squared error sense – between the estimator and the true value of the parameter θ . As is clear from Eq. 5.6, 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.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting discussed in Section. 5.3. 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. We thus see again the U-shaped curve of generalization error as a function of capacity, as in Section 5.3.3 and Figure 5.3.

TODO

TODO Example trading off bias and variance let's compare two estimators for the variance of a Gaussian distribution. One known as the sample variance:

$$\text{SampleVariance} \quad (5.7)$$

TODO

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 $\lim_{n \rightarrow \infty} \hat{\theta}_n \xrightarrow{P} \theta$.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.

Asymtotic unbiasedness is not equivalent to consistency. For example, consider estimating the mean parameter μ of a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with a dataset consisting of n samples: $\{x_1, \dots, x_n\}$. We could use the first sample x_1 of the dataset as

²The symbol \xrightarrow{P} means that the convergence is in probability, i.e. for any $\epsilon > 0$, $P(|\hat{\theta}_n - \theta| > \epsilon) \rightarrow 0$ as $n \rightarrow \infty$.

³This is sometime referred to as weak consistency, with strong consistency referring to the *almost sure* convergence of $\hat{\theta}$ to θ .

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 $\hat{\theta}_n \rightarrow \theta$ as $n \rightarrow \infty$.

5.6 Maximum Likelihood Estimation

TODO: refer back to sec:linreg, we promised we'd justify using mean squared error as a performance measure for linear regression justify minimizing mean squared error as a learning algorithm for linear regression TODO: relate it to KL divergence TODO

In the previous section we discussed a number of common properties of estimators but we never mentioned where these estimators come from. In this section we discuss one of the most common approaches to deriving estimators: via the maximum likelihood principle.

5.6.1 Properties of Maximum Likelihood

TODO: Asymptotic convergence and efficiency

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.

5.6.2 Regularized Likelihood

In practice, many machine learning algorithms actually do not fit the maximum likelihood framework but instead a slightly more general one, in which we introduce an additional term in the objective function, besides the objective function. We call that term the *regularizer* and it depends on the chosen function or parameters, indicating a *preference over some functions or parameters over others*. Typically, the regularizer prefers *simpler solutions*, such as the *weight decay regularizer*

$$\lambda \|\theta\|^2$$

where θ is the parameter vector and λ is a scalar that controls the strength of the regularizer (here, to be minimized) against the negative log-likelihood. Regularization is treated in much greater depth in Chapter 7. When we interpret the regularizer as an a priori log-probability over parameters, it is also related to Bayesian statistics, treated next.

5.7 Bayesian Statistics

TODO: a quick overview of Bayesian statistics and the Bayesian world view

In this section we will briefly consider estimation theory from a Bayesian perspective. Historically, the statistic community has been divided between what has become known as orthodox statistics and Bayesian statistics. The difference is mainly one of world view but can have important practical implications.

As we mentioned above, the orthodox perspective is that the true parameter value θ is fixed but unknown, while the point estimate $\hat{\theta}$ 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 states of knowledge (actually the lack thereof, i.e. probability represents the certainty of states of knowledge). The data is directly observed and so is not random. On the other hand, the true parameter θ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent our knowledge of θ using the *prior probability distribution*, $p(\theta)$ (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 θ before observing any data. For example, we might assume a priori that θ 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).

We can recover the effect of data on our belief about θ by combining the data likelihood $p(\mathbf{x} | \theta)$ with the prior via Bayes’ rule:

$$p(\theta | \mathbf{x}) = \frac{p(\mathbf{x} | \theta)p(\theta)}{p(\mathbf{x})} \quad (5.8)$$

If the data is at all informative about the value of θ , the *posterior distribution* $p(\theta | \mathbf{x})$ will have less entropy (will be more ‘peaky’) than the prior $p(\theta)$.

We can consider the posterior distribution over values of θ as the basis for an estimator of θ . Just as with the orthodox perspective, our Bayesian estimator of θ is associated with a probability distribution. If we need a single point estimate, a good choice would be to use the *maximum a posteriori* (MAP) point.

$$\theta_{MAP} = \arg \max_{\theta} p(\theta | \mathbf{x}) = \arg \max_{\theta} \log p(\mathbf{x} | \theta) + \log p(\theta) \quad (5.9)$$

where we recognize the regularized maximum likelihood training criterion, on the right hand side, with $\log p(\mathbf{x} | \theta)$ being the log-likelihood and $\log p(\theta)$ corresponding to the regularizer. This shows that a more peaking prior corresponds to a stronger regularizer, while a completely flat prior (e.g. $p(\theta)$ a constant that does not depend on θ) means no regularization at all.

TODO: Redo the regression example with a prior.

5.8 Supervised Learning

5.8.1 Estimating Conditional Expectation by Minimizing Squared Error

In the context of linear regression, we discussed in Section 5.2 the squared error criterion

$$\mathbb{E}[||Y - f(X)||^2]$$

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 Y given the input variable X , i.e.,

$$\arg \min_f \mathbb{E}[|Y - f(X)|^2] = \mathbb{E}[Y|X] \quad (5.10)$$

where the expectations on the left and the right are over the same distribution over (X, Y) . In practice we will minimize over the training set (mean squared error) but hope to obtain an estimator of the conditional expectation with respect to the data generating distribution, i.e., to generalize. Note that the minimization in Eq. 5.10 is over all possible functions f , i.e., it is a non-parametric result. The equation remains true if f belongs to a particular parametric family (such as the affine functions) so long as the true expected value also belongs to the same parametric family. The above can be proven by computing derivatives of the expected squared error with respect to each particular value $f(x)$ (for every particular x), and setting the derivative to 0.

5.8.2 Estimating Probabilities or Conditional Probabilities by Maximum Likelihood

A demonstration similar to the above can be written down to show that the maximum likelihood criterion yields an estimator of the data generating probability distribution (or conditional distribution).

We show the conditional case, estimating the true $Q(Y|X)$, but this immediately applies to the unconditional case by ignoring X . Consider the negative log-likelihood (NLL) training criterion,

$$\text{NLL} = \mathbb{E}[-\log P_\theta(Y|X)] = -\sum_{x,y} Q(x,y) \log P_\theta(y|x) \quad (5.11)$$

where the expectation is over the data generating distribution $P(X, Y)$ and we have used sums (which apply to discrete variables) but they should be replaced by integrals when the variables are continuous. We can transform the NLL into a Kullback-Liebler divergence between $P_\theta(Y|X)$ and $P(Y|X)$, by subtracting the entropy term

$$H(Q(Y|X=x)) = \mathbb{E}_{Q(Y|X=x)}[-\log Q(Y|X=x)] = -\sum_y Q(y|x) \log Q(y|x)$$

where again we used sums but they should be replaced by integrals for continuous variables. Subtracting the conditional entropy of $Q(Y|X)$ (which does not depend on P_θ) from the NLL yields the KL-divergence between $Q(Y|X=x)$ and $P_\theta(Y|X=x)$, with Q as the reference:

$$\begin{aligned} -\sum_{x,y} Q(x,y) \log P_\theta(y|x) + \sum_y Q(y|x) \log Q(y|x) &= \sum_x Q(x) \sum_y Q(y|x) \log \frac{Q(y|x)}{P_\theta(y|x)} \\ &= \mathbb{E}_{Q(X)} KL(Q(Y|X)||P_\theta(Y|X)). \end{aligned}$$

As we have seen in Section 3.9, the KL divergence is minimized when the two distributions are equal, which proves that the non-parametric minimizer of the NLL over $P_\theta(Y|X)$ is the target conditional distribution $Q(Y|X)$.

The above motivates the maximum likelihood criterion (or minimizing the NLL) when trying to estimate probabilities or conditional probability distributions.

TODO– logistic regression TODO– SVMs

5.9 Unsupervised Learning

Unsupervised learning is another broad class of machine learning methods that is distinct from supervised learning. In supervised learning, the goal is to use input-label pairs, x, y to learn a function f that predicts a label (or a distribution over labels) given the input, i.e. $\hat{y} = f(x)$. In unsupervised learning, no such label or other target is provided. The data consists of a set of examples x and the objective is to learn about the statistical structure of x itself.

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 reserves 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 representation⁴ 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 are more-or-less mutually independent. This happens because the pressure to encode as much information about the data x as possible into a low-dimensional representation drives the elements of this representation to be more independent. Any dependency between the variables in $f(x)$ is evidence of redundancy and implies that the representation $f(x)$ could have captured more information about x .

The notion of representation is one of the central themes of deep learning and therefore one of the central themes in this book. Section 5.11 of this chapter discusses some of the qualities we would like in our learned representations.

⁴sparse representations often use over-complete representations: the representation dimension is greater than the original dimensionality of the data.

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 5.5).

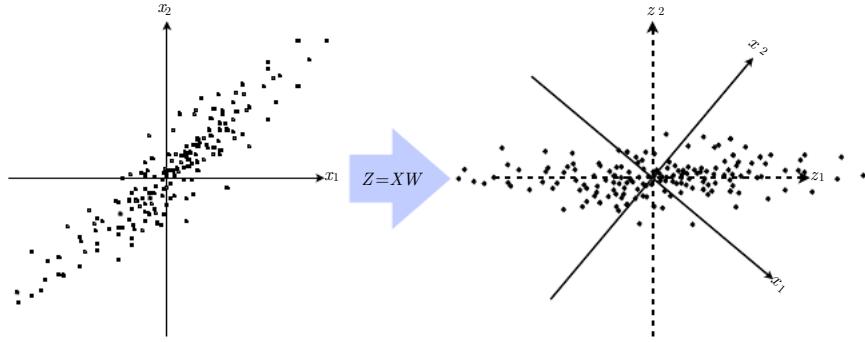


Figure 5.5: Illustration of the data representation learned via PCA.

In section 2.11, we saw that we could learn a one-dimensional representation that best reconstructs the original data (in the sense of mean-squared error) and that this representation actually corresponds to the first principle 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 (again, as measured by least-squares reconstruction error) as possible. In the following, e 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 \mathbf{X} .

Let us consider the $n \times m$ -dimensional design matrix \mathbf{X} . We will assume that the data has a mean of zero, $\mathbb{E}[\mathbf{x}] = \mathbf{0}$. If this is not the case, the data can easily be centered (mean removed). The sample covariance matrix associated with \mathbf{X} is given by:

$$\text{Var}[\mathbf{x}] = \frac{1}{n-1} \mathbf{X}^\top \mathbf{X} \quad (5.12)$$

One important aspect of PCA is that it finds a representation (through linear transformation) $\mathbf{Z} = \mathbf{X}\mathbf{W}$ where $\text{Var}[\mathbf{z}]$ is diagonal. To do this, we will make use of the singular value decomposition (SVD) of \mathbf{X} : $\mathbf{X} = \mathbf{U}\Sigma\mathbf{W}^\top$, where Σ is an $n \times m$ -dimensional rectangular diagonal matrix with the singular values of \mathbf{X} on the diagonal, \mathbf{U} is an $n \times n$ matrix whose columns are orthonormal (i.e. unit length and orthogonal) and \mathbf{W} is an $m \times m$ matrix also composed of orthonormal column vectors.

Using the SVD of \mathbf{X} , we can re-express the variance of \mathbf{X} as:

$$\text{Var}[\mathbf{x}] = \frac{1}{n-1} \mathbf{X}^\top \mathbf{X} \quad (5.13)$$

$$= \frac{1}{n-1} (\mathbf{U} \boldsymbol{\Sigma} \mathbf{W}^\top)^\top \mathbf{U} \boldsymbol{\Sigma} \mathbf{W}^\top \quad (5.14)$$

$$= \frac{1}{n-1} \mathbf{W} \boldsymbol{\Sigma}^\top \mathbf{U}^\top \mathbf{U} \boldsymbol{\Sigma} \mathbf{W}^\top \quad (5.15)$$

$$= \frac{1}{n-1} \mathbf{W} \boldsymbol{\Sigma}^2 \mathbf{W}^\top, \quad (5.16)$$

where we use the orthonormality of \mathbf{U} ($\mathbf{U}^\top \mathbf{U} = \mathbf{I}$) and define \mathbf{D}^2 as an $m \times m$ -dimensional diagonal matrix with the squares of the singular values of \mathbf{X} on the diagonal, i.e. the i th diagonal elements is given by $D_{i,i}^2$.

This shows that if we take $\mathbf{Z} = \mathbf{X}\mathbf{V}$, we can ensure that the covariance associated with \mathbf{Z} is diagonal as required.

$$\text{Var}[z] = \frac{1}{n-1} \mathbf{Z}^\top \mathbf{Z} \quad (5.17)$$

$$= \frac{1}{n-1} \mathbf{W}^\top \mathbf{X}^\top \mathbf{X} \mathbf{W} \quad (5.18)$$

$$= \frac{1}{n-1} \mathbf{W} \mathbf{W}^\top \boldsymbol{\Sigma}^2 \mathbf{W} \mathbf{W}^\top \quad (5.19)$$

$$= \frac{1}{n-1} \boldsymbol{\Sigma}^2 \quad (5.20)$$

Similar to our analysis of the variance of \mathbf{X} above, we exploit the orthonormality of \mathbf{W} ($\mathbf{W}^\top \mathbf{W} = \mathbf{I}$). Our use of SVD to solve for the PCA components of \mathbf{X} (i.e. elements of \mathbf{z}) reveals an interesting connection to the eigen-decomposition of a matrix related to \mathbf{X} . Specifically, the columns of \mathbf{W} are the eigenvectors of the matrix $n \times n$ -dimensional matrix $\mathbf{X}^\top \mathbf{X}$.

The above analysis shows that when we project the data \mathbf{X} to \mathbf{Z} , via the linear transformation \mathbf{W} , the resulting representation has a diagonal covariance matrix (as given by $\boldsymbol{\Sigma}^2$) which immediately implies that the individual elements of \mathbf{Z} are mutually uncorrelated.

This ability of PCA to transform data into a representation where the elements are mutually exclusive is a very important property of PCA. It is a simple example of a representation that *disentangling the 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 \mathbf{W}) that aligns the principle axes of variance with the basis of the new representation space associated with \mathbf{Z} , as illustrated in Fig. 5.5. 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. 5.11 and later in detail in Chapter 17.

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 x, y pairs, as in supervised learning, but where the labels y are either unreliable 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.

5.11 The Smoothness Prior, Local Generalization and Non-Parametric Models

A fairly large number of machine learning algorithms are exploiting a single basic prior: the *smoothness prior*. It can be expressed in many different ways, but it basically says that the target function or distribution of interest f^* is smooth, i.e.,

$$f^*(x) \approx f^*(x + \epsilon) \quad (5.21)$$

for most configurations x and small change ϵ .

For example, if we force the learned $f(\cdot)$ to be piecewise constant, we obtain a *histogram*, with the number of pieces being equal to the number of *distinguishable regions*, or “bins” in which the different x values can fall. Another example of piecewise constant learned function is what we obtain with *K-nearest neighbors* predictors, where $f(x)$ is constant in some region R containing all the points x that have the same set of K nearest neighbors from the training set. If we are doing classification and $K=1$, $f(x)$ is just the output class associated with the nearest neighbor of x in the training set. If we are doing regression, $f(x)$ is the average of the outputs associated with the K nearest neighbors of x . Note that in both cases, for $K = 1$, the number of distinguishable regions of cannot be more than the number of training examples.

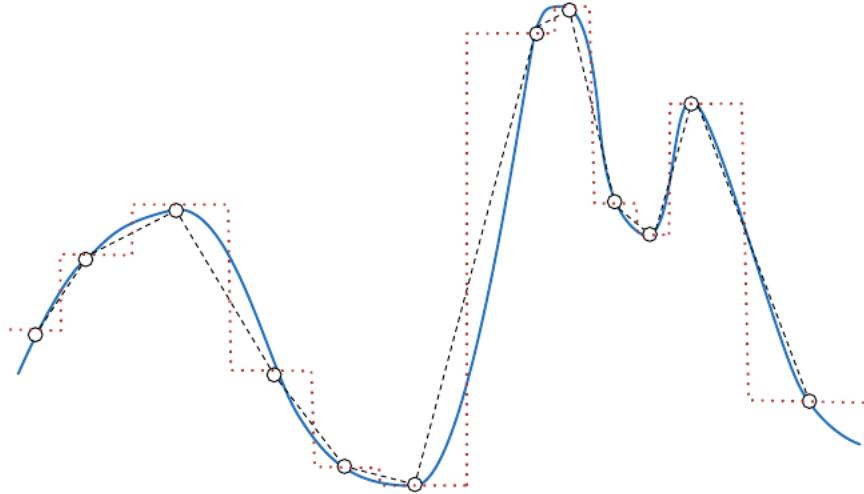


Figure 5.6: Illustration of interpolation and kernel-based methods, which construct a smooth function by interpolating in various ways between the training examples (circles), which act like knot points controlling the shape of the implicit regions that separate them as well as the values to output within 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 smooth, i.e., does not vary much locally. It allows to *generalize locally*, 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 5.6. For example, *non-parametric kernel density estimation methods* and *kernel regression* methods construct a learned function f of the form

$$f(x) = b + \sum_{i=1}^n \alpha_i K(x, x_i) \quad (5.22)$$

where x_i are the n training examples, or

$$f(x) = b + \sum_{i=1}^n \alpha_i \frac{K(x, x_i)}{\sum_{j=1}^n K(x, x_j)}.$$

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

$$K(u, v) = N(u - v; 0, \sigma^2 I)$$

where $N(x; \mu, \Sigma)$ is the standard normal density. With K a *local kernel* (Bengio *et al.*,

2006a; Bengio and LeCun, 2007a; Bengio, 2009)⁵, we can think of each 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 estimators α_i of Eq. 5.22 are fixed (e.g. to $1/n$ for density estimation and to y_i for supervised learning from examples (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).

However, as illustrated in Figure 5.6, 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.

Another type of non-parametric 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. Each node of the decision tree is associated with a region, and it breaks it into one sub-region for each child of the node (typically using an axis-aligned cut). Typically, a constant output is returned for all inputs associated with a leaf node, i.e., within the associated region. Because each example only informs the region in which it falls about the target output, 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). Below (Section 5.12) we examine how this makes decision trees and other such learning algorithms based only on the smoothness prior potential victims of the curse of dimensionality.

In all cases, the smoothness assumption (Eq. 5.21) allows the learner to *generalize locally*. Since we assume that the target function obeys $f^*(x) \approx f^*(x + \epsilon)$ most of the time for small ϵ , we can generalize the empirical distribution (or the (x, y) training pairs) to their neighborhood. If (x_i, y_i) is a supervised training example, then we expect $f^*(x_i) \approx y_i$, and therefore if x_i is a near neighbor of x , we expect that $f^*(x) \approx y_i$. By

⁵i.e., with $K(u, v)$ large when $u = v$ and decreasing as they get farther apart

considering more neighbors, we can obtain better generalization, by better executing the smoothness assumption.

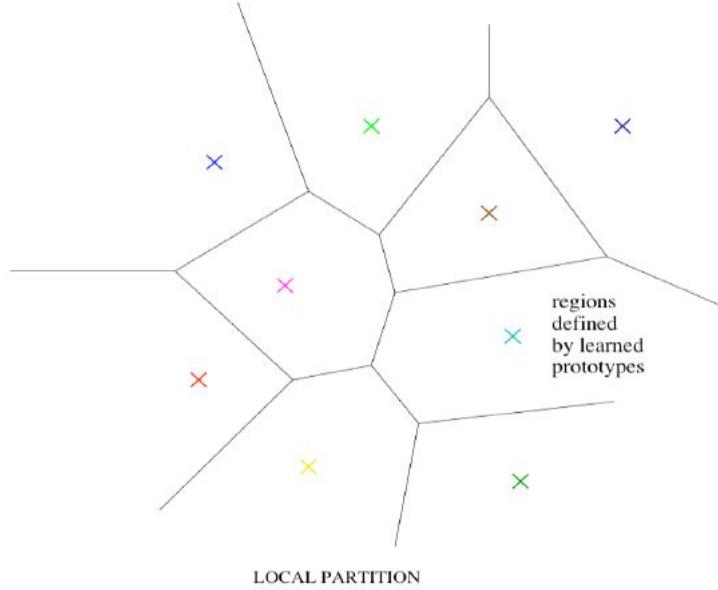


Figure 5.7: Illustration of how learning algorithms that exploit only the smoothness prior typically break up the input space into regions, with examples in those regions being used both to define the region boundaries and what the output should be within each region. The figure shows the case of clustering 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 5.7 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?

The smoothness assumption and the associated 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 17. 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 associated with additional priors about the underlying data generating distribution. In this way, we can actually generalize non-locally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006b). For example, if we assume that the target function is periodic, then we can generalize from examples of a few periods to an arbitrarily large number of repetitions of the basic pattern. However, we want a more general-purpose assumption, and the 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 17.

5.12 Manifold Learning and the Curse of Dimensionality

Manifold learning algorithms assume that the data distribution is concentrated in a small number of dimensions. 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 consider that if the assumption was false, then sampling uniformly at random should produce probable (data-like) configurations reasonably often. Instead, take the example of generating images by independently picking the grey level (or a binary 0 vs 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 5.8.



Figure 5.8: 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 experiment, which is 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 establishes that for a large class of datasets, the *manifold hypothesis* is true: the data generating distribution concentrates in a small number of dimensions, as in the cartoon of Figure 18.4, from Chapter 18. 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.

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 reducing the data to a lower dimensional representation. The curse of dimensionality refers to the exponential increase of the number of configurations of interest as a function of the number of dimensions. This is illustrated in Figure 5.9.

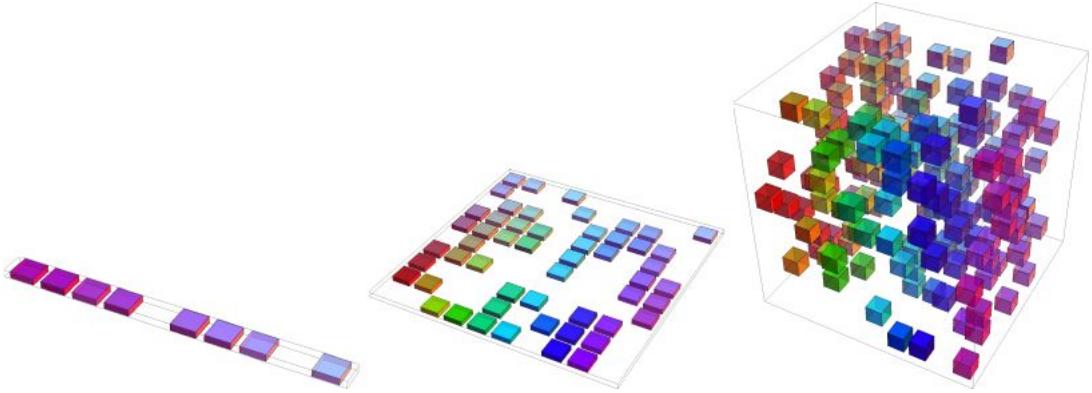


Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. In the figure we first consider one-dimensional data (left), i.e., one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (cells, in the figure), we 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 $10 \times 10 = 100$ regions, and we need at least that many examples to cover all those regions. With 3 dimensions (right) this grows to $10^3 = 1000$ regions and at least that many examples. For d dimensions and V values to be distinguished along each axis, it looks like we need $O(V^d)$ 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 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 18, 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). And even if we were to reduce the dimensionality of an input from 10000 (e.g. 100×100 binary pixels) to 100, V^{100} is still too large to hope covering with a training set. This rules out the use of purely local generalization (i.e., the smoothness prior only) to model such manifolds, as discussed in Chapter 18 around Figure 18.4 and 18.5. It may also be that although the effective dimensionality of the data could be small, that some examples fall outside of the main manifold and that we don't 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 16.7.

5.13 Challenges of High-Dimensional Distributions

High-dimensional random variables actually bring two challenges: a statistical challenge and a computational challenge.

The *statistical challenge* regards generalization: as sketched in the previous section, 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). This raises the need to generalize to never-seen configurations, and not just those in the immediate neighborhood of the training examples. This is difficult because the volume of these regions where we want to generalize grows exponentially with the number of dimensions.

In order to achieve that kind of non-local generalization, it is necessary to introduce priors other than the smoothness prior, and much of deep learning research is about such priors. In particular, 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 17 discusses these questions from the angle of representation learning and the objective of *disentangling the underlying factors of variation*.

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.

For example, suppose that we have been able to learn a probability function $p(X)$ which maps points $\mathbf{x} = (x_1, x_2, \dots, x_d)$ in a d -dimensional space to a scalar $p(X = \mathbf{x})$. Then simply predicting the joint probability of a subset of d' of these variables given d'' of the others requires computing a sum (or an integral) over all the configurations of $d - d''$ variables:

$$p(x_1, \dots, x_d | x_{d'+1}, \dots, x_{d'+d''}) = \frac{\sum_{x_{d'+d''+1}, \dots, x_d} p(x_1, \dots, x_d)}{\sum_{x_1, \dots, x'_d, x_{d'+d''+1}, \dots, x_d} p(x_1, \dots, x_d)} \quad (5.23)$$

This is an example of *inference*. Exact inference is generally intractable, but approximate techniques are discussed in Chapter 20.

Another common example of inference is where we want to find the most probable configuration out of some exponentially large set, e.g.,

$$\arg \max_{x_1, \dots, x_d} p(x_1, \dots, x_d | x_{d'+1}, \dots, x_d).$$

In that case, the answer involves maximizing rather than summing over an exponentially large number of configurations, and optimization or search methods can be used to help

approximating a solution (an exact solution is rarely achievable with non-exponential computational resources).

Another even more common example is the difficulty of computing the normalization constant, called *partition function*, associated with a probability function, studied in more detail in Section 14.2.3 and Chapter 19. Unless we strongly restrict the mathematical form of the probability function, we may end up with a probability function whose value can be computed efficiently only up to a normalization constant $Z(\theta)$:

$$p(\mathbf{x}) = \frac{p_{\theta}^*(\mathbf{x})}{Z(\theta)}$$

where the normalization constant or partition function $Z(\theta)$ is a generally intractable sum or integral

$$Z(\theta) = \sum_{\mathbf{x}} p_{\theta}^*(\mathbf{x})$$

and p_{θ}^* is an unnormalized probability function with parameters θ . This is the case, for example, with Restricted Boltzmann Machines (Sections 14.6.1 and 21.1) and many other undirected graphical models. In order to learn θ , we would like to compute the derivative of $Z(\theta)$ with respect to θ and this is generally not computationally tractable in an exact way, but many approximations have been proposed. Techniques to face the issues raised with the partition function and its gradient are discussed in chapter 19.

Sometimes the \mathbf{x} in the above examples should be viewed as including not just *visible variables* (the variables in our dataset) but also *latent variables*. Latent variables are unobserved variables that we introduce in our probabilistic models in order to increase their expressive power. Probabilistic graphical models (including ones with latent variables) are discussed at length in Chapters 14 and 21. In the context of probabilistic models, one often resorts to Monte-Carlo Markov Chain (MCMC) methods in order to estimate such intractable sums (seen as expectated values). MCMC is introduced in Chapter 14 and comes back in Chapters 19 and 21. One issue that arises with MCMC is that it may requiring sequentially visiting a large fraction of the *modes* (regions of high probability) of the distribution. If the number of these modes (which would be larger for more complex distributions) is very large (and it is potentially exponentially large), then the MCMC estimators may be too unreliable. Again, it has to do with having an exponentially large set of configurations (or modes, or regions) that we would like to visit for performing some computation.

One way to confront these intractable computations is to approximate them, and many approaches have been proposed, discussed in the chapters listed above. Another way is to avoid these intractable computations altogether by design, and methods that do not require such computations are thus very appealing, although maybe at the price of losing the ability to actually compute the probability function itself. Several generative models based on auto-encoders have been proposed in recent years, with that motivation, and are discussed at the end of Chapter 21.

Part II

Modern practical deep networks

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 parts 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 are easy for a person to do can be accomplished via deep learning given a large enough model and a large enough dataset of labeled training examples.

This part of the book describes this core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. This part of the book includes details such as how to efficiently model specific kinds of inputs, such as how to process image inputs with convolutional networks and 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.

This part of the book is 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

6.1 Formalizing and Generalizing Neural Networks

Feedforward supervised neural networks were among the first and most successful learning algorithms (Rumelhart *et al.*, 1986d,c). They are also called deep networks, multi-layer Perceptron (MLP), or simply neural networks and the vanilla architecture with a single hidden layer is illustrated in Figure 6.1.

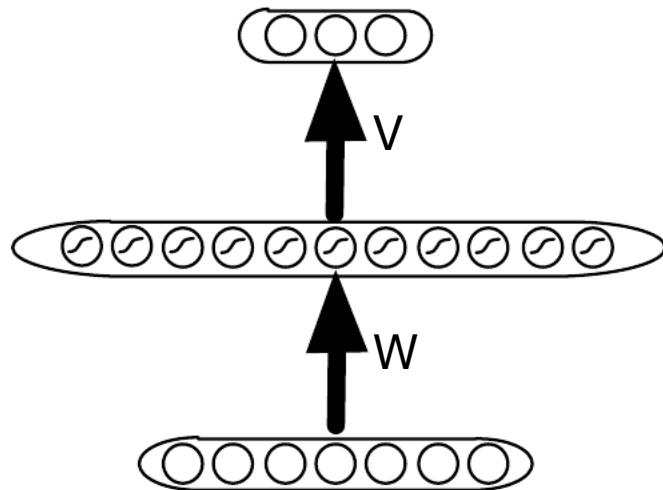


Figure 6.1: Vanilla (shallow) MLP, with one sigmoid hidden layer, computing vector-valued hidden unit vector $\mathbf{h} = \text{sigmoid}(\mathbf{c} + \mathbf{W}\mathbf{x})$ with weight matrix \mathbf{W} and bias or offset vector \mathbf{c} . The output vector is obtained via another learned affine transformation $\hat{\mathbf{y}} = \mathbf{b} + \mathbf{V}\mathbf{h}$, with weight matrix \mathbf{V} and output bias (offset) vector \mathbf{b} .

MLPs can learn powerful non-linear transformations: in fact, with enough hidden units they can represent arbitrarily complex but smooth functions (see Section 6.4). This is achieved by composing simpler but still 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 Figure 6.2.

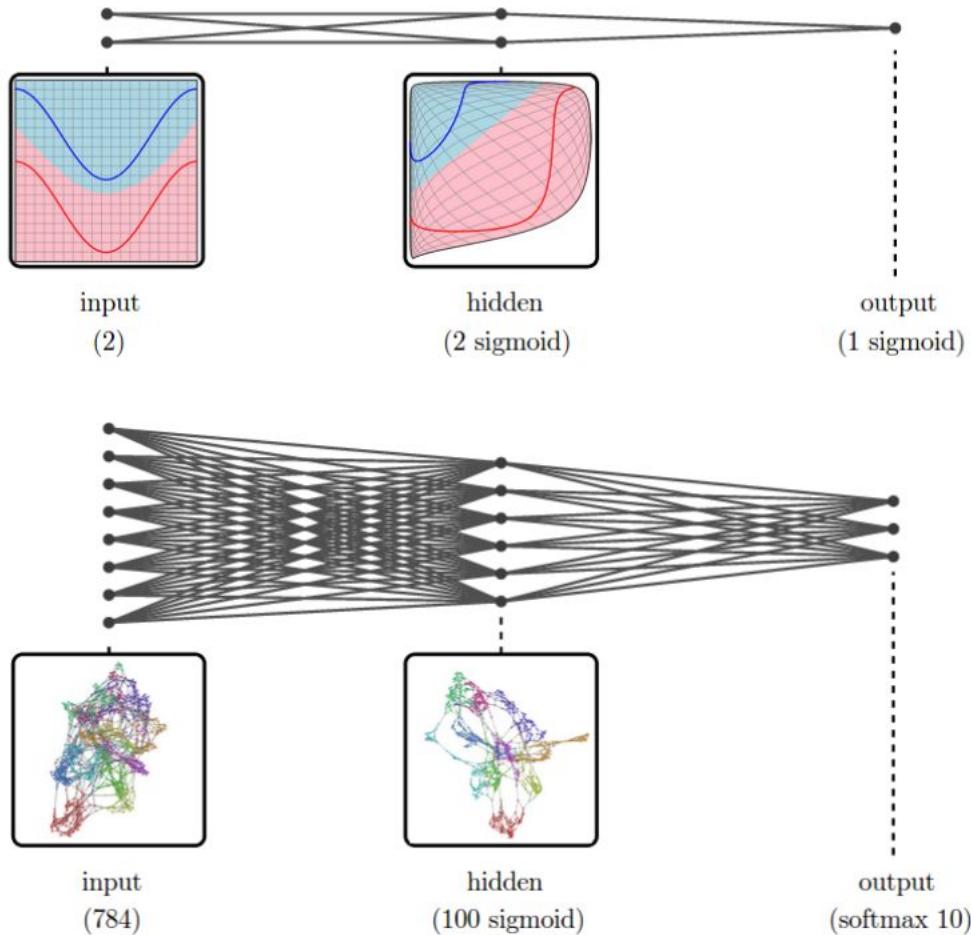


Figure 6.2: Each layer of a trained neural network non-linearly transforms its input, distorting the space so that the task becomes easier to perform, e.g., linear classification in the above figures. Top: a vanilla neural network with 2 hidden units in its single hidden layer can transform the 2-D input space so that the two classes (in red/pink vs blue) become linearly separable (the red and blue curves correspond to the “manifold” near which examples concentrate). Bottom: with a larger hidden layer (100 here), the MNIST digit images (with $28 \times 28 = 784$ pixels) can be transformed so that the classes (each shown with a different color) can be much more easily classified by the output linear+softmax layer (over 10 digit categories). Both figures are reproduced with permission by Chris Olah from <http://colah.github.io/>, where many more insightful visualizations can be found.

In addition to covering the basics of such networks, this chapter introduces a general formalism for gradient-based optimization of parametrized families of functions, often in the context of conditional maximum likelihood criteria (Section 6.2).

They bring together a number of important machine learning concepts already introduced in the previous chapters:

- Define a **parametrized family of functions** f_{θ} describing how the learner will behave on new examples, i.e., what output $f_{\theta}(\mathbf{x})$ will produce given some input \mathbf{x} . Training consists in choosing the parameter θ (usually represented by a vector) given some training examples (\mathbf{x}, \mathbf{y}) sampled from an unknown data generating distribution $P(\mathbf{X}, \mathbf{Y})$.
- Define a **loss function** L describing what scalar loss $L(\hat{\mathbf{y}}, \mathbf{y})$ is associated with each supervised example (\mathbf{x}, \mathbf{y}) , as a function of the learner's output $\hat{\mathbf{y}} = f_{\theta}(\mathbf{x})$ and the target output \mathbf{y} .
- Define a **training criterion** and a **regularizer**. The objective of training is ideally to minimize the expected loss $\mathbb{E}_{\mathbf{X}, \mathbf{Y}}[L(f_{\theta}(\mathbf{X}), \mathbf{Y})]$ over \mathbf{X}, \mathbf{Y} sampled from the unknown data generating distribution $P(\mathbf{X}, \mathbf{Y})$. However this is not possible because the expectation makes use of the true underlying $P(\mathbf{X}, \mathbf{Y})$ but we only have access to a finite number of *training examples*, i.e. of pairs (\mathbf{X}, \mathbf{Y}) . Instead, one defines a training criterion which usually includes an empirical average of the loss over the training set, plus some additional terms (called regularizers) which enforce some preferences over the choices of θ .
- Define an **optimization procedure** to *approximately* minimize the training criterion¹. The simplest such optimization procedure is stochastic gradient descent, described in Section 4.3.

Example 6.1.1 illustrates these concepts for the case of a vanilla neural network for regression.

In chapter 17, we consider generalizations of the above framework to the unsupervised and semi-supervised cases, where \mathbf{Y} may not exist or may not always be present. An unsupervised loss can then be defined solely as a function of the input \mathbf{x} and some function $f_{\theta}(\mathbf{x})$ that one wishes to learn.

¹It is generally not possible to analytically obtain a global minimum of the training criterion, and iterative numerical optimization methods are used instead.

Example 6.1.1. Vanilla (Shallow) Multi-Layer Neural Network for Regression

Based on the above definitions, we could pick the family of input-output functions to be

$$f_{\theta}(\mathbf{x}) = \mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\mathbf{x}),$$

illustrated in Figure 6.1, where $\text{sigmoid}(a) = 1/(1 + e^{-a})$ is applied element-wise, the input is the vector $\mathbf{x} \in \mathbb{R}^{n_i}$, the hidden layer outputs are the elements of the vector $\mathbf{h} = \text{sigmoid}(\mathbf{c} + \mathbf{W}\mathbf{x})$ with n_h entries, the parameters are $\theta = (\mathbf{b}, \mathbf{c}, \mathbf{V}, \mathbf{W})$ (viewed as the flattened vectorized version of the tuple) with $\mathbf{b} \in \mathbb{R}^{n_o}$ a vector the same dimension as the output (n_o), $\mathbf{c} \in \mathbb{R}^{n_h}$ of the same dimension as \mathbf{h} (number of hidden units), $\mathbf{V} \in \mathbb{R}^{n_o \times n_h}$ and $\mathbf{W} \in \mathbb{R}^{n_h \times n_i}$ being weight matrices.

The loss function for this classical example could be the squared error $L(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$ (see Section 5.8.1 showing that it makes $\hat{\mathbf{y}}$ an estimator of $\mathbb{E}[\mathbf{Y}|\mathbf{x}]$). The regularizer could be the ordinary L^2 weight decay $\|\omega\|^2 = (\sum_{ij} W_{ij}^2 + \sum_{ki} V_{ki}^2)$, where we define the set of weights ω as the concatenation of the elements of matrices \mathbf{W} and \mathbf{V} . The L^2 weight decay thus penalizes the squared norm of the weights, with λ a scalar that is larger to penalize stronger and yield smaller weights. Combining the loss function and the regularizer gives the training criterion, which is the objective function during training:

$$C(\theta) = \lambda \|\omega\|^2 + \frac{1}{n} \sum_{t=1}^n \|\mathbf{y}^{(t)} - (\mathbf{b} + \mathbf{V}\text{sigmoid}(\mathbf{c} + \mathbf{W}\mathbf{x}^{(t)}))\|^2.$$

where $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$ 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 θ according to

$$\begin{aligned} \omega &\leftarrow \omega - \epsilon \left(2\lambda\omega + \nabla_{\omega} L(f_{\theta}(\mathbf{x}^{(t)}), \mathbf{y}^{(t)}) \right) \\ \beta &\leftarrow \beta - \epsilon \nabla_{\beta} L(f_{\theta}(\mathbf{x}^{(t)}), \mathbf{y}^{(t)}), \end{aligned}$$

where $\beta = (\mathbf{b}, \mathbf{c})$ contains the offset (bias) parameters, $\omega = (\mathbf{W}, \mathbf{V})$ the weight matrices, ϵ is a learning rate and t is incremented after each training example, modulo n .

6.2 Parametrizing a Learned Predictor

There are many ways to define the family of input-output functions, loss function, regularizer and optimization procedure, and the most common ones are described below, while more advanced ones are left to later chapters, in particular Chapters 10 and 16.

6.2.1 Family of Functions

A motivation for the family of functions defined by multi-layer neural networks is to *compose affine transformations and non-linearities*, where the choice of parameter values controls the degree and location of these non-linearities. As discussed in Section 6.4 below, with the appropriate choice of parameters, multi-layer neural networks can in principle approximate smooth functions, 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 tangent² activation functions instead of sigmoid activation functions:

$$\mathbf{h}^k = \tanh(\mathbf{b}^k + \mathbf{W}^k \mathbf{h}^{k-1})$$

where $\mathbf{h}^0 = \mathbf{x}$ is the input of the neural net, \mathbf{h}^k (for $k > 0$) is the output of the k -th hidden layer, which has weight matrix \mathbf{W}^k and offset (or bias) vector \mathbf{b}^k . If we want the output $f_\theta(x)$ 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 of course generally different from the *tanh*, depending on the type of output to be predicted and the associated loss function (see below).

As discussed in Chapter 12 (Section 12.5) there are several other non-linearities besides the sigmoid and the hyperbolic tangent which have been successfully used with neural networks. In particular, TODO discusses alternative non-linear units, such as the rectified linear unit ($\max(0, b + \mathbf{w} \cdot \mathbf{x})$) and the maxout unit ($\max_i(b_i + \mathbf{W}_{:,i} \cdot \mathbf{x})$) which have been particularly successful in the case of deep feedforward or convolutional networks. These and other non-linear neural network activation functions commonly found in the literature are summarized below.

- **Rectifier or rectified linear unit (ReLU) or positive part:** $h(\mathbf{x}) = \max(0, b + \mathbf{w} \cdot \mathbf{x})$, also written $(b + \mathbf{w} \cdot \mathbf{x})^+$.
- **Hyperbolic tangent:** $h(\mathbf{x}) = \tanh(b + \mathbf{w} \cdot \mathbf{x})$.
- **Sigmoid:** $h(\mathbf{x}) = 1/(1 + e^{-(b+\mathbf{w}\cdot\mathbf{x})})$.
- **Softmax:** $\mathbf{h}(\mathbf{x}) = \text{softmax}(\mathbf{b} + \mathbf{W}\mathbf{x})$. It is mostly used as output non-linearity for predicting discrete probabilities. See definition and discussion below, Eq. 6.1.

²which is linearly related to the sigmoid as follows: $\tanh(\mathbf{x}) = 2 \times \text{sigmoid}(2\mathbf{x}) - 1$

- **Radial basis function** or **RBF** unit: $h(x) = e^{-\|w-x\|^2/\sigma^2}$. 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 as a random subset of the input examples (Powell, 1987; Niranjan and Fallside, 1990).
- **Softplus**: this is a smooth version of the rectifier, introduced in Dugas *et al.* (2001) for function approximation and in Nair and Hinton (2010) 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, $h(x) = \max(-1, \min(1, x))$. It was introduced by Collobert (2004).
- **Absolute value rectification**: $h(x) = |x|$ (may be applied on the dot product output or on the output of a tanh unit). It is also related to the rectifier and has been used for object recognition from images (Jarrett *et al.*, 2009b), 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 TODO It generalizes the rectifier but introduces multiple weight vectors w_i (called filters) for each unit. $h(x) = \max_i(b_i + w_i \cdot x)$.

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 6.3, 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 (for which 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*. In general, the set of connections between units of the whole network only needs to form a directed acyclic graph in order to define a meaningful computation (see the flow graph formalism below, Section 6.3). When units of the network are connected to themselves through a cycle, one has to properly define what computation is to be done, and this is what is done in the case of *recurrent networks*, treated in Chapter 10.

6.2.2 Loss Function and Conditional Log-Likelihood

In the 80’s and 90’s the most commonly used loss function was the *squared error* $L(f_\theta(x), y) = \|f_\theta(x) - y\|^2$. As discussed in Section 5.8.1, if f is unrestricted (non-parametric), minimizing the expected value of the loss function over some data-generating

distribution $P(X, Y)$ yields $f(x) = \mathbb{E}[Y|X]$, the true conditional expectation of Y given X ³. See also Section 5.8.1 for more details. 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.

However, when y is a discrete label, i.e., for classification problems, other loss functions such as the Bernoulli negative log-likelihood⁴ have been found to be more appropriate than the squared error. In the case where $y \in \{0, 1\}$ is binary this gives $L(f_\theta(x), y) = -y \log f_\theta(x) - (1 - y) \log(1 - f_\theta(x))$ and it can be shown that the optimal (non-parametric) f minimizing this criterion is $f(x) = P(Y = 1|x)$. Note that in order for the above expression of the criterion to make sense, $f_\theta(x)$ 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 the output layer, which matches well with the Binomial negative log-likelihood criterion⁵. As explained below (Section 6.2.2), the cross-entropy criterion 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) criterion

$$L_{\text{NLL}}(f_\theta(x), y) = -\log P_\theta(Y = y|X = x).$$

See Section 5.8.2 which shows that this criterion yields an estimator of the true conditional probability of Y given X . For example, if Y is a continuous random variable and we assume that, given X , it has a Gaussian distribution with mean $f_\theta(X)$ and variance σ^2 , then

$$-\log P_\theta(Y|X) = \frac{1}{2}(f_\theta(X) - Y)^2/\sigma^2 + \log(2\pi\sigma^2).$$

Up to an additive and multiplicative constant (which would give the same choice of θ), this is therefore equivalent to 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 parametrized function of x (for example with output units that are guaranteed to be positive, or forming a positive definite matrix).

Similarly, for discrete variables, the Binomial negative log-likelihood criterion corresponds to the conditional log-likelihood associated with the Bernoulli distribution with

³Proving this is not difficult and is very instructive.

⁴This is often called *cross entropy* in the literature, even though the term cross entropy should also apply to many other losses that can be viewed as negative log-likelihood, discussed below in more detail. TODO: where do we actually discuss this? may as well discuss it in maximum likelihood section

⁵In reference to statistical models, this “match” between the loss function and the output non-linearity is similar to the choice of a *link function* in generalized linear models (McCullagh and Nelder, 1989).

probability $p = f_\theta(x)$ of generating $Y = 1$ given $X = x$ (and probability $1 - p$ of generating $Y = 0$):

$$-\log P_\theta(Y|X) = -\mathbf{1}_{Y=1} \log p - \mathbf{1}_{Y=0} \log(1-p) = -Y \log f_\theta(X) - (1-Y) \log(1-f_\theta(X)).$$

Softmax

When y is discrete (in some finite set, say $\{1, \dots, N\}$) but not binary, the Bernoulli distribution is extended to the Multinoulli (Murphy, 2012), Section 3.10.2 ⁶. This distribution is specified by a vector of $N - 1$ probabilities whose sum is less or equal to 1, each element of which provides the probability $\mathbf{p}_i = P(y = i|\mathbf{x})$. We thus need a vector-valued output non-linearity that produces such a vector of probabilities, and a commonly used non-linearity for this purpose is the *softmax* non-linearity introduced earlier.

$$\mathbf{p} = \text{softmax}(\mathbf{a}) \iff \mathbf{p}_i = \frac{e^{\mathbf{a}_i}}{\sum_j e^{\mathbf{a}_j}}. \quad (6.1)$$

where typically $\mathbf{a} = \mathbf{b} + \mathbf{W}\mathbf{h}$ is the vector of scores whose elements \mathbf{a}_i are associated with each category i . The corresponding loss function is therefore $L_{\text{NLL}}(\mathbf{p}, y) = -\log \mathbf{p}_y$. Note how minimizing this loss will push \mathbf{a}_y up (increase the score \mathbf{a}_y associated with the correct label y) while pushing \mathbf{a}_i (for $i \neq y$) down (decreasing the score of the other labels, in the context \mathbf{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 $\mathbf{p}_y = 1$ and they cancel in average over examples (say sharing the same x) if \mathbf{p}_i equals the fraction of times that $y = i$ for this value \mathbf{x} . 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 14. Note other interesting properties of the softmax. First of all, the gradient of the Multinoulli negative log-likelihood with respect to the softmax argument \mathbf{a} is

$$\begin{aligned} \frac{\partial}{\partial \mathbf{a}} L_{\text{NLL}}(\mathbf{p}(\mathbf{a}), y) &= \frac{\partial}{\partial \mathbf{p}_y} (-\log \mathbf{p}_y(\mathbf{a})) \cdot \frac{\partial \mathbf{p}_y}{\partial \mathbf{a}} \\ &= -\frac{1}{\mathbf{p}_y(\mathbf{a})} \cdot \mathbf{p}_y(\mathbf{a})(\mathbf{1}_{i=y} - \mathbf{p}) \\ &= (\mathbf{p} - \mathbf{1}_{i=y}) \end{aligned} \quad (6.2)$$

which does not *saturate*, the gradient does not vanish when the output probabilities approach 0 or 1, except when the model is providing the correct answer. Specifically, let's 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

$$\frac{\partial}{\partial \mathbf{a}_j} L_{\text{NLL}}(\mathbf{p}(\mathbf{a}), y) = \mathbf{p}_j. \quad (6.3)$$

⁶The Multinoulli is also known as the categorical or generalized Bernoulli distribution, which is a special case of the Multinomial distribution with one trial, and the term multinomial is often used even in this case.

So if the model correctly predicts a low probability that the $ry = j$, i.e. that $\mathbf{p}_j \approx 0$, then the gradient is also close to zero.

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, $\mathbf{y} = [0, \dots, 0, 1, 0, \dots, 0]$, i.e for the label $y = i$, we have $\mathbf{y}_i = 1$ and $\mathbf{y}_j = 0, \forall j \neq i$. We will again consider that we have the output of the network to be $\mathbf{p} = \text{softmax}(\mathbf{a})$, where, as before, \mathbf{a} is the input to the softmax function (i.e. $\mathbf{a} = \mathbf{b} + \mathbf{W}\mathbf{h}$ with \mathbf{h} the output of the last hidden layer).

For the squared error loss $L_2(\mathbf{p}(\mathbf{a}), \mathbf{y}) = \|\mathbf{p}(\mathbf{a}) - \mathbf{y}\|^2$, the gradient of the loss with respect to the input vector to the softmax, \mathbf{a} , is given by:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{a}} L(\mathbf{p}(\mathbf{a}), \mathbf{y}) &= \frac{\partial}{\partial \mathbf{p}} \frac{\partial}{\partial \mathbf{p}} \\ &= 2(\mathbf{p}(\mathbf{a}) - \mathbf{y}) \odot \mathbf{p} \odot (1 - \mathbf{p}),\end{aligned}$$

where \odot indicates an element-wise multiplication. Let us again consider the case where $y = i$, the gradient associated with an erroneous label, say $j \neq i$, is

$$\frac{\partial}{\partial \mathbf{a}_j} L_2(\mathbf{p}(\mathbf{a}), \mathbf{y}) = 2(\mathbf{p}(\mathbf{a}) - \mathbf{y}) \odot \mathbf{p} \odot (1 - \mathbf{p}). \quad (6.4)$$

So if the model correctly predicts a low probability that the $ry = j$, i.e. that $\mathbf{p}_j \approx 0$, then the gradient is also close to zero.

Another property that could potentially be interesting is that the softmax output is invariant under additive changes of its input vector: $\text{softmax}(\mathbf{a}) = \text{softmax}(\mathbf{a} + b)$ when b is a scalar added to all the elements of vector \mathbf{a} . 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 strongest filter output \mathbf{a}_{i^*} gets reinforced (because the exponential increases faster for larger values) and the other units get inhibited. This is analogous to the *lateral inhibition* that is believed to exist between nearby neurons in cortex, and at the extreme (when the \mathbf{a}_i 's are large in magnitude) becomes a form of *winner-take-all* (one of the outputs is 1 and the others is 0). A more computationally expensive form of competition is found with *sparse coding*, described in Section 20.3.

Neural Net Outputs as Parameters of a Conditional Distribution

In general, for any parametric probability distribution $P(\mathbf{y}|\boldsymbol{\omega})$ with parameters $\boldsymbol{\omega}$, we can construct a *conditional distribution* $P(\mathbf{y}|\mathbf{x})$ by making $\boldsymbol{\omega}$ a parametrized function of \mathbf{x} , and learning that function:

$$P(\mathbf{y}|\boldsymbol{\omega} = f_{\boldsymbol{\theta}}(\mathbf{x}))$$

where $f_\theta(\mathbf{x})$ is the *output* of a predictor, \mathbf{x} is its input, and \mathbf{y} can be thought of as a *target*. This established choice of words comes from the common cases of classification and regression, where $f_\theta(X)$ is really a prediction associated with Y , or its expected value. However, in general $\omega = f_\theta(X)$ may contain parameters of the distribution of Y other than its expected value. For example, it could contain its variance or covariance, in the case where Y is conditionally Gaussian. In the above examples, with the squared error loss, ω is the mean of the Gaussian which captures the conditional distribution of Y (which means that the variance is considered fixed, not a function of X). In the common classification case, ω contains the probabilities associated with the various events of interest.

Once we view things in this way, we automatically get as the natural cost function the negative log-likelihood $L(X, Y) = -\log P(Y|\omega = f_\theta(X))$. Besides θ , there could be other parameters that control the distribution of Y . For example, we may wish to learn the variance of a conditional Gaussian, even when the latter is not a function of X . In this case, the solution can be computed analytically because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations Y and their expected value (here $f_\theta(X)$). In other words, the conditional variance can be estimated from the mean squared error.

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 $\omega = P(Y = 1|X)$. In the Multinoulli case, ω is generally specified by a vector of probabilities 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 3.10.5. 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,

$$P(y|x) = \sum_{i=1}^N P(C = i|x) \mathcal{N}(y|\mu_i(x), \Sigma_i(x)),$$

the neural network must have three kinds of outputs, $P(C = i|x)$, $\mu_i(x)$, and $\Sigma_i(x)$, which must satisfy different constraints:

1. Mixture components $P(C = i|x)$: these form a Multinoulli distribution over the N different components C , and can typically be obtained by a softmax over an N -vector, to guarantee that these outputs are positive and sum to 1.
2. Means $\mu_i(x)$: these indicate the center or mean associated with each Gaussian component, and are unconstrained (typically with no non-linearity at all for these output units). If Y is a d -vector, then the network must output an $N \times d$ matrix containing all these N d -vectors.
3. Covariances $\Sigma_i(x)$: these specify the covariance matrix for each component i . In many models the variance is unconditional (does not depend on x) and diagonal or

even scalar. If it is diagonal or scalar, only positivity must be enforced (e.g., using the *softplus* non-linearity). If it 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 $\Sigma_i(x) = B_i(x)B_i^\top(x)$, where B_i is an unconstrained square matrix. One practical issue if the matrix is full is that computing the likelihood is expensive, requiring $O(d^3)$ computation for the determinant and inverse of $\Sigma_i(x)$ (or equivalently, and more commonly done, its eigendecomposition or that of $B_i(x)$).

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 10.6.7 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 Y is actually a tuple formed by multiple random variables Y_1, Y_2, \dots, Y_k , then one has to choose an appropriate form for their joint distribution, conditional on $X = x$. The simplest and most common choice is to assume that the Y_i are conditionally independent, i.e.,

$$P(Y_1, Y_2, \dots, Y_k|x) = \prod_{i=1}^k P(Y_i|x).$$

This brings us back to the single variable case, especially since the log-likelihood now decomposes into a sum of terms $\log P(Y_i|x)$. If each $P(Y_i|x)$ 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 Y_i share some common factors that can be represented in some hidden layer of the network (such as the top hidden layer). See Sections 6.5 and 7.12 for a deeper treatment of the notion of underlying factors of variation and multi-task training. 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. Any probability distribution $P_\omega(Y)$ parametrized by parameters ω can be turned into a conditional distribution $P_\theta(Y|x)$ by making ω a function $\omega = f_\theta(x)$ parametrized by θ . This is not only true of the simple parametric distributions we have seen above (Gaussian, Bernoulli, Multinoulli) but also of more complex joint distributions typically represented by a graphical model. Much of this book is devoted to such graphical models (see Chapters 14, 19, 20, 21). For more concrete examples of such *structured output* models with deep learning, see Section 13.4.

6.2.3 Training Criterion and Regularizer

The loss function (often interpretable as a negative log-likelihood) tells us what we would like the learner to capture. Maximizing the conditional log-likelihood over the true distribution, i.e., minimizing the expected loss $E_{X,Y}[-\log P_\theta(Y|X)]$, makes $P_\theta(Y|X)$ estimate the true $P(Y|X)$ associated with the unknown data generating distribution, within the boundaries of what the chosen family of functions allows. See Section 5.8.2 for a proof. In practice we cannot minimize this expectation because we do not know $P(X, Y)$ and because computing and minimizing this integral exactly would generally be intractable. Instead we are going to *approximately* minimize a **training criterion** $C(\theta)$ based on the empirical average of the loss (over the training set). The simplest such criterion is the average training loss $\frac{1}{n} \sum_{t=1}^n L(f_\theta(\mathbf{x}^{(t)}), y^{(t)})$, where the training set is a set of n examples $(\mathbf{x}^{(t)}, y^{(t)})$. However, better results can often be achieved by crippling the learner and preventing it from simply finding the best θ that minimizes the average training loss. This means that we combine the evidence coming from the data (the training set average loss) with some a priori preference on the different values that θ can take (the regularizer). If this concept (and the related concepts of generalization, overfitting and underfitting) are not clear, please return to Sections 5.3 and 5.6.2 for a refresher.

The most common regularizer is simply an additive term equal to a regularization coefficient λ times the squared norm of the parameters⁷, $\|\theta\|_2^2$. This regularizer is often called the **weight decay** or L2 weight decay or shrinkage because adding the squared L2 norm to the training criterion pushes the weights towards zero, in proportion to their magnitude. For example, when using stochastic gradient descent, each step of the update with regularizer term $\frac{\lambda}{2}\|\theta\|^2$ would look like

$$\theta \leftarrow \theta - \epsilon \nabla_\theta L(f_\theta(x), y) - \epsilon \lambda \theta$$

where ϵ is the learning rate (see Section 4.3). This can be rewritten

$$\theta \leftarrow \theta(1 - \epsilon \lambda) - \epsilon \nabla_\theta L(f_\theta(x), y)$$

where we see that the first term systematically shaves off a small proportion of θ before adding the gradient.

Another commonly used regularizer is the so-called L1 regularizer, which adds a term proportional to the L1 (absolute value) norm, $|\theta|_1 = \sum_i |\theta_i|$. The L1 regularizer also prefers values that are small, but whereas the L2 weight decay has only a weak preference for 0 rather than small values, the L1 regularizer continues pushing the parameters towards 0 with a constant gradient even when they get very small. As a consequence, it tends to bring some of the parameters to exactly 0 (how many depends on how large the regularization coefficient λ is chosen). When optimizing with an approximate iterative and noisy method such as stochastic gradient descent, no actual 0 is obtained. On the

⁷In principle, one could have different priors on different parameters, e.g., it is common to treat the output weights with a separate regularization coefficient, but the more hyper-parameters, the more difficult is their optimization, discussed in Chapter 12.

other hand, the L1 regularizer tolerates large values of some parameters (only additively removing a small quantity compared to the situation without regularization) whereas the L2 weight decay aggressively punishes and prevents large parameter values. The L1 and L2 regularizers can be combined, as in the so-called elastic net (Zou and Hastie, 2005), and this is commonly done in deep networks⁸.

Note how in the context of maximum likelihood, regularizers can generally be interpreted as Bayesian priors on the parameters $P(\theta)$ or on the learned function, as discussed in Sections 5.6.2 and 5.7. Later in this book we discuss regularizers that are data-dependent (i.e., cannot be expressed easily as a pure prior), such as the contractive penalty (Chapter 16) as well as regularization methods that are difficult to interpret simply as added terms in the training criterion, such as dropout (Section 7.11).

6.2.4 Optimization Procedure

Once a training criterion is defined, we enter the realm of iterative optimization procedures, with the slight twist that if possible we want to monitor held-out performance (for example the average loss on a validation set, which usually does not include the regularizing terms) and not just the value of the training criterion. Monitoring the validation set performance allows one to stop training at a point where generalization is estimated to be the best among the training iterations. This is called *early stopping* and is a standard machine learning technique, discussed in Sec. 7.8.

Section 4.3 has already covered the basics of gradient-based optimization. The simplest such technique is stochastic gradient descent (with minibatches), in which the parameters are updated after computing the gradient of the average loss over a mini-batch of examples (e.g. 128 examples) and making an update in the direction opposite to that gradient (or opposite to some accumulated average of such gradients, i.e., the momentum technique, reviewed in Section 12.5.3). The most commonly used optimization procedures for multi-layer neural networks and deep learning in general are either variants of stochastic gradient descent (typically with minibatches), second-order methods (the most commonly used being L-BFGS and nonlinear conjugate gradients) applied on large minibatches (e.g. of size 10000) or on the whole training set at once, or natural gradient methods (Amari, 1997; Park *et al.*, 2000; Le Roux *et al.*, 2008; Pascanu and Bengio, 2013). Exact second-order and natural gradient methods are computationally too expensive for large neural networks because they involve matrices of dimension equal to the square of the number of parameters. Approximate methods are discussed in Section 8.4.

On smaller datasets or when computing can be parallelized, second-order methods have a computational advantage because they are easy to parallelize and can still perform many updates per unit of time. On larger datasets (and in the limit of an infinite dataset, i.e., a stream of training examples) one cannot afford to wait for seeing the whole dataset before making an update, so that favors either stochastic gradient descent variants

⁸See the Deep Learning Tutorials at <http://deeplearning.net/tutorial/gettingstarted.html#l1-and-l2-regularization>

(possibly with minibatches to take advantage of fast or parallelized implementations of matrix-matrix products) or second-order methods with minibatches.

A more detailed discussion of issues arising with optimization methods in deep learning can be found in Chapter 8. In particular, many design choices in the construction of the family of functions, loss function and regularizer can have a major impact on the difficulty of optimizing the training criterion. Furthermore, instead of using generic optimization techniques, one can design optimization procedures that are specific to the learning problem and chosen architecture of the family of functions, for example by initializing the parameters of the final optimization routine from the result of a different optimization (or a series of optimizations, each initialized from the previous one). Because of the non-convexity of the training criterion, the initial conditions can make a very important difference, and this is what is exploited in the various *pre-training* strategies, Sections 8.6.4 and 17.1, as well as with curriculum learning (Bengio *et al.*, 2009), Section 8.7.

6.3 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 functions (for which computing a gradient is meaningful), and 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 loss L or the training criterion C) 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 θ). The partial derivative of C with respect to θ (called the gradient) tells us whether θ should be increased or decreased in order to decrease C , and is a crucial tool in optimizing the training objective. It can be readily proven that the back-propagation algorithm 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 C with respect to parameters θ can be *decomposed recursively* by taking into consideration the composition of functions that relate θ to C , via intermediate quantities that mediate that influence, e.g., the activations of hidden units in a deep neural network. In this section we call these intermediate values u_j (indexed by j) and consider the general case in which they form a directed acyclic graph that has C 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 $\frac{\partial C}{\partial u_j}$ when $\frac{\partial C}{\partial u_i}$ 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 $\frac{\partial C}{\partial u_N} = 1$ 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 .

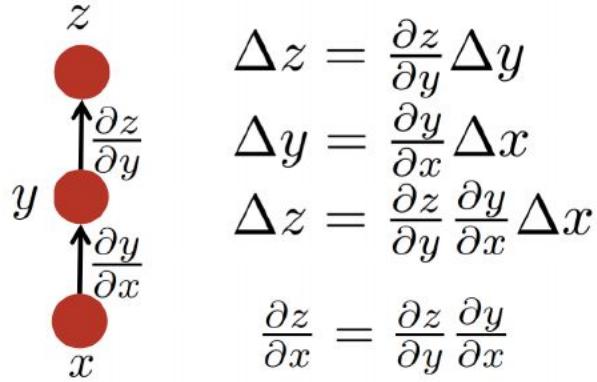


Figure 6.3: The chain rule, illustrated in the simplest possible case, with z a scalar function of y , which is itself a scalar function of x . A small change Δx in x gets turned into a small change Δy in y through the partial derivative $\frac{\partial y}{\partial x}$, from the first-order Taylor approximation of $y(x)$, and similarly for $z(y)$. Plugging the equation for Δy into the equation for Δz yields the chain rule.

6.3.1 Chain Rule

In order to apply the back-propagation algorithm, we take advantage of the **chain rule**:

$$\nabla_{\theta} C(g(\theta)) = \nabla_{g(\theta)} C(g(\theta)) \frac{\partial g(\theta)}{\partial \theta} \quad (6.5)$$

which works also when C , g or θ 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 θ will propagate into a small change in $g(\theta)$ by getting multiplied by $\frac{\partial g(\theta)}{\partial \theta}$. Similarly, a small change in $g(\theta)$ will propagate into a small change in $C(g(\theta))$ by getting multiplied by $\nabla_{g(\theta)} C(g(\theta))$. Hence a small change in θ first gets multiplied by $\frac{\partial g(\theta)}{\partial \theta}$ to obtain the change in $g(\theta)$ and this then gets multiplied by $\nabla_{g(\theta)} C(g(\theta))$ to obtain the change in $C(g(\theta))$. Hence the ratio of the change in $C(g(\theta))$ to the change in θ is the product of these partial derivatives. The partial derivative measures the locally linear influence of a variable on another. This is illustrated in Figure 6.3, with $x = \theta$, $y = g(\theta)$, and $z = C(g(\theta))$.

Now, if g is a vector, we can rewrite the above as follows:

$$\nabla_{\theta} C(g(\theta)) = \sum_i \frac{\partial C(g(\theta))}{\partial g_i(\theta)} \frac{\partial g_i(\theta)}{\partial \theta}$$

which sums over the influences of θ on $C(g(\theta))$ through all the intermediate variables $g_i(\theta)$. This is illustrated in Figure 6.4 with $x = \theta$, $y_i = g_i(\theta)$, and $z = C(g(\theta))$.

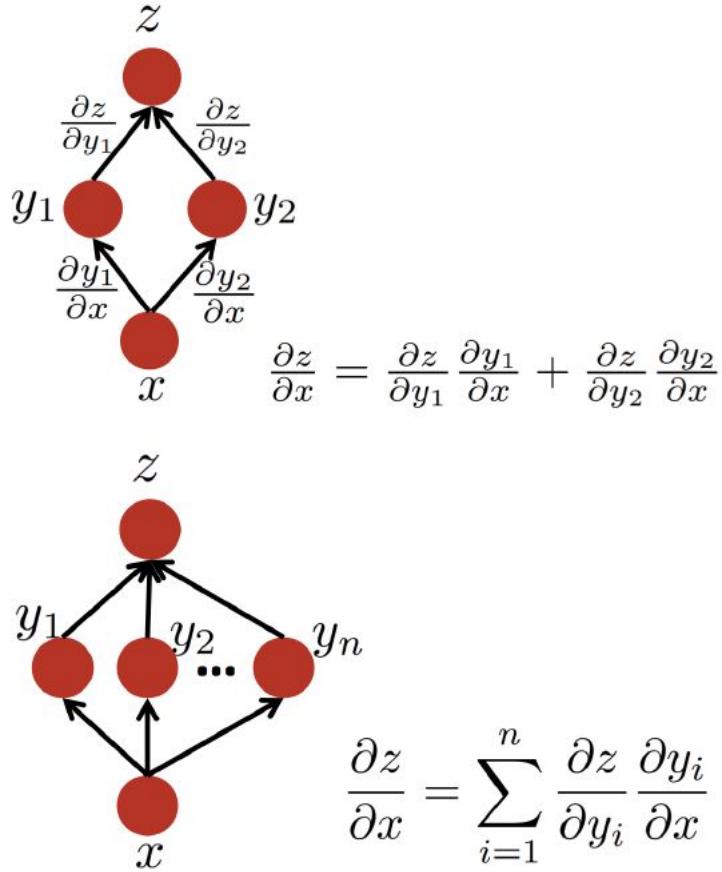


Figure 6.4: Top: The chain rule, when there are two intermediate variables y_1 and y_2 between x and z , creating two paths for changes in x to propagate and yield changes in z . Bottom: more general case, with n intermediate variables y_1 to y_n .

6.3.2 Back-Propagation in an MLP

Whereas example 6.1.1 illustrated the case of of an MLP with a single hidden layer let us consider in this section back-propagation for an ordinary but deep MLP, i.e., like the above vanilla MLP but with several hidden layers. For this purpose, we will recursively apply the chain rule illustrated in Figure 6.4. The algorithm proceeds by first computing the gradient of the cost C with respect to output units, and these are used to compute the gradient of C with respect to the top hidden layer activations. We can then 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 C 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

Algorithm 6.1 *Forward* computation associated with input \mathbf{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 \mathbf{h}_k to a pre-activation vector \mathbf{a}_k via a weight matrix $\mathbf{W}^{(k)}$ which is then transformed via f into \mathbf{h}_{k+1} . The input vector \mathbf{x} corresponds to \mathbf{h}_0 and the predicted outputs $\hat{\mathbf{y}}$ corresponds to \mathbf{h}_M . The cost function $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on a target \mathbf{y} (see Section 6.2.2 for examples of loss functions). The loss may be added to a regularizer Ω (see Section 6.2.3 and Chapter 7) to obtain the example-wise cost C . Algorithm 6.2 shows how to compute gradients of C with respect to parameters \mathbf{W} and \mathbf{b} .

```

 $\mathbf{h}_0 = \mathbf{x}$ 
for  $k = 1 \dots, M$  do
     $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$ 
     $\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$ 
end for
 $\hat{\mathbf{y}} = \mathbf{h}^{(M)}$ 
 $C = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega$ 

```

transformation (defined by a bias vector $\mathbf{b}^{(k)}$ and a weight matrix $\mathbf{W}^{(k)}$ followed by a non-linearity f . In general, the non-linearity may be different on different layers, and this is typically the case for the output layer (see Section 6.2.1). Hence each unit at layer k computes an output $\mathbf{h}_i^{(k)}$ as follows:

$$\begin{aligned} \mathbf{a}_i^{(k)} &= b_i^{(k)} + \sum_j W_{ij}^{(k)} h_j^{(k-1)} \\ h_i^{(k)} &= f(a_i^{(k)}) \end{aligned} \tag{6.6}$$

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.

6.3.3 Back-Propagation in a General Flow Graph

More generally, we can think about decomposing a function $C(\theta)$ 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 $C(\theta)$ 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 `numpy` library can perform on arrays of numbers).

Algorithm 6.2 Backward computation for the deep neural network of Algorithm 6.1, which uses in addition to the input \mathbf{x} a target y . This computation yields the gradients on the activations $\mathbf{a}^{(k)}$ 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.

After the forward computation, compute the gradient on the output layer:

$$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} C = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, y) + \lambda \nabla_{\hat{\mathbf{y}}} \Omega$$

(typically Ω 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):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} C = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

 Compute gradients on weights and biases (including the regularization term, where needed):

$$\nabla_{\mathbf{b}^{(k)}} C = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega$$

$$\nabla_{\mathbf{W}^{(k)}} C = \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega$$

 Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} C = \mathbf{W}^{(k)\top} \nabla_{\mathbf{a}^{(k)}} C$$

end for

We will define the back-propagation in a general flow-graph, using the following generic notation: $u_i = f_i(a_i)$, where 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: $a_i = (u_j)_{j \in \text{parents}(i)}$. This is illustrated in Figure 6.5.

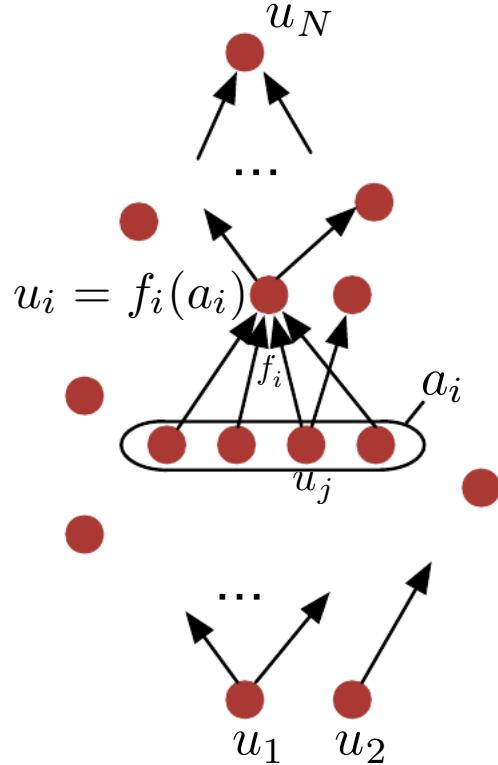


Figure 6.5: Illustration of recursive forward computation, where at each node u_i we compute a value $u_i = f_i(a_i)$, with a_i being the list of values from parents u_j of node u_i . Following Algorithm 6.3, the overall inputs to the graph are $u_1 \dots, u_M$ (e.g., the parameters we may want to tune during training), and there is a single scalar output u_N (e.g., the loss which we want to minimize).

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 $f_i(a_i)$ for some values in the vector a_i , we also need some code that tells us how to compute its partial derivatives, $\frac{\partial f_i(a_i)}{\partial a_{ik}}$ with respect to any immediate argument a_{ik} . Let $k = \pi(i, j)$ denote the index of u_j in the list 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, $\frac{\partial f_i(a_i)}{\partial a_{ik}}$ 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, one should keep clear in one's mind (and implementation) the notational and semantic distinction between a partial derivative that includes all paths and one that includes only the immediate effect of a function's argument on the function output, with the other arguments considered fixed. 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 $u_3 = f_3(u_2, u_1)$

Algorithm 6.3 Flow graph *forward* computation. Each node computes numerical value u_i by applying a function f_i to its argument list 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 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  $i = 1 \dots, M$  do
     $u_i \leftarrow x_i$ 
end for
for  $i = M + 1 \dots, N$  do
     $a_i \leftarrow (u_j)_{j \in \text{parents}(i)}$ 
     $u_i \leftarrow f_i(a_i)$ 
end for
return  $u_N$ 
```

and $u_2 = f_2(u_1)$, illustrated in Figure 6.6. The direct derivative of f_3 with respect to its argument $a_{3,2}$ is $\frac{\partial f_3}{\partial a_{3,2}} = e^{a_{3,1}+a_{3,2}}$ while if we consider the variables u_3 and u_1 to which these correspond, there are two paths from u_1 to u_3 , and we obtain as 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 .

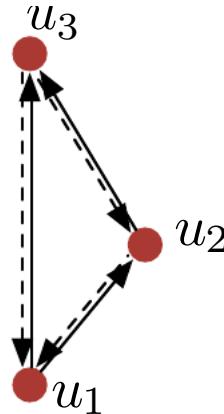


Figure 6.6: Illustration of indirect effect and direct effect of variable u_1 on variable u_3 in a flow graph, which means that the derivative of u_3 with respect to u_1 must include the sum of two terms, one for the direct effect (derivative of u_3 with respect to its first argument) and one for the indirect effect through u_2 (involving the product of the derivative of u_3 with respect to u_2 times the derivative of u_2 with respect to u_1). Forward computation of u_i 's (as in Figure 6.5) is indicated with upward full arrows, while backward computation (of derivatives with respect to u_i 's, as in Figure 6.7) is indicated with downward dashed arrows.

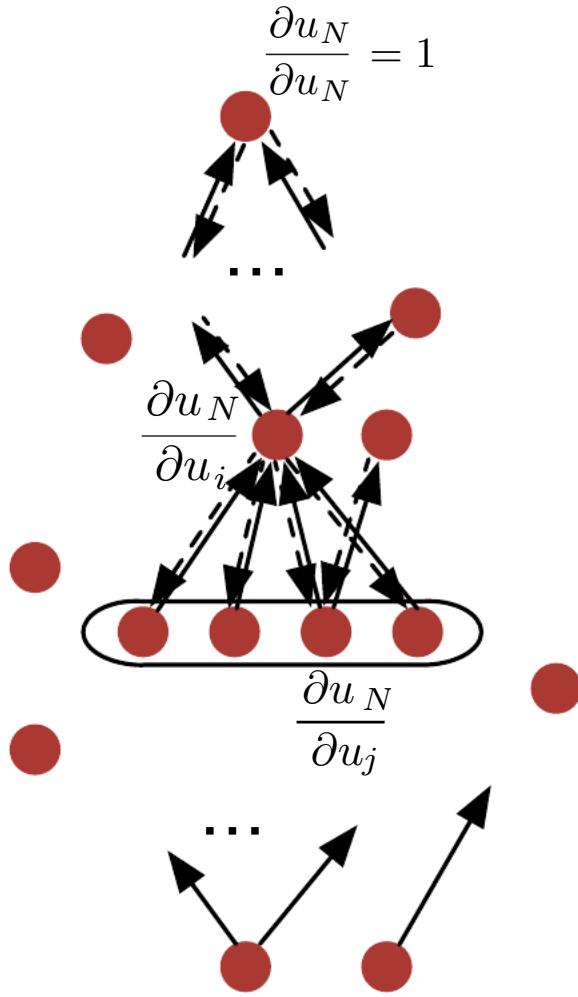


Figure 6.7: Illustration of recursive backward computation, where we associate to each node j not just the values u_j computed in the forward pass (Figure 6.5, bold upward arrows) but also the gradient $\frac{\partial u_N}{\partial u_j}$ with respect to the output scalar node u_N . These gradients are recursively computed in exactly the opposite order, as described in Algorithm 6.4 by using the already computed $\frac{\partial u_N}{\partial u_i}$ 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 6.7.

Algorithm 6.4 *Back-propagation* computation of a flow graph (full, upward arrows), 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, \dots, 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  $\left( \frac{\partial u_N}{\partial u_i} \right)_{i=1}^M$ 

```

This recursion is a form of efficient factorization of the total gradient, i.e., it is an application of the principles of dynamic programming⁹. Indeed, the derivative of the output node with respect to any node can also be written down in this intractable form:

$$\frac{\partial u_N}{\partial u_i} = \sum_{\text{paths } u_{k_1}, \dots, u_{k_n}: k_1 = i, k_n = N} \prod_{j=2}^n \frac{\partial u_{k_j}}{\partial u_{k_{j-1}}}$$

where the paths u_{k_1}, \dots, u_{k_n} go from the node $k_1 = i$ to the final node $k_n = N$ in the flow graph. 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 re-use 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). In that case the equations remain valid, and the multiplication of scalar partial derivatives becomes the multiplication of a row vector of gradients $\frac{\partial u_N}{\partial u_i}$ with a Jacobian of partial derivatives associated with the $j \rightarrow i$ arc of the graph, $\frac{\partial f_i(a)}{\partial a_{i,\pi(i,j)}}$. In the case where minibatches are used during training, u_i would actually be a whole matrix (the extra dimension being for the examples in the minibatch). This would then turn the basic computation into matrix-matrix products rather than matrix-vector products, and the

⁹ 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.

former can be computed much more efficiently than a sequence of matrix-vector products (e.g. with the BLAS library), especially so on modern computers and GPUs, that rely more and more on parallelization through many cores.

More implementation issues regarding the back-propagation algorithm are discussed in Chapters 11 and 12, regarding respectively GPU implementations (Section 11.2) and debugging tricks (Section 12.5.1). Section 12.5.2 also discusses a natural generalization of the back-propagation algorithm in which one manipulates not numbers but symbolic expressions, i.e., turning a program that performs a computation (decomposed as a flow graph expressed in the forward propagation algorithm) into another program (another flow graph) that performs gradient computation (i.e. automatically generating the back-propagation program, given the forward propagation graph). Using such symbolic automatic differentiation procedures (implemented for example in the `Theano` library¹⁰), one can compute second derivatives and other kinds of derivatives, as well as apply numerical stabilization and simplifications on a flow graph.

6.4 Universal Approximation Properties and Depth

When there is no hidden layer, and for convex loss functions and regularizers (which is typically the case), we obtain a convex training criterion. We end up with linear regression, logistic regression or other log-linear models that are used in many applications of machine learning. This is appealing (no local minima, no saddle point) and convenient (no strong dependency on initial conditions) but comes with a major disadvantage: such learners are very limited in their ability to represent more complex functions. In the case of classification tasks, we are limited to linear decision surfaces. In practice, this limitation can be somewhat circumvented by handcrafting a large enough or discriminant enough set of hardwired features extracted from the raw input and on which the linear predictor can be easily trained. See next section for a longer discussion about feature learning.

To make the family of functions rich enough, the other option is to introduce one or more hidden layers. It can be proven (White, 1990; Barron, 1993; Girosi, 1994) that with a single hidden layer *of a sufficient size* and a reasonable choice of non-linearity (including the sigmoid, hyperbolic tangent, and RBF unit), one can represent any smooth function to any desired accuracy (the greater the required accuracy, the more hidden units are required). However, these theorems generally do not tell us how many hidden units will be required to achieve a given accuracy for particular data distributions: in fact the worse case scenario is generally going to require an exponential number of hidden units (to basically record every input configuration that needs to be distinguished). It is easier to understand how bad things can be by considering the case of binary input vectors: the number of possible binary functions on vectors $v \in \{0, 1\}^d$ is 2^{2^d} and selecting one such function requires 2^d bits, which will in general require $O(2^d)$ degrees of freedom.

However, machine learning is not about learning any possible function with equal

¹⁰<http://deeplearning.net/software/theano/>

ease: we mostly care about the kinds of functions that are needed to represent the world around us or the task of interest. The **no-free-lunch theorems** for machine learning (Wolpert, 1996) essentially say that no learning algorithm is “universal”, i.e., dominates all the others against all possible ground truths (data generating distribution or target function). However, for a given family of tasks, some learning algorithms are definitely better. Choosing a learning algorithm is equivalent to choosing a prior, i.e., making a larger bet for some guesses of the underlying target function than for others. If the target function is likely under the chosen prior, then this prior (i.e., this learning algorithm) turns out to be a good choice. The best choice is the unrealistic prior that puts all probability mass on the true target function. This would only happen if there is nothing to be learned any more because we already know the target function. Having a universal approximation property just means that this prior is broad enough to cover or come close to any possible target function – and this is a useful property – but is, in itself, clearly not sufficient to provide good generalization (not to speak of the optimization issues and other computational concerns that may be associated with a choice of prior and learning algorithm).

One of the central priors that is exploited in deep learning is that the target function to be learned can be efficiently represented as a deep composition of simpler functions (“features”), where features at one level can be re-used to define many features at the next level. This is connected to the notion of *underlying factors* described in the next section. One therefore assumes that these factors or features are organized at multiple levels, corresponding to multiple levels of representation. The number of such levels is what we call **depth** in this context. The computation of these features can therefore be laid down in a flow graph or circuit, whose depth is the length of the longest path from an input node to an output node. Note that we can define the operations performed in each node in different ways. For example, do we consider a node that computes the affine operations of a neural net followed by a node that computes the non-linear neuron activation, or do we consider both of these operations as one node or one level of the graph? Hence the notion of depth really depends on the allowed operations at each node and one flow graph of a given depth can be converted into an equivalent flow graph of a different depth by redefining which operations can be performed at each node. However, for neural networks, we typically talk about a depth 1 network if there is no hidden layer, a depth 2 network if there is one hidden layer, etc. The universal approximation properties for neural nets basically tell us that depth 2 is sufficient to approximate any reasonable function to any desired finite accuracy.

From the point of view of approximation properties, the important result is that one can find families of functions which can be approximated very efficiently when a particular depth is allowed, but which might require a much larger (typically exponentially larger) model (e.g. more hidden units) if depth is insufficient (or is limited to 2). Such results have been proven for logic gates (Håstad, 1986), linear threshold units with non-negative weights (Håstad and Goldmann, 1991), polynomials (Delalleau and Bengio, 2011) organized as deep sum-product networks (Poon and Domingos, 2011), and more recently, for deep networks of rectifier units (Pascamu *et al.*, 2013b). 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. However, a lot of experimental evidence suggests that better generalization can be obtained with more depth, for many such AI tasks (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; Ebrahimi *et al.*, 2013). This suggests that indeed depth is a useful prior and that in order to take advantage of it, the learner’s family of function needs to allow multiple levels of representation.

6.5 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 problem¹¹, i.e., an optimization problem with some convergence guarantees towards a global optimum, irrespective of initial conditions. Simple and well-understood optimization algorithms are available in this case. However, this limits the representational capacity too much: many tasks, for a given choice of input representation 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 x to $\phi(x)$, where $\phi(x)$ is of much higher dimension. In this case, $f_\theta(x) = b + w \cdot \phi(x)$ 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 $\phi(x)$ (or even an infinite-dimensional one) so long as the kernel $K(u, v) = \phi(u) \cdot \phi(v)$ (where \cdot is the appropriate dot product for the space of $\phi(\cdot)$) can be computed efficiently. If $\phi(x)$ 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 ϕ as a feature space for our task. Kernel machines theory clearly identifies the choice of ϕ 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 $K(u, v) = e^{-\|u-v\|/\sigma^2}$. Unfortunately, this prior may be insufficient, i.e., too broad and sensitive to the curse of dimensionality, as introduced in Section 5.12 and developed in more detail in Chapter 17.
2. Another option is to *manually engineer the representation or features* $\phi(x)$. 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

¹¹or even one for which an analytic solution can be computed, with linear regression or the case of some Gaussian process regression models

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 a lot more detail in [Bengio and LeCun \(2007b\)](#), 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 6.4). Other priors are discussed in Chapter 17.

This whole discussion is clearly not specific to neural networks and supervised learning, and is one of the central motivations for this book.

6.6 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 very 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, rectified linear units. Such units consist of two linear pieces and their behavior is driven by a single weight vector. [Jarrett et al. \(2009a\)](#) 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. \(2009a\)](#) observed that 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 becomes relevant because it can extract more knowledge from it, and learning typically beats fixed or random settings of 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, and the response of such a network to a change in a single parameter may be highly nonlinear even in a small neighborhood.

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.

This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks, which need to propagate information through several time steps, are much easier to train when they are more linear. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation—a particular straightforward kind of linear activation. This is discussed further in Section [10.6.4](#).

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, but piecewise linear units are now by far the most popular kind of hidden units.

6.7 Historical Notes

Chapter 7

Regularization

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. The main strategy for achieving good generalization is known as *regularization*. Regularization consists of putting extra constraints on a machine learning model, such as restrictions of parameter values or extra terms in the cost function, that are not designed to help fit the training set. If chosen carefully, these extra constraints can lead to improved performance on the test set, either by encoding prior knowledge into the model, or by forcing the model to consider multiple hypotheses that explain the training data. Sometimes regularization also helps to make an underdetermined problem determined.

This chapter builds on the concepts of generalization, overfitting, underfitting, bias and variance introduced in Chapter 5. If you are not already familiar with these notions, please refer to that chapter before continuing with the more advance material presented here.

Simply put, regularizers work 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 the situation where the model family being trained either (1) excluded true data generating process – corresponding to underfitting and inducing bias, or (2) matched to the true data generating process – the “Goldilocks” model space, or (3) was more complex than the generating process – the regime where variance dominates the estimation error (as measured by the MSE – see Section 5.5).

Note that an overly complex model family does not necessarily include (or even come close to) the target function or the true data generating process. In practice, we almost never have access to the true data generating process so we can never know if our 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 the model family we are training does not include the data generating process. We can assume that – to some extend – 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, i.e. 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.

Regularization is a method of limiting the domain of these parameters in such a way as to limit the capacity of the model. With respect to minimizing the empirical risk, regularization induces bias in an attempt to limit variance that results from using a finite dataset.

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 $\hat{f}(\mathbf{x})$ parametrized by a vector of parameters $\boldsymbol{\theta}$. The data consists of inputs $\mathbf{x}^{(i)}$ and (for some tasks) targets $y^{(i)}$ for $i \in \{1, \dots, n\}$. In the case of classification, each $y^{(i)}$ is an integer class label in $\{1, \dots, k\}$. For regression tasks, each $y^{(i)}$ is a real number. In the case of a density estimation task, there are no targets. We may group these examples into a design matrix \mathbf{X} and a vector of targets \mathbf{y} .

In deep learning, we are mainly interested in the case where $\hat{f}(\mathbf{x})$ 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 either require very large datasets so that the data can fully specify such complicated models, or they require careful regularization.

7.1 Classical Regularization: Parameter Norm Penalty

Regularization has been used for decades prior to the advent of deep learning. Traditional 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 loss function J . We denote the regularized loss function by \tilde{J} :

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\boldsymbol{\theta}) \quad (7.1)$$

where α is a hyperparameter that weighs the relative contribution of the norm penalty term, Ω , relative to the standard loss function $J(\mathbf{x}; \boldsymbol{\theta})$. The hyperparameter α should be a non-negative real number, with $\alpha = 0$ corresponding to no regularization, and larger values of α corresponding to more regularization.

When our training algorithm minimizes the regularized loss function \tilde{J} it will decrease both the original loss 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 Ω can result in different solutions being preferred.

For models such as linear or logistic regression, where $\boldsymbol{\theta} = [\mathbf{w}^\top, b]^\top$, we typically choose $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|_2^2$ or $\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1$. That is, we leave the biases unregularized ¹, and penalize half² the squared L^2 or the L^1 norm.

In the following sections, we discuss the effects of the various norms when used as penalties on the weights.

7.1.1 L^2 Parameter Regularization

One of the simplest and most common kind of classical regularization is the L^2 parameter norm penalty.³, $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|_2^2$. This form of regularization is also known as *ridge regression*. It is equally applicable to neural networks, where the penalty is equal to the sum of the squared L^2 of all of the weight vectors. In the context of neural networks, this is known as *weight decay*. Typically, for neural networks, we use a different coefficient α for the weights at each layer of the network. This coefficient should be tuned using a validation set.

We can gain some insight into the behaviour of weight decay regularization by considering the gradient of the regularized loss function. To simplify the presentation, we assume a linear model with no bias term, so $\boldsymbol{\theta}$ is just \mathbf{w} . Such a model has the following gradient:

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \quad (7.2)$$

We will further simplify the analysis by considering a quadratic approximation to the loss function in the neighborhood of the empirically optimal value of the weights \mathbf{w}^* . (If the loss is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect).

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \quad (7.3)$$

where \mathbf{H} is the Hessian matrix of J with respect to \mathbf{w} evaluated at \mathbf{w}^* . There is no first order term in this quadratic approximation, because \mathbf{w}^* is defined to be a minimum,

¹The biases typically require less data to fit accurately than the weights. Each weight specifies how two variables interact, and requires observing both variables in a variety of conditions to fit well. Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. Regularizing the biases can introduce a significant amount of underfitting. For example, making sparse features usually requires being able to set the biases to significantly negative values.

²The $\frac{1}{2}$ in the L^2 penalty may seem arbitrary. Conceptually, it is not necessary and could be folded into the α hyperparameter. However, the $\frac{1}{2}$ results in a simpler gradient (\mathbf{w} instead of $2\mathbf{w}$) and simplifies the interpretation of the penalty as being a Gaussian prior on \mathbf{w} .

³More generally, we could consider regularizing the parameters to a parameter value $\mathbf{w}^{(o)}$ that is perhaps not zero. In that case the L^2 penalty term would be $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w} - \mathbf{w}^{(o)}\|_2^2 = \frac{1}{2}\sum_i (\mathbf{w}_i - \mathbf{w}_i^{(o)})^2$. Since it is far more common to consider regularizing the model parameters to zero, we will focus on this special case in our exposition.

where the gradient vanishes. Likewise, because \mathbf{w}^* is a minimum, we can conclude that \mathbf{H} is positive semi-definite.

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.4)$$

If we replace the exact gradient in equation 7.2 with the approximate gradient in equation 7.4, we can write an equation for the location of the minimum of the regularized loss function:

$$\alpha \mathbf{w} + \mathbf{H}(\mathbf{w} - \mathbf{w}^*) = 0 \quad (7.5)$$

$$(\mathbf{H} + \alpha \mathbf{I})\mathbf{w} = \mathbf{H}\mathbf{w}^* \quad (7.6)$$

$$\tilde{\mathbf{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H}\mathbf{w}^* \quad (7.7)$$

The presence of the regularization term moves the optimum from \mathbf{w}^* to $\tilde{\mathbf{w}}$. As α approaches 0, $\tilde{\mathbf{w}}$ approaches \mathbf{w}^* . But what happens as α grows? Because \mathbf{H} is real and symmetric, we can decompose it into a diagonal matrix Λ and an ortho-normal basis of eigenvectors, \mathbf{Q} , such that $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$. Applying the decomposition to equation 7.7, we obtain:

$$\begin{aligned} \mathbf{w} &= (\mathbf{Q}\Lambda\mathbf{Q}^\top + \alpha \mathbf{I})^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \mathbf{w}^* \\ &= \left[\mathbf{Q}(\Lambda + \alpha \mathbf{I})\mathbf{Q}^\top \right]^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \mathbf{w}^* \\ &= \mathbf{Q}(\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*, \\ \mathbf{Q}^\top \mathbf{w} &= (\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*. \end{aligned} \quad (7.8)$$

If we interpret the $\mathbf{Q}^\top \mathbf{w}$ as rotating our parameters \mathbf{w} into the basis as defined by the eigenvectors \mathbf{Q} of \mathbf{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 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 loss are preserved relatively intact. In directions that do not contribute to reducing the loss, 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 \mathbf{H} is captured in the concept of the *effective number of parameters*, defined to be

$$\gamma = \sum_i \frac{\lambda_i}{\lambda_i + \alpha}. \quad (7.9)$$

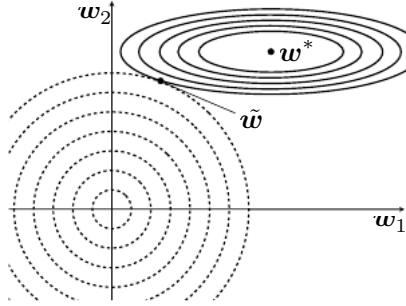


Figure 7.1: An illustration of the effect of L2 (or weight decay) regularization on the value of the optimal \mathbf{w} .

As α is increased, the effective number of parameters decreases.

Another way to gain some intuition for the effect of L^2 regularization is to consider its effect on linear regression. The unregularized objective function for linear regression is the sum of squared errors:

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

When we add L^2 regularization, the objective function changes to

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha\mathbf{w}^\top \mathbf{w}.$$

This changes the normal equations for the solution from

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

to

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}.$$

We can see L^2 regularization causes the learning algorithm to “perceive” the input \mathbf{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.

TODO—make sure the chapter includes maybe a table showing relationships between early stopping, priors, constraints, penalties, and adding noise? e.g. look up L1 penalty and it tells you what prior it corresponds to scratchwork thinking about how to do it:

L2 penalty L2 constraint add noise early stopping Gaussian prior L1 penalty L1 constraint Laplace prior Max-norm penalty

7.1.2 L^1 Regularization

While L^2 regularization is the most common form of regularization for model parameters – such as the weights of a neural network. It is not the only form of regularization in common usage. L^1 regularization is another kind of penalization on model parameters that behaves differently from L^2 regularization.

Formally, L^1 regularization on the model parameter \mathbf{w} is defined as:

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |\mathbf{w}_i|. \quad (7.10)$$

That is, as the sum of absolute values of the individual parameters.⁴ We will now consider the effect of L^1 regularization on the simple linear model, with no bias term, that we considered 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, if we consider the gradient (actually the sub-gradient) on the regularized objective function $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$, we have:

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \beta \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{y}; \mathbf{w}) \quad (7.11)$$

where $\text{sign}(\mathbf{w})$ is simply sign of \mathbf{w} applied element-wise.

By inspecting Eqn. 7.11, 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 \mathbf{w} , instead it is a constant factor with a sign equal to $\text{sign}(\mathbf{w})$. One consequence of this form of the gradient is that we will not necessarily see clean solutions to quadratic forms of $\nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{y}; \mathbf{w})$ 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.

For the sake of comparison with L^2 regularization, we will again consider a simplified setting of a quadratic approximation to the loss function in the neighborhood of the empirical optimum \mathbf{w}^* . (Once again, if the 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

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.12)$$

where, again, \mathbf{H} is the Hessian matrix of J with respect to \mathbf{w} evaluated at \mathbf{w}^* . We will also make the further simplifying assumption that the Hessian is diagonal, $\mathbf{H} = \text{diag}([\gamma_1, \dots, \gamma_N])$, where each $\gamma_i > 0$. With this rather restrictive assumption, the solution of the minimum of the L^1 regularized loss function decomposes into a systems of equations of the form:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{1}{2} \gamma_i (\mathbf{w}_i - \mathbf{w}_i^*)^2 + \beta |\mathbf{w}_i|.$$

Which admits an optimal solution (for each dimension i) in the following form:

$$\mathbf{w}_i = \text{sign}(\mathbf{w}_i^*) \max(|\mathbf{w}_i^*| - \frac{\beta}{\gamma_i}, 0)$$

⁴As with L^2 regularization, we could consider regularizing the parameters to a value that is not zero, but instead to some parameter value $\mathbf{w}^{(o)}$. In that case the L^1 regularization would introduce the term $\Omega(\boldsymbol{\theta}) = \|\mathbf{w} - \mathbf{w}^{(o)}\|_1 = \beta \sum_i |\mathbf{w}_i - \mathbf{w}_i^{(o)}|$.

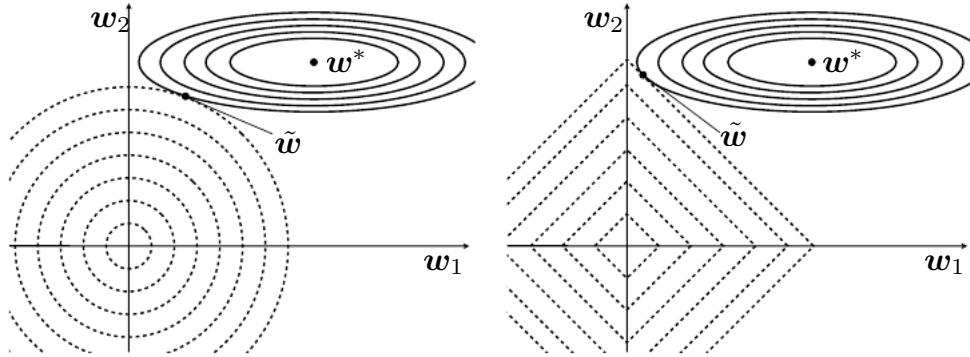


Figure 7.2: An illustration of the effect of L^1 regularization (RIGHT) on the value of the optimal \mathbf{w} , in comparison to the effect of L^2 regularization (LEFT).

Let's consider the situation where $\mathbf{w}_i^* > 0$ for all i , there are two possible outcomes. **Case 1:** $\mathbf{w}_i^* \leq \frac{\beta}{\gamma_i}$, here the optimal value of \mathbf{w}_i under the regularized objective is simply $\mathbf{w}_i = 0$, this occurs because the contribution of $J(\mathbf{w}; \mathbf{X}, \mathbf{y})$ to the regularized objective $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ is overwhelmed – in direction i , by the L^1 regularization which pushes the value of \mathbf{w}_i to zero. **Case 2:** $\mathbf{w}_i^* > \frac{\beta}{\gamma_i}$, here the regularization does not move the optimal value of \mathbf{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.

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 free parameters of the model that – through *normlone* regularization – with an optimal value (under the regularized objective) of zero. As we discussed, for each element i of the parameter vector, this happened when $\mathbf{w}_i^* \leq \frac{\beta}{\gamma_i}$. Comparing this to the situation for *normltwo* regularization, where (under the same assumptions of a diagonal Hessian \mathbf{H}) we get $\mathbf{w}_{L^2} = \frac{\gamma_i}{\gamma_i + \alpha} \mathbf{w}^*$, which is nonzero as long as \mathbf{w}^* is nonzero.

In Fig. 7.2, we see that even when the optimal value of \mathbf{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. Finally, L^1 is known as the only norm that is both sparsifying and convex for non-degenerative problems ⁵.

⁵For degenerative problems, where more than one solution exists, L^2 regularization can find the “sparse” solution in the sense that redundant parameters shrink to zero.

7.1.3 L^∞ Regularization

7.2 Classical Regularization as Constrained Optimization

Classical regularization adds a penalty term to the training objective:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\boldsymbol{\theta}).$$

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 ⁶, and a function representing whether the constraint is satisfied. If we wanted to constrain that $\Omega(\boldsymbol{\theta})$ is less than some constant k , we could construct a generalized Lagrange function

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k).$$

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 α . Specifically, α must increase whenever $\|\boldsymbol{\theta}\|_p > k$ and decrease whenever $\|\boldsymbol{\theta}\|_p < k$. However, after we have solved the problem, we can fix α^* and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha^* \Omega(\boldsymbol{\theta}).$$

This is exactly the same as the regularized training problem of minimizing \tilde{J} . Note that the value of α^* does not directly tell us the value of k . In principle, one can solve for k , but the relationship between k and α^* 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 α will result in a smaller constraint region, and smaller α 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 α 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 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

⁶KKT multipliers generalize Lagrange multipliers to allow for inequality constraints

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 θ 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.* (2012) recommend using constraints combined with a high learning rate to allow rapid exploration of parameter space while maintaining some stability.

TODO how L2 penalty is equivalent to L2 constraint (with unknown value), L1 penalty is equivalent to L1 constraint maybe move the earlier L2 regularization figure to here, now that the sublevel sets will make more sense? show the shapes induced by the different norms separate L2 penalty on each hidden unit vector is different from L2 penalty on all theta; is equivalent to a penalty on the max across columns of the column norms

7.3 Regularization from a Bayesian Perspective

In section 5.7, we briefly reviewed TODO- quick discussion of how to do Bayesian inference TODO- justification of MAP approximate Bayesian inference does it minimize some divergence under some constraint? or is it just a heuristic? maybe a figure showing the true posterior and a Dirac at the MAP TODO- specific priors leading to specific penalties, Gaussian -; L2, TODO- do we want to talk about ALL regularization or just “classical regularization”?

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 $\mathbf{X}^\top \mathbf{X}$. This is not possible whenever $\mathbf{X}^\top \mathbf{X}$ 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 \mathbf{X}) than input features (columns of \mathbf{X}). In this case, many forms of regularization correspond to inverting $\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I}$ 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. For example, consider logistic regression applied to a problem where the classes are linearly separable. If a weight vector \mathbf{w} is able to achieve perfect classification, then $2\mathbf{w}$ will also achieve perfect classification and higher likelihood. An iterative optimization

procedure like stochastic gradient descent will continually increase the magnitude of \mathbf{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. One way of generalizing the concept of matrix inversion to non-square matrices, the *Moore-Penrose pseudoinverse*, can be viewed as a form of *normltwo* regularization. The Moore-Penrose pseudo-inverse is defined as

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

Using the Moore-Penrose pseudoinverse we can generalize linear regression to underconstrained problems using minimal regularization by multiplying both sides of

$$\mathbf{X}\mathbf{w} = \mathbf{y}$$

by \mathbf{X}^+ to yield

$$\mathbf{w} = \mathbf{X}^+ \mathbf{y}.$$

When a true inverse for \mathbf{X} exists, this returns the exact solution. When \mathbf{X} is not invertible because no exact solution exists, this returns the \mathbf{w} corresponding to the least possible mean squared error. When \mathbf{X} is not invertible because many solutions exist, this returns \mathbf{w} with the minimum possible *normltwo* norm.

The Moore-Penrose pseudo-inverse is also closely related to the singular value decomposition. Specifically, if the SVD is given by $\mathbf{X} = \mathbf{U}\Sigma\mathbf{W}^\top$, then $\mathbf{X}^+ = \mathbf{W}\Sigma^+\mathbf{U}^\top$. To compute the pseudo-inverse of the diagonal matrix of singular values Σ , we simply replace each non-zero element of the diagonal with its reciprocal, and leave all zero elements equal to zero.

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 more fake data. 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 \mathbf{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 (\mathbf{x}, y) pairs easily just by transforming the \mathbf{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 are relevant to the classification problem. 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° 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.

For many classification and even some regression tasks, the task should still be possible to solve even if random noise is added to the input. Neural networks prove not to be very robust to noise, however. One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. 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.

In a multilayer network, it can often be beneficial to apply transformations such as noise to the hidden units, as well as the inputs. This can be viewed as augmenting the dataset as seen by the deeper layers.

When reading machine learning research papers, 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. It is important to look for 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. If algorithm A performs poorly with no dataset augmentation and algorithm B performs

well when combined with numerous synthetic transformations of the input, then it is likely the synthetic transformations and not algorithm B itself that cause the improved performance. Sometimes the line is blurry, such as when a new machine learning algorithm involves injecting noise into the inputs. In these cases, it's best to consider how generally applicable to the new algorithm is, and to make sure that pre-existing algorithms are re-run in as similar of conditions as possible.

TODO– tangent propagation

7.6 Classical Regularization as Noise Robustness

Some classical regularization techniques can be derived in terms of training on noisy inputs. For example, consider

TODO how L2 penalty and L1 penalty can be derived in different ways, noise on inputs, noise on weights results for deep nets, see "Training with noise is equivalent to Tikhonov regularization" by Bishop et al

7.7 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 make different errors on the test set to some extent.

Consider for example a set of k regression models. Suppose that each model makes an error ϵ_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 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 $c = v$, this reduces to v , and the model averaging does not help at all. But in the case where the errors are perfectly uncorrelated and $c = 0$, then the expected 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 of 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 cost function. Bagging is a method that allows the same kind of model and same kind of training algorithm and cost function to be re-used 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 that, with high probability, each dataset is missing some of the examples from the original dataset and also contains several duplicate examples. 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. 7.3 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 averaging 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* constructs an ensemble with higher capacity than the individual models.

7.8 Early Stopping as a Form of Regularization

When training large models with high capacity, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See Fig. 7.4 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, 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

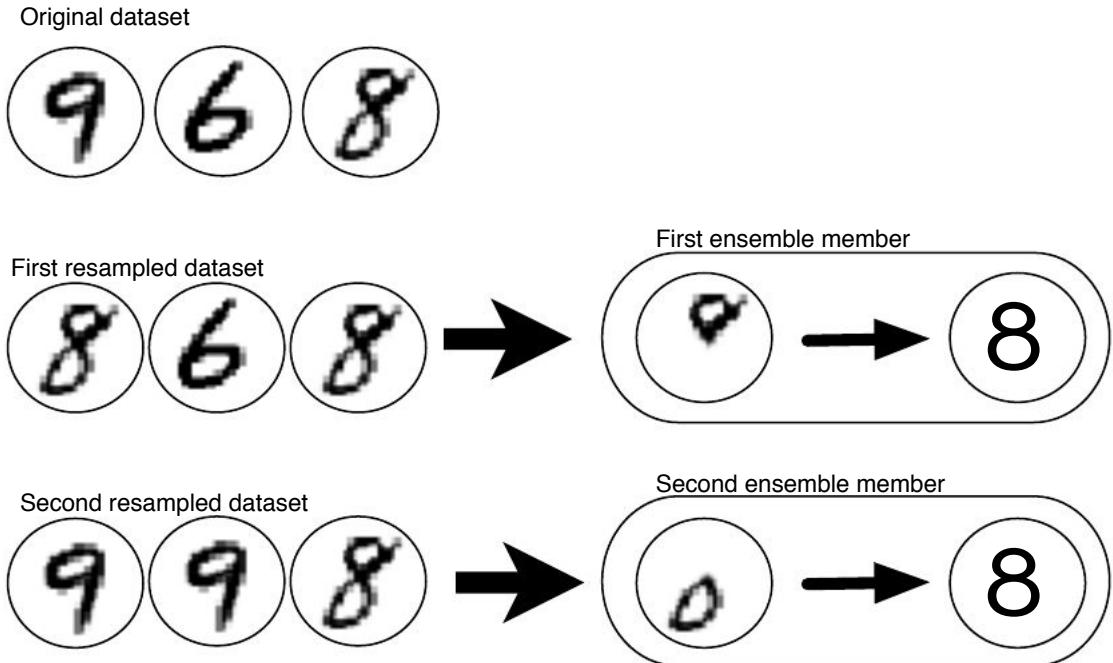


Figure 7.3: A cartoon depiction of how bagging works. Suppose we train an '8' detector on the dataset depicted above, containing an '8', a '6', and a '9'. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. 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.

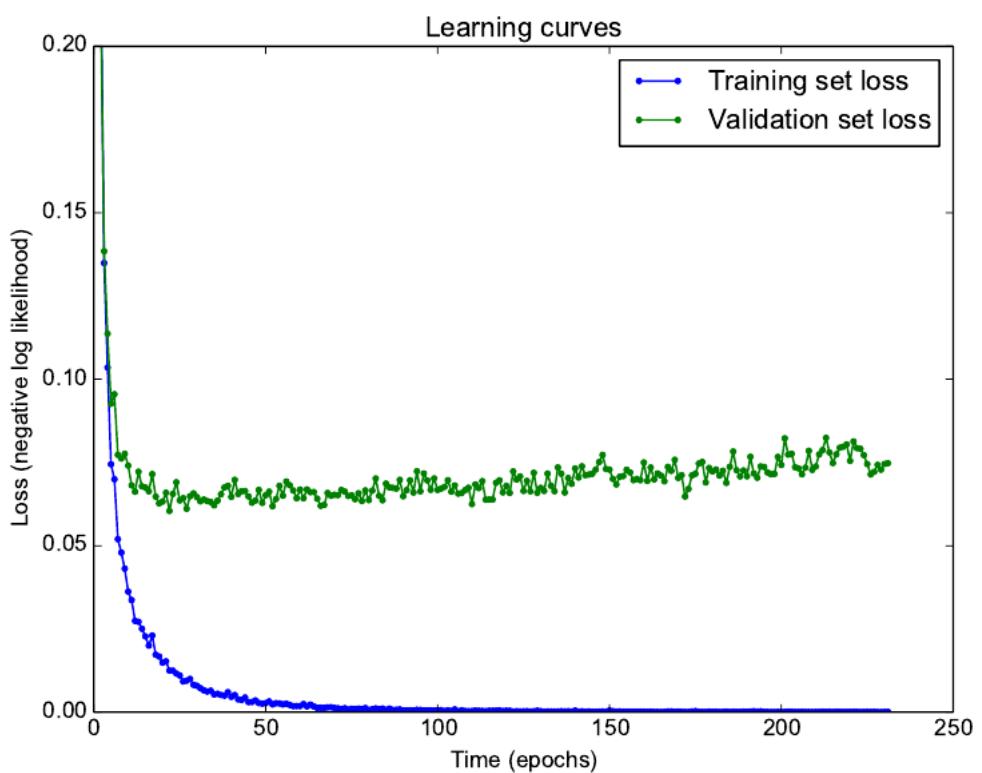


Figure 7.4: Learning curves showing how the negative log likelihood loss changes over time. In this example, we train a maxout network on MNIST, regularized with dropout. Observe that the training loss decreases consistently over time, but the validation set loss eventually begins to increase again.

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 θ_o be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

while $j < p$ **do**

 Update θ by running the training algorithm for n steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

if $v' < v$ **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

else

$j \leftarrow j + 1$

end if

end while

Best parameters are θ^* , 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. 7.4 that this hyperparameter has a U-shaped validation set performance curve, just like most other model capacity control parameters. 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.

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 have little effect on the total training time.

Early stopping is a very inobtrusive form of regularization, in that it requires no change to 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 minima 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's 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.

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 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.

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ be the training set
Split $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ into $\mathbf{X}^{(\text{subtrain})}$, $\mathbf{y}^{(\text{subtrain})}$, $\mathbf{X}^{(\text{valid})}$, $\mathbf{y}^{(\text{valid})}$
Run early stopping (Alg. 7.1) starting from random $\boldsymbol{\theta}$ using $\mathbf{X}^{(\text{subtrain})}$ and $\mathbf{y}^{(\text{subtrain})}$ for training data and $\mathbf{X}^{(\text{valid})}$ and $\mathbf{y}^{(\text{valid})}$ for validation data. This returns i^* , the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again
Train on $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ for i^* steps.

Algorithm 7.3 A meta-algorithm for using early stopping to determining at what objective value we start to overfit, then continuing training.

Let $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ be the training set
Split $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ into $\mathbf{X}^{(\text{subtrain})}$, $\mathbf{y}^{(\text{subtrain})}$, $\mathbf{X}^{(\text{valid})}$, $\mathbf{y}^{(\text{valid})}$
Run early stopping (Alg. 7.1) starting from random $\boldsymbol{\theta}$ using $\mathbf{X}^{(\text{subtrain})}$ and $\mathbf{y}^{(\text{subtrain})}$ for training data and $\mathbf{X}^{(\text{valid})}$ and $\mathbf{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$
 $\epsilon \leftarrow J(\boldsymbol{\theta}, \mathbf{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \mathbf{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$ **do**
 Train on $\mathbf{X}^{(\text{train})}$ and $\mathbf{y}^{(\text{train})}$ for n steps.
end while

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 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. Often the 0-1 loss continues to decrease for long after the log likelihood has begun to worsen on the validation set. TODO: figures. in figures/regularization, I have extracted the 0-1 loss but only used the nll for the regularization chapter's figures.

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 regularizer, but we have only backed up this claim by showing learning curves where the validation set error has a U-shaped curve. What is the actual mechanism by

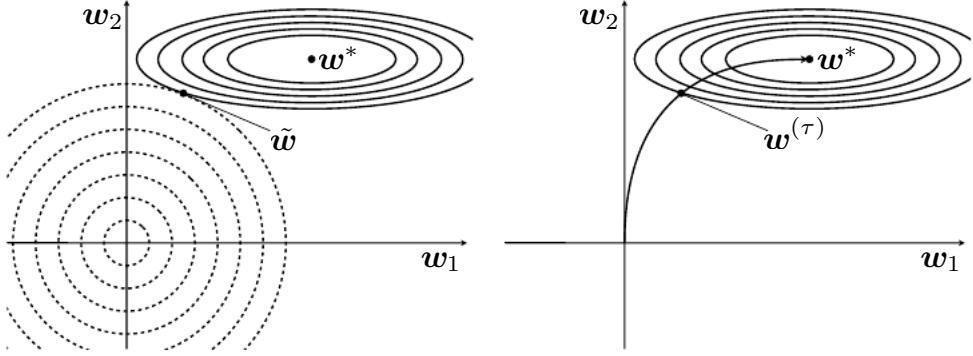


Figure 7.5: An illustration of the effect of early stopping (Right) as a form of regularization on the value of the optimal \mathbf{w} , as compared to L2 regularization (Left) discussed in Sec. 7.1.1.

which early stopping regularizes the model?⁷

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 τ optimization steps (corresponding to τ training iterations) and taking η as the learning rate. We can view the product $\eta\tau$ as the reciprocal of a regularization parameter. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from $\boldsymbol{\theta}_o$.

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 as seen in Section 7.1.1.

In order to compare with classical L^2 regularization, we again consider the simple setting where we will take as the parameters to be optimized as $\boldsymbol{\theta} = \mathbf{w}$ and we take a quadratic approximation to the objective function J in the neighborhood of the empirically optimal value of the weights \mathbf{w}^* .

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \quad (7.13)$$

where, as before, \mathbf{H} is the Hessian matrix of J with respect to \mathbf{w} evaluated at \mathbf{w}^* . Given the assumption that \mathbf{w}^* is a minimum of $J(\mathbf{w})$, we can consider that \mathbf{H} is positive semi-definite and that the gradient is given by:

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.14)$$

Let us consider initial parameter vector chosen at the origin, i.e. $\mathbf{w}^{(0)} = \mathbf{0}$. We will

⁷Material for this section was taken from [Bishop \(1995\)](#); [Sjöberg and Ljung \(1995\)](#), for further details regarding the interpretation of early-stopping as a regularizer, please consult these works.

consider updating the parameters via gradient descent:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}^{(\tau-1)}) \quad (7.15)$$

$$= \mathbf{w}^{(\tau-1)} - \eta \mathbf{H}(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.16)$$

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \eta \mathbf{H})(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.17)$$

Let us now consider this expression in the space of the eigenvectors of \mathbf{H} , i.e. we will again consider the eigendecomposition of \mathbf{H} : $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$, where Λ is a diagonal matrix and \mathbf{Q} is an ortho-normal basis of eigenvectors.

$$\begin{aligned} \mathbf{w}^{(\tau)} - \mathbf{w}^* &= (\mathbf{I} - \eta \mathbf{Q}\Lambda\mathbf{Q}^\top)(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \\ \mathbf{Q}^\top(\mathbf{w}^{(\tau)} - \mathbf{w}^*) &= (\mathbf{I} - \eta \Lambda)\mathbf{Q}^\top(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \end{aligned}$$

Assuming $\mathbf{w}^0 = 0$, and that $|1 - \eta \lambda_i| < 1$, we have after τ training updates, (TODO: derive the expression below).

$$\mathbf{Q}^\top \mathbf{w}^{(\tau)} = [\mathbf{I} - (\mathbf{I} - \eta \Lambda)^\tau] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.18)$$

Now, the expression for $\mathbf{Q}^\top \tilde{\mathbf{w}}$ in Eqn. 7.8 for L^2 regularization can rearrange as:

$$\begin{aligned} \mathbf{Q}^\top \tilde{\mathbf{w}} &= (\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^* \\ \mathbf{Q}^\top \tilde{\mathbf{w}} &= [\mathbf{I} - (\Lambda + \alpha \mathbf{I})^{-1} \alpha] \mathbf{Q}^\top \mathbf{w}^* \end{aligned} \quad (7.19)$$

Comparing Eqns 7.18 and 7.19, we see that if

$$(\mathbf{I} - \eta \Lambda)^\tau = (\Lambda + \alpha \mathbf{I})^{-1} \alpha,$$

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 λ_i are small (i.e. $\eta \lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx 1/\eta \alpha. \quad (7.20)$$

That is, under these assumptions, the number of training iterations τ plays a role inversely proportional to the L^2 regularization parameter.

Parameter values corresponding to directions of significant curvature (of the loss) 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.

7.9 Parameter Sharing

TODO: start with bayesian perspective (parameters should be close), add practical constraints to get parameter sharing.

7.10 Sparse Representations

TODO Most deep learning models have some concept of representations.

7.11 Dropout

Because deep models have a high degree of expressive power, 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 neural networks. Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a neural network, since training and evaluating a neural network is costly in terms of runtime and storing a neural network is costly in terms of 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 state 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.

TODO—describe training algorithm, with reference to bagging TODO—include figures from IG’s job talk TODO—training doesn’t rely on the model being probabilistic
TODO—describe inference algorithm, with reference to bagging TODO— inference does rely on the model being probabilistic. and specifically, exponential family?

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} :

$$P(Y = y | \mathbf{v}) = \text{softmax} \left(\mathbf{W}^\top \mathbf{v} + b \right)_y.$$

We can index into the family of sub-models by element-wise multiplication of the input with a binary vector d :

$$P(Y = 1 | \mathbf{v}; \mathbf{d}) = \text{softmax} \left(\mathbf{W}^\top \mathbf{d} \odot \mathbf{v} + b \right)_y.$$

The ensemble predictor is defined by re-normalizing the geometric mean over all ensemble members’ predictions:

$$P_{\text{ensemble}}(Y = y | \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(Y = y | \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(Y = y' | \mathbf{v})} \quad (7.21)$$

where

$$\tilde{P}_{\text{ensemble}}(Y = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(Y = y \mid \mathbf{v}; \mathbf{d})}.$$

To see that the weight scaling rule is exact, we can simplify $\tilde{P}_{\text{ensemble}}$:

$$\begin{aligned} \tilde{P}_{\text{ensemble}}(Y = y \mid \mathbf{v}) &= \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(Y = y \mid \mathbf{v}; \mathbf{d})} \\ &= \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \text{softmax}(\mathbf{w}^\top \mathbf{d} \odot \mathbf{v} + b)_y} \\ &= \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \frac{\exp(\mathbf{W}_{y,:}^\top \mathbf{d} \odot \mathbf{v} + b)}{\sum_{y'} \exp(\mathbf{W}_{y',:}^\top \mathbf{d} \odot \mathbf{v} + b)}} \\ &= \frac{\sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top \mathbf{d} \odot \mathbf{v} + b)}}}{\sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \sum_{y'} \exp(\mathbf{W}_{y',:}^\top \mathbf{d} \odot \mathbf{v} + b)}} \end{aligned}$$

Because \tilde{P} will be normalized, we can safely ignore multiplication by factors that are constant with respect to y :

$$\begin{aligned} \tilde{P}_{\text{ensemble}}(Y = y \mid \mathbf{v}) &\propto \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top \mathbf{d} \odot \mathbf{v} + b)} \\ &= \exp\left(\frac{1}{2^n} \sum_{\mathbf{d} \in \{0,1\}^n} \mathbf{W}_{y,:}^\top \mathbf{d} \odot \mathbf{v} + b\right) \\ &= \exp\left(\frac{1}{2} \mathbf{W}_{y,:}^\top \mathbf{v} + b\right) \end{aligned}$$

Substituting this back into equation 7.21 we obtain a softmax classifier with weights $\frac{1}{2}\mathbf{W}$.

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, and this approximation has not been theoretically characterized. Fortunately, it works well, empirically. Goodfellow *et al.* (2013a) found empirically that for deep networks with nonlinearities, the weight scaling rule can work better (in terms of classification accuracy) than Monte Carlo approximations to the ensemble predictor, even if the Monte Carlo approximation is allowed to sample up to 1,000 sub-networks.

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. As of this writing, the state of the art classification error rate on the permutation invariant

MNIST dataset (not using any prior knowledge about images) is attained by a classifier that uses both dropout regularization and deep Boltzmann machine pretraining. However, combining dropout with unsupervised pretraining has not become a popular strategy for larger models and more challenging datasets.

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 (Pascanu *et al.*, 2014a). This is very different from many other neural network regularization strategies, such as those based on unsupervised pretraining or semi-supervised learning. Such regularization strategies often impose restrictions such as not being able to use rectified linear units or max pooling. Often these restrictions incur enough harm to outweigh the benefit provided by the regularization strategy.

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. This is because the size of the optimal model (in terms of validation set error) is usually much larger, and because the number of steps required to reach convergence increases. This is of course to be expected from a regularization method, but it does mean that for very large datasets (as a rough rule of thumb, dropout is unlikely to be beneficial when more than 15 million training examples are available, though the exact boundary may be highly problem dependent) it is often preferable not to use dropout at all, just to speed training and reduce the computational cost of the final model.

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.

TODO– ”Dropout Training as Adaptive Regularization” ? (Wager *et al.*, 2013)
TODO–perspective as L2 regularization TODO–connection to adagrad? TODO–semi-supervised variant TODO–Baldi paper (Baldi and Sadowski, 2013) TODO–DWF paper (Warde-Farley *et al.*, 2014) TODO–using geometric mean is not a problem TODO–dropout boosting, it’s not just noise robustness TODO–what was the conclusion about mixability?

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 yielded a significant improvement or been 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.

TODO—improved performance with maxout units and probably ReLUs

7.12 Multi-Task Learning

Multi-task learning (Caruana, 1993) is a way to improve generalization by pooling the examples (i.e., constraints) arising out of several tasks.

Figure 7.6 illustrates a very common form of multi-task learning, in which different supervised tasks (predicting Y_i given X) share the same input X , as well as some intermediate-level representation 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, i.e., 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.*

TODO adversarial training

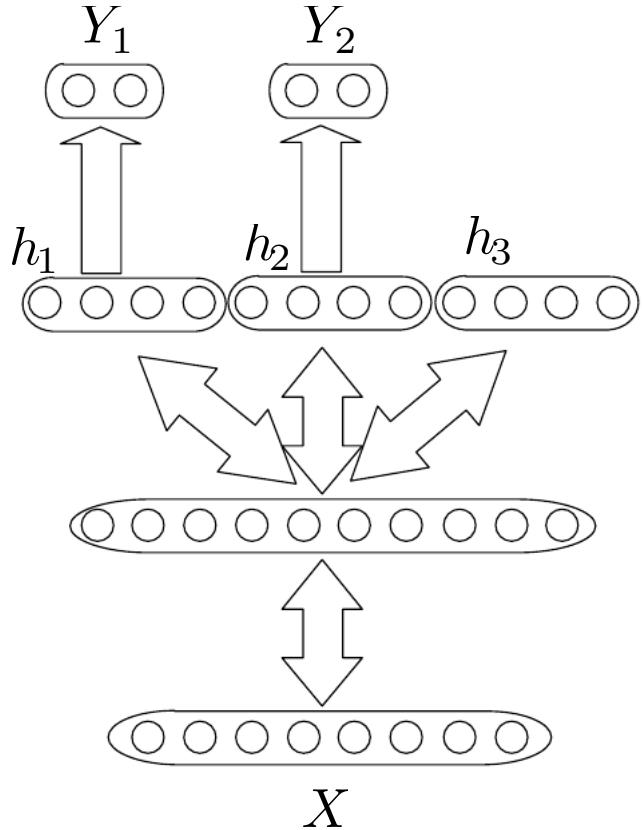


Figure 7.6: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters can be learned on top of a shared representation (associated respectively with h_1 and h_2 in the figure). The underlying assumption is that there exist a common pool of factors that explain the variations in the input X , while each task is associated with a subset of these factors. In the figure, it is additionally assumed that top-level hidden units are specialized to each task, while some intermediate-level representation 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 (h_3): these are the factors that explain some of the input variations but are not relevant for these tasks.

Chapter 8

Optimization for Training Deep Models

In Section 4.3, we saw a brief overview of numerical optimization—the task of iteratively searching for values of \mathbf{x} that result in maximal or minimal values of $f(\mathbf{x})$. That introductory section was fairly general and described the basic ideas that are used in a range of optimization problems. Optimization arises in many contexts in deep learning, not just in optimizing parameters but also in inference (which optimizes over the internal states of the network).

Here we present optimization algorithms that are intended specifically for *training* deep models. In this case, we will always minimize some cost function $J(\mathbf{X}^{(\text{train})}, \boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$.

8.1 Optimization for Model Training

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways.

8.1.1 Empirical Risk Minimization

Suppose that we have input feature \mathbf{x} , targets y , and some loss function $L(\mathbf{x}, y)$. Our ultimate goal is to minimize $\mathbb{E}_{\mathbf{x}, y \sim p(\mathbf{x}, y)}[L(\mathbf{x}, y)]$. This quantity is known as the *risk*. If we knew the true distribution $p(\mathbf{x}, y)$, this would be an optimization task solveable by an optimization algorithm. However, when we do not know $p(\mathbf{x}, y)$ 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 $p(\mathbf{x}, y)$ with the empirical distribution $\hat{p}(\mathbf{x}, y)$ defined by the training set. We now minimize the *empirical risk*

$$\mathbb{E}_{\mathbf{x}, y \sim \hat{p}(\mathbf{x}, y)}[L(\mathbf{x}, y)] = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, y^{(i)})$$

where m is the number of training examples.

This process 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.

TODO– make sure 0-1 loss is defined and in the index

8.1.2 Surrogate Loss Functions

TODO– coordinate with Yoshua / coordinate with MLP / ML chapters do we use term loss function = map from a specific example to a real number or do we use it interchangeably with objective function / cost function? it seems some literature uses "loss function" in a very general sense while others use it to mean specifically a single-example cost that you can take the expectation of, etc. this terminology seems a bit sub-optimal since it relies a lot on using English words with essentially the same meaning to represent different things with precise technical meanings are "surrogate loss functions" specifically replacing the cost for an individual examples, or does this also include things like minimizing the empirical risk rather than the true risk, adding a regularization term to the likelihood terms, etc.?

TODO– in some cases, surrogate loss function actually results in being able to learn more. for example, test 0-1 loss continues to decrease for a long time after train 0-1 loss has reached zero when training using log likelihood surrogate

In some cases, using a surrogate loss function allows us to extract more information

8.1.3 Generalization

TODO– SGD on an infinite dataset optimizes the generalization error directly (note that SGD is not introduced until later so this will need to be presented carefully) TODO–

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 a regularization method known as early stopping (see Sec. 7.8), 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.

8.1.4 Batches and Minibatches

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 decompose into a sum over each example: TODO equation, using same format as in original maximum likelihood section, which isn't written yet

TODO stochastic gradient descent

TODO examples can be redundant, so best computational efficiency comes from minibatches TODO too small of batch size -; bad use of multicore architectures TODO issues with Hessian being different for different batches, etc. TODO importance of shuffling shuffle-once versus shuffle per epoch todo: define the term "epoch"

8.1.5 Data Parallelism

TODO asynch implementations, hogwild, distbelief

8.2 Challenges in Optimization

8.2.1 Local Minima

TODO check whether this is already covered in numerical.tex

8.2.2 Ill-Conditioning

TODO this is definitely already covered in numerical.tex

8.2.3 Plateaus, Saddle Points, and Other Flat Regions

The long-held belief that neural networks are hopeless to train because they are fraught with local minima has been one of the reasons for the “neural networks winter” in the 1995–2005 decade. Indeed, one can show that there may be an exponentially large number of local minima, even in the simplest neural network optimization problems ([Sontag and Sussman, 1989](#); [Brady *et al.*, 1989](#); [Gori and Tesi, 1992](#)).

Theoretical work has shown that saddle points (and the flat regions surrounding them) are important barriers to training neural networks, and may be more important than local minima.

8.2.4 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. (TODO: REFER TO A PLOT FROM THE ILL-CONDITIONING SECTION WITH CONTOURS OF VALLEY). Second-order methods and momentum or gradient-averaging methods introduced in Section 8.4 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).

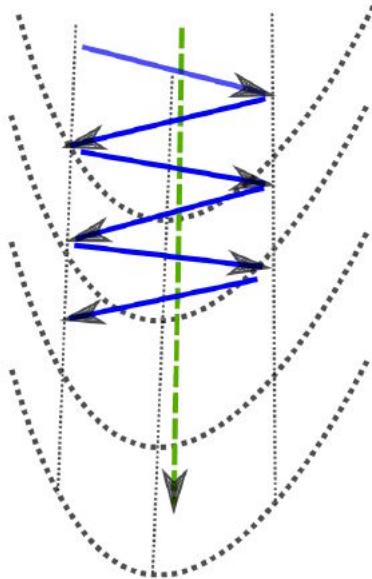


Figure 8.1: The traditional view of the optimization difficulty in neural networks is inspired by the ill-conditioning problem in quadratic optimization: some directions have a high curvature (second derivative), corresponding to the rising sides of the valley, and other directions have a low curvature, corresponding to the smooth slope of the valley. 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’s most paying in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations. The objective is to smoothly go down, staying at the bottom of the valley.

However, although classical second order methods can help, as shown in Figure 8.2, due to higher order derivatives, the objective function may have a lot more non-linearity, which 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.

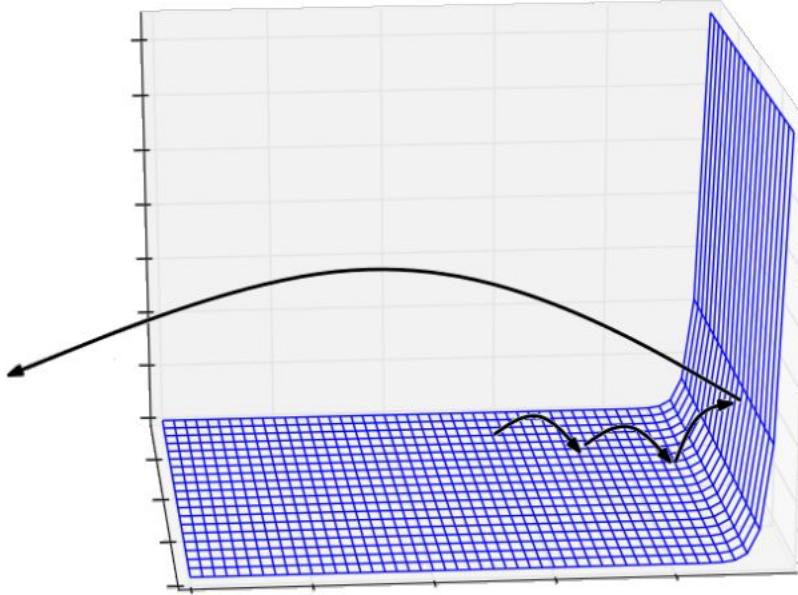


Figure 8.2: Contrary to what is shown in Figure 8.1, the cost function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. 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 θ 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 θ , it typically only remains like this (especially, keeping its sign) in a small region around θ . 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.

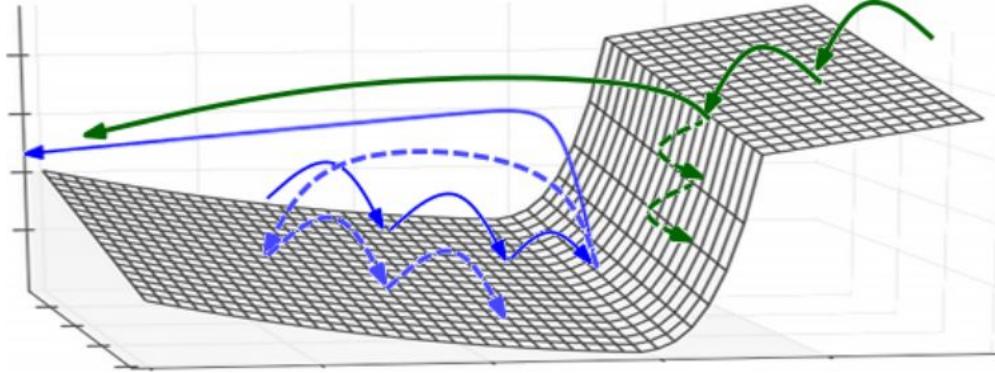


Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyper-parameter, although not a very critical one). This helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below. Figure graciously provided by Razvan Pascanu ([Pascanu, 2014](#)).

The gradient clipping heuristics are described in more detail in Section 10.6.7. 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.

8.2.5 Vanishing and Exploding Gradients - An Introduction to the Issue of 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 Section 10.6 for a deeper treatment along with a discussion of approaches that have been proposed to reduce this difficulty.

Exploding or Vanishing Product of Jacobians

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 more ups and downs. This arises simply because the Jacobian (matrix of derivatives) of a composition is the product of the Jacobians of each stage, i.e., if

$$f = f_T \circ f_{T-1} \circ \dots, f_2 \circ f_1$$

then the Jacobian matrix of derivatives of $f(x)$ with respect to its input vector x is the product

$$f' = f'_T f'_{T-1} \dots, f'_2 f_1 \quad (8.1)$$

where

$$f' = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

and

$$f'_t = \frac{\partial f_t(\mathbf{a}_t)}{\partial \mathbf{a}_t}$$

where $\mathbf{a}_t = f_{t-1}(f_{t-2}(\dots, f_2(f_1(\mathbf{x}))))$, i.e. composition has been replaced by matrix multiplication. This is illustrated in Figure 8.4.

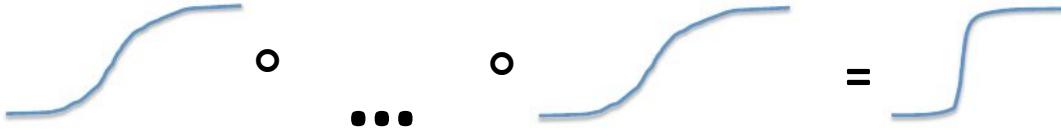


Figure 8.4: When composing many non-linearities (like the activation non-linearity in a deep or recurrent neural network), the result is highly non-linear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many ups and downs (not shown here).

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 α , this is obvious, since α^T goes to 0 if $\alpha < 1$ and goes to ∞ 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 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.

Consequence for Recurrent Networks: Difficulty of Learning Long-Term Dependencies

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, \dots, x_t, \dots , involving iterating over the transition operator:

$$s_t = F_\theta(s_{t-1}, x_t) \quad (8.2)$$

where s_t is called the state of the system and F_θ 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,

$$o_t = g_\omega(s_t), \quad (8.3)$$

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 θ of the recurrent function F_θ . One particular way to decompose the gradient $\frac{\partial L_T}{\partial \theta}$ using the chain rule is the following:

$$\begin{aligned} \frac{\partial L_T}{\partial \theta} &= \sum_{t \leq T} \frac{\partial L_T}{\partial s_t} \frac{\partial s_t}{\partial \theta} \\ \frac{\partial L_T}{\partial \theta} &= \sum_{t \leq T} \frac{\partial L_T}{\partial s_T} \frac{\partial s_T}{\partial s_t} \frac{\partial F_\theta(s_{t-1}, x_t)}{\partial \theta} \end{aligned} \quad (8.4)$$

where the last Jacobian matrix only accounts for the immediate effect of θ as a parameter of F_θ when computing $s_t = F_\theta(s_{t-1}, x_t)$, i.e., not taking into account the indirect effect of θ 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 Section 6.3, and note that we can construct a graph in which θ influences each s_t , each of which influences L_T via s_T . Now let us note that each Jacobian matrix $\frac{\partial s_T}{\partial s_t}$ can be decomposed as follows:

$$\frac{\partial s_T}{\partial s_t} = \frac{\partial s_T}{\partial s_{T-1}} \frac{\partial s_{T-1}}{\partial s_{T-2}} \cdots \frac{\partial s_{t+1}}{\partial s_t} \quad (8.5)$$

which is of the same form as Eq. 8.1 discussed above, i.e., which tends to either vanish or explode.

As a consequence, we see from Eq. 8.4 that $\frac{\partial L_T}{\partial \theta}$ 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 $\frac{\partial s_t}{\partial s_{t-1}}$ 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 vanilla recurrent net and stochastic gradient descent.

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). Section 10.6 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.

8.3 Optimization Algorithms

In Sec. 6.3, 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.

Deep learning

8.3.1 Gradient Descent

Gradient descent is the most basic gradient-based algorithms one might apply to train a deep model. The algorithm involves updating the model parameters θ (in the case of a deep neural network, these parameters would include the weights and biases associated with each layer) with a small step in the direction of the gradient of the loss function (including any regularization terms). For the case of supervised learning with data pairs

$[\mathbf{x}^{(t)}, \mathbf{y}^{(t)}]$ we have:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), \mathbf{y}^{(t)}; \boldsymbol{\theta}), \quad (8.6)$$

where ϵ is the *learning rate*, an optimization hyperparameter that controls the size of the step the parameters take in the direction of the gradient. Of course, following the gradient in this way is only guaranteed to reduce the loss in the limit as $\epsilon \rightarrow 0$.

learning rates

8.3.2 Stochastic Gradient Descent

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 decompose into a sum over each example:

TODO equation, using same format as in original maximum likelihood section, which isn't written yet

More general, we are really interested in minimizing the *expected* loss with the expectation taken with respect to the data distribution, i.e. $\mathbb{E}_{\mathbf{X}, \mathbf{Y}}[L(f_{\boldsymbol{\theta}}(\mathbf{X}), \mathbf{Y})]$, with $\mathbf{X}, \mathbf{Y} \sim P(\mathbf{X}, \mathbf{Y})$. As discussed in Sec. 6.1, we replace this expectation with an average over the training data (eg. for n examples):

$$\text{expected loss} = \frac{1}{n} \sum_{t=1}^n L(f(\mathbf{x}^{(t)}), \mathbf{y}^{(t)}) \quad (8.7)$$

This form of the loss implies that the gradient also consists of an average of the gradient contributions for each data point:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \text{expected loss} = \frac{1}{n} \sum_{t=1}^n \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^{(t)}), \mathbf{y}^{(t)}) \quad (8.8)$$

Now

So we can interpret the right hand side of Eqn. 8.8 as an estimator of the gradient of the expected loss. Seen in this light, it's reasonable to think about the properties of this estimator, such as its mean and variance.

Provided that there are a relatively large number of examples in the training set, computing the gradient over all examples in the training dataset – also known as batch gradient descent – would yeild a relatively small variance on the estimate

In application to training deep learning models, straightforward gradient descent – where each gradient step involves computing the gradient for all training examples – is well known to be inefficient. This is especially true when we are dealing with large datasets.

TODO stochastic gradient descent

Algorithm 8.1 Stochastic gradient descent (SGD) update at time t

Require: Learning rate η .

Require: Initial parameter θ

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$.

 Set $\mathbf{g} = \mathbf{0}$

for $t = 1$ to m **do**

 Compute gradient estimate: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$

end for

 Apply update: $\theta \leftarrow \theta - \eta \mathbf{g}$

end while

TODO examples can be redundant, so best computational efficiency comes from minibatches TODO too small of batch size - ζ , bad use of multicore architectures TODO issues with Hessian being different for different batches, etc.

TODO importance of shuffling shuffle-once versus shuffle per epoch todo: define the term “epoch”

discuss learning rates.

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, but consistent across minibatches. From the consistency of the gradient, we know that we can afford to take larger steps in this direction, yet we have no way of really knowing when we are in such situations.

The Momentum method 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 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.

Formally, we introduce a variable \mathbf{v} that plays the role of velocity (or momentum) that accumulates gradient. The update rule is given by:

$$\mathbf{v} \leftarrow +\alpha \mathbf{v} + \eta \nabla_{\theta} \left(\frac{1}{n} \sum_{t=1}^n L(\mathbf{f}(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)}) \right)$$

$$\theta \leftarrow \theta + \mathbf{v}$$

where the

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate η , momentum parameter α .

Require: Initial parameter θ , initial velocity v .

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$.

 Set $g = 0$

for $t = 1$ to m **do**

 Compute gradient estimate: $g \leftarrow g + \nabla_{\theta} L(f(x^{(t)}; \theta), y^{(t)})$

end for

 Compute velocity update: $v \leftarrow \alpha v - \eta g$

 Apply update: $\theta \leftarrow \theta + v$

end while

Nesterov momentum Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov.

$$v \leftarrow +\alpha v + \eta \nabla_{\theta} \left(\frac{1}{n} \sum_{t=1}^n L(f(x^{(t)}; \theta + \alpha v), y^{(t)}) \right),$$

$$\theta \leftarrow \theta + v,$$

where the parameters α and η play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate η , momentum parameter α .

Require: Initial parameter θ , initial velocity v .

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$.

 Apply interim update: $\theta \leftarrow \theta + \alpha v$

 Set $g = 0$

for $t = 1$ to m **do**

 Compute gradient (at interim point): $g \leftarrow g + \nabla_{\theta} L(f(x^{(t)}; \theta), y^{(t)})$

end for

 Compute velocity update: $v \leftarrow \alpha v - \eta g$

 Apply update: $\theta \leftarrow \theta + v$

end while

8.3.4 Adagrad

8.3.5 RMSprop

Algorithm 8.4 The Adagrad algorithm

Require: Global learning rate η ,

Require: Initial parameter θ

Initialize gradient accumulation variable $r = \mathbf{0}$,

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$.

 Set $\mathbf{g} = \mathbf{0}$

for $t = 1$ to m **do**

 Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$

end for

 Accumulate gradient: $r \leftarrow r + \mathbf{g}^2$

 Compute update: $\Delta\theta \leftarrow -\frac{\eta}{\sqrt{r}} \mathbf{g}$. % ($\frac{1}{\sqrt{r}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta\theta_t$

end while

8.3.6 Adadelta

8.3.7 No Pesky Learning Rates

8.4 Approximate Natural Gradient and Second-Order Methods

8.5 Conjugate Gradients

8.6 BFGS

TONGA and “actual” NG, links with HF.

8.6.1 New

8.6.2 Optimization Strategies and Meta-Algorithms

8.6.3 Coordinate Descent

In some cases, it may be possible to solve an optimization problem quickly by breaking it into separate pieces. If we minimize $f(\mathbf{x})$ 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

Algorithm 8.5 The RMSprop algorithm

Require: Global learning rate η , decay rate ρ .

Require: Initial parameter θ

Initialize accumulation variables $r = 0$

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$.

 Set $\mathbf{g} = \mathbf{0}$

for $t = 1$ to m **do**

 Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$

end for

 Accumulate gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g}^2$

 Compute parameter update: $\Delta \theta = -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$. % ($\frac{1}{\sqrt{\mathbf{r}}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$

end while

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 $f(\mathbf{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + y_1^2)$ where α is a positive constant. As α approaches 0, coordinate descent ceases to make any progress at all, while Newton's method could solve the problem in a single step.

8.6.4 Greedy Supervised Pre-training

TODO

8.7 Hints and Curriculum Learning

TODO

Algorithm 8.6 RMSprop algorithm with Nesterov momentum

Require: Global learning rate η , decay rate ρ , momentum para α .

Require: Initial parameter θ , initial velocity v .

```
Initialize accumulation variable  $r = \mathbf{0}$ 
while Stopping criterion not met do
    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .
    Compute interim update:  $\theta \leftarrow \theta + \alpha v$ 
    Set  $\mathbf{g} = \mathbf{0}$ 
    for  $t = 1$  to  $m$  do
        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$ 
    end for
    Accumulate gradient:  $r \leftarrow \rho r + (1 - \rho) \mathbf{g}^2$ 
    Compute velocity update:  $v \leftarrow \alpha v - \frac{\eta}{\sqrt{r}} \odot \mathbf{g}$ . % ( $\frac{1}{\sqrt{r}}$  applied element-wise)
    Apply update:  $\theta \leftarrow \theta + v$ 
end while
```

Algorithm 8.7 The Adadelta algorithm

Require: Decay rate ρ , constant ϵ

Require: Initial parameter θ

```
Initialize accumulation variables  $r = \mathbf{0}$ ,  $s = \mathbf{0}$ ,
while Stopping criterion not met do
    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ .
    Set  $\mathbf{g} = \mathbf{0}$ 
    for  $t = 1$  to  $m$  do
        Compute gradient:  $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$ 
    end for
    Accumulate gradient:  $r \leftarrow \rho r + (1 - \rho) \mathbf{g}^2$ 
    Compute update:  $\Delta\theta = -\frac{\sqrt{s+\epsilon}}{\sqrt{r+\epsilon}} \mathbf{g}$  % (operations applied element-wise)
    Accumulate update:  $s \leftarrow \rho s + (1 - \rho) [\Delta\theta]^2$ 
    Apply update:  $\theta \leftarrow \theta + \Delta\theta$ 
end while
```

Algorithm 8.8 The vSGD-1 algorithm from Schaul *et al.* (2012)

Require: Initial parameter θ_0

Initialize accumulation variables $q = \mathbf{0}$, $r = \mathbf{0}$, $s = \mathbf{0}$

while Stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$.

 Initialize the gradient $\mathbf{g} = \mathbf{0}$

for $t = 1$ to m **do**

 Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$

end for

 Accumulate gradient: $\mathbf{q} \leftarrow \rho \mathbf{q} + (1 - \rho) \mathbf{g}$

 Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g}^2$

 Accumulate: $\mathbf{s} \leftarrow \rho \mathbf{s} + (1 - \rho) \left| \text{bbprop}(\theta)_i^{(j)} \right|$

 estimate learning rate (element-wise calc.): $\eta^* \leftarrow \frac{\mathbf{q}^2}{\mathbf{s} \odot \mathbf{r}}$

 Update memory size: $\rho \leftarrow \left(\frac{\mathbf{q}^2}{\mathbf{r}} - 1 \right)^{-1} (1 - \rho)$

 Compute update: $\Delta\theta = -\eta^* \mathbf{g}$

 % All operations above should be interpreted as element-wise.

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Algorithm 8.9 Conjugate gradient method

Require: Initial parameters θ_0

Initialize $\rho_0 = \mathbf{0}$

while stopping criterion not met **do**

 Initialize the gradient $\mathbf{g} = \mathbf{0}$

for $t = 1$ to n % loop over the training set. **do**

 Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta} L(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}^{(t)})$

end for backpropagation

 Compute $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^\top \mathbf{g}_t}{\mathbf{g}_{t-1}^\top \mathbf{g}_{t-1}}$ (Polak — Ribi  re)

 Compute search direction: $\alpha_t = -\mathbf{g}_t + \beta_t \rho_{t-1}$

 Perform line search to find: $\eta^* = \text{argmin}_{\eta} J(\theta_t + \eta \rho_t)$

 Apply update: $\theta_{t+1} = \theta_t + \eta^* \rho_t$

end while

Algorithm 8.10 BFGS method

Require: Initial parameters $\boldsymbol{\theta}_0$

Initialize inverse Hessian $\mathbf{M}_0 = \mathbf{I}$

while stopping criterion not met **do**

 Compute gradient: $\mathbf{g}_t = \nabla J(\boldsymbol{\theta}_t)$ (via batch backpropagation)

 Compute $\boldsymbol{\phi} = \mathbf{g}_t - \mathbf{g}_{t-1}$, $\Delta = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$

 Approx \mathbf{H}^{-1} : $\mathbf{M}_t = \mathbf{M}_{t-1} + \left(1 + \frac{\boldsymbol{\phi}^\top \mathbf{M}_{t-1} \boldsymbol{\phi}}{\Delta^\top \boldsymbol{\phi}}\right) \frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{\Delta^\top \boldsymbol{\phi}} - \left(\frac{\Delta \boldsymbol{\phi}^\top \mathbf{M}_{t-1} + \mathbf{M}_{t-1} \boldsymbol{\phi} \Delta^\top}{\Delta^\top \boldsymbol{\phi}}\right)$

 Compute search direction: $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$

 Perform line search to find: $\eta^* = \operatorname{argmin}_\eta J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$

 Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$

end while

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. See chapter 13.1.2 for details. 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 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. We 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 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 an array of data and the kernel is usually an array of learnable parameters. 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 .

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 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 ma-

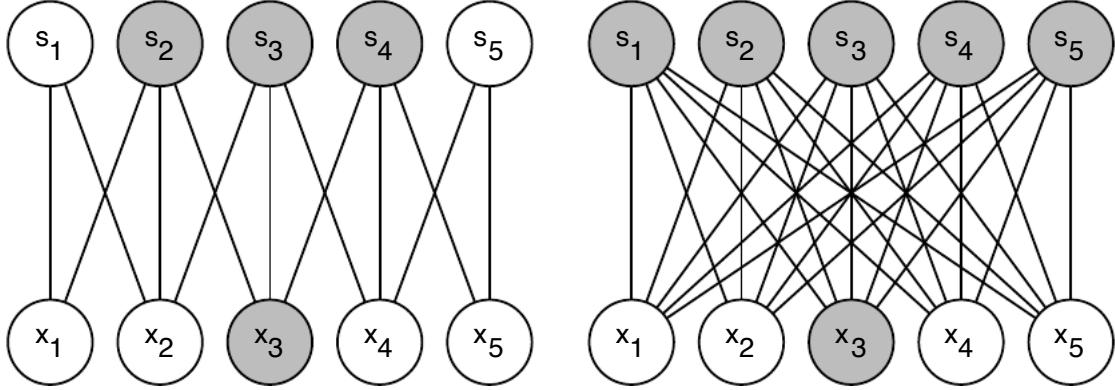


Figure 9.1: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3 , and also highlight the output units in s that are affected by this unit. (Left) When s is formed by convolution with a kernel of width 3, only three outputs are affected by x_3 . (Right) When s is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3 .

chine learning task while keeping k several orders of magnitude smaller than m . For graphical demonstrations of sparse connectivity, see Fig. 9.1 and Fig. 9.2.

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 usually 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. 9.3.

As an example of both of these first two principles in action, Fig. 9.4 shows how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

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

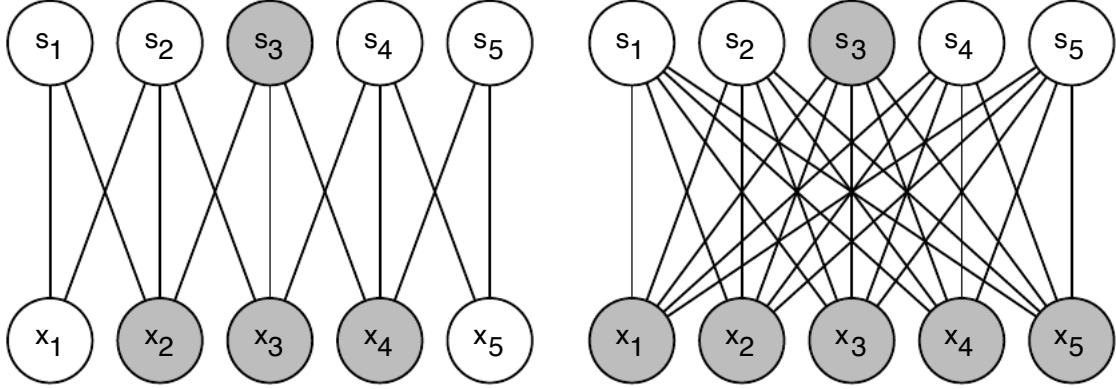


Figure 9.2: *Sparse connectivity, viewed from above:* We highlight one output unit, s_3 , and also highlight the input units in \mathbf{x} that affect this unit. These units are known as the *receptive field* of s_3 . (Left) When \mathbf{s} is formed by convolution with a kernel of width 3, only three 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 .

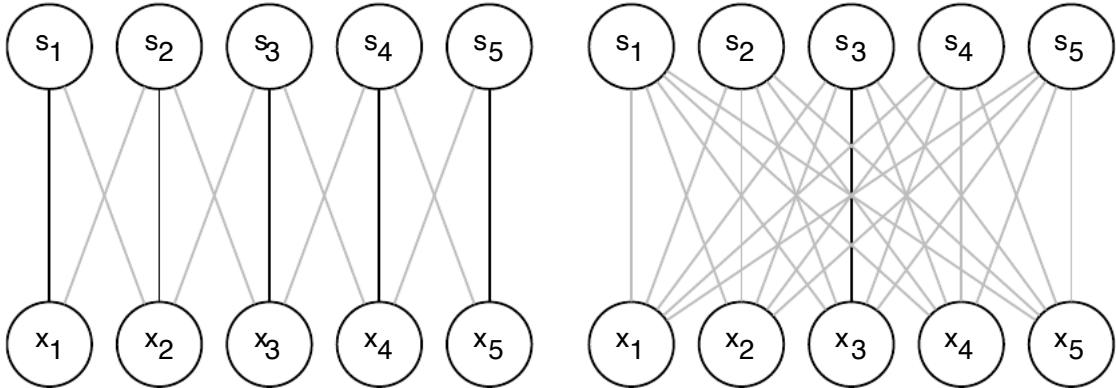


Figure 9.3: *Parameter sharing:* We highlight the connections that use a particular parameter in two different models. (Left) We highlight uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. (Right) 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.

convolution, if we let g be any function that translates 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, and an edge looks the same regardless of where it appears in the image. This property is not always useful. For example, if we want to recognize a face, some portion of the network needs to vary with spatial location, because the top of a face does not look the same as the bottom of a face—the part of the network processing 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.

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. This view of convolution as an infinitely strong prior makes it clear that the efficiency improvements of convolution come with a caveat: convolution is only applicable when the assumptions made by this prior are close to true. The use of convolution constrains the class of functions that the layer can represent. If the function that a layer needs to learn is indeed a local, translation invariant function, then the layer will be dramatically more efficient if it uses convolution rather than matrix multiplication. If the necessary function does not have these properties, then using a convolutional layer will cause the model to have high training error.

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 9.5.

9.3 Pooling

A typical layer of a convolutional network consists of three stages (see Fig. 9.5). 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



Figure 9.4: *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.

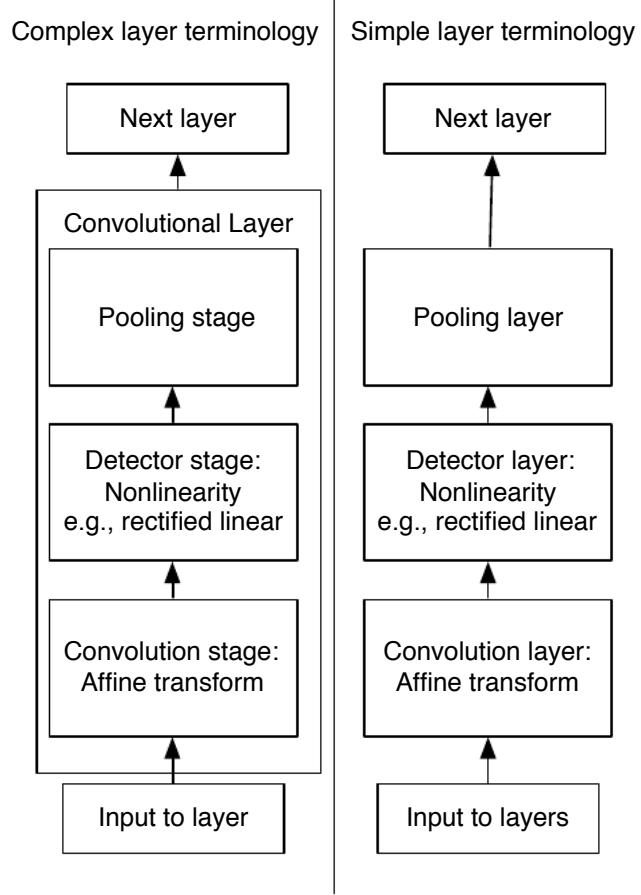


Figure 9.5: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. *Left)* In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many “stages.” In 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.

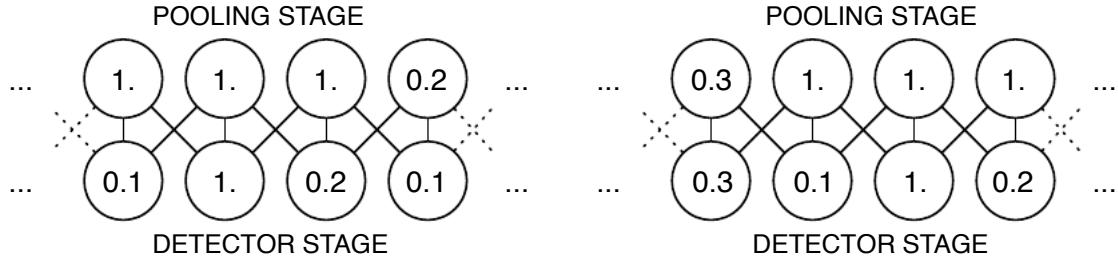


Figure 9.6: *Max pooling introduces invariance.* Left: A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of 1 between pooling regions and a pooling region width of 3. Right: A view 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.

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. 9.6 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. 9.7).

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. 9.8 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

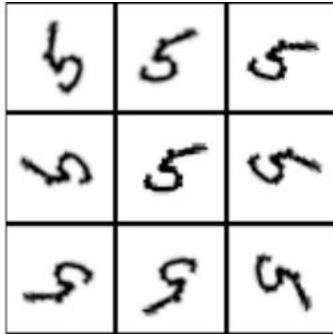


Figure 9.7: *Example of learned invariances:* If each of these filters drive units that appear in the same max-pooling region, then the pooling unit will detect “5”s in any rotation. By learning to have each filter be a different rotation of the “5” template, this pooling unit has learned to be invariant 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.

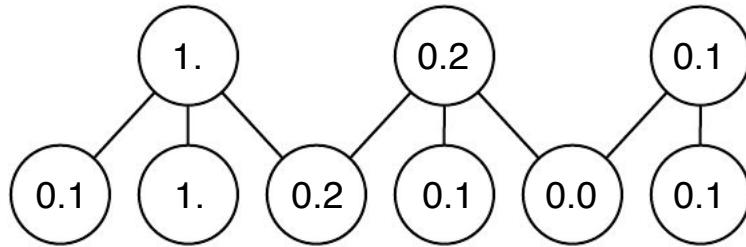


Figure 9.8: *Pooling with downsampling.* Here we use max-pooling with a pool width of 3 and a stride between pools of 2. This reduces the representation size by a factor of 2, which reduces the computational and statistical burden on the next layer. Note that the final pool has a smaller size, but must be included if we do not want to ignore some of the detector units.

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 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.

9.4 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 arrays, 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 arrays, 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.

Convolution is traditionally defined to index the kernel in the opposite order as the input. This definition makes convolution commutative, which can be convenient for writing proofs. In the context of neural networks, the commutative property of convolution is not especially helpful, so many implementations of convolution index both the kernel and the input in the same order.

Assume we have a 4-D kernel array \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 array 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). If we want

to sample only every s pixels in each direction, then we can defined a downsampled convolution function c such that:

$$z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [v_{l,j \times s + m, k \times s + n} k_{i,l,m,n}]. \quad (9.1)$$

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. 9.9 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 1×1 , 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 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 array \mathbf{W} : TODO—DWF wants better explanation of the 6 dimensions

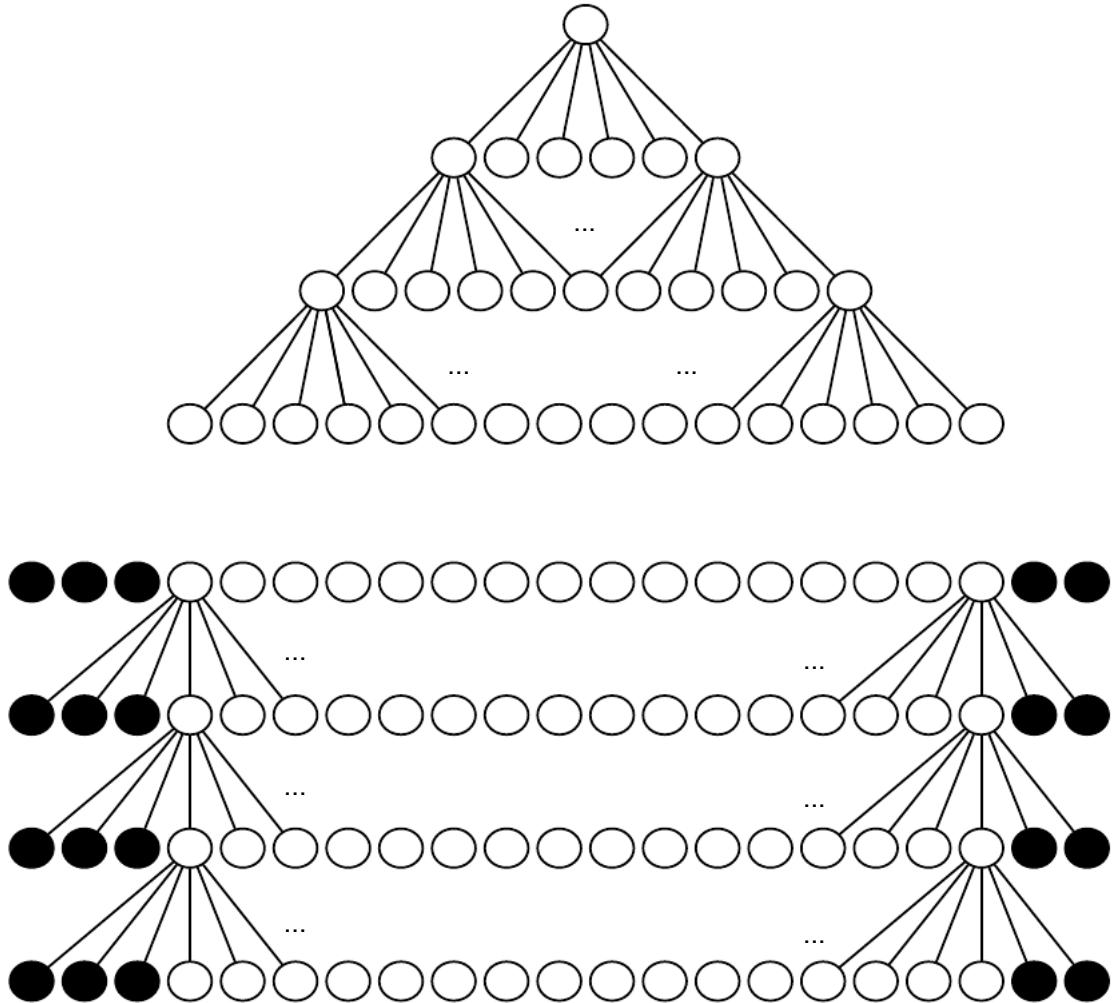


Figure 9.9: *The effect of zero padding on network size:* Consider a convolutional network with a kernel of width six at every layer. In this example, do not use any pooling, so only the convolution operation itself shrinks the network size. *Top)* In this convolutional network, we do not use any implicit 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.

$$z_{i,j,k} = \sum_{l,m,n} [v_{l,j+m,k+n} w_{i,j,k,l,m,n}] .$$

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 .

Tiled convolution offers a compromise between a convolutional layer and a locally connected layer. We make 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.

$$z_{i,j,k} = \sum_{l,m,n} v_{l,j+m,k+n} k_{i,l,m,n,j \% t, k \% t}$$

Note that 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. 9.7). 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.

Convolution is a linear operation and can thus be described as a matrix multiplication (if we first reshape the input array 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 it can be conceptually useful to think of it in this way.

Multiplication by the transpose of the matrix defined by convolution is also a useful 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 an array \mathbf{G} such that $G_{i,j,k} = \frac{\partial}{\partial z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

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

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial v_{i,j,k}} J(\mathbf{V}, \mathbf{K}) = \sum_{l,m|s \times l + m = jn, p|s \times n + p = k} \sum_q k_{q,i,m,p} g_{i,l,n}.$$

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 an array \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.

9.5 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.

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

	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 array 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.

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.

9.6 Efficient Convolution Algorithms

Modern convolutional network applications often involve networks containing more than one million units. Powerful implementations exploiting parallel computation resources are essential. TODO: add reference to efficient implementations section of applications chapter. However, it is also possible to obtain an algorithmic speedup in many cases.

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.

9.7 Deep Learning History

TODO: deep learning history, conv nets were first really successful deep net

TODO neocognitron

TODO: figure out where in the book to put: TODOsectionPooling and Linear-by-Part Non-Linearities: Drednets and Maxout

Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

One of the early ideas found in machine learning and statistical models of the 80's is that of *sharing parameters*¹ across different parts of a model, allowing to *extend the model*, to examples of different forms, e.g. of different lengths. This can be found in hidden Markov models (HMMs) (Rabiner and Juang, 1986), where the same state-to-state transition matrix $P(s_t|s_{t-1})$ is used for different time steps, allowing one to model variable length sequences. It can also be found in the idea of *recurrent neural network* (Rumelhart *et al.*, 1986c) or RNN²: the same weights are used for different instances of the artificial neurons at different time steps, allowing us to apply the network to input sequences of different lengths. 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 the hidden units. Such networks are the ancestors of *convolutional neural networks*, covered in more detail in Section 9. Recurrent nets are covered below in Section 10.2. As shown in Section 10.1 below, the flow graph (notion introduced in Section 6.3) associated with a recurrent network is structured like a chain. Recurrent neural networks have been generalized into *recursive neural networks*, in which the structure can be more general, i.e., and it is typically viewed as a tree. Recursive neural networks are discussed in more detail in Section 10.4.

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 6.3 for a general introduction. In this section we explain the idea of *unfolding* a

¹see Section 7.9 for an introduction to the concept of parameter sharing

²Unfortunately, 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 Network.

recursive or recurrent computation into a flow graph that has a repetitive structure.

For example, consider the classical form of a dynamical system:

$$\mathbf{s}_t = F_\theta(\mathbf{s}_{t-1}) \quad (10.1)$$

where \mathbf{s}_t is called the state of the system. The unfolded flow graph of such a system looks like in Figure 10.1.

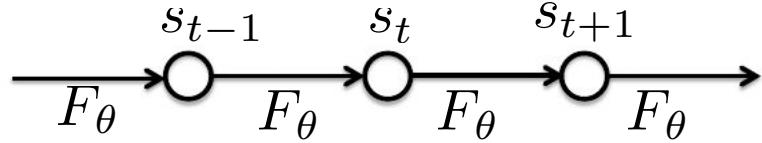


Figure 10.1: Classical dynamical system equation 10.1 illustrated as an unfolded flow-graph. Each node represents the state at some time t and function F_θ maps the state at t to the state at $t + 1$. The same parameters (the same function F_θ) is used for all time steps.

As another example, let us consider a dynamical system driven by an external signal x_t ,

$$\mathbf{s}_t = F_\theta(\mathbf{s}_{t-1}, \mathbf{x}_t) \quad (10.2)$$

illustrated in Figure 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

$$\mathbf{s}_t = G_t(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_2, \mathbf{x}_1) \quad (10.3)$$

which maps the whole past sequence $(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_2, \mathbf{x}_1)$ to the current state. Equation 10.2 is actually part of the definition of a recurrent net. We can think of \mathbf{s}_t is 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 $(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_2, \mathbf{x}_1)$ to a fixed length vector \mathbf{s}_t . Depending on the training criterion, this summary might selectively keeping 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 \mathbf{s}_t to be rich enough to allow one to approximately recover the input sequence, as in auto-encoder frameworks (Chapter 16).

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. 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.

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 Figure 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 Figure 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 $t + 1$.

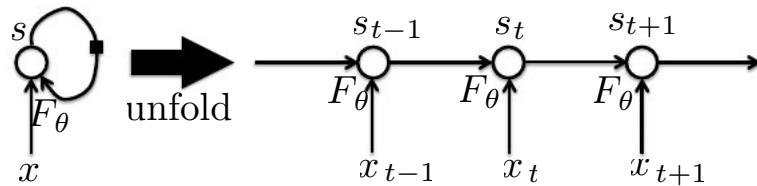


Figure 10.2: Left: input processing part of a recurrent neural network, seen as a circuit. The black square indicates a delay of 1 time steps. Right: the same seen as an unfolded flow graph, where each node is now associated with one particular time instance.

The other important observation to make from Figure 10.2 is that *the same parameters (θ) 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, and automatically obtain their equivalent unfolded graph, which are useful computationally and also help focus on the idea of information flow forward (computing outputs and losses) and backward in time (computing gradients).

Some of the early circuit designs for recurrent neural networks are illustrated in Figures 10.3, 10.5 and 10.4. Figure 10.3 shows the vanilla recurrent network whose equations are laid down below in Eq. 10.4, and which has been shown to be a universal approximation machine for sequences, i.e., able to implement a Turing machine (Siegelmann and Sontag, 1991; Siegelmann, 1995; Hyotyniemi, 1996). On the other hand, the network with *output recurrence* shown in Figure 10.4 has a more limited memory or state, which is its output, i.e., the prediction of the previous target, which 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. *Teacher forcing* is the training process in which the fed back inputs are not the predicted outputs but the targets themselves, as shown in figure TODO. 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, 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 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.

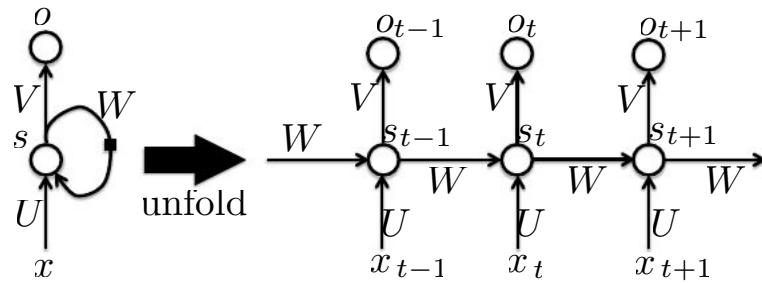


Figure 10.3: Left: vanilla recurrent network with hidden-to-hidden recurrence, seen as a circuit, with weight matrices U , V , W for the three different kinds of connections (input-to-hidden, hidden-to-output, and hidden-to-hidden, respectively). Each circle indicates a whole vector of activations. Right: the same seen as an time-unfolded flow graph, where each node is now associated with one particular time instance.

The vanilla recurrent network of Figure 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):

$$\begin{aligned}\mathbf{a}_t &= \mathbf{b} + \mathbf{W} \mathbf{s}_{t-1} + \mathbf{U} \mathbf{x}_t \\ \mathbf{s}_t &= \tanh(\mathbf{a}_t) \\ \mathbf{o}_t &= \mathbf{c} + \mathbf{V} \mathbf{s}_t \\ \mathbf{p}_t &= \text{softmax}(\mathbf{o}_t)\end{aligned}\tag{10.4}$$

where the parameters are the bias vectors \mathbf{b} and \mathbf{c} along with the weight matrices \mathbf{U} , \mathbf{V} and \mathbf{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 (\mathbf{x}, \mathbf{y}) would then be just the sum of the losses over all the time steps, e.g.,

$$L(\mathbf{x}, \mathbf{y}) = \sum_t L_t = \sum_t -\log p_{y_t}\tag{10.5}$$

where y_t is the category that should be associated with time step t in the output sequence.

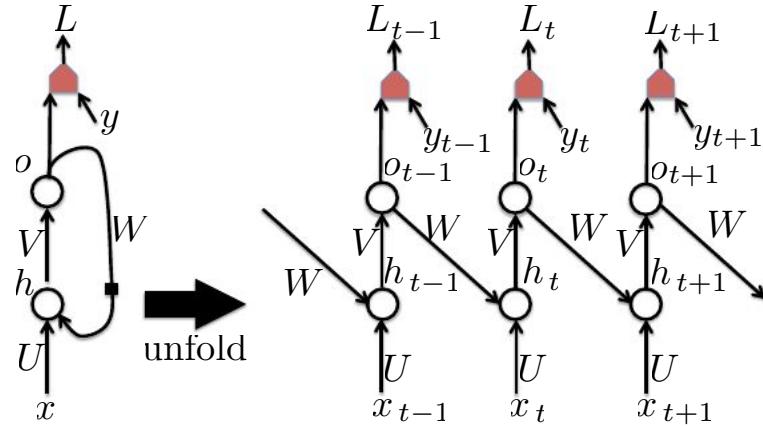


Figure 10.4: Left: RNN whose recurrence is only through the output. Right: computational flow graph unfolded in time. At each time step t , the input is x_t , the hidden layer activations h_t , the output o_t , the target y_t and the loss L_t . Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by Figure 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. The reason why this architecture is less powerful is because the only state information (which carries all the information about the past) is the previous *prediction*. Unless the predicted variable (and hence the target output variable) is very high-dimensional and rich, this will usually miss important information from the past inputs that needs to be carried in the state for *future predictions*.

10.2.1 Computing the Gradient in a Recurrent Neural Network

Using the generalized back-propagation algorithm (for arbitrary flow-graphs) introduced in Section 6.3, one can thus 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 gradient-based techniques to train an RNN, 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 be the sequence of \mathbf{x}_t 's, \mathbf{s}_t 's, \mathbf{o}_t 's, L_t 's, and the parameters \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} , \mathbf{c} . For each node a we need to compute the gradient $\nabla_a L$ 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 on the outputs i at time step t , for all i, t :

$$\frac{\partial L}{\partial o_{ti}} = \frac{\partial L}{\partial L_t} \frac{\partial L_t}{\partial o_{ti}} = p_{t,i} - \mathbf{1}_{i,y_t}$$

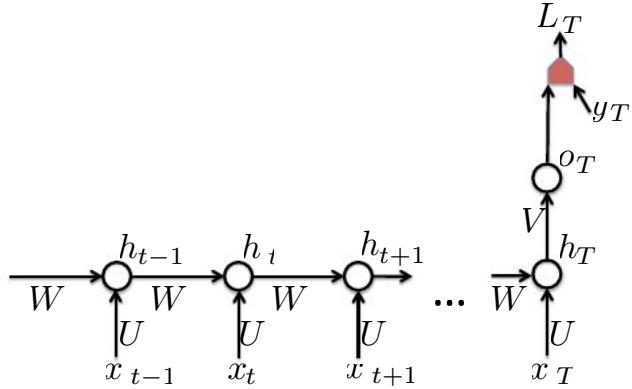


Figure 10.5: Time-Unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (like in the figure) or the gradient on the output \mathbf{o}_T can be obtained by back-propagating from further downstream modules.

and work our way backwards, starting from the end of the sequence, say T , at which point \mathbf{s}_T only has \mathbf{o}_T as descendent:

$$\nabla_{\mathbf{s}_T} L = \nabla_{\mathbf{o}_T} L \frac{\partial \mathbf{o}_T}{\partial \mathbf{s}_T} = \nabla_{\mathbf{o}_T} L \mathbf{V}.$$

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 \mathbf{s}_t (for $t < T$) has as descendent both \mathbf{q} and \mathbf{s}_{t+1} :

$$\nabla_{\mathbf{s}_t} L = \nabla_{\mathbf{s}_{t+1}} L \frac{\partial \mathbf{s}_{t+1}}{\partial \mathbf{s}_t} + \nabla_{\mathbf{o}_t} L \frac{\partial \mathbf{o}_t}{\partial \mathbf{s}_t} = \nabla_{\mathbf{s}_{t+1}} L \text{diag}(1 - \mathbf{s}_{t+1}^2) \mathbf{W} + \nabla_{\mathbf{o}_t} L \mathbf{V}$$

where $\text{diag}(1 - \mathbf{s}_{t+1}^2)$ indicates the diagonal matrix containing the elements $1 - \mathbf{s}_{t+1,i}^2$, 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 descendants at all the time

steps:

$$\begin{aligned}
\nabla_{\mathbf{c}} L &= \sum_t \nabla_{\boldsymbol{\alpha}} L \frac{\partial \mathbf{o}_t}{\partial \mathbf{c}} = \sum_t \nabla_{\boldsymbol{\alpha}} L \\
\nabla_{\mathbf{b}} L &= \sum_t \nabla_{\mathbf{s}_t} L \frac{\partial \mathbf{s}_t}{\partial \mathbf{b}} = \sum_t \nabla_{\mathbf{s}_t} L \text{diag}(1 - \mathbf{s}_t^2) \\
\nabla_{\mathbf{V}} L &= \sum_t \nabla_{\boldsymbol{\alpha}} L \frac{\partial \mathbf{o}_t}{\partial \mathbf{V}} = \sum_t \nabla_{\boldsymbol{\alpha}} L \mathbf{s}_t^\top \\
\nabla_{\mathbf{W}} L &= \sum_t \nabla_{\mathbf{s}_t} \frac{\partial \mathbf{s}_t}{\partial \mathbf{W}} = \sum_t \nabla_{\mathbf{s}_t} L \text{diag}(1 - \mathbf{s}_t^2) \mathbf{s}_{t-1}^\top
\end{aligned}$$

Note in the above (and elsewhere) that whereas $\nabla_{\mathbf{s}_t} L$ refers to the full influence of \mathbf{s}_t through all paths from \mathbf{s}_t to L , $\frac{\partial \mathbf{s}_t}{\partial \mathbf{W}}$ or $\frac{\partial \mathbf{s}_t}{\partial \mathbf{b}}$ 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 Generative Directed Acyclic Models

Up to here, we have not clearly stated what the losses L_t associated with the outputs \mathbf{o}_t of a recurrent net should be. It is because there are many possible ways in which RNNs can be used. Here we consider the case where the RNN models a probability distribution over a sequence of observations.

As introduced in Section 14.2.1, directed graphical models represent a probability distribution over a sequence of T random variables $\mathbf{x} = (x_1, x_2, \dots, x_T)$ by parametrizing their joint distribution as

$$P(\mathbf{x}) = P(\mathbf{x}_1, \dots, \mathbf{x}_T) = \prod_{t=1}^T P(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_1) \quad (10.6)$$

where the right-hand side is empty for $t = 1$, of course. Hence the negative log-likelihood of \mathbf{x} according to such a model is

$$L = \sum_t L_t$$

where

$$L_t = -\log P(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_1).$$

In general directed graphical models, \mathbf{x}_t can be predicted using only a subset of its predecessors $(\mathbf{x}_1, \dots, \mathbf{x}_{t-1})$. However, for RNNs, the graphical model is generally fully connected, not removing any dependency a priori. This can be achieved efficiently through the recurrent parametrization, such as in Eq. 10.2, since \mathbf{s}_t summarizes the whole previous sequence (Eq. 10.3).

Hence, instead of cutting statistical complexity by removing arcs in the directed graphical model for $(\mathbf{x}_1, \dots, \mathbf{x}_T)$, the core idea of recurrent networks is that we introduce a state variable which decouples all the past and future observations, but we make that state variable a generally deterministic function of the past, through the recurrence, Eq. 10.2. Consequently, the number of parameters required to parametrize $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_1)$ does not grow exponentially with t (as it would if we parametrized that probability by a straight probability table, when the data is discrete) but remains constant with t . It only grows with the dimension of the state \mathbf{s}_t . The price to be paid for that great advantage is that *optimizing* the parameters may be more difficult, as discussed below in Section 10.6. The decomposition of the likelihood thus becomes:

$$P(\mathbf{x}) = \prod_{t=1}^T P(\mathbf{x}_t | G_t(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_1))$$

where

$$\mathbf{s}_t = G_t(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_2, \mathbf{x}_1) = F_\theta(\mathbf{s}_{t-1}, \mathbf{x}_t).$$

Note that if the self-recurrence function F_θ is *learned*, it can discard some of the information in some of the past values \mathbf{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.

The above decomposition of the joint probability of a sequence of variables into ordered conditionals precisely corresponds to what an RNN can compute: the *target* 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, and the *output* is interpreted as parametrizing the probability distribution of the target given the state. This is illustrated in Figure 10.6.

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 \mathbf{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 $P(\mathbf{x}_1, \dots, \mathbf{x}_T)$ into $P(T)$ and $P(\mathbf{x}_1, \dots, \mathbf{x}_T | T)$. In general, one must therefore keep in mind that in order to fully generate a sequence we must not only generate the 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.

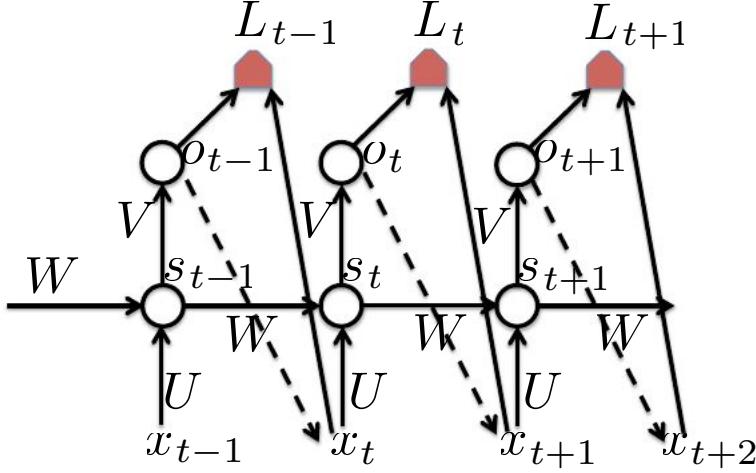


Figure 10.6: A generative recurrent neural network modeling $P(\mathbf{x}_1, \dots, \mathbf{x}_T)$, able to generate sequences from this distribution. Each element x_t of the observed sequence serves both as input (for the current time step) and as target (for the previous time step). The output \mathbf{o}_t encodes the parameters of a conditional distribution $P(\mathbf{x}_{t+1}|\mathbf{x}_1, \dots, \mathbf{x}_t) = P(\mathbf{x}_{t+1}|\mathbf{o}_t)$ for \mathbf{x}_{t+1} , given the past sequence $\mathbf{x}_1, \dots, \mathbf{x}_t$. The loss L_t is the negative log-likelihood associated with the output prediction (or more generally, distribution parameters) \mathbf{o}_t when the actual observed target is \mathbf{x}_{t+1} . In training mode, one measures and minimizes the sum of the losses over observed sequence(s) \mathbf{x} . In generative mode, \mathbf{x}_t is sampled from the conditional distribution $P(\mathbf{x}_{t+1}|\mathbf{x}_1, \dots, \mathbf{x}_t) = P(\mathbf{x}_{t+1}|\mathbf{o}_t)$ (dashed arrows) and then that generated sample \mathbf{x}_{t+1} is fed back as input for computing the next state s_{t+1} , the next output \mathbf{o}_{t+1} , and generating the next sample \mathbf{x}_{t+2} , etc.

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 \mathbf{y}_t equal to the next input \mathbf{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., \mathbf{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 6.2.2.

10.2.3 RNNs to Represent Conditional Probability Distributions

In general, as discussed in Section 6.2.2 (see especially the end of that section, in Sub-section 6.2.2), when we can represent a parametric probability distribution $P(\mathbf{y}|\omega)$, we can make it conditional by making ω a function of the desired conditioning variable:

$$P(\mathbf{y}|\omega = f(\mathbf{x})).$$

In the case of an RNN, this can be achieved in different ways, and we review here the most common and obvious choices.

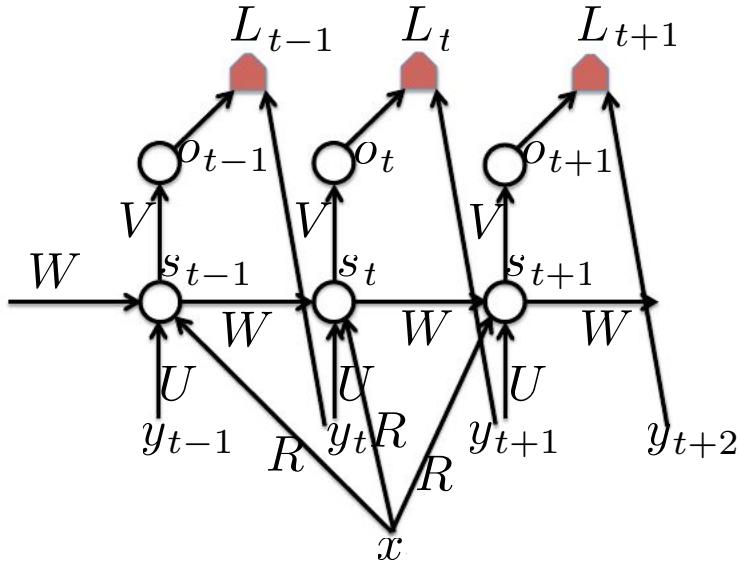


Figure 10.7: A conditional generative recurrent neural network maps a fixed-length vector \mathbf{x} into a distribution over sequences \mathbf{Y} . Each element \mathbf{y}_t of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step). The generative semantics are the same as in the unconditional case (Fig. 10.6). The only difference is that the state is now conditioned on the input \mathbf{x} , and the same parameters (weight matrix \mathbf{R} in the figure) is used at every time step to parametrize that dependency. Although this was not discussed in Fig. 10.6, in both figures one should note that the length of the sequence must also be generated (unless known in advance). This could be done by a special binary output unit that encodes the fact that the next output is the last.

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 s_0 , or
3. both.

In general, one may need to add extra parameters (and parametrization) to map $\mathbf{x} = \mathbf{x}$ into the “extra bias” going either into only s_0 , into the other s_t ($t > 0$), or into both. The first (and most commonly used) approach is illustrated in Figure 10.7.

As an example, we could imagine that \mathbf{x} is encoding the identity of a phoneme and the identity of a speaker, and that \mathbf{y} represents 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

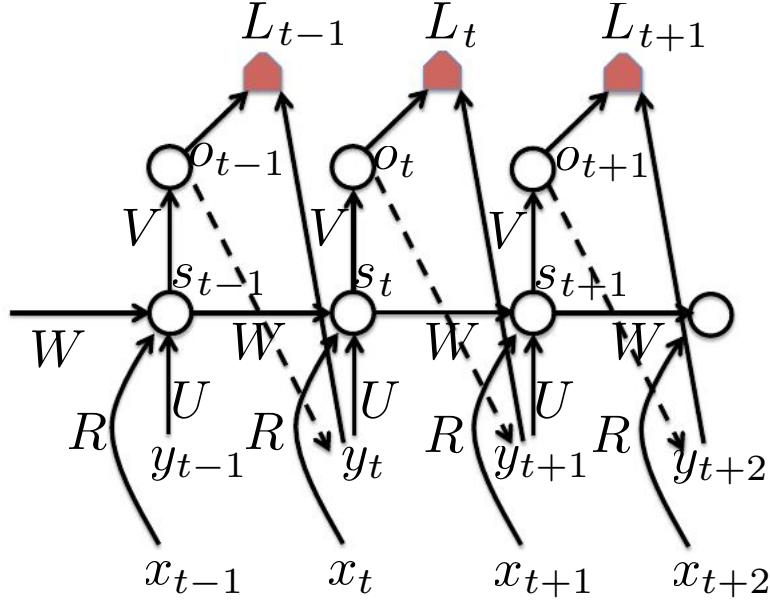


Figure 10.8: A conditional generative recurrent neural network mapping a variable-length sequence \mathbf{x} into a distribution over sequences \mathbf{y} of the same length. This architecture assumes that the \mathbf{y}_t 's are causally related to the \mathbf{x}_t 's, i.e., that we want to predict the \mathbf{y}_t 's only using the past \mathbf{x}_t 's. Note how the prediction of \mathbf{y}_{t+1} is based on both the past \mathbf{x} 's and the past \mathbf{y} 's. The dashed arrows indicate that \mathbf{y}_t can be generated by sampling from the output distribution \mathbf{o}_{t-1} . When \mathbf{y}_t is clamped (known), it is used as a target in the loss L_{t-1} which measures the log-probability that \mathbf{y}_t would be sampled from the distribution \mathbf{o}_{t-1} .

is given, i.e., $P(\mathbf{y}_t | \mathbf{y}_{t-1}, \dots, \mathbf{y}_1, \mathbf{x}) = P(\mathbf{y}_t | \mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$. 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 \mathbf{o}_t as parameters for a conditional distribution for \mathbf{y}_t , given $\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_1$.

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. 10.8.

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, $\mathbf{x}_1, \dots, \mathbf{x}_t$. 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

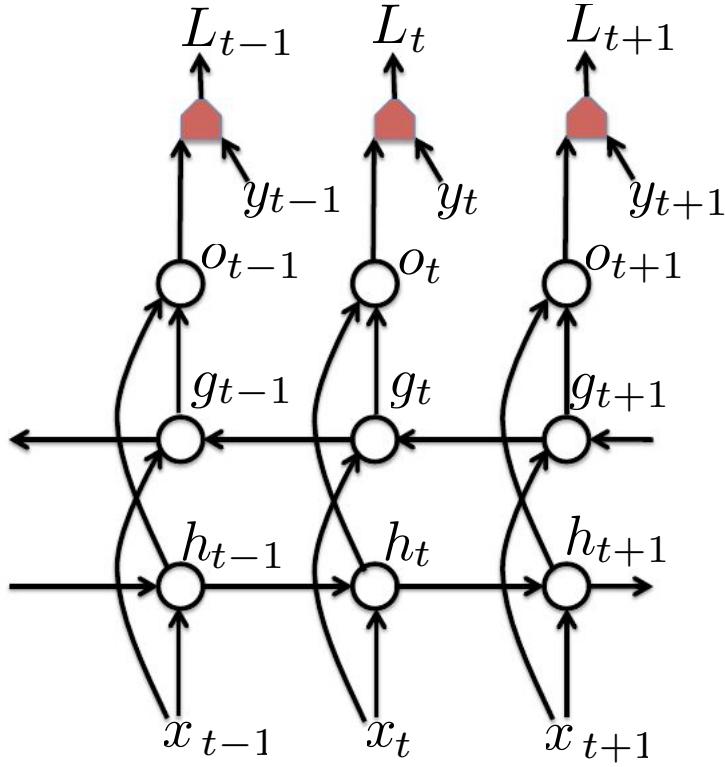


Figure 10.9: Computational of a typical bidirectional recurrent neural network, meant to learn to map input sequences \mathbf{x} to target sequences \mathbf{y} , with loss L_t at each step t . The \mathbf{h} recurrence propagates information forward in time (towards the right) while the \mathbf{g} recurrence propagates information backward in time (towards the left). Thus at each point t , the output units \mathbf{o}_t can benefit from a relevant summary of the past in its \mathbf{h}_t input and from a relevant summary of the future in its \mathbf{g}_t input.

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.

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 (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. Figure 10.9 illustrates the typical bidirectional RNN, with h_t standing for the state of the forward-going RNN and g_t standing for the state of the backward-going RNN. This allows the units \mathbf{o}_t to compute a rep-

resentation 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.

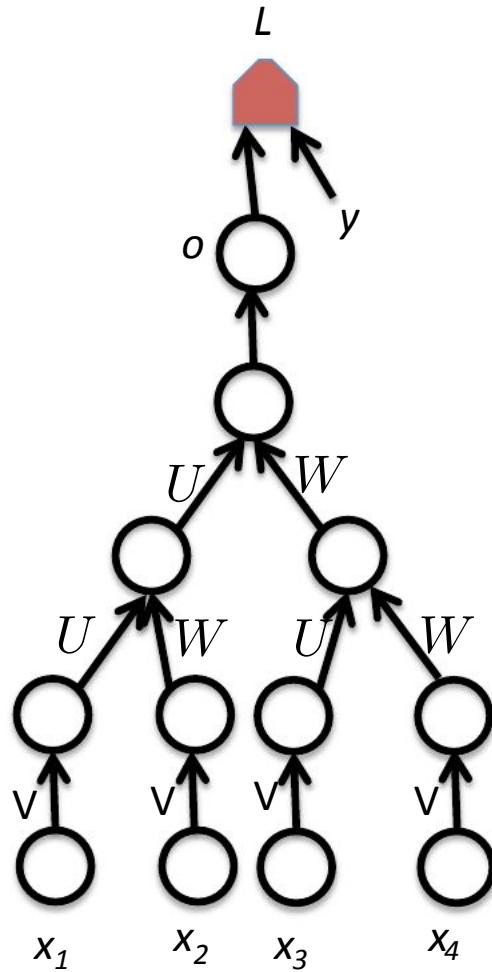


Figure 10.10: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. In the figure, a variable-size sequence $\mathbf{x}_1, \mathbf{x}_2, \dots$ can be mapped to a fixed-size representation (the output o), with a fixed number of parameters (e.g. the weight matrices U , V , W). The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.

10.4 Recursive Neural Networks

Recursive net represent yet another generalization of recurrent networks, with a different kind of computational graph, which this times looks like a tree. The typical computational graph for a recursive network is illustrated in Figure 10.10. 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 net over recurrent nets is that for a sequence of the same length N , depth can be drastically reduced from N to $O(\log N)$. 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).

10.5 Auto-Regressive Networks

One of the basic ideas behind recurrent networks is that of directed graphical models with a twist: we decompose a probability distribution as a product of conditionals *without explicitly cutting any arc in the graphical model*, but instead reducing complexity 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 7.9 for an introduction to the concept). Instead of reducing $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$ to something like $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-k})$ (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. When the above conditional probability distribution is in some sense 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, then this form of parameter sharing makes a lot of sense, and for recurrent nets it allows to use the same model for sequences of different lengths.

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 $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$ but we drop the form of parameter sharing that makes these conditionals all share the same parametrization. This makes sense when the variables are *not* elements of a translation-invariant sequence, but instead form an arbitrary tuple without any particular ordering that would correspond to a translation-invariant form of relationship between variables at position k and variables at position k' . In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described in Section 10.5.3 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 10.5.2, using a deep learning concept of *re-use of features*, we can *share* features that have been computed for predicting \mathbf{x}_{t-k} with the sub-network that predicts \mathbf{x}_t .

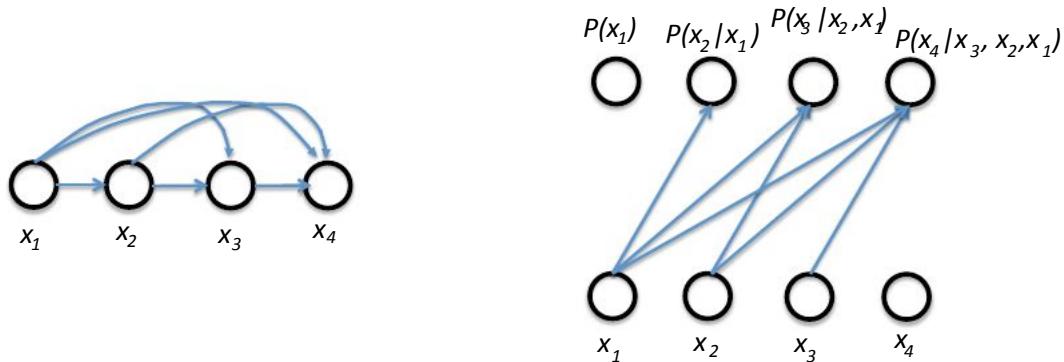


Figure 10.11: An auto-regressive network predicts the i -th variable from the $i-1$ previous ones. Left: corresponding graphical model (which is the same as that of a recurrent network). Right: corresponding computational graph, in the case of the logistic auto-regressive network, where each prediction has the form of a logistic regression, i.e., with i free parameters (for the $i-1$ weights associated with $i-1$ inputs, and an offset parameter).

10.5.1 Logistic Auto-Regressive Networks

Let us first consider the simplest auto-regressive network, without hidden units, and hence no sharing at all. Each $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$ is parametrized as a linear model, e.g., a logistic regression. 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 Figure 10.11, 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.

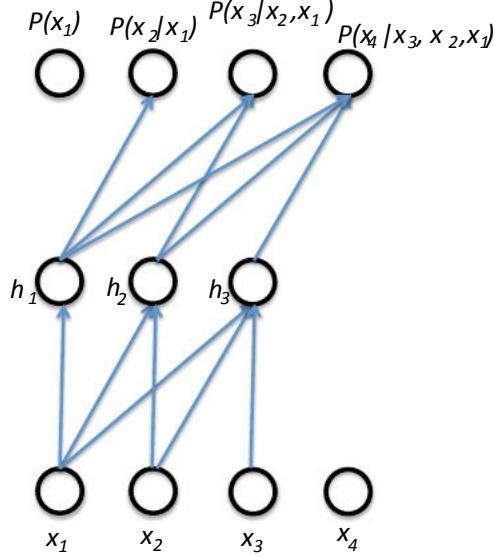


Figure 10.12: A neural auto-regressive network predicts the i -th variable \mathbf{x}_i from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $\mathbf{x}_1, \dots, \mathbf{x}_i$ can be re-used in predicting all of the subsequent variables $\mathbf{x}_{i+1}, \mathbf{x}_{i+2}, \dots$

10.5.2 Neural Auto-Regressive Networks

Neural Auto-Regressive Networks have the same left-to-right graphical model as logistic auto-regressive networks (Figure 10.11, 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 Figure 10.11 (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 $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$ 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 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 \mathbf{x}_t , a *left-to-right* connectivity illustrated in Figure 10.12 allows to merge all the neural

networks into one. Equivalently, it means that the hidden layer features computed for predicting \mathbf{x}_t can be re-used for predicting \mathbf{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 $\mathbf{x}_1, \dots, \mathbf{x}_t$. In fact the parameters used to compute these hidden units are jointly optimized to help the prediction of all the variables $\mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}, \dots$. This is an instance of the *re-use principle* that makes *multi-task learning* and *transfer learning* successful with neural networks and deep learning in general (See Sections 7.12 and 17.2).

Each $P(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$ can represent a conditional distribution by having outputs of the neural network predict *parameters* of the conditional distribution of \mathbf{x}_t , as discussed in Section 6.2.2. Although the original neural auto-regressive networks were initially evaluated in the context of purely discrete multivariate data (e.g., with a sigmoid - Bernoulli case - or softmax output - Multinoulli case) it is straightforward to extend such models to continuous variables or joint distributions involving both discrete and continuous variables, as for example with RNADE introduced below (Uria *et al.*, 2013).

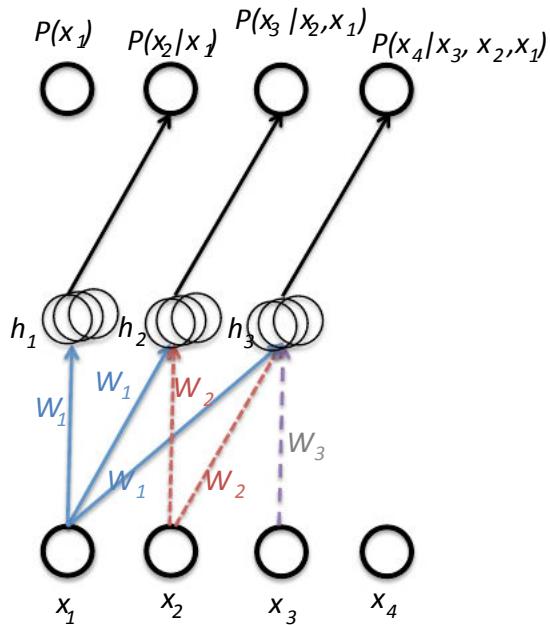


Figure 10.13: NADE (Neural Auto-regressive Density Estimator) is a neural auto-regressive network, i.e., the hidden units are organized in groups h_j so that only the inputs $\mathbf{x}_1, \dots, \mathbf{x}_i$ participate in computing h_i and predicting $P(\mathbf{x}_j | \mathbf{x}_{j-1}, \dots, \mathbf{x}_1)$, for $j > i$. The particularity of NADE is the use of a particular weight sharing pattern: the same $W'_{jki} = W_{ki}$ is shared (same color and line pattern in the figure) for all the weights outgoing from \mathbf{x}_i to the k -th unit of any group $j \geq i$. The vector (W_{1i}, W_{2i}, \dots) is denoted W_i here.

10.5.3 NADE

A very successful recent form of neural auto-regressive network was proposed by [Larochelle and Murray \(2011\)](#). The architecture is basically the same as for the original neural auto-regressive network of [Bengio and Bengio \(2000b\)](#) *except for the introduction of a weight-sharing scheme*: as illustrated in Figure 10.13. The parameters of the hidden units of different groups j are shared, i.e., the weights W'_{jki} from the i -th input \mathbf{x}_i to the k -th element of the j -th group of hidden unit h_{jk} ($j \geq i$) are shared:

$$W'_{jki} = W_{ki}$$

with (W_{1i}, W_{2i}, \dots) denoted W_i in Figure 10.13.

This particular sharing pattern is motivated in [Larochelle and Murray \(2011\)](#) by the computations performed in the mean-field inference³ of an RBM, when only the first i inputs are given and one tries to infer the subsequent ones. 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 $P(\mathbf{x}_j | \mathbf{x}_{j-1}, \dots, \mathbf{x}_1)$) by continuous ones, following general practice to predict continuous random variables with neural networks (see Section 6.2.2) using the log-likelihood framework. A fairly generic way of parametrizing a continuous density is as a Gaussian mixture, and this avenue has been successfully evaluated for the neural auto-regressive architecture with RNADE ([Uria et al., 2013](#)). One interesting point to note is that stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means and the conditional variances (especially 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](#)). 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 - conditioning side of the 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 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 are possible, the joint probability of some set of variables can be computed in many ways ($N!$ for N

³Here, unlike in Section 14.5, the inference is over some of the input variables that are missing, given the observed ones.

variables), and this can be exploited to obtain a more robust probability estimation and better log-likelihood, by simply averaging the log-probabilities predicted by different randomly chosen orders. 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 Figures 10.13 and 10.12), whereas it is $O(N^2H)$ in Bengio and Bengio (2000b). However, for the other hidden layers, the computation is $O(N^2H^2)$ if every “previous” group at layer k participates in predicting the “next” group at layer $k+1$, assuming N groups of H hidden units at each layer. Making the i -th group at layer $k+1$ only depend on the i -th group, as in Murray and Larochelle (2014) at layer k reduces it to $O(NH^2)$, which is still H times worse than the regular NADE.

10.6 Facing the Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks was introduced in Section 8.2.5. 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.

10.6.1 Echo State Networks: Choosing Weights to Make Dynamics 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 s_t (hidden units vector) at time t (Eq. 10.2) captures and summarizes information from the previous inputs $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t$. Since learning the recurrent and input weights is difficult, one option that has been proposed (Jaeger and Haas, 2004; Jaeger, 2007b; Maass *et al.*, 2002) 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 ouput 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 allow to map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state 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 on the edge of chaos*. (TODO make that more precise). As alluded to in 8.2.5, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $J_t = \frac{\partial s_t}{\partial s_{t-1}}$, and in particular the *spectral radius* of J_t , i.e., its largest eigenvalue. If it is greater than 1, the dynamics can diverge, meaning that small differences in the state value at t can yield a very large difference at T later in the sequence. To see this, consider the simpler case where the Jacobian matrix J does not change with t . If a change Δs in the state at t is aligned with an eigenvector v of J with eigenvalue $\lambda > 1$, then the small change Δs becomes $\lambda \Delta s$ after one time step, and $\lambda^N \Delta s$ after N time steps. If $\lambda > 1$ this makes the change exponentially large. With a non-linear map, the Jacobian keeps changing and the dynamics is more complicated but what remains is that a small initial variation can turn into a large variation after a number of steps. In general, the recurrent dynamics are bounded (for example, if the hidden units use a bounded non-linearity such as the hyperbolic tangent) so that the change after N steps must also be bounded. Instead when the largest eigenvalue $\lambda < 1$, we say that the map from t to $t+1$ is *contractive*: a small change gets *contracted*, becoming smaller after each time step. This necessarily makes the network *forgetting* information about the long-term past, but it also makes its dynamics stable and easier to use.

What has been proposed to set the weights of reservoir computing machines is to make the Jacobians *slightly contractive*. This is achieved by making the spectral radius of the weight matrix large but slightly less than 1. However, in practice, good results are often found with a spectral radius of the recurrent weight matrix being slightly larger than 1, e.g., 1.2 (Sutskever, 2012; Sutskever *et al.*, 2013). Keep in mind that with hyperboling 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, information can be kept in a stable way, and back-propagated gradients neither vanish nor explode. The next two sections show methods to make some paths in the unfolded graph correspond to “multiplying by 1” at each step, i.e., keeping information for a very long time.

10.6.2 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. 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).

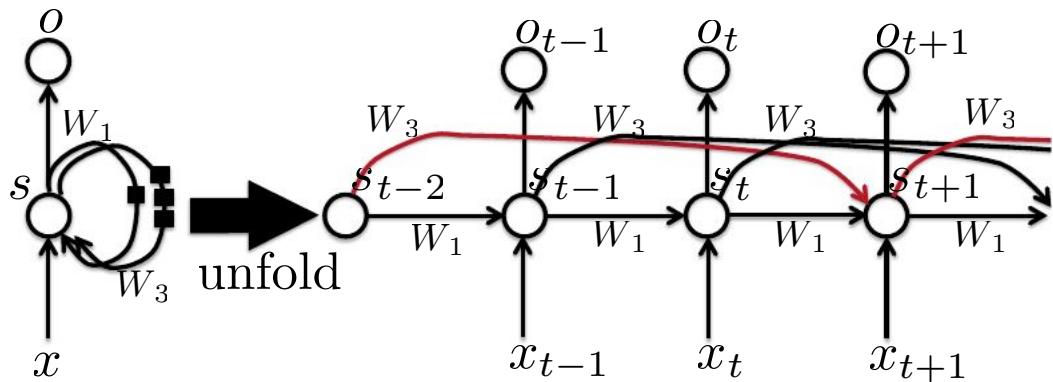


Figure 10.14: A recurrent neural networks with *delays*, in which some of the connections reach back in time to more than one time step. Left: connectivity of the recurrent net, with square boxes indicating the number of time delays associated with a connection. Right: unfolded recurrent network. In the figure there are regular recurrent connections with a delay of 1 time step (W_1) and recurrent connections with a delay of 3 time steps (W_3). 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 8.2.5, gradients will vanish 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 λ 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 Figure 10.14.

10.6.3 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.

The idea of leaky units with a self-connection actually arises naturally when considering a *continuous-time* recurrent neural network such as

$$\dot{s}_i \tau_i = -s_i + \sigma(b_i + Ws + Ux)$$

where σ 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

$$\dot{s}_i \tau_i = -s_i + (b_i + W\sigma(s) + Ux)$$

where the state vector 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 τ), one gets

$$\begin{aligned} s_{t+1,i} - s_{t,i} &= -\frac{s_{t,i}}{\tau_i} + \frac{1}{\tau_i} \sigma(b_i + Ws_t + Ux_t) \\ s_{t+1,i} &= \left(1 - \frac{1}{\tau_i}\right)s_{t,i} + \frac{1}{\tau_i} \sigma(b_i + Ws_t + Ux_t). \end{aligned} \quad (10.7)$$

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 τ_i is large, the state changes very slowly, integrating the past values associated with the input sequence.

By associating different time scales τ_i with different units, one obtains different paths corresponding to different forgetting rates. Those time constants can be fixed manually (e.g., by sampling from a distribution of time scales) or can be learned as free parameters, and 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 τ thus corresponds to a *self-weight* of $(1 - \frac{1}{\tau})$, but *without any non-linearity involved in the self-recurrence*.

Consider the extreme case where $\tau \rightarrow \infty$: because the leaky unit just *averages* contributions from the past, the contribution of each time step is equivalent and there is no associated vanishing or exploding effect. An alternative is to avoid the weight of $\frac{1}{\tau_i}$ in front of $\sigma(b_i + W s_t + U x_t)$, thus making the state *sum* all the past values when τ_i is large, instead of averaging them.

10.6.4 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.

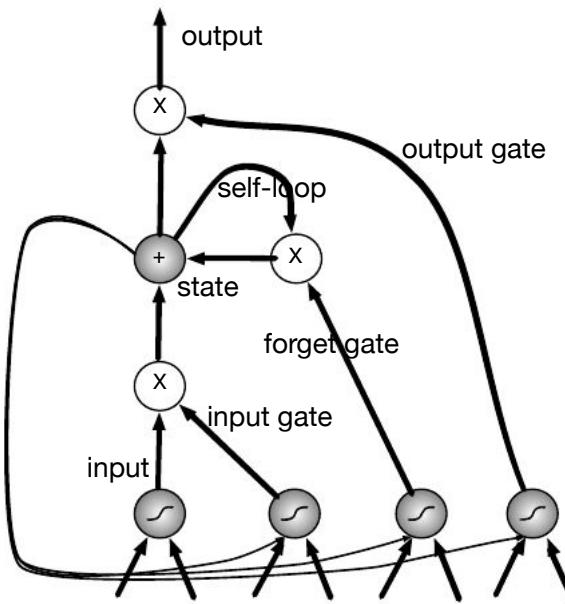


Figure 10.15: Block diagram of the LSTM recurrent network “cell”. Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit, and its value can be accumulated into the state if the 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.

It is worthwhile considering the role played by leaky units: they allow 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.*, 2014) 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 block diagram is illustrated in Figure 10.15. The corresponding forward (state update equations) are follows, in the case of the vanilla recurrent network architecture. Deeper architectures have been successfully used in Graves *et al.* (2013); Pascanu *et al.* (2014b). Instead of a unit that simply applies a squashing function on the affine transformation of inputs and recurrent units, LSTM networks have “LSTM cells”. Each cell has the same inputs and outputs as a vanilla 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 that has a linear self-loop similar to the leaky units described in the previous section, but where the self-loop weight (or the associated time constant) is controlled by a *forget gate* unit $h_{t,i}^f$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit:

$$h_{t,i}^f = \text{sigmoid}(b_i^f + \sum_j U_{ij}^f x_{t,j} + \sum_j W_{ij}^f h_{t,j}). \quad (10.8)$$

where \mathbf{x}_t is the current input vector and h_t is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\mathbf{b}^f, \mathbf{U}^f, \mathbf{W}^f$ 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 $h_{t,i}^f$:

$$s_{t+1,i} = h_{t,i}^f s_{t,i} + h_{t,i}^e \sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h_{t,j}). \quad (10.9)$$

\mathbf{b} , \mathbf{U} and \mathbf{W} respectively the biases, input weights and recurrent weights into the LSTM cell, and the *external input gate* unit $h_{t,i}^e$ 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:

$$h_{t,i}^e = \text{sigmoid}(b_i^e + \sum_j U_{ij}^e x_{t,j} + \sum_j W_{ij}^e h_{t,j}). \quad (10.10)$$

The output $h_{t+1,i}$ of the LSTM cell can also be shut off, via the *output gate* $h_{t,i}^o$, which also uses a sigmoid unit for gating:

$$\begin{aligned} h_{t+1,i} &= \tanh(s_{t+1,i})h_{t,i}^o \\ h_{t,i}^o &= \text{sigmoid}(b_i^o + \sum_j U_{ij}^o x_{t,j} + \sum_j W_{ij}^o h_{t,j}) \end{aligned} \quad (10.11)$$

which has parameters \mathbf{b}^o , \mathbf{U}^o , \mathbf{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 Figure 10.15. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than vanilla 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.*, 2014).

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, which was successfully used in reaching the MOSES state-of-the-art for English-to-French machine translation (Cho *et al.*, 2014). 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:

$$h_{t+1,i} = h_{t,i}^u h_{t,i} + (1 - h_{t,i}^u) \sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h_{t,j}). \quad (10.12)$$

where \mathbf{g}^u stands for “update” gate and \mathbf{g}^r for “reset” gate. Their value is defined as usual:

$$h_{t,i}^u = \text{sigmoid}(b_i^u + \sum_j U_{ij}^u x_{t,j} + \sum_j W_{ij}^u h_{t,j}) \quad (10.13)$$

and

$$h_{t,i}^r = \text{sigmoid}(b_i^r + \sum_j U_{ij}^r x_{t,j} + \sum_j W_{ij}^r h_{t,j}). \quad (10.14)$$

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.

In addition, as discussed in the next section, different ways of making such RNNs “deeper” are possible.

10.6.5 Deep RNNs

Figure 10.16 illustrate a number of architectural variations for RNNs which introduce additional depth beyond what the vanilla architecture provides. In general, the input-to-hidden, hidden-to-hidden, and hidden-to-output mappings can be made more powerful than what the usual single-layer affine + nonlinearity transformation provides. This idea was introduced in Pascanu *et al.* (2014b). The more established way of adding depth is simply to stack different “layers” of RNNs on top of each other (Schmidhuber, 1992; El Hihi and Bengio, 1996; Jaeger, 2007a; Graves, 2013) (bottom right in the Figure), possibly with a different time scale at different levels in the hierarchy (El Hihi and Bengio, 1996; Koutnik *et al.*, 2014).

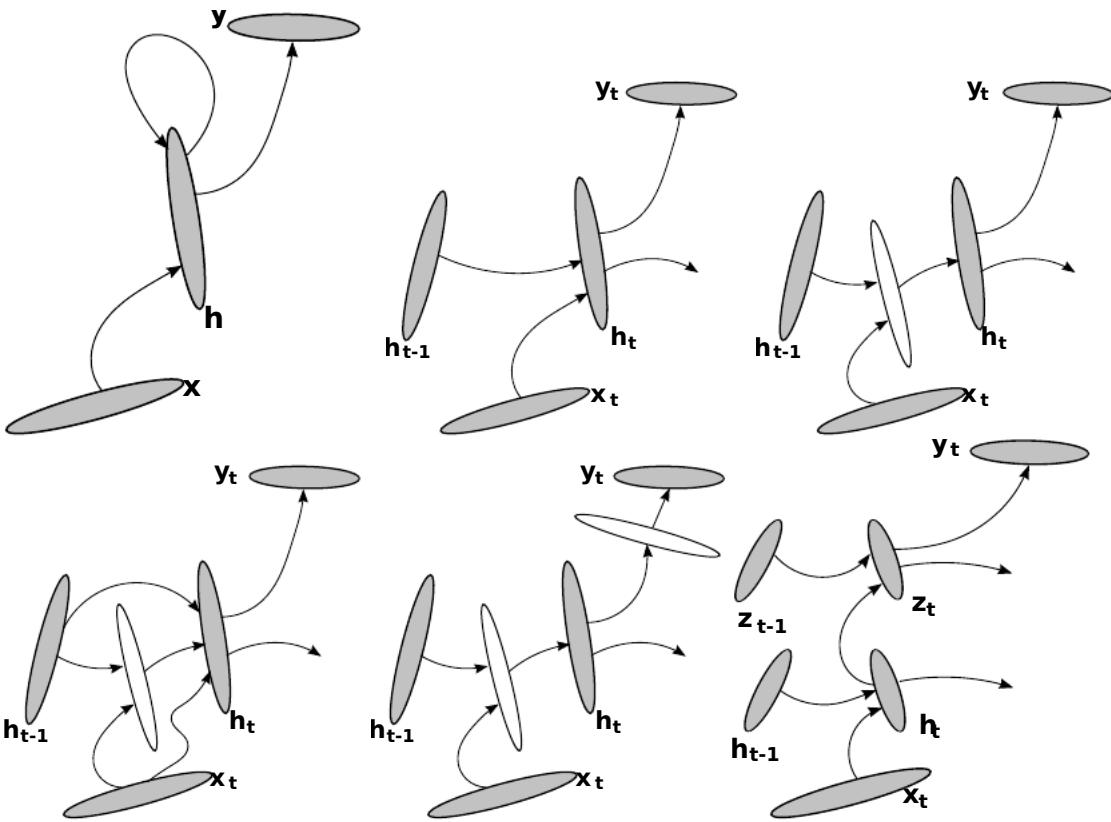


Figure 10.16: Examples of deeper RNN architecture variants. Top left: vanilla RNN with input sequence and an output sequence. Top middle: one step of the corresponding unfolded flow graph. Top right: with a deep hidden-to-hidden transformation, instead of the usual single-layer transformation (e.g., replacing the affine + softmax layer by a full MLP with a single hidden layer, taking both the previous state and current input as inputs). Bottom left: same but with skip connections that allow gradients to flow more easily backwards in time in spite of the extra non-linearity due to the intermediate hidden layer. Bottom middle: similarly, depth can also be added in the hidden-to-output transformation. Bottom right: a hierarchy of RNNs, which can be stacked on top of each other in various ways.

10.6.6 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 8.2.5. The jist 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 $\frac{\partial \mathbf{s}_T}{\partial \mathbf{s}_t}$) tend to either vanish or explode as $T - t$ increases, because it is equal to the product of the state transition Jacobian matrices $\frac{\partial \mathbf{s}_{t+1}}{\partial \mathbf{s}_t}$)

If it explodes, the parameter gradient $\nabla_{\theta} L$ also explodes, yielding gradient-based

parameter updates that are poor. A simple solution to this problem is discussed in the next section (Sec. 10.6.7). 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).

If the eigenvalues of the state transition Jacobian are less than 1, then derivatives associated with long-term effects tend to vanish as $T - t$ increases, making them exponentially smaller in magnitude (as components of the total gradient) than derivatives associated with short-term effects. This therefore makes it difficult (but not impossible) to learn long-term dependencies.

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 rather than “stochastic” updates (where only one 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 8.4), and the clipping trick described below.

10.6.7 Clipping Gradients

As discussed in Section 8.2.4, 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 Figures 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, wasting a lot of the work that had been done to reach the current solution. This is because gradient descent is hinged on the assumption of *small enough steps*, and this assumption can easily be violated when the same learning rate is used for both the flatter parts and the steeper parts of the landscape.

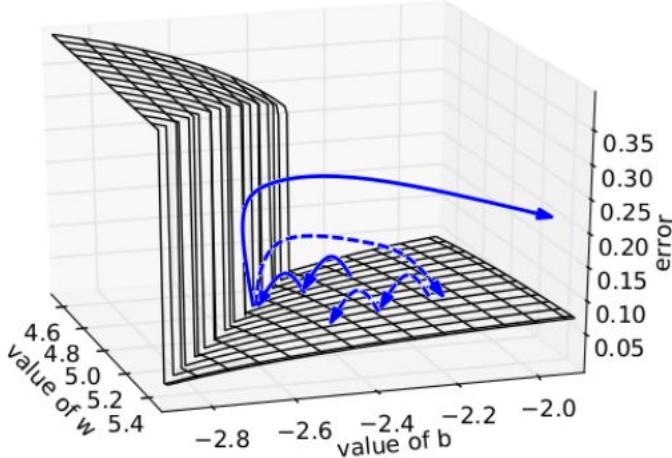


Figure 10.17: Example of the effect of gradient clipping in a recurrent network with two parameters w and b . Vertical axis is the objective function to minimize. Note the cliff where the gradient explodes and from where gradient descent can get pushed very far. Clipping the gradient when its norm is 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 gradient element-wise (Mikolov, 2012). Another is to *clip the norm of the gradient* (Pascanu *et al.*, 2013a). The latter has the advantage that it guarantees that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well.

10.6.8 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 LSTM and other self-loops and gating mechanisms, described above in Section 10.6.4. Another idea is to regularize or constrain the parameters so as to encourage “information flow”. In particular, we would like the

gradient vector $\nabla_{\mathbf{s}_t} L$ being back-propagated to maintain its magnitude (even if there is only a loss at the end of the sequence), i.e., we want

$$\nabla_{\mathbf{s}_t} L \frac{\partial \mathbf{s}_t}{\partial \mathbf{s}_{t-1}}$$

to be as large as

$$\nabla_{\mathbf{s}_t} L.$$

With this objective, [Pascanu et al. \(2013a\)](#) propose the following regularizer:

$$\Omega = \sum_t \left(\frac{\left| \left| \nabla_{\mathbf{s}_t} L \frac{\partial \mathbf{s}_t}{\partial \mathbf{s}_{t-1}} \right| \right|}{\left| \left| \nabla_{\mathbf{s}_t} L \right| \right|} - 1 \right)^2. \quad (10.15)$$

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 $\nabla_{\mathbf{s}_t} L$ 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.

10.6.9 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 Figure 10.18.

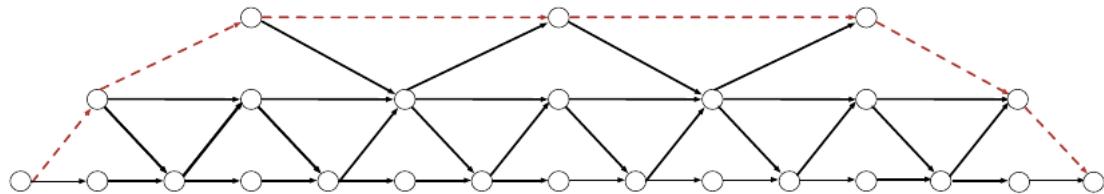


Figure 10.18: Example of a multi-scale recurrent net architecture (unfolded in time), with higher levels operating at a slower time scale. Information can flow unhampered (either forward or backward in time) over longer durations at the higher levels, thus creating long-paths (such as the red dotted 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 Figure 10.18. 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.

10.7 Handling Temporal Dependencies with N-Grams, HMMs, CRFs and Other 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 dynamical Bayes nets (as directed graphical models with deterministic latent variables).

10.7.1 N-grams

N-grams are non-parametric estimators of conditional probabilities based on counting relative frequencies of occurrence, and they 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, which relies on estimates $P(x_t|x_{t-1}, \dots, x_1)$ to compute $P(x_1, \dots, x_T)$. What is particular of n-grams is that

1. they estimate these conditional probabilities based only on the last $n - 1$ values (to predict the next one)
2. they assume that the data is symbolic, i.e., x_t is a symbol taken from a finite alphabet V (for vocabulary), and
3. the conditional probability estimates are obtained from frequency counts of all the observed length- n subsequences, hence the names unigram (for $n=1$), bigram (for $n=2$), trigram (for $n=3$), and n -gram in general.

The maximum likelihood estimator for these conditional probabilities is simply the relative frequency of occurrence of the left hand symbol in the context of the right hand symbols, compared to all the other possible symbols in V :

$$P(x_t|x_{t-1}, \dots, x_{t-n+1}) = \frac{\#\{x_t, x_{t-1}, \dots, x_{t-n+1}\}}{\sum_{x_t \in V} \#\{x_t, x_{t-1}, \dots, x_{t-n+1}\}} \quad (10.16)$$

where $\#\{a, b, c\}$ denotes the cardinality of the set of tuples (a, b, c) in the training set, and where the denominator is also a count (if border effects are handled properly).

A fundamental limitation of the above estimator is that it is very likely to be zero in many cases, even though the tuple $(x_t, x_{t-1}, \dots, x_{t-n+1})$ may show up in the test set. In that case, the test log-likelihood would be infinitely bad ($-\infty$). To avoid that

catastrophic outcome, n-grams employ some form of *smoothing*, i.e., techniques to shift probability mass from the observed tuples to unobserved ones that are similar (a central idea behind most non-parametric statistical methods). See [Chen and Goodman \(1999\)](#) for a review and empirical comparisons. One basic technique consists in assigning a non-zero probability mass to any of the possible next symbol values. 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 $x_{t-1}, \dots, x_{t-n+1}$ is too small, i.e., considering the contexts $x_{t-1}, \dots, x_{t-n+k}$, for increasing k , until a sufficiently reliable estimate is found.

Another interesting idea that is related to neural language models (Section 13.3) is to break up the symbols into classes (by some form of clustering) and back-up to, or mix with, less precise models that only consider the classes of the words in the context (i.e. aggregating statistics from a larger portion of the training set). One can view the word classes as a very impoverished learned representation of words which help to generalize (across words of the same class). What distributed representations (e.g. neural word embeddings) bring is a richer notion of similarity by which individual words keep their own identity (instead of being undistinguishable from the other words in the same class) and yet share learned attributes with other words with which they have some elements in common (but not all). This kind of richer notion of similarity makes generalization more specific and the representation not necessarily lossy, unlike with word classes.

10.7.2 Efficient Marginalization and Inference for Temporally Structured Outputs by Dynamic Programming

Many temporal modeling approaches can be cast in the following framework, which also includes hybrids of neural networks with HMMs, CRFs, first introduced in [Bottou et al. \(1997\)](#); [LeCun et al. \(1998a\)](#) and later developed and applied with great success in [Graves et al. \(2006\)](#); [Graves \(2012\)](#) with the Connectionist Temporal Classification 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 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 10.8 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) $\{x_t\}$ and a symbolic output sequence $\{y_t\}$ (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 $\{s_t\}$ that is also discrete

and inference needs to be done over $\{s_t\}$, either via marginalization (summing over all possible values of the state sequence) or maximization (picking exactly or approximately the MAP sequence, with the largest probability). If the state variables s_t and the target variables y_t 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 application of the principle of dynamic programming in these setups is the same as what is used in the Forward-Backward algorithm for graphical models and HMMs (next Section) and the Viterbi algorithm for inference. We outline the main lines of this type of efficient computation here.

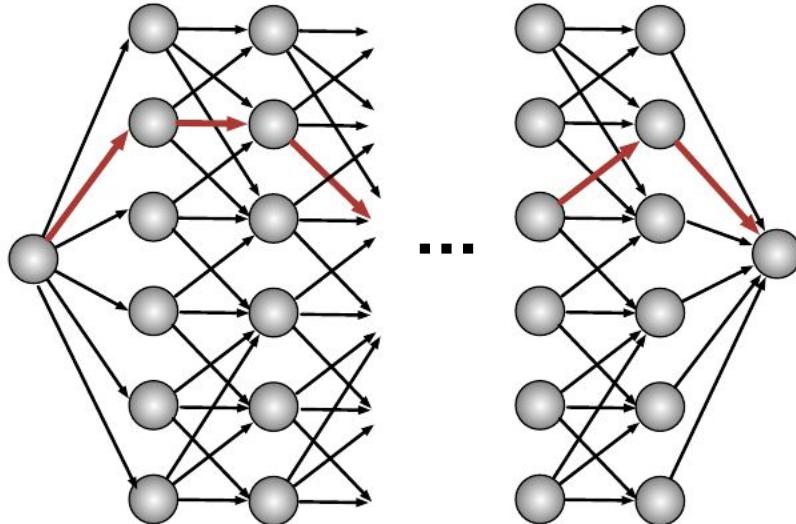


Figure 10.19: Example of a temporally structured output graph, as can be found in CRFs, HMMs, and neural net hybrids. Each node corresponds to a particular **value** of an output random variable at a particular point in the output sequence (contrast with a graphical model representation, where each node 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 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 Figure 10.19. In the above example, let z_t represent the choice variable

(e.g., s_t and y_t in the above example), and each arc with score a corresponds to a particular value of z_t in its Markov context. In the language of undirected graphical models, if a is the score associated with an arc from the node for $z_{t-1} = j$ to the one for $z_t = i$, then a is minus the energy of a term of the energy function associated with the event $1_{z_{t-1}=j, z_t=i}$ and the associated information from the input \mathbf{x} (e.g. some value of x_t).

For example, with a Markov chain of order 1, each $z_t = i$ can be linked via an arc to a $z_{t-1} = j$. The order 1 Markov property means that $P(z_t | z_{t-1}, z_{t-2}, \dots, z_1, \mathbf{x}) = P(z_t | z_{t-1}, \mathbf{x})$, where \mathbf{x} is conditioning information (the input sequence), so the number of nodes would be equal to the length of the sequence, T , times the number of values of z_t , N , and the number of arcs of the graph would be up to TN^2 (if every value of z_t can follow every value of z_{t-1} , although in practice the connectivity is often much smaller). 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 \mathbf{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:

$$S(G) = \sum_{\text{path} \in G} \prod_{a \in \text{path}} e^a \quad (10.17)$$

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). $S(G)$ may correspond to a likelihood, numerator or denominator of a probability. For example,

$$P(\{z_t\} \in Y | \mathbf{x}) = \frac{S(G_Y)}{S(G)} \quad (10.18)$$

where G_Y is the subgraph of G which is restricted to sequences that are compatible with some target answer Y .

For the **inference** task, we want to compute

$$\begin{aligned} \pi(G) &= \arg \max_{\text{path} \in G} \prod_{a \in \text{path}} e^a = \arg \max_{\text{path} \in G} \sum_{a \in \text{path}} a \\ V(G) &= \max_{\text{path} \in G} \sum_{a \in \text{path}} a \end{aligned}$$

where $\pi(G)$ is the most probable path and $V(G)$ 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 of the sequence.

The principle of dynamic programming is to recursively compute intermediate quantities that can be re-used 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 G^n . As before, G_Y^n indicates the additional restriction to subsequences that are compatible with the target sequence Y , i.e., with the beginning of the sequence Y .

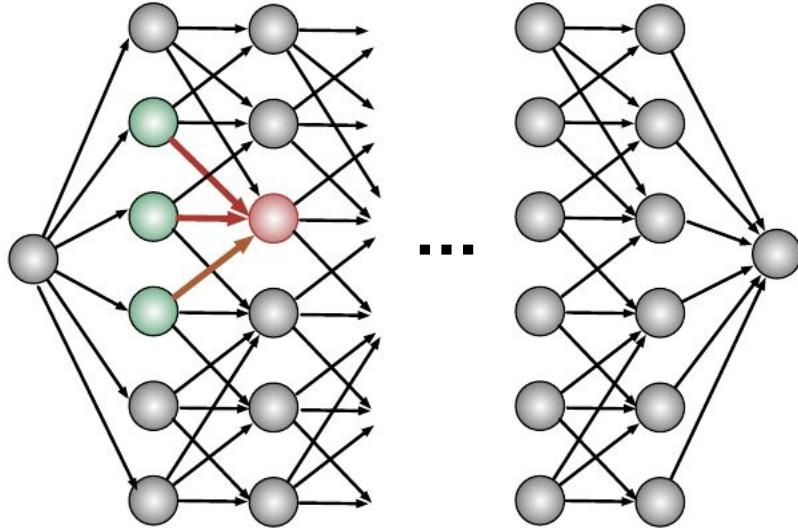


Figure 10.20: Illustration of the recursive computation taking place for inference or marginalization by dynamic programming. See Figure 10.19. These recursions involve sums or maximizations over sub-paths ending at one of the particular interior nodes (red in the figure), each time only requiring to look up previously computed values at the predecessor nodes (green).

We can thus perform marginalization efficiently as follows, and illustrated in Figure 10.20.

$$S(G) = \sum_{n \in \text{final}(G)} S(G^n) \quad (10.19)$$

where $\text{final}(G)$ is the set of final nodes in the graph G , and we can recursively compute the node-restricted sum via

$$S(G^n) = \sum_{m \in \text{pred}(n)} S(G^m) e^{a_{m,n}} \quad (10.20)$$

where $\text{pred}(n)$ is the set of predecessors of node n in the graph and $a_{m,n}$ 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. 10.17.

Similarly, we can perform efficient MAP inference (also known as Viterbi decoding)

as follows.

$$V(G) = \max_{n \in \text{final}(G)} V(G^n) \quad (10.21)$$

and

$$V(G^n) = \max_{m \in \text{pred}(n)} V(G^m) + a_{m,n}. \quad (10.22)$$

To obtain the corresponding path, it is enough to keep track of the argmax associated with each of the above maximizations and trace back $\pi(G)$ starting from the nodes in $\text{final}(G)$. For example, the last element of $\pi(G)$ is

$$n^* \leftarrow \arg \max_{n \in \text{final}(G)} V(G^n)$$

and (recursively) the argmax node before n^* along the selected path is a new n^* ,

$$n^* \leftarrow \arg \max_{m \in \text{pred}(n^*)} V(G^m) + a_{m,n^*},$$

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.

10.7.3 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 [20.2](#)). 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_t at time t , given the previous states, only depends on the previous state s_{t-1} :

$$P(s_t | s_{t-1}, s_{t-2}, \dots, s_1) = P(s_t | s_{t-1}),$$

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 $(s_t = i, s_{t-1} = j)$.

Given the state value, a generative probabilistic model of the visible variable \mathbf{x}_t is defined, that specifies how each observation \mathbf{x}_t in a sequence $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$ can be generated, via a model $P(\mathbf{x}_t | s_t)$. Two kinds of parameters are distinguished: those that define the transition distribution, which can be given by a matrix

$$A_{ij} = P(s_t = i | s_{t-1} = j),$$

and those that define the output model $P(\mathbf{x}_t|s_t)$. For example, if the data are discrete and \mathbf{x}_t is a symbol x_t , then another matrix can be used to define the output (or emission) model:

$$B_{ki} = P(x_t = k|s_t = i).$$

Another common parametrization for $P(\mathbf{x}_t|s_t = i)$, in the case of continuous vector-valued \mathbf{x}_t , is the Gaussian mixture model, where we have a different mixture (with its own means, covariances and component probabilities) for each state $s_t = i$. 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

$$P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T) = \sum_{s_1, s_2, \dots, s_T} \prod_t P(\mathbf{x}_t|s_t)P(s_t|s_{t-1}). \quad (10.23)$$

In the language established earlier in Section 10.7.2, we have a graph G with one node n per time step t and state value i , i.e., for $s_t = i$, and one arc between each node n (for $\mathbf{1}_{s_t=i}$) and its predecessors m for $\mathbf{1}_{s_{t-1}=j}$ (when the transition probability is non-zero, i.e., $P(s_t = i|s_{t-1} = j) \neq 0$). Following Eq. 10.23, the log-score $a_{m,n}$ for the transition between m and n would then be

$$a_{m,n} = \log P(x_t|s_t = i) + \log P(s_t = i|s_{t-1} = j).$$

As explained in Section 10.7.2, 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 20.2), 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

$$P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T, s_1, s_2, \dots, s_T)$$

can also be obtained using a dynamic programming algorithm (called the Viterbi algorithm). This is a form of *inference* (see Section 14.5) 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 10.7.2) for the nodes and log-score of the graph G in which we search for the optimal path, the Viterbi algorithm corresponds to the recursion defined by Eq. 10.22.

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, and approximate search must be performed. A common family of search algorithms for HMMs is the Beam Search algorithm ([Lowerre, 1976](#)): basically, a set of promising candidate paths are kept and gradually extended, at each step pruning the set of candidates to only keep the best ones according to their cumulative score (log of the joint likelihood of states and observations up to the current time step and state being considered).

More details about speech recognition are given in Section [13.2](#). An HMM can be used to associate a sequence of labels (y_1, y_2, \dots, y_N) with the input $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$, where the output sequence is typically shorter than the input sequence, i.e., $N < T$. Knowledge of (y_1, y_2, \dots, y_N) constrains the set of compatible state sequences (s_1, s_2, \dots, s_T) , and the generative conditional likelihood

$$P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T | y_1, y_2, \dots, y_N) = \sum_{s_1, s_2, \dots, s_T \in \mathcal{S}(y_1, y_2, \dots, y_N)} \prod_t P(\mathbf{x}_t | s_t) P(s_t | s_{t-1}). \quad (10.24)$$

can be computed using the same forward-backward technique, and its logarithm maximized during training, as discussed above.

Various discriminative alternatives to the generative likelihood of Eq. [10.24](#) have been proposed ([Brown, 1987](#); [Bahl et al., 1987](#); [Nadas et al., 1988](#); [Juang and Katagiri, 1992](#); [Bengio et al., 1992](#); [Bengio, 1993](#); [Leprieur and Haffner, 1995](#); [Bengio, 1999a](#)), the simplest of which is simply $P(y_1, y_2, \dots, y_N | \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$, which is obtained from Eq. [10.24](#) by Bayes rule, i.e., involving a normalization over all sequences, i.e., the unconstrained likelihood of Eq. [10.23](#):

$$P(y_1, y_2, \dots, y_N | \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T) = \frac{P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T | y_1, y_2, \dots, y_N) P(y_1, y_2, \dots, y_N)}{P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)}.$$

Both the numerator and denominator can be formulated in the framework of the previous section (Eqs. [10.18-10.20](#)), where for the numerator we merge (add) the log-scores coming from the structured output model $P(y_1, y_2, \dots, y_N)$ and from the input likelihood model $P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T | y_1, y_2, \dots, y_N)$. 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_i), associated with an input vector \mathbf{x}_t . 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 [10.8](#).

10.7.4 CRFs

Whereas HMMs are typically trained to maximize the probability of an input sequence \mathbf{x} given a target sequence \mathbf{y} 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, $P(\mathbf{y}|\mathbf{x})$. CRFs are special cases of the graph transformer model introduced in Bottou *et al.* (1997); LeCun *et al.* (1998a), where neural nets are replaced by affine transformations and there is a single graph involved.

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 14.2.2), *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 $P(\mathbf{y}|\mathbf{x})$.

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 homogenous⁴ Markov chain of order k (given the input variables). A typical linear CRF example with binary outputs would have the following structure:

$$P(\mathbf{y} = \mathbf{y}|\mathbf{x}) = \frac{1}{Z} e^{\sum_t y_t (b + \sum_j w_{ij} x_{tj}) + \sum_{i=1}^k y_t y_{t-i} (u_i + \sum_j u_{ij} x_{tj})} \quad (10.25)$$

where Z is the normalization constant, which is the sum over all \mathbf{y} sequences of the numerator. In that case, the score marginalization framework of Section 10.7.2 and coming from Bottou *et al.* (1997); LeCun *et al.* (1998a) can be applied by making terms in the above exponential correspond to scores associated with nodes t of a graph 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 formulation for Markov chains of order d is the following:

$$P(\mathbf{y} = \mathbf{y}|\mathbf{x}) = \frac{1}{Z} e^{\sum_t \sum_{d'=0}^d f_{d'}(y_t, y_{t-1}, \dots, y_{t-d'}, x_t)} \quad (10.26)$$

where $f_{d'}$ computes a potential of the energy function, a parametrized function of both the past target values (up to $y_{t-d'}$) and of the current input value x_t . For example, as discussed below $f_{d'}$ 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. 10.18-10.20). 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 \mathbf{y}), ending in each of the possible state values at t , e.g., $y_t = 1$ and $y_t = 0$ 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

⁴meaning that the same parameters are used for every time step

in the cross-product of $d - 1$ state values, i.e., N^{d-1} . 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 $N^{\max(0,d-2)}$ past values for the past $d-2$ time steps) can be obtained by only considering the sub-sums for the N^{d-1} joint state values for the last $d - 1$ time steps before $t + 1$.

Following Eq. 10.22, the same kind of decomposition can be performed to efficiently find the MAP configuration of y 's given 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).

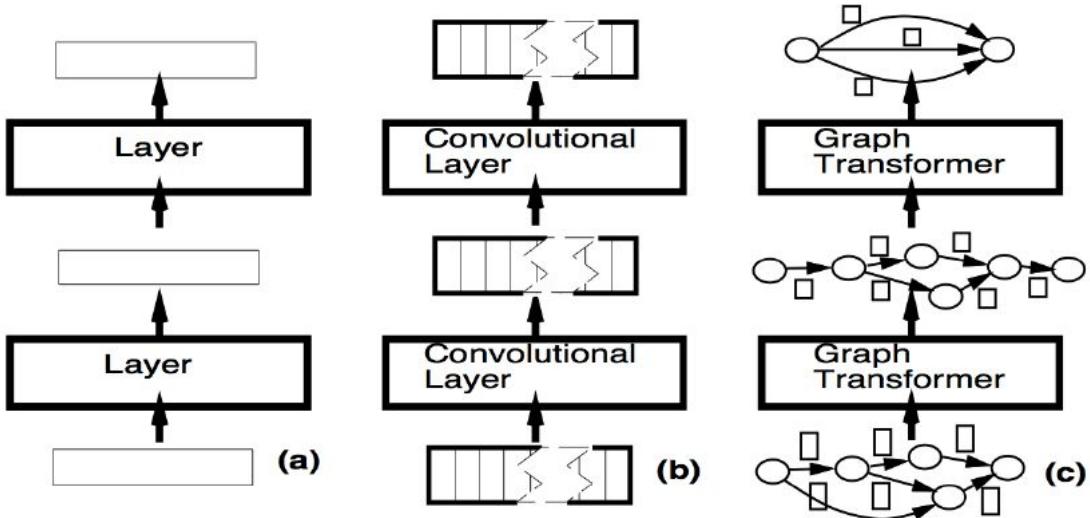


Figure 10.21: Illustration of the stacking of graph transformers (right, c) as a generalization of the stacking of convolutional layers (middle, b) or of regular feedforward layers that transform fixed-size vectors (left, a). Figure reproduced with permission from the authors of Bottou *et al.* (1997). Quoting from that paper, (c) shows that “multilayer graph transformer networks are composed of trainable modules that operate on and produce graphs whose args carry numerical information”.

10.8 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.*, 1992; 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 13.4 for combining recurrent and other deep learners with generative models such as CRFs, GSNs or RBMs.

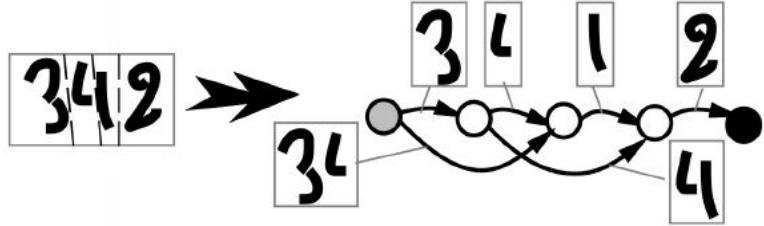


Figure 10.22: Illustration of the input and output of a simple graph transformer that maps a singleton graph corresponding to an input image to a graph representing hypothesized segmentation hypotheses. Reproduced with permission from the authors of Bottou *et al.* (1997).

The principle of efficient marginalization and inference for temporally structured outputs by exploiting dynamic programming (Sec. 10.7.2) can be applied not just when the log-scores of Eqs. 10.17 and 10.19 are parameters or linear functions of the input, but also when they are *learned non-linear functions* of the input, e.g., via a neural network transformation, as was first done in Bottou *et al.* (1997); LeCun *et al.* (1998a). These papers additionally introduced the powerful idea of *learned graph transformers*, illustrated in Figure 10.21. In this context, a graph transformer is a machine that can map a *directed acyclic graph* G_{in} to another graph G_{out} . Both input and output graphs have paths that represent hypotheses about the observed data.

For example, in the above papers, and as illustrated in Figure 10.22, a segmentation graph transformer takes a singleton input graph (the image x) and outputs a graph representing segmentation hypotheses (regarding sequences of segments that could each contain a character in the image). 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 Figures 10.23 and 10.24.

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 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.

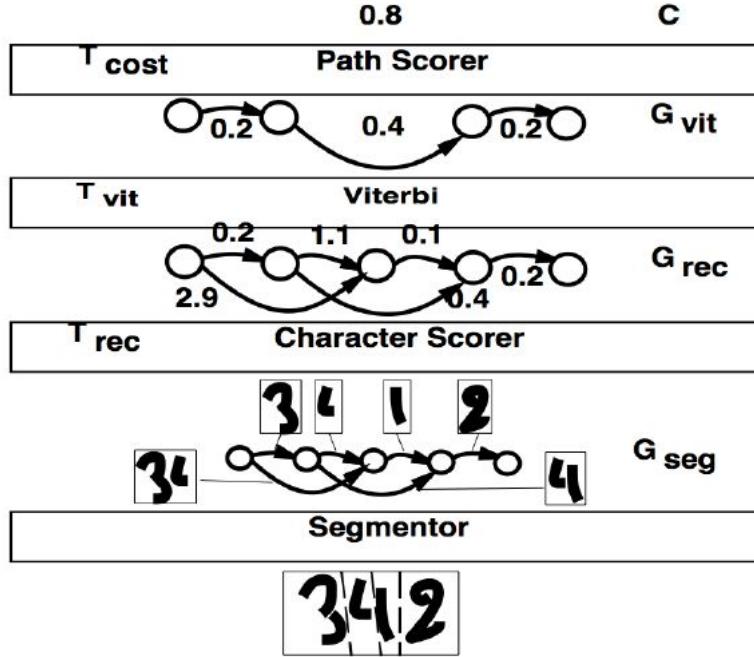


Figure 10.23: Illustration of the graph transformer network that has been used for finding the best segmentation of a handwritten word, for handwriting recognition. Reproduced with permission from Bottou *et al.* (1997).

10.8.1 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 13.3). Since the prediction at step t depends on all $t - 1$ previous choices, the number of states (nodes of the search graph G) grows exponentially with the length of the sequence.

In that case, one has to resort to *approximate 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).

The basic idea of beam search algorithms is the following:

- Break the nodes of the graph into T 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 S_t of the nodes (the “beam”), chosen based on the subset S_{t-1} . Each node

in S_t is associated with a score $\hat{V}(G^n)$ that represents an approximation (a lower bound) on the maximum total log-score of the path ending at the node, $V(G^n)$ (defined in Eq. 10.22, Viterbi decoding).

- S_t is obtained by following all the arcs from the nodes in S_{t-1} , and sorting all the resulting group t nodes n according to their estimated (lower bound) score

$$\hat{V}(G^n) = \max_{m \in S_{t-1} \text{ and } m \in \text{pred}(n)} \hat{V}(G^m) + a_{m,n},$$

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 S_t , and k is called the *beam width*.

- The estimated best final node can be read off from $\max_{n \in S_T} \hat{V}(G^n)$ 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.

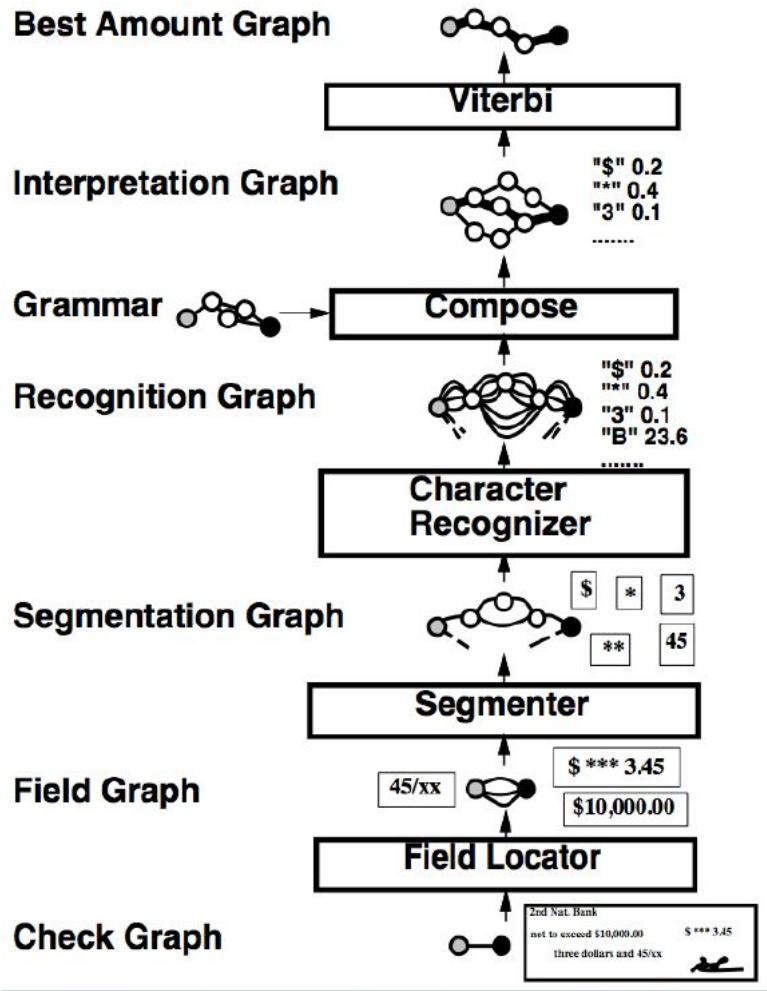


Figure 10.24: Illustration of the graph transformer network that has been used for reading amounts on checks, starting from the single graph containing the image of the graph to the recognized sequences of characters corresponding to the amount on the graph, with currency and other recognized marks. 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).

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 14

Structured Probabilistic Models: A Deep Learning Perspective

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*. 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 14.6, 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 14.1. Next, we describe how to use a graph to describe the structure of a probability distribution in section 14.2. We then revisit the challenges we described in section 14.1 and show how the structured approach to probabilistic modeling can overcome these challenges in section 14.3. 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 14.4. Finally, we close with a discussion of the unique emphasis that deep learning practitioners place on

specific approaches to graphical modeling in section 14.6.

14.1 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 images¹, audio waveforms representing speech, and documents containing multiple words and punctuation characters.

Classification algorithms 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 one 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 \mathbf{x} , the machine learning system returns an estimate of $p(\mathbf{x})$. 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.
- Denoising: given a damaged or incorrectly observed input $\tilde{\mathbf{x}}$, the machine learning system returns an estimate of the original or correct \mathbf{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 \mathbf{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 \mathbf{x} , the model is asked to return estimates of or a probability distribution over some or all of the unobserved elements of \mathbf{x} . This requires multiple outputs, and because the model could be asked to restore any of the elements of \mathbf{x} , it must understand the entire input.
- Sampling: the model generates new samples from the distribution $p(\mathbf{x})$. Applications include speech synthesis, i.e. producing new waveforms that sound like natural human speech. This requires multiple output values and a good model

¹ A *natural image* is an image that might be captured by a camera in a reasonably ordinary environment, as opposed to synthetically rendered images, screenshots of web pages, etc.

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. 14.1.

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, 32×32 pixel 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.
- **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 $P(x_1)$ or the conditional distribution $P(x_2 | x_1)$. 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

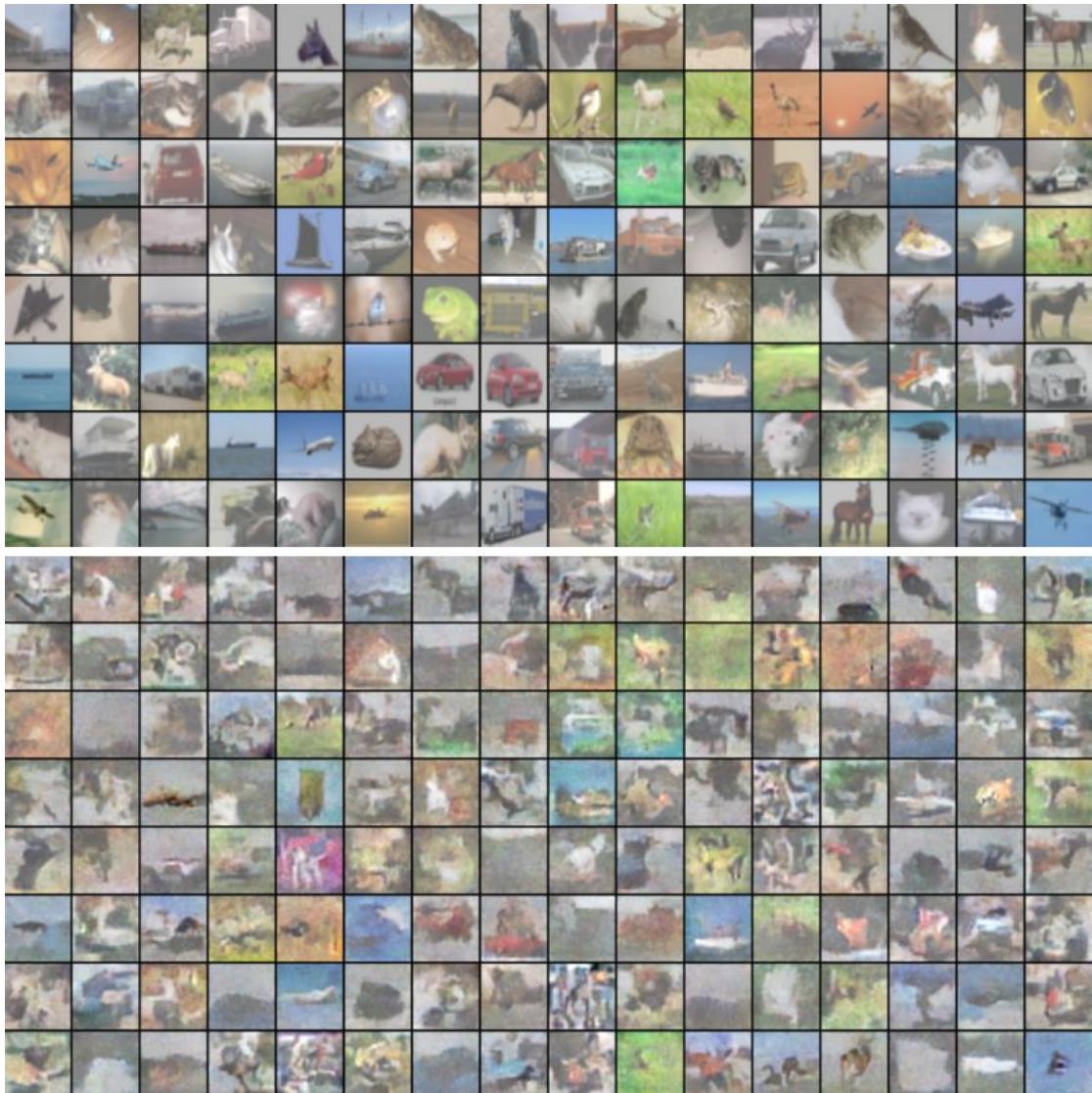


Figure 14.1: Probabilistic modeling of natural images. *Top*: Example 32×32 pixel color images from the CIFAR-10 dataset (Krizhevsky and Hinton, 2009). *Bottom*: Samples drawn from a structured probabilistic model trained on this dataset. Each sample appears at the same position in the grid as the training 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).

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.

14.2 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.

14.2.1 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

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.

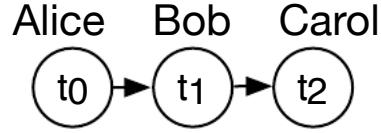


Figure 14.2: A directed graphical model depicting the relay race example. Alice’s finishing time t_0 influences Bob’s finishing time t_1 , because Bob does not get to start running until Alice finishes. Likewise, Carol only gets to start running after Bob finishes, so Bob’s finishing time t_1 influences Carol’s finishing time t_2 .

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 14.1. 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. 14.2.

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* $p(x_i | Pa_{\mathcal{G}}(x_i))$ where $Pa_{\mathcal{G}}(x_i)$ gives the parents of x_i in \mathcal{G} . The probability distribution over \mathbf{x} is given by

$$p(\mathbf{x}) = \prod_i p(x_i | Pa_{\mathcal{G}}(x_i)).$$

In our relay race example, this means that, using the graph drawn in Fig. 14.2,

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 | t_0)p(t_2 | t_1).$$

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 ($100 \text{ values of } t_0 \times 100 \text{ values of } t_1 \times 100 \text{ values of } t_2$, 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 $m \ll n$, 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 t_0 and t_1 are still directly dependent with this assumption, because 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 t_0 to t_1 . The assumption that Bob's personal running time is independent from all other factors cannot be encoded in a graph over t_0 , t_1 , and 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 t_0 and 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.

14.2.2 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

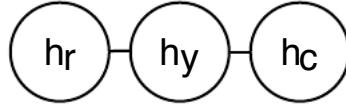


Figure 14.3: An undirected graph representing how your roommate’s health h_r , your health h_y , and your work colleague’s health h_c affect each other. You and your roommate might infect each other with a cold, and you and your work colleague might do the same, but assuming that your roommate and your colleague don’t know each other, they can only infect each other indirectly via you.

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 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. 14.3 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 graph³, 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*

$$\tilde{p}(\mathbf{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}).$$

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

³A clique of the graph is a subset of nodes that are all connected to each other by an arc of the graph.

valid probability distribution. See Fig. 14.4 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 h_y and h_c . The factor for this clique can be defined by a table, and might have values resembling these:

	$h_y = 0$	$h_y = 1$
$h_c = 0$	2	1
$h_c = 1$	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 h_y and h_r .

14.2.3 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 distribution⁴:

$$p(\mathbf{x}) = \frac{1}{Z} \tilde{p}(\mathbf{x})$$

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 ϕ functions are held constant. Note that if the ϕ 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 ϕ 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 Chap. 19.

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

⁴A distribution defined by normalizing a product of clique potentials is also called a *Gibbs distribution*.

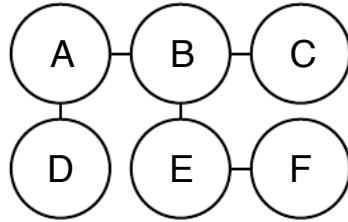


Figure 14.4: This graph implies that $p(A, B, C, D, E, F)$ can be written as $\frac{1}{Z} \phi_{A,B}(A, B) \phi_{B,C}(B, C) \phi_{A,D}(A, D) \phi_{B,E}(B, E) \phi_{E,F}(E, F)$ for an appropriate choice of the ϕ functions.

are continuous and the integral of \tilde{p} over their domain diverges. For example, suppose we want to model a single scalar variable $x \in \mathbb{R}$ 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 ϕ functions determines whether the probability distribution is defined. For example, for $\phi(x; \beta) = \exp(-\beta x^2)$, the β parameter determines whether Z exists. Positive β results in a Gaussian distribution over x but all other values of β make ϕ 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 ϕ 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 ϕ 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 \mathbf{b} . Suppose we have one clique for each element of \mathbf{x} , $\phi_i(x_i) = \exp(b_i x_i)$. 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 $p(x_i = 1) = \text{sigmoid}(b_i)$. If the domain of \mathbf{x} is the set of elementary basis vectors ($\{[1, 0, \dots, 0], [0, 1, \dots, 0], \dots, [0, 0, \dots, 1]\}$) then $p(\mathbf{x}) = \text{softmax}(\mathbf{b})$, so a large value of b_i actually reduces $p(x_j = 1)$ 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 ϕ functions. We'll explore a practical application of this idea later, in Chap. 21.6.

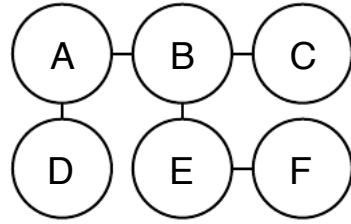


Figure 14.5: This graph implies that $E(a, b, c, d, e, f)$ can be written as $E_{a,b}(a, b) + E_{b,c}(b, c) + E_{a,d}(a, d) + E_{b,e}(b, e) + E_{e,f}(e, f)$ for an appropriate choice of the per-clique energy functions. Note that we can obtain the ϕ functions in Fig. 14.4 by setting each ϕ to the exp of the corresponding negative energy, e.g., $\phi_{a,b}(a, b) = \exp(-E(a, b))$.

14.2.4 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

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \quad (14.1)$$

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 optimization⁵, and the probabilities in the model can approach arbitrarily close to zero but never reach it.

Any distribution of the form given by equation 14.1 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. 14.5 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. 14.1. This $-$ sign could be incorporated into the definition of E , or for many functions E the learning algorithm could

⁵For some models, we may still need to use constrained optimization to make sure Z exists.

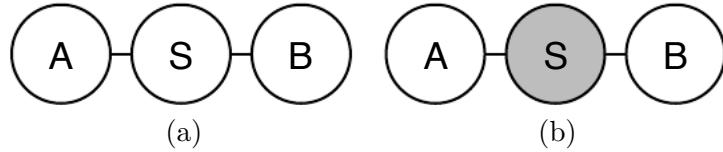


Figure 14.6: a) The path between random variable a and random variable b through s is active, because s is not observed. This means that a and b are not separated. b) Here s is shaded in, to indicate that it is observed. Because the only path between a and b is through s , and that path is inactive, we can conclude that a and b are separated given s .

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.

14.2.5 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. 14.6 for a depiction of how active and inactive paths in an undirected look when drawn in this way. See Fig. 14.7 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:

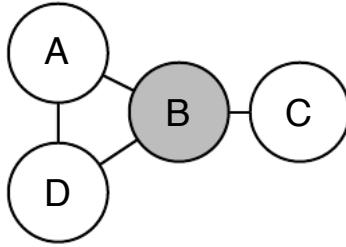


Figure 14.7: An example of reading separation properties from an undirected graph. Here b is shaded to indicate that it is observed. Because observing b blocks the only path from a to c, we say that a and c are separated from each other given b. The observation of b also blocks one path between a and d, but there is a second, active path between them. Therefore, a and d are not separated given b.

We say that a set of variables \mathbb{A} is \mathbb{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 \mathbb{d} -separated if no such path exists. In directed nets, determining whether a path is active is somewhat more complicated. See Fig. 14.8 for a guide to identifying active paths in a directed model. See Fig. 14.9 for an example of reading some properties from a graph.

It is important to remember that separation and \mathbb{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.

14.2.6 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.

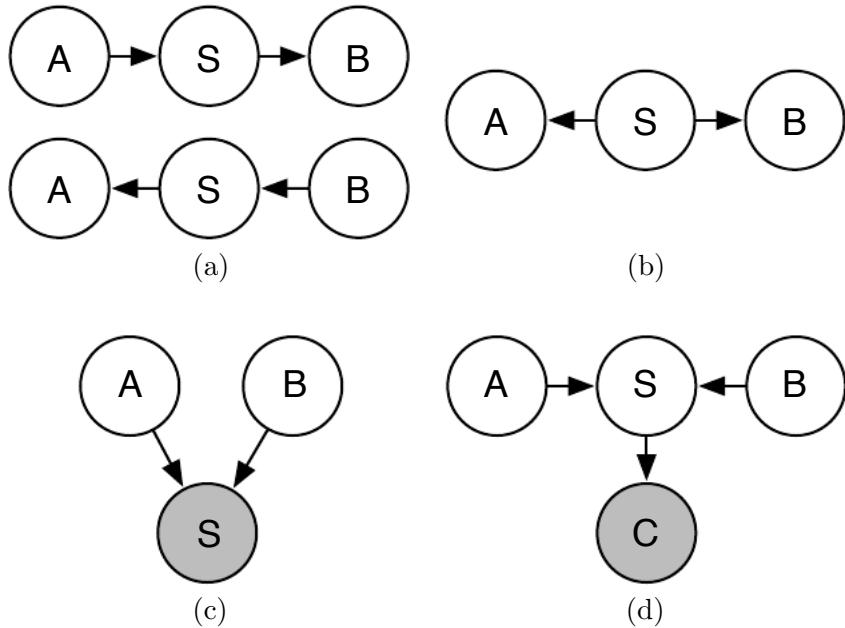


Figure 14.8: All of the kinds of active paths of length two that can exist between random variables a and rb . a) Any path with arrows proceeding directly from a to b or vice versa. This kind of path becomes blocked if s is observed. We have already seen this kind of path in the relay race example. b) 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 b 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.

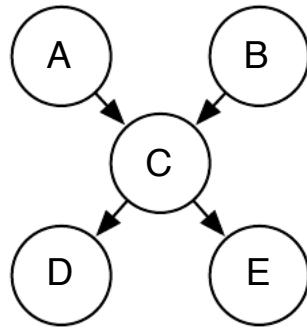


Figure 14.9: From this graph, we can read out several d-separation properties. Examples include:

- 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.

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 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. 14.10 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

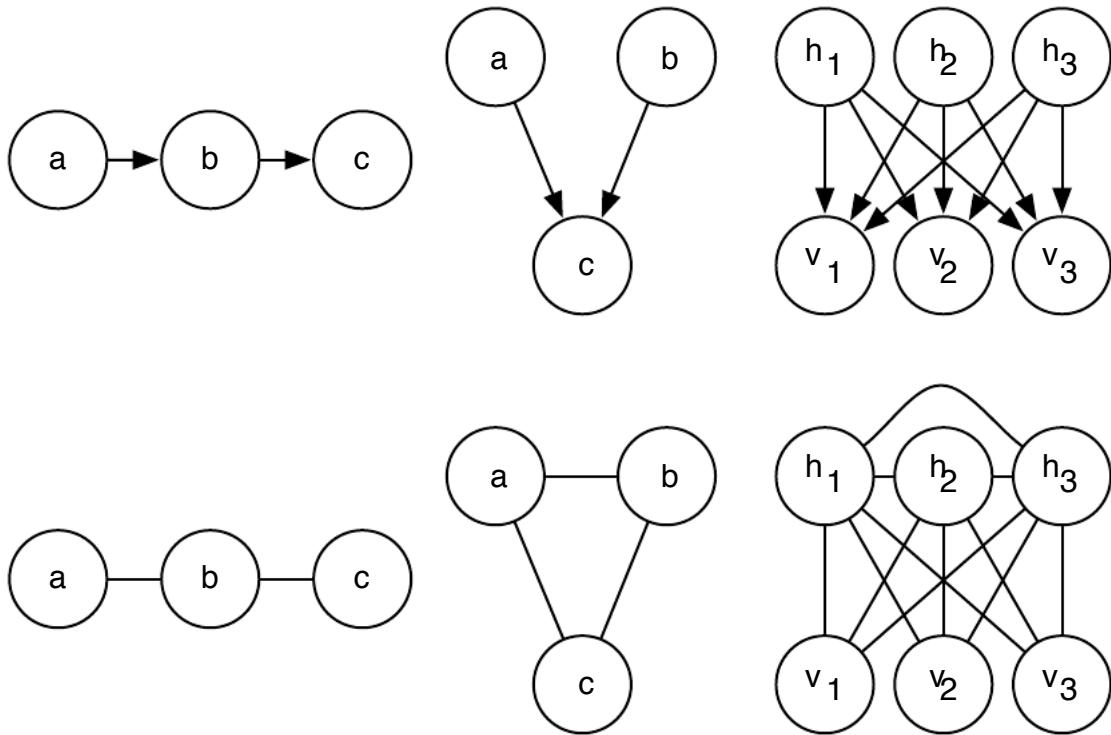


Figure 14.10: Examples of converting directed models to undirected models by constructing moralized graphs. *Left)* This simple chain can be converted to a moralized graph merely by replacing its directed edges with undirected edges. The resulting undirected model implies exactly the same set of independences 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 latent variables, thus introducing a quadratic number of new direct dependences.

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

14.2.7 Marginalizing Variables out of a Graph

TODO: marginalizing variables out of a graph

14.2.8 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 ϕ function must be a subset of some clique in the graph. However, it is not necessary that there exist any ϕ whose scope contains the entirety of every clique. Factor graphs explicitly represent the scope of each ϕ 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 ϕ 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. 14.11 for an example of how factor graphs can resolve ambiguity in the interpretation of undirected networks.

14.3 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

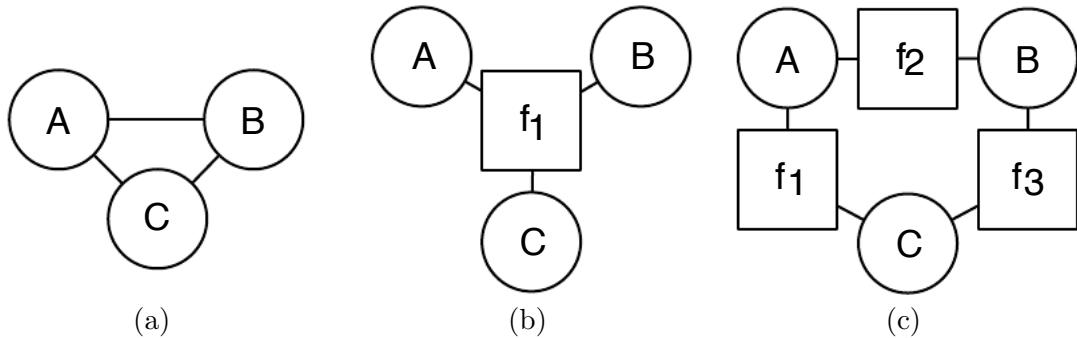


Figure 14.11: An example of how a factor graph can resolve ambiguity in the interpretation of undirected networks. a) An undirected network with a clique involving three variables A , B , and C . b) A factor graph corresponding to the same undirected model. This factor graph has one factor over all three 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.

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

14.4 Learning About Dependencies

We consider 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.

14.4.1 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 $p(v)$.

14.4.2 Latent Variables for Feature Learning

Another advantage of using latent variables is that they often develop useful semantics.

As discussed in section 3.10.5, 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 16 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 kinds of interactions can create even richer descriptions of the input. Most of the approaches mentioned in sec. 14.4.2 accomplish feature learning by learning latent variables. Often, given some model of v and h , it turns out that $\mathbb{E}[h | v]$ TODO: uh-oh, is there a collision between set notation and expectation notation? or $\text{argmax}_h p(h, v)$ is a good feature mapping for v .

TODO: appropriate links to Monte Carlo methods chapter spun off from here

14.5 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 h given values of the visible variables 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

14.5.1 Reparametrization Trick

Sometimes, in order to estimate the stochastic gradient of an expected loss over some random variable h , with respect to parameters that influence 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

$$\mathbf{h} \sim p(\mathbf{h}|\theta) \quad (14.2)$$

as

$$\mathbf{h} = f(\theta, \eta) \quad (14.3)$$

where η is some independent noise source of the appropriate dimension with density $p(\eta)$, and f is a continuous (differentiable almost everywhere) function. Basically, 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

$$\frac{\partial}{\partial \theta} \int L(\mathbf{h}) p(\mathbf{h}|\theta) d\mathbf{h} \quad (14.4)$$

where the parameters θ 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 achieve⁶ this is to perform the reparametrization in Eq. 14.3 and the corresponding change of variable in the integral of Eq. 14.4, integrating over η rather than \mathbf{h} :

$$\frac{\partial}{\partial \theta} \int L(f(\theta, \eta)) p(\eta) d\eta. \quad (14.5)$$

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 used by Bengio (2013); Bengio *et al.* (2013a) to train a neural network with stochastic hidden units. It was described at the same time by Kingma (2013), but see the further developments in Kingma and Welling (2014b). It was used to train generative stochastic networks (GSNs) (Bengio *et al.*, 2014a,b), described in Section 21.10, 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 21.8.2.

⁶compared to approaches that do not back-propagate through the generation of \mathbf{h}

14.6 The Deep Learning Approach to Structured Probabilistic Modeling

Deep learning practitioners 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 re-useable 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

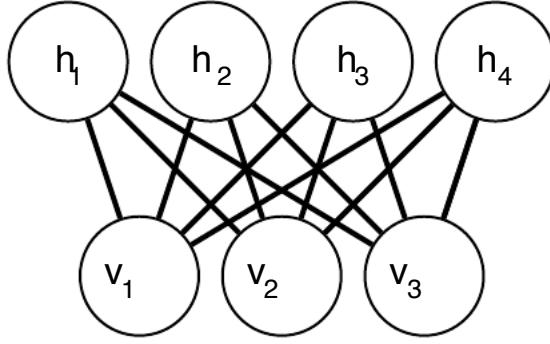


Figure 14.12: An example RBM drawn as a Markov network

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.

14.6.1 Example: The Restricted Boltzmann Machine

TODO: rework this section. Add pointer to Chapter 21.1. 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

$$E(v, h) = -b^\top v - c^\top h - v^\top Wh$$

where \mathbf{b} , \mathbf{c} , and \mathbf{W} are unconstrained, real-valued, learnable parameters. The model is depicted graphically in Fig. 14.12. 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

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_i p(h_i \mid \mathbf{v})$$

and

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_i p(v_i \mid \mathbf{h}).$$

The individual conditionals are simple to compute as well, for example

$$p(h_i = 1 | \mathbf{v}) = \sigma(\mathbf{v}^\top \mathbf{W}_{:,i} + b_i).$$

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,

$$\frac{\partial}{\partial \mathbf{W}_{i,j}} \mathbb{E}_{\mathbf{v}, \mathbf{h}} E(\mathbf{v}, \mathbf{h}) = -v_i h_j.$$

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 15

Monte Carlo Methods

TODO plan organization of chapter (spun off from graphical models chapter)

15.1 Markov Chain Monte Carlo Methods

Drawing a sample x from the probability distribution $p(x)$ 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 $p(a, b)$. In order to sample a , we must draw it from $p(a | b)$, and in order to sample b , we must draw it from $p(b | a)$. 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 $T(\mathbf{x}' | \mathbf{x})$. Running the Markov chain means repeatedly updating the state \mathbf{x} to a value \mathbf{x}' sampled from $T(\mathbf{x}' | \mathbf{x})$.

Under certain distributions, a Markov chain is eventually guaranteed to draw \mathbf{x} from an equilibrium distribution $\pi(\mathbf{x}')$, defined by the condition

$$\forall \mathbf{x}', \pi(\mathbf{x}') = \sum_{\mathbf{x}} T(rvx' | \mathbf{x})\pi(\mathbf{x}).$$

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 π as a vector (with the probability for each possible value \mathbf{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 π 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 $T(x' | x) > 0$). 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 $x_0 \sim p_0$, 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 $t \rightarrow \infty$, 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, 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 $T(\mathbf{x}' | \mathbf{x})$ is accomplished by selecting one variable 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

15.1.1 Markov Chain Theory

TODO

State Perron's theorem

DEFINE detailed balance

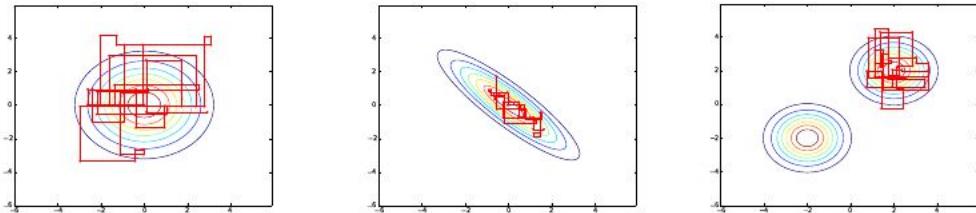


Figure 15.1: Paths followed by Gibbs sampling for three distributions, with the Markov chain initialized at the mode in both cases. Left) A multivariate normal distribution with two independent variables. Gibbs sampling *mixes* well because the variables are independent. Center) A multivariate normal 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.

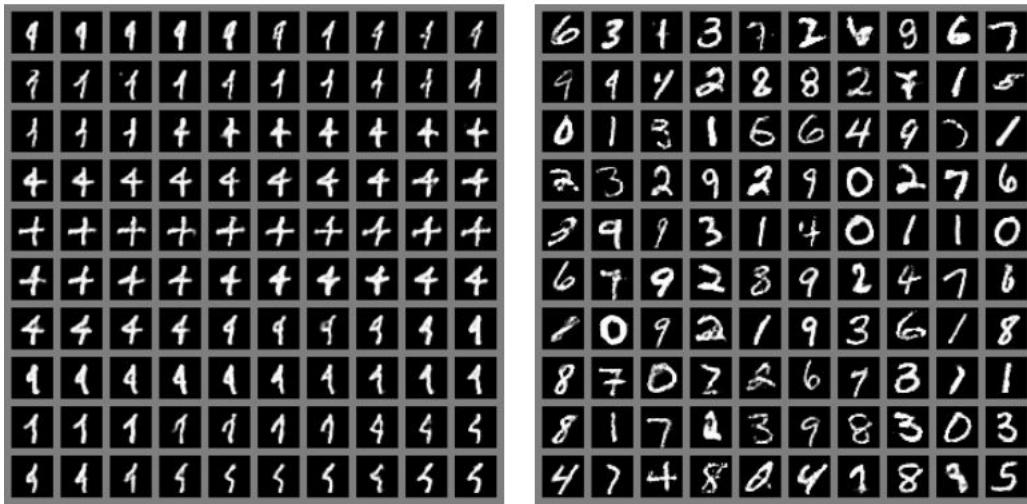


Figure 15.2: An illustration of the slow mixing problem in deep probabilistic models. Each panel should be read left to right, top to bottom. Left) Consecutive samples from Gibbs sampling applied to a deep Boltzmann machine trained on the MNIST dataset. Consecutive samples are similar to each other. 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 16

Linear Factor Models and Auto-Encoders

Linear factor models are generative unsupervised learning models in which we imagine that some unobserved factors \mathbf{h} explain the observed variables \mathbf{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 \mathbf{x} to \mathbf{h} , typically a feedforward neural network, but not necessarily. They also learn a transformation going backwards from the representation to the data, from \mathbf{h} to \mathbf{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 \mathbf{h} that could have generated the observed \mathbf{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 18 and 21.10.

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 16.1:

- an input, \mathbf{x}
- an encoder function f
- a “code” or internal representation $\mathbf{h} = f(\mathbf{x})$
- a decoder function g
- an output, also called “reconstruction” $\mathbf{r} = g(\mathbf{h}) = g(f(\mathbf{x}))$

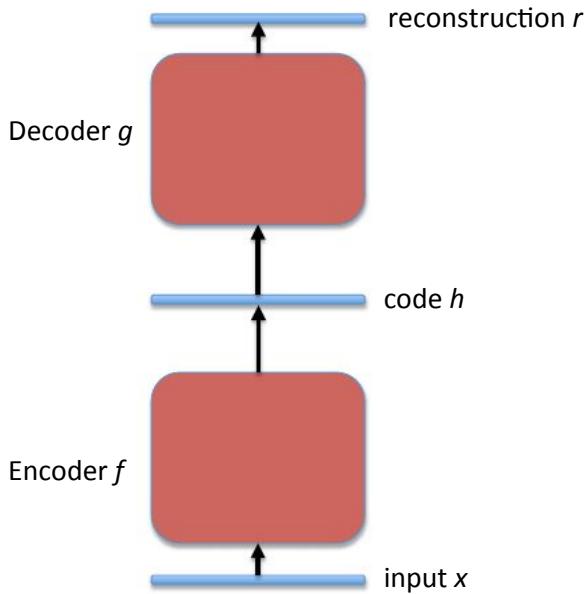


Figure 16.1: General schema of an auto-encoder, mapping an input \mathbf{x} to an output (called reconstruction) \mathbf{r} through an internal representation or code \mathbf{h} . The auto-encoder has two components: the encoder f (mapping \mathbf{x} to \mathbf{h}) and the decoder g (mapping \mathbf{h} to \mathbf{r}).

- a loss function L computing a scalar $L(\mathbf{r}, \mathbf{x})$ measuring how good of a reconstruction \mathbf{r} is of the given input \mathbf{x} . The objective is to minimize the expected value of L over the training set of examples $\{\mathbf{x}\}$.

16.1 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 \mathbf{h} to be smaller than the dimension of the input \mathbf{x} .

Figure 16.2 illustrates the two typical cases of auto-encoders: undercomplete vs overcomplete, i.e., with the dimension of the representation \mathbf{h} respectively smaller vs larger than the input \mathbf{x} . Whereas early work with auto-encoders, just like PCA, uses the undercompleteness – i.e. a bottleneck in the sequence of layers – to avoid learning the identity function, more recent work 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

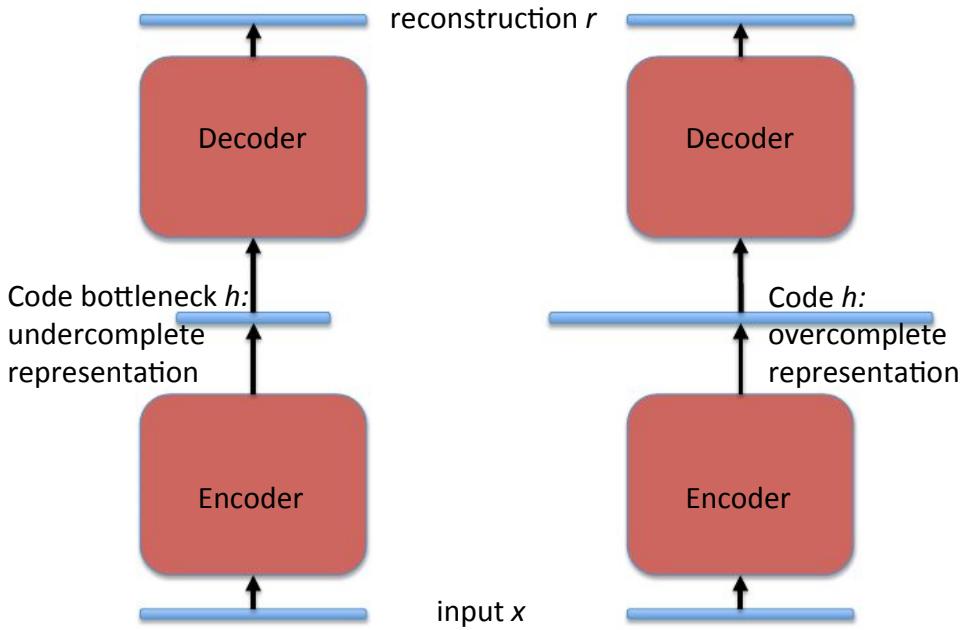


Figure 16.2: Left: undercomplete representation (dimension of code h is less than dimension of input x). Right: overcomplete representation. Overcomplete auto-encoders require some other form of regularization (instead of the constraint on the dimension of h) to avoid the trivial solution where $r = x$ for all x .

capture the input distribution (indirectly, not as 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 system among the probable x 's) could be much smaller if most of the elements of h are zero (or any other constant, such that $\|\frac{\partial h_i}{\partial x}\|$ is close to zero). When $\|\frac{\partial h_i}{\partial x}\|$ is close to zero, h_i does not participate in encoding local changes in x . There is a geometrical interpretation of this situation in terms of *manifold learning* that is discussed in more depth in Chapter 18. The discussion in Chapter 17 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 inference instead to compute the 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 instead considered like free variable that is obtained through an optimization, i.e., a particular form of inference:

$$\mathbf{h}^* = f(\mathbf{x}) = \arg \min_{\mathbf{h}} L(g(\mathbf{h}), \mathbf{x}) + \lambda \Omega(\mathbf{h}) \quad (16.1)$$

where L is the reconstruction loss, f the (non-parametric) encoder, g the (parametric) decoder, $\Omega(\mathbf{h})$ 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 16.7. It also has an interpretation as a directed graphical model, described in more details in Section 20.3. 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 $|\mathbf{h}|_1 = \sum_i |h_i|$.

- 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 16.7.2.
- 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 16.7.1. Besides the L1 penalty, other sparsity penalties that have been explored include the Student-t penalty (Olshausen and Field, 1996; Bergstra, 2011),

$$\sum_i \log(1 + \alpha^2 h_i^2)$$

(i.e. where αh_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 16.9, explicitly penalize $\|\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\|_F^2$, i.e., the sum of the squared norm of the vectors

$\frac{\partial h_i(\mathbf{x})}{\partial \mathbf{x}}$ (each indicating how much each hidden unit h_i responds to changes in \mathbf{x} and what direction of change in \mathbf{x} that unit is most sensitive to, around a particular \mathbf{x}). With such a regularization penalty, the auto-encoder is called **contractive**¹ because the mapping from input \mathbf{x} to representation \mathbf{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 16.8. 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 \mathbf{x} to $\mathbf{r} = g(f(\mathbf{x}))$. In other words, since both \mathbf{x} and $\mathbf{x} + \epsilon$ (where ϵ is some small noise vector) must yield the same target output \mathbf{x} , the reconstruction function is encouraged to be insensitive to changes in all directions ϵ . The only thing that prevents reconstruction \mathbf{r} from simply being a constant (completely insensitive to the input \mathbf{x}), is that one also has to reconstruct correctly for different training examples \mathbf{x} . However, the auto-encoder can learn to be approximately constant around training examples \mathbf{x} while producing a different answer for different training examples. As discussed in Section 18.3, 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 21.8.2) and the *generative stochastic networks* (Section 21.10) also involve the injection of noise, but typically in the representation-space itself, thus introducing the notion of \mathbf{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

$$-\log P(\mathbf{h})$$

which captures the assumption that we would like to find a representation that has a simple distribution (if $P(\mathbf{h})$ has a simple form, such as a factorized distribution²),

¹A function $f(\mathbf{x})$ is contractive if $\|f(\mathbf{x}) - f(\mathbf{y})\| < \|\mathbf{x} - \mathbf{y}\|$ for nearby \mathbf{x} and \mathbf{y} , or equivalently if its derivative $\|f'(\mathbf{x})\| < 1$.

²all the sparse priors we have described correspond to a factorized distribution

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 \mathbf{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 21.8.2) 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 $-\log P(\mathbf{h})$ 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, the prior on the latent variables acts like a *data-dependent prior*, in the sense that it depends on the particular values \mathbf{h} that are going to be sampled (usually from a posterior or an encoder), based on the input example \mathbf{x} . Of course, indirectly, this is also a regularization on the parameters, but one that depends on the particular data distribution.

16.2 Representational Power, Layer Size and Depth

Nothing in the above description of auto-encoders restricts the encoder or decoder to be shallow, but in the literature on the subject, most trained auto-encoders have had a single hidden layer which is also the representation layer or code³

For one, we know by the usual universal approximator abilities of single hidden-layer neural networks that a sufficiently large hidden layer can represent any function with a given accuracy. This observation justifies overcomplete auto-encoders: in order to represent a rich enough distribution, one probably needs many hidden units in the intermediate representation layer. We also know that Principal Components Analysis (PCA) corresponds to an undercomplete auto-encoder with no intermediate non-linearity, and that PCA can only capture a set of directions of variation that are the same everywhere in space. This notion is discussed in more details in Chapter 18 in the context of manifold learning.

For two, it has also been reported many times that training a deep neural network, and in particular a deep auto-encoder (i.e. with a deep encoder and a deep decoder) is more difficult than training a shallow one. This was actually a motivation for the initial work on the *greedy layerwise unsupervised pre-training procedure*, described below in Section 17.1, by which we only need to train a series of shallow auto-encoders in order to initialize a deep auto-encoder. It was shown early on (Hinton and Salakhutdinov, 2006)

³as argued in this book, this is probably not a good choice, and we would like to independently control the constraints on the representation, e.g. dimension and sparsity of the code, and the capacity of the encoder.

that, if trained properly, such deep auto-encoders could yield much better compression than corresponding shallow or linear auto-encoders (which are basically doing the same as PCA, see Section 16.5 below). As discussed in Section 17.7, deeper architectures can be in some cases exponentially more efficient (both in terms of computation and statistically) than shallow ones. However, because we can usefully pre-train a deep net by training and stacking shallow ones, it makes it interesting to consider single-layer (or at least shallow and easy to train) auto-encoders, as has been done in most of the literature discussed in this chapter.

16.3 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 \mathbf{x} is a vector of discrete variables or when $P(\mathbf{x}|\mathbf{h})$ is not well approximated by a Gaussian distribution⁴. Just like in the case of other types of neural networks (starting with the feedforward neural networks, Section 6.2.2), 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 21.8.2, 21.9 and 21.10 about generative extensions of auto-encoders and stochastic recurrent networks, where the output of the auto-encoder is interpreted as a probability distribution $P(\mathbf{x}|\mathbf{h})$, for reconstructing \mathbf{x} , given hidden units \mathbf{h} . This distribution captures not just the expected reconstruction but also the *uncertainty* about the original \mathbf{x} (which gave rise to \mathbf{h} , either deterministically or stochastically, given \mathbf{h}). In the simplest and most ordinary cases, this distribution factorizes, i.e., $P(\mathbf{x}|\mathbf{h}) = \prod_i P(x_i|\mathbf{h})$. This covers the usual cases of $x_i|\mathbf{h}$ being Gaussian (for unbounded real values) and $x_i|\mathbf{h}$ having a Bernoulli distribution (for binary values x_i), but one can readily generalize this to other distributions, such as mixtures (see Sections 3.10.5 and 6.2.2).

Thus we can generalize the notion of *decoding function* $g(\mathbf{h})$ to *decoding distribution* $P(\mathbf{x}|\mathbf{h})$. Similarly, we can generalize the notion of *encoding function* $f(\mathbf{x})$ to *encoding distribution* $Q(\mathbf{h}|\mathbf{x})$, as illustrated in Figure 16.3. We use this to capture the fact that noise is injected at the level of the representation \mathbf{h} , now considered like a latent variable. This generalization is crucial in the development of the variational auto-encoder (Section 21.8.2) and the generalized stochastic networks (Section 21.10).

We also find a stochastic encoder and a stochastic decoder in the RBM, described in Section 21.1. In that case, the encoding distribution $Q(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ “match”, in the sense that $Q(\mathbf{h}|\mathbf{x}) = P(\mathbf{h}|\mathbf{x})$, i.e., there is a unique joint distribution which has both $Q(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ as conditionals. This is not true in general for two independently parametrized conditionals like $Q(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$, 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).

⁴See the link between squared error and normal density in Sections 5.6 and 6.2.2

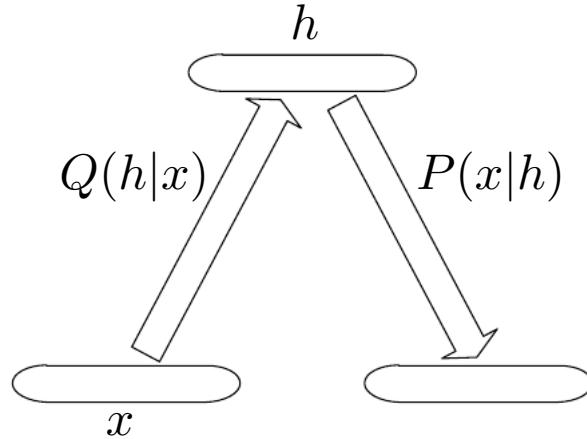


Figure 16.3: Basic scheme of a stochastic auto-encoder, in which both the encoder and the decoder are not simple functions but instead involve some noise injection, meaning that their output can be seen as sampled from a distribution, $Q(\mathbf{h}|\mathbf{x})$ for the encoder and $P(\mathbf{x}|\mathbf{h})$ for the decoder. RBMs are a special case where $P = Q$ (in the sense of a unique joint corresponding to both conditionals) but in general these two distributions are not necessarily conditional distributions compatible with a unique joint distribution $P(\mathbf{x}, \mathbf{h})$.

16.4 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 \mathbf{h} generates \mathbf{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,

$$\mathbf{h} \sim P(\mathbf{h}), \quad (16.2)$$

and then sample the real-valued observable variables given the factors:

$$\mathbf{x} = \mathbf{W}\mathbf{h} + \mathbf{b} + \text{noise} \quad (16.3)$$

where the noise is typically Gaussian and diagonal (independent across dimensions). This is illustrated in Figure 16.4.

16.5 Probabilistic PCA and Factor Analysis

Probabilistic PCA (Principal Components Analysis) and factor analysis are both special cases of the above equations (16.2 and 16.3) and only differ in the choices made for the

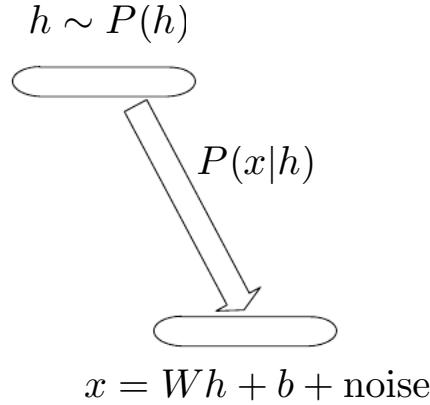


Figure 16.4: Basic scheme of a linear factors model, in which we assume that an observed data vector \mathbf{x} is obtained by a linear combination of latent factors \mathbf{h} , plus some noise. Different models, such as probabilistic PCA, factor analysis or ICA, make different choices about the form of the noise and of the prior $P(\mathbf{h})$.

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

$$\mathbf{h} \sim \mathcal{N}(0, \mathbf{I})$$

while the observed variables x_i are assumed to be *conditionally independent*, given \mathbf{h} , i.e., the noise is assumed to be coming from a diagonal covariance Gaussian distribution, with covariance matrix $\boldsymbol{\psi} = \text{diag}(\boldsymbol{\sigma}^2)$, with $\boldsymbol{\sigma}^2 = (\sigma_1^2, \sigma_2^2, \dots)$ 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 \mathbf{x} is just a Gaussian-distribution (multivariate normal) random variable, with

$$\mathbf{x} \sim \mathcal{N}(\mathbf{b}, \mathbf{W}\mathbf{W}^\top + \boldsymbol{\psi})$$

where we see that the weights \mathbf{W} induce a dependency between two variables x_i and x_j through a kind of auto-encoder path, whereby x_i influences $\hat{\mathbf{h}}_k = \mathbf{W}_k \mathbf{x}$ via w_{ki} (for every k) and $\hat{\mathbf{h}}_k$ 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 σ_i equal to each other. In that case the covariance of \mathbf{x} is just $\mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}$, where σ^2 is now a scalar, i.e.,

$$\mathbf{x} \sim \mathcal{N}(\mathbf{b}, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I})$$

or equivalently

$$\mathbf{x} = \mathbf{W}\mathbf{h} + \mathbf{b} + \sigma\mathbf{z}$$

where $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ is white noise. Tipping and Bishop (1999) then show an iterative EM algorithm for estimating the parameters \mathbf{W} and σ^2 .

What the probabilistic PCA model is basically saying is that the covariance is mostly captured by the latent variables \mathbf{h} , up to some small residual *reconstruction error* σ^2 . As shown by Tipping and Bishop (1999), probabilistic PCA becomes PCA as $\sigma \rightarrow 0$. In that case, the conditional expected value of \mathbf{h} given \mathbf{x} becomes an orthogonal projection onto the space spanned by the d columns of \mathbf{W} , like in PCA. See Section 18.1 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 \mathbf{x} . That section also explains the very insightful *geometric and manifold interpretation* of PCA.

However, as $\sigma \rightarrow 0$, the density model becomes very sharp around these d dimensions spanned the columns of \mathbf{W} , as discussed in Section 18.1, 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 18 for more).

16.5.1 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. 16.2 and 16.3. 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.,

$$P(\mathbf{h}) = \prod_i P(h_i). \quad (16.4)$$

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 ($P(\mathbf{x}|\mathbf{h})$) and parts being non-specified or non-parametric ($P(\mathbf{h})$). In fact, this typically yields to *non-Gaussian* priors: if the priors were Gaussian, then one could not distinguish between the factors \mathbf{h} and a rotation of \mathbf{h} . Indeed, note that if

$$\mathbf{h} = \mathbf{U}\mathbf{z}$$

with \mathbf{U} an orthonormal (rotation) square matrix, i.e.,

$$\mathbf{z} = \mathbf{U}^\top \mathbf{h},$$

then, although \mathbf{h} might have a $\text{Normal}(0, \mathbf{I})$ distribution, the \mathbf{z} *also have a unit covariance*, i.e., they are uncorrelated:

$$\text{Var}[\mathbf{z}] = \mathbb{E}[\mathbf{z}\mathbf{z}^\top] = \mathbb{E}[\mathbf{U}^\top \mathbf{h}\mathbf{h}^\top \mathbf{U}] = \mathbf{U}^\top \text{Var}[\mathbf{h}] \mathbf{U} = \mathbf{U}^\top \mathbf{U} = \mathbf{I}.$$

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 \mathbf{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 $\mathbf{s} = \mathbf{V}\mathbf{x}$ such that they are *maximally non-Gaussian*. An intuitive explanation for these approaches is that although the true latent variables \mathbf{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., $\mathbf{x} = f(\mathbf{h}) + \text{noise}$. 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).

16.5.2 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 tail⁵. Like the other linear factor models (Eq. 16.3), sparse coding corresponds to a linear factor model, but one with a “sparse” latent variable \mathbf{h} , i.e., $P(\mathbf{h})$ 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

$$P(\mathbf{h}) = \prod_i P(h_i) = \prod_i \frac{\lambda}{2} e^{-\lambda|h_i|} \quad (16.5)$$

and the factorized Student-t prior is

$$P(\mathbf{h}) = \prod_i P(h_i) \propto \prod_i \frac{1}{1 + \frac{h_i^2}{\nu}}^{\frac{\nu+1}{2}}. \quad (16.6)$$

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 recon-

⁵with 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.

struction 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 $P(\mathbf{h}|\mathbf{x})$ will not generate values $\mathbf{h} = 0$. However, sparse coding is normally considered under a maximum a posteriori (MAP) inference framework, in which the inferred values of \mathbf{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. 16.1, reproduced here,

$$\mathbf{h} = f(\mathbf{x}) = \arg \min_{\mathbf{h}} L(g(\mathbf{h}), \mathbf{x})) + \lambda \Omega(\mathbf{h})$$

where $L(g(\mathbf{h}), \mathbf{x})$ is interpreted as $-\log P(\mathbf{x}|g(\mathbf{h}))$ and $\Omega(\mathbf{h})$ as $-\log P(\mathbf{h})$. This MAP inference view of sparse coding and an interesting probabilistic interpretation of sparse coding are further discussed in Section 20.3.

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 $P(\mathbf{h})$, 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 16.5.1.

See Section 18.2 about the manifold interpretation of sparse coding.

TODO: relate to and point to Spike-and-slab sparse coding (Goodfellow *et al.*, 2012) (section?)

16.6 Probabilistic Interpretation of 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 \mathbf{x} , given the representation \mathbf{h} .

We have already covered negative log-likelihood as a loss function in general for feed-forward neural networks in Section 6.2.2. 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

$$L = -\log P(\mathbf{x}|\mathbf{h})$$

where \mathbf{h} is the representation, which may generally be obtained through an encoder taking \mathbf{x} as input.

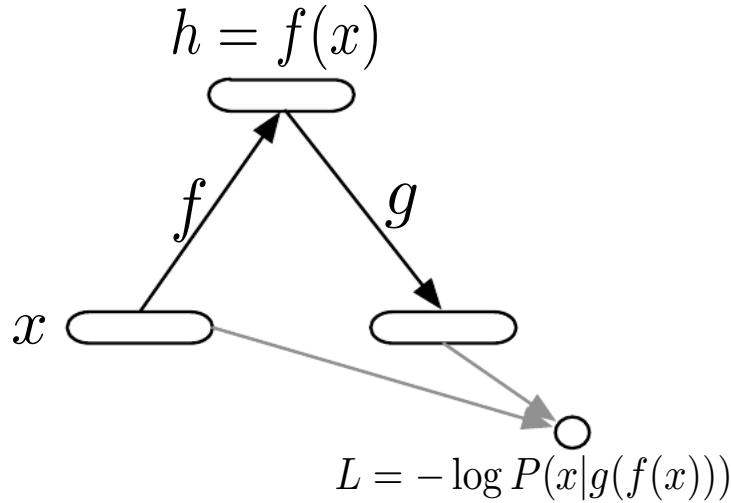


Figure 16.5: The computational graph of an auto-encoder, which is trained to maximize the probability assigned by the decoder g to the data point \mathbf{x} , given the output of the encoder $\mathbf{h} = f(\mathbf{x})$. The training objective is thus $L = -\log P(\mathbf{x}|g(f(\mathbf{x})))$, which ends up being squared reconstruction error if we choose a Gaussian reconstruction distribution with mean $g(f(\mathbf{x}))$, and cross-entropy if we choose a factorized Bernoulli reconstruction distribution with means $g(f(\mathbf{x}))$.

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 $P(\mathbf{x}|\mathbf{h})$ being Normal. If the input is a vector of bits, then cross-entropy is a more reasonable choice, and corresponds to $P(\mathbf{x}|\mathbf{h}) = \prod_i P(x_i|\mathbf{h})$ with $x_i|\mathbf{h}$ being Bernoulli-distributed. We then view the decoder $g(\mathbf{h})$ as computing the *parameters* of the reconstruction distribution, i.e., $P(\mathbf{x}|\mathbf{h}) = P(\mathbf{x}|g(\mathbf{h}))$.

Another advantage of this view is that we can think about the training of the decoder as estimating the conditional distribution $P(\mathbf{x}|\mathbf{h})$, which comes handy in the probabilistic interpretation of denoising auto-encoders, allowing us to talk about the distribution $P(\mathbf{x})$ explicitly or implicitly represented by the auto-encoder (see Sections 16.8, 21.8.2 and 21.9 for more details). In the same spirit, we can rethink the notion of encoder from a simple function to a conditional distribution $Q(\mathbf{h}|\mathbf{x})$, with a special case being when $Q(\mathbf{h}|\mathbf{x})$ 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 21.8.2 and 21.10.

16.7 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 16.5.2 as a linear factor model in which the prior $P(\mathbf{h})$ on the representation $\mathbf{h} = f(\mathbf{x})$ encourages values at or near 0. In Section 16.7.1, 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

$$h^*(\mathbf{x}) = \arg \max_{\mathbf{h}} \log P(\mathbf{h}|\mathbf{x}) = \arg \min_{\mathbf{h}} \frac{\|\mathbf{x} - (\mathbf{b} + \mathbf{W}\mathbf{h})\|^2}{\sigma^2} - \log P(\mathbf{h}) \quad (16.7)$$

where σ^2 is a reconstruction variance parameter (which should equal the average squared reconstruction error⁶), and $P(\mathbf{h})$ is a “sparse” prior that puts more probability mass around $\mathbf{h} = 0$, such as the Laplacian prior, with factorized marginals

$$P(h_i) = \frac{\lambda}{2} e^{\lambda|h_i|} \quad (16.8)$$

or the Student-t prior, with factorized marginals

$$P(h_i) \propto \frac{1}{(1 + \frac{h_i^2}{\nu})^{\frac{\nu+1}{2}}}. \quad (16.9)$$

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 14.8), 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 \mathbf{x} , i.e., performing inference (computing the representation \mathbf{h} that goes with the given \mathbf{x}) can be substantially larger than with a parametric encoder (because an optimization must be performed *for each example \mathbf{x}*), and
2. the resulting encoder function could be non-smooth and possibly too non-linear (with two nearby \mathbf{x} ’s being associated with very different \mathbf{h} ’s), potentially making it more difficult for the downstream layers to properly generalize.

⁶but can be lumped into the regularizer λ which controls the strength of the sparsity prior, defined in Eq. 16.8, for example.

In Section 16.7.2, 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 16.8 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 16.9 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.

16.7.1 Sparse Auto-Encoders

A sparse auto-encoder is simply an auto-encoder whose training criterion involves a sparsity penalty $\Omega(\mathbf{h})$ in addition to the reconstruction error:

$$L = -\log P(\mathbf{x}|g(\mathbf{h})) + \Omega(\mathbf{h}) \quad (16.10)$$

where $g(\mathbf{h})$ is the decoder output and typically we have $\mathbf{h} = f(\mathbf{x})$, the encoder output.

We can think of that penalty $\Omega(\mathbf{h})$ simply as a regularizer or as a log-prior on the representations \mathbf{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. 16.8 above):

$$\begin{aligned} \Omega(\mathbf{h}) &= \lambda \sum_i |h_i| \\ -\log P(\mathbf{h}) &= \sum_i \log \frac{\lambda}{2} + \lambda |h_i| = \text{const} + \Omega(\mathbf{h}) \end{aligned} \quad (16.11)$$

where the constant term depends only of λ and not \mathbf{h} (which we typically ignore in the training criterion because we consider λ as a hyper-parameter rather than a parameter). Similarly (as per Eq. 16.9), the sparsity penalty corresponding to the Student-t prior (Olshausen and Field, 1997) is

$$\Omega(\mathbf{h}) = \sum_i \frac{\nu+1}{2} \log\left(1 + \frac{h_i^2}{\nu}\right) \quad (16.12)$$

where ν is considered to be a hyper-parameter.

The early work on sparse auto-encoders (Ranzato *et al.*, 2007, 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 $\rho = 0.05$.

$$\Omega(\mathbf{h}) = \sum_i \rho \log h_i + (1 - \rho) \log(1 - h_i) \quad (16.13)$$

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 \mathbf{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 (2010) 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 \mathbf{x} and parameters $\boldsymbol{\theta}$ (see Section 5.7):

$$\arg \max_{\boldsymbol{\theta}} P(\boldsymbol{\theta} | \mathbf{x}) = \arg \max_{\boldsymbol{\theta}} (\log P(\mathbf{x} | \boldsymbol{\theta}) + \log P(\boldsymbol{\theta}))$$

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.

16.7.2 Predictive Sparse Decomposition

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, 2010; Jarrett *et al.*, 2009b; 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 \mathbf{h} (after inference) to be close to the output of the encoder $f(\mathbf{x})$:

$$L = \arg \min_{\mathbf{h}} (||\mathbf{x} - g(\mathbf{h})||^2 + \lambda |\mathbf{h}|_1 + \gamma ||\mathbf{h} - f(\mathbf{x})||^2) \quad (16.14)$$

where f is the encoder and g is the decoder. Like in sparse coding, for each example \mathbf{x} an iterative optimization is performed in order to obtain a representation \mathbf{h} . However, because the iterations can be initialized from the output of the encoder, i.e., with $\mathbf{h} = f(\mathbf{x})$, only a few steps (e.g. 10) are necessary to obtain good results. Simple gradient descent on \mathbf{h} has been used by the authors. After \mathbf{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. 20.6). 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. 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 21.8.2). See also Section 21.8.3 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.

16.8 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 16.6: the encoder sees as input a corrupted version of the input, but the decoder tries to reconstruct the clean uncorrupted input.

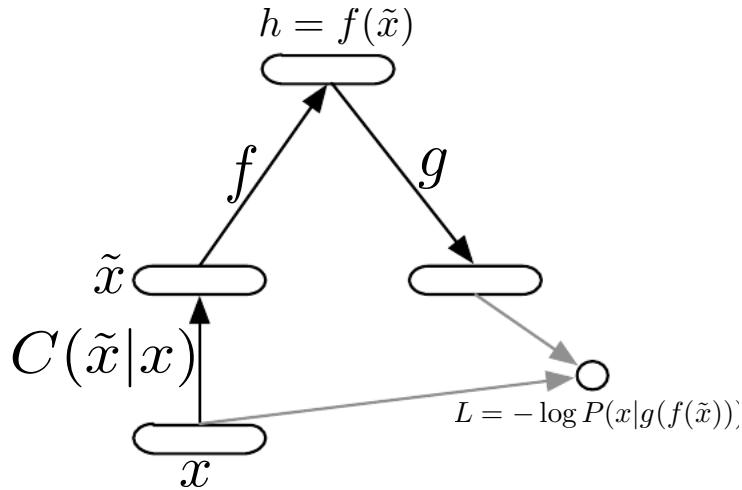


Figure 16.6: The computational graph of a denoising auto-encoder, which is trained to reconstruct the clean data point \mathbf{x} from its corrupted version $\tilde{\mathbf{x}}$, i.e., to minimize the loss $L = - \log P(\mathbf{x}|g(f(\tilde{\mathbf{x}})))$, where $\tilde{\mathbf{x}}$ is a corrupted version of the data example \mathbf{x} , obtained through a given corruption process $C(\tilde{\mathbf{x}}|\mathbf{x})$.

Mathematically, and following the notations used in this chapter, this can be formalized as follows. We introduce a corruption process $C(\tilde{\mathbf{x}}|\mathbf{x})$ 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* $P(\mathbf{x}|\tilde{\mathbf{x}})$ estimated from training pairs $(\mathbf{x}, \tilde{\mathbf{x}})$, as follows:

1. Sample a training example $\mathbf{x} = \mathbf{x}$ from the data generating distribution (the training set).
2. Sample a corrupted version $\tilde{\mathbf{x}} = \tilde{\mathbf{x}}$ from the conditional distribution $C(\tilde{\mathbf{x}}|\mathbf{x} = \mathbf{x})$.
3. Use $(\mathbf{x}, \tilde{\mathbf{x}})$ as a training example for estimating the auto-encoder reconstruction distribution $P(\mathbf{x}|\tilde{\mathbf{x}}) = P(\mathbf{x}|g(\mathbf{h}))$ with \mathbf{h} the output of encoder $f(\tilde{\mathbf{x}})$ and $g(\mathbf{h})$ the output of the decoder.

Typically we can simply perform gradient-based approximate minimization (such as minibatch gradient descent) on the negative log-likelihood $-\log P(\mathbf{x}|\mathbf{h})$, 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 \mathbf{x} from the data set, and
2. the random corruption applied to \mathbf{x} to obtain $\tilde{\mathbf{x}}$.

We can therefore consider that the DAE is performing stochastic gradient descent on the following expectation:

$$-E_{\mathbf{x} \sim Q(\mathbf{x})} E_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}}|\mathbf{x})} \log P(\mathbf{x}|g(f(\tilde{\mathbf{x}})))$$

where $Q(\mathbf{x})$ is the training distribution.

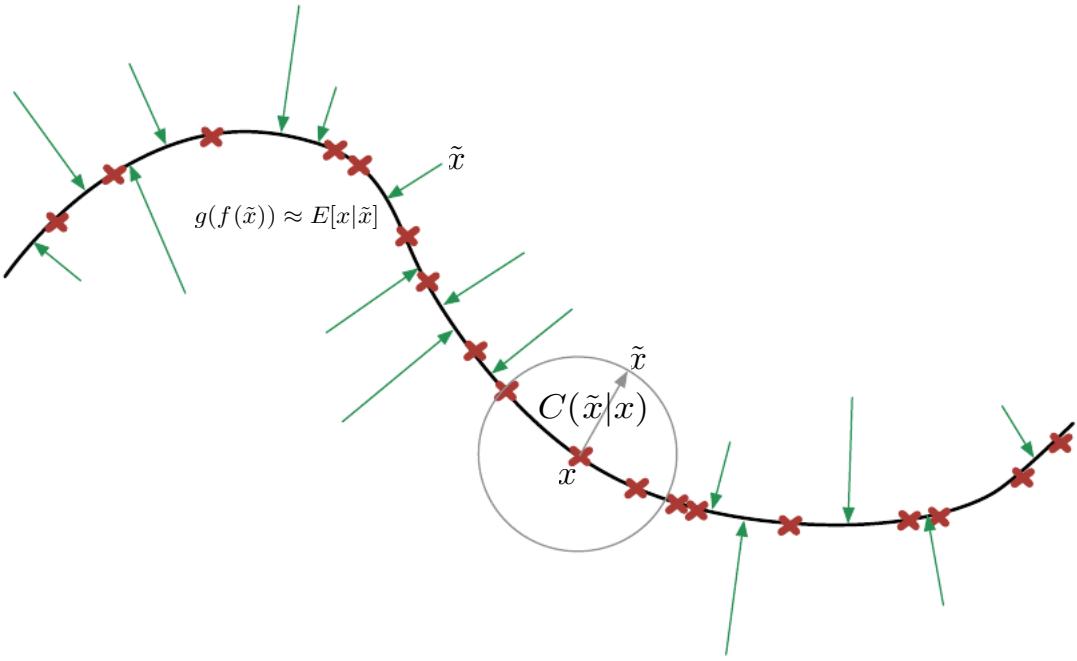


Figure 16.7: A denoising auto-encoder is trained to reconstruct the clean data point \mathbf{x} from its corrupted version $\tilde{\mathbf{x}}$. In the figure, we illustrate the corruption process $C(\tilde{\mathbf{x}}|\mathbf{x})$ by a grey circle of equiprobable corruptions, and grey arrow for the corruption process) acting on examples \mathbf{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 $\|g(f(\tilde{\mathbf{x}})) - \mathbf{x}\|^2$, the reconstruction $g(f(\tilde{\mathbf{x}}))$ estimates $E[\mathbf{x}|\tilde{\mathbf{x}}]$, which approximately points orthogonally towards the manifold, since it estimates the center of mass of the clean points \mathbf{x} which could have given rise to $\tilde{\mathbf{x}}$. The auto-encoder thus learns a vector field $g(f(\tilde{\mathbf{x}})) - \mathbf{x}$ (the green arrows) and it turns out that this vector field estimates the gradient field $\frac{\partial \log Q(\mathbf{x})}{\partial \mathbf{x}}$ (up to a multiplicative factor that is the average root mean square reconstruction error), where Q is the unknown data generating distribution.

16.8.1 Learning a Vector Field that Estimates a Gradient Field

As illustrated in Figure 16.7, a very important property of DAEs is that their training criterion makes the auto-encoder learn a vector field $(g(f(\mathbf{x})) - \mathbf{x})$ that estimates the gradient field (or *score*) $\frac{\partial \log Q(\mathbf{x})}{\partial \mathbf{x}}$, as per Eq. 16.15. 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 19.4. The denoising version is discussed in Section 19.5.

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 \mathbf{x} is continuous-valued), i.e., with the squared error denoising error

$$\|g(f(\tilde{\mathbf{x}})) - \mathbf{x}\|^2$$

and corruption

$$C(\tilde{\mathbf{x}} = \tilde{\mathbf{x}}|\mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}}; \mu = \mathbf{x}, \Sigma = \sigma^2 \mathbf{I})$$

with noise variance σ^2 .

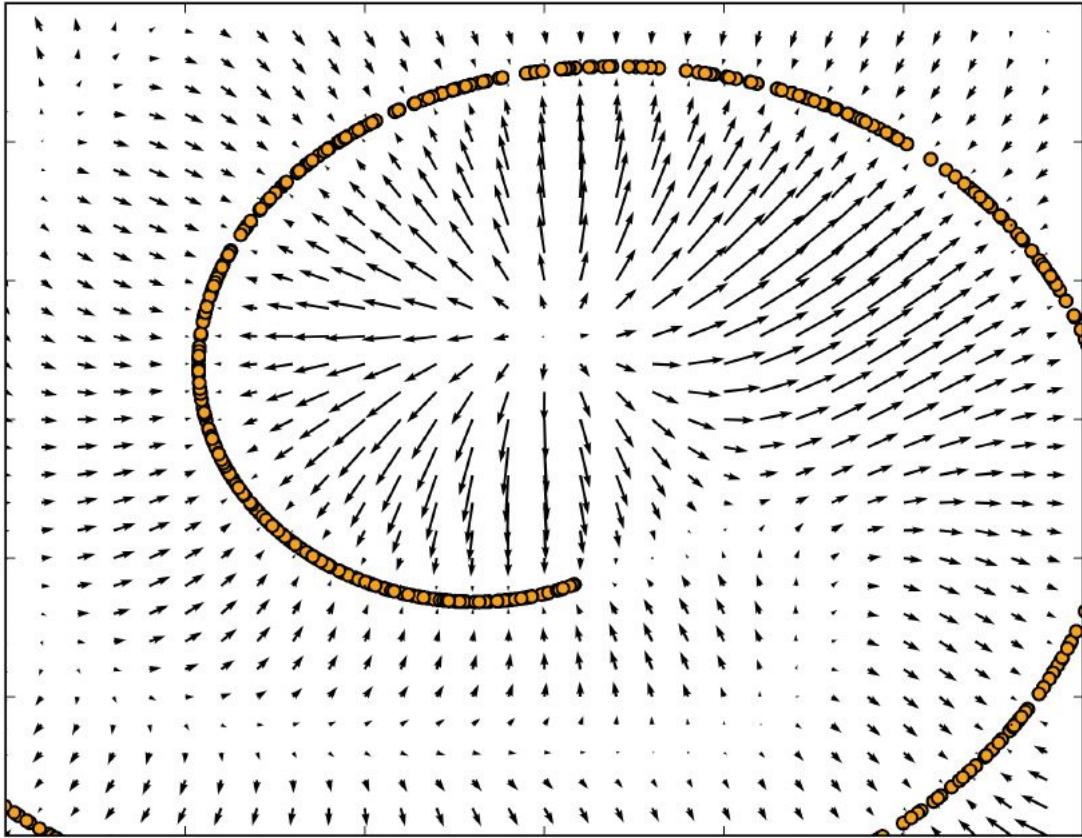


Figure 16.8: Vector field learned by a denoising auto-encoder around a 1-D curved manifold near which the data (orange circles) concentrates in a 2-D space. Each arrow is proportional to the reconstruction minus input vector of the auto-encoder and points towards higher probability according to the implicitly 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 $\frac{g(f(\mathbf{x})) - \mathbf{x}}{\sigma^2}$ is a consistent estimator of $\frac{\partial \log Q(\mathbf{x})}{\partial \mathbf{x}}$, where $Q(\mathbf{x})$ is the data generating distribution,

$$\frac{g(f(\mathbf{x})) - \mathbf{x}}{\sigma^2} \rightarrow \frac{\partial \log Q(\mathbf{x})}{\partial \mathbf{x}}, \quad (16.15)$$

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 $g(f(\mathbf{x}))$ minus the input \mathbf{x} corresponds to the gradient of something (the estimated score should be the gradient of the estimated log-density with respect to the input \mathbf{x}). That is why

the early results (Vincent, 2011a) are specialized to particular parametrizations where $g(f(\mathbf{x})) - \mathbf{x}$ 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 16.8 (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 21.9 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 21.9, provides a probabilistic – and in fact generative – interpretation to every denoising auto-encoder.

16.9 Contractive Auto-Encoders

The Contractive Auto-Encoder or CAE (Rifai *et al.*, 2011a,c) introduces an explicit regularizer on the code $\mathbf{h} = f(\mathbf{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\mathbf{h}) = \left\| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right\|_F^2 \quad (16.16)$$

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 18.13 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 \mathbf{h} keep all the information necessary to *reconstruct training examples*, the CAE penalty would yield a code \mathbf{h} that is constant and does not depend on the input \mathbf{x} . The compromise between these two forces yields an auto-encoder whose derivatives $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ 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., $f(\mathbf{x})$ must vary as \mathbf{x} moves from one to the other, i.e., in the direction of a tangent to the manifold.

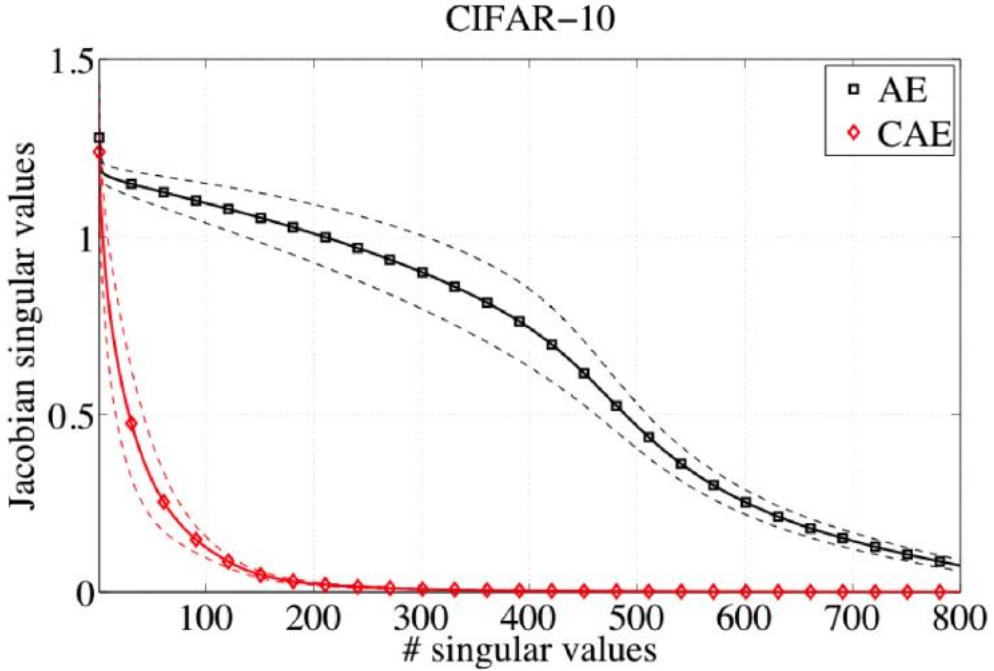


Figure 16.9: Average (over test examples) of the singular value spectrum of the Jacobian matrix $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ for the encoder f learned by a regular auto-encoder (AE) versus a contractive auto-encoder (CAE). This illustrates how the contractive regularizer yields a smaller set of directions in input space (those corresponding to large singular value of the Jacobian) which provoke a response in the representation \mathbf{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 $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ for different auto-encoders, as shown in Figure 16.9. We see that the CAE manages to concentrate the sensitivity of the representation in fewer dimensions than a regular (or sparse) auto-encoder. Figure 18.3 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 16.10 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).

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

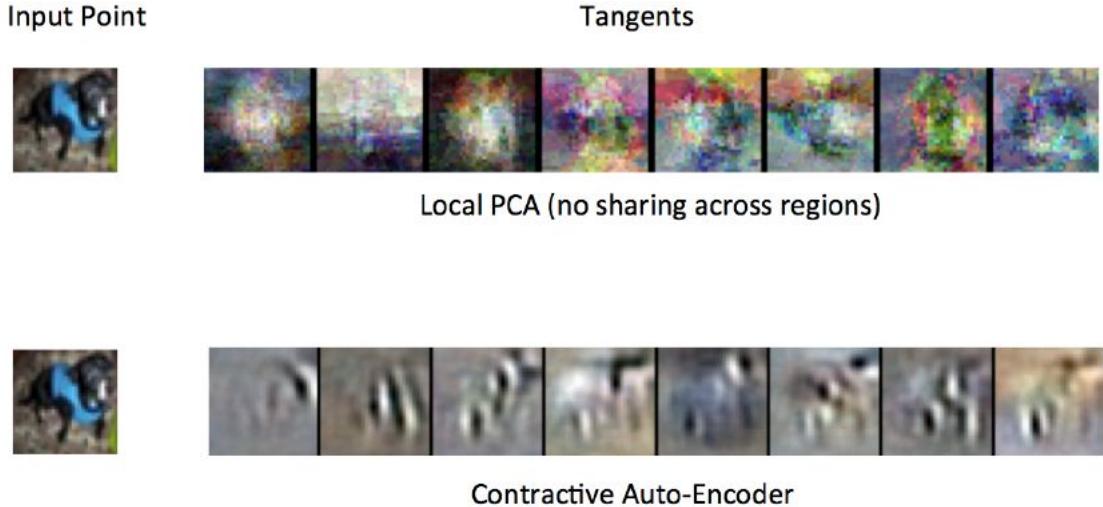


Figure 16.10: Illustration of tangent vectors (bottom) of the manifold estimated by a contractive auto-encoder (CAE), at some input point (left, CIFAR-10 image of a dog). See also Fig. 18.3. Each image on the right corresponds to a tangent vector, either estimated by a local PCA (equivalent to a Gaussian mixture), top, or by a CAE (bottom). The tangent vectors are estimated by the leading singular vectors of the Jacobian matrix $\frac{\partial h}{\partial x}$ of the input-to-code mapping. 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 meaningfully 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.

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 17

Representation Learning

What is a good representation? Many answers are possible, and this remains a question to be further explored in future research. What we propose as answer in this book is that in general, a good representation is one that makes further learning tasks easy. In an unsupervised learning setting, this could mean that the joint distribution of the different elements of the representation (e.g., elements of the representation vector \mathbf{h}) is one that is easy to model (e.g., in the extreme, these elements are marginally independent of each other). But that would not be enough: a representation that throws away all information (e.g., $\mathbf{h} = 0$ for all inputs \mathbf{x}) is very easy to model but is also useless. Hence we want to learn a representation that keeps the information (or at least all the relevant information, in the supervised case) and makes it easy to learn functions of interest from this representation.

In Chapter 1, we have introduced the notion of *representation*, the idea that some representations were more helpful (e.g. to classify objects from images or phonemes from speech) than others. As argued there, this suggests *learning representations* in order to “select” the best ones in a systematic way, i.e., by optimizing a function that maps raw data to its representation, instead of - or in addition to - handcrafting them. This motivation for learning input features is discussed in Section 6.5, and is one of the major side-effects of training a feedforward deep network (treated in Chapter 6), typically via supervised learning, i.e., when one has access to (input,target) pairs¹, available for some task of interest. In the case of supervised learning of deep nets, we learn a representation with the objective of selecting one that is best suited to the task of predicting targets given inputs.

Whereas supervised learning has been the workhorse of recent industrial successes of deep learning, the authors of this book believe that it is likely that a key element of future advances will be *unsupervised learning of representations*.

So how can we exploit the information in data if we don’t have labeled examples? or too few? Pure supervised learning with few labeled examples can easily overfit. On the other hand, humans (and other animals) can sometimes learn a task from just one

¹typically obtained by *labeling* inputs with some target answer that we wish the computer would produce

or very few examples. How is that possible? Clearly they must rely on previously acquired knowledge, either innate or (more likely the case for humans) via previous learning experience. Can we discover good representations purely out of unlabeled examples? (this is treated in the first four sections of this chapter). Can we combine unlabeled examples (which are often easy to obtain) with labeled examples? (this is semi-supervised learning, Section 17.3). And what if instead of one task we have many tasks that could share the same representation or parts of it? (this is multi-task learning, discussed in Section 7.12). What if we have “training tasks” (on which enough labeled examples are available) as well as “test tasks” (not known at the time of learning the representation, and for which only very few labeled examples will be provided)? What if the test task is similar but different from the training task? (this is transfer learning and domain adaptation, discussed in Section 17.2).

17.1 Greedy Layerwise Unsupervised Pre-Training

Unsupervised learning played a key historical role in the revival of deep neural networks, allowing for the first time to train a deep supervised network. We call this procedure *unsupervised pre-training*, or more precisely, *greedy layer-wise unsupervised pre-training*, and it is the topic of this section.

This recipe relies on a one-layer representation learning algorithm such as those introduced in this part of the book, i.e., the auto-encoders (Chapter 16) and the RBM (Section 21.1). Each layer is pre-trained by unsupervised learning, taking the output of the previous layer and producing as output a new representation of the data, whose distribution (or its relation to other variables such as categories to predict) is hopefully simpler.

Greedy layerwise unsupervised pre-training was introduced in Hinton *et al.* (2006); Hinton and Salakhutdinov (2006); Bengio *et al.* (2007); Ranzato *et al.* (2007). These papers are generally credited with founding the renewed interest in learning deep models as it provided a means of initializing subsequent supervised training and often led to notable performance gains when compared to models trained without unsupervised pretraining, at least for the small kinds of datasets (like the 60,000 examples of MNIST) used in these experiments.

It is called *layerwise* because it proceeds one layer at a time, training the k -th layer while keeping the previous ones fixed. It is called *unsupervised* because each layer is trained with an unsupervised representation learning algorithm. It is called *greedy* because the different layers are not jointly trained with respect to a global training objective, which could make the procedure sub-optimal. In particular, the lower layers (which are first trained) are not adapted after the upper layers are introduced. However it is also called *pre-training*, because it is supposed to be only a first step before a joint training algorithm is applied to *fine-tune* all the layers together with respect to a criterion of interest. In the context of a supervised learning task, it can be viewed as a regularizer (see Chapter 7) and a sophisticated form of parameter initialization.

When we refer to pre-training we will be referring to a specific protocol with two

main phases of training: the pretraining phase and the fine-tuning phase. No matter what kind of unsupervised learning algorithm or what model type you employ, in the vast majority of cases, the overall training scheme is nearly the same. While the choice of unsupervised learning algorithm will obviously impact the details, in the abstract, most applications of unsupervised pre-training follows this basic protocol.

As outlined in Algorithm 17.1, in the *pretraining phase*, the layers of the model are trained, in order, in an unsupervised way on their input, beginning with the bottom layer, i.e. the one in direct contact with the input data. Next, the second lowest layer is trained taking the activations of the first layer hidden units as input for unsupervised training. Pretraining proceeds in this fashion, from bottom to top, with each layer training on the “output” or activations of the hidden units of the layer below. After the last layer is pretrained, a supervised layer is put on top, and all the layers are jointly trained with respect to the overall supervised training criterion. In other words, the pre-training was only used to initialize a deep supervised neural network (which could be a convolutional neural network (Ranzato *et al.*, 2007)). This is illustrated in Figure 17.1.

Algorithm 17.1 Greedy layer-wise unsupervised pre-training protocol.

Given the following: Unsupervised feature learner \mathcal{L} , which takes a training set \mathcal{D} of examples and returns an encoder or feature function $f = \mathcal{L}(\mathcal{D})$. The raw input data is \mathbf{X} , with one row per example and $f(\mathbf{X})$ is the dataset used by the second level unsupervised feature learner. In the case fine-tuning is performed, we use a learner \mathcal{T} which takes an initial function f , input examples \mathbf{X} (and in the supervised fine-tuning case, associated targets \mathbf{Y}), and returns a tuned function. The number of stages is M .

```

 $\mathcal{D}^{(0)} = \mathbf{X}$ 
 $f \leftarrow$  Identity function
for  $k = 1 \dots, M$  do
     $f^{(k)} = \mathcal{L}(\mathcal{D})$ 
     $f \leftarrow f^{(k)} \circ f$ 
end for
if fine-tuning then
     $f \leftarrow \mathcal{T}(f, \mathbf{X}, \mathbf{Y})$ 
end if
Return  $f$ 
```

However, greedy layerwise unsupervised pre-training can also be used as initialization for other unsupervised learning algorithms, such as deep auto-encoders (Hinton and Salakhutdinov, 2006), deep belief networks (Hinton *et al.*, 2006) (Section 21.3), or deep Boltzmann machines (Salakhutdinov and Hinton, 2009a) (Section 21.4).

As discussed in Section 8.6.4, it is also possible to have greedy layerwise *supervised* pre-training, to help *optimize* deep supervised networks. This builds on the premise that training a shallow network is easier than training a deep one, which seems to have been validated in several contexts (Erhan *et al.*, 2010).

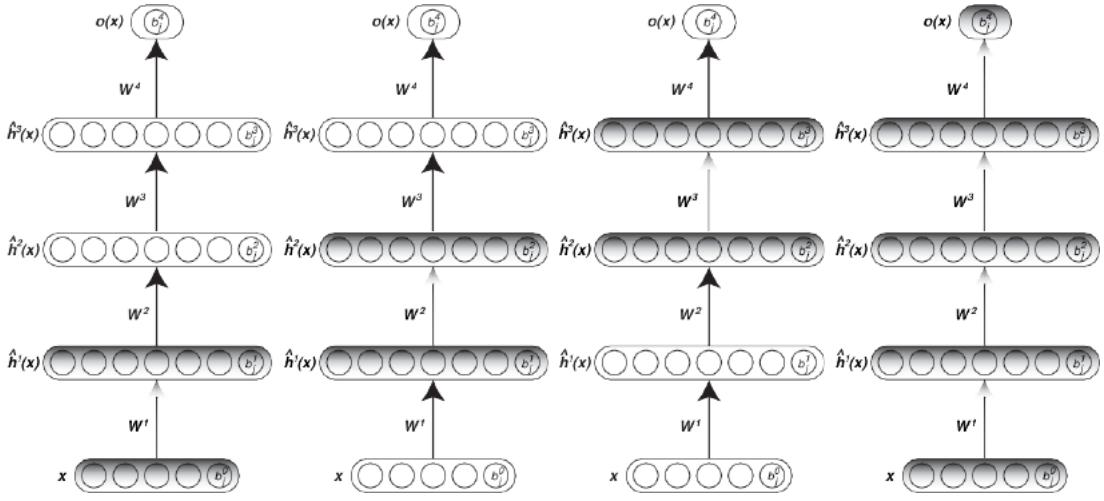


Figure 17.1: Illustration of the greedy layer-wise unsupervised pre-training scheme, in the case of a network with 3 hidden layers. The protocol proceeds in 4 phases (one per hidden layer, plus the final supervised fine-tuning phase), from left to right. For the unsupervised steps, each layer (darker grey) is trained to learn a better representation of the output of the previously trained layer (initially, the raw input). These representations learned by unsupervised learning form the initialization of a deep supervised net, which is then trained (fine-tuned) as usual (last phase, right), with all parameters being free to change (darker grey).

17.1.1 Why Does Unsupervised Pre-Training Work?

What has been observed on several datasets starting in 2006 ([Hinton et al., 2006](#); [Bengio et al., 2007](#); [Ranzato et al., 2007](#)) is that greedy layer-wise unsupervised pre-training can yield substantial improvements in test error for classification tasks. Later work suggested that the improvements were less marked (or not even visible) when very large labeled datasets are available, although the boundary between the two behaviors remains to be clarified, i.e., it may not just be an issue of number of labeled examples but also how this relates to the complexity of the function to be learned.

A question that thus naturally arises is the following: why and when does unsupervised pre-training work? Although previous studies have mostly focused on the case when the final task is supervised (with supervised fine-tuning), it is also interesting to keep in mind that one gets improvements in terms of both training and test performance in the case of unsupervised fine-tuning, e.g., when training deep auto-encoders ([Hinton and Salakhutdinov, 2006](#)).

This “why does it work” question is at the center of the paper by [Erhan et al. \(2010\)](#), and their experiments focused on the supervised fine-tuning case. They consider different machine learning hypotheses to explain the results observed, and attempted to confirm those via experiments. We summarize some of this investigation here.

First of all, they studied the *trajectories* of neural networks during supervised fine-tuning, and evaluated how different they were depending on initial conditions, i.e., due to random initialization or due to performing unsupervised pre-training or not. The main result is illustrated and discussed in Figures 17.2 and 17.2. Note that it would not make sense to plot the evolution of parameters of these networks directly, because the same input-to-output function can be represented by different parameter values (e.g., by relabeling the hidden units). Instead, this work plots the trajectories in *function space*, by considering the output of a network (the class probability predictions) for a given set of test examples as a proxy for the function computed. By concatenating all these outputs (over say 1000 examples) and doing dimensionality reduction on these vectors, we obtain the kinds of plots illustrated in the figure.

The main conclusions of these kinds of plots are the following:

1. Each training trajectory goes to a different place, i.e., different trajectories do not converge to the same place. These “places” might be in the vicinity of a local minimum or as we understand it better now ([Dauphin et al., 2014](#)) these are more likely to be an “apparent local minimum” in the region of flat derivatives near a saddle point. This suggests that the number of these apparent local minima is huge, and this also is in agreement with theory ([Dauphin et al., 2014](#); [Choromanska et al., 2014](#)).
2. Depending on whether we initialize with unsupervised pre-training or not, very different functions (in function space) are obtained, covering regions that do not overlap. Hence there is a qualitative effect due to unsupervised pre-training.
3. With unsupervised pre-training, the region of space covered by the solutions asso-

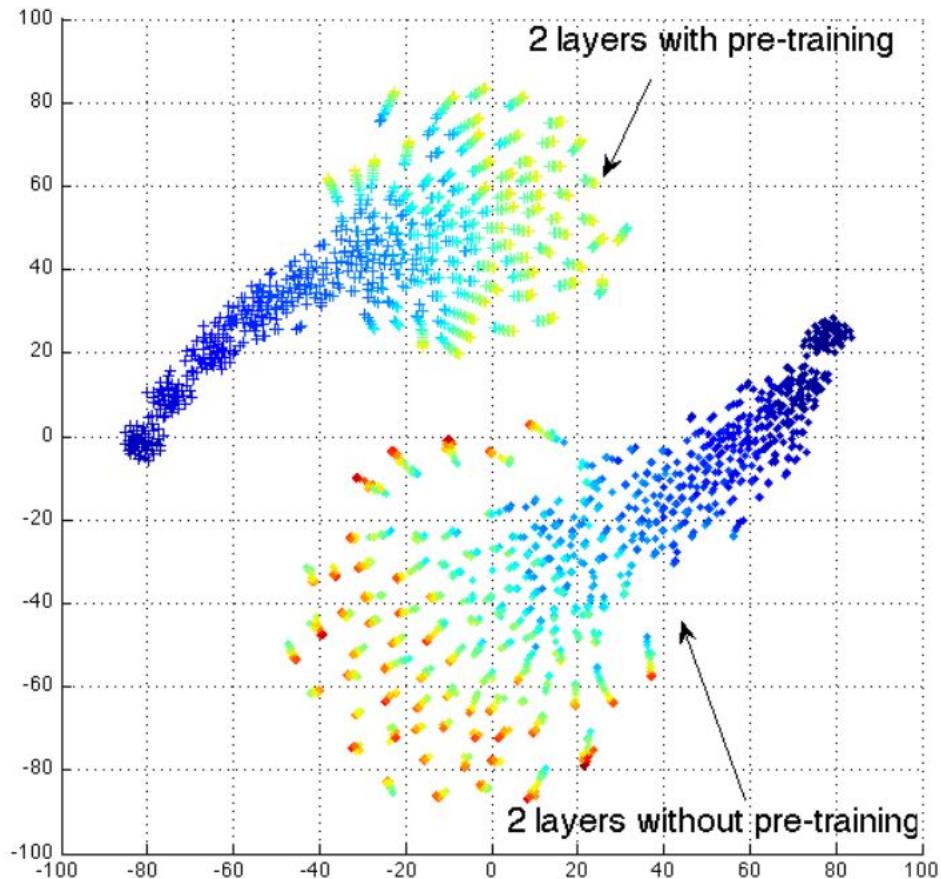


Figure 17.2: Illustrations of the trajectories of different neural networks in *function space* (not parameter space, to avoid the issue of many-to-one mapping from parameter vector to function), with different random initializations and with or without unsupervised pre-training. Each plus or diamond point corresponds to a different neural network, at a particular time during its training trajectory, with the function it computes projected to 2-D by t-SNE (van der Maaten and Hinton, 2008a) (this figure) or by Isomap (Tenenbaum *et al.*, 2000) (Figure 17.3). Color indicates the number of training epochs. What we see is that no two networks converge to the same function (so a large number of *apparent* local minima seems to exist), and that networks initialized with pre-training learn very different functions, in a region of function space that does not overlap at all with those learned by networks without pre-training. Such curves were introduced by Erhan *et al.* (2010) and are reproduced here with permission.

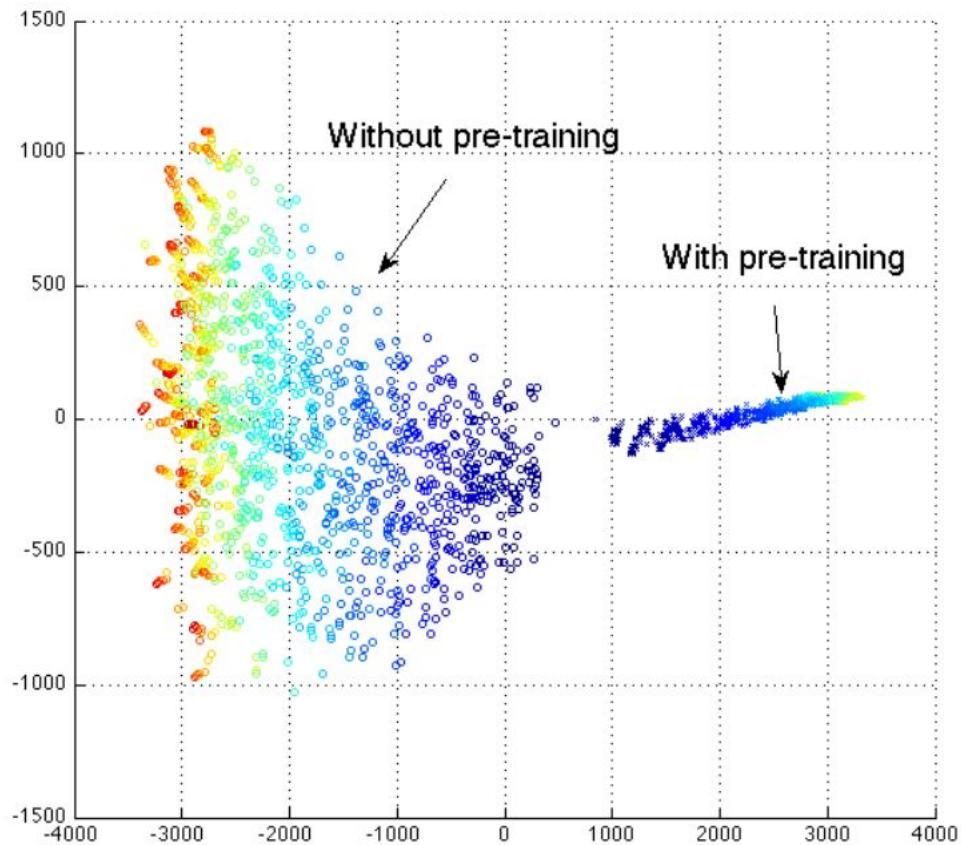


Figure 17.3: See Figure 17.2's caption. This figure only differs in the use of Isomap (Tenenbaum *et al.*, 2000) rather than t-SNE (van der Maaten and Hinton, 2008b) for dimensionality reduction. Note that Isomap tries to preserve global relative distances (and hence volumes), whereas t-SNE only cares about preserving local geometry and neighborhood relationships. We see with the Isomap dimensionality reduction that the volume in function space occupied by the networks with pre-training is much smaller (in fact that volume gets reduced rather than increased, during training), suggesting that the set of solutions enjoy smaller variance, which would be consistent with the observed improvements in generalization error. Such curves were introduced by Erhan *et al.* (2010) and are reproduced here with permission.

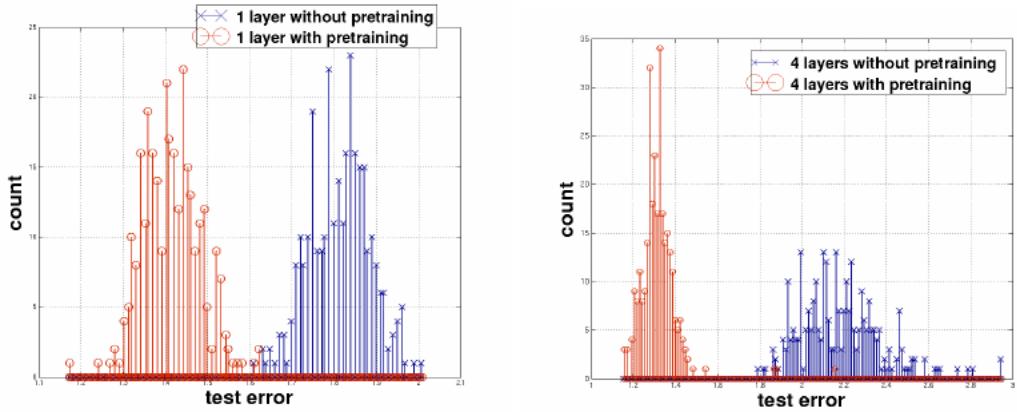


Figure 17.4: Histograms presenting the test errors obtained on MNIST using denoising auto-encoder models trained with or without pre-training (400 different initializations each). **Left:** 1 hidden layer. **Right:** 4 hidden layers. We see that the advantage brought by pre-training increases with depth, both in terms of mean error and in terms of the variance of the error (w.r.t. random initialization).

ciated with different initializations *shrinks* as we consider more training iterations, whereas it *grows* without unsupervised pre-training. This is only apparent in the visualization of Figure 17.3, which attempts to preserve volume. A larger region is bad for generalization (because not all these functions can be the right one together), yielding higher variance. This is consistent with the better generalization observed with unsupervised pre-training.

Another interesting effect is that the advantage of pre-training seems to increase with depth, as illustrated in Figure 17.4, with both the mean and the variance of the error decreasing more for deeper networks.

An important question is whether the advantage brought by pre-training can be seen as a form of regularizer (which could help test error but hurt training error) or simply a way to find a better minimizer of training error (e.g., by initializing near a better minimum of training error). The experiments suggest pre-training actually acts as a regularizer, i.e., hurting training error at least in some cases (with deeper networks). So if it also helps optimization, it is only because it initializes closer to a good solution from the point of view of generalization, not necessarily from the point of view of the training set.

How could unsupervised pre-training act as regularizer? Simply by imposing an extra constraint: the learned representations should not only be consistent with better predicting outputs \mathbf{y} but they should also be consistent with better capturing the variations in the input \mathbf{x} , i.e., modeling $P(\mathbf{x})$. This is associated implicitly with a prior, i.e., that $P(\mathbf{y}|\mathbf{x})$ and $P(\mathbf{x})$ share structure, i.e., that learning about $P(\mathbf{x})$ can help to generalize better on $P(\mathbf{y}|\mathbf{x})$. Obviously this needs not be the case in general, e.g., if

\mathbf{y} is an effect of \mathbf{x} . However, if \mathbf{y} is a cause of \mathbf{x} , then we would expect this a priori assumption to be correct, as discussed at greater length in Section 17.4 in the context of semi-supervised learning.

A disadvantage of unsupervised pre-training is that it is difficult to choose the capacity hyper-parameters (such as when to stop training) for the pre-training phases. An expensive option is to try many different values of these hyper-parameters and choose the one which gives the best supervised learning error after fine-tuning. Another potential disadvantage is that unsupervised pre-training may require larger representations than what would be necessarily strictly for the task at hand, since presumably, \mathbf{y} is only one of the factors that explain \mathbf{x} .

Today, as many deep learning researchers and practitioners have moved to working with very large labeled datasets, unsupervised pre-training has become less popular in favor of other forms of regularization such as dropout – to be discussed in section 7.11. Nevertheless, unsupervised pre-training remains an important tool in the deep learning toolbox and should particularly be considered when the number of labeled examples is low, such as in the semi-supervised, domain adaptation and transfer learning settings, discussed next.

17.2 Transfer Learning and Domain Adaptation

Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P_1) is exploited to improve generalization in another setting (say distribution P_2).

In the case of *transfer learning*, we consider that the task is different but many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2 . This is typically understood in a supervised learning context, where the input is the same but the target may be of a different nature, e.g., learn about visual categories that are different in the first and the second setting. If there is a lot more data in the first setting (sampled from P_1), then that may help to learn representations that are useful to quickly generalize when examples of P_2 are drawn. For example, many visual categories *share* low-level notions of edges and visual shapes, the effects of geometric changes, changes in lighting, etc. In general, transfer learning, multi-task learning (Section 7.12), and domain adaptation can be achieved via representation learning when there exist features that would be useful for the different settings or tasks, i.e., there are *shared underlying factors*. This is illustrated in Figure 7.6, with shared lower layers and task-dependent upper layers.

However, sometimes, what is shared among the different tasks is not the semantics of the input but the semantics of the output, or maybe the input needs to be treated differently (e.g., consider user adaptation or speaker adaptation). In that case, it makes more sense to share the upper layers (near the output) of the neural network, and have a task-specific pre-processing, as illustrated in Figure 17.5.

In the related case of *domain adaptation*, we consider that the task (and the optimal input-to-output mapping) is the same but the input distribution is slightly different.

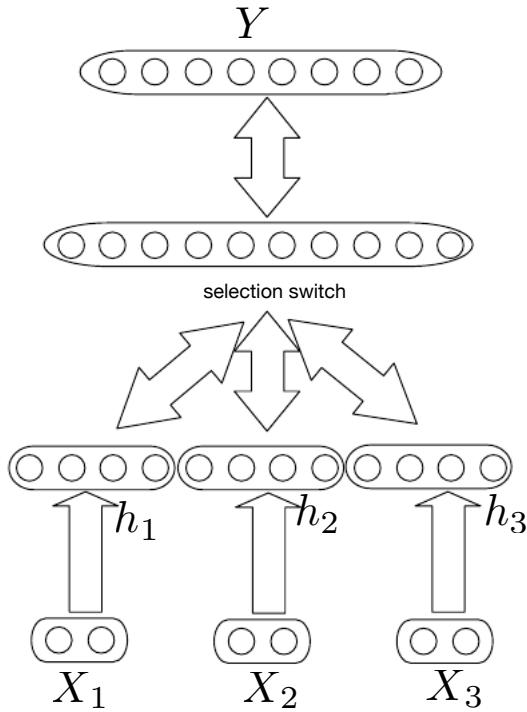


Figure 17.5: Example of architecture for multi-task or transfer learning when the output variable Y has the same semantics for all tasks while the input variable X has a different meaning (and possibly even a different dimension) for each task (or, for example, each user), called X_1 , X_2 and X_3 for three tasks in the figure. The lower levels (up to the selection switch) are task-specific, while the upper levels are shared. The lower levels learn to translate their task-specific input into a generic set of features.

For example, if we predict sentiment (positive or negative judgement) associated with textual comments posted on the web, the first setting may refer to consumer comments about books, videos and music, while the second setting may refer to televisions or other products. One can imagine that there is an underlying function that tells whether any statement is positive, neutral or negative, but of course the vocabulary, style, accent, may vary from one domain to another, making it more difficult to generalize across domains. Simple unsupervised pre-training (with denoising auto-encoders) has been found to be very successful for sentiment analysis with domain adaptation (Glorot *et al.*, 2011c).

A related problem is that of *concept drift*, which we can view as a form of transfer learning due to gradual changes in the data distribution over time. Both concept drift and transfer learning can be viewed as particular forms of multi-task learning (Section 7.12). Whereas multi-task learning is typically considered in the context of supervised learning, the more general notion of transfer learning is applicable for unsupervised learning and reinforcement learning as well. Figure 7.6 illustrates an architecture

in which different tasks share underlying features or factors, taken from a larger pool that explain the variations in the input.

In all of these cases, the objective is to take advantage of data from a first setting to extract information that may be useful when learning or even when directly making predictions in the second setting. One of the potential advantages of representation learning for such generalization challenges, and especially of deep representation learning, is that it may considerably help to generalize by extracting and disentangling a set of explanatory factors from data of the first setting, some of which may be relevant to the second setting. In the case of object recognition from an image, many of the factors of variation that explain visual categories in natural images remain the same when we move from one set of categories to another.

This discussion raises a very interesting and important question which is one of the core questions of this book: *what is a good representation?* Is it possible to learn representations that disentangle the underlying factors of variation? This theme is further explored at the end of this chapter (Section 17.4 and beyond). We claim that learning the most *abstract features* helps to maximize our chances of success in transfer learning, domain adaptation, or concept drift. More abstract features are more general and more likely to be close to the underlying causal factor, i.e., be relevant over many domains, many categories, and many time periods.

A good example of the success of unsupervised deep learning for transfer learning is its success in two competitions that took place in 2011, with results presented at ICML 2011 (and IJCNN 2011) in one case ([Mesnil et al., 2011](#)) (the Transfer Learning Challenge, <http://www.causality.inf.ethz.ch/unsupervised-learning.php>) and at NIPS 2011 ([Goodfellow et al., 2011](#)) in the other case (the Transfer Learning Challenge that was held as part of the NIPS'2011 workshop on Challenges in learning hierarchical models, <https://sites.google.com/site/nips2011workshop/transfer-learning-challenge>).

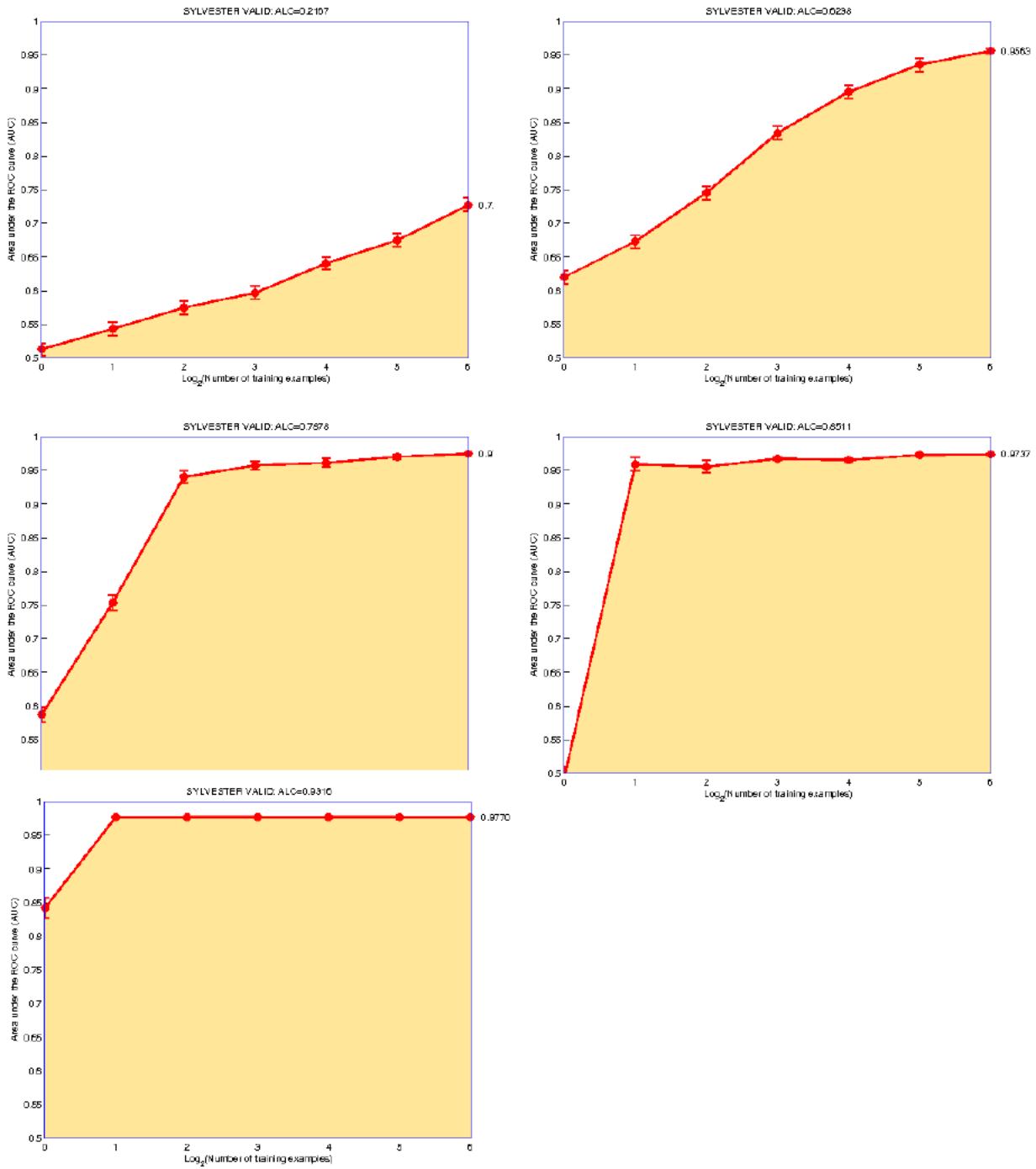


Figure 17.6: Results obtained on the Sylvester validation set (Transfer Learning Challenge). From left to right and top to bottom, respectively 0, 1, 2, 3, and 4 pre-trained layers. Horizontal axis is logarithm of number of labeled training examples on transfer setting (test task). Vertical axis is Area Under the Curve, which reflects classification accuracy. With deeper representations (learned unsupervised), the learning curves considerably improve, requiring fewer labeled examples to achieve the best generalization.

In the first of these competitions, the experimental setup is the following. Each participant is first given a dataset from the first setting (from distribution P_1), basically illustrating examples of some set of categories. The participants must use this to learn a good feature space (mapping the raw input to some representation), such that when we apply this learned transformation to inputs from the transfer setting (distribution P_2), a linear classifier can be trained and generalize well from very few labeled examples. Figure 17.6 illustrates one of the most striking results: as we consider deeper and deeper representations (learned in a purely unsupervised way from data of the first setting P_1), the learning curve on the new categories of the second (transfer) setting P_2 becomes much better, i.e., less labeled examples of the transfer tasks are necessary to achieve the apparently asymptotic generalization performance.

An extreme form of transfer learning is *one-shot learning* or even *zero-shot learning* or *zero-data learning*, where one or even zero example of the new task are given.

One-shot learning (Fei-Fei *et al.*, 2006) is possible because, in the learned representation, the new task corresponds to a very simple region, such as a ball-like region or the region around a corner of the space (in a high dimensional space, there are exponentially many corners). This works to the extent that the factors of variation corresponding to these invariances have been cleanly separated from other factors, in the learned representation space, and we have somehow learned which factors do and do not matter when discriminating objects of certain categories.

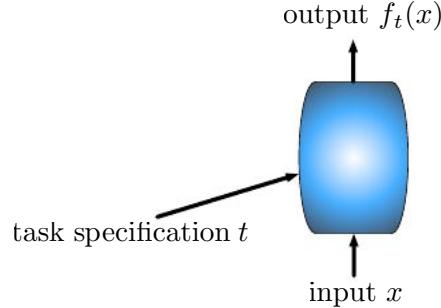


Figure 17.7: Figure illustrating how zero-data or zero-shot learning is possible. The trick is that the new context or task on which no example is given but on which we want a prediction is *represented* (with an input t), e.g., with a set of features, i.e., a distributed representation, and that representation is used by the predictor $f_t(x)$. If t was a one-hot vector for each task, then it would not be possible to generalize to a new task, but with a distributed representation the learner can benefit from the meaning of the individual task features (as they influence the relationship between inputs x and targets y , say), learned on other tasks for which examples are available.

Zero-data learning (Larochelle *et al.*, 2008) and zero-shot learning (Richard Socher and Ng, 2013) are only possible because additional information has been exploited during training that provides representations of the “task” or “context”, helping the learner figure out what is expected, even though no example of the new task has ever been seen. For example, in a multi-task learning setting, if each task is associated with a set of

features, i.e., a distributed representation (that is always provided as an extra input, in addition to the ordinary input associated with the task), then one can generalize to new tasks based on the similarity between the new task and the old tasks, as illustrated in Figure 17.7. One learns a parametrized function from inputs to outputs, parametrized by the task representation. In the case of zero-shot learning (Richard Socher and Ng, 2013), the “task” is a representation of a semantic object (such as a word), and its representation has already been learned from data relating different semantic objects together (such as natural language data, relating words together). On the other hand, for some of the tasks (e.g., words) one has data associating the variables of interest (e.g., words, pixels in images). Thus one can generalize and associate images to words for which no labeled images were previously shown to the learner. A similar phenomenon happens in machine translation (Klementiev *et al.*, 2012; Mikolov *et al.*, 2013; Gouws *et al.*, 2014): we have words in one language, and the relationships between words can be learned from unilingual corpora; on the other hand, we have translated sentences which relate words in one language with words in the other. Even though we may not have labeled examples translating word A in language X to word B in language Y, we can generalize and guess a translation for word A because we have learned a distributed representation for words in language X, a distributed representation for words in language Y, and created a link (possibly two-way) relating the two spaces, via translation examples. Note that this transfer will be most successful if all three ingredients (the two representations and the relations between them) are learned jointly.

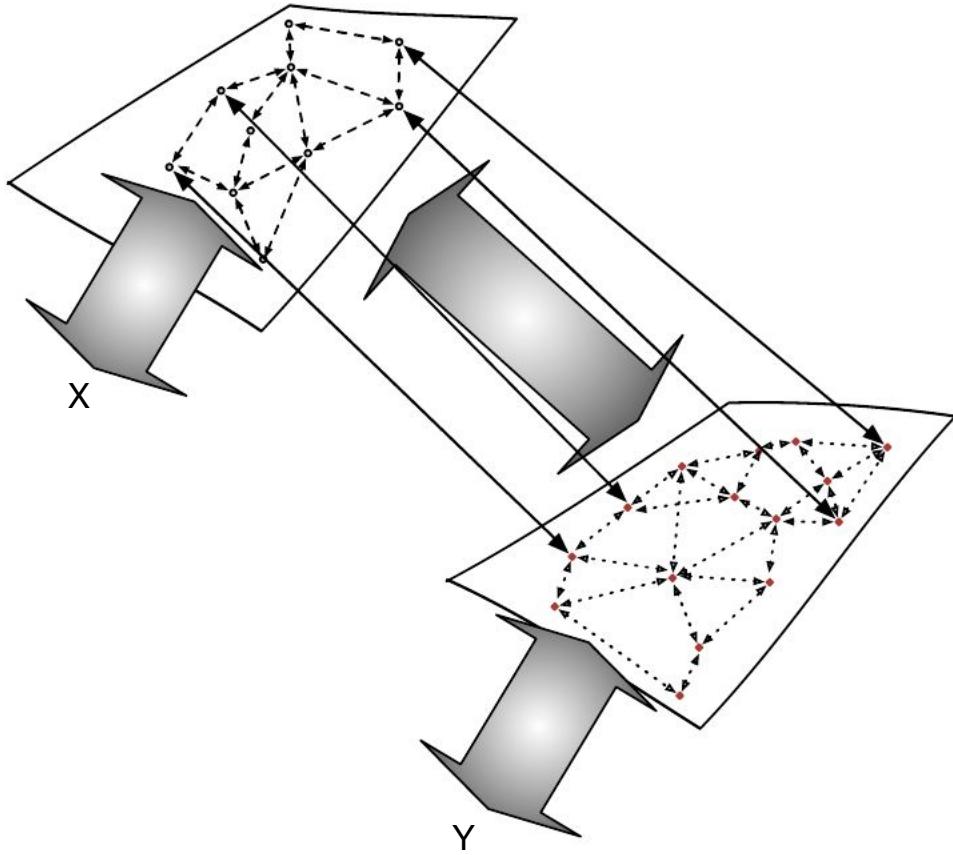


Figure 17.8: Transfer learning between two domains corresponds to zero-shot learning. A first set of data (dashed arrows) can be used to relate examples in one domain (top left, vX) and fix a relationship between their representations, a second set of data (dotted arrows) can be used to similarly relate examples and their representation in the other domain (bottom right, \mathbf{Y}), while a third dataset (full large arrows) *anchors* the two representations together, with examples consisting of pairs (\mathbf{x}, \mathbf{y}) taken from the two domains. In this way, one can for example associate an image to a word, even if no images of that word were ever presented, simply because word-representations (top) and image-representations (bottom) have been learned jointly with a two-way relationship between them.

This is illustrated in Figure 17.8, where we see that zero-shot learning is a particular form of transfer learning. The same principle explains how one can perform *multi-modal learning*, capturing a representation in one modality, a representation in the other, and the relationship (in general a joint distribution) between pairs (\mathbf{x}, \mathbf{y}) consisting of one observation \mathbf{x} in one modality and another observation \mathbf{y} in the other modality (Srivastava and Salakhutdinov, 2012). By learning all three sets of parameters (from \mathbf{x} to its representation, from \mathbf{y} to its representation, and the relationship between the two

representation), concepts in one map are anchored in the other, and vice-versa, allowing one to meaningfully generalize to new pairs.

17.3 Semi-Supervised Learning

As discussed in Section 17.1.1 on the advantages of unsupervised pre-training, unsupervised learning can have a regularization effect in the context of supervised learning. This fits in the more general category of combining unlabeled examples with unknown distribution $P(\mathbf{x})$ with labeled examples (\mathbf{x}, \mathbf{y}) , with the objective of estimating $P(\mathbf{y}|\mathbf{x})$. Exploiting unlabeled examples to improve performance on a labeled set is the driving idea behind semi-supervised learning (Chapelle *et al.*, 2006). For example, one can use unsupervised learning to map X into a representation (also called embedding) such that two examples \mathbf{x}_1 and \mathbf{x}_2 that belong to the same cluster (or are reachable through a short path going through neighboring examples in the training set) end up having nearby embeddings. One can then use supervised learning (e.g., a linear classifier) in that new space and achieve better generalization in many cases (Belkin and Niyogi, 2002; Chapelle *et al.*, 2003). A long-standing variant of this approach is the application of Principal Components Analysis as a pre-processing step before applying a classifier (on the projected data). In these models, the data is first transformed in a new representation using unsupervised learning, and a supervised classifier is stacked on top, learning to map the data in this new representation into class predictions.

Instead of having separate unsupervised and supervised components in the model, one can consider models in which $P(\mathbf{x})$ (or $P(\mathbf{x}, \mathbf{y})$) and $P(\mathbf{y}|\mathbf{x})$ share parameters (or whose parameters are connected in some way), and one can trade-off the supervised criterion $-\log P(\mathbf{y}|\mathbf{x})$ with the unsupervised or generative one ($-\log P(\mathbf{x})$ or $-\log P(\mathbf{x}, \mathbf{y})$). It can then be seen that the generative criterion corresponds to a particular form of prior (Lasserre *et al.*, 2006), namely that the structure of $P(\mathbf{x})$ is connected to the structure of $P(\mathbf{y}|\mathbf{x})$ in a way that is captured by the shared parametrization. By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion (Lasserre *et al.*, 2006; Larochelle and Bengio, 2008b).

In the context of deep architectures, a very interesting application of these ideas involves adding an unsupervised embedding criterion at each layer (or only one intermediate layer) to a traditional supervised criterion (Weston *et al.*, 2008). This has been shown to be a powerful semi-supervised learning strategy, and is an alternative to the unsupervised pre-training approach described earlier in this chapter, which also combine unsupervised learning with supervised learning.

In the context of scarcity of labeled data (and abundance of unlabeled data), deep architectures have shown promise as well. Salakhutdinov and Hinton (2008) describe a method for learning the covariance matrix of a Gaussian Process, in which the usage of unlabeled examples for modeling $P(\mathbf{x})$ improves $P(\mathbf{y}|\mathbf{x})$ quite significantly. Note that such a result is to be expected: with few labeled samples, modeling $P(\mathbf{x})$ usually helps, as argued below (Section 17.4). These results show that even in the context of

abundant labeled data, unsupervised pre-training can have a pronounced positive effect on generalization: a somewhat surprising conclusion.

17.4 Causality, Semi-Supervised Learning and Disentangling the Underlying Factors

What we put forward as a hypothesis, going a bit further, is that an *ideal representation* is one that *disentangles the underlying causal factors of variation that generated the observed data*. Note that this may be different from “easy to model”, but we further assume that for most problems of interest, these two properties coincide: once we “understand” the underlying explanations for what we observe, it generally becomes easy to predict one thing from others.

A very basic question is whether unsupervised learning on input variables \mathbf{x} can yield representations that are useful when later trying to learn to predict some target variable \mathbf{y} , given \mathbf{x} . More generally, when does semi-supervised learning work? See also Section 17.3 for an earlier discussion.

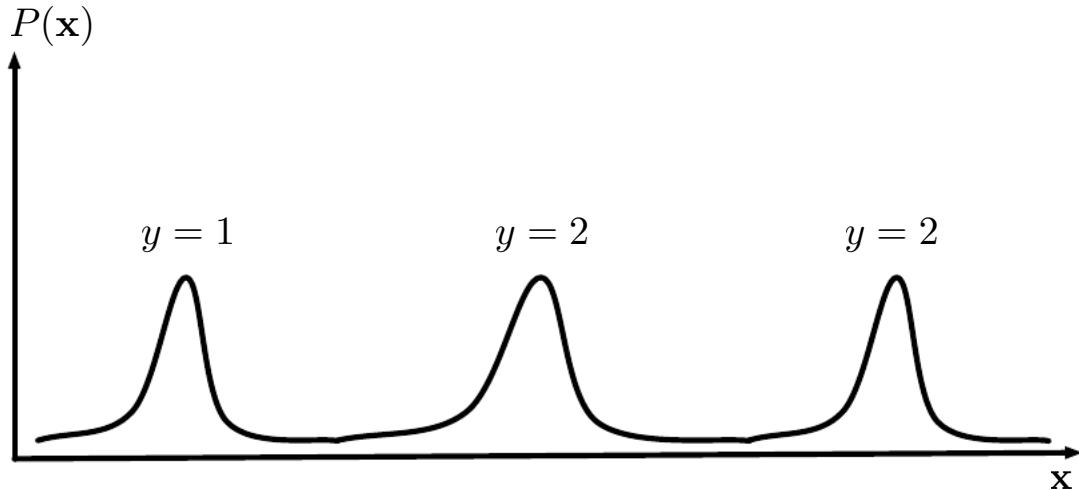


Figure 17.9: Example of a density over \mathbf{x} that is a mixture over three components. The component identity is an underlying explanatory factor, y . Because the mixture components (e.g., natural object classes in image data) are statistically salient, just modeling $P(\mathbf{x})$ in an unsupervised way with no labeled example already reveals the factor y .

It turns out that the answer to this question is very different dependent on the underlying relationship between \mathbf{x} and \mathbf{y} . Put differently, the question is whether $P(\mathbf{y}|\mathbf{x})$, seen as a function of \mathbf{x} has anything to do with $P(\mathbf{x})$. If not, then unsupervised learning of $P(\mathbf{x})$ can be of no help to learn $P(\mathbf{y}|\mathbf{x})$. Consider for example the case where $P(\mathbf{x})$ is uniformly distributed and $\mathbb{E}[\mathbf{y}|\mathbf{x}]$ is some function of interest. Clearly, observing \mathbf{x} alone gives us no information about $P(\mathbf{y}|\mathbf{x})$. As a better case, consider the situation where

\mathbf{x} arises from a mixture, with one mixture component per value of \mathbf{y} , as illustrated in Figure 17.9. If the mixture components are well separated, then modeling $P(\mathbf{x})$ tells us precisely where each component is, and a single labeled example of each example will then be enough to perfectly learn $P(\mathbf{y}|\mathbf{x})$. But more generally, what could make $P(\mathbf{y}|\mathbf{x})$ and $P(\mathbf{x})$ tied together?

If \mathbf{y} is closely associated with one of the causal factors of \mathbf{x} , then, as first argued by Janzing et al. (2012), $P(\mathbf{x})$ and $P(\mathbf{y}|\mathbf{x})$ will be strongly tied, and unsupervised representation learning that tries to disentangle the underlying factors of variation is likely to be useful as a semi-supervised learning strategy.

Consider the assumption that \mathbf{y} is one of the causal factors of \mathbf{x} , and let \mathbf{h} represent all those factors. Then the true generative process can be conceived as structured according to this directed graphical model, with \mathbf{h} as the parent of \mathbf{x} :

$$P(\mathbf{h}, \mathbf{x}) = P(\mathbf{x}|\mathbf{h})P(\mathbf{h}).$$

As a consequence, the data has marginal probability

$$P(\mathbf{x}) = \int P(\mathbf{x}|\mathbf{h})p(\mathbf{h})d\mathbf{h}$$

or, in the discrete case (like in the mixture example above):

$$P(\mathbf{x}) = \sum_{\mathbf{h}} P(\mathbf{x}|\mathbf{h})P(\mathbf{h}).$$

From this straightforward observation, we conclude that the best possible model of \mathbf{x} (from a generalization point of view) is the one that uncovers the above “true” structure, with \mathbf{h} as a latent variable that explains the observed variations in \mathbf{x} . The “ideal” representation learning discussed above should thus recover these latent factors. If \mathbf{y} is one of them (or closely related to one of them), then it will be very easy to learn to predict \mathbf{y} from such a representation. We also see that the conditional distribution of \mathbf{y} given \mathbf{x} is tied by Bayes rule to the components in the above equation:

$$P(\mathbf{y}|\mathbf{x}) = \frac{P(\mathbf{x}|\mathbf{y})P(\mathbf{y})}{P(\mathbf{x})}.$$

Thus the marginal $P(\mathbf{x})$ is intimately tied to the conditional $P(\mathbf{y}|\mathbf{x})$ and knowledge of the structure of the former should be helpful to learn the latter, i.e., semi-supervised learning works. Furthermore, not knowing which of the factors in \mathbf{h} will be the one of interest, say $\mathbf{y} = \mathbf{h}_i$, an unsupervised learner should learn a representation that disentangles all the generative factors \mathbf{h}_j from each other, then making it easy to predict \mathbf{y} from \mathbf{h} .

In addition, as pointed out by Janzing et al. (2012), if the true generative process has \mathbf{x} as an effect and \mathbf{y} as a cause, then modeling $P(\mathbf{x}|\mathbf{y})$ is robust to changes in $P(\mathbf{y})$. If the cause-effect relationship was reversed, it would not be true, since by Bayes rule, $P(\mathbf{x}|\mathbf{y})$ would be sensitive to changes in $P(\mathbf{y})$. Very often, when we consider changes in distribution due to different domains, temporal non-stationarity, or changes in the

nature of the task, *the causal mechanisms remain invariant* (“the laws of the universe are constant”) whereas what changes are the marginal distribution over the underlying causes (or what factors are linked to our particular task). Hence, better generalization and robustness to all kinds of changes can be expected via learning a generative model that attempts to recover the causal factors \mathbf{h} and $P(\mathbf{x}|\mathbf{h})$.

17.5 Assumption of Underlying Factors and Distributed Representation

A very basic notion that comes out of the above discussion and of the notion of “disentangled factors” is the very idea that there are underlying factors that generate the observed data. It is a core assumption behind most neural network and deep learning research, more precisely relying on the notion of *distributed representation*.

What we call a distributed representation is one which can express an exponentially large number of concepts by allowing to compose the activation of many features. An example of distributed representation is a vector of n binary features, which can take 2^n configurations, each potentially corresponding to a different region in input space. This can be compared with a *symbolic representation*, where the input is associated with a single symbol or category. If there are n symbols in the dictionary, one can imagine n feature detectors, each corresponding to the detection of the presence of the associated category. In that case only n different configurations of the representation-space are possible, carving n different regions in input space. Such a symbolic representation is also called a one-hot representation, since it can be captured by a binary vector with n bits that are mutually exclusive (only one of them can be active). These ideas are developed further in the next section.

Examples of learning algorithms based on non-distributed representations include:

- Clustering methods, including the k-means algorithm: only one cluster “wins” the competition.
- K-nearest neighbors algorithms: only one template or prototype example is associated with a given input.
- Decision trees: only one leaf (and the nodes on the path from root to leaf) is activated when an input is given.
- Gaussian mixtures and mixtures of experts: the templates (cluster centers) or experts are now associated with a *degree* of activation, which makes the posterior probability of components (or experts) given input look more like a distributed representation. However, as discussed in the next section, these models still suffer from a poor statistical scaling behavior compared to those based on distributed representations (such as products of experts and RBMs).
- Kernel machines with a Gaussian kernel (or other similarly local kernel): although the degree of activation of each “support vector” or template example is now continuous-valued, the same issue arises as with Gaussian mixtures.

- Language or translation models based on N-grams: the set of contexts (sequences of symbols) is partitioned according to a tree structure of suffixes (e.g. a leaf may correspond to the last two words being w_1 and w_2), and separate parameters are estimated for each leaf of the tree (with some sharing being possible of parameters associated with internal nodes, between the leaves of the sub-tree rooted at the same internal node).

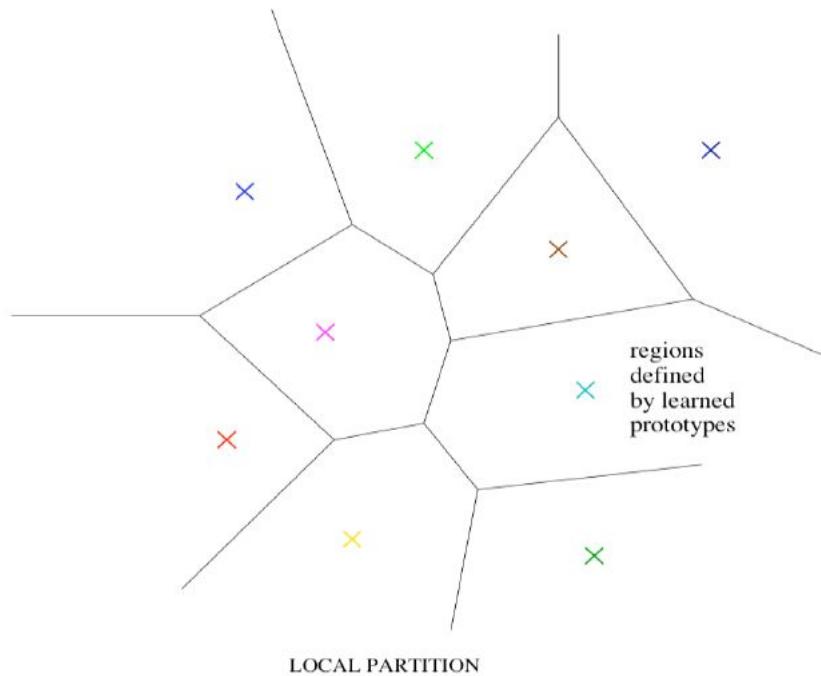


Figure 17.10: Illustration of how a learning algorithm based on a non-distributed representation breaks up the input space into regions, *with a separate set of parameters for each region*. For example, a clustering algorithm or a 1-nearest-neighbor algorithm associates one template (colored X) to each region. This is also true of decision trees, mixture models, and kernel machines with a local (e.g., Gaussian) kernel. In the latter algorithms, the output is not constant by parts but instead interpolates between neighboring regions, but the relationship between the number of parameters (or examples) and the number of regions they can define remains linear. The advantage is that a different answer (e.g., density function, predicted output, etc.) can be *independently* chosen for each region. The disadvantage is that there is no generalization to new regions, except by extending the answer for which there is data, exploiting solely a *smoothness prior*. It makes it difficult to learn a complicated function, with more ups and downs than the available number of examples. Contrast this with a distributed representation, Figure 17.11.

An important related concept that distinguishes a distributed representation from a

symbolic one is that *generalization arises due to shared attributes* between different concepts. As pure symbols, “`tt cat`” and “`dog`” are as far from each other as any other two symbols. However, if one associates them with a meaningful distributed representation, then many of the things that can be said about cats can generalize to dogs and vice-versa. This is what allows neural language models to generalize so well (Section 13.3). Distributed representations induce a rich *similarity space*, in which semantically close concepts (or inputs) are close in distance, a property that is absent from purely symbolic representations. Of course, one would get a distributed representation if one would associate *multiple symbolic attributes* to each symbol.

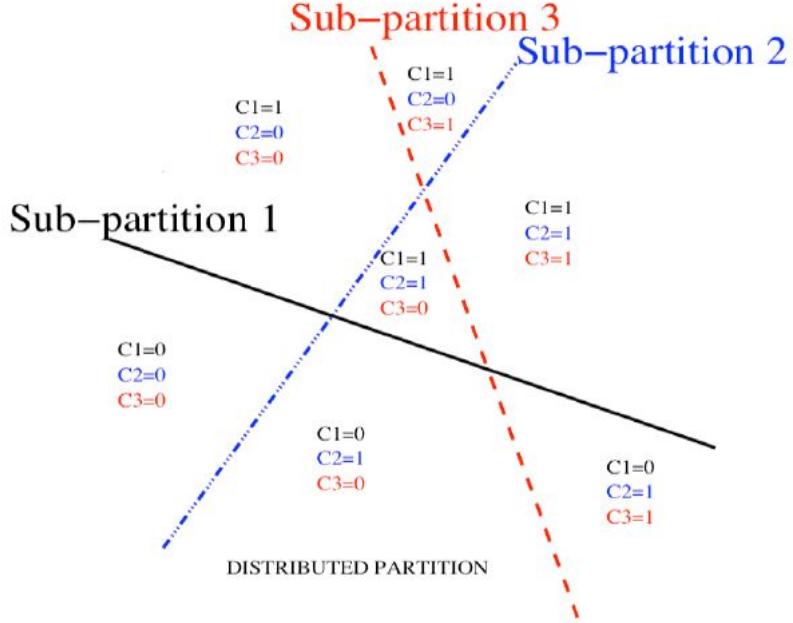


Figure 17.11: Illustration of how a learning algorithm based on a distributed representation breaks up the input space into regions, *with exponentially more regions than parameters*. Instead of a single partition (as in the non-distributed case, Figure 17.10), we have many partitions, one per “feature”, and all their possible intersections. In the example of the figure, there are 3 binary features C_1 , C_2 , and C_3 , each corresponding to partitioning the input space in two regions according to a hyperplane, i.e., each is a linear classifier. Each possible intersection of these half-planes forms a region, i.e., each region corresponds to a configuration of the bits specifying whether each feature is 0 or 1, on which side of their hyperplane is the input falling. If the input space is large enough, the number of regions grows exponentially with the number of features, i.e., of parameters. However, the way these regions carve the input space still depends on few parameters: this huge number of regions are not placed independently of each other. We can thus represent a function that *looks complicated* but actually has structure. Basically, the assumption is that one can learn about each feature without having to see the examples for all the configurations of all the other features, i.e., these features correspond to underlying factors explaining the data.

Note that a *sparse representation* is a distributed representation where the number of attributes that are active together is small compared to the total number of attributes. For example, in the case of binary representations, one might have only $k \ll n$ of the n bits that are non-zero. The power of representation grows exponentially with the number of active attributes, e.g., $O(n^k)$ in the above example of binary vectors. At the extreme, a symbolic representation is a very sparse representation where only one

attribute at a time can be active.

17.6 Exponential Gain in Representational Efficiency from Distributed Representations

When and why can there be a statistical advantage from using a distributed representation as part of a learning algorithm?

Figures 17.10 and 17.11 explain that advantage in intuitive terms. The argument is that a function that “looks complicated” can be compactly represented using a small number of parameters, if some “structure” is uncovered by the learner. Traditional “non-distributed” learning algorithms generalize only due to the smoothness assumption, which states that if $u \approx v$, then the target function f to be learned has the property that $f(u) \approx f(v)$, in general. There are many ways of formalizing such an assumption, but the end result is that if we have an example (x, y) for which we know that $f(x) \approx y$, then we choose an estimator \hat{f} that approximately satisfies these constraints while changing as little as possible. This assumption is clearly very useful, but it suffers from the curse of dimensionality: in order to learn a target function that takes many different values (e.g. many ups and downs) in a large number of regions², we may need a number of examples that is at least as large as the number of distinguishable regions. One can think of each of these regions as a category or symbol: by having a separate degree of freedom for each symbol (or region), we can learn an arbitrary mapping from symbol to value. However, this does not allow us to generalize to new symbols, new regions.

If we are lucky, there may be some regularity in the target function, besides being smooth. For example, the same pattern of variation may repeat itself many times (e.g., as in a periodic function or a checkerboard). If we only use the smoothness prior, we will need additional examples for each repetition of that pattern. However, as discussed by Montufar *et al.* (2014), a deep architecture could represent and discover such a repetition pattern and generalize to new instances of it. Thus a small number of parameters (and therefore, a small number of examples) could suffice to represent a function that looks complicated (in the sense that it would be expensive to represent with a non-distributed architecture). Figure 17.11 shows a simple example, where we have n binary features in a d -dimensional space, and where each binary feature corresponds to a linear classifier that splits the input space in two parts. The exponentially large number of intersections of n of the corresponding half-spaces corresponds to as many distinguishable regions that a distributed representation learner could capture. How many regions are generated by an arrangement of n hyperplanes in \mathbb{R}^d ? This corresponds to the number of regions that a shallow neural network (one hidden layer) can distinguish (Pascanu *et al.*, 2014c), which is

$$\sum_{j=0}^d \binom{n}{j} = O(n^d),$$

²e.g., exponentially many regions: in a d -dimensional space with at least 2 different values to distinguish per dimension, we might want f to differ in 2^d different regions, requiring $O(2^d)$ training examples.

following a more general result from [Zaslavsky \(1975\)](#), known as Zaslavsky's theorem, one of the central results from the theory of hyperplane arrangements. Therefore, we see a growth that is exponential in the input size and polynomial in the number of hidden units.

Although a distributed representation (e.g. a shallow neural net) can represent a richer function with a smaller number of parameters, there is no free lunch: to construct an *arbitrary* partition (say with 2^d different regions) one will need a correspondingly large number of hidden units, i.e., of parameters and of examples. The use of a distributed representation therefore also corresponds to a prior, which comes on top of the smoothness prior. To return to the hyperplanes examples of Figure 17.11, we see that we are able to get this generalization because we can learn about the location of each hyperplane with only $O(d)$ examples: we do not need to see examples corresponding to all $O(n^d)$ regions.

Let us consider a concrete example. Imagine that the input is the image of a person, and that we have a classifier that detects whether the person is a child or not, another that detects if that person is a male or a female, another that detects whether that person wears glasses or not, etc. Keep in mind that these features are discovered automatically, not fixed a priori. We can learn about the male vs female distinction, or about the glasses vs no-glasses case, without having to consider all of the configurations of the n features. This form of statistical separability is what allows one to generalize to new configurations of a person's features that have never been seen during training. It corresponds to the prior discussed above regarding the existence of multiple underlying explanatory factors. This prior is very plausible for most of the data distributions on which human intelligence would be useful, but it may not apply to every possible distribution. However, this apparently innocuous assumption buys us a lot, statistically speaking, because it allows the learner to discover structure with a reasonably small number of examples that would otherwise require exponentially more training data.

Another interesting result illustrating the statistical effect of a distributed representations versus a non-distributed one is the mathematical analysis ([Montufar and Morton, 2014](#)) of *products of mixtures* (which include the RBM as a special case) versus *mixture of products* (such as the mixture of Gaussians). The analysis shows that a mixture of products can require an exponentially larger number of parameters in order to represent the probability distributions arising out of a product of mixtures.

17.7 Exponential Gain in Representational Efficiency from Depth

In the above example with the input being an image of a person, it would not be reasonable to expect factors such as gender, age, and the presence of glasses to be detected simply from a linear classifier, i.e., a shallow neural network. The kinds of factors that can be chosen almost independently in order to generate data are more likely to be very high-level and related in highly non-linear ways to the input. This demands *deep* distributed representations, where the higher level features (seen as functions of

the input) or factors (seen as generative causes) are obtained through the composition of many non-linearities.

It turns out that organizing computation through the composition of many non-linearities and a hierarchy of re-used features can give another exponential boost to statistical efficiency. Although 2-layer networks (e.g., with saturating non-linearities, boolean gates, sum/products, or RBF units) can generally be shown to be universal approximators³, the required number of hidden units may be very large. The main results on the expressive power of deep architectures state that there are families of functions that can be represented efficiently with a deep architecture (say depth k) but would require an exponential number of components (with respect to the input size) with insufficient depth (depth 2 or depth $k - 1$).

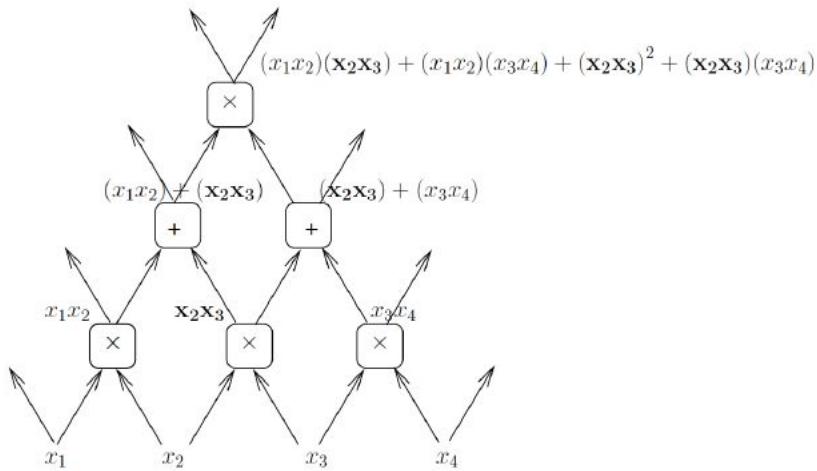


Figure 17.12: A sum-product network (Poon and Domingos, 2011) composes summing units and product units, so that each node computes a polynomial. Consider the product node computing x_2x_3 : its value is re-used in its two immediate children, and indirectly incorporated in its grand-children. In particular, in the top node shown the product x_2x_3 would arise 4 times if that node's polynomial was expanded as a sum of products. That number could double for each additional layer. In general a deep sum of product can represent polynomials with a number of min-terms that is exponential in depth, and some families of polynomials are represented efficiently with a deep sum-product network but not efficiently representable with a simple sum of products, i.e., a 2-layer network (Delalleau and Bengio, 2011).

More precisely, a feedforward neural network with a single hidden layer is a universal approximator (of Borel measurable functions) (Hornik *et al.*, 1989; Cybenko, 1989). Other works have investigated universal approximation of probability distributions by deep belief networks (Le Roux and Bengio, 2010; Montúfar and Ay, 2011), as well as their approximation properties (Montúfar, 2014; Krause *et al.*, 2013).

³with enough hidden units they can approximate a large class of functions (e.g. continuous functions) up to some given tolerance level

Regarding the advantage of depth, early theoretical results have focused on circuit operations (neural net unit computations) that are substantially different from those being used in real state-of-the-art deep learning applications, such as logic gates (Håstad, 1986) and linear threshold units with non-negative weights (Håstad and Goldmann, 1991). More recently, Delalleau and Bengio (2011) showed that a shallow network requires exponentially many more sum-product hidden units⁴ than a deep sum-product network (Poon and Domingos, 2011) in order to compute certain families of polynomials. Figure 17.12 illustrates a sum-product network for representing polynomials, and how a deeper network can be exponentially more efficient because the same computation can be re-used exponentially (in depth) many times. Note however that Martens and Medabalimi (2014) showed that sum-product networks were somewhat limited in their expressive power, in the sense that there are distributions that can easily be represented by other generative models but that cannot be efficiently represented under the decomposability and completeness conditions associated with the probabilistic interpretation of sum-product networks (Poon and Domingos, 2011).

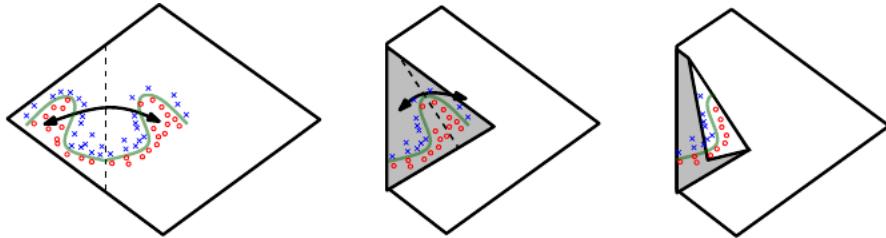


Figure 17.13: An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit. If one considers a function computed on top of that unit (the green decision surface), it will be formed of a mirror image of a simpler pattern, across that axis of symmetry. The middle image shows how it can be obtained by folding the space around that axis of symmetry, and the right image shows how another repeating pattern can be folded on top of it (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers). This is an intuitive explanation of the exponential advantage of deeper rectifier networks formally shown in Pascanu *et al.* (2014b); Montufar *et al.* (2014).

Closer to the kinds of deep networks actually used in practice (Pascanu *et al.*, 2014b; Montufar *et al.*, 2014) showed that piecewise linear networks (e.g. obtained from rectifier non-linearities or maxout units) could represent functions with exponentially more piecewise-linear regions, as a function of depth, compared to shallow neural networks. Figure 17.13 illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value non-linearity). By com-

⁴Here, a single sum-product hidden layer summarizes a layer of product units followed by a layer of sum units.

posing these folding operations, we obtain an exponentially large number of piecewise linear regions which can capture all kinds of regular (e.g. repeating) patterns.

More precisely, the main theorem in [Montufar et al. \(2014\)](#) states that the number of linear regions carved out by a deep rectifier network with d inputs, depth L , and n units per hidden layer, is

$$O \left(\binom{n}{d}^{d(L-1)} n^d \right),$$

i.e., exponential in the depth L . In the case of maxout networks with k filters per unit, the number of linear regions is

$$O \left(k^{(L-1)+d} \right).$$

17.8 Priors Regarding The Underlying Factors

To close this chapter, we come back to the original question: what is a good representation? We proposed that an ideal representation is one that disentangles the underlying causal factors of variation that generated the data, especially those factors that we care about in our applications. It seems clear that if we have direct clues about these factors (like if a factor $\mathbf{y} = \mathbf{h}_i$, a label, is observed at the same time as an input \mathbf{x}), then this can help the learner separate these observed factors from the others. This is already what supervised learning does. But in general, we may have a lot more unlabeled data than labeled data: can we use other clues, other hints about the underlying factors, in order to disentangle them more easily?

What we propose here is that indeed we can provide all kinds of broad priors which are as many hints that can help the learner discover, identify and disentangle these factors. The list of such priors is clearly not exhaustive, but it is a starting point, and yet most learning algorithms in the machine learning literature only exploit a small subset of these priors. With absolutely no priors, we know that it is not possible to generalize: this is the essence of the *no-free-lunch theorem for machine learning*. In the space of all functions, which is huge, with any finite training set, there is no general-purpose learning recipe that would dominate all other learning algorithms. Whereas some assumptions are required, when our goal is to build AI or understand human intelligence, it is tempting to focus our attention on the most general and broad priors, that are relevant for most of the tasks that humans are able to successfully learn.

This list was introduced in section 3.1 of [Bengio et al. \(2013c\)](#).

- **Smoothness:** we want to learn functions f s.t. $x \approx y$ generally implies $f(x) \approx f(y)$. This is the most basic prior and is present in most machine learning, but is insufficient to get around the curse of dimensionality, as discussed above and in [Bengio et al. \(2013c\)](#). below.
- **Multiple explanatory factors:** the data generating distribution is generated by different underlying factors, and for the most part what one learns about one factor generalizes in many configurations of the other factors. This assumption is behind the idea of **distributed representations**, discussed in Section 17.5 above.

- **Depth, or a hierarchical organization of explanatory factors:** the concepts that are useful at describing the world around us can be defined in terms of other concepts, in a hierarchy, with more **abstract** concepts higher in the hierarchy, being defined in terms of less abstract ones. This is the assumption exploited by having **deep representations**.
- **Causal factors:** the input variables \mathbf{x} are consequences, effects, while the explanatory factors are causes, and not vice-versa. As discussed above, this enables the **semi-supervised learning** assumption, i.e., that $P(\mathbf{x})$ is tied to $P(\mathbf{y}|\mathbf{x})$, making it possible to improve the learning of $P(\mathbf{y}|\mathbf{x})$ via the learning of $P(\mathbf{x})$. More precisely, this entails that representations that are useful for $P(\mathbf{x})$ are useful when learning $P(\mathbf{y}|\mathbf{x})$, allowing sharing of statistical strength between the unsupervised and supervised learning tasks.
- **Shared factors across tasks:** in the context where we have many tasks, corresponding to different \mathbf{y}_i 's sharing the same input \mathbf{x} or where each task is associated with a subset or a function $f_i(\mathbf{x})$ of a global input \mathbf{x} , the assumption is that each \mathbf{y}_i is associated with a different subset from a common pool of relevant factors \mathbf{h} . Because these subsets overlap, learning all the $P(\mathbf{y}_i|\mathbf{x})$ via a shared intermediate representation $P(\mathbf{h}|\mathbf{x})$ allows sharing of statistical strength between the tasks.
- **Manifolds:** probability mass concentrates, and the regions in which it concentrates are locally connected and occupy a tiny volume. In the continuous case, these regions can be approximated by low-dimensional manifolds that a much smaller dimensionality than the original space where the data lives. This is the manifold hypothesis and is covered in Chapter 18, especially with algorithms related to auto-encoders.
- **Natural clustering:** different values of categorical variables such as object classes⁵ are associated with separate manifolds. More precisely, the local variations on the manifold tend to preserve the value of a category, and a linear interpolation between examples of different classes in general involves going through a low density region, i.e., $P(\mathbf{x}|y = i)$ for different i tend to be well separated and not overlap much. For example, this is exploited explicitly in the Manifold Tangent Classifier discussed in Section 18.4. This hypothesis is consistent with the idea that humans have *named* categories and classes because of such statistical structure (discovered by their brain and propagated by their culture), and machine learning tasks often involve predicting such categorical variables.
- **Temporal and spatial coherence:** this is similar to the cluster assumption but concerns sequences or tuples of observations; consecutive or spatially nearby observations tend to be associated with the same value of relevant categorical concepts, or result in a small move on the surface of the high-density manifold. More generally, different factors change at different temporal and spatial scales, and many

⁵it is often the case that the y of interest is a category

categorical concepts of interest change slowly. When attempting to capture such categorical variables, this prior can be enforced by making the associated representations slowly changing, i.e., penalizing changes in values over time or space. This prior was introduced in [Becker and Hinton \(1992\)](#).

- **Sparsity:** for any given observation x , only a small fraction of the possible factors are relevant. In terms of representation, this could be represented by features that are often zero (as initially proposed by [Olshausen and Field \(1996\)](#)), or by the fact that most of the extracted features are *insensitive* to small variations of \mathbf{x} . This can be achieved with certain forms of priors on latent variables (peaked at 0), or by using a non-linearity whose value is often flat at 0 (i.e., 0 and with a 0 derivative), or simply by penalizing the magnitude of the Jacobian matrix (of derivatives) of the function mapping input to representation. This is discussed in Section [16.7](#).
- **Simplicity of Factor Dependencies:** in good high-level representations, the factors are related to each other through simple dependencies. The simplest possible is marginal independence, $P(\mathbf{h}) = \prod_i P(\mathbf{h}_i)$, but linear dependencies or those captured by a shallow auto-encoder are also reasonable assumptions. This can be seen in many laws of physics, and is assumed when plugging a linear predictor or a factorized prior on top of a learned representation.

Chapter 18

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 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

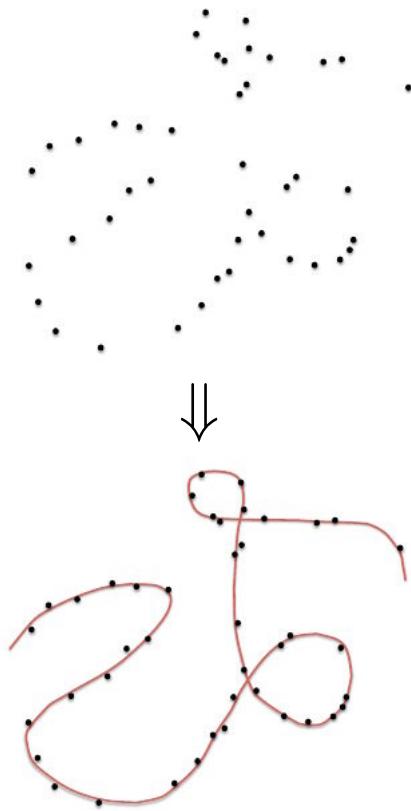


Figure 18.1: Top: data sampled from a distribution in a high-dimensional space (one 2 dimensions shown for illustration) that is actually concentrated near a one-dimensional manifold, which here is like a twisted string. Bottom: the underlying manifold that the learner should infer.

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.

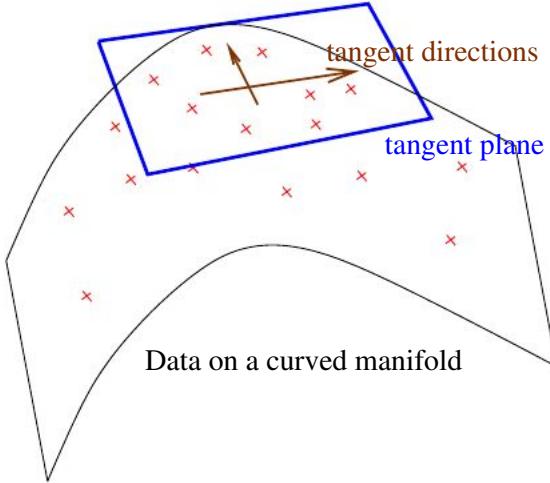


Figure 18.2: A two-dimensional manifold near which training examples are concentrated, along with a tangent plane and its associated tangent directions, forming a basis that specify the directions of small moves one can make to stay on the manifold.

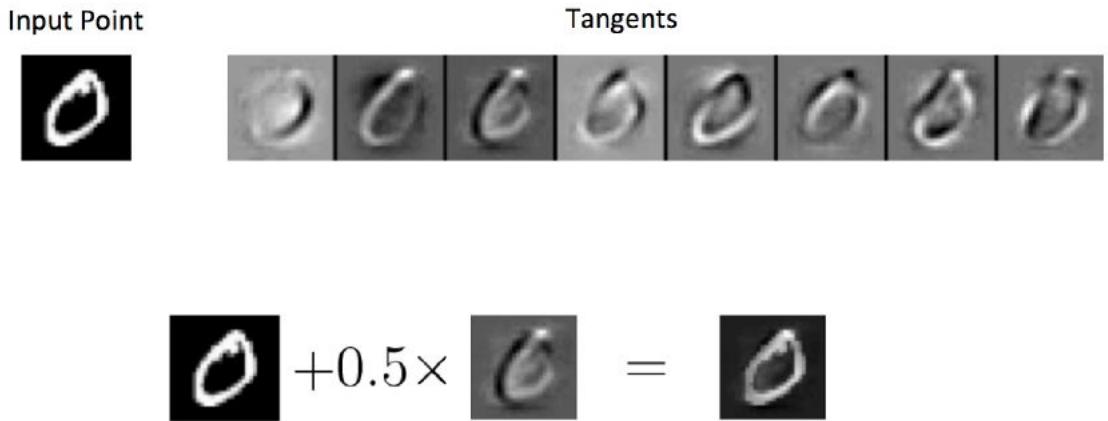


Figure 18.3: Illustration of tangent vectors of the manifold estimated by a contractive auto-encoder (CAE), at some input point (top left, image of a zero). Each image on the top right corresponds to a tangent vector. They are obtained by picking the dominant singular vectors (with largest singular value) of the Jacobian $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ (see Section 16.9). 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 configu-

ration 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 18.1.

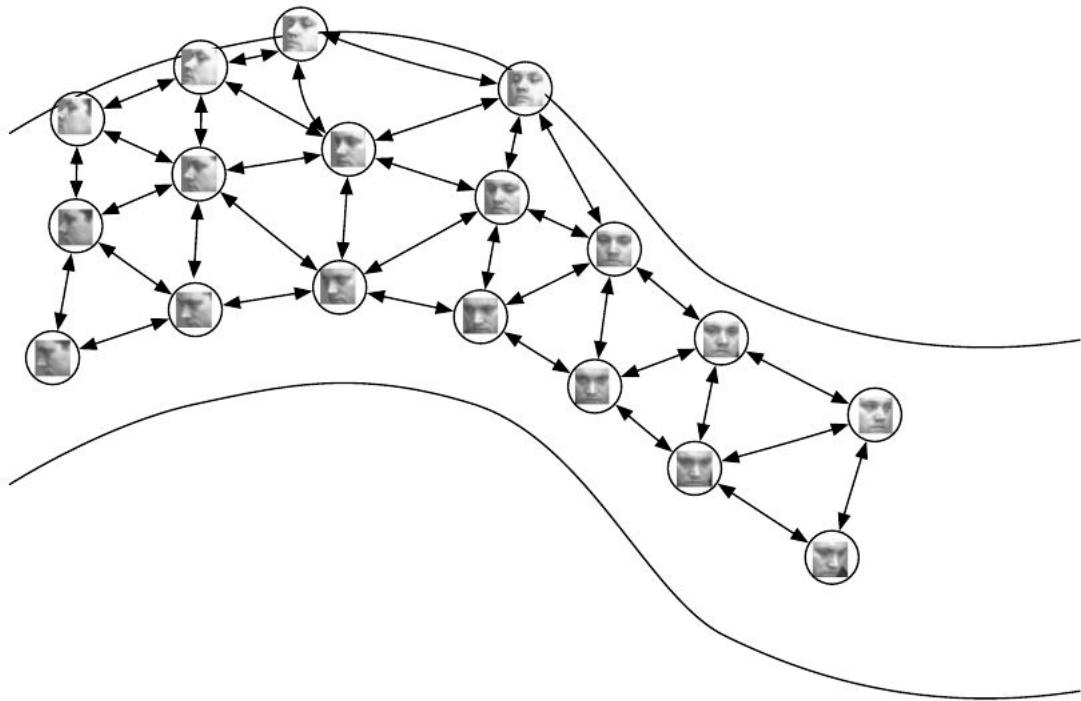


Figure 18.4: Non-parametric manifold learning procedures build a nearest neighbor graph whose nodes are training examples and arcs connect nearest neighbors. Various procedures can thus obtain the tangent plane associated with a neighborhood of the graph, and a coordinate system that associates each 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).

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 18.2, these local directions specify how one can change x infinitesimally while staying on the manifold.

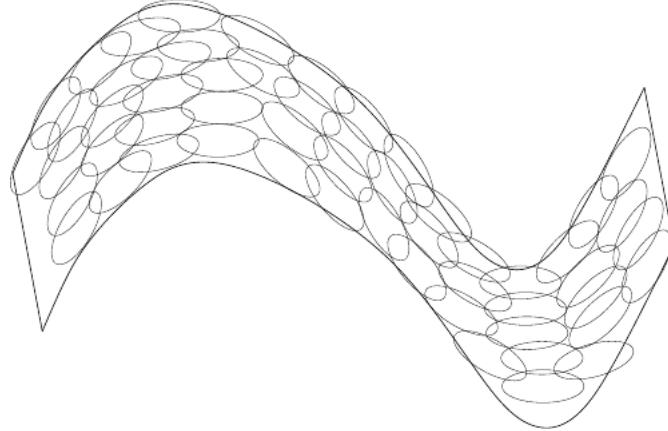


Figure 18.5: If the tangent plane at each location is known, then they can be tiled to form a global coordinate system or a density function. In the figure, each local patch can be thought of as a local Euclidean coordinate system or as a locally flat Gaussian, or “pancake”, with a very small variance 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.*, 2006b).

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 18.4.

A global coordinate system can then be obtained through an optimization or solving a linear system. Figure 18.5 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).

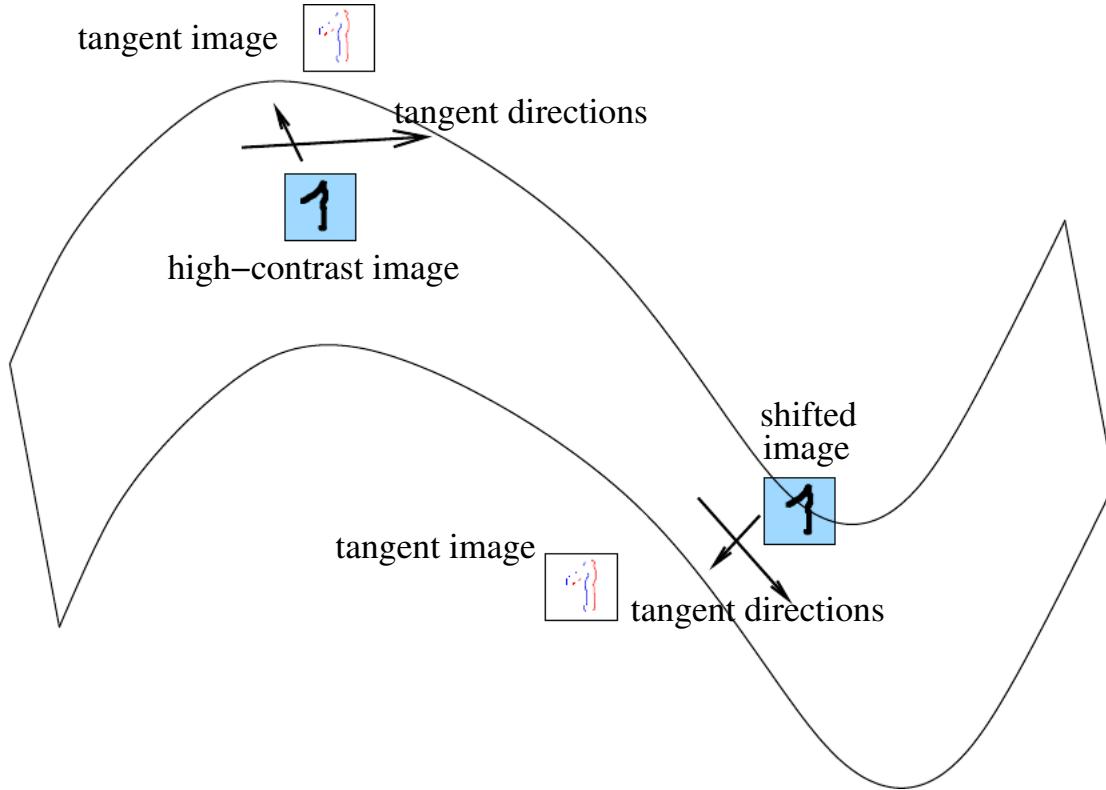


Figure 18.6: When the data are images, the tangent vectors can also be visualized like images. Here we show the tangent vector associated with translation: it corresponds to the difference between an image and a slightly translated version. This basically extracts edges, with the sign (shown as red or green color) indicating whether ink is being added or subtracted when moving to the right. The figure illustrates the fact that the tangent vectors along the manifolds associated with translation, rotation and such apparently benign transformations of images, can be *highly curved*: the tangent vector at a position x is very different from the tangent vector at a nearby position (slightly translated image). Note how the two tangent vectors shown have almost no overlap, i.e., their dot product is nearly 0, and they are as far as two such images can be. If the tangent vectors of a manifold change quickly, it means that the manifold is highly curved. This is going to make it very difficult to capture such manifolds (with a huge numbers of twists, ups and downs) with local non-parametric methods such as graph-based non-parametric manifold learning. This point was made in [Bengio and Monperrus \(2005\)](#).

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 18.6. This motivates the use of distributed representations and deep learning for capturing manifold structure, which is the subject of this chapter.



Figure 18.7: 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 18.8 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 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 18.7, 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 18.8 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*.

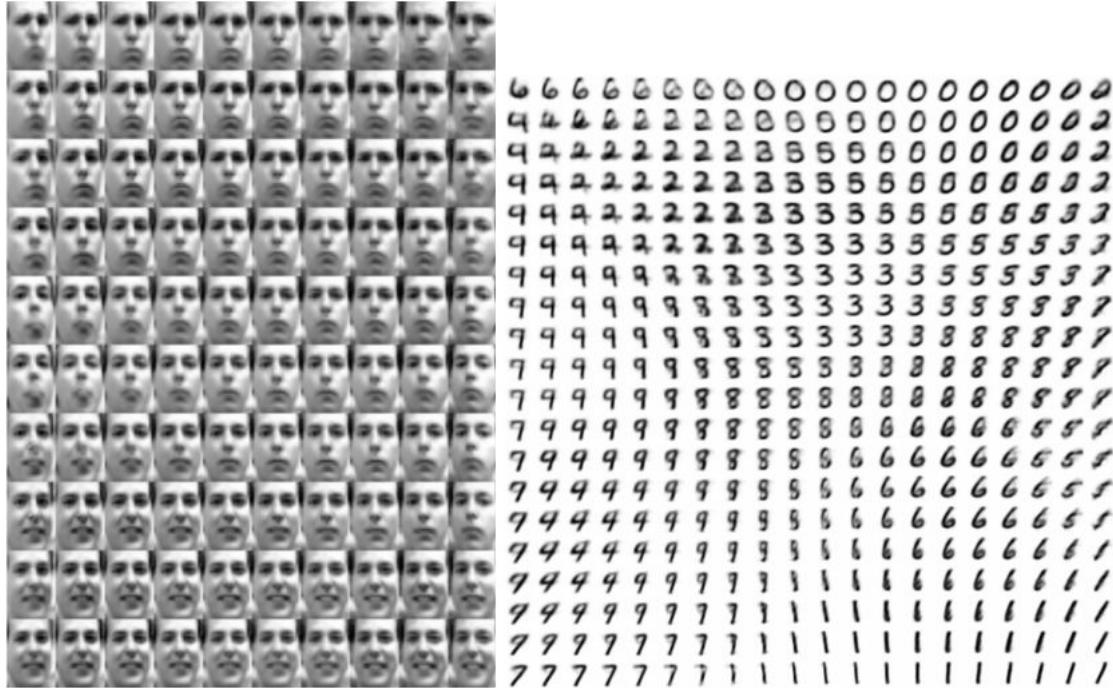


Figure 18.8: Two-dimensional representation space (for easier visualization), i.e., a Euclidean coordinate system for Frey faces (left) and MNIST digits (right), learned by a variational auto-encoder (Kingma and Welling, 2014a). Figures reproduced with permission from the authors. The images shown are not examples from the training set but images \mathbf{x} actually generated by the model $P(\mathbf{x}|\mathbf{h})$, simply by changing the 2-D “code” \mathbf{h} (each image corresponds to a different choice of “code” \mathbf{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 18.9 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 syntactically close end up having nearby embeddings.

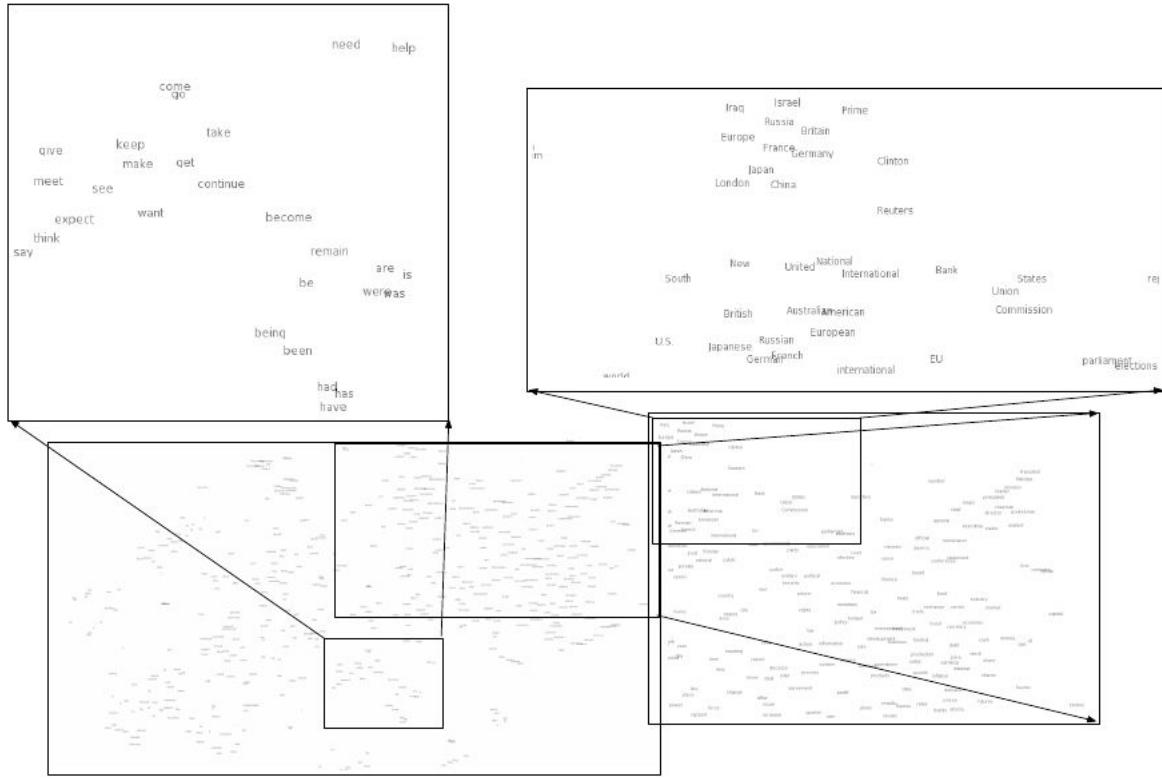


Figure 18.9: Two-dimensional representation space (for easier visualization), of English words, learned by a neural language model as in Bengio *et al.* (2001c, 2003b), with t-SNE for the non-linear dimensionality reduction from 100 to 2. Different regions are zoomed to better see the details. At the global level one can identify big clusters corresponding to part-of-speech, while locally one sees mostly semantic similarity explaining the neighborhood structure.

18.1 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 \mathbf{x} into a subspace that preserves as much information as possible about \mathbf{x} . This is illustrated in Figure 18.10. Let the encoder be

$$h = f(x) = \mathbf{W}^\top(x - \mu)$$

computing such a projection, a low-dimensional representation of h . With the auto-encoder view, we have a decoder computing the reconstruction

$$\hat{\boldsymbol{x}} = g(\boldsymbol{h}) = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}.$$

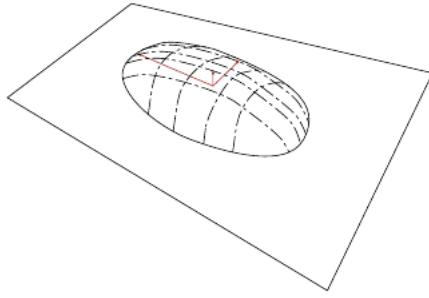


Figure 18.10: Flat Gaussian capturing probability concentration near a low-dimensional manifold. The figure shows the upper half of the “pancake” above the “manifold plane” which goes through its middle. The variance in the direction orthogonal to the manifold is very small (upward red arrow) and can 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

$$\mathbb{E}[||\mathbf{x} - \hat{\mathbf{x}}||^2]$$

correspond to $\mathbf{V} = \mathbf{W}$, $\boldsymbol{\mu} = \mathbf{b} = \mathbb{E}[\mathbf{x}]$ and the rows of \mathbf{W} form an orthonormal basis which spans the same subspace as the principal eigenvectors of the covariance matrix

$$\mathbf{C} = \mathbb{E}[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top].$$

In the case of PCA, the rows of \mathbf{W} are these eigenvectors, ordered by the magnitude of the corresponding eigenvalues (which are all real and non-negative). This is illustrated in Figure 18.11.

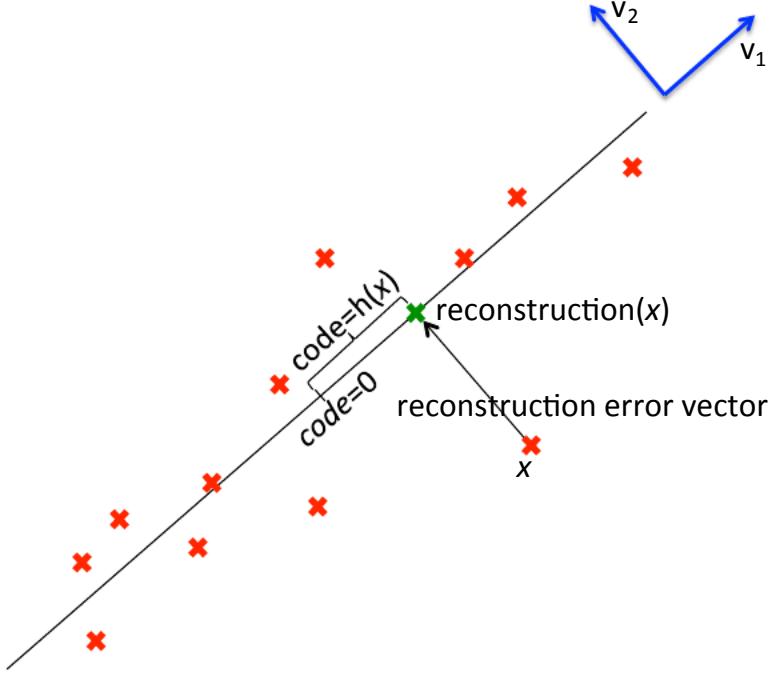


Figure 18.11: Manifold view of PCA and linear auto-encoders. The data distribution is concentrated near a manifold aligned with the leading eigenvectors (here, this is just \mathbf{v}_1) of the data covariance matrix. The other eigenvectors (here, just \mathbf{v}_2) are orthogonal to the manifold. A data point (in red, \mathbf{x}) is encoded into a lower-dimensional representation or code \mathbf{h} (here the scalar which indicates the position on the manifold, starting from $\mathbf{h} = 0$). The decoder (transpose of the encoder) maps \mathbf{h} to the data space, and corresponds to a point lying exactly on the manifold (green cross), the orthogonal projection of \mathbf{x} on the manifold. The optimal encoder and decoder minimize the sum of reconstruction errors (difference vector between \mathbf{x} and its reconstruction).

One can also show that eigenvalue λ_i of \mathbf{C} corresponds to the variance of \mathbf{x} in the direction of eigenvector \mathbf{v}_i . If $\mathbf{x} \in \mathbb{R}^D$ and $\mathbf{h} \in \mathbb{R}^d$ with $d < D$, then the optimal reconstruction error (choosing $\boldsymbol{\mu}$, \mathbf{b} , \mathbf{V} and \mathbf{W} as above) is

$$\min \mathbb{E}[|\mathbf{x} - \hat{\mathbf{x}}|^2] = \sum_{i=d+1}^D \lambda_i.$$

Hence, if the covariance has rank d , the eigenvalues λ_{d+1} to λ_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 \mathbf{h} , under orthonormal \mathbf{W} , instead of minimizing reconstruction error.

18.2 Manifold Interpretation of Sparse Coding

Sparse coding was introduced in Section 16.5.2 a linear factors generative model. It also has an interesting *manifold learning interpretation*. The codes \mathbf{h} inferred with the above equation do not fill the space in which \mathbf{h} lives. Instead, probability mass is concentrated on axis-aligned subspaces: sets of values of \mathbf{h} for which most of the axes are set at 0. We can thus decompose \mathbf{h} into two pieces of information:

- A binary pattern β 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 $\alpha \in \mathbb{R}^{N_a}$ which specifies the coordinates for each of the active dimensions.

The pattern β can be viewed as specifying an N_a -dimensional region in input space (the set of $\mathbf{x} = \mathbf{W}\mathbf{h} + \mathbf{b}$ where $h_i = 0$ if $\beta_i = 0$). That region is actually a linear manifold, an N_a -dimensional hyperplane. All those hyperplanes go through a “center” $\mathbf{x} = \mathbf{b}$. The vector α then specifies a Euclidean coordinate on that hyperplane.

Because the prior $P(\mathbf{h})$ is concentrated around 0, the probability mass of $P(\mathbf{x})$ is concentrated on the regions of these hyperplanes near $\mathbf{x} = \mathbf{b}$. Depending on the amount of reconstruction error (output variance for $P(\mathbf{x}|g(\mathbf{h}))$), 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 $\mathbf{h} \in \mathbb{R}^d$. 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 \mathbf{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 \mathbf{h} . This property is shared with other distributed representation learning algorithms described in this chapter, such as the regularized auto-encoders.

18.3 Manifold Learning via Regularized Auto-Encoders

Auto-encoders have been described in Section 16. What is their connection to manifold learning? This is what we discuss here.

We denote f the encoder function, with $\mathbf{h} = f(\mathbf{x})$ the representation of \mathbf{x} , and g the decoding function, with $\hat{\mathbf{x}} = g(\mathbf{h})$ the reconstruction of \mathbf{x} , although in some cases the encoder is a conditional distribution $q(\mathbf{h}|\mathbf{x})$ and the decoder is a conditional distribution $P(\mathbf{x}|\mathbf{h})$.

What all auto-encoders have in common, when they are prevented from simply learning the identity function for all possible input \mathbf{x} , is that training them involves a compromise between two “forces”:

1. Learning a representation \mathbf{h} of training examples \mathbf{x} such that \mathbf{x} can be approximately recovered from \mathbf{h} through a decoder. Note that this needs not be true for any \mathbf{x} , only for those that are probable under the data generating distribution.
2. Some constraint or regularization is imposed, either on the code \mathbf{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 \mathbf{x} . In the case of sparse auto-encoders (Section 16.7) the representation elements h_i are pushed towards 0. In the case of denoising auto-encoders (Section 16.8), the encoder/decoder function is encouraged to be contractive (have small derivatives). In the case of the contractive auto-encoder (Section 16.9), 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 21.8.2), a prior $\log P(\mathbf{h})$ is imposed on \mathbf{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 representations that implicitly capture a local coordinate for this manifold: only the variations tangent to the manifold around \mathbf{x} need to correspond to changes in $\mathbf{h} = f(\mathbf{x})$. Hence the encoder learns a mapping from the embedding space \mathbf{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 18.12. A one-dimensional example is illustrated in Figure 18.13, 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).

18.4 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

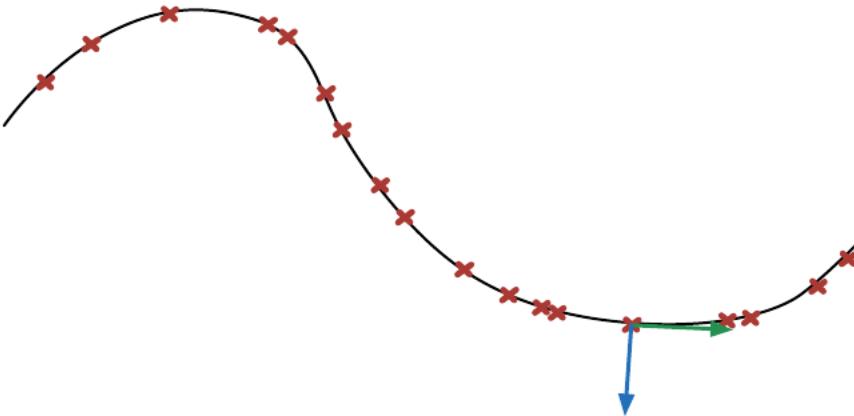


Figure 18.12: A regularized auto-encoder or a bottleneck auto-encoder has to reconcile two forces: reconstruction error (which forces it to keep enough information to distinguish training examples from each other), and a regularizer or constraint that aims at reducing its representational ability, to 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.

is derived from knowledge of the manifolds near which probability concentrates. It is assumed that we are trying to classify examples 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 \mathbf{x}_1 and \mathbf{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 \mathbf{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 $f(\mathbf{x})$ 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 $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ to be orthogonal to the known manifold tangent vectors \mathbf{v}_i at \mathbf{x} , or equivalently that the directional derivative of f at \mathbf{x} in the directions

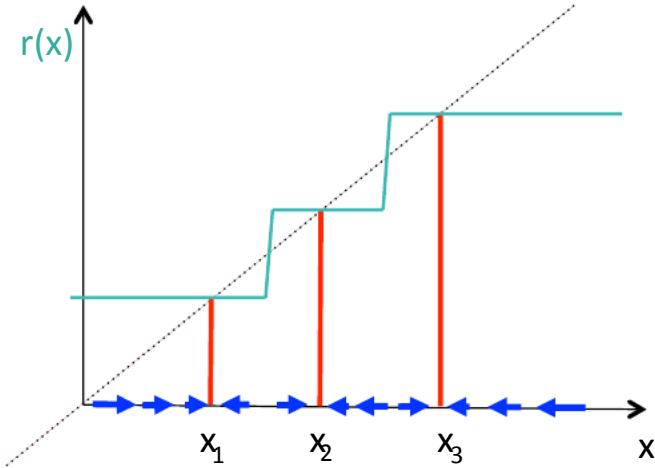


Figure 18.13: If the auto-encoder learns to be contractive around the data points, with the reconstruction pointing towards the nearest data points, it captures the manifold structure of the data. This is a 1-dimensional version of Figure 18.12. The denoising auto-encoder explicitly tries to make the derivative of the reconstruction function $r(\mathbf{x})$ small around the data points. The contractive auto-encoder does the same thing for the encoder. Although the derivative of $r(\mathbf{x})$ 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 ($r(\mathbf{x}) \approx \mathbf{x}$ for data points \mathbf{x}) and contraction (small derivatives of $r(\mathbf{x})$ near data points).

v_i be small:

$$\text{regularizer} = \lambda \sum_i \left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \cdot \mathbf{v}_i \right)^2. \quad (18.1)$$

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. Tanget-Prop has been used not just for supervised learning (Simard *et al.*, 1992) but also in the context of reinforcement learning (Thrun, 1995).

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 16.9, auto-encoders in general, and contractive auto-encoders especially well, learn a representation \mathbf{h} that is most sensitive to the factors of variation present in the data \mathbf{x} , so that the leading singular vectors of $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$ correspond to the estimated tangent vectors. As illustrated in Figure 16.10, 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

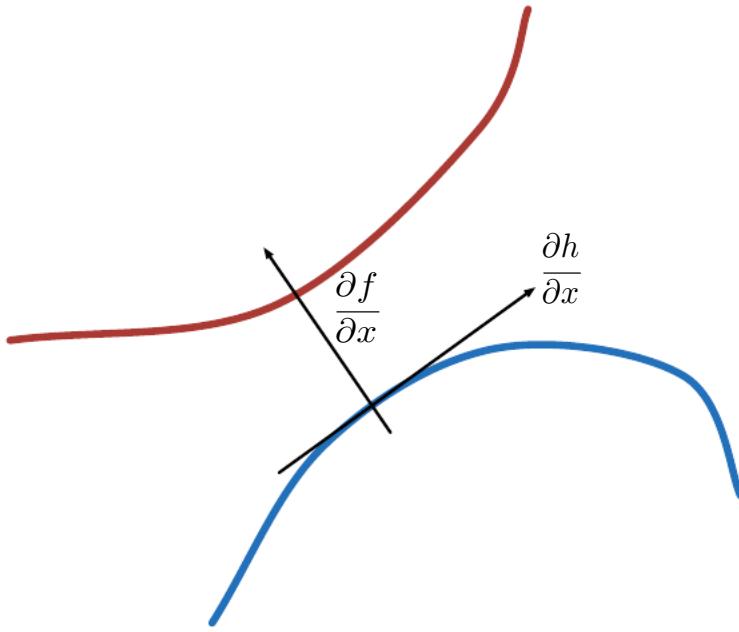


Figure 18.14: Illustration of the main idea of the tangent-prop algorithm ([Simard *et al.*, 1992](#)) and manifold tangent classifier ([Rifai *et al.*, 2011d](#)), which both regularize the classifier output function $f(\mathbf{x})$ (e.g. estimating conditional class probabilities given the input) so as to make it invariant to the local directions of variations $\frac{\partial h}{\partial \mathbf{x}}$ (manifold tangent directions). This can be achieved by penalizing the magnitude of the dot product of all the rows of $\frac{\partial h}{\partial \mathbf{x}}$ (the tangent directions) with all the rows of $\frac{\partial f}{\partial \mathbf{x}}$ (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 $\mathbf{h}(\mathbf{x})$ being the learned representation of the input \mathbf{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.

learning (2) use these tangents to regularize a neural net classifier as in Tangent Prop (Eq. 18.1).

Chapter 19

Confronting the Partition Function

TODO– make sure the book explains asymptotic consistency somewhere, add links to it here

In chapter 14.2.2 we saw that many probabilistic models (commonly known as undirected graphical models) are defined by an unnormalized probability distribution $\tilde{p}(\mathbf{x}; \theta)$ and a partition function $Z(\theta)$ such that $p(\mathbf{x}; \theta) = \frac{1}{Z}\tilde{p}(\mathbf{x}; \theta)$ 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 21, 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 confront the challenge of intractable partition functions head on. In this chapter, we describe techniques used for training and evaluating models that have intractable partition functions.

19.1 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 undirected 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, monitor training performance, and compare it to other models.

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 IID test dataset of size N_{test} : $D_{\text{test}} = \{\mathbf{x}_i^{(t)}\}_{i=1}^{N_{\text{test}}}$ 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:

$$\begin{aligned} \sum_t \ln p_A(\boldsymbol{x}^{(t)}; \theta_A) - \sum_t \ln p_B(\boldsymbol{x}^{(t)}; \theta_B) &> 0 \\ \sum_t (\ln \tilde{p}_A(\boldsymbol{x}^{(t)}; \theta_A) - \ln Z(\theta_A)) - \sum_t (\ln \tilde{p}_B(\boldsymbol{x}^{(t)}; \theta_B) - \ln Z(\theta_B)) &> 0 \\ \sum_t (\ln \tilde{p}_A(\boldsymbol{x}^{(t)}; \theta_A) - \ln \tilde{p}_B(\boldsymbol{x}^{(t)}; \theta_B)) - N_{test} \ln Z(\theta_A) + N_{test} \ln Z(\theta_B) &> 0 \\ \sum_t \left(\ln \frac{\tilde{p}_A(\boldsymbol{x}^{(t)}; \theta_A)}{\tilde{p}_B(\boldsymbol{x}^{(t)}; \theta_B)} \right) - N_{test} \ln \frac{Z(\theta_A)}{Z(\theta_B)} &> 0. \end{aligned}$$

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 $p_1(\boldsymbol{x})$, the partition function is defined as

$$Z_1 = \int \tilde{p}_1(\boldsymbol{x}) d\boldsymbol{x} \quad (19.1)$$

where the integral is over the domain of \boldsymbol{x} . Of course, in the case of discrete \boldsymbol{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 $p_0(\boldsymbol{x})$, from which we can *sample* and *evaluate* both its partition function Z_0 , and its

unnormalized distribution $\tilde{p}_0(\mathbf{x})$.

$$\begin{aligned}
Z_1 &= \int \tilde{p}_1(\mathbf{x}) d\mathbf{x} \\
&= \int \frac{p_0(\mathbf{x})}{\tilde{p}_0(\mathbf{x})} \tilde{p}_1(\mathbf{x}) d\mathbf{x} \\
&= Z_0 \int p_0(\mathbf{x}) \frac{\tilde{p}_1(\mathbf{x})}{\tilde{p}_0(\mathbf{x})} d\mathbf{x} \\
\frac{Z_1}{Z_0} &\approx \sum_{k=1}^K \frac{\tilde{p}(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t. : } \mathbf{x}^{(k)} \sim p_0
\end{aligned} \tag{19.2}$$

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. 19.2. 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 .

19.1.1 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 $KL(p_0||p_1)$ 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 $p_{\eta_0}, \dots, p_{\eta_n}$, with $0 = \eta_0 < \eta_1 < \dots < \eta_{n-1} < \eta_n = 1$ 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

$$\begin{aligned}\frac{Z_1}{Z_0} &= \frac{Z_1}{Z_0} \frac{Z_{\eta_1}}{Z_{\eta_1}} \dots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-1}}} \\ &= \frac{Z_{\eta_1}}{Z_0} \frac{Z_{\eta_2}}{Z_{\eta_1}} \dots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-2}}} \frac{Z_1}{Z_{\eta_{n-1}}} \\ &= \prod_{j=0}^{n-1} \frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}\end{aligned}\tag{19.3}$$

Provided the distributions p_{η_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 $p_{\eta_1} \dots p_{\eta_{n-1}}$. 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 :

$$p_{\eta_j} \propto p_1^{\eta_j} p_0^{1-\eta_j}\tag{19.4}$$

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:

$$p_{\eta_j}(\mathbf{x}) = \int p_{\eta_j}(\mathbf{x}') T_{\eta_j}(\mathbf{x}', \mathbf{x}) d\mathbf{x}'\tag{19.5}$$

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 $\mathbf{x}_{\eta_1}^{(k)} \sim p_0(\mathbf{x})$
 - Sample
 $v\mathbf{x}_{\eta_2}^{(k)} \sim T_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)}, \mathbf{x})$
 - ...
 - Sample $\mathbf{x}_{\eta_{n-1}}^{(k)} \sim T_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}}^{(k)}, \mathbf{x})$
 - Sample $\mathbf{x}_{\eta_n}^{(k)} \sim T_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}^{(k)}, \mathbf{x})$

- 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. 19.3.

$$w^{(k)} = \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})} \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\mathbf{x}_{\eta_1}^{(k)})} \dots \frac{\tilde{p}_{\eta_l}(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_{l-1}}(\mathbf{x}_{\eta_{l-1}}^{(k)})} \quad (19.6)$$

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 defined and the importance weights given in Eq. 19.6, the estimate of the ratio of partition functions is given by:

$$\frac{Z_1}{Z_0} \approx \frac{1}{K} \sum_{k=1}^K w^{(k)} \quad (19.7)$$

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:

$$\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) = \tilde{p}_1(\mathbf{x}_1) \tilde{T}_{\eta_{n-1}}(\mathbf{x}_1, \mathbf{x}_{\eta_{n-1}}) \tilde{T}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-1}}, \mathbf{x}_{\eta_{n-2}}) \dots \tilde{T}_{\eta_1}(\mathbf{x}_{\eta_2}, \mathbf{x}_{\eta_1}) \quad (19.8)$$

where \tilde{T}_a is the reverse of the transition operator defined by T_a (via an application of Bayes' rule):

$$\tilde{T}_a(\mathbf{x}, \mathbf{x}') = \frac{p_a(\mathbf{x}')}{p_a(\mathbf{x})} T_a(\mathbf{x}', \mathbf{x}) = \frac{\tilde{p}_a(\mathbf{x}')}{\tilde{p}_a(\mathbf{x})} T_a(\mathbf{x}', \mathbf{x}). \quad (19.9)$$

Plugging the above into the expression for the joint distribution on the extended state space given in Eq. 19.8, we get:

$$\begin{aligned} & \tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \\ &= \tilde{p}_1(\mathbf{x}_1) \frac{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_1)} T_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \frac{\tilde{p}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}})}{\tilde{p}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-1}})} T_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}}, \mathbf{x}_{\eta_{n-1}}) \dots \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1})}{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_2})} T_{\eta_1}(\mathbf{x}_{\eta_1}, \mathbf{x}_{\eta_2}) \\ &= \frac{\tilde{p}_1(\mathbf{x}_1)}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_1)} T_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \frac{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}})}{\tilde{p}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-1}})} T_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}}, \mathbf{x}_{\eta_{n-1}}) \dots \frac{\tilde{p}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}})}{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_2})} T_{\eta_1}(\mathbf{x}_{\eta_1}, \mathbf{x}_{\eta_2}) \tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}) \end{aligned} \quad (19.10)$$

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:

$$q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) = p_0(\mathbf{x}_{\eta_1}) T_{\eta_1}(\mathbf{x}_{\eta_1}, \mathbf{X}_{\eta_2}) \dots T_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}, \mathbf{X}_1) \quad (19.11)$$

We have a joint distribution on the extended space given by Eq. 19.10. Taking $q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)$ as the proposal distribution on the extended state space from which we will draw samples, it remains to determine the importance weights:

$$w^{(k)} = \frac{\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)}{q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)} = \frac{\tilde{p}_1(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}^{(k)})} \dots \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\mathbf{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})} \quad (19.12)$$

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).

19.1.2 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}_* :

$$\frac{Z_1}{Z_0} \approx \sum_{k=1}^K \frac{\tilde{p}_*(x_0^{(k)})}{\tilde{p}_0(x_0^{(k)})} \Bigg/ \sum_{k=1}^K \frac{\tilde{p}_*(x_1^{(k)})}{\tilde{p}_1(x_1^{(k)})} \quad (19.13)$$

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, $KL(p_0||p_1)$) to be much larger than with standard importance sampling.

It can be shown that 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

19.1.3 Extensions

Linked importance sampling Both AIS and bridge sampling has their advantages. If $KL(p_0||p_1)$ 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.

19.2 Stochastic Maximum Likelihood and Contrastive Divergence

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 θ is one of the parameters of $p(\mathbf{x})$. These derivatives are given simply by

$$\frac{\partial}{\partial \theta} \log p(\mathbf{x}) = \frac{\partial}{\partial \theta} (\log \tilde{p}(\mathbf{x}) - \log Z).$$

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 $\exp(\log \tilde{p}(\mathbf{x}))$ for $\tilde{p}(\mathbf{x})$:

$$= \frac{\sum_{\mathbf{x}} \frac{\partial}{\partial \theta} \exp(\log \tilde{p}(\mathbf{x}))}{Z}$$

$$\begin{aligned}
&= \frac{\sum_{\mathbf{x}} \exp(\log \tilde{p}(\mathbf{x})) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x})}{Z} \\
&= \frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x})}{Z} \\
&= \sum_{\mathbf{x}} p(\mathbf{x}) \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x}) \\
&= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x}).
\end{aligned}$$

This derivation made use of summation over discrete \mathbf{x} , but a similar result applies using integration over continuous \mathbf{x} . In the continuous version of the derivation, we use Leibniz's rule for differentiation under the integral sign to obtain the identity

$$\frac{\partial}{\partial \theta} \int \tilde{p}(\mathbf{x}) d\mathbf{x} = \int \frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x}) d\mathbf{x}.$$

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

$$\frac{\partial}{\partial \theta} \log Z = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \frac{\partial}{\partial \theta} \log \tilde{p}(\mathbf{x}) \quad (19.14)$$

is the basis for a variety of Monte Carlo methods for approximately maximizing the likelihood of models with intractable partition functions.

The naive way of implementing equation 19.14 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 19.1. 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.

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. 19.1 illustrates this process. The two forces correspond to maximizing $\log \tilde{p}$ and minimizing $\log Z$. Terms of the approximate gradient intended to maximize $\log \tilde{p}$ are referred to as the *positive phase* and terms of the approximate gradient intended to minimize $\log Z$ are known as the *negative phase*. In this chapter, we assume the positive phase is tractable and may be performed exactly, but other chapters, especially chapter 20 deal with intractable positive phases. In this chapter, we present several approximations to the negative phase. Each of these approximations can be understood

¹In measure theoretic terms, the conditions are: (i) \tilde{p} must be a Lebesgue-integrable function of \mathbf{x} for every value of θ ; (ii) $\frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x})$ must exist for all θ and almost all \mathbf{x} ; (iii) There exists an integrable function $R(\mathbf{x})$ that bounds $\frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x})$ (i.e. such that $|\frac{\partial}{\partial \theta} \tilde{p}(\mathbf{x})| \leq R(\mathbf{x})$ for all θ and almost all \mathbf{x}).

Algorithm 19.1 A naive MCMC algorithm for maximizing the log likelihood with an intractable partition function using gradient ascent.

Set ϵ , 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 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 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 $\{\tilde{\mathbf{x}}^{(1)}, \dots, \tilde{\mathbf{x}}^{(m)}\}$ 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**
 $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{gibbs_update}(\tilde{\mathbf{x}}^{(j)})$
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

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 [20.6](#), 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 [19.1](#). 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

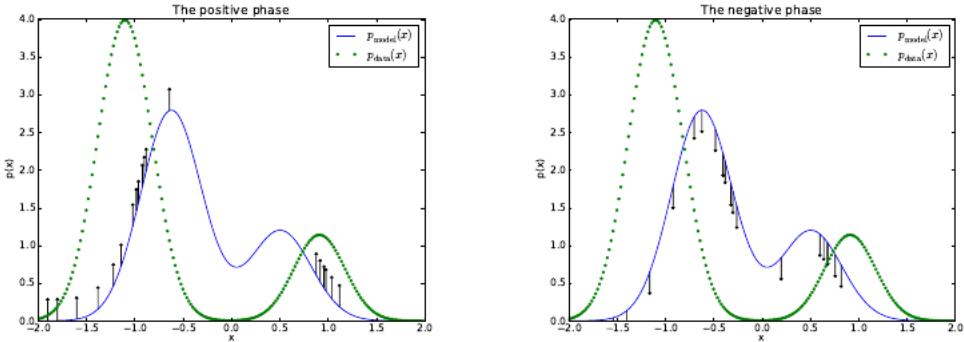


Figure 19.1: The view of Algorithm 19.1 as having a “positive phase” and “negative phase”. Left) In the positive phase, we sample points from the data distribution, and push up on their unnormalized probability. This means points that are likely in the data get pushed up on more. Right) In the negative 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.

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 19.2. 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.

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. 19.2 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

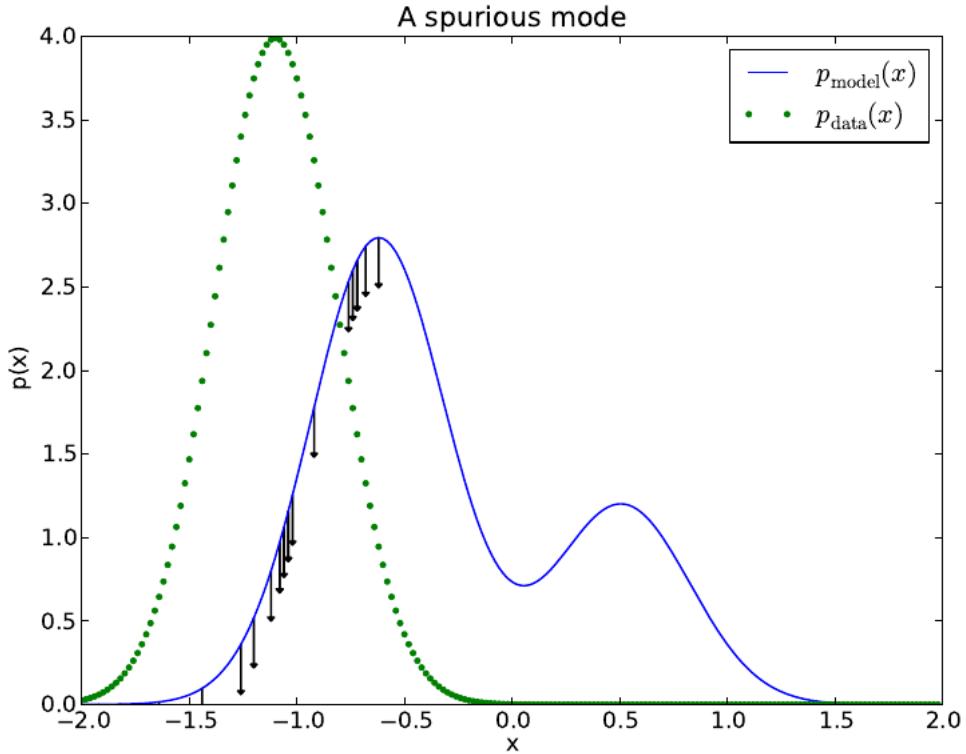


Figure 19.2: An illustration of how the negative phase of contrastive divergence (Algorithm 19.2) can fail to suppress spurious modes. A spurious mode is a mode that is present in the model distribution but absent in the data distribution. Because contrastive divergence initializes its Markov chains 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.

Algorithm 19.2 The contrastive divergence algorithm, using gradient ascent as the optimization procedure.

Set ϵ , 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 initialized from p_{data} . Perhaps 1-20 to train an RBM on a small image patch.
while Not converged **do**
 Sample a minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from the training set.
 $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log \tilde{p}(\mathbf{x}^{(i)}; \theta)$
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
 $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \theta)$
 $\theta \leftarrow \theta + \epsilon \mathbf{g}$
end while

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 20 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 divergence* (PCD, or PCD- k to indicate the use of k Gibbs steps per update) in the deep learning community (Tieleman, 2008). See Algorithm 19.3. 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 ϵ 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

Algorithm 19.3 The stochastic maximum likelihood / persistent contrastive divergence algorithm using gradient ascent as the optimization procedure.

Set ϵ , the step size, to a small positive number
 Set k , the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{x}; \boldsymbol{\theta} + \epsilon \mathbf{g})$ to burn in, starting from samples from $p(\mathbf{x}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch, or 5-50 for a more complicated model like a DBM.
 Initialize a set of m samples $\{\tilde{\mathbf{x}}^{(1)}, \dots, \tilde{\mathbf{x}}^{(m)}\}$ 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 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 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 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 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.

19.3 Pseudolikelihood

Monte Carlo approximations to the partition function and its gradient confront the partition function head on. 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.

$$p(\mathbf{a} | \mathbf{b}) = \frac{p(\mathbf{a}, p(\mathbf{b}))}{p(\mathbf{b})} = \frac{p(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} p(\mathbf{a}, \mathbf{b}, \mathbf{c})} = \frac{\tilde{p}(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} \tilde{p}(\mathbf{a}, \mathbf{b}, \mathbf{c})}.$$

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,

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2 | x_1) + \cdots + p(x_n | \mathbf{x}_{1:n-1}).$$

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:

$$\sum_{i=1}^n \log p(x_i | \mathbf{x}_{-i}).$$

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 $S^{(i)}$, $i = 1, \dots, m$ of indices variables that appear together on the left side of the conditioning bar. In the extreme case of $m = 1$ and $S^{(1)} = 1, \dots, n$ 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 20.4. 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.

19.4 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:

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\boldsymbol{x}} \| \nabla_{\boldsymbol{x}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) - \nabla_{\boldsymbol{x}} \log p_{\text{data}}(\boldsymbol{x}) \|_2^2.$$

Because the $\nabla_{\boldsymbol{x}} Z = 0$, 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_{data} . Fortunately, minimizing $J(\boldsymbol{\theta})$ turns out to be equivalent to minimizing

$$\tilde{J}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n \left(\frac{\partial^2}{\partial x_j^2} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) + \frac{1}{2} \left(\frac{\partial}{\partial x_i} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) \right)^2 \right)$$

where $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ is the training set and n is the dimensionality of \boldsymbol{x} .

Because score matching requires taking derivatives with respect to \boldsymbol{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 $\log \tilde{p}(\boldsymbol{x})$ and its derivatives directly. It is not compatible with methods that only provide a lower bound on $\log \tilde{p}(\boldsymbol{x})$, 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:

$$J^{(\text{RM})}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n \left(\frac{1}{1 + \frac{p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})}{p_{\text{model}}(f(\boldsymbol{x}^{(i)}, j); \boldsymbol{\theta})}} \right)^2$$

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.

19.5 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_{data} . This is especially useful because in practice we usually do not have access to the true p_{data} but rather only an empirical distribution defined by samples from it. Any consistent estimator will, given enough capacity, make p_{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).

19.6 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 Hyvärinen, 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 θ , the noise contrastive estimation procedure treats c as just another parameter and estimates θ 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 $p_{\text{joint-model}}(y = 1) = \frac{1}{2}$, $p_{\text{joint-model}}(\mathbf{x} | y = 1) = p_{\text{model}}(\mathbf{x})$, and $p_{\text{joint-model}}(\mathbf{x} | y = 0) = p_{\text{noise}}(\mathbf{x})$. 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, $p_{\text{train}}(y = 1) = \frac{1}{2}$, $p_{\text{train}}(\mathbf{x} | y = 1) = p_{\text{data}}(\mathbf{x})$, and $p_{\text{train}}(\mathbf{x} | y = 0) = p_{\text{noise}}(\mathbf{x})$.

We can now just use standard maximum likelihood learning on the *supervised* learning problem of fitting $p_{\text{joint-model}}$ to p_{train} :

$$\theta, c = \arg \max_{\theta, c} \mathbb{E}_{\mathbf{x}, y \sim p_{\text{train}}} \log p_{\text{joint-model}}(y | \mathbf{x}).$$

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:

$$p_{\text{joint-model}}(y = 1 | \mathbf{x}) = \frac{p_{\text{model}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})}$$

²NCE is also applicable to problems with a tractable partition function, where there is no need to introduce the extra parameter c . However, it has generated the most interest as a means of estimating models with difficult partition functions.

$$\begin{aligned}
&= \frac{1}{1 + \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}} \\
&= \frac{1}{1 + \exp\left(\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right)} \\
&= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right) \\
&= \sigma(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})).
\end{aligned}$$

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_{model} has learned the basic marginal statistics. Imagine learning a model of images of faces, using unstructured Gaussian noise as p_{noise} . If p_{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_{noise} must be easy to evaluate and easy to sample from can be overly restrictive. When p_{noise} is simple, most samples are likely to be too obviously distinct from the data to force p_{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 $p_{\text{joint_model}}(y = 1 | \mathbf{x})$, but it can only be used to construct an upper bound on $p_{\text{joint_model}}(y = 0 | \mathbf{x})$, which appears in half the terms of the NCE objective. Likewise, a lower bound on p_{noise} is not useful, because it provides only an upper bound on $p_{\text{joint_model}}(y = 1 | \mathbf{x})$.

TODO– put herding in this chapter? and if not, where to put it?

TODO– cite the Bregman divergence paper?

Chapter 20

Approximate inference

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 \mathbf{v} and a set of latent variables \mathbf{h} . The challenge of inference usually refers to the difficult problem of computing $p(\mathbf{h} | \mathbf{v})$ 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 $p(\mathbf{h} | \mathbf{v})$ 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. 20.1 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.

20.1 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 \mathbf{v} and latent variables \mathbf{h} . We would like to compute the log probability of the observed data, $\log p(\mathbf{v}; \boldsymbol{\theta})$. Sometimes it is too difficult to compute $\log p(\mathbf{v}; \boldsymbol{\theta})$ if it is costly to marginalize out \mathbf{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

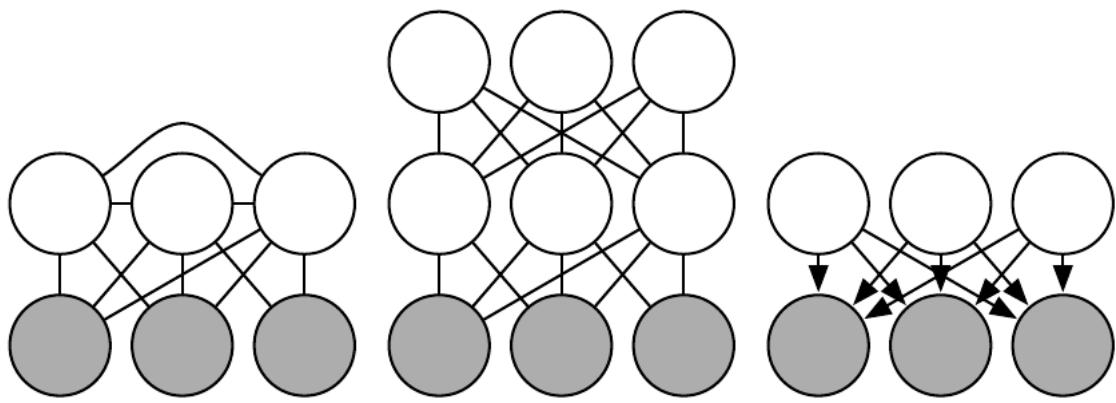


Figure 20.1: Intractable inference problems are usually the result of interactions between latent variables in a structured graphical model. These can be due to direct edges, or due to paths that are activated when the child of a V-structure is observed. Left) A semi-restricted Boltzmann machine with 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 every 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

places

$$\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, \mathbf{q}) = \log p(\mathbf{v}; \boldsymbol{\theta}) - D_{\text{KL}}(q(\mathbf{h}) \| p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta}))$$

where q is an arbitrary probability distribution over \mathbf{h} .

It is straightforward to see that this is a lower bound on $\log p(\mathbf{v})$:

$$\begin{aligned} \ln p(\mathbf{v}) &= \ln p(\mathbf{v}) + \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) \\ &= \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) + \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \log p(\mathbf{v}) \\ &= \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \left[\ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \ln p(\mathbf{v}) \right] \\ &= \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{p(\mathbf{v})q(\mathbf{h} | \mathbf{v})} \right) \\ &= \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{v}, \mathbf{h})}{q(\mathbf{h} | \mathbf{v})} \right) - \sum_{\mathbf{h}} q(\mathbf{h} | \mathbf{v}) \ln \left(\frac{p(\mathbf{h} | \mathbf{v})}{q(\mathbf{h} | \mathbf{v})} \right) \\ &= \mathcal{L}(q) + \text{KL}(q \| p) \end{aligned}$$

Because the difference $\log p(\mathbf{v})$ and $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, \mathbf{q})$ 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 $p(\mathbf{h} | \mathbf{v})$.

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:

$$\begin{aligned} \mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, \mathbf{q}) &= \log p(\mathbf{v}; \boldsymbol{\theta}) - D_{\text{KL}}(q(\mathbf{h}) \| p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta})) \\ &= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \frac{\log q(\mathbf{h})}{\log p(\mathbf{h} | \mathbf{v})} \\ &= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \frac{\log q(\mathbf{h})}{\log \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})}} \\ &= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} [\log q(\mathbf{h}) - \log p(\mathbf{h}, \mathbf{v}) + \log p(\mathbf{v})] \\ &= -\mathbb{E}_{\mathbf{h} \sim q} [\log q(\mathbf{h}) - \log p(\mathbf{h}, \mathbf{v})]. \end{aligned}$$

This yields the more canonical definition of the evidence lower bound,

$$\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, \mathbf{q}) = \mathbb{E}_{\mathbf{h} \sim q} [\log p(\mathbf{h}, \mathbf{v})] + H(q). \quad (20.1)$$

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 $p(\mathbf{h} | \mathbf{v})$ the distribution q will be. This determines how good of an approximation \mathcal{L} will be for $\log p(\mathbf{v})$.

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 computation cost by using an imperfect optimization procedure, or by using a perfect optimization procedure over a restricted family of q distributions.

20.2 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 $q(\mathbf{h}^{(i)}) = p(\mathbf{h}^{(i)} | \mathbf{v}^{(i)}; \boldsymbol{\theta})$ for all indices i of the training examples $\mathbf{v}^{(i)}$ 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 $\boldsymbol{\theta}$ then $p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta})$ will change but $q(\mathbf{h})$ will not.
- The *M-step* (Maximization step): Completely or partially maximize $\sum_i \mathcal{L}(\mathbf{v}^{(i)}, \boldsymbol{\theta}, q)$ 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 $\log p(\mathbf{v})$ 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

20.3 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 $\mathbf{v} \in \mathbb{R}^n$ with latent variables $\mathbf{h} \in \mathbb{R}^m$. We can use a prior to encourage our latent code variables to be sparse:

$$p(\mathbf{h}) = \text{TODO}.$$

We can define the visible units to be Gaussian with an affine transformation from the code to the mean of the Gaussian:

$$\mathbf{v} \sim \mathcal{N}(\mathbf{v} | \boldsymbol{\mu} + \mathbf{W}\mathbf{h}, \boldsymbol{\beta}^{-1})$$

where $\boldsymbol{\beta}$ is a diagonal precision matrix to maintain tractability.

Computing $p(\mathbf{h} | \mathbf{v})$ 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:

$$\mathbf{h}^* = \arg \max p(\mathbf{h} | \mathbf{v}).$$

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 \mathbf{h} given \mathbf{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 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 \mathbf{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.

20.4 Variational Inference and Learning

One common difficulty in probabilistic modeling is that the posterior distribution $p(\mathbf{h} | \mathbf{v})$ is infeasible to compute for many models with hidden variables \mathbf{h} and visible variables \mathbf{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 $\mathbf{v} \in \mathbb{R}^n$ 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 $\mathbf{h} \in \{0, 1\}^m$:

$$p(h_i = 1) = \sigma(b_i)$$

$$p(\mathbf{v} | \mathbf{h}) = \mathcal{N}(\mathbf{v} | \mathbf{W}\mathbf{h}, \beta^{-1})$$

where \mathbf{b} is a learnable set of biases, \mathbf{W} is a learnable weight matrix, and $\boldsymbol{\beta}$ is a learnable, 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:

$$\begin{aligned} & \frac{\partial}{\partial b_i} \log p(\mathbf{v}) \\ &= \frac{\frac{\partial}{\partial b_i} p(\mathbf{v})}{p(\mathbf{v})} \\ &= \frac{\frac{\partial}{\partial b_i} \sum_{\mathbf{h}} p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\ &= \frac{\frac{\partial}{\partial b_i} \sum_{\mathbf{h}} p(\mathbf{h}) p(\mathbf{v} | \mathbf{h})}{p(\mathbf{v})} \\ &= \frac{\sum_{\mathbf{h}} p(\mathbf{v} | \mathbf{h}) \frac{\partial}{\partial b_i} p(\mathbf{h})}{p(\mathbf{v})} \\ &= \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\frac{\partial}{\partial b_i} p(\mathbf{h})}{p(\mathbf{h})} \\ &= \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\frac{\partial}{\partial b_i} p(\mathbf{h})}{p(\mathbf{h})} \\ &= \mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} | \mathbf{v})} \frac{\partial}{\partial b_i} \log p(\mathbf{h}). \end{aligned}$$

This requires computing expectations with respect to $p(\mathbf{h} | \mathbf{v})$. Unfortunately, $p(\mathbf{h} | \mathbf{v})$ is a complicated distribution. See Fig. 20.2 for the graph structure of $p(\mathbf{h}, \mathbf{v})$ and $p(\mathbf{h} | \mathbf{v})$. 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 $q(\mathbf{h})$ to approximate the true, complicated posterior $p(\mathbf{h} | \mathbf{v})$. 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

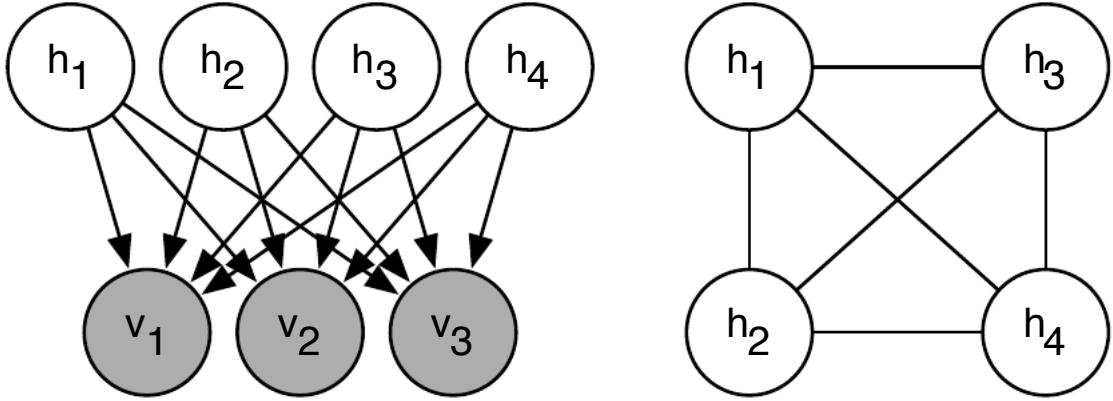


Figure 20.2: The graph structure of a binary sparse coding model with four hidden units. Left) The graph structure of $p(\mathbf{h}, \mathbf{v})$. Note that the edges are directed, and that every two hidden units co-parents of every visible unit. Right) The graph structure of $p(\mathbf{h} | \mathbf{v})$. In order to account for the active paths between co-parents, the posterior distribution needs an edge between all of the hidden units.

20.4.1 Discrete Latent Variables

TODO– BSC example

20.4.2 Calculus of Variations

Many machine learning techniques are based on minimizing a function $J(\boldsymbol{\theta})$ by finding the input vector $\boldsymbol{\theta} \in \mathbb{R}^n$ 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 $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0$. In some cases, we actually want to solve for a function $f(\mathbf{x})$, 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 individual values of the function $f(\mathbf{x})$. The functional derivative of the functional J with respect to the value of the function f at point \mathbf{x} is denoted $\frac{\delta}{\delta f(\mathbf{x})} J$.

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 $f(\mathbf{x})$ and differentiable functions $g(y, \mathbf{x})$ with continuous derivatives, that

$$\frac{\delta}{\delta f(\mathbf{x})} \int g(f(\mathbf{x}), \mathbf{x}) d\mathbf{x} = \frac{\partial}{\partial y} g(f(\mathbf{x}), \mathbf{x}). \quad (20.2)$$

To gain some intuition for this identity, one can think of $f(\mathbf{x})$ as being a vector with uncountably many elements, indexed by a real vector \mathbf{x} . In this (somewhat incomplete

view), the identity providing the functional derivatives is the same as we would obtain for a vector $\theta \in \mathbb{R}^n$ indexed by positive integers:

$$\frac{\partial}{\partial \theta} \sum_j g(\theta_j, j) = \frac{\partial}{\partial \theta_i} g(\theta_i, i).$$

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 $x \in \mathbb{R}$ that has maximal Shannon entropy. Recall that the entropy of a probability distribution $p(x)$ is defined as

$$H[p] = -\mathbb{E}_x \log p(x).$$

For continuous values, the expectation is an integral:

$$H[p] = - \int p(x) \log p(x) dx.$$

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 σ^2 . 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 μ . The Lagrangian functional for this optimization problem is

$$\begin{aligned} \mathcal{L}[p] &= \lambda_1 \left(\int p(x) dx - 1 \right) + \lambda_2 (\mathbb{E}[x] - \mu) + \lambda_3 (\mathbb{E}[(x - \mu)^2] - \sigma^2) + H[p] \\ &= \int (\lambda_1 p(x) + \lambda_2 p(x)x + \lambda_3 p(x)(x - \mu)^2 - p(x) \log p(x)) dx - \lambda_1 - \mu \lambda_2 - \sigma^2 \lambda_3. \end{aligned}$$

To minimize the Lagrangian with respect to p , we set the functional derivatives equal to 0:

$$\forall x, \frac{\delta}{\delta p(x)} \mathcal{L} = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0.$$

This condition now tells us the functional form of $p(x)$. By algebraically re-arranging the equation, we obtain

$$p(x) = \exp(-\lambda_1 - \lambda_2 x + \lambda_3 (x - \mu)^2 + 1).$$

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 λ values to ensure that all of our constraints are satisfied. We are free to choose any λ values, because the gradient of the Lagrangian with respect to the λ variables is zero so long as the constraints are satisfied. To satisfy all of the constraints, we may set $\lambda_1 = \log \sigma \sqrt{2\pi}$, $\lambda_2 = 0$, and $\lambda_3 = -\frac{1}{2\sigma^2}$ to obtain

$$p(x) = \mathcal{N}(x | \mu, \sigma^2).$$

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.

20.4.3 Continuous Latent Variables

TODO: Gaussian example from IG's thesis? TODO: S3C example

20.5 Stochastic Inference

TODO: Charlie Tang's SFNNs? Is there anything else where sampling-based inference actually gets used?

20.6 Learned Approximate Inference

TODO: wake-sleep algorithm

In chapter 19.2 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 $p(\mathbf{h}, \mathbf{v})$ which can be used to train an inference network to predict \mathbf{h} given \mathbf{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 ***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 21

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 14, 19 and 20. 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 14. Others can not easily be described in terms of factors, but represent probability distributions nonetheless.

21.1 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. 21.1 for some examples. In particular, Fig. 21.1a 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.

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 random variables collectively as \mathbf{h} , with the j th random elements as h_j .

The restricted Boltzmann machine is an energy-based model, which means that the joint probability distribution is fully-specified by its energy function:

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp \{-E(\mathbf{v}, \mathbf{h})\}.$$

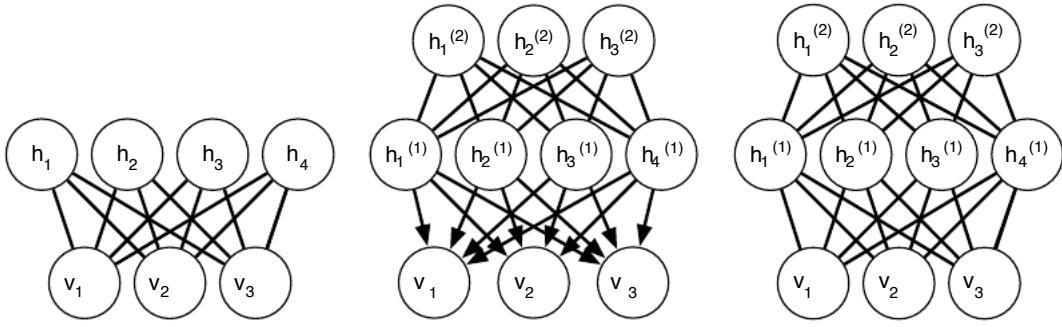


Figure 21.1: Examples of models that may be built with restricted Boltzmann machines.
a) The restricted Boltzmann machine itself is an undirected graphical model based on a bipartite graph. There are no connections among the visible units, nor any connections among the hidden units. Typically every visible 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.

Where $E(\mathbf{v}, \mathbf{h})$ is the energy function that parametrizes the relationship between the visible and hidden variables:

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{v}^\top \mathbf{W} \mathbf{h}, \quad (21.1)$$

and the Z is the normalizing constant known as the partition function:

$$Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\}.$$

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 partition function's intractability. In the case of restricted Boltzmann machines, there is actually a hardness result, proven by [Long and Servedio \(2010\)](#).

21.1.1 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 \mathbf{v}, \mathbf{h} is generally not available). However, due to the bipartite graph structure, the restricted Boltzmann machine has the very special property that its conditional distributions $P(\mathbf{h} | \mathbf{v})$ and $P(\mathbf{v} | \mathbf{h})$ 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 (CITE), speech processing (CITE) and natural language processing (CITE).

Deriving the conditional distributions from the joint distribution is straightforward.

$$\begin{aligned} p(\mathbf{h} | \mathbf{v}) &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\ &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\ &= \frac{1}{p(\mathbf{v})} \frac{1}{Z} \exp \left\{ \mathbf{b}^\top \mathbf{v} + \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h} \right\} \\ &= \frac{1}{Z'} \exp \left\{ \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h} \right\} \\ &= \frac{1}{Z'} \exp \left\{ \sum_{j=1}^n c_j h_j + \sum_{j=1}^n \mathbf{v}^\top \mathbf{W}_{:,j} \mathbf{h}_j \right\} \\ &= \frac{1}{Z'} \prod_{j=1}^n \exp \left\{ c_j h_j + \mathbf{v}^\top \mathbf{W}_{:,j} \mathbf{h}_j \right\} \end{aligned} \quad (21.2)$$

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 write the joint probability over the vector \mathbf{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 .

$$\begin{aligned} P(h_j = 1 \mid \mathbf{v}) &= \frac{\tilde{P}(h_j = 1 \mid \mathbf{v})}{\tilde{P}(h_j = 0 \mid \mathbf{v}) + \tilde{P}(h_j = 1 \mid \mathbf{v})} \\ &= \frac{\exp \{c_j + \mathbf{v}^\top \mathbf{W}_{:,j}\}}{\exp \{0\} + \exp \{c_j + \mathbf{v}^\top \mathbf{W}_{:,j}\}} \\ &= \text{sigmoid} \left(c_j + \mathbf{v}^\top \mathbf{W}_{:,j} \right). \end{aligned} \quad (21.3)$$

We can now express the full conditional over the hidden layer as the factorial distribution:

$$P(\mathbf{h} \mid \mathbf{v}) = \prod_{j=1}^n \text{sigmoid} \left(c_j + \mathbf{v}^\top \mathbf{W}_{:,j} \right). \quad (21.4)$$

A similar derivation will show that the other condition of interest to us, $P(\mathbf{v} \mid \mathbf{h})$, is also a factorial distribution:

$$P(\mathbf{v} \mid \mathbf{h}) = \prod_{i=1}^d \text{sigmoid} (b_i + \mathbf{W}_{i,:} \mathbf{h}). \quad (21.5)$$

21.1.2 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 15.1 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 $\mathbf{h}^{(l)} \sim P(\mathbf{h} \mid \mathbf{v}^{(l)})$. Due to the factorial nature of the conditionals, we can simultaneously and independently sample from all the elements of $\mathbf{h}^{(l)}$ given $\mathbf{v}^{(l)}$. **Step 2:** Sample $\mathbf{v}^{(l+1)} \sim P(\mathbf{v} \mid \mathbf{h}^{(l)})$. Again, the factorial nature of the conditional $P(\mathbf{v} \mid \mathbf{h}^{(l)})$ allows us to sample from all the elements of $\mathbf{v}^{(l+1)}$ given $\mathbf{h}^{(l)}$.

21.2 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. 21.2.3. 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) $\{\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(t)}, \dots, \mathbf{v}^{(n)}\}$. The log likelihood under the RBM with parameters \mathbf{b} (visible unit biases), \mathbf{c} (hidden unit biases) and \mathbf{W} (interaction weights) is given by:

$$\begin{aligned}
\ell(\mathbf{W}, \mathbf{b}, \mathbf{c}) &= \sum_{t=1}^n \log P(\mathbf{v}^{(t)}) \\
&= \sum_{t=1}^n \log \sum_{\mathbf{h}} P(\mathbf{v}_{n,:}^{(t)}, \mathbf{h}) \\
&= \left. \sum_{t=1}^n \log \sum_{\mathbf{h}} \exp \left\{ -E(\mathbf{v}^{(t)}, \mathbf{h}) \right\} \right) - n \log Z \\
&= \left. \sum_{t=1}^n \log \sum_{\mathbf{h}} \exp \left\{ -E(\mathbf{v}^{(t)}, \mathbf{h}) \right\} \right) - n \log \sum_{\mathbf{v}, \mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\}
\end{aligned} \tag{21.6}$$

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 $\boldsymbol{\theta} = \{\mathbf{b}, \mathbf{c}, \mathbf{W}\}$:

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\theta}} \ell(\boldsymbol{\theta}) &= \frac{\partial}{\partial \boldsymbol{\theta}} \left(\sum_{t=1}^n \log \sum_{\mathbf{h}} \exp \left\{ -E(\mathbf{v}^{(t)}, \mathbf{h}) \right\} \right) - n \frac{\partial}{\partial \boldsymbol{\theta}} \log \sum_{\mathbf{v}, \mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\} \\
&= \sum_{t=1}^n \frac{\sum_{\mathbf{h}} \exp \left\{ -E(\mathbf{v}^{(t)}, \mathbf{h}) \right\} \frac{\partial}{\partial \boldsymbol{\theta}} - E(\mathbf{v}^{(t)}, \mathbf{h})}{\sum_{\mathbf{h}} \exp \left\{ -E(\mathbf{v}^{(t)}, \mathbf{h}) \right\}} - n \frac{\sum_{\mathbf{v}, \mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\} \frac{\partial}{\partial \boldsymbol{\theta}} - E(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}, \mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\}} \\
&= \sum_{t=1}^n \mathbb{E}_{P(\mathbf{h}|\mathbf{v}^{(t)})} \left[\frac{\partial}{\partial \boldsymbol{\theta}} - E(\mathbf{v}^{(t)}, \mathbf{h}) \right] - n \mathbb{E}_{P(\mathbf{v}, \mathbf{h})} \left[\frac{\partial}{\partial \boldsymbol{\theta}} - E(\mathbf{v}, \mathbf{h}) \right]
\end{aligned} \tag{21.7}$$

As we can see from Eq. 21.7, 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, $P(\mathbf{v}) = 1/n \sum_{t=1}^n \delta(\mathbf{x} - \mathbf{v}^{(t)})$ ¹ and the conditional distribution $P(\mathbf{h} | \mathbf{v}^{(t)})$. The second expectation (the *model term*) is with respect to the joint model distribution $P(\mathbf{v}, \mathbf{h})$.

This difference between a data-driven term and a model-driven term is not unique to RBMs, as discussed in some detail in Sec. 19.2, 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: $\frac{\partial}{\partial \boldsymbol{\theta}} - E(\mathbf{v}, \mathbf{h})$. We will consider first the gradient of the negative energy function of \mathbf{W} .

¹As discussed in Sec. 3.10.4, we use the term empirical distribution to refer to a mixture over delta functions placed on training examples

$$\begin{aligned}\frac{\partial}{\partial \mathbf{W}} - E(\mathbf{v}, \mathbf{h}) &= \frac{\partial}{\partial \mathbf{W}} \left(\mathbf{b}^\top \mathbf{v} + \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h} \right) \\ &= \mathbf{h} \mathbf{v}^\top\end{aligned}\tag{21.8}$$

The gradients with respect to \mathbf{b} and \mathbf{c} are similarly derived:

$$\frac{\partial}{\partial \mathbf{b}} - E(\mathbf{v}, \mathbf{h}) = \mathbf{v}, \quad \frac{\partial}{\partial \mathbf{c}} - E(\mathbf{v}, \mathbf{h}) = \mathbf{h}\tag{21.9}$$

Putting it all together we can the following equations for the gradients with respect to the RBM parameters and given n training examples:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{W}} \ell(\mathbf{W}, \mathbf{b}, \mathbf{c}) &= \sum_{t=1}^n \hat{\mathbf{h}}^{(t)} \mathbf{v}^{(t)\top} - N \mathbb{E}_{P(\mathbf{v}, \mathbf{h})} [\mathbf{h} \mathbf{v}^\top] \\ \frac{\partial}{\partial \mathbf{b}} \ell(\mathbf{W}, \mathbf{b}, \mathbf{c}) &= \sum_{t=1}^n \mathbf{v}^{(t)} - n \mathbb{E}_{P(\mathbf{v}, \mathbf{h})} [\mathbf{v}] \\ \frac{\partial}{\partial \mathbf{c}} \ell(\mathbf{W}, \mathbf{b}, \mathbf{c}) &= \sum_{t=1}^n \hat{\mathbf{h}}^{(t)} - n \mathbb{E}_{P(\mathbf{v}, \mathbf{h})} [\mathbf{h}]\end{aligned}$$

where we have defined $\hat{\mathbf{h}}^{(t)}$ as

$$\hat{\mathbf{h}}^{(t)} = \mathbb{E}_{P(\mathbf{h}|\mathbf{v}^{(t)})} [\mathbf{h}] = \text{sigmoid} \left(\mathbf{c} + \mathbf{v}^{(t)} \mathbf{W} \right).\tag{21.10}$$

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 $P(\mathbf{v}, \mathbf{h})$. While we have conditional distributions $P(\mathbf{v} | \mathbf{h})$ and $P(\mathbf{h} | \mathbf{v})$ that are easy to work with, the RBM joint distribution is not amenable to analytic evaluation of the expectation $\mathbb{E}_{P(\mathbf{v}, \mathbf{h})} [f(\mathbf{v}, \mathbf{h})]$.

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. 19.2, 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 $P(\mathbf{v}, \mathbf{h})$, 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. 21.1.2).

Thus, we can imagine using, for example, T MCMC samples from $P(\mathbf{v}, \mathbf{h})$ to form a Monte Carlo estimate of the expectations over the joint distribution:

$$\mathbb{E}_{P(\mathbf{v}, \mathbf{h})} [f(\mathbf{v}, \mathbf{h})] \approx \frac{1}{T} \sum_{t=1}^T f(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}). \quad (21.11)$$

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

21.2.1 Contrastive Divergence Training of the RBM

As discussed in a more general context in Sec. 19.2, 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 nevertheless has been empirically shown to be effective. The contrastive divergence algorithm, as applied to RBMs, is given in Algorithm 21.1.

21.2.2 Stochastic Maximum Likelihood (Persistent Contrastive Divergence) for the RBM

While contrastive divergence has been the most popular method of training RBMs, the *stochastic maximum likelihood* (SML) algorithm (Younes, 1998; Tielemans, 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. 19.2. 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 21.2 describes the SML algorithm as applied to RBMs.

TODO: include experimental examples, i.e. an RBM trained with CD on MNIST

21.2.3 Other Inductive Principles

TODO: Other inductive principles have been used to train RBMs. In this section we briefly discuss these.

Algorithm 21.1 The contrastive divergence algorithm, using gradient ascent as the optimization procedure.

Set ϵ , the step size, to a small positive number
 Set k , the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{v}; \theta)$ to mix when initialized from p_{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 $\{\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(m)}\}$.
 $\Delta_{\mathbf{W}} \leftarrow \frac{1}{m} \sum_{t=1}^m \hat{\mathbf{h}}^{(t)} \mathbf{v}^{(t) \top}$
 $\Delta_{\mathbf{b}} \leftarrow \frac{1}{m} \sum_{t=1}^m \mathbf{v}^{(t)}$
 $\Delta_{\mathbf{c}} \leftarrow \frac{1}{m} \sum_{t=1}^m \hat{\mathbf{h}}^{(t)}$
for $t = 1$ to m **do**
 $\tilde{\mathbf{v}}^{(t)} \leftarrow \mathbf{v}^{(t)}$
end for
for $l = 1$ to k **do**
for $t = 1$ to m **do**
 $\tilde{\mathbf{h}}^{(t)}$ sampled from $\prod_{j=1}^n \text{sigmoid}(c_j + \tilde{\mathbf{v}}^{(t) \top} \mathbf{W}_{:,j})$.
 $\tilde{\mathbf{v}}^{(t)}$ sampled from $\prod_{i=1}^d \text{sigmoid}(b_i + \mathbf{W}_{i,:} \tilde{\mathbf{h}}^{(t)})$.
end for
end for
 $\bar{\mathbf{h}}^{(t)} \leftarrow \text{sigmoid}(\mathbf{c} + \tilde{\mathbf{v}}^{(t)} \mathbf{W})$
 $\Delta_{\mathbf{W}} \leftarrow \Delta_{\mathbf{W}} - \frac{1}{m} \sum_{t=1}^m \bar{\mathbf{h}}^{(t)} \tilde{\mathbf{v}}^{(t) \top}$
 $\Delta_{\mathbf{b}} \leftarrow \Delta_{\mathbf{b}} - \frac{1}{m} \sum_{t=1}^m \bar{\mathbf{h}}^{(t)}$
 $\Delta_{\mathbf{c}} \leftarrow \Delta_{\mathbf{c}} - \frac{1}{m} \sum_{t=1}^m \bar{\mathbf{h}}^{(t)}$
 $\mathbf{W} \leftarrow \mathbf{W} + \epsilon \Delta_{\mathbf{W}}$
 $\mathbf{b} \leftarrow \mathbf{b} + \epsilon \Delta_{\mathbf{b}}$
 $\mathbf{c} \leftarrow \mathbf{c} + \epsilon \Delta_{\mathbf{c}}$
end while

21.3 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 learning algorithms, but they are still deservedly recognized for their important

Algorithm 21.2 The stochastic maximum likelihood / persistent contrastive divergence algorithm for training an RBM.

Set ϵ , the step size, to a small positive number
 Set k , the number of Gibbs steps, high enough to allow a Markov chain of $p(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta} + \epsilon \Delta \boldsymbol{\theta})$ to burn in, starting from samples from $p(\mathbf{v}, \mathbf{h}; \boldsymbol{\theta})$. Perhaps 1 for RBM on a small image patch.
 Initialize a set of m samples $\{\tilde{\mathbf{v}}^{(1)}, \dots, \tilde{\mathbf{v}}^{(m)}\}$ 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 $\{\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(m)}\}$ from the training set.

$$\Delta_{\mathbf{W}} \leftarrow \frac{1}{m} \sum_{t=1}^m \hat{\mathbf{h}}^{(t)} \mathbf{v}^{(t) \top}$$

$$\Delta_{\mathbf{b}} \leftarrow \frac{1}{m} \sum_{t=1}^m \mathbf{v}^{(t)}$$

$$\Delta_{\mathbf{c}} \leftarrow \frac{1}{m} \sum_{t=1}^m \hat{\mathbf{h}}^{(t)}$$
for $l = 1$ to k **do**
for $t = 1$ to m **do**
 $\tilde{\mathbf{h}}^{(t)}$ sampled from $\prod_{j=1}^n \text{sigmoid}(c_j + \tilde{\mathbf{v}}^{(t) \top} \mathbf{W}_{:,j})$.
 $\tilde{\mathbf{v}}^{(t)}$ sampled from $\prod_{i=1}^d \text{sigmoid}(b_i + \mathbf{W}_{i,:} \tilde{\mathbf{h}}^{(t)})$.
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

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. 21.1b for an example.

A DBN with L hidden layers contains L weight matrices: $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}$. It also contains $L + 1$ bias vectors: $b^{(0)}, \dots, b^{(L)}$ with $b^{(0)}$ providing the biases for the visible layer. The probability distribution represented by the DBN is given by

$$p(\mathbf{h}^{(L)}, \mathbf{h}^{(L-1)}) \propto \exp\left(\mathbf{b}^{(L)\top} \mathbf{h}^{(L)} + \mathbf{b}^{(L-1)\top} \mathbf{h}^{(L-1)} + \mathbf{h}^{(L-1)\top} \mathbf{W}^{(L)} \mathbf{h}^{(L)}\right),$$

$$p(h_i^{(l)} = 1 | \mathbf{h}^{(l+1)}) = \sigma\left(b_i^{(l)} + \mathbf{W}_{:,i}^{(l+1)\top} \mathbf{h}^{(l+1)}\right) \forall i, \forall l \in 1, \dots, L-2,$$

$$p(v_i = 1 | \mathbf{h}^{(1)}) = \sigma\left(b_i^{(0)} + \mathbf{W}_{:,i}^{(1)\top} \mathbf{h}^{(1)}\right) \forall i.$$

In the cause of real-valued visible units, substitute

$$\mathbf{v} \sim \mathcal{N}(\mathbf{v} | \mathbf{b}^{(0)} + \mathbf{W}^{(1)\top} \mathbf{h}^{(1)}, \boldsymbol{\beta}^{-1})$$

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.

21.4 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. 21.2 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 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, \mathbf{v} , and three hidden layers, $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}$ and $\mathbf{h}^{(3)}$, the joint probability is given by:

$$P(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}; \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}; \boldsymbol{\theta})). \quad (21.12)$$

The DBM energy function is:

$$E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}; \boldsymbol{\theta}) = -\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} - \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)} - \mathbf{h}^{(2)\top} \mathbf{W}^{(3)} \mathbf{h}^{(3)}. \quad (21.13)$$

In comparison to the RBM energy function (Eq. 21.1), the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($\mathbf{W}^{(2)}$ and $\mathbf{W}^{(3)}$). 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 connectioned 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 bipartite graph, with odd layers on one side and 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 $P(\mathbf{h}^{(1)} = 1 | \mathbf{v}, \mathbf{h}^{(2)})$, in the case of a DBM with two hidden layers (of course, this result generalizes to a DBM with

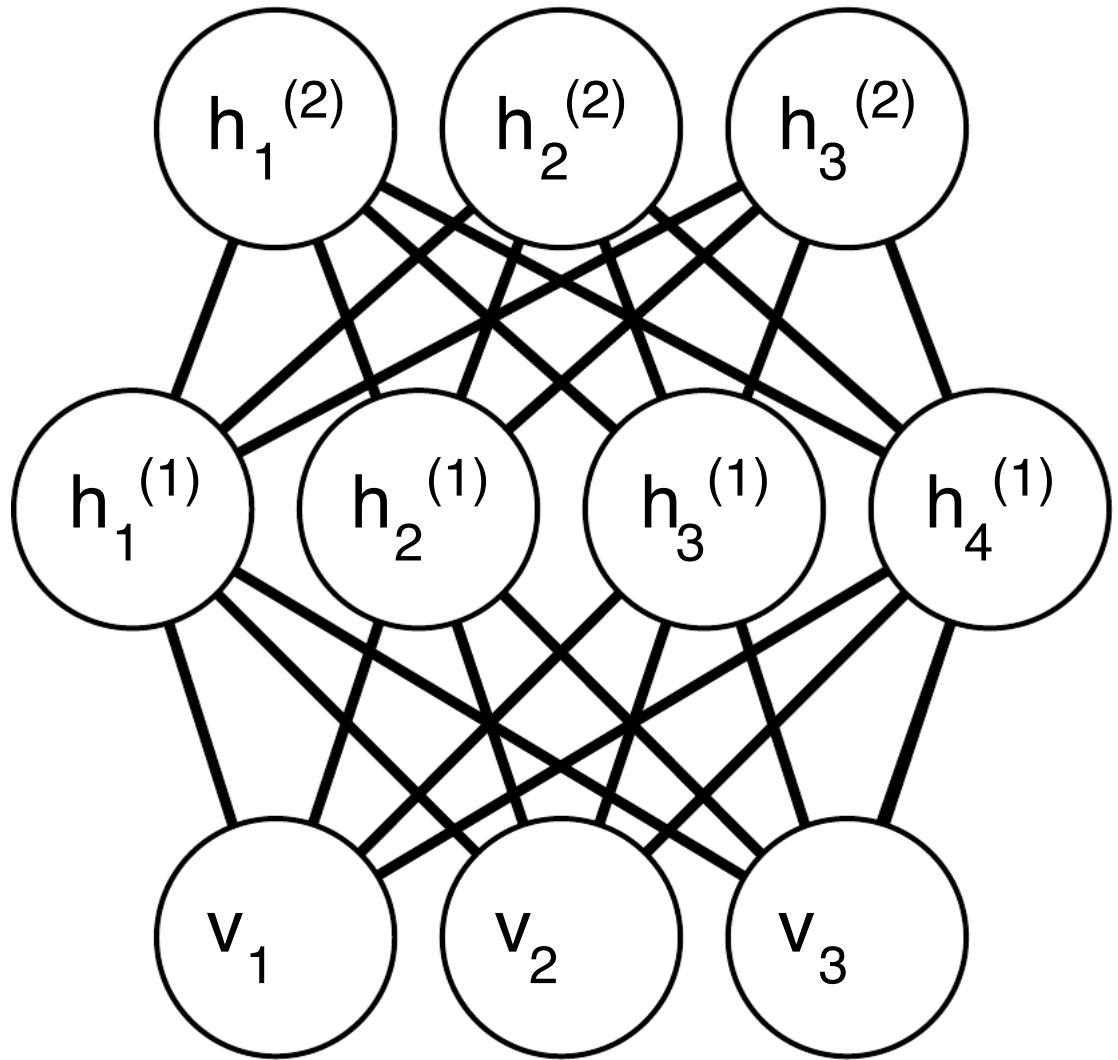


Figure 21.2: The deep Boltzmann machine (biases on all units are present but suppressed to simplify notation).

any number of layers).

$$\begin{aligned}
P(\mathbf{h}^{(1)} \mid \mathbf{v}, \mathbf{h}^{(2)}) &= \frac{P(\mathbf{h}^{(1)}, \mathbf{v}, \mathbf{h}^{(2)})}{P(\mathbf{v}, \mathbf{h}^{(2)})} \\
&= \frac{\exp(\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} + \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)})}{\sum_{h_1^{(1)}=0}^1 \cdots \sum_{h_n^{(1)}=0}^1 \exp(\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} + \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)})} \\
&= \frac{\exp(\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} + \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)})}{\sum_{h_1^{(1)}=0}^1 \cdots \sum_{h_n^{(1)}=0}^1 \exp(\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} + \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)})} \\
&= \frac{\exp\left(\sum_{j=1}^n \mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^1 \cdots \sum_{h_n^{(1)}=0}^1 \exp\left(\sum_{j'=1}^n \mathbf{v}^\top \mathbf{W}_{:,j'}^{(1)} h_{j'}^{(1)} + h_{j'}^{(1)\top} \mathbf{W}_{j',:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \frac{\prod_j \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{\sum_{h_1^{(1)}=0}^1 \cdots \sum_{h_n^{(1)}=0}^1 \prod_{j'} \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j'}^{(1)} h_{j'}^{(1)} + h_{j'}^{(1)\top} \mathbf{W}_{j',:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \prod_j \frac{\exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{\sum_{h_j^{(1)}=0}^1 \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \prod_j \frac{\exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{1 + \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} + \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \prod_j \frac{\exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} h_j^{(1)} + h_j^{(1)\top} \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{1 + \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} + \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \prod_j P(h_j^{(1)} \mid \mathbf{v}, \mathbf{h}^{(2)}). \tag{21.14}
\end{aligned}$$

From the above we can conclude that the conditional distribution for any layer of the DBM conditioned on the neighboring layers, is factorial (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:

$$\begin{aligned}
P(h_j^{(1)} = 1 \mid \mathbf{v}, \mathbf{h}^{(2)}) &= \frac{\exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} + \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)}{1 + \exp\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} + \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \frac{1}{1 + \exp\left(-\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} - \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right)} \\
&= \text{sigmoid}\left(\mathbf{v}^\top \mathbf{W}_{:,j}^{(1)} + \mathbf{W}_{j,:}^{(2)} \mathbf{h}^{(2)}\right). \tag{21.15}
\end{aligned}$$

For the two layer DBM, the conditional distributions of the remaining two layers ($\mathbf{v}, \mathbf{h}^{(2)}$) also factorize. That is $P(\mathbf{v} \mid \mathbf{h}^{(1)}) = \prod_{i=1}^d P(v_i \mid h^{(1)})$, where

$$P(v_i = 1 \mid h^{(1)}) = \text{sigmoid } \mathbf{W}_{i,:}^{(1)} h^{(1)}. \tag{21.16}$$

Also, $P(h^{(2)} | h^{(1)}) = \prod_{k=1}^m P(h_k^{(2)} | h^{(1)})$, where

$$P(h_k^{(2)} = 1 \mid h^{(1)}) = \text{sigmoid} \left(h^{(1)\top} W_{:,k}^{(2)} \right). \quad (21.17)$$

21.4.1 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

21.4.2 Mean Field Inference in the DBM

For the two hidden layer DBM, the conditional distributions, $P(\mathbf{v} \mid \mathbf{h}^{(1)})$, $P(\mathbf{h}^{(1)} \mid \mathbf{v}, \mathbf{h}^{(2)})$, and $P(\mathbf{h}^{(2)} \mid \mathbf{h}^{(1)})$ are factorial, however the posterior distribution over all the hidden units given the visible unit, i.e. $P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v})$, can be complicated. This is, of course, due to the interaction weights $\mathbf{W}^{(2)}$ between $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$ which render these variables mutually dependent, given an observed \mathbf{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. 20.1), 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 $Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v})$ be the approximation of $P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v})$. The mean field assumption implies that

$$Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v}) = \prod_{j=1}^n Q(h_j^{(1)} \mid \mathbf{v}) \prod_{k=1}^m Q(h_k^{(2)} \mid \mathbf{v}). \quad (21.18)$$

The mean field approximation attempts to find *for every observation* a member of this family of distributions that “best fits” the true posterior $P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})$. By best fit, we specifically mean that we wish to find the approximation Q that minimizes the KL-divergence with P , i.e. $\text{KL}(Q \| P)$ where:

$$\text{KL}(Q\|P) = \int Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v}) \log \frac{Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v})}{P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} \mid \mathbf{v})} \quad (21.19)$$

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 $\mathbf{h}^{(1)}$ to be associated with a parameter. Specifically, for each $j \in \{1, \dots, n\}$, $\hat{h}_j^{(1)} = P(h_j^{(1)} = 1)$, where $\hat{h}_j^{(1)} \in [0, 1]$ and for each $k \in \{1, \dots, m\}$, $\hat{h}_k^{(2)} = P(h_k^{(2)} = 1)$, where $\hat{h}_k^{(2)} \in [0, 1]$. Thus we have the following approximation to the posterior:

$$\begin{aligned} Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) &= \prod_{j=1}^n Q(h_j^{(1)} | \mathbf{v}) \prod_{k=1}^m Q(h_k^{(2)} | \mathbf{v}) \\ &= \prod_{j=1}^n (\hat{h}_j^{(1)})^{h_j^{(1)}} (1 - \hat{h}_j^{(1)})^{(1-h_j^{(1)})} \times \prod_{k=1}^m (\hat{h}_k^{(2)})^{h_k^{(2)}} (1 - \hat{h}_k^{(2)})^{(1-h_k^{(2)})} \end{aligned} \quad (21.20)$$

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 $\text{KL}(Q \| P)$ 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 $\text{KL}(Q \| P)$, we will maximize the variational lower bound (or evidence lower bound: see Sec. 20.1), which in the context of the 2-hidden-layer deep Boltzmann machine is given by:

$$\begin{aligned} \mathcal{L}(Q) &= \sum_{\mathbf{h}^{(1)}, \mathbf{h}^{(2)}} Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) \log \left(\frac{P(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta})}{q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})} \right) \\ &= - \sum_{\mathbf{h}^{(1)}, \mathbf{h}^{(2)}} Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta}) - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q) \end{aligned}$$

21.4.3 Variational Expectation Maximization

TODO Variational EM for DBM

21.4.4 Variational Learning With SML

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 19 for a 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 20. Mean field is incompatible with most of the methods for approximating the gradients of the log partition function described in chapter 19. Moreover, contrastive divergence offers no speedup relative to naive MCMC methods when the posterior is intractable, because sampling the negative phase fantasy particles requires sampling from the posterior. This means that DBMs are usually trained using 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.

Unfortunately, training a DBM using variational learning and SML 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 pre-trained configuration, training usually succeeds. See section 21.4.5 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 21.4.6, *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 21.4.7, which modify the model family in order to obtain a better conditioned Hessian matrix.

21.4.5 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

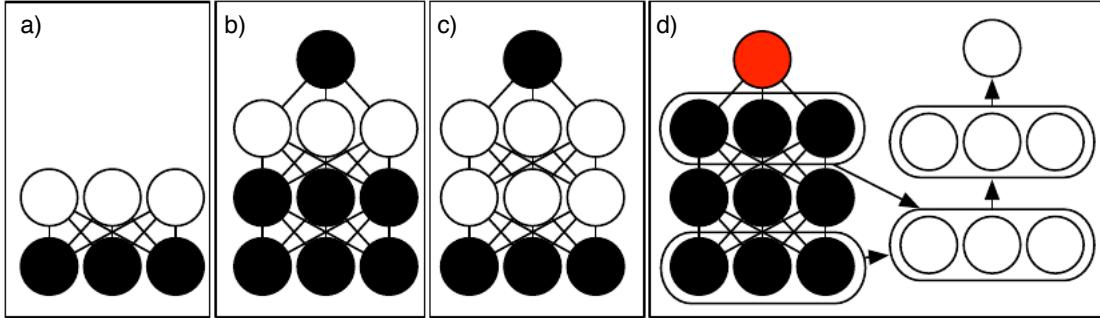


Figure 21.3: The deep Boltzmann machine training procedure used to obtain the state of the art classification accuracy on the MNIST dataset (Srivastava *et al.*, 2014; Salakhutdinov and Hinton, 2009a). a) Train an RBM by using CD to approximately maximize $\log P(\mathbf{v})$. b) Train a second RBM that models $\mathbf{h}^{(1)}$ and Y by using CD- k to approximately maximize $\log P(\mathbf{h}^{(1)}, Y)$ where $\mathbf{h}^{(1)}$ 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 $\log P(\mathbf{v}, \mathbf{y})$ using stochastic maximum likelihood with $k = 5$. d) Delete Y from the model. Define a new set of features $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$ 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 $\log P(Y | \mathbf{v})$ using stochastic gradient descent and dropout. Figure reprinted from (Goodfellow *et al.*, 2013b).

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

21.4.6 Multi-Prediction Deep Boltzmann Machines

TODO– cite stoyanov TODO

21.4.7 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.

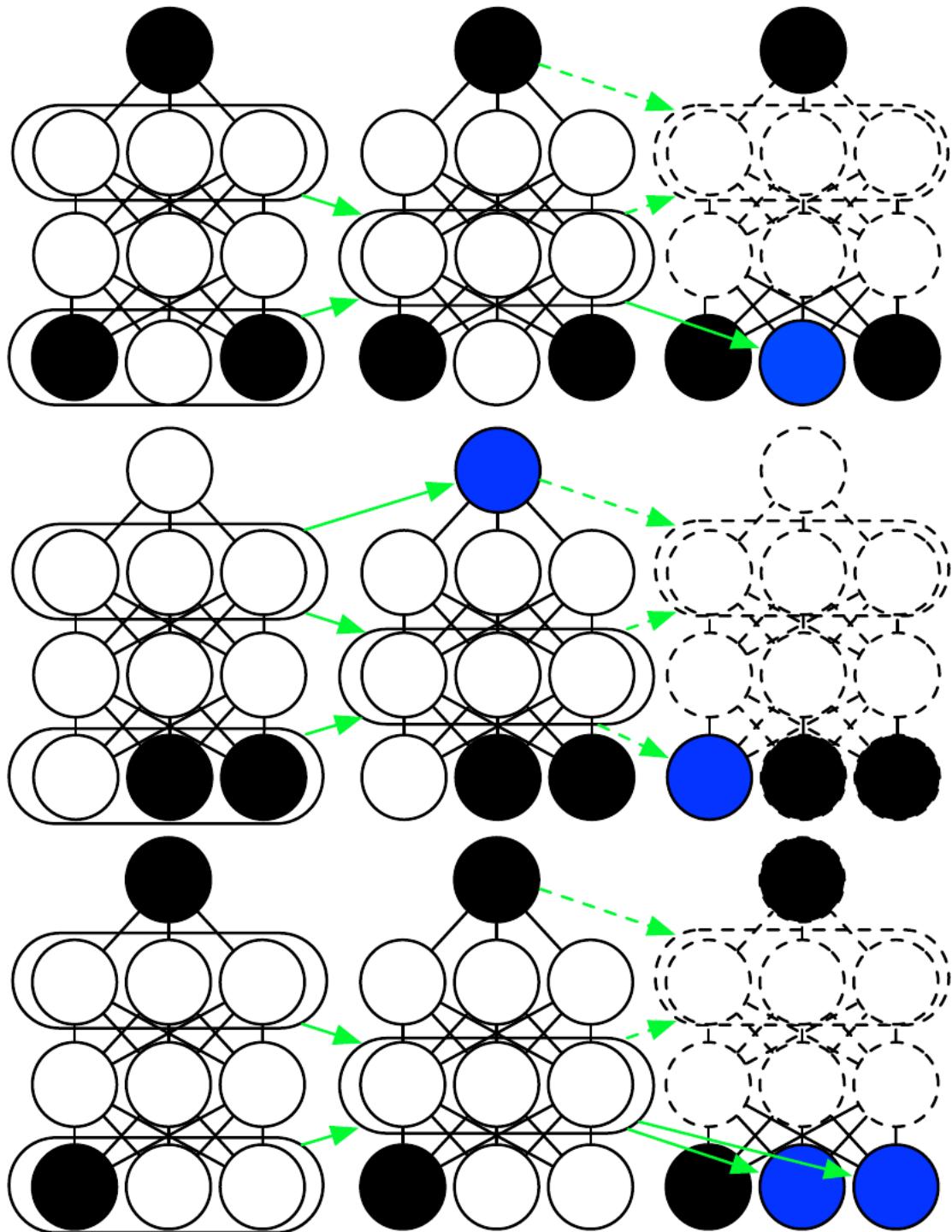


Figure 21.4: TODO caption and label, reference from text Figure reprinted from (Goodfellow *et al.*, 2013b).

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 19.2. If $p(h | v)$ is intractable, one may still train the model using the negative variational free energy rather than the likelihood, as described in 20.4.

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 essentially a sequence of RBMs composed together. The model is depicted graphically in Fig. 21.1c.

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 $P(h | v)$ 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

21.5 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.

21.5.1 Gaussian-Bernoulli RBMs

TODO—cite exponential family harmoniums? TODO—multiple ways of parametrizing them (citations?)

21.5.2 mcRBMs

TODO—mcRBMs² TODO—HMC

21.5.3 mPoT Model

TODO—mPoT

²The 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.”

21.5.4 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 $(\mathbf{h} \odot \mathbf{s})\mathbf{W}^\top$. In other words, each column $\mathbf{W}_{:,i}$ defines a component that can appear in the input. The corresponding spike variable h_i determines whether that component is present at all. The corresponding slab variable 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 $\mathbf{W}_{:,i}$. 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 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. 14.1.

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.

21.6 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 p over n binary detector units \mathbf{d} and enforce $p = \max_i d_i$ by setting the energy function to be ∞ 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 3×3 pooling region this requires $2^9 = 512$ 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 ∞ 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 machines³. 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

21.7 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

³The 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.

21.8 Directed Generative Nets

TODO: sigmoid belief nets TODO: refer back to DBN TODO sparse coding TODO deconvolutional nets? TODO refer back to S3C and BSC 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

21.8.1 Sigmoid Belief Nets

21.8.2 Variational Autoencoders

The variational autoencoder is model

$$\mathcal{L} \quad (21.21)$$

TODO

21.8.3 Variational Interpretation of PSD

TODO, develop the explanation of Sec. 9.1 of [Bengio et al. \(2013c\)](#).

21.8.4 Generative Adversarial Networks

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, 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 \mathbf{z} to samples \mathbf{x} . This function $g(\mathbf{z})$ defines the generative model. The distribution $p(\mathbf{z})$ 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 $G(\mathbf{z})$ as defining a conditional distribution

$$p(\mathbf{x} | \mathbf{z}) = \mathcal{N}(\mathbf{x} | g(\mathbf{z}), \frac{1}{\beta} \mathbf{I}) ,$$

but in all learning rules we take limit as $\beta \rightarrow \infty$ so we can treat $g(\mathbf{z})$ 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 \mathbf{x} as input and outputs its estimate of the probability that \mathbf{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:

$$g^* = \arg \min_g \max_d V(g, d)$$

where

$$v(g, d) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{x} \sim p_{\text{model}}} \log (1 - d(\mathbf{x})).$$

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.

21.9 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 16.8.1 and 19.5 for details. Since the early work (Vincent, 2011a) made the connection with Gaussian RBMs, this gave denoising auto-encoders 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 16.9 (see also Figures 16.9, 18.3). 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 16.8.1, 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 configuration 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).

21.9.1 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.

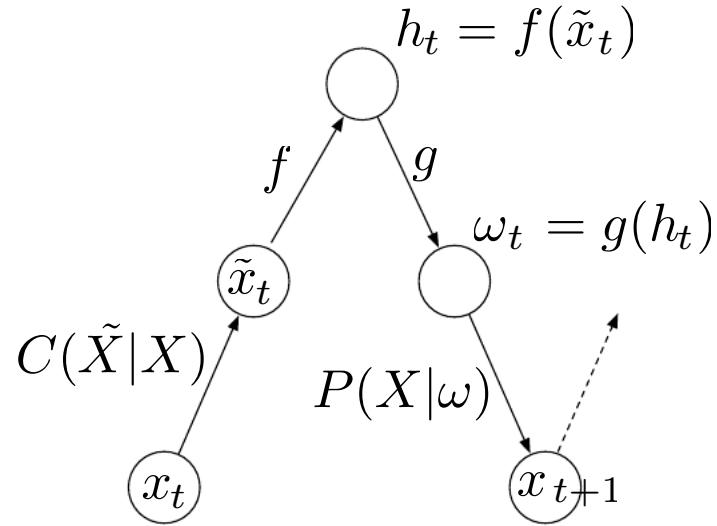


Figure 21.5: Each step of the Markov chain associated with a trained denoising auto-encoder, that generates the samples from the probabilistic model implicitly trained by the denoising reconstruction criterion. Each step consists in (a) injecting corruption C in state \mathbf{x} , yielding $\tilde{\mathbf{x}}$, (b) encoding it with f , yielding $\mathbf{h} = f(\tilde{\mathbf{x}})$, (c) decoding the result with g , yielding parameters ω for the reconstruction distribution, and (d) given ω , sampling a new state from the reconstruction distribution $P(\mathbf{x}|\omega = g(f(\tilde{\mathbf{x}})))$. In the typical squared reconstruction error case, $g(\mathbf{h}) = \hat{\mathbf{x}}$, which estimates $E[\mathbf{x}|\tilde{\mathbf{x}}]$, corruption consists in adding Gaussian noise and sampling from $P(\mathbf{x}|\omega)$ consists in adding another Gaussian noise to the reconstruction $\hat{\mathbf{x}}$. The latter noise level should correspond to the mean squared error of reconstructions, whereas the injected noise is a hyper-parameter 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 21.5:

1. starting from the previous state \mathbf{x} , inject corruption noise, sampling $\tilde{\mathbf{x}}$ from

$$C(\tilde{\mathbf{x}}|\mathbf{x}).$$

2. Encode $\tilde{\mathbf{x}}$ into $\mathbf{h} = f(\tilde{\mathbf{x}})$.
3. Decode \mathbf{h} to obtain the parameters $\omega = g(\mathbf{h})$ of $P(\mathbf{x}|\omega = g(\mathbf{h})) = P(\mathbf{x}|\tilde{\mathbf{x}})$.
4. Sample the next state \mathbf{x} from $P(\mathbf{x}|\omega = g(\mathbf{h})) = P(\mathbf{x}|\tilde{\mathbf{x}})$.

The theorem states that if the auto-encoder $P(\mathbf{x}|\tilde{\mathbf{x}})$ 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} .

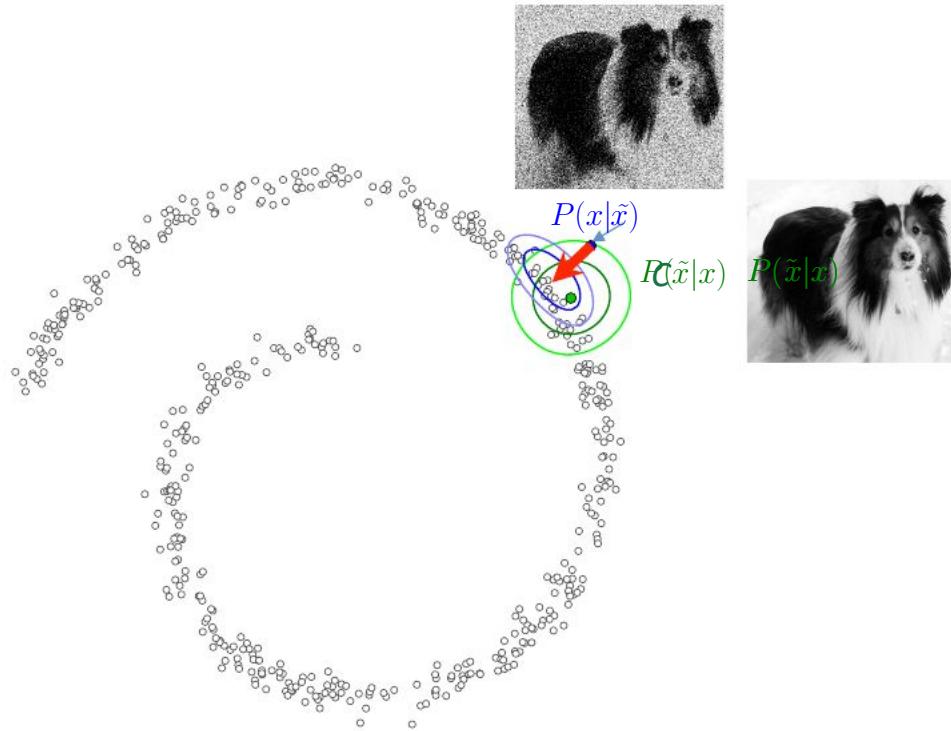


Figure 21.6: Illustration of one step of the sampling Markov chain associated with a denoising auto-encoder (see also Figure 21.5). In the figure, the data (black circles) are sitting near a low-dimensional manifold (a spiral, here), and the two stochastic steps of the Markov chain are first to corrupt \mathbf{x} (clean image of dog, green circle) into $\tilde{\mathbf{x}}$ (noisy image of dog, blue circle) via $C(\tilde{\mathbf{x}}|\mathbf{x})$ (here an isotropic Gaussian noise in green), and then to sample a new \mathbf{x} via the estimated denoising $P(\mathbf{x}|\tilde{\mathbf{x}})$. Note how there are many possible \mathbf{x} which could have given rise to $\tilde{\mathbf{x}}$, and these all lie on the manifold in the neighborhood of $\tilde{\mathbf{x}}$, hence the flattened shape of $P(\mathbf{x}|\tilde{\mathbf{x}})$ (in blue). *Modified from a figure first created and graciously authorized by Jason Yosinski.*

Figure 21.6 illustrates the sampling process of the DAE in a way that complements Figure 21.5, with a specific imagined example. For a more elaborate discussion of the

probabilistic nature of denoising auto-encoders, and their generalization ([Bengio et al., 2014b](#)), *Generative Stochastic Networks* (GSNs), see Section 21.10 below. In particular, the noise does not have to be injected only in the input, and it could be injected anywhere along the chain. GSNs also generalize DAEs by allowing the state of the Markov chain to be extended beyond the visible variable \mathbf{x} , to include also some latent variable \mathbf{h} . Finally, Section 21.10 discusses training strategies for DAEs that are aimed at making it a better generative model and not just a good feature learner.

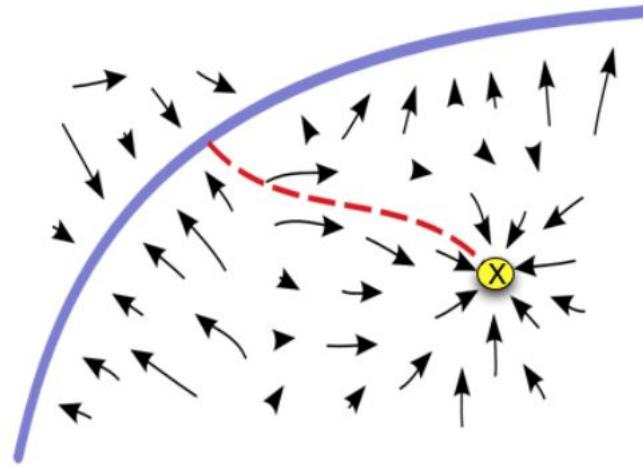


Figure 21.7: Illustration of the effect of the walk-back training procedure, used for denoising auto-encoders or GSNs in general. The objective is to remove spurious modes faster by letting the Markov chain go towards them (along the red path, starting on the purple data manifold and following the arrows 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).

21.9.2 Clamping and Conditional Sampling

Similarly to Boltzmann machines, denoising auto-encoders and GSNs can be used to sample from a conditional distribution $P(\mathbf{x}_f | \mathbf{x}_o)$, 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 15.1.1.

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 21.8.



Figure 21.8: 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.

21.9.3 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 19.2 and 21.2.1) 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 21.7.

Figure 21.9 illustrates the application of the walk-back procedure in a generative stochastic network, which is described in the next section.

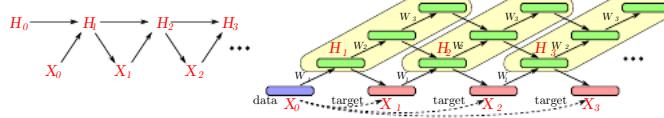


Figure 21.9: Left: graphical model of the generative Markov chain associated with a generative stochastic network (GSN). Right specific case where the latent variable is formed of several layers, each connected to the one above and the one below, making the generative process very similar to Gibbs sampling 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).

21.10 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 \mathbf{x}). The generative Markov chain looks like the one in Figure 21.10. An example of a GSN structured like a deep Boltzmann machine and trained by the walk-back procedure is shown in Figure 21.9.

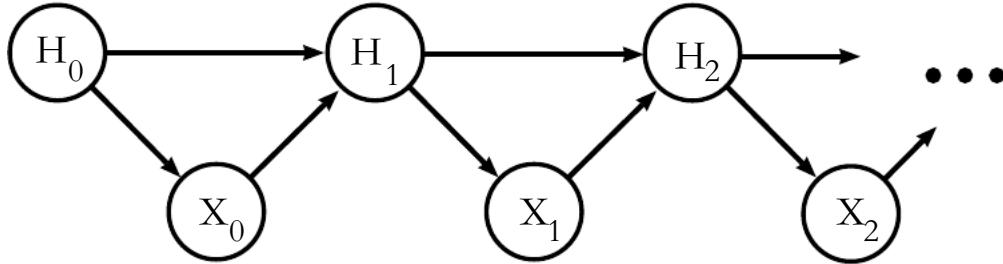


Figure 21.10: Markov chain of a GSN (Generative Stochastic Network) with latent variables H and visible variable X , i.e., an unfolding of the generative process with X_k and H_k at step k of the chain.

A GSN is parametrized by two conditional probability distributions which specify one step of the Markov chain:

1. $P(X_k|H_k)$ 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. $P(H_k|H_{k-1}, X_{k-1})$ 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 $X_0 = x$ to the observed example and maximizing the probability of generating x at some subsequent time steps, i.e., maximizing $\log P(X_k = x|H_k)$, where H_k is sampled from the chain, given $X_0 = x$. In order to estimate the gradient of $\log P(X_k = x|H_k)$ with respect to the other pieces of the model, Bengio *et al.* (2014b) use the reparametrization trick, introduced in Section 14.5.1.

The *walk-back training* protocol (described in Section 21.9.3 was used (Bengio *et al.*, 2014b) to improve training convergence of GSNs.

21.10.1 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 $P(\mathbf{y}|\mathbf{x})$.

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 21.11.

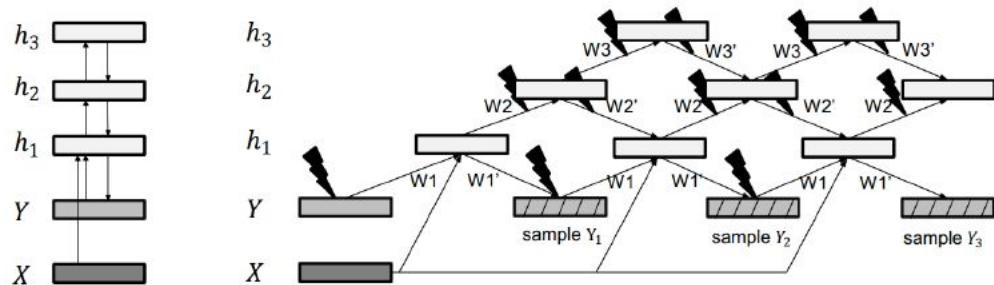


Figure 21.11: Markov chain arising out of a discriminant GSN, i.e., where a GSN is used as a structured output model over a variable Y , conditioned on an input X . Reproduced with permission from Zhou and Troyanskaya (2014). The structure is as in a GSN (over the output) but with computations being conditioned on the input X at each step.

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 $P(\mathbf{y}|\mathbf{x})$ 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 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.

21.11 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 subtlety 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 $\log \tilde{p}(\mathbf{x}) - \log Z$ 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 $\log p(\mathbf{x})$. 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 un-

acceptable. 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 19.6) 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 21.8.4). However, it would require an expensive operation (training a discriminator) each time one would have to evaluate performance

TODO– have a section on sum-product networks including a citation to James Martens’ recent paper

Bibliography

- 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. [302](#)
- Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In *ICLR'2013*. also arXiv report 1211.4246. [286](#), [302](#), [304](#)
- Alain, G., Bengio, Y., Yao, L., Éric Thibodeau-Laufer, Yosinski, J., and Vincent, P. (2015). GSNs: Generative stochastic networks. arXiv:1503.05571. [288](#)
- Amari, S. (1997). Neural learning in structured parameter spaces - natural Riemannian gradient. In *Advances in Neural Information Processing Systems*, pages 127–133. MIT Press. [116](#)
- Anderson, E. (1935). The Irises of the Gaspe Peninsula. *Bulletin of the American Iris Society*, **59**, 2–5. [14](#)
- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. Technical report, arXiv:1409.0473. [251](#)
- 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. [48](#), [229](#)
- Baldi, P. and Brunak, S. (1998). *Bioinformatics, the Machine Learning Approach*. MIT Press. [231](#)
- Baldi, P. and Sadowski, P. J. (2013). Understanding dropout. In *Advances in Neural Information Processing Systems 26*, pages 2814–2822. [153](#)
- 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. [202](#)
- Barron, A. E. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Trans. on Information Theory*, **39**, 930–945. [126](#)
- Bartholomew, D. J. (1987). *Latent variable models and factor analysis*. Oxford University Press. [290](#)
- Basilevsky, A. (1994). *Statistical Factor Analysis and Related Methods: Theory and Applications*. Wiley. [290](#)
- 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. [57](#)

- Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Ann. Math. Stat.*, **37**, 1559–1563. [227](#)
- 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. [154](#)
- Becker, S. and Hinton, G. (1992). A self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, **355**, 161–163. [335](#)
- Belkin, M. and Niyogi, P. (2002). Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS'01*, Cambridge, MA. MIT Press. [322](#)
- Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, **15**(6), 1373–1396. [98](#), [340](#)
- 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. [207](#)
- Bengio, Y. (1991). *Artificial Neural Networks and their Application to Sequence Recognition*. Ph.D. thesis, McGill University, (Computer Science), Montreal, Canada. [212](#), [231](#)
- Bengio, Y. (1993). A connectionist approach to speech recognition. *International Journal on Pattern Recognition and Artificial Intelligence*, **7**(4), 647–668. [229](#)
- Bengio, Y. (1999a). Markovian models for sequential data. *Neural Computing Surveys*, **2**, 129–162. [229](#)
- Bengio, Y. (1999b). Markovian models for sequential data. *Neural Computing Surveys*, **2**, 129–162. [231](#)
- Bengio, Y. (2009). *Learning deep architectures for AI*. Now Publishers. [95](#), [128](#)
- Bengio, Y. (2013). Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal. [275](#)
- Bengio, Y. and Bengio, S. (2000b). Modeling high-dimensional discrete data with multi-layer neural networks. In *NIPS'99*, pages 400–406. MIT Press. [207](#), [209](#), [210](#)
- Bengio, Y. and Delalleau, O. (2009). Justifying and generalizing contrastive divergence. *Neural Computation*, **21**(6), 1601–1621. [302](#), [363](#), [388](#)
- Bengio, Y. and Frasconi, P. (1996). Input/Output HMMs for sequence processing. *IEEE Transactions on Neural Networks*, **7**(5), 1231–1249. [231](#)
- Bengio, Y. and LeCun, Y. (2007a). Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. [95](#)
- Bengio, Y. and LeCun, Y. (2007b). Scaling learning algorithms towards AI. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*. MIT Press. [129](#)
- Bengio, Y. and Monperrus, M. (2005). Non-local manifold tangent learning. In *NIPS'04*, pages 129–136. MIT Press. [97](#), [341](#)

- 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*. 17
- Bengio, Y., De Mori, R., Flammia, G., and Kompe, R. (1992). Global optimization of a neural network-hidden Markov model hybrid. *IEEE Transactions on Neural Networks*, **3**(2), 252–259. 229, 231
- 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). 163, 218
- Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Tr. Neural Nets.* **163**, 164, 210, 216, 218, 219
- 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. 231
- Bengio, Y., Ducharme, R., and Vincent, P. (2001a). A neural probabilistic language model. In *NIPS'00*, pages 932–938. MIT Press. 16
- Bengio, Y., Ducharme, R., and Vincent, P. (2001b). A neural probabilistic language model. In *NIPS'2000*, pages 932–938. 248, 249
- 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. 343, 344
- Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003a). A neural probabilistic language model. *JMLR*, **3**, 1137–1155. 248
- Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003b). A neural probabilistic language model. *Journal of Machine Learning Research*, **3**, 1137–1155. 343, 344
- Bengio, Y., Delalleau, O., and Le Roux, N. (2006a). The curse of highly variable functions for local kernel machines. In *NIPS'2005*. 94
- Bengio, Y., Larochelle, H., and Vincent, P. (2006b). Non-local manifold Parzen windows. In *NIPS'2005*. MIT Press. 97, 340
- Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS'2006*. 16, 308, 311
- Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *ICML'09*. 117
- Bengio, Y., Léonard, N., and Courville, A. (2013a). Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv:1308.3432. 275
- Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013b). Generalized denoising auto-encoders as generative models. In *NIPS'2013*. 304, 405, 408
- 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. 333, 403

- Bengio, Y., Thibodeau-Laufer, E., Alain, G., and Yosinski, J. (2014a). Deep generative stochastic networks trainable by backprop. Technical Report arXiv:1306.1091. [275](#)
- Bengio, Y., Thibodeau-Laufer, E., Alain, G., and Yosinski, J. (2014b). Deep generative stochastic networks trainable by backprop. In *Proceedings of the 30th International Conference on Machine Learning (ICML'14)*. [275](#), [405](#), [407](#), [409](#), [410](#)
- Bennett, C. (1976). Efficient estimation of free energy differences from Monte Carlo data. *Journal of Computational Physics*, **22**(2), 245–268. [357](#)
- Berglund, M. and Raiko, T. (2013). Stochastic gradient estimate variance in contrastive divergence and persistent contrastive divergence. *CoRR*, **abs/1312.6002**. [364](#)
- Bergstra, J. (2011). *Incorporating Complex Cells into Neural Networks for Pattern Classification*. Ph.D. thesis, Université de Montréal. [285](#)
- Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. [57](#)
- Besag, J. (1975). Statistical analysis of non-lattice data. *The Statistician*, **24**(3), 179–195. [366](#)
- Bishop, C. M. (1994). Mixture density networks. [113](#)
- Bishop, C. M. (1995). Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN'95*, volume 1, page 141–148. [149](#)
- 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. [78](#), [79](#)
- Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2012). Joint learning of words and meaning representations for open-text semantic parsing. *AISTATS'2012*. [205](#)
- 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. [13](#), [95](#), [109](#)
- Bottou, L. (1991). *Une approche théorique de l'apprentissage connexioniste; applications à la reconnaissance de la parole*. Ph.D. thesis, Université de Paris XI. [231](#)
- Bottou, L. (2011). From machine learning to machine reasoning. Technical report, arXiv.1102.1808. [204](#), [205](#)
- Bottou, L., Fogelman-Soulie, F., Blanchet, P., and Lienard, J. S. (1990). Speaker independent isolated digit recognition: multilayer perceptrons vs dynamic time warping. *Neural Networks*, **3**, 453–465. [231](#)
- 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. [223](#), [230](#), [231](#), [232](#), [233](#), [235](#)

- Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, **59**, 291–294. [282](#)
- 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. [231](#)
- 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. [231](#)
- Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA. [65](#)
- 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. [158](#)
- Brand, M. (2003). Charting a manifold. In *NIPS'2002*, pages 961–968. MIT Press. [98](#), [340](#)
- Breiman, L. (1994). Bagging predictors. *Machine Learning*, **24**(2), 123–140. [142](#)
- Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, **16**(3), 199–231. [5](#)
- Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA. [95](#)
- Brown, P. (1987). *The Acoustic-Modeling problem in Automatic Speech Recognition*. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University. [229](#)
- 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. [250](#)
- Buciluă, 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. [236](#)
- 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. [361](#), [388](#)
- Caruana, R. (1993). Multitask connectionist learning. In *Proc. 1993 Connectionist Models Summer School*, pages 372–379. [154](#)
- Cauchy, A. (1847). Méthode générale pour la résolution de systèmes d'équations simultanées. In *Compte rendu des séances de l'académie des sciences*, pages 536–538. [58](#)
- Cayton, L. (2005). Algorithms for manifold learning. Technical Report CS2008-0923, UCSD. [98](#), [336](#)
- Chapelle, O., Weston, J., and Schölkopf, B. (2003). Cluster kernels for semi-supervised learning. In *NIPS'02*, pages 585–592, Cambridge, MA. MIT Press. [322](#)
- Chapelle, O., Schölkopf, B., and Zien, A., editors (2006). *Semi-Supervised Learning*. MIT Press, Cambridge, MA. [322](#)

- 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>. 15, 17
- 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. 222, 223
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)*. 216
- Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks. arXiv 1412.0233. 311
- Ciresan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012). Multi-column deep neural network for traffic sign classification. *Neural Networks*, **32**, 333–338. 128
- 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. 15, 17
- Coates, A. and Ng, A. Y. (2011). The importance of encoding versus training with sparse coding and vector quantization. In *ICML’2011*. 17
- 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)*. 243
- 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, pages 1337–1345. JMLR Workshop and Conference Proceedings. 15, 17
- Collobert, R. (2004). *Large Scale Machine Learning*. Ph.D. thesis, Université de Paris VI, LIP6. 109
- Comon, P. (1994). Independent component analysis - a new concept? *Signal Processing*, **36**, 287–314. 291, 292
- Cortes, C. and Vapnik, V. (1995). Support vector networks. *Machine Learning*, **20**, 273–297. 13, 95
- Couprise, C., Farabet, C., Najman, L., and LeCun, Y. (2013). Indoor semantic segmentation using depth information. In *International Conference on Learning Representations (ICLR2013)*. 128
- Courville, A., Bergstra, J., and Bengio, Y. (2011). Unsupervised models of images by spike-and-slab RBMs. In *ICML’11*. 258, 401
- 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. 401

- Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory, 2nd Edition*. Wiley-Interscience. 42
- Cox, R. T. (1946). Probability, frequency and reasonable expectation. *American Journal of Physics*, **14**, 1—10. 36
- Crick, F. H. C. and Mitchison, G. (1983). The function of dream sleep. *Nature*, **304**, 111–114. 360
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, **2**, 303–314. 331
- Dauphin, Y. and Bengio, Y. (2013). Stochastic ratio matching of RBMs for sparse high-dimensional inputs. In *NIPS’26*. NIPS Foundation. 369
- 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*. 61, 311
- 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. 241
- de Finetti, B. (1937). La prévision: ses lois logiques, ses sources subjectives. *Annales de l’institut Henri Poincaré*, **7**, 1–68. 36
- Delalleau, O. and Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *NIPS*. 127, 331, 332
- 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*. 14
- Deng, J., Berg, A. C., Li, K., and Fei-Fei, L. (2010). 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. 14
- 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. 223
- 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. 402
- Desjardins, G., Courville, A., and Bengio, Y. (2011). On tracking the partition function. In *NIPS’2011*. 358
- Do, T.-M.-T. and Artières, T. (2010). Neural conditional random fields. In *International Conference on Artificial Intelligence and Statistics*, pages 177–184. 223
- 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. 98, 340

- Doob, J. (1953). *Stochastic processes*. Wiley: New York. 36
- Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, **1**, 75–80. 164, 210
- 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. 109
- Ebrahimi, S., Pal, C., Bouthillier, X., Froumenty, P., Jean, S., Konda, K. R., Vincent, P., Courville, A., and Bengio, Y. (2013). Combining modality specific deep neural network models for emotion recognition in video. In *Emotion Recognition In The Wild Challenge and Workshop (Emotiw2013)*. 128
- El Hihi, S. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS 8*. MIT Press. 217, 221, 222
- ElHihi, S. and Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS'1995*. 213
- 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.* **309**, 311, 312, 313
- 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. 298
- Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013a). Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 128
- 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. 223
- 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. 319
- Fischer, A. and Igel, C. (2011). Bounding the bias of contrastive divergence learning. *Neural Computation*, **23**(3), 664–73. 388
- Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, 179–188. 14, 74
- 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*. 204, 205
- 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. 205
- Frey, B. J. (1998). *Graphical models for machine learning and digital communication*. MIT Press. 206

- 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. [15](#), [16](#), [17](#)
- 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. [14](#)
- Girosi, F. (1994). Regularization theory, radial basis functions and networks. In V. Cherkassky, J. Friedman, and H. Wechsler, editors, *From Statistics to Neural Networks*, volume 136 of *NATO ASI Series*, pages 166–187. Springer Berlin Heidelberg. [126](#)
- Glorot, X., Bordes, A., and Bengio, Y. (2011a). Deep sparse rectifier neural networks. In *AISTATS'2011*. [109](#), [297](#)
- 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)*. [130](#), [297](#)
- Glorot, X., Bordes, A., and Bengio, Y. (2011c). Domain adaptation for large-scale sentiment classification: A deep learning approach. In *ICML'2011*. [297](#), [316](#)
- Gong, S., McKenna, S., and Psarrou, A. (2000). *Dynamic Vision: From Images to Face Recognition*. Imperial College Press. [339](#), [342](#)
- Goodfellow, I., Le, Q., Saxe, A., and Ng, A. (2009). Measuring invariances in deep networks. In *NIPS'2009*, pages 646–654. [285](#), [297](#)
- 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. [71](#)
- Goodfellow, I., Courville, A., and Bengio, Y. (2012). Large-scale feature learning with spike-and-slab sparse coding. In *ICML'2012*. [293](#)
- Goodfellow, I. J. (2010). Technical report: Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal. [187](#)
- 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*. [128](#), [317](#)
- 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. [130](#), [152](#), [243](#)
- Goodfellow, I. J., Mirza, M., Courville, A., and Bengio, Y. (2013b). Multi-prediction deep Boltzmann machines. In *NIPS26*. NIPS Foundation. [367](#), [398](#), [399](#)
- 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. [401](#)

- 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. [158](#)
- Gosset, W. S. (1908). The probable error of a mean. *Biometrika*, **6**(1), 1–25. Originally published under the pseudonym “Student”. [14](#)
- Gouws, S., Bengio, Y., and Corrado, G. (2014). Bilbowa: Fast bilingual distributed representations without word alignments. Technical report, arXiv:1410.2455. [320](#)
- Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Springer. [202](#), [215](#), [216](#), [223](#)
- Graves, A. (2013). Generating sequences with recurrent neural networks. Technical report, arXiv:1308.0850. [114](#), [215](#), [217](#)
- Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, **18**(5), 602–610. [202](#)
- 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. [202](#)
- 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. [223](#)
- 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. [202](#)
- Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *ICASSP’2013*, pages 6645–6649. [202](#), [215](#), [216](#)
- Gutmann, M. and Hyvärinen, 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)*. [370](#)
- 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. [231](#)
- 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. [127](#), [332](#)
- Håstad, J. and Goldmann, M. (1991). On the power of small-depth threshold circuits. *Computational Complexity*, **1**, 113–129. [127](#), [332](#)
- Henaff, M., Jarrett, K., Kavukcuoglu, K., and LeCun, Y. (2011). Unsupervised learning of sparse features for scalable audio classification. In *ISMIR’11*. [298](#)
- 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. [291](#)

- Hinton, G. E. (2000). Training products of experts by minimizing contrastive divergence. Technical Report GCNU TR 2000-004, Gatsby Unit, University College London. [361](#)
- Hinton, G. E. and Roweis, S. (2003). Stochastic neighbor embedding. In *NIPS'2002*. [340](#)
- Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786), 504–507. [287](#), [308](#), [309](#)
- Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, **313**, 504–507. [311](#)
- Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In *NIPS'1993*. [282](#)
- Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, **18**, 1527–1554. [16](#), [17](#), [308](#), [309](#), [311](#), [389](#)
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580. [139](#)
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, T.U. München. [163](#), [210](#), [218](#)
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, **9**(8), 1735–1780. [215](#), [216](#)
- 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. [216](#)
- Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, **2**, 359–366. [331](#)
- Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. [4](#)
- 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. [366](#)
- Hyvönen, H. (1996). Turing machines are recurrent neural networks. In *STeP'96*, pages 13–24. [193](#)
- Hyvärinen, A. (1999). Survey on independent component analysis. *Neural Computing Surveys*, **2**, 94–128. [291](#)
- Hyvärinen, A. (2005a). Estimation of non-normalized statistical models using score matching. *J. Machine Learning Res.*, **6**. [301](#)
- Hyvärinen, A. (2005b). Estimation of non-normalized statistical models using score matching. *Journal of Machine Learning Research*, **6**, 695–709. [367](#)
- Hyvärinen, A. (2007a). Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables. *IEEE Transactions on Neural Networks*, **18**, 1529–1531. [368](#)

- Hyvärinen, A. (2007b). Some extensions of score matching. *Computational Statistics and Data Analysis*, **51**, 2499–2512. [368](#)
- Hyvärinen, A. and Pajunen, P. (1999). Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, **12**(3), 429–439. [292](#)
- Hyvärinen, A., Karhunen, J., and Oja, E. (2001). *Independent Component Analysis*. Wiley-Interscience. [291](#)
- Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixture of local experts. *Neural Computation*, **3**, 79–87. [113](#)
- Jaeger, H. (2003). Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems 15*. [211](#)
- Jaeger, H. (2007a). Discovering multiscale dynamical features with hierarchical echo state networks. Technical report, Jacobs University. [217](#)
- Jaeger, H. (2007b). Echo state network. *Scholarpedia*, **2**(9), 2330. [210](#)
- Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, **304**(5667), 78–80. [17](#), [210](#)
- 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. [324](#)
- Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009a). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE. [15](#), [17](#), [129](#), [130](#)
- Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009b). What is the best multi-stage architecture for object recognition? In *ICCV'09*. [109](#), [298](#)
- Jarzynski, C. (1997). Nonequilibrium equality for free energy differences. *Phys. Rev. Lett.*, **78**, 2690–2693. [357](#)
- Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. [35](#)
- 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. [222](#)
- Jordan, M. I. (1998). *Learning in Graphical Models*. Kluwer, Dordrecht, Netherlands. [13](#)
- Juang, B. H. and Katagiri, S. (1992). Discriminative learning for minimum error classification. *IEEE Transactions on Signal Processing*, **40**(12), 3043–3054. [229](#)
- Jutten, C. and Herault, J. (1991). Blind separation of sources, part I: an adaptive algorithm based on neuromimetic architecture. *Signal Processing*, **24**, 1–10. [291](#)
- Kamyshanska, H. and Memisevic, R. (2015). The potential energy of an autoencoder. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. [304](#)
- 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*. [14](#)

- 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. [222](#)
- 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. [285](#)
- 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. [298](#)
- Kavukcuoglu, K., Ranzato, M.-A., Fergus, R., and LeCun, Y. (2009). Learning invariant features through topographic filter maps. In *CVPR'2009*. [298](#)
- Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., and LeCun, Y. (2010). Learning convolutional feature hierarchies for visual recognition. In *NIPS'2010*. [298](#)
- Kindermann, R. (1980). *Markov Random Fields and Their Applications (Contemporary Mathematics ; V. 1)*. American Mathematical Society. [261](#)
- Kingma, D. and LeCun, Y. (2010a). Regularized estimation of image statistics by score matching. In *NIPS'2010*. [301](#)
- 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. [369](#)
- Kingma, D., Rezende, D., Mohamed, S., and Welling, M. (2014). Semi-supervised learning with deep generative models. In *NIPS'2014*. [275](#)
- Kingma, D. P. (2013). Fast gradient-based inference with continuous latent variable models in auxiliary form. Technical report, arxiv:1306.0733. [275](#)
- Kingma, D. P. and Welling, M. (2014a). Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*. [275, 342, 343](#)
- Kingma, D. P. and Welling, M. (2014b). Efficient gradient-based inference through transformations between bayes nets and neural nets. Technical report, arxiv:1402.0480. [275](#)
- Klementiev, A., Titov, I., and Bhattacharai, B. (2012). Inducing crosslingual distributed representations of words. In *Proceedings of COLING 2012*. [320](#)
- Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press. [227, 273, 279](#)
- Koren, Y. (2009). 1 the bellkor solution to the netflix grand prize. [143](#)
- Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork RNN. In *ICML'2014*. [217, 222](#)
- 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*. [331](#)
- Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto. [14, 258](#)

- 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)*. 15, 17, 71
- Krizhevsky, A., Sutskever, I., and Hinton, G. (2012b). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*. 128, 297
- 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. 223, 229
- 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. 191, 212
- Lappalainen, H., Giannakopoulos, X., Honkela, A., and Karhunen, J. (2000). Nonlinear independent component analysis using ensemble learning: Experiments and discussion. In *Proc. ICA*. Citeseer. 292
- Larochelle, H. and Bengio, Y. (2008a). Classification using discriminative restricted Boltzmann machines. In *ICML'2008*. 285, 411
- Larochelle, H. and Bengio, Y. (2008b). Classification using discriminative restricted Boltzmann machines. In *ICML'08*, pages 536–543. ACM. 322
- Larochelle, H. and Murray, I. (2011). The Neural Autoregressive Distribution Estimator. In *AISTATS'2011*. 205, 209
- Larochelle, H., Erhan, D., and Bengio, Y. (2008). Zero-data learning of new tasks. In *AAAI Conference on Artificial Intelligence*. 319
- 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. 322
- 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*. 15, 17
- Le Roux, N. and Bengio, Y. (2010). Deep belief networks are compact universal approximators. *Neural Computation*, 22(8), 2192–2207. 331
- Le Roux, N., Manzagol, P.-A., and Bengio, Y. (2008). Topmoumoute online natural gradient algorithm. In *NIPS'07*. 116
- LeCun, Y. (1987). *Modèles connexionnistes de l'apprentissage*. Ph.D. thesis, Université de Paris VI. 13, 282
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 541–551. 16
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324. 13, 14, 223, 230, 232

- 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. [17](#)
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998c). Gradient based learning applied to document recognition. *Proc. IEEE*. [16](#)
- Lee, H., Ekanadham, C., and Ng, A. (2008). Sparse deep belief net model for visual area V2. In *NIPS'07*. [285](#)
- 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. [402](#)
- 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. [5](#)
- Leprieur, H. and Haffner, P. (1995). Discriminant learning with minimum memory loss for improved non-vocabulary rejection. In *EUROSPEECH'95*, Madrid, Spain. [229](#)
- 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. [213](#)
- Linde, N. (1992). The machine that changed the world, episode 3. Documentary miniseries. [5](#)
- 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)*. [384](#)
- Lovelace, A. (1842). Notes upon L. F. Menabrea’s “Sketch of the Analytical Engine invented by Charles Babbage”. [4](#)
- Lowerre, B. (1976). *The Harpy Speech Recognition System*. Ph.D. thesis. [224, 229, 233](#)
- Lukoševičius, M. and Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, **3**(3), 127–149. [210](#)
- 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*. [72](#)
- Lyu, S. (2009). Interpretation and generalization of score matching. In *UAI'09*. [368](#)
- 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. [210](#)
- MacKay, D. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. [42](#)
- 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. [364, 368, 369, 385](#)

- Martens, J. and Medabalimi, V. (2014). On the expressive efficiency of sum product networks. *arXiv:1411.7717*. 332
- Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In *Proc. ICML'2011*. ACM. 219
- 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. 366
- 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. 231
- McCullagh, P. and Nelder, J. (1989). *Generalized Linear Models*. Chapman and Hall, London. 110
- 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. 128, 317
- Mesnil, G., Rifai, S., Dauphin, Y., Bengio, Y., and Vincent, P. (2012). Surfing on the manifold. Learning Workshop, Snowbird. 404
- Mikolov, T. (2012). *Statistical Language Models based on Neural Networks*. Ph.D. thesis, Brno University of Technology. 114, 220
- Mikolov, T., Le, Q. V., and Sutskever, I. (2013). Exploiting similarities among languages for machine translation. Technical report, arXiv:1309.4168. 320
- Minka, T. (2005). Divergence measures and message passing. *Microsoft Research Cambridge UK Tech Rep MSRTR2005173*, 72(TR-2005-173). 354
- Minsky, M. L. and Papert, S. A. (1969). *Perceptrons*. MIT Press, Cambridge. 13
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York. 70
- 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. 371
- Montúfar, G. (2014). Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, 26. 331
- 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. 331
- Montufar, G. and Morton, J. (2014). When does a mixture of products contain a product of mixtures? *SIAM Journal on Discrete Mathematics (SIDMA)*. 330
- Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *NIPS'2014*. 329, 332, 333

- 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. [5](#)
- Mozer, M. C. (1992). The induction of multiscale temporal structure. In *NIPS'91*, pages 275–282, San Mateo, CA. Morgan Kaufmann. [213](#), [222](#)
- Murphy, K. P. (2012). *Machine Learning: a Probabilistic Perspective*. MIT Press, Cambridge, MA, USA. [111](#)
- Murray, B. U. I. and Larochelle, H. (2014). A deep and tractable density estimator. In *ICML'2014*. [114](#), [209](#), [210](#)
- 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. [229](#)
- Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*. [109](#), [297](#)
- Narayanan, H. and Mitter, S. (2010). Sample complexity of testing the manifold hypothesis. In *NIPS'2010*. [98](#), [336](#)
- Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics. Springer. [153](#)
- Neal, R. M. (2001). Annealed importance sampling. *Statistics and Computing*, **11**(2), 125–139. [356](#), [357](#)
- Neal, R. M. (2005). Estimating ratios of normalizing constants using linked importance sampling. [357](#), [358](#)
- 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. [14](#)
- 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. [250](#)
- 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. [250](#)
- Niranjan, M. and Fallside, F. (1990). Neural networks and radial basis functions in classifying static speech patterns. *Computer Speech and Language*, **4**, 275–289. [109](#)
- Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer. [65](#), [68](#)
- 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. [285](#), [335](#)

- 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. [296](#)
- Park, H., Amari, S.-I., and Fukumizu, K. (2000). Adaptive natural gradient learning algorithms for various stochastic models. *Neural Networks*, **13**(7), 755 – 764. [116](#)
- Pascanu, R. (2014). *On recurrent and deep networks*. Ph.D. thesis, Université de Montréal. [160](#), [161](#)
- Pascanu, R. and Bengio, Y. (2012). On the difficulty of training recurrent neural networks. Technical Report arXiv:1211.5063, Universite de Montreal. [114](#)
- Pascanu, R. and Bengio, Y. (2013). Revisiting natural gradient for deep networks. Technical report, arXiv:1301.3584. [116](#)
- Pascanu, R., Mikolov, T., and Bengio, Y. (2013a). On the difficulty of training recurrent neural networks. In *ICML’2013*. [114](#), [164](#), [210](#), [213](#), [220](#), [221](#), [222](#)
- 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. [127](#)
- Pascanu, R., Gülcühre, Ç., Cho, K., and Bengio, Y. (2014a). How to construct deep recurrent neural networks. In *ICLR’2014*. [153](#)
- Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014b). How to construct deep recurrent neural networks. In *ICLR’2014*. [215](#), [217](#), [332](#)
- Pascanu, R., Montufar, G., and Bengio, Y. (2014c). On the number of inference regions of deep feed forward networks with piece-wise linear activations. In *ICLR’2014*. [329](#)
- 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. [259](#)
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. [36](#)
- Petersen, K. B. and Pedersen, M. S. (2006). The matrix cookbook. Version 20051003. [20](#)
- Pinto, N., Cox, D. D., and DiCarlo, J. J. (2008). Why is real-world visual object recognition hard? *PLoS Comput Biol*, **4**. [402](#)
- Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, **46**(1), 77–105. [204](#)
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, **4**(5), 1–17. [166](#)
- Poon, H. and Domingos, P. (2011). Sum-product networks: A new deep architecture. In *UAI’2011*, Barcelona, Spain. [127](#), [331](#), [332](#)
- Poundstone, W. (2005). *Fortune’s Formula: The untold story of the scientific betting system that beat the casinos and Wall Street*. Macmillan. [42](#)

- Powell, M. (1987). Radial basis functions for multivariable interpolation: A review. [109](#)
- Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, **77**(2), 257–286. [227](#)
- Rabiner, L. R. and Juang, B. H. (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 257–285. [191](#), [227](#)
- Raiko, T., Yao, L., Cho, K., and Bengio, Y. (2014). Iterative neural autoregressive distribution estimator (NADE-k). Technical report, arXiv:1406.1485. [209](#)
- 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. [17](#)
- 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. [37](#)
- Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007). Efficient learning of sparse representations with an energy-based model. In *NIPS'2006*. [16](#), [296](#), [308](#), [309](#), [311](#)
- Ranzato, M., Boureau, Y., and LeCun, Y. (2008). Sparse feature learning for deep belief networks. In *NIPS'2007*. [296](#)
- Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *ICML'2014*. [275](#)
- 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)*. [319](#), [320](#)
- Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011a). Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML'2011*. [304](#), [305](#), [306](#), [338](#)
- 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)*. [285](#)
- Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., and Glorot, X. (2011c). Higher order contractive auto-encoder. In *ECML PKDD*. [304](#)
- Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., and Muller, X. (2011d). The manifold tangent classifier. In *NIPS'2011*. [350](#), [351](#)
- Rifai, S., Bengio, Y., Dauphin, Y., and Vincent, P. (2012). A generative process for sampling contractive auto-encoders. In *ICML'2012*. [404](#)
- Roberts, S. and Everson, R. (2001). *Independent component analysis: principles and practice*. Cambridge University Press. [292](#)
- Robinson, A. J. and Fallside, F. (1991). A recurrent error propagation network speech recognition system. *Computer Speech and Language*, **5**(3), 259–274. [17](#)

- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**, 386–408. [13](#), [17](#)
- Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, New York. [13](#), [17](#)
- Roweis, S. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, **290**(5500). [98](#), [340](#)
- Rumelhart, D., Hinton, G., and Williams, R. (1986a). Learning representations by back-propagating errors. *Nature*, **323**, 533–536. [13](#), [248](#)
- 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. [14](#), [17](#)
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986c). Learning representations by back-propagating errors. *Nature*, **323**, 533–536. [104](#), [191](#)
- Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986d). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge. [104](#)
- 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. (2014). ImageNet Large Scale Visual Recognition Challenge. [14](#)
- 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. [15](#), [17](#), [309](#), [391](#), [395](#), [398](#)
- 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. [400](#), [409](#)
- 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. [322](#)
- 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. [357](#)
- 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. [17](#)
- Schaul, T., Zhang, S., and LeCun, Y. (2012). No More Pesky Learning Rates. Technical report, New York University, arxiv 1206.1106. [171](#)
- Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, **4**(2), 234–242. [16](#), [217](#)
- Schölkopf, B. and Smola, A. (2002). *Learning with kernels*. MIT Press. [95](#)
- Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, **10**, 1299–1319. [98](#), [340](#)

- Schölkopf, B., Burges, C. J. C., and Smola, A. J. (1999). *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA. 13, 109, 128
- Schulz, H. and Behnke, S. (2012). Learning two-layer contractive encodings. In *ICANN'2012*, pages 620–628. 305
- Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681. 202
- 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. 128
- Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. *International Conference on Learning Representations*. 71
- Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3), 379—423. 42
- Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, 37(1), 10–21. 42
- Shilov, G. (1977). *Linear Algebra*. Dover Books on Mathematics Series. Dover Publications. 20
- Siegelmann, H. (1995). Computation beyond the Turing limit. *Science*, 268(5210), 545–548. 193
- Siegelmann, H. and Sontag, E. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, 4(6), 77–80. 193
- 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. 164
- 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*. 349, 350, 351
- Simard, P. Y., LeCun, Y., and Denker, J. (1993). Efficient pattern recognition using a new transformation distance. In *NIPS'92*. 348
- 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. 348
- 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. 149
- 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. 266, 277
- 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*. 205

- 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)*. 205
- 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*. 205
- 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*. 205
- Solla, S. A., Levin, E., and Fleisher, M. (1988). Accelerated learning in layered neural networks. *Complex Systems*, **2**, 625–639. 112
- 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. 158
- Srivastava, N. and Salakhutdinov, R. (2012). Multimodal learning with deep Boltzmann machines. In *NIPS'2012*. 321
- 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. 151, 152, 153, 398
- 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. 223
- Sutskever, I. (2012). *Training Recurrent Neural Networks*. Ph.D. thesis, Departement of computer science, University of Toronto. 211, 219
- 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. 364
- Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *ICML*. 167, 211, 219
- Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. Technical report, arXiv:1409.3215. 215, 216
- Swersky, K. (2010). *Inductive Principles for Learning Restricted Boltzmann Machines*. Master's thesis, University of British Columbia. 302
- 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. 369
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. Technical report, arXiv:1409.4842. 15, 17
- Tenenbaum, J., de Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, **290**(5500), 2319–2323. 98, 312, 313, 340

- Thrun, S. (1995). Learning to play the game of chess. In *NIPS'1994*. 350
- Tibshirani, R. J. (1995). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society B*, **58**, 267–288. 137
- 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. 364, 388
- Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal components analysis. *Journal of the Royal Statistical Society B*, **61**(3), 611–622. 291
- Uria, B., Murray, I., and Larochelle, H. (2013). Rnade: The real-valued neural autoregressive density-estimator. In *NIPS'2013*. 208, 209
- Utgoff, P. E. and Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation*, **14**, 2497–2539. 16
- van der Maaten, L. and Hinton, G. E. (2008a). Visualizing data using t-SNE. *J. Machine Learning Res.*, **9**. 312, 340, 343
- van der Maaten, L. and Hinton, G. E. (2008b). Visualizing data using t-SNE. *Journal of Machine Learning Research*, **9**, 2579–2605. 313
- Vapnik, V. N. (1982). *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, Berlin. 78, 79
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer, New York. 78, 79, 81
- 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. 78, 79
- Vincent, P. (2011a). A connection between score matching and denoising autoencoders. *Neural Computation*, **23**(7). 301, 302, 304, 404
- Vincent, P. (2011b). A connection between score matching and denoising autoencoders. *Neural Computation*, **23**(7), 1661–1674. 369, 405
- Vincent, P. and Bengio, Y. (2003). Manifold Parzen windows. In *NIPS'2002*. MIT Press. 340
- Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *ICML 2008*. 298
- 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**. 298
- Wager, S., Wang, S., and Liang, P. (2013). Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems 26*, pages 351–359. 153
- 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. 191

- Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML'2013*. 154
- Wang, S. and Manning, C. (2013). Fast dropout training. In *ICML'2013*. 153
- Warde-Farley, D., Goodfellow, I. J., Courville, A., and Bengio, Y. (2014). An empirical analysis of dropout in piecewise linear networks. In *ICLR'2014*. 153
- Weinberger, K. Q. and Saul, L. K. (2004). Unsupervised learning of image manifolds by semidefinite programming. In *CVPR'2004*, pages 988–995. 98, 340
- 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. 322
- 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. 205
- White, H. (1990). Connectionist nonparametric regression: Multilayer feedforward networks can learn arbitrary mappings. *Neural Networks*, **3**(5), 535–549. 126
- 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, 14, 15, 17
- Wikipedia (2015). List of animals by number of neurons — wikipedia, the free encyclopedia. [Online; accessed 4-March-2015]. 15, 17
- Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In *NIPS'95*, pages 514–520. MIT Press, Cambridge, MA. 128
- Wolpert, D. H. (1996). The lack of a priori distinction between learning algorithms. *Neural Computation*, **8**(7), 1341–1390. 127
- 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. 153
- Xu, L. and Jordan, M. I. (1996). On convergence properties of the EM algorithm for gaussian mixtures. *Neural Computation*, **8**, 129–151. 228
- Younes, L. (1998). On the convergence of Markovian stochastic algorithms with rapidly decreasing ergodicity rates. In *Stochastics and Stochastics Models*, pages 177–228. 364, 388
- 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. 330
- Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *ECCV'14*. 9, 71
- Zhou, J. and Troyanskaya, O. G. (2014). Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. In *ICML'2014*. 410
- Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, **67**(2), 301–320. 116
- Zöhrer, M. and Pernkopf, F. (2014). General stochastic networks for classification. In *NIPS'2014*. 411

Index

- L^p norm, 26
- Active constraint, 68
- ADALINE, *see* Adaptive Linear Element
- Adaptive Linear Element, 13, 15, 17
- AIS, *see* annealed importance sampling
- Almost everywhere, 52
- Ancestral sampling, 279
- Annealed importance sampling, 354, 396
- Approximate inference, 274
- Artificial intelligence, 4
- Asymptotically unbiased, 84
- Autoencoder, 7
- Bagging, 142
- Bayes' rule, 51
- Bayesian network, *see* directed graphical model
- Bayesian probability, 37
- Beam Search, 233
- Belief network, *see* directed graphical model
- Bernoulli distribution, 44
- Boltzmann distribution, 265
- Boltzmann machine, 265
- Broadcasting, 22
- Calculus of variations, 378
- CD, *see* contrastive divergence
- Centering trick (DBM), 398
- Central limit theorem, 45
- Chain rule of probability, 40
- Chess, 4
- Chord, 272
- Chordal graph, 272
- Classical regularization, 132
- Classification, 71
- Cliffs, 159
- Clipping the gradient, 220
- Clique potential, *see* factor (graphical model)
- CNN, *see* convolutional neural network
- Collider, *see* explaining away
- Computer vision, 241
- Conditional computation, *see* dynamically structured nets, 237
- Conditional independence, 40
- Conditional probability, 39
- Constrained optimization, 67
- Context-specific independence, 267
- Contrast, 242
- Contrastive divergence, 361, 397, 398
- Convolution, 173, 401
- Convolutional neural network, 173
- Coordinate descent, 168, 169, 398
- Correlation, 41
- Cost function, *see* objective function
- Covariance, 41
- Covariance matrix, 42
- curse of dimensionality, 98
- Cyc, 5
- D-separation, 266
- Dataset augmentation, 242, 247
- DBM, *see* deep Boltzmann machine
- Decoder, 7
- Deep belief network, 17, 372, 383, 389, 402
- Deep Blue, 4
- Deep Boltzmann machine, 15, 17, 372, 383, 391, 398, 402
- Deep learning, 4, 7
- Denoising score matching, 369
- Density estimation, 71
- Derivative, 58
- Detector layer, 178
- Dirac delta function, 47
- Directed graphical model, 259
- Directional derivative, 62
- Distributed Representation, 325
- domain adaptation, 315
- Dot product, 23

Doubly block circulant matrix, 175
 Dream sleep, 360, 381
 DropConnect, 154
 Dropout, 151, 398
 Dynamically structured networks, 237
 E-step, 375
 Early stopping, 116, 143, 146, 148
 EBM, *see* energy-based model
 Echo state network, 15, 17
 Effective number of parameters, 134
 Eigendecomposition, 29
 Eigenvalue, 29
 Eigenvector, 29
 ELBO, *see* evidence lower bound
 Element-wise product, *see* Hadamard product
 EM, *see* expectation maximization
 Embedding, 339
 Empirical distribution, 47
 Empirical risk, 156
 Empirical risk minimization, 157
 Encoder, 7
 Energy function, 265
 Energy-based model, 265, 392
 Ensemble methods, 142
 Epoch, 158, 166
 Equality constraint, 67
 Equivariance, 176
 Error function, *see* objective function
 Euclidean norm, 27
 Euler-Lagrange equation, 379
 Evidence lower bound, 372, 374–376, 391
 Expectation, 41
 Expectation maximization, 375
 Expected value, *see* expectation
 Explaining away, 268
 Factor (graphical model), 262
 Factor graph, 272
 Factors of variation, 7
 Frequentist probability, 37
 Functional derivatives, 378
 Gaussian distribution, *see* Normal distribution 45
 Gaussian mixture, 48
 GCN, *see* Global contrast normalization
 Generalized Lagrange function, *see* Generalized Lagrangian
 Generalized Lagrangian, 67
 Gibbs distribution, 263
 Gibbs sampling, 280
 Global contrast normalization, 243
 Gradient, 62
 Gradient clipping, 220
 Gradient descent, 62
 Graph Transformer, 232
 Graphical model, *see* structured probabilistic model
 Greedy layer-wise unsupervised pre-training, 308
 Hadamard product, 23
 Harmonium, *see* Restricted Boltzmann machine 277
 Harmony theory, 266
 Helmholtz free energy, *see* evidence lower bound
 Hessian matrix, 63
 Hidden layer, 9
 Identity matrix, 24
 Immorality, 270
 Independence, 40
 Inequality constraint, 67
 Inference, 257, 274, 372, 374–376, 378, 380
 Invariance, 181
 Jacobian matrix, 52, 62
 Joint probability, 38
 Karush-Kuhn-Tucker conditions, 68
 Karush–Kuhn–Tucker, 67
 Kernel (convolution), 174
 KKT, *see* Karush–Kuhn–Tucker
 KKT conditions, *see* Karush–Kuhn–Tucker conditions
 KL divergence, *see* Kullback–Leibler divergence 43
 Knowledge base, 5
 Kullback–Leibler divergence, 43
 Lagrange multipliers, 67, 68, 379
 Lagrangian, *see* Generalized Lagrangian 67
 Latent variable, 286
 Line search, 62
 Linear combination, 25
 Linear dependence, 26
 Local conditional probability distribution, 260
 Logistic regression, 5
 Logistic sigmoid, 48
 Loop, 272
 Loss function, *see* objective function

M-step, 375
 Machine learning, 5
 Manifold hypothesis, 336
 manifold hypothesis, 98
 Manifold learning, 97, 336
 MAP inference, 376
 Marginal probability, 39
 Markov chain, 279
 Markov network, see undirected model 261
 Markov random field, see undirected model 261
 Matrix, 21
 Matrix inverse, 24
 Matrix product, 22
 Max pooling, 181
 Mean field, 397, 398
 Measure theory, 51
 Measure zero, 52
 Method of steepest descent, *see* gradient descent
 Missing inputs, 71
 Mixing (Markov chain), 281
 Mixture distribution, 48
 MLP, *see* multilayer perception
 MNIST, 398
 Model averaging, 142
 Model compression, 236
 Moore-Penrose pseudoinverse, 140
 Moralized graph, 270
 MP-DBM, *see* multi-prediction DBM
 MRF (Markov Random Field), see undirected model 261
 Multi-modal learning, 321
 Multi-prediction DBM, 397, 398
 Multi-task learning, 154
 Multilayer perception, 8
 Multilayer perceptron, 17
 Multinomial distribution, 44
 Multinoulli distribution, 44
 Naive Bayes, 5, 53
 Nat, 42
 natural image, 256
 Negative definite, 63
 Negative phase, 360
 Neocognitron, 15, 17
 Nesterov momentum, 167
 Netflix Grand Prize, 143
 Noise-contrastive estimation, 370
 Norm, 26
 Normal distribution, 45
 Normal equations, 135
 Object detection, 241
 Object recognition, 241
 Objective function, 58
 one-shot learning, 319
 Orthogonality, 28
 Overfitting, 79
 Parameter sharing, 176
 Partial derivative, 58
 Partition function, 101, 263, 352, 397
 PCA, *see* principal components analysis
 PCD, *see* stochastic maximum likelihood
 Perceptron, 13, 17
 Persistent contrastive divergence, *see* stochastic maximum likelihood
 Pooling, 173, 402
 Positive definite, 63
 Positive phase, 360
 Pre-training, 308
 Precision (of a normal distribution), 45, 47
 Predictive sparse decomposition, 285, 296
 Preprocessing, 242
 Principal components analysis, 32, 244, 372
 Principle components analysis, 91
 Probabilistic max pooling, 402
 Probability density function, 38
 Probability distribution, 37
 Probability mass function, 38
 Product rule of probability, *see* chain rule of probability
 PSD, *see* predictive sparse decomposition
 Pseudolikelihood, 365
 Random variable, 37
 Ratio matching, 368
 RBM, *see* restricted Boltzmann machine
 Receptive field, 177
 Recurrent network, 17
 Regression, 71
 Regularization, 131
 Representation learning, 5
 Restricted Boltzmann machine, 277, 372, 382, 383, 398, 400–402
 Ridge regression, 133
 Risk, 156
 Scalar, 20
 Score matching, 367

Second derivative, 62
 Second derivative test, 63
 Self-information, 42
 Separable convolution, 190
 Separation (probabilistic modeling), 266
 SGD, *see* stochastic gradient descent, *see also* stochastic gradient descent
 Shannon entropy, 42, 379
 Sigmoid, *see* logistic sigmoid
 Sigmoid belief network, 17
 Singular value decomposition, 30, 140
 SML, *see* stochastic maximum likelihood
 Softmax, 111
 Softplus, 48
 Spam detection, 5
 Sparse coding, 292, 372
 spectral radius, 211
 Sphering, *see* Whitening, 244
 Spike and slab restricted Boltzmann machine, 401
 Square matrix, 26
 ssRBM, *see* spike and slab restricted Boltzmann machine
 Standard deviation, 41
 Statistic, 83
 Steepest descent, *see* gradient descent
 Stochastic gradient descent, 158, 165, 398
 Stochastic maximum likelihood, 364, 397, 398
 Stochastic pooling, 154
 Structure learning, 273
 Structured output, 71
 Structured probabilistic model, 255
 Sum rule of probability, 39
 Surrogate loss function, 157
 SVD, *see* singular value decomposition
 Symmetric matrix, 28
 Tangent plane, 340
 Tensor, 21
 Tiled convolution, 186
 Toeplitz matrix, 175
 Trace operator, 31
 Transcription, 71
 Transfer learning, 315
 Transpose, 22
 Triangle inequality, 27
 Triangulated graph, *see* chordal graph
 Unbiased, 84
 Underfitting, 79

Undirected model, 261
 Uniform distribution, 38
 Unit norm, 28
 Unnormalized probability distribution, 262
 Unsupervised pre-training, 308
 V-structure, *see* explaining away
 Vapnik-Chervonenkis dimension, 78
 Variance, 41
 Variational derivatives, *see* functional derivatives
 Variational free energy, *see* evidence lower bound
 VC dimension, *see* Vapnik-Chervonenkis dimension, 41
 Vector, 20
 Visible layer, 9
 Viterbi decoding, 226
 Weight decay, 133
 Whitening, 244
 ZCA, *see* zero-phase components analysis
 zero-data learning, 319
 Zero-phase components analysis, 244
 zero-shot learning, 319