

O'REILLY®

US-Bestseller zu
Deep Learning

Praxiseinstieg Machine Learning mit Scikit-Learn & TensorFlow

KONZEPTE, TOOLS UND TECHNIKEN
FÜR INTELLIGENTE SYSTEME



Aurélien Géron
Übersetzung von Kristian Rother

klaus.engel@t-systems.com
plus_7576a8882a38a7e9a10a-20481-13111-1

Papier
plus⁺
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus⁺: www.oreilly.plus

Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow

*Konzepte, Tools und Techniken
für intelligente Systeme*

Aurélien Géron

*Deutsche Übersetzung von
Kristian Rother*

O'REILLY®

Aurélien Géron

Lektorat: Alexandra Follenius

Übersetzung: Kristian Rother

Korrektorat: Claudia Lötschert, www.richtiger-text.de

Satz: III-Satz, www.drei-satz.de

Herstellung: Susanne Bröckelmann

Umschlaggestaltung: Michael Oréal, www.oreal.de

Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:

Print 978-3-96009-061-8

PDF 978-3-96010-114-7

ePub 978-3-96010-115-4

mobi 978-3-96010-116-1

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.

O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

1. Auflage 2018

Copyright © 2018 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Authorized German translation of the English edition of *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, ISBN 978-1-491-96229-9 © 2017 Aurélien Géron. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

5 4 3 2 1 0

Inhalt

Vorwort	xv
---------------	----

Teil I Die Grundlagen des Machine Learning

1 Die Machine Learning-Umgebung	3
Was ist Machine Learning?	4
Warum wird Machine Learning verwendet?	4
Arten von Machine-Learning-Systemen	7
Überwachtes/unüberwachtes Lernen	8
Batch- und Online-Learning	14
Instanzbasiertes versus modellbasiertes Lernen	17
Die wichtigsten Herausforderungen beim Machine Learning	22
Unzureichende Menge an Trainingsdaten	23
Nicht repräsentative Trainingsdaten	24
Minderwertige Daten	26
Irrelevante Merkmale	26
Overfitting der Trainingsdaten	26
Underfitting der Trainingsdaten	29
Zusammenfassung	29
Testen und Validieren	30
Übungen	32

2 Ein Machine-Learning-Projekt von A bis Z	33
Der Umgang mit realen Daten	33
Betrachte das Gesamtbild	35
Die Aufgabe abstecken	35
Wähle ein Qualitätsmaß aus	37
Überprüfe die Annahmen	40
Beschaffe die Daten	40
Erstelle eine Arbeitsumgebung	40
Die Daten herunterladen	44
Wirf einen kurzen Blick auf die Datenstruktur	45
Erstelle einen Testdatensatz	49
Erkunde und visualisiere die Daten, um Erkenntnisse zu gewinnen	53
Visualisieren geografischer Daten	53
Suche nach Korrelationen	56
Experimentieren mit Kombinationen von Merkmalen	58
Bereite die Daten für Machine-Learning-Algorithmen vor	60
Aufbereiten der Daten	60
Bearbeiten von Text und kategorischen Merkmalen	63
Eigene Transformer	65
Skalieren von Merkmalen	66
Pipelines zur Transformation	67
Wähle ein Modell aus und trainiere es	69
Trainieren und Auswerten auf dem Trainingsdatensatz	69
Bessere Auswertung mittels Kreuzvalidierung	71
Optimiere das Modell	73
Gittersuche	73
Zufällige Suche	75
Ensemble-Methoden	76
Analysiere die besten Modelle und ihre Fehler	76
Evaluiere das System auf dem Testdatensatz	77
Nimm das System in Betrieb, überwache und warte es	78
Probieren Sie es aus!	78
Übungen	79
3 Klassifikation	81
MNIST	81
Trainieren eines binären Klassifikators	83
Qualitätsmaße	84
Messen der Genauigkeit über Kreuzvalidierung	84
Konfusionsmatrix	86
Relevanz und Sensitivität	88

Die Wechselbeziehung zwischen Relevanz und Sensitivität	89
Die ROC-Kurve	92
Klassifikatoren mit mehreren Kategorien	96
Fehleranalyse	98
Klassifikation mit mehreren Labels	102
Klassifikation mit mehreren Ausgaben	103
Übungsaufgaben	105
4 Trainieren von Modellen	107
Lineare Regression	108
Die Normalengleichung	110
Komplexität der Berechnung	112
Das Gradientenverfahren	112
Batch-Gradientenverfahren	116
Stochastisches Gradientenverfahren	118
Mini-Batch-Gradientenverfahren	121
Polynomielle Regression	122
Lernkurven	124
Regularisierte lineare Modelle	128
Ridge-Regression	129
Lasso-Regression	131
Elastic Net	133
Early Stopping	134
Logistische Regression	135
Abschätzen von Wahrscheinlichkeiten	135
Trainieren und Kostenfunktion	136
Entscheidungsgrenzen	138
Softmax-Regression	140
Übungen	144
5 Support Vector Machines	145
Lineare Klassifikation mit SVMs	145
Soft-Margin-Klassifikation	146
Nichtlineare SVM-Klassifikation	149
Polynomieller Kernel	150
Hinzufügen von ähnlichkeitsbasierten Merkmalen	151
Der Gaußsche RBF-Kernel	152
Komplexität der Berechnung	153
SVM-Regression	154
Hinter den Kulissen	156
Entscheidungsfunktion und Vorhersagen	156
Zielfunktionen beim Trainieren	157

Quadratische Programme	159
Das duale Problem	160
Kernel-SVM	161
Online-SVMs	163
Übungen	165
6 Entscheidungsbäume	167
Trainieren und Visualisieren eines Entscheidungsbaums	167
Vorhersagen treffen	168
Schätzen von Wahrscheinlichkeiten für Kategorien	171
Der CART-Trainings-Algorithmus	171
Komplexität der Berechnung	172
Gini-Unreinheit oder Entropie?	173
Hyperparameter zur Regularisierung	173
Regression	175
Instabilität	177
Übungen	178
7 Ensemble Learning und Random Forests	181
Abstimmverfahren unter Klassifikatoren	181
Bagging und Pasting	185
Bagging und Pasting in Scikit-Learn	186
Out-of-Bag-Evaluation	187
Zufällige Patches und Subräume	188
Random Forests	189
Extra-Trees	190
Wichtigkeit von Merkmalen	190
Boosting	191
AdaBoost	192
Gradient Boosting	195
Stacking	200
Übungen	203
8 Dimensionsreduktion	205
Der Fluch der Dimensionalität	206
Die wichtigsten Ansätze zur Dimensionsreduktion	207
Projektion	207
Manifold Learning	210
Hauptkomponentenzerlegung (PCA)	211
Erhalten der Varianz	211
Hauptkomponenten	212
Die Projektion auf d Dimensionen	213

Verwenden von Scikit-Learn	214
Der Anteil erklärter Varianz	214
Auswählen der richtigen Anzahl Dimensionen	215
PCA als Komprimierungsverfahren	216
Inkrementelle PCA	217
Randomisierte PCA	217
Kernel PCA	218
Auswahl eines Kernels und Optimierung der Hyperparameter	219
LLE	221
Weitere Techniken zur Dimensionsreduktion	223
Übungen	224

Teil II Neuronale Netze und Deep Learning

9 Einsatzbereit mit TensorFlow	227
Installation	230
Erstellen und Ausführen eines ersten Graphen	230
Graphen verwalten	232
Lebenszyklus des Werts von Knoten	232
Lineare Regression mit TensorFlow	233
Implementieren des Gradientenverfahrens	235
Manuelle Berechnung der Gradienten	235
Verwenden von Autodiff	236
Verwenden von Optimierungsverfahren	237
Daten in den Trainingsalgorithmus einspeisen	238
Modelle speichern und wiederherstellen	239
Graphen und Lernkurven mit TensorBoard visualisieren	240
Name Scopes	244
Modularität	245
Teilen von Variablen	247
Übungen	250
10 Einführung in künstliche neuronale Netze	253
Von biologischen zu künstlichen Neuronen	254
Biologische Neuronen	255
Logische Berechnungen mit Neuronen	256
Das Perzeptron	257
Mehrschichtiges Perzeptron und Backpropagation	261
Ein MLP mit der TensorFlow-API trainieren	264
Ein DNN direkt mit TensorFlow trainieren	265
Konstruktionsphase	265

Ausführungsphase	269
Verwenden des neuronalen Netzes	270
Feinabstimmung der Hyperparameter eines neuronalen Netzes	270
Anzahl verborgener Schichten	271
Anzahl Neuronen pro verborgene Schicht	272
Aktivierungsfunktionen	272
Übungen	273
11 Trainieren von Deep-Learning-Netzen	275
Das Problem schwindender/explodierender Gradienten	275
Initialisierung nach Xavier und He	277
Nicht sättigende Aktivierungsfunktionen	279
Batch-Normalisierung	282
Gradient Clipping	286
Wiederverwenden vortrainierter Schichten	287
Wiederverwenden eines TensorFlow-Modells	288
Modelle aus anderen Frameworks wiederverwenden	290
Einfrieren der unteren Schichten	291
Caching der eingefrorenen Schichten	291
Verändern, Auslassen oder Ersetzen der oberen Schichten	292
Modell-Zoos	293
Unüberwachtes Vortrainieren	293
Vortrainieren anhand einer Hilfsaufgabe	294
Schnellere Optimierer	295
Momentum Optimization	296
Beschleunigter Gradient nach Nesterov	297
AdaGrad	298
RMSProp	300
Adam-Optimierung	300
Scheduling der Lernrate	302
Vermeiden von Overfitting durch Regularisierung	305
Early Stopping	305
ℓ_1 - und ℓ_2 -Regularisierung	305
Drop-out	306
Max-Norm-Regularisierung	309
Data Augmentation	311
Praktische Tipps	312
Übungen	313
12 TensorFlow über mehrere Geräte und Server verteilen	317
Mehrere Recheneinheiten auf einem Computer	318
Installation	318

Das RAM der GPU verwalten	321
Operationen auf Recheneinheiten platzieren	322
Paralleles Ausführen	326
Control Dependencies	327
Mehrere Recheneinheiten auf mehreren Servern	328
Öffnen einer Session	329
Master- und Worker-Dienste	330
Operationen auf Tasks pinnen	330
Sharding von Variablen über mehrere Parameterserver	331
Zustände mit Resource Containers zwischen Sessions teilen	332
Asynchrone Kommunikation mit TensorFlow-Queues	334
Daten direkt aus dem Graphen laden	339
Parallelisieren neuronaler Netze auf einem TensorFlow-Cluster	346
Ein neuronales Netz pro Recheneinheit	346
In-Graph- und Between-Graph-Replikation	347
Parallelisierte Modelle	349
Parallelisierte Daten	351
Übungen	356
13 Convolutional Neural Networks	359
Der Aufbau des visuellen Cortex	360
Convolutional Layer	361
Filter	363
Stapeln mehrerer Feature Maps	364
Implementierung in TensorFlow	366
Speicherbedarf	368
Pooling Layer	369
Architekturen von CNNs	371
LeNet-5	372
AlexNet	373
GoogLeNet	375
ResNet	378
Übungen	382
14 Rekurrente neuronale Netze	385
Rekurrente Neuronen	386
Gedächtniszellen	388
Ein- und Ausgabesequenzen	388
Einfache RNNs in TensorFlow	390
Statisches Aufrollen entlang der Zeitachse	391
Dynamisches Aufrollen entlang der Zeitachse	393
Eingabesequenzen unterschiedlicher Länge	393

Ausgabesequenzen unterschiedlicher Länge	394
Trainieren von RNNs	395
Trainieren eines Sequenz-Klassifikators	395
Trainieren der Vorhersage von Zeitreihen	397
Kreative RNNs	401
Deep-RNNs.	402
Ein Deep-RNN über mehrere GPUs verteilen	403
Drop-out verwenden	404
Die Schwierigkeit, über viele Schritte zu trainieren	405
LSTM-Zellen	406
Peephole-Verbindungen	409
GRU-Zellen.	410
Natürliche Sprachverarbeitung	411
Word Embeddings	411
Ein Encoder-Decoder-Netz zur maschinellen Übersetzung	413
Übungen	416
15 Autoencoder	419
Effiziente Repräsentation von Daten	419
Hauptkomponentenzerlegung mit einem unvollständigen	
linearen Autoencoder	421
Stacked Autoencoder	423
Implementierung mit TensorFlow	423
Kopplung von Gewichten	424
Trainieren mehrerer Autoencoder nacheinander	426
Visualisieren der Rekonstruktionen	428
Visualisieren von Merkmalen	429
Unüberwachtes Vortrainieren mit Stacked Autoencoder	430
Denoising Autoencoder	432
Implementierung mit TensorFlow	433
Sparse Autoencoder	434
Implementierung in TensorFlow	435
Variational Autoencoder	436
Generieren von Ziffern	439
Weitere Autoencoder	440
Übungen	441
16 Reinforcement Learning	445
Lernen zum Optimieren von Belohnungen	446
Suche nach Policies	447
Einführung in OpenAI Gym	449
Neuronale Netze als Policies	452

Auswerten von Aktionen: Das Credit-Assignment-Problem	455
Policy-Gradienten	456
Markov-Entscheidungsprozesse	461
Temporal Difference Learning und Q-Learning	465
Erkundungspolicies	467
Approximatives Q-Learning	467
Ms. Pac-Man mit dem DQN-Algorithmus spielen lernen	469
Übungen	477
Vielen Dank!	477
A Lösungen zu den Übungsaufgaben	479
B Checkliste für Machine-Learning-Projekte	509
C Das duale Problem bei SVMs	515
D Autodiff	519
E Weitere verbreitete Architekturen neuronaler Netze	527
Index	537

Vorwort

Der Machine-Learning-Tsunami

Im Jahr 2006 erschien ein Artikel von Geoffrey Hinton et al.¹, in dem vorgestellt wurde, wie sich ein neuronales Netz zum Erkennen handgeschriebener Ziffern mit ausgezeichneter Genauigkeit (> 98%) trainieren lässt. Die Autoren nannten diese Technik »Deep Learning.« Zu dieser Zeit wurde das Trainieren eines Deep-Learning-Netzes im Allgemeinen als unmöglich angesehen,² und die meisten Forscher hatten die Idee in den 1990ern aufgegeben. Dieser Artikel ließ das Interesse der wissenschaftlichen Gemeinde wieder aufleben, und schon nach kurzer Zeit zeigten weitere Artikel, dass Deep Learning nicht nur möglich war, sondern umwerfende Dinge vollbringen konnte, zu denen kein anderes Machine-Learning-(ML-)Verfahren auch nur annähernd in der Lage war (mithilfe enormer Rechenleistung und riesiger Datenmengen). Dieser Enthusiasmus breitete sich schnell auf weitere Teilgebiete des Machine Learning aus.

Zehn Jahre später hat Machine Learning ganze Industriezweige erobert: Es ist zu einem Herzstück heutiger Spitzentechnologien geworden und dient zum Ranking von Suchergebnissen im Web, kümmert sich um die Spracherkennung Ihres Smartphones, gibt Empfehlungen für Videos und schlägt den Weltmeister im Brettspiel Go. Über kurz oder lang wird ML vermutlich auch Ihr Auto steuern.

Machine Learning in Ihren Projekten

Deshalb interessieren Sie sich natürlich auch für Machine Learning und möchten an der Party teilnehmen!

1 Verfügbar auf der Homepage von Hinton unter <http://www.cs.toronto.edu/~hinton/>.

2 Obwohl die Konvolutionsnetze von Yann Lecun bei der Bilderkennung seit den 1990ern gut funktioniert hatten, auch wenn sie nicht allgemein anwendbar waren.

Womöglich möchten Sie Ihrem selbst gebauten Roboter einen eigenen Denkapparat geben? Ihn Gesichter erkennen lassen? Oder lernen lassen, herumzulaufen?

Oder vielleicht besitzt Ihr Unternehmen Unmengen Daten (Logdateien, Finanzdaten, Produktionsdaten, Sensordaten, Hotline-Statistiken, Personalstatistiken und so weiter), und Sie könnten vermutlich einige verborgene Schätze heben, wenn Sie nur wüssten, wo Sie danach suchen müssten, beispielsweise:

- Kundensegmente finden und für jede Gruppe die beste Marketingstrategie entwickeln.
- Jedem Kunden anhand des Kaufverhaltens ähnlicher Kunden Produktempfehlungen geben.
- Betrügerischen Transaktionen mit hoher Wahrscheinlichkeit erkennen.
- Den Unternehmensgewinn im nächsten Jahr vorhersagen.
- Und vieles mehr. (<https://www.kaggle.com/wiki/DataScienceUseCases>)

Was immer der Grund ist, Sie haben beschlossen, Machine Learning zu erlernen und in Ihren Projekten umzusetzen. Eine ausgezeichnete Idee!

Ziel und Ansatz

Dieses Buch geht davon aus, dass Sie noch so gut wie nichts über Machine Learning wissen. Unser Ziel ist es, Ihnen die Grundbegriffe, ein Grundverständnis und die Werkzeuge an die Hand zu geben, mit denen Sie Programme zum *Lernen aus Daten* entwickeln können.

Wir werden eine Vielzahl Techniken besprechen, von den einfachsten und am häufigsten eingesetzten (wie der linearen Regression) bis zu einigen Deep-Learning-Verfahren, die regelmäßig Wettbewerbe gewinnen.

Anstatt eigene Übungsversionen jedes Algorithmus zu entwickeln, werden wir dazu für den Produktionsbetrieb geschaffene Python-Frameworks verwenden:

- *Scikit-Learn* (<http://scikit-learn.org/>) ist sehr einfach zu verwenden, enthält aber effiziente Implementierungen vieler Machine-Learning-Algorithmen. Damit ist es ein großartiger Ausgangspunkt, um Machine Learning zu erlernen.
- *TensorFlow* (<http://tensorflow.org/>) ist eine komplexere Bibliothek für verteiltes Rechnen anhand von Datenfluss-Graphen. Mit ihr können Sie sehr große neuronale Netze effizient trainieren und ausführen, indem Sie die Berechnungen auf bis zu Tausende Server mit mehreren GPUs verlagern. TensorFlow wurde von Google entwickelt und läuft in vielen großflächigen Machine-Learning-Anwendungen. Die Bibliothek wurde im November 2015 als Open Source veröffentlicht.

Dieses Buch verfolgt einen praxisorientierten Ansatz, bei dem Sie ein intuitives Verständnis von Machine Learning entwickeln, indem Sie sich mit konkreten Beispie-

len und ein klein wenig Theorie beschäftigen. Auch wenn Sie dieses Buch lesen können, ohne Ihren Laptop in die Hand zu nehmen, empfehlen wir Ihnen, mit den als Jupyter Notebooks unter <https://github.com/ageron/handson-ml> verfügbaren Codebeispielen herumzuexperimentieren.

Voraussetzungen

Dieses Buch geht davon aus, dass Sie ein wenig Programmiererfahrung mit Python haben und dass Sie mit den wichtigsten wissenschaftlichen Bibliotheken in Python vertraut sind, insbesondere mit *NumPy* (<http://numpy.org/>), *Pandas* (<http://pandas.pydata.org/>) und *Matplotlib* (<http://matplotlib.org/>).

Wenn Sie sich für das interessieren, was hinter den Kulissen passiert, sollten Sie ein Grundverständnis von Oberstufenmathematik haben (Analysis, lineare Algebra, Wahrscheinlichkeiten und Statistik).

Wenn Sie Python noch nicht kennen, ist <http://learnpython.org/> ein ausgezeichneter Ausgangspunkt. Das offizielle Tutorial auf *python.org* (<https://docs.python.org/3/tutorial/>) ist ebenfalls recht gut.

Falls Sie noch nie Jupyter verwendet haben, führt Sie Kapitel 2 durch die Installation und die Grundlagen: Es ist ein prima Werkzeug in Ihrem Werkzeugkasten.

Wenn Sie mit den wissenschaftlichen Bibliotheken für Python nicht vertraut sind, beinhalten die mitgelieferten Jupyter Notebooks einige Tutorials. Es gibt dort auch ein kurzes Mathematiktutorial über lineare Algebra.

Wegweiser durch dieses Buch

Dieses Buch ist in zwei Teile aufgeteilt. Teil I behandelt folgende Themen:

- Was ist Machine Learning? Welche Aufgaben lassen sich damit lösen? Welches sind die wichtigsten Kategorien und Grundbegriffe von Machine-Learning-Systemen?
- Die wichtigsten Schritte in einem typischen Machine-Learning-Projekt.
- Lernen durch Anpassen eines Modells an Daten.
- Optimieren einer Kostenfunktion.
- Bearbeiten, Säubern und Vorbereiten von Daten.
- Merkmale auswählen und entwickeln.
- Ein Modell auswählen und dessen Hyperparameter über Kreuzvalidierung optimieren.
- Die wichtigsten Herausforderungen beim Machine Learning, insbesondere Underfitting und Overfitting (das Gleichgewicht zwischen Bias und Varianz).

- Dimensionsreduktion der Trainingsdaten, um dem Fluch der Dimensionalität etwas entgegenzusetzen.
- Die verbreitetsten Lernalgorithmen: lineare und polynomiale Regression, logistische Regression, k-nächste Nachbarn, Support Vector Machines, Entscheidungsbäume, Random Forests und Ensemble-Methoden.

Teil II behandelt folgende Themen:

- Was sind neuronale Netze? Wofür sind sie geeignet?
- Erstellen und Trainieren neuronaler Netze mit TensorFlow.
- Die wichtigsten Architekturen neuronaler Netze: Feed-Forward-Netze, Convolutional Neural Networks, rekurrente Netze, Long-Short-Term-Memory-(LSTM-)Netze und Autoencoder.
- Techniken zum Trainieren von Deep-Learning-Netzen.
- Skalieren neuronaler Netze bei riesigen Datensätzen.
- Reinforcement Learning.

Der erste Teil baut vor allem auf Scikit-Learn auf, der zweite Teil verwendet TensorFlow.



Springen Sie nicht zu schnell ins tiefen Wasser: Auch wenn Deep Learning zweifelsohne eines der aufregendsten Teilgebiete im Machine Learning ist, sollten Sie zuerst Erfahrungen mit den Grundlagen sammeln. Außerdem lassen sich die meisten Aufgabenstellungen recht gut mit einfacheren Techniken wie Random Forests und Ensemble-Methoden lösen (die in Teil I besprochen werden). Deep Learning ist am besten für komplexe Aufgaben wie Bilderkennung, Spracherkennung und Sprachverarbeitung geeignet, vorausgesetzt, Sie haben genug Daten und Geduld.

Ressourcen im Netz

Es gibt viele Ressourcen, mithilfe derer sich Machine Learning erlernen lässt. Der ML-Kurs auf Coursera (<https://www.coursera.org/learn/machine-learning/>) von Andrew Ng und der Kurs über neuronale Netze und Deep Learning (<https://www.coursera.org/course/neuralnets>) von Geoffrey Hinton sind faszinierend, auch wenn beide einen beträchtlichen Zeitaufwand bedeuten (in Monaten).

Es gibt auch viele interessante Webseiten über Machine Learning, darunter natürlich den ausgezeichneten User Guide (http://scikit-learn.org/stable/user_guide.html) von Scikit-Learn. Auch Dataquest (<https://www.dataquest.io/>), das sehr ansprechende Tutorials und ML-Blogs bietet, sowie die auf Quora (<http://goo.gl/GwtU3A>) aufgeführten ML-Blogs könnten Ihnen gefallen. Schließlich sind auf der Deep-Learning-Webseite (<http://deeplearning.net/>) Ressourcen aufgezählt, mit denen Sie mehr lernen können.

Natürlich bieten auch viele andere Bücher eine Einführung in Machine Learning, insbesondere:

- Joel Grus, *Einführung in Data Science: Grundprinzipien der Datenanalyse mit Python* (O'Reilly). Dieses Buch stellt die Grundlagen von Machine Learning vor und implementiert die wichtigsten Algorithmen in reinem Python (von null auf).
- Stephen Marsland, *Machine Learning: An Algorithmic Perspective* (Chapman and Hall). Dieses Buch ist eine großartige Einführung in Machine Learning, die viele Themen ausführlich behandelt. Es enthält Codebeispiele in Python (ebenfalls von null auf, aber mit NumPy).
- Sebastian Raschka, *Machine Learning mit Python: Das Praxis-Handbuch für Data Science, Predictive Analytics und Deep Learning* (mitp Professional). Eine weitere ausgezeichnete Einführung in Machine Learning. Dieses Buch konzentriert sich auf Open-Source-Bibliotheken in Python (Pylearn 2 und Theano).
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin, *Learning from Data* (MLBook). Als eher theoretische Abhandlung von ML enthält dieses Buch sehr tief gehende Erkenntnisse, insbesondere zum Gleichgewicht zwischen Bias und Varianz (siehe Kapitel 4).
- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach, 3rd Edition* (Pearson). Dieses ausgezeichnete (und umfangreiche) Buch deckt eine unglaubliche Stoffmenge ab, darunter Machine Learning. Es hilft dabei, ML in einem breiteren Kontext zu betrachten.

Eine gute Möglichkeit zum Lernen sind schließlich Webseiten mit ML-Wettbewerben wie *Kaggle.com* (<https://www.kaggle.com/>). Dort können Sie Ihre Fähigkeiten an echten Aufgaben üben und Hilfe und Tipps von einigen der besten ML-Profis erhalten.

In diesem Buch verwendete Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

Kursiv

Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

Konstante Zeichenbreite

Wird für Programm listings und für Programmelemente in Textabschnitten wie Namen von Variablen und Funktionen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter verwendet.

Konstante Zeichenbreite, fett

Kennzeichnet Befehle oder anderen Text, den der Nutzer wörtlich eingeben sollte.

Konstante Zeichenbreite, kursiv

Kennzeichnet Text, den der Nutzer je nach Kontext durch entsprechende Werte ersetzen sollte.



Dieses Symbol steht für einen Tipp oder eine Empfehlung.



Dieses Symbol steht für einen allgemeinen Hinweis.



Dieses Symbol steht für eine Warnung oder erhöhte Aufmerksamkeit.

Verwenden von Codebeispielen

Zusätzliche Materialien (Codebeispiele, Übungen und so weiter) können Sie von der Adresse <https://github.com/ageron/hands-on-ml> herunterladen.

Dieses Buch ist dazu da, Ihnen beim Erledigen Ihrer Arbeit zu helfen. Im Allgemeinen dürfen Sie die Codebeispiele aus diesem Buch in Ihren eigenen Programmen und der dazugehörigen Dokumentation verwenden. Sie müssen uns dazu nicht um Erlaubnis fragen, solange Sie nicht einen beträchtlichen Teil des Codes reproduzieren. Beispielsweise benötigen Sie keine Erlaubnis, um ein Programm zu schreiben, in dem mehrere Codefragmente aus diesem Buch vorkommen. Wollen Sie dagegen eine CD-ROM mit Beispielen aus Büchern von O'Reilly verkaufen oder verteilen, benötigen Sie eine Erlaubnis. Eine Frage zu beantworten, indem Sie aus diesem Buch zitieren und ein Codebeispiel wiedergeben, benötigt keine Erlaubnis. Eine beträchtliche Menge Beispielcode aus diesem Buch in die Dokumentation Ihres Produkts aufzunehmen, bedarf hingegen einer Erlaubnis.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN. Beispiel: »*Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow* von Aurélien Géron (O'Reilly). Copyright 2017 Aurélien Géron, 978-3-96009-061-8.«

Wenn Sie glauben, dass Ihre Verwendung von Codebeispielen über die übliche Nutzung hinausgeht oder außerhalb der oben vorgestellten Nutzungsbedingungen liegt, kontaktieren Sie uns bitte unter kommentar@oreilly.de.

Danksagungen

Ich möchte meinen Kollegen bei Google danken, insbesondere dem Team zur Klassifikation von YouTube-Videos, von denen ich sehr viel über Machine Learning gelernt habe. Ohne sie hätte ich dieses Projekt niemals starten können. Besonderer Dank gebührt meinen persönlichen ML-Gurus: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington und jedem bei YouTube Paris.

Ich bin all den fantastischen Leuten unglaublich dankbar, die in ihrem geschäftigen Leben die Zeit gefunden haben, mein Buch im Detail gegenzulesen. Ich danke Pete Warden für das Beantworten meiner Fragen zu TensorFlow, das Begutachten von Teil II, viele hilfreiche Hinweise und natürlich für seine Teilnahme im TensorFlow-Kernteam. Sie sollten sich auf jeden Fall sein Blog (<https://petewarden.com/>) anschauen! Ich danke auch Lukas Biewald für dessen besonders gründliche Prüfung von Teil II: Er ließ keinen Stein auf dem anderen, probierte alle Codebeispiele aus (und fand einige Fehler) und gab viele ausgezeichnete Hinweise. Sein Enthusiasmus war ansteckend. Sie sollten sich sein Blog (<https://lukasbiewald.com/>) und seine coolen Roboter (<https://goo.gl/Eu5u28>) anschauen! Ich danke auch Justin Francis, der Teil II ebenfalls sehr gründlich geprüft, Fehler gefunden und hilfreiche Hinweise gegeben hat, insbesondere zu Kapitel 16. Lesen Sie seine Blogbeiträge (<https://goo.gl/28ve8z>) zu TensorFlow!

Großer Dank gebührt auch David Andrzejewski, der Teil I begutachtet und außerordentlich nützliches Feedback dazu gegeben hat. Er fand unklare Textabschnitte und gab Verbesserungsvorschläge. Schauen Sie sich seine Webseite (<http://www.david-andrzejewski.com/>) an! Ich danke Grégoire Mesnil, der Teil II begutachtet und sehr interessante praktische Tipps zum Trainieren neuronaler Netze beigesteuert hat. Ich danke außerdem Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn, Iain Smears und Vincent Guilbeau für das Gegenlesen von Teil I und die vielen nützlichen Vorschläge. Ich möchte meinem Schwiegervater, Michel Tessier, einem ehemaligen Mathematiklehrer und neuerdings dem Übersetzer von Anton Chekhov, für die Hilfe beim Glattbügeln der Mathematik und der Notationen in diesem Buch sowie dem Gegenlesen des Jupyter Notebooks zu linearer Algebra danken.

Natürlich geht ein riesiges Dankeschön an meinen lieben Bruder Sylvain, der sich jedes einzelne Kapitel durchgelesen, jede Codezeile getestet und zu praktisch jedem Abschnitt Feedback geliefert und mich von der ersten bis zur letzten Zeile ermutigt hat. Ich liebe Dich, Bro!

Ich danke auch den fantastischen Leuten bei O'Reilly, insbesondere Nicole Tache für ihr aufschlussreiches, immer freundliches, ermutigendes und hilfreiches Feedback. Ich danke Marie Beaugureau, Ben Lorica, Mike Loukides und Laurel Ruma dafür, dass sie an dieses Projekt geglaubt und mir geholfen haben, den Rahmen

abzustecken. Ich danke Matt Hacker und dem gesamten Atlas-Team für das Beantworten aller meiner technischer Fragen zu Formatierung, AsciiDoc und LaTeX sowie Rachel Monaghan, Nick Adams und dem gesamten Produktionsteam für deren finale Begutachtung und Hunderte Korrekturen.

Schließlich bin ich meiner geliebten Frau Emmanuelle und unseren drei wunderbaren Kindern Alexandre, Rémi und Gabrielle unendlich dafür dankbar, dass sie mich zur Arbeit an diesem Buch ermutigt, viele Fragen gestellt (wer hat gesagt, man könne Siebenjährigen keine neuronalen Netze beibringen?) und mir sogar Kekse und Kaffee vorbeigebracht haben. Was kann man sich noch wünschen?

TEIL I

Die Grundlagen des Machine Learning

KAPITEL 1

Die Machine Learning-Umgebung

Die meisten Menschen denken beim Begriff »Machine Learning« an einen Roboter: einen zuverlässigen Butler oder einen tödlichen Terminator, je nachdem wen Sie fragen. Aber Machine Learning ist keine futuristische Fantasie, es ist bereits Gegenwart. Tatsächlich gibt es Machine Learning in bestimmten, spezialisierten Anwendungsbereichen wie der *optischen Zeichenerkennung* (OCR) schon seit Jahrzehnten. Aber die erste weitverbreitete Anwendung von ML, die das Leben von Hunderten Millionen Menschen verbesserte, hat die Welt in den 1990er-Jahren erobert: Es war der *Spamfilter*. Es ist nicht gerade ein Skynet mit eigenem Bewusstsein, aber technisch gesehen ist es Machine Learning (es hat inzwischen so gut gelernt, dass Sie nur noch selten eine E-Mail als Spam kennzeichnen müssen). Dem Spamfilter folgten Hunderte weiterer Anwendungen von ML, die still und heimlich Hunderte Produkte und Funktionen aus dem Alltag steuern, darunter Einkaufsempfehlungen und Stimmsuche.

Wo aber beginnt Machine Learning und wo hört es auf? Worum genau geht es, wenn eine Maschine etwas *lernt*? Wenn ich mir eine Kopie von Wikipedia herunterlade, hat mein Computer dann schon etwas »gelernt«? Ist er auf einmal schlauer geworden? In diesem Kapitel werden wir erst einmal klarstellen, was Machine Learning ist und wofür Sie es einsetzen könnten.

Bevor wir aber beginnen, den Kontinent des Machine Learnings zu erforschen, werfen wir einen Blick auf die Landkarte und lernen die wichtigsten Regionen und Orientierungspunkte kennen: überwachtes und unüberwachtes Lernen, Online- und Batch-Lernen, instanzbasiertes und modellbasiertes Lernen. Anschließend betrachten wir die Arbeitsabläufe in einem typischen ML-Projekt, diskutieren die dabei wichtigsten Herausforderungen und besprechen, wie Sie ein Machine-Learning-System auswerten und optimieren können.

In diesem Kapitel werden diverse Grundbegriffe (und Fachjargon) eingeführt, die jeder Data Scientist auswendig kennen sollte. Es wird ein abstrakter und recht einfacher Überblick bleiben (das einzige Kapitel mit wenig Code), aber Ihnen sollte alles glasklar sein, bevor Sie mit dem Buch fortfahren. Schnappen Sie sich also einen Kaffee und los geht's!



Wenn Sie bereits sämtliche Grundlagen von Machine Learning kennen, können Sie direkt mit Kapitel 2 fortfahren. Falls Sie sich nicht sicher sind, versuchen Sie die Fragen am Ende des Kapitels zu beantworten, bevor Sie fortfahren.

Was ist Machine Learning?

Machine Learning ist die Wissenschaft (und Kunst), Computer so zu programmieren, dass sie *anhand von Daten lernen*.

Hier ist eine etwas allgemeinere Definition:

[Maschinelles Lernen ist das] Fachgebiet, das Computern die Fähigkeit zu Lernen verleiht, ohne explizit programmiert zu werden.

Arthur Samuel 1959

Und eine eher technisch orientierte:

Man sagt, dass ein Computerprogramm dann aus Erfahrungen E im Bezug auf eine Aufgabe T und ein Maß für die Leistung P lernt, wenn seine durch P gemessene Leistung bei T mit der Erfahrung E anwächst.

Tom Mitchell 1997

Beispielsweise lernt Ihr Spamfilter, ein maschinelles Lernprogramm, aus Beispielen für Spam-E-Mails (z. B. vom Nutzer markierten) und gewöhnlichen E-Mails (Nicht-Spam, auch »ham« genannt), Spam zu erkennen. Diese vom System verwendeten Lernbeispiele nennt man den *Trainingsdatensatz*. Jedes Trainingsbeispiel nennt man einen *Trainingsdatenpunkt* (oder *Instanz*). In diesem Fall besteht die Aufgabe T darin, neue E-Mails als Spam zu kennzeichnen, die Erfahrung E entspricht den *Trainingsdaten*. Nur das Leistungsmaß P ist noch zu definieren; Sie könnten z. B. den Anteil korrekt klassifizierter E-Mails verwenden. Dieses Leistungsmaß nennt man *Genauigkeit*. Es wird bei Klassifikationsaufgaben häufig verwendet.

Falls Sie gerade eine Kopie von Wikipedia heruntergeladen haben, verfügt Ihr Computer über eine Menge zusätzlicher Daten, verbessert sich dadurch aber bei keiner Aufgabe. Deshalb ist dies kein Machine Learning.

Warum wird Machine Learning verwendet?

Überlegen Sie einmal, wie Sie mit herkömmlichen Programmietechniken einen Spamfilter schreiben würden (Abbildung 1-1):

1. Zuerst würden Sie sich ansehen, wie Spam typischerweise aussieht. Sie würden feststellen, dass einige Wörter oder Phrasen (wie »Für Sie«, »Kreditkarte«, »kostenlos«, und »erstaunlich«) in der Betreffzeile gehäuft auftreten. Mögli-

cherweise würden Ihnen auch weitere Muster im Namen des Absenders, dem Text der E-Mail und so weiter auffallen.

2. Sie würden für jedes der von Ihnen erkannten Muster einen Algorithmus schreiben, der dieses erkennt, und Ihr Programm würde E-Mails als Spam markieren, sobald eine bestimmte Anzahl dieser Muster erkannt wird.
3. Sie würden Ihr Programm testen und die Schritte 1 und 2 wiederholen, bis es gut genug ist.

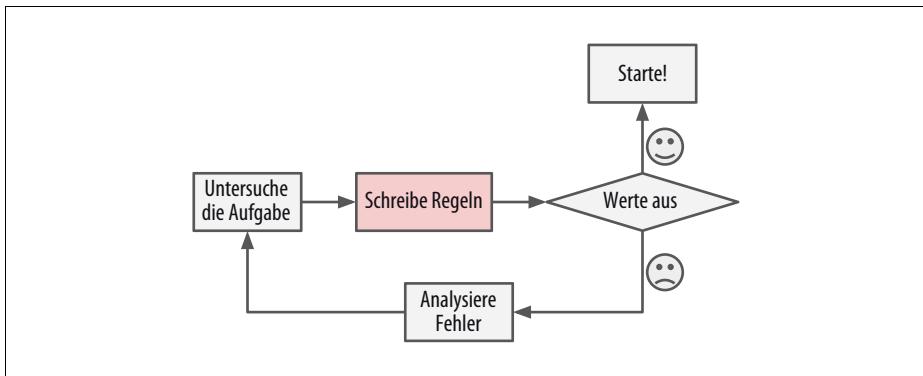


Abbildung 1-1: Die herkömmliche Herangehensweise

Da diese Aufgabe nicht trivial ist, wird Ihr Programm vermutlich eine lange Liste komplexer Regeln beinhalten – und ganz schön schwierig zu warten sein.

Dagegen lernt ein mit Machine-Learning-Techniken entwickelter Spamfilter automatisch, welche Wörter und Phrasen Spam gut vorhersagen, indem er im Vergleich zu den Ham-Beispielen ungewöhnlich häufige Wortmuster in den Spam-Beispielen erkennt (Abbildung 1-2). Das Programm wird viel kürzer, leichter zu warten und wahrscheinlich auch treffsicherer.

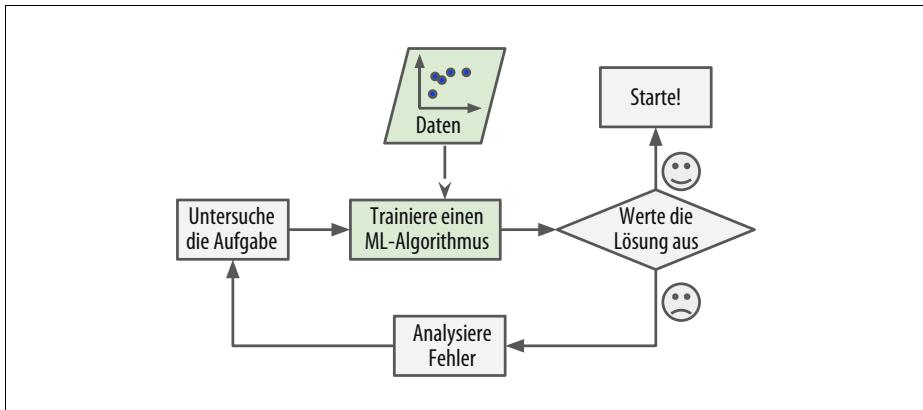


Abbildung 1-2: Der Machine-Learning-Ansatz

Wenn außerdem die Spammer bemerken, dass alle ihre E-Mails mit »für Sie« geblockt werden, könnten sie stattdessen auf »4U« umsatteln. Ein mit traditionellen Programmiertechniken entwickelter Spamfilter müsste aktualisiert werden, um die E-Mails mit »4U« zu markieren. Wenn die Spammer sich ständig um Ihren Spamfilter herumarbeiten, werden Sie ewig neue Regeln schreiben müssen.

Ein auf Machine Learning basierender Spamfilter bemerkt dagegen automatisch, dass »4U« auffällig häufig in von Nutzern als Spam markierten Nachrichten vorkommt und beginnt, diese ohne weitere Intervention auszusortieren (Abbildung 1-3).

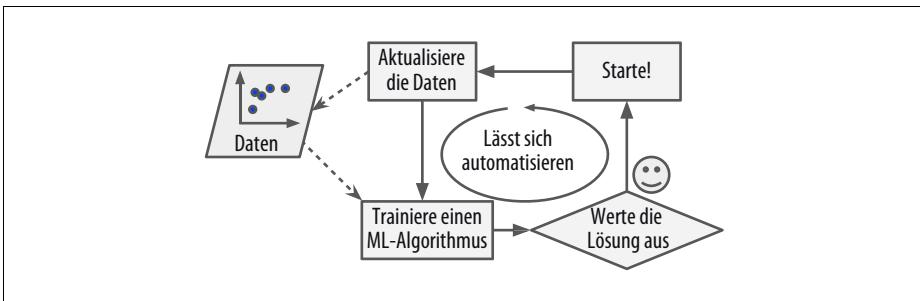


Abbildung 1-3: Automatisches Einstellen auf Änderungen

Machine Learning brilliert außerdem auch bei Aufgaben, die entweder zu komplex für herkömmliche Verfahren sind oder für die kein bekannter Algorithmus existiert. Betrachten Sie z.B. Spracherkennung: Sagen wir, Sie beginnen mit einer einfachen Aufgabe und schreiben ein Programm, dass die Wörter »one« und »two« unterscheiden kann. Sie bemerken, dass das Wort »two« mit einem hochfrequenten Ton (»T«) beginnt, und könnten demnach einen Algorithmus hart codieren, der die Intensität hochfrequenter Töne misst und dadurch »one« und »two« unterscheiden kann. Natürlich skaliert diese Technik nicht auf Tausende Wörter, die von Millionen von Menschen mit Hintergrundgeräuschen und in Dutzenden Sprachen gesprochen werden. Die (zumindest heutzutage) beste Möglichkeit ist, einen Algorithmus zu schreiben, der eigenständig aus vielen aufgenommenen Beispielen für jedes Wort lernt.

Schließlich kann Machine Learning auch Menschen beim Lernen unterstützten (Abbildung 1-4): ML-Algorithmen lassen sich untersuchen, um zu erkennen, was sie gelernt haben (auch wenn dies bei manchen Algorithmen kompliziert sein kann). Wenn z.B. der Spamfilter erst einmal mit genug Spam trainiert wurde, lässt sich die Liste von Wörtern und Wortkombinationen inspizieren, die als gute Merkmale von Spam erkannt wurden. Manchmal kommen dabei überraschende Korrelationen oder neue Trends ans Tageslicht und führen dadurch zu einem besseren Verständnis der Aufgabe.

Das Anwenden von ML-Techniken zum Durchwühlen großer Datenmengen hilft dabei, nicht unmittelbar ersichtliche Muster zu finden. Dies nennt man auch *Data Mining*.

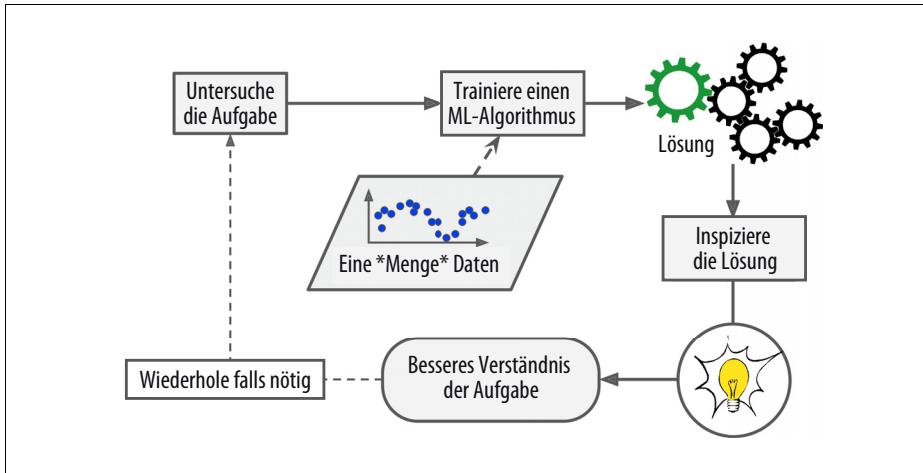


Abbildung 1-4: Machine Learning hilft Menschen beim Lernen

Zusammengefasst ist Machine Learning hervorragend geeignet für:

- Aufgaben, bei denen die existierenden Lösungen eine Menge Handarbeit oder lange Listen von Regeln erfordern: Ein maschinelles Lernalgorithmus vereinfacht oft den Code und schneidet besser ab.
- Komplexe Aufgaben, für die es mit herkömmlichen Methoden überhaupt keine gute Lösung gibt: Die besten Machine-Learning-Techniken können eine Lösung finden.
- Fluktuierende Umgebungen: Ein Machine-Learning-System kann sich neuen Daten anpassen.
- Erkenntnisse über komplexe Aufgabenstellungen und große Datenmengen zu gewinnen.

Arten von Machine-Learning-Systemen

Es gibt so viele unterschiedliche Arten von Machine-Learning-Systemen, dass es hilfreich ist, die Verfahren nach folgenden Kriterien in grobe Kategorien einzuteilen:

- Ob sie mit menschlicher Überwachung trainiert werden oder nicht (überwachtes, unüberwachtes und halbüberwachtes Lernen sowie Reinforcement Learning)
- Ob sie inkrementell dazulernen können oder nicht (Online-Lernen gegenüber Batch-Learning)

- Ob sie einfach neue Datenpunkte mit den bereits bekannten Datenpunkten vergleichen oder stattdessen Muster in den Trainingsdaten erkennen, um ein Vorhersagemodell zu aufzubauen, wie es auch Wissenschaftler tun (instanzbasiertes gegenüber modellbasiertem Lernen)

Diese Kriterien schließen sich nicht gegenseitig aus; sie lassen sich beliebig miteinander kombinieren. Zum Beispiel kann ein moderner Spamfilter ständig mit einem neuronalen Netzwerkmodell mit Beispielen für Spam und Ham dazulernen; damit ist er ein modellbasiertes, überwachtes Online-Lernsystem.

Betrachten wir jedes dieser Kriterien etwas genauer.

Überwachtes/unüberwachtes Lernen

Machine-Learning-Systeme lassen sich entsprechend der Menge und Art der Überwachung beim Trainieren einordnen. Es gibt dabei vier größere Kategorien: überwachtes Lernen, unüberwachtes Lernen, halbüberwachtes Lernen und verstärkendes Lernen (Reinforcement Learning).

Überwachtes Lernen

Beim *überwachten Lernen* enthalten die dem Algorithmus gelieferten Trainingsdaten die gewünschten Lösungen, genannt *Labels* (Abbildung 1-5).

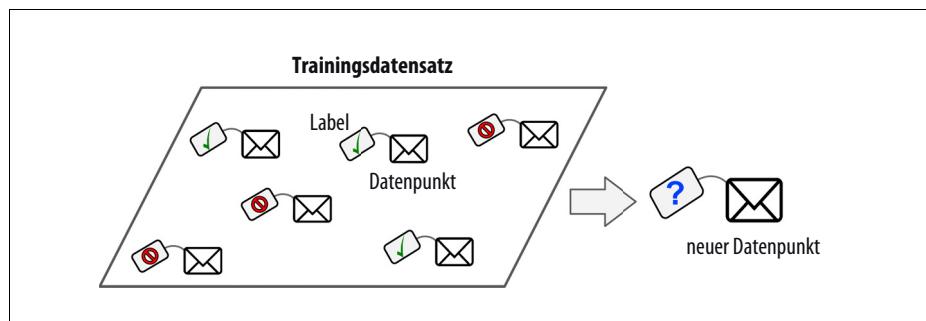


Abbildung 1-5: Ein Trainingsdatensatz mit Labels für überwachtes Lernen (z.B. Spam-Erkennung)

Klassifikation ist eine typische überwachte Lernaufgabe. Spamfilter sind hierfür ein gutes Beispiel: Sie werden mit vielen Beispiel-E-Mails und deren *Kategorie* (Spam oder Ham) trainiert und müssen lernen, neue E-Mails zu klassifizieren.

Eine weitere typische Aufgabe ist, eine numerische *Zielgröße* vorherzusagen, wie etwa den Preis eines Autos auf Grundlage gegebener *Merkmale* (gefahren Kilometer, Alter, Marke und so weiter), den sogenannten *Prädiktoren*. Diese Art Aufgabe bezeichnet man als *Regression* (Abbildung 1-6).¹ Um das System zu trainieren, benötigt es viele Beispelfahrzeuge mitsamt ihren Prädiktoren und Labels (also den Preisen).



Beim Machine Learning ist ein *Attribut* ein Datentyp (z.B. »Kilometerzahl«), wohingegen ein *Merkmal* je nach Kontext mehrere Bedeutungen haben kann. Meist ist damit ein Attribut und dessen Wert gemeint (z.B. »Kilometerzahl = 15000«). Allerdings verwenden viele Anwender *Attribut* und *Merkmal* synonym.

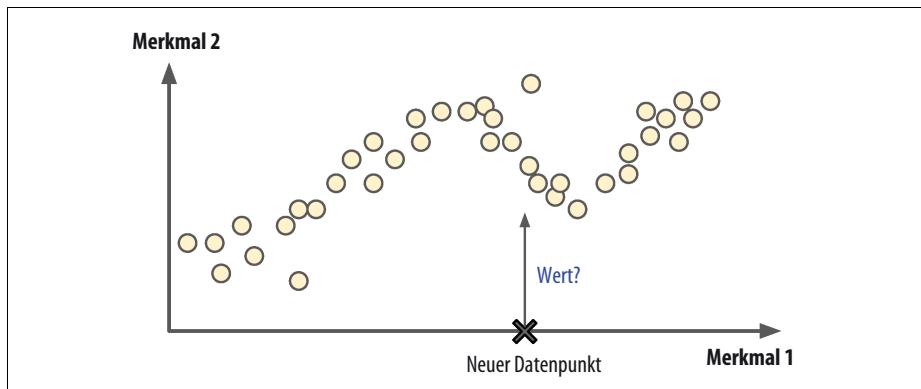


Abbildung 1-6: Regression

Viele Regressionsalgorithmen lassen sich auch zur Klassifikation einsetzen und umgekehrt. Zum Beispiel ist die *logistische Regression* eine verbreitete Methode für Klassifizierungsaufgaben, da sie die Wahrscheinlichkeit der Zugehörigkeit zu einer bestimmten Kategorie als Ergebnis liefert (z.B. 20%ige Chance für Spam).

Hier sind einige der wichtigsten (in diesem Buch besprochenen) überwachten Lernalgorithmen:

- k-nächste-Nachbarn
- lineare Regression
- logistische Regression
- Support Vector Machines (SVMs)
- Entscheidungsbäume und Random Forests
- neuronale Netzwerke²

1 Wissenswert: Dieser seltsam anmutende Begriff wurde von Francis Galton in die Statistik eingeführt, der beobachtete, dass die Kinder hochgewachsener Eltern zu einer geringeren Körpergröße als ihre Eltern neigen. Da die Kinder kleiner waren, nannte er dies *Regression zum Mittelwert*. Dieser Name wurde anschließend auch für seine Methode zur Analyse von Korrelationen zwischen Variablen verwendet.

2 Einige neuronale Netzwerkarchitekturen können unüberwacht sein, wie beispielsweise Autoencoder und Restricted Boltzmann Machines. Sie können auch halbüberwacht sein, wie bei Deep Belief Networks und unüberwachtem Vtrainieren.

Unüberwachtes Lernen

Beim *unüberwachten Lernen* sind die Trainingsdaten, wie der Name vermuten lässt, nicht gelabelt (Abbildung 1-7). Das System versucht, ohne Anleitung zu lernen.

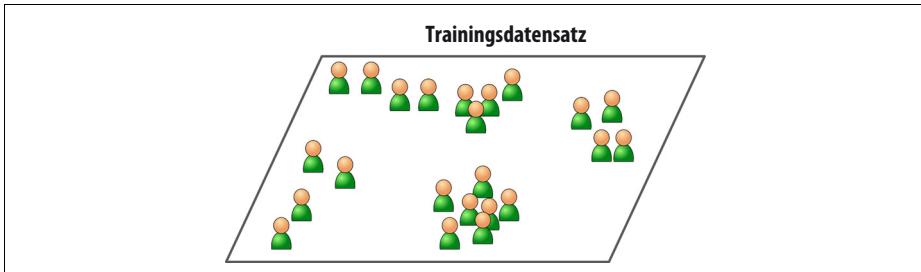


Abbildung 1-7: Ein Trainingsdatensatz ohne Labels für unüberwachtes Lernen

Hier sind einige der wichtigsten unüberwachten Lernalgorithmen (wir werden die Dimensionsreduktion in Kapitel 8 behandeln):

- Clustering
 - k-Means
 - hierarchische Clusteranalyse (HCA)
 - Expectation Maximization
- Visualisierung und Dimensionsreduktion
 - Hauptkomponentenzerlegung (PCA)
 - Kernel PCA
 - Locally-Linear Embedding (LLE)
 - t-verteiltes Stochastic Neighbor Embedding (t-SNE)
- Lernen mit Assoziationsregeln
 - Apriori
 - Eclat

Nehmen wir an, Sie hätten eine Menge Daten über Besucher Ihres Blogs. Sie möchten einen *Clustering-Algorithmus* verwenden, um Gruppen ähnlicher Besucher zu entdecken (Abbildung 1-8). Sie verraten dem Algorithmus nichts darüber, welcher Gruppe ein Besucher angehört: Er findet die Verbindungen ohne Ihr Zutun heraus. Beispielsweise könnte der Algorithmus bemerken, dass 40% Ihrer Besucher Männer mit einer Vorliebe für Comics sind, die Ihr Blog abends lesen, 20% dagegen sind junge Science-Fiction-Fans, die am Wochenende vorbeischauen und so weiter. Wenn Sie ein *hierarchisches Clusterverfahren* verwenden, können Sie sogar jede Gruppe in weitere Untergruppen zerlegen, was hilfreich sein kann, wenn Sie Ihre Blogartikel auf diese Zielgruppen zuschneiden möchten.

Algorithmen zur *Visualisierung* sind ebenfalls ein gutes Beispiel für unüberwachtes Lernen: Sie übergeben diesen eine Menge komplexer Daten ohne Labels und erhalten

ten eine 2-D- oder 3-D-Repräsentation der Daten, die Sie leicht grafisch darstellen können (Abbildung 1-9). Solche Algorithmen versuchen, die Struktur der Daten so gut wie möglich zu erhalten (z.B. Cluster in den Eingabedaten am Überlappen in der Visualisierung zu hindern), sodass Sie leichter verstehen können, wie die Daten aufgebaut sind, und womöglich auf unvermutete Muster stoßen.

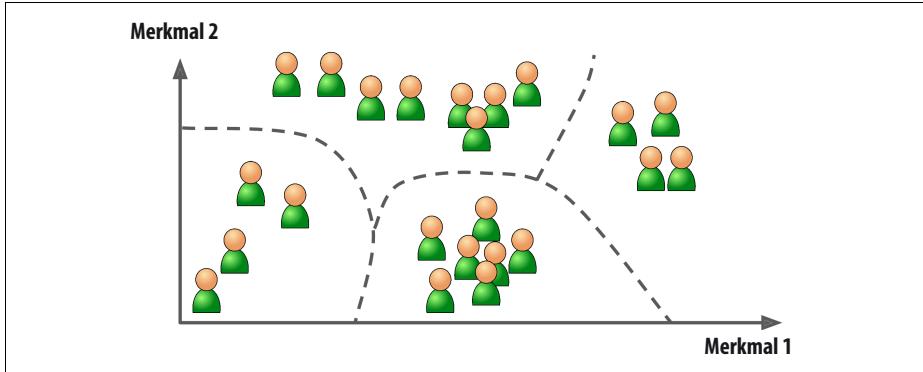


Abbildung 1-8: Clustering

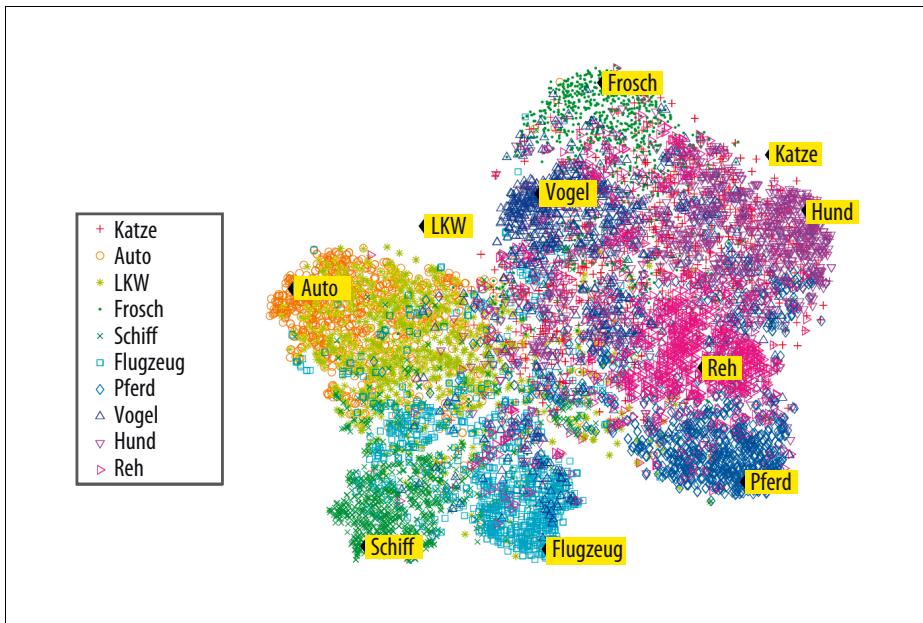


Abbildung 1-9: Beispiel für eine t-SNE-Visualisierung semantischer Cluster³

3 Beachten Sie, dass die Tiere recht gut von Fahrzeugen unterschieden werden, wie nahe Pferde und Rehe einander sind, aber weit entfernt von Vögeln und so weiter. Abbildung mit Erlaubnis reproduziert, Quelle: Socher, Ganjoo, Manning, and Ng (2013), »T-SNE visualization of the semantic word space.«

Eine verwandte Aufgabe ist die *Dimensionsreduktion*, bei der das Ziel die Vereinfachung der Daten ist, ohne dabei allzu viel Information zu verlieren. Dazu lassen sich mehrere korrelierende Merkmale zu einem vereinigen. Beispielsweise korreliert der Kilometerstand eines Autos mit seinem Alter, daher kann ein Algorithmus zur Dimensionsreduktion beide zu einem Merkmal verbinden, das die Abnutzung des Fahrzeugs repräsentiert. Dies nennt man auch *Extraktion von Merkmalen*.



Meist ist es eine gute Idee, die Dimensionen Ihrer Trainingsdaten zu reduzieren, bevor Sie sie in einen anderen Machine-Learning-Algorithmus einspeisen (wie etwa einen überwachten Lernalgorithmus). Er wird viel schneller arbeiten, und die Daten beanspruchen weniger Platz auf der Festplatte und im Speicher. In manchen Fällen ist auch das Ergebnis besser.

Eine weitere wichtige unüberwachte Aufgabe ist das *Erkennen von Anomalien* – beispielsweise ungewöhnliche Transaktionen auf Kreditkarten, die auf Betrug hindeuten, das Abfangen von Produktionsfehlern oder das automatische Entfernen von Ausreißern aus einem Datensatz, bevor dieser in einen weiteren Lernalgorithmus eingespeist wird. Das System wird mit gewöhnlichen Datenpunkten trainiert, und wenn es einen neuen Datenpunkt sieht, kann es entscheiden, ob dieser wie ein normaler Punkt oder eine Anomalie aussieht (siehe Abbildung 1-10).

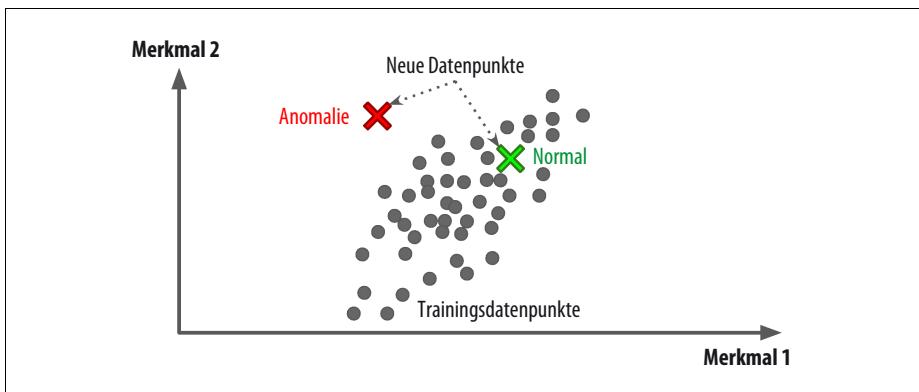


Abbildung 1-10: Erkennen von Anomalien

Schließlich ist auch das *Lernen von Assoziationsregeln* eine verbreitete unüberwachte Lernaufgabe, bei der das Ziel ist, in große Datenmengen einzutauchen und interessante Beziehungen zwischen Merkmalen zu entdecken. Wenn Sie beispielsweise einen Supermarkt führen, könnten Assoziationsregeln auf Ihren Verkaufsdaten ergeben, dass Kunden, die Grillsoße und Kartoffelchips einkaufen, tendenziell auch Steaks kaufen. Daher sollten Sie diese Artikel in unmittelbarer Nähe zueinander platzieren.

Halbüberwachtes Lernen

Einige Algorithmen können mit nur teilweise gelabelten Trainingsdaten arbeiten. Normalerweise sind die Trainingsdaten mehrheitlich ohne Labels. Dies bezeichnet man als *halbüberwachtes Lernen* (Abbildung 1-11).

Einige Fotodienste wie Google Photos bieten hierfür ein gutes Beispiel. Sobald Sie all Ihre Familienfotos in den Dienst hochgeladen haben, erkennt dieser automatisch, dass die gleiche Person A auf den Fotos 1, 5 und 11 vorkommt, während Person B auf den Fotos 2, 5 und 7 zu sehen ist. Dies ist der unüberwachte Teil des Algorithmus (Clustering). Nun muss das System nur noch wissen, wer diese Personen sind. Ein Label pro Person⁴ genügt, um jede Person in jedem Foto zuzuordnen, was bei der Suche nach Fotos nützlich ist.

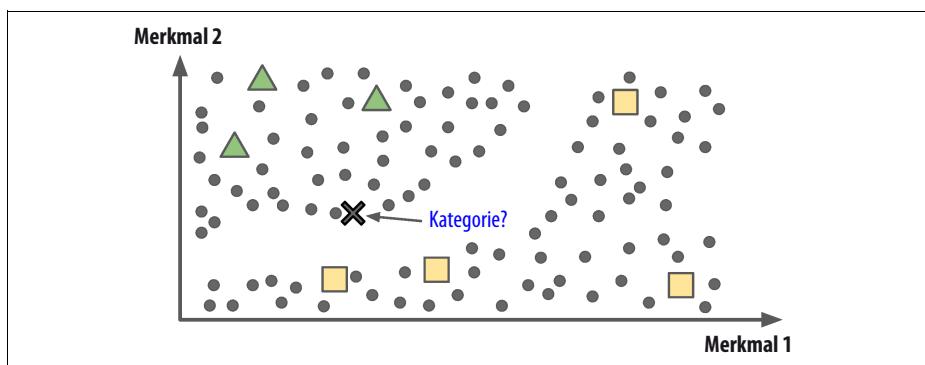


Abbildung 1-11: Halbüberwachtes Lernen

Die meisten Algorithmen für halbüberwachtes Lernen sind Kombinationen von unüberwachten und überwachten Verfahren. Beispielsweise beruhen *Deep Belief Networks* (DBNs) auf in Reihe geschalteten unüberwachten Komponenten namens *restricted Boltzmann Machines* (RBMs). Die RBMs werden nacheinander unüberwacht trainiert. Die Feinabstimmung des Gesamtsystems findet anschließend mit überwachten Lerntechniken statt.

Reinforcement Learning

Reinforcement Learning ist etwas völlig anderes. Das Lernsystem, in diesem Zusammenhang als *Agent* bezeichnet, beobachtet eine Umgebung, wählt Aktionen und führt diese aus. Dafür erhält es *Belohnungen* (oder *Strafen* in Form negativer Belohnungen wie in Abbildung 1-12). Das System muss selbst herausfinden, was die beste

4 Dies ist der Fall, wenn das System perfekt funktioniert. In der Praxis werden meist einige Cluster pro Person erstellt. Manchmal werden zwei ähnliche Personen verwechselt, sodass Sie einige Labels pro Person angeben und außerdem einige Cluster von Hand aufräumen müssen.

Strategie oder *Policy* ist, um mit der Zeit die meisten Belohnungen zu erhalten. Eine *Policy* definiert, welche Aktion der Agent in einer gegebenen Situation auswählt.

Beispielsweise verwenden viele Roboter Reinforcement-Learning-Algorithmen, um laufen zu lernen. Auch das Programm AlphaGo von DeepMind ist ein gutes Beispiel für Reinforcement Learning: Es geriet im Mai 2017 in die Schlagzeilen, als es den Weltmeister Ke Jie im Brettspiel Go schlug. AlphaGo erlernte die zum Sieg führende *Policy*, indem es Millionen von Partien analysierte und anschließend viele Spiele gegen sich selbst spielte. Beachten Sie, dass das Lernen während der Partien gegen den Weltmeister abgeschaltet war; AlphaGo wandte nur die bereits erlernte *Policy* an.

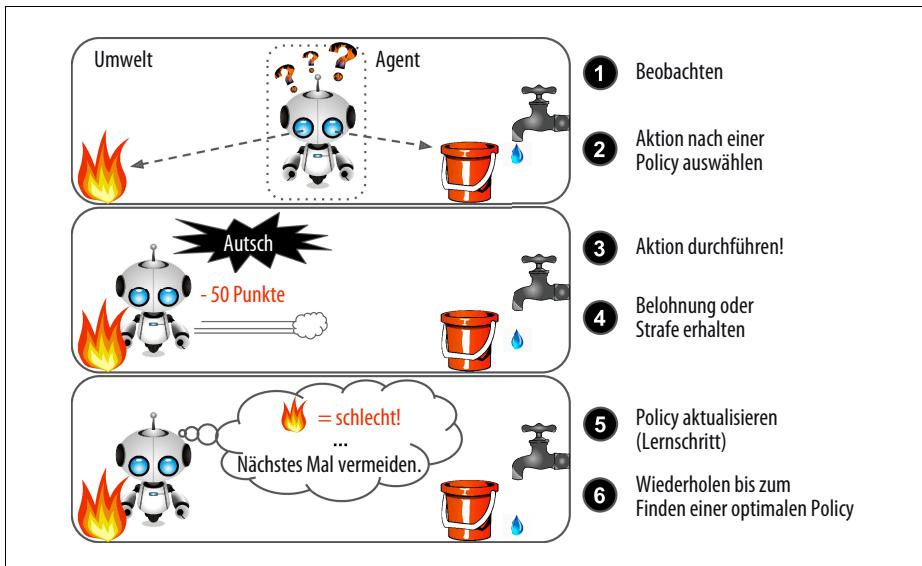


Abbildung 1-12: Reinforcement Learning

Batch- und Online-Learning

Ein weiteres Kriterium zum Einteilen von Machine-Learning-Systemen ist, ob das System aus einem kontinuierlichen Datenstrom inkrementell lernen kann.

Batch-Learning

Beim *Batch-Learning* kann das System nicht inkrementell lernen: Es muss mit sämtlichen verfügbaren Daten trainiert werden. Dies dauert meist lange und beansprucht Rechenkapazitäten. Es wird daher in der Regel offline durchgeführt. Zuerst wird das System trainiert und anschließend in einer Produktivumgebung eingesetzt, wo es ohne weiteres Lernen läuft; es wendet lediglich das bereits Erlernte an. Dies nennt man *Offline-Learning*.

Wenn Sie möchten, dass ein Batch-Learning-System etwas über neue Daten erfährt (beispielsweise eine neue Art Spam), müssen Sie eine neue Version des Systems von Neuem mit dem gesamten Datensatz trainieren (nicht einfach nur den neuen Datensatz, sondern auch den alten). Anschließend müssen Sie das alte System anhalten und durch das neue ersetzen.

Glücklicherweise lässt sich der gesamte Prozess aus Training, Evaluation und Inbetriebnahme eines Machine-Learning-Systems recht leicht automatisieren (wie in Abbildung 1-3 gezeigt). So kann sich selbst ein Batch-Learning-System anpassen. Aktualisieren Sie einfach die Daten und trainieren Sie eine neue Version des Systems so oft wie nötig.

Dies ist eine einfache Lösung und funktioniert meist gut, aber das Trainieren mit dem gesamten Datensatz kann viele Stunden beanspruchen. Daher würde man das neue System nur alle 24 Stunden oder wöchentlich trainieren. Wenn Ihr System sich an schnell ändernde Daten anpassen muss (z.B. um Aktienkurse vorherzusagen), benötigen Sie eine anpassungsfähigere Lösung.

Außerdem beansprucht das Trainieren auf dem gesamten Datensatz eine Menge Rechenkapazität (CPU, Hauptspeicher, Plattenplatz, I/O-Kapazität, Netzwerkbandbreite und so weiter). Wenn Sie eine Menge Daten haben und Ihr System automatisch jeden Tag trainieren lassen, kann Sie das am Ende eine Stange Geld kosten. Falls die Datenmenge sehr groß ist, kann der Einsatz von Batch-Learning sogar unmöglich sein.

Wenn Ihr System autonom lernen muss und die Ressourcen dazu begrenzt sind (z.B. eine Applikation auf einem Smartphone oder ein Fahrzeug auf dem Mars), dann ist das Herumschleppen großer Mengen Trainingsdaten oder das Belegen einer Menge Ressourcen für mehrere Stunden am Tag kein gangbarer Weg.

In all diesen Fällen sind Algorithmen, die inkrementell lernen können, eine bessere Alternative.

Online-Learning

Beim *Online-Learning*, wird das System nach und nach trainiert, indem einzelne Datensätze nacheinander oder in kleinen Paketen, sogenannten *Mini-Batches*, hinzugefügt werden. Jeder Lernschritt ist schnell und billig, sodass das System aus neuen Daten lernen kann, sobald diese verfügbar sind (siehe Abbildung 1-13).

Online-Learning eignet sich großartig für ein System mit kontinuierlich eintreffenden Daten (z.B. Aktienkurse), das sich entweder schnell oder autonom an Veränderungen anpassen muss. Auch wenn Ihnen begrenzte Rechenkapazitäten zur Verfügung stehen, ist es eine sinnvolle Option: Sobald ein Online-Learning-System die neuen Datenpunkte erlernt hat, werden diese nicht mehr benötigt und können verworfen werden (außer Sie möchten in der Lage sein, zu einem früheren Zustand zurückzukehren und den Datenstrom erneut »abzuspielen«). Dies kann enorme Mengen an Speicherplatz einsparen.

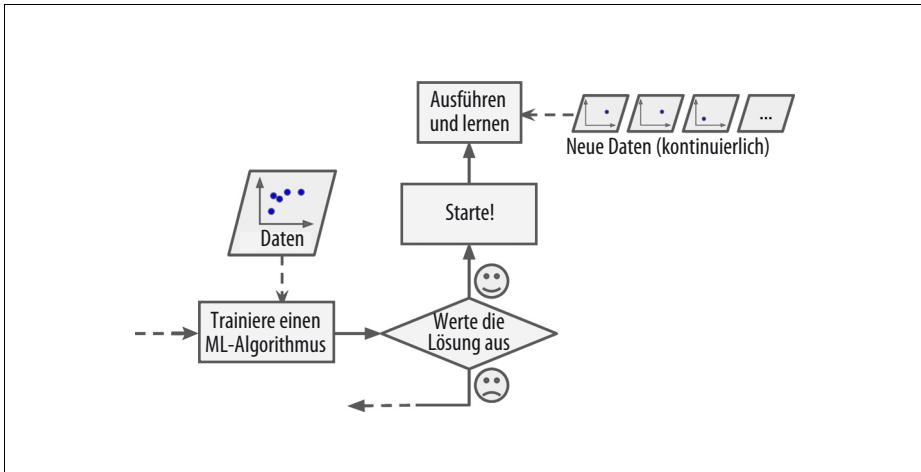


Abbildung 1-13: Online-Learning

Algorithmen zum Online-Learning lassen sich auch zum Trainieren von Systemen mit riesigen Datensätzen einsetzen, die nicht in den Hauptspeicher eines Rechners passen (dies nennt man auch *Out-of-Core-Lernen*). Der Algorithmus lädt einen Teil der Daten, führt einen Trainingsschritt auf den Daten aus und wiederholt den Prozess, bis er sämtliche Daten verarbeitet hat (siehe Abbildung 1-14).



Der gesamte Prozess wird für gewöhnlich offline durchgeführt (also nicht auf einem Produktivsystem), daher ist der Begriff *Online-Learning* etwas irreführend. Stellen Sie sich darunter eher *inkrementelles Lernen* vor.

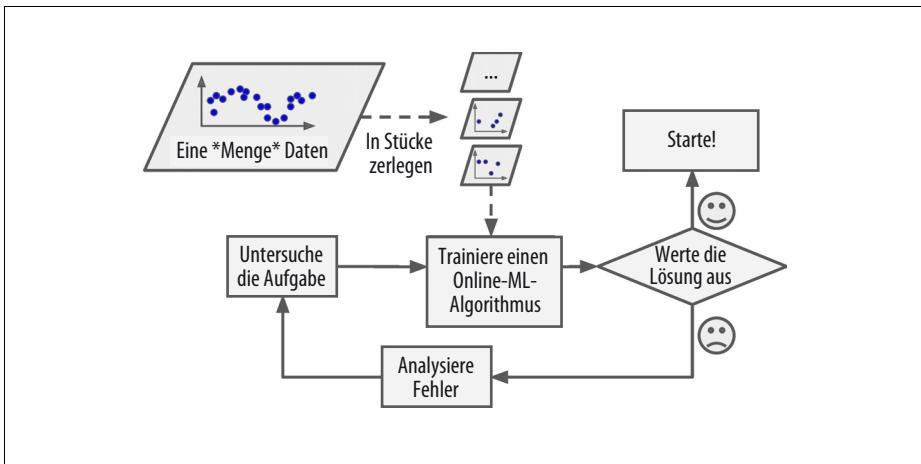


Abbildung 1-14: Verwenden von Online-Learning zum Bewältigen riesiger Datensätze

Ein wichtiger Parameter bei Online-Learning-Systemen ist, wie schnell sie sich an sich verändernde Daten anpassen. Man spricht hier von der *Lernrate*. Wenn Sie die Lernrate hoch ansetzen, wird Ihr System sich schnell auf neue Daten einstellen, aber die alten Daten auch leicht wieder vergessen (Sie möchten sicher nicht, dass ein Spamfilter nur die zuletzt gesehenen Arten von Spam erkennt). Wenn Sie die Lernrate dagegen niedrig ansetzen, entwickelt das System eine höhere Trägheit; das bedeutet, es lernt langsamer, ist aber auch weniger anfällig für Rauschen in den neuen Daten oder für Folgen nicht repräsentativer Datenpunkte.

Eine große Herausforderung beim Online-Learning besteht darin, dass in das System eingespeiste minderwertige Daten zu einer allmählichen Verschlechterung seiner Leistung führen. Wenn es sich dabei um ein Produktivsystem handelt, werden Ihre Kunden dies bemerken. Beispielsweise könnten minderwertige Daten von einem fehlerhaften Sensor an einem Roboter oder jemandem stammen, der sein Ranking in einer Suchmaschine durch massenhafte Anfragen zu verbessern versucht. Um dieses Risiko zu reduzieren, müssen Sie Ihr System aufmerksam beobachten und den Lernprozess beherzt abschalten (und eventuell auf einen früheren Zustand zurücksetzen), sobald Sie einen Leistungsabfall bemerken. Sie können auch die Eingabedaten verfolgen und auf ungewöhnliche Daten reagieren (z.B. über einen Algorithmus zur Erkennung von Anomalien).

Instanzbasiertes versus modellbasiertes Lernen

Eine weitere Sichtachse zum Einteilen maschineller Lernverfahren ist die Art, wie diese *verallgemeinern*. Bei den meisten Aufgaben im Machine Learning geht es um das Treffen von Vorhersagen. Dabei muss ein System in der Lage sein, aus einer Anzahl von Trainingsbeispielen auf nie zuvor gesehene Beispiele zu verallgemeinern. Es ist hilfreich, aber nicht ausreichend, eine gute Leistung auf den Trainingsdaten zu erzielen; das wirkliche Ziel ist, eine gute Leistung auf neuen Datenpunkten zu erreichen.

Es gibt beim Verallgemeinern zwei Ansätze: instanzbasiertes Lernen und modellbasiertes Lernen.

Instanzbasiertes Lernen

Die vermutlich trivialste Art zu lernen, ist das einfache Auswendiglernen. Wenn Sie auf diese Weise einen Spamfilter erstellen würden, würde dieser einfach alle E-Mails aussortieren, die zu bereits von Nutzern markierten E-Mails identisch sind – nicht die schlechteste Lösung, aber sicher nicht die beste.

Anstatt einfach zu bekannten Spam-E-Mails identische Nachrichten zu markieren, könnte Ihr Spamfilter auch so programmiert sein, dass auch zu bekannten Spam-E-Mails sehr ähnliche Nachrichten markiert werden. Dazu ist ein *Ähnlichkeitsmaß* zwischen zwei E-Mails nötig. Ein (sehr einfaches) Maß für die Ähnlichkeit zweier E-Mails könnte die Anzahl gemeinsamer Wörter sein. Das System könnte eine

E-Mail als Spam markieren, wenn diese viele gemeinsame Wörter mit einer bekannten Spammnachricht aufweist.

Dies nennt man *instanzbasiertes Lernen*: Das System lernt die Beispiele auswendig und verallgemeinert dann mithilfe eines Ähnlichkeitsmaßes auf neue Fälle (Abbildung 1-15).

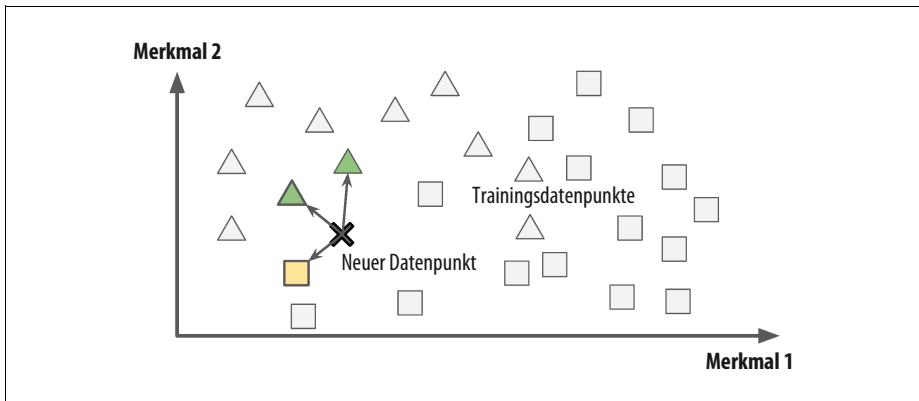


Abbildung 1-15: Instanzbasiertes Lernen

Modellbasiertes Lernen

Eine andere Möglichkeit, von einem Beispieldatensatz zu verallgemeinern, ist, ein Modell aus den Beispielen zu entwickeln und dieses Modell dann für *Vorhersagen* zu verwenden. Dies wird als *modellbasiertes Lernen* bezeichnet (Abbildung 1-16).

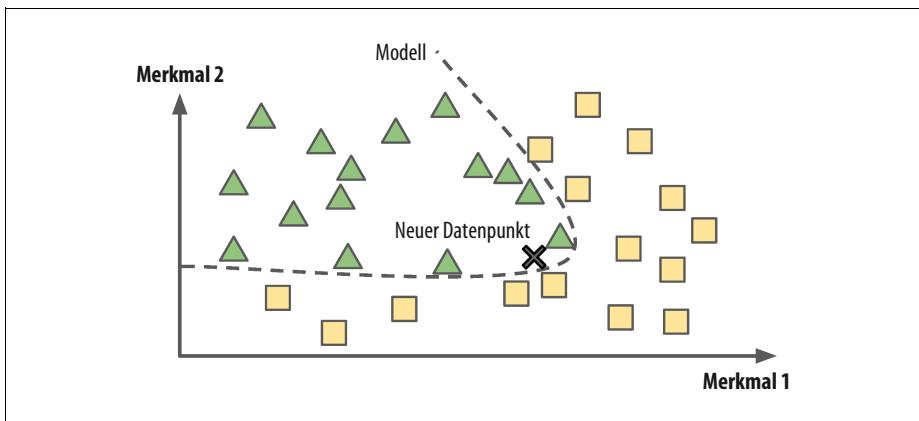


Abbildung 1-16: Modellbasiertes Lernen

Nehmen wir an, Sie möchten herausfinden, ob Geld glücklich macht. Sie laden dazu die Daten des *Better Life Index* von der Webseite des OECD (<https://goo.gl/0Eht9W>) herunter und Statistiken zum Pro-Kopf-Bruttoinlandsprodukt (BIP) von der Webseite des IMF (<http://goo.gl/j1MSKe>). Anschließend führen Sie beide Tabel-

len zusammen und sortieren nach dem BIP pro Kopf. Tabelle 1-1 zeigt einen Ausschnitt des Ergebnisses.

Tabelle 1-1: Macht Geld Menschen glücklicher?

Land	BIP pro Kopf (USD)	Zufriedenheit
Ungarn	12240	4.9
Korea	27195	5.8
Frankreich	37675	6.5
Australien	50962	7.3
Vereinigte Staaten	55805	7.2

Stellen wir diese Daten für einige zufällig ausgewählte Länder dar (Abbildung 1-17).

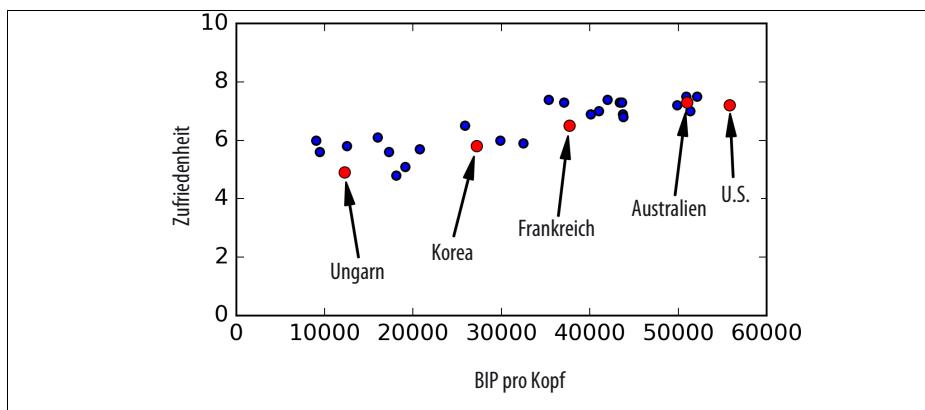


Abbildung 1-17: Sehen Sie hier eine Tendenz?

Es scheint so etwas wie einen Trend zu geben! Auch wenn die Daten *verrauscht* sind (also teilweise zufällig), sieht es so aus, als wenn die Zufriedenheit mehr oder weniger mit dem Pro-Kopf-BIP des Landes linear ansteigt. Sie beschließen also, die Zufriedenheit als lineare Funktion des Pro-Kopf-Bruttoinlandsprodukts zu modellieren. Diesen Schritt bezeichnet man als *Modellauswahl*: Sie wählen ein *lineares Modell* der Zufriedenheit mit genau einem Merkmal aus, nämlich dem Pro-Kopf-BIP (Formel 1-1).

Formel 1-1: Ein einfaches lineares Modell

$$\text{Zufriedenheit} = \theta_0 + \theta_1 \times \text{BIP_pro_Kopf}$$

Diesen Modell enthält zwei *Modellparameter*, θ_0 und θ_1 .⁵ Indem Sie diese Parameter verändern, kann das Modell jede lineare Funktion annehmen, wie Sie in Abbildung 1-18 sehen können.

5 Der griechische Buchstabe θ (Theta) wird konventionsgemäß häufig zum Darstellen von Modellparametern verwendet.

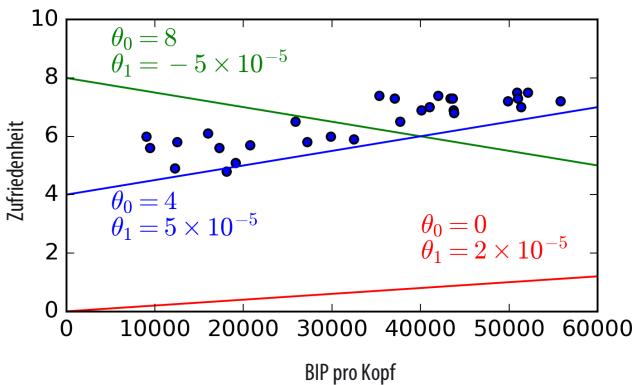


Abbildung 1-18: Einige mögliche lineare Modelle

Bevor Sie Ihr Modell verwenden können, müssen Sie die Werte der Parameter θ_0 und θ_1 festlegen. Woher sollen Sie wissen, welche Werte zur bestmöglichen Leistung führen? Um diese Frage zu beantworten, müssen Sie ein Maß für die Leistung festlegen. Sie können dafür entweder eine *Nutzenfunktion* (oder *Fitnessfunktion*) verwenden, die die *Güte* Ihres Modells bestimmt. Alternativ können Sie eine *Kostenfunktion* definieren, die misst, wie *schlecht* das Modell ist. Bei linearen Modellen verwendet man typischerweise eine Kostenfunktion, die die Entfernung zwischen den Vorhersagen des linearen Modells und den Trainingsbeispielen bestimmt; das Ziel ist, diese Entfernung zu minimieren.

An dieser Stelle kommt der Algorithmus zur linearen Regression ins Spiel: Sie speisen Ihre Trainingsdaten ein, und der Algorithmus ermittelt die für Ihre Daten bestmöglichen Parameter des linearen Modells. Dies bezeichnet man als *Trainieren* des Modells. In unserem Fall ermittelt der Algorithmus $\theta_0 = 4.85$ und $\theta_1 = 4.91 \times 10^{-5}$ als optimale Werte für die beiden Parameter.

Nun passt das Modell (für ein lineares Modell) bestmöglich zu den Trainingsdaten, wie Abbildung 1-19 zeigt.

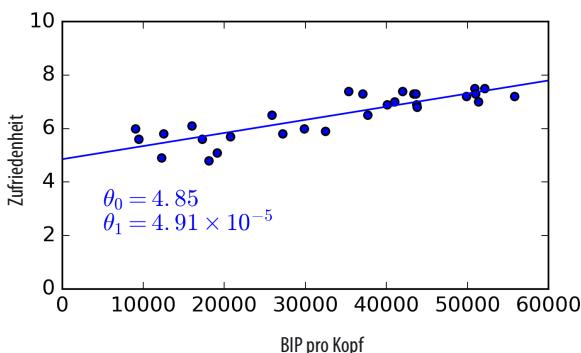


Abbildung 1-19: Das an die Trainingsdaten optimal angepasste lineare Modell

Sie sind nun endlich soweit, das Modell für Vorhersagen einzusetzen. Nehmen wir an, Sie möchten wissen, wie glücklich Zyprioten sind. Die Daten des OECD liefern darauf keine Antwort. Glücklicherweise können Sie unser Modell verwenden, um eine gute Vorhersage zu treffen: Sie schlagen das Pro-Kopf-BIP für Zypern nach, finden 22587 USD und wenden Ihr Modell an. Dabei finden Sie heraus, dass die Zufriedenheit irgendwo um $4.85 + 22,587 \times 4.91 \times 10^{-5} = 5.96$ liegt.

Um Ihren Appetit auf die folgenden Kapitel anzuregen, zeigt Beispiel 1-1 den Python-Code, der die Daten lädt, vorbereitet,⁶ einen Scatterplot zeichnet, ein lineares Modell trainiert und eine Vorhersage trifft.⁷

Beispiel 1-1: Trainieren und Ausführen eines linearen Modells mit Scikit-Learn

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# Laden der Daten
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv", thousands=',', delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Vorbereiten der Daten
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualisieren der Daten
country_stats.plot(kind='scatter', x="Pro-Kopf-BIP", y='Zufriedenheit')
plt.show()

# Auswahl eines linearen Modells
model = sklearn.linear_model.LinearRegression()

# Trainieren des Modells
model.fit(X, y)

# Treffen einer Vorhersage für Zypern
X_new = [[22587]] # Pro-Kopf-BIP für Zypern
print(model.predict(X_new)) # Ausgabe [[ 5.96242338]]
```



Wenn Sie einen instanzbasierten Lernalgorithmus verwendet hätten, würden Sie herausbekommen, dass Slowenien das zu Zypern ähnlichste Pro-Kopf-BIP hat (20732 USD). Da uns die OECD-Daten die Zufriedenheit mit 5.7 angeben, hätten Sie für Zypern eine Zufriedenheit von 5.7 vorhergesagt. Wenn Sie ein wenig herauszoomen und sich die nächstgelegenen Länder ansehen, finden

⁶ Der Code geht davon aus, dass die Funktion `prepare_country_stats()` bereits definiert ist: Sie verbindet die Daten zu BIP und Zufriedenheit zu einem einzelnen Pandas-DataFrame.

⁷ Es ist in Ordnung, wenn Sie nicht gleich den gesamten Code nachvollziehen können; wir werden Scikit-Learn in den folgenden Kapiteln vorstellen.

Sie Portugal und Spanien mit Zufriedenheitswerten von jeweils 5.1 und 6.5. Der Mittelwert dieser drei Werte ist 5.77, was sehr nah an Ihrer modellbasierten Vorhersage liegt. Dieses einfache Verfahren nennt man *k-nächste-Nachbarn*-Regression (in diesem Beispiel mit $k = 3$).

Das Ersetzen des linearen Regressionsmodells durch k-nächste-Nachbarn-Regression im obigen Code erfordert lediglich das Ersetzen der Zeile:

```
model = sklearn.linear_model.LinearRegression()  
durch diese:
```

```
model = sklearn.neighbors.KNeighborsRegressor(  
n_neighbors=3)
```

Wenn alles gut gegangen ist, wird Ihr Modell gute Vorhersagen treffen. Wenn nicht, müssen Sie weitere Merkmale heranziehen (Beschäftigungsquote, Gesundheit, Luftverschmutzung und so weiter), sich mehr oder hochwertigere Trainingsdaten beschaffen oder ein mächtigeres Modell auswählen (z.B. ein polynomielles Regressionsmodell).

Zusammengefasst:

- Sie haben die Daten untersucht.
- Sie haben ein Modell ausgewählt.
- Sie haben es auf Trainingsdaten trainiert (d.h., der Trainingsalgorithmus hat nach den Modellparametern gesucht, die eine Kostenfunktion minimieren).
- Schließlich haben Sie das Modell verwendet, um für neue Fälle Vorhersagen zu treffen (dies nennt man *Inferenz*), und hoffen, dass das Modell gut verallgemeinert.

So sieht ein typisches Machine-Learning-Projekt aus. In Kapitel 2 können Sie dies selbst erfahren, indem Sie ein Projekt vom Anfang bis zum Ende durcharbeiten. Wir haben bisher ein weites Feld beschritten: Sie wissen bereits, worum es beim Machine Learning wirklich geht, warum es nützlich ist, welche Arten von ML-Systemen verbreitet sind und wie ein typischer Arbeitsablauf aussieht. Nun werden wir uns anschauen, was beim Lernen schiefgehen kann und präzise Vorhersagen verhindert.

Die wichtigsten Herausforderungen beim Machine Learning

Kurz gesagt, da Ihre Hauptaufgabe darin besteht, einen Lernalgorithmus auszuwählen und mit Daten zu trainieren, sind die zwei möglichen Fehlerquellen dabei ein »schlechter Algorithmus« und »schlechte Daten«. Beginnen wir mit Beispielen für schlechte Daten.

Unzureichende Menge an Trainingsdaten

Damit ein Kleinkind lernt, was ein Apfel ist, genügt es, dass Sie auf einen Apfel zeigen und »Apfel« sagen (und die Prozedur einige Male wiederholen). Damit ist das Kind in der Lage, Äpfel in allen möglichen Farben und Formen zu erkennen. Genial.

Machine Learning ist noch nicht ganz so weit; bei den meisten maschinellen Lernverfahren sind eine Menge Daten erforderlich, damit sie funktionieren. Selbst bei sehr einfachen Aufgaben benötigen Sie üblicherweise Tausende von Beispielen, und bei komplexen Aufgaben wie Bild- oder Spracherkennung können es auch Millionen sein (außer Sie können Teile eines existierenden Modells wiederverwenden).

Die unverschämte Effektivität von Daten

In einem berühmten Artikel (<http://goo.gl/R5enIE>) aus dem Jahr 2001 zeigten die Forscher Michele Banko und Eric Brill bei Microsoft, dass sehr unterschiedliche maschinelle Lernalgorithmen, darunter sehr primitive, bei einem sehr komplexen Problem wie der Unterscheidung von Sprache etwa gleich gut abschnitten,⁸ wenn man ihnen nur genug Daten zur Verfügung stellt (wie Sie in Abbildung 1-20 sehen können).

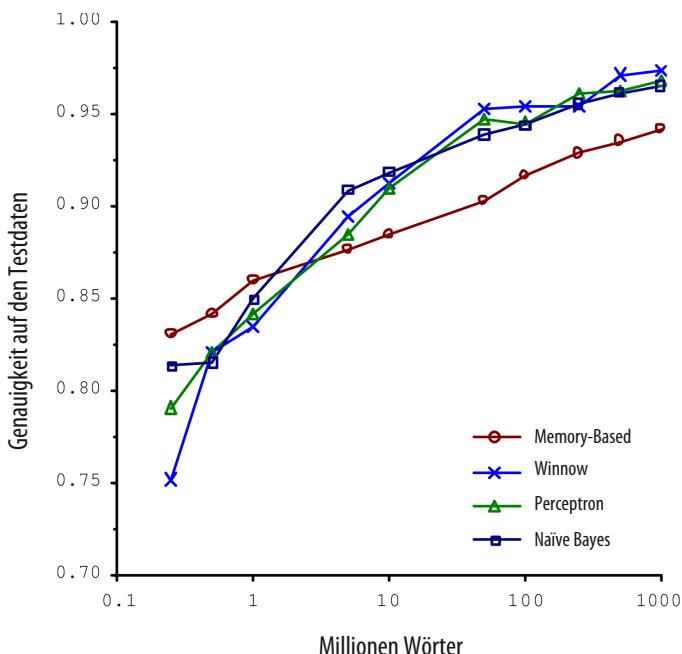


Abbildung 1-20: Die Wichtigkeit der Daten im Vergleich zum Algorithmus⁹

8 Beispielsweise aus dem Kontext zu ermitteln, ob man »to«, »two« oder »too« schreiben muss.

Die Autoren drücken dies folgendermaßen aus: »Diese Ergebnisse legen nahe, dass wir unsere Entscheidung über das Investieren von Zeit und Geld in die Entwicklung von Algorithmen gegenüber der Entwicklung eines Datenkorpus neu bewerten sollten.«

Dass Daten bei komplexen Problemen wichtiger als Algorithmen sind, wurde von Peter Norvig et al. in einem Artikel mit dem Titel »The Unreasonable Effectiveness of Data« (<http://goo.gl/q6LaZ8>) published in 2009.¹⁰ weiter thematisiert. Es sollte jedoch betont werden, dass kleine und mittelgroße Datensätze nach wie vor sehr häufig sind und dass es nicht immer einfach oder billig ist, an zusätzliche Trainingsdaten heranzukommen. Daher schreiben Sie die Algorithmik besser nicht gleich ab.

Nicht repräsentative Trainingsdaten

Um gut zu verallgemeinern, ist es entscheidend, dass Ihre Trainingsdaten die zu verallgemeinernden neuen Situationen repräsentieren. Dies ist sowohl beim instanzbasierten und beim modellbasierten Lernen der Fall.

Beispielsweise waren die zuvor zum Trainieren eines linearen Modells eingesetzten Länder nicht perfekt repräsentativ; einige Länder fehlten. Abbildung 1-21 zeigt, wie die Daten mit den fehlenden Ländern aussehen.

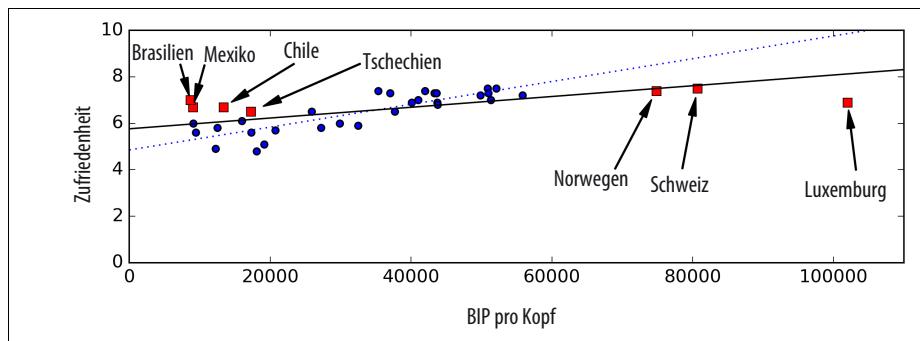


Abbildung 1-21: Ein repräsentativerer Trainingsdatensatz

Wenn Sie mit diesen Daten ein lineares Modell trainieren, erhalten Sie die durchgezogene Linie. Das alte Modell ist durch die gepunktete Linie dargestellt. Wie Sie sehen, verändert sich nicht nur das Modell durch die Daten. Es wird auch deutlich, dass ein einfaches lineares Modell vermutlich nie gut funktionieren wird. Es sieht ganz danach aus, dass reiche Länder nicht glücklicher als Länder mit mittlerem

9 Die Abbildung wurde mit Erlaubnis von Banko and Brill (2001), »Learning Curves for Confusion Set Disambiguation« reproduziert.

10 »The Unreasonable Effectiveness of Data«, Peter Norvig et al. (2009).

Wohlstand sind (sie wirken sogar weniger glücklich). Auch einige arme Länder scheinen glücklicher als viele reiche Länder zu sein.

Das mit einem nicht repräsentativen Datensatz trainierte Modell trifft also unge nauen Vorhersagen, besonders bei sehr armen und sehr reichen Ländern.

Es ist daher wichtig, einen Trainingsdatensatz zu verwenden, in dem die verallgemeinernden Fälle abgebildet sind. Oft ist dies schwieriger, als es sich an hört: Wenn die Stichprobe zu klein ist, erhalten Sie *Stichprobenrauschen* (also durch Zufall nicht repräsentative Daten). Selbst sehr große Stichproben können nicht repräsentativ sein, wenn die Methode zur Erhebung fehlerhaft ist. Dies nennt man auch *Stichprobenverzerrung*.

Ein berühmtes Beispiel für Stichprobenverzerrung

Das vermutlich berühmteste Beispiel für Stichprobenverzerrung stammt aus der US-Präsidentenwahl von 1936, bei der Landon gegen Roosevelt antrat: Der *Literary Digest* führte damals eine sehr große Umfrage durch, bei der Briefe an etwa 10 Millionen Menschen verschickt wurden. Es wurden 2.4 Millionen Antworten gesammelt und daraus mit hoher Konfidenz vorhergesagt, dass Landon 57% der Stimmen erhalten würde. Tatsächlich gewann aber Roosevelt mit 62% der Stimmen. Der Fehler lag in der Methode, die *Literary Digest* beim Erheben der Stichprobe einsetzte:

- Erstens verwendete *Literary Digest* Telefonbücher, Abonnentenlisten, Mitgliederlisten von Klubs und so weiter, um an die Adressen zum Verschicken der Umfrage zu kommen. In allen diesen Listen waren wohlhabendere Menschen stärker vertreten, die wahrscheinlich für die Republikaner (und damit Landon) stimmen würden.
- Zweitens antworteten weniger als 25% der Menschen auf die Umfrage. Auch dies führte zu einer Stichprobenverzerrung, da Menschen, die sich nicht für Politik interessieren, Menschen, die den *Literary Digest* nicht mögen, und andere wichtige Gruppen aussortiert wurden. Diese Art von Stichprobenverzerrung nennt man auch *Schweigeverzerrung*.

Ein weiteres Beispiel: Sagen wir, Sie möchten ein System zum Erkennen von Funk-Musikvideos konstruieren. Eine Möglichkeit zum Zusammenstellen der Trainingsdaten wäre, »funk music« bei YouTube einzugeben und die erhaltenen Videos zu verwenden. Allerdings nehmen Sie dabei an, dass Ihnen die Suchmaschine von YouTube Videos liefert, die repräsentativ für alle Funk-Musikvideos auf YouTube sind. In der Realität werden einige beliebte Künstler in den Suchergebnissen überrepräsentiert sein (und wenn Sie in Brasilien leben, erhalten Sie eine Menge Videos zu »funk carioca«, die sich überhaupt nicht wie James Brown anhören). Andererseits, wie sonst sollten Sie einen großen Datensatz sammeln?

Minderwertige Daten

Wenn Ihre Trainingsdaten voller Fehler, Ausreißer und Rauschen sind (z.B. wegen schlechter Messungen), ist es für das System schwieriger, die zugrunde liegenden Muster zu erkennen. Damit ist weniger wahrscheinlich, dass Ihr System eine hohe Leistung erzielt. Meistens lohnt es sich, Zeit in das Säubern der Trainingsdaten zu investieren. Tatsächlich verbringen die meisten Data Scientists einen Großteil Ihrer Zeit mit nichts anderem. Beispielsweise:

- Wenn einige Datenpunkte deutliche Ausreißer sind, hilft es, diese einfach zu entfernen oder die Fehler von Hand zu beheben.
- Wenn manche Merkmale lückenhaft sind (z.B. 5% Ihrer Kunden ihr Alter nicht angegeben haben), müssen Sie sich entscheiden, ob Sie dieses Merkmal insgesamt ignorieren möchten, die entsprechenden Datenpunkte entfernen, die fehlenden Werte ergänzen (z.B. mit dem Median) oder ein Modell mit diesem Merkmal und eines ohne dieses Merkmal trainieren möchten und so weiter.

Irrelevante Merkmale

Eine Redewendung besagt: Müll rein, Müll raus. Ihr System wird nur etwas erlernen können, wenn Ihre Trainingsdaten genug relevante Merkmale und nicht zu viele irrelevante enthalten. Ein für den Erfolg eines Machine-Learning-Projekts maßgeblicher Schritt ist, die Merkmale zum Trainieren gut auszuwählen. Zu diesem *Entwicklung von Merkmalen* genannten Vorgang gehören:

- die *Auswahl von Merkmalen*: aus den vorhandenen die nützlichsten Merkmale für das Trainieren auszuwählen.
- die *Extraktion von Merkmalen*: vorhandene Merkmale miteinander zu kombinieren, sodass ein nützlicheres entsteht (wie wir oben gesehen haben, helfen dabei Algorithmen zur Dimensionsreduktion).
- das Erstellen neuer Merkmale durch das Erheben neuer Daten.

Nun, da wir viele Beispiele für schlechte Daten kennengelernt haben, schauen wir uns auch einige schlechte Algorithmen an.

Overfitting der Trainingsdaten

Sagen wir, Sie sind im Ausland unterwegs, und der Taxifahrer zockt Sie ab. Sie mögen versucht sein zu sagen, dass *alle* Taxifahrer in diesem Land Betrüger seien. Menschen neigen häufig zu übermäßiger Verallgemeinerung, und Maschinen können leider in die gleiche Falle tappen, wenn wir nicht vorsichtig sind. Beim Machine Learning nennt man dies *Overfitting*: Dabei funktioniert das Modell auf den Trainingsdaten, kann aber nicht gut verallgemeinern.

Abbildung 1-22 zeigt ein Beispiel für ein polynomielles Modell höheren Grades für die Zufriedenheit, das die Trainingsdaten stark overfittet. Es erzielt zwar auf den Trainingsdaten eine höhere Genauigkeit als das einfache lineare Modell, aber würden Sie den Vorhersagen dieses Modells wirklich trauen?

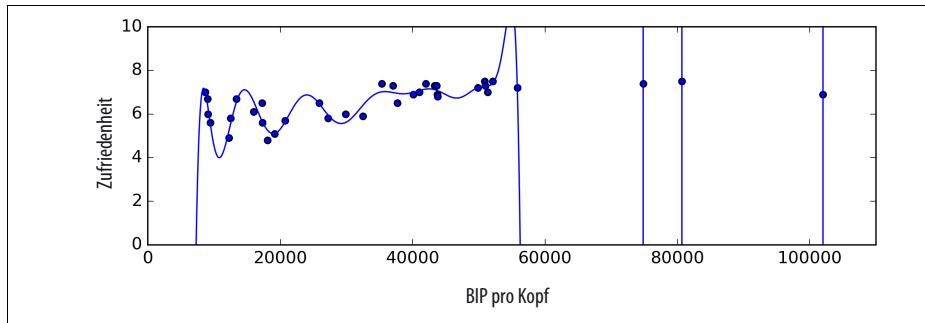


Abbildung 1-22: Overfitting der Trainingsdaten

Komplexe Modelle wie Deep-Learning-Netze können subtile Muster in den Daten erkennen. Wenn aber der Trainingsdatensatz verrauscht oder zu klein ist (wodurch die Stichprobe verrauscht ist), entdeckt das Modell Muster im Rauschen selbst. Diese Muster lassen sich natürlich nicht auf neue Daten übertragen. Nehmen wir beispielsweise an, Sie stellen Ihrem Modell für die Zufriedenheit viele weitere Merkmale zur Verfügung, darunter wenig informative wie den Namen des Landes. Ein komplexes Modell könnte dann herausfinden, dass alle Länder mit einer Zufriedenheit über 7 ein *w* im Namen haben: New Zealand (7.3), Norway (7.4), Sweden (7.2) und Switzerland (7.5). Wie sicher können Sie sich sein, dass diese W-Regel sich auf Rwanda oder Zimbabwe anwenden lässt? Natürlich trat dieses Muster in den Trainingsdaten rein zufällig auf, aber das Modell ist nicht in der Lage zu entscheiden, ob ein Muster echt oder durch das Rauschen in den Daten bedingt ist.



Overfitting tritt auf, wenn das Modell angesichts der Menge an Trainingsdaten und der Menge an Rauschen zu komplex ist. Mögliche Lösungen sind:

- das Modell zu vereinfachen, indem man es durch eines mit weniger Parametern ersetzt (z.B. ein lineares Modell statt eines polynomiellem Modells höheren Grades), die Anzahl der Merkmale im Trainingsdatensatz verringert oder dem Modell Restriktionen auferlegt,
- mehr Trainingsdaten zu sammeln,
- oder das Rauschen in den Trainingsdaten zu reduzieren (z.B. Datenfehler zu beheben und Ausreißer zu entfernen).

Einem Modell Restriktionen aufzuerlegen, um es zu vereinfachen und das Risiko für Overfitting zu reduzieren, wird als *Regularisierung* bezeichnet. Beispielsweise

hat das oben definierte lineare Modell zwei Parameter, θ_0 und θ_1 . Damit hat der Lernalgorithmus zwei *Freiheitsgrade*, mit denen das Modell an die Trainingsdaten angepasst werden kann: Sowohl die Höhe (θ_0) als auch die Steigung (θ_1) der Geraden lassen sich verändern. Wenn wir $\theta_1 = 0$ erzwingen würden, hätte der Algorithmus nur noch einen Freiheitsgrad, und es würde viel schwieriger, die Daten gut zu fitten: Die Gerade könnte sich nur noch nach oben oder unten bewegen, um so nah wie möglich an den Trainingsdatenpunkten zu landen. Sie würde daher in der Nähe des Mittelwerts landen. Wirklich ein sehr einfaches Modell! Wenn wir dem Modell erlauben, θ_1 zu verändern, aber einen kleinen Wert erzwingen, hat der Lernalgorithmus zwischen einem und zwei Freiheitsgraden. Das entstehende Modell ist einfacher als das mit zwei Freiheitsgraden, aber komplexer als mit nur einem. Ihre Aufgabe ist, die richtige Balance zwischen dem perfekten Fitten der Daten und einem möglichst einfachen Modell zu finden, sodass es gut verallgemeinert.

Abbildung 1-23 zeigt drei Modelle: Die gepunktete Linie steht für das ursprüngliche mit einigen fehlenden Ländern trainierte Modell, die gestrichelte Linie für unser zweites, mit allen Ländern trainierte Modell, und die durchgezogene Linie ist ein Modell, das mit den gleichen Daten wie das erste Modell trainiert wurde, aber mit zusätzlicher Regularisierung. Sie sehen, dass die Regularisierung eine geringere Steigung erzwungen hat, was etwas schlechter zu den Trainingsdaten passt, aber eine bessere Verallgemeinerung auf neue Beispiele erlaubt.

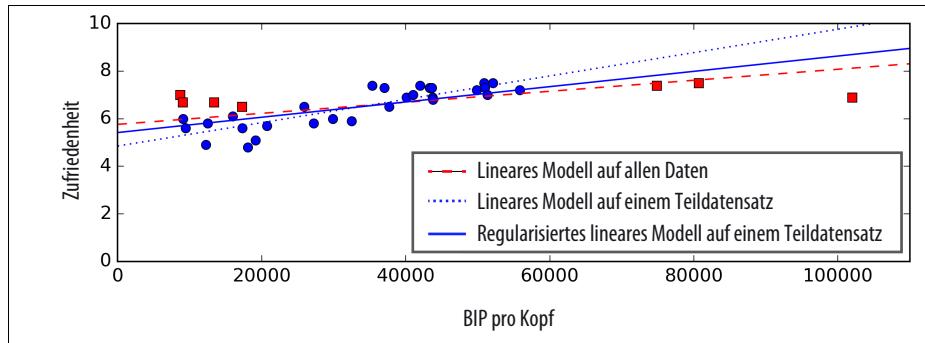


Abbildung 1-23: Regularisierung reduziert das Risiko für Overfitting

Die Stärke der Regularisierung beim Lernen lässt sich über einen *Hyperparameter* kontrollieren. Ein Hyperparameter ist ein Parameter des Lernalgorithmus (nicht des Modells). Als solcher unterliegt er nicht dem Lernprozess selbst; er muss vor dem Training gesetzt werden und bleibt über den gesamten Trainingszeitraum konstant. Wenn Sie den Hyperparameter zur Regularisierung auf einen sehr großen Wert setzen, erhalten Sie ein beinahe flaches Modell (eine Steigung nahe null); Sie können sich sicher sein, dass der Lernalgorithmus die Trainingsdaten nicht overfittet, eine gute Lösung wird aber ebenfalls unwahrscheinlicher. Die Feineinstellung der Hyperparameter ist ein wichtiger Teil bei der Entwicklung eines Machine-Learning-Systems (im nächsten Kapitel lernen Sie ein detailliertes Beispiel kennen).

Underfitting der Trainingsdaten

Wie Sie sich denken können, ist *Underfitting* das genaue Gegenteil von Overfitting: Es tritt auf, wenn Ihr Modell zu einfach ist, um die in den Daten enthaltene Struktur zu erlernen. Beispielsweise ist ein lineares Modell der Zufriedenheit anfällig für Underfitting; die Realität ist einfacher komplexer als unser Modell, sodass Vorhersagen selbst auf den Trainingsdaten zwangsläufig ungenau werden.

Die wichtigsten Möglichkeiten, dieses Problem zu beheben, sind:

- ein mächtigeres Modell mit mehr Parametern zu verwenden,
- dem Lernalgorithmus bessere Merkmale zur Verfügung zu stellen (Entwicklung von Merkmalen) oder
- die Restriktionen des Modells zu verringern (z.B. die Hyperparameter zur Regularisierung zu verringern).

Zusammenfassung

Inzwischen wissen Sie schon eine Menge über Machine Learning. Wir haben aber so viele Begriffe behandelt, dass Sie sich vielleicht ein wenig verloren vorkommen. Betrachten wir deshalb noch einmal das Gesamtbild:

- Beim Machine Learning geht es darum, Maschinen bei der Lösung einer Aufgabe zu verbessern, indem sie aus Daten lernen, anstatt explizit definierte Regeln zu erhalten.
- Es gibt viele unterschiedliche Arten von ML-Systemen: überwachte und unüberwachte, Batch- und Online-Learning, instanzbasierte und modellbasierte Systeme und so weiter.
- In einem ML-Projekt sammeln Sie Daten in einem Trainingsdatensatz und speisen diesen in einen Lernalgorithmus ein. Wenn der Algorithmus auf einem Modell basiert, stellt er einige Parameter ein, um das Modell den Trainingsdaten anzupassen (d.h., um gute Vorhersagen auf den Trainingsdaten selbst zu treffen). Danach ist es hoffentlich in der Lage, auch für neue Daten gute Vorhersagen zu treffen. Wenn der Algorithmus instanzbasiert ist, lernt er die Beispiele einfach auswendig und verwendet zur Verallgemeinerung auf neue Daten ein Ähnlichkeitsmaß.
- Wenn der Trainingsdatensatz zu klein ist, die Daten nicht repräsentativ, verrauscht oder durch irrelevante Merkmale verunreinigt sind, wird das System keine hohe Leistung erbringen (Müll rein, Müll raus). Schließlich darf Ihr Modell weder zu einfach (dann underfittet es) noch zu komplex sein (dann overfittet es).

Es gilt noch ein letztes wichtiges Thema zu behandeln: Sobald Sie ein Modell trainiert haben, sollten Sie nicht nur »hoffen«, dass es gut verallgemeinert. Sie sollten es

evaluieren und, falls nötig, Feinabstimmungen vornehmen. Sehen wir einmal, wie das geht.

Testen und Validieren

Ein Modell mit neuen Datenpunkten auszuprobieren, ist tatsächlich die einzige Möglichkeit zu erfahren, ob es gut auf neue Daten verallgemeinert. Sie können das Modell dazu in einem Produktivsystem einsetzen und beobachten, wie es funktioniert. Wenn sich Ihr Modell aber als definitiv untauglich herausstellt, werden sich Ihre Nutzer beschweren – nicht die beste Idee.

Eine bessere Alternative ist, Ihre Daten in zwei Datensätze zu unterteilen: den *Trainingsdatensatz* und den *Testdatensatz*. Wie die Namen vermuten lassen, trainieren Sie Ihr Modell mit dem Trainingsdatensatz und testen es mit dem Testdatensatz. Die Abweichung bei neuen Datenpunkten bezeichnet man als *Verallgemeinerungsfehler* (engl. *out-of-sample error*). Indem Sie Ihr Modell auf dem Testdatensatz evaluieren, erhalten Sie eine Schätzung dieser Abweichung. Dieser Wert verrät Ihnen, wie gut Ihr Modell auf zuvor nicht bekannten Datenpunkten abschneidet.

Wenn der Fehler beim Training gering (Ihr Modell also auf dem Trainingsdatensatz wenige Fehler begeht), aber der Verallgemeinerungsfehler groß ist, overfittet Ihr Modell die Trainingsdaten.



Es ist üblich, 80% der Daten zum Trainieren und 20% zum Testen *zurückzuhalten*.

Das Evaluieren eines Modells ist einfach: Verwenden Sie einen Testdatensatz. Nehmen wir an, Sie schwanken zwischen zwei Modellen (z.B. einem linearen und einem polynomiellen Modell): Wie sollen Sie sich entscheiden? Sie können natürlich beide trainieren und mit dem Testdatensatz vergleichen, wie gut beide verallgemeinern.

Nehmen wir an, das lineare Modell verallgemeinert besser, aber Sie möchten durch Regularisierung dem Overfitting entgegenwirken. Die Frage ist: Wie sollen Sie einen Wert Hyperparameter zur Regularisierung wählen? Natürlich können Sie 100 unterschiedliche Modelle mit 100 unterschiedlichen Werten für diesen Hyperparameter trainieren. Nehmen wir an, Sie finden einen optimalen Wert für den Hyperparameter, der den niedrigsten Verallgemeinerungsfehler liefert, z.B. 5 %.

Sie bringen Ihr Modell daher in die Produktivumgebung, aber leider schneidet es nicht wie erwartet ab und produziert einen Fehler von 15 %. Was ist passiert?

Das Problem ist, dass Sie den Verallgemeinerungsfehler mithilfe des Testdatensatzes mehrfach bestimmt und das Modell und seine Hyperparameter so angepasst

haben, dass es *für diesen Datensatz* das beste Modell hervorbringt. Damit erbringt das Modell bei neuen Daten wahrscheinlich keine gute Leistung.

Eine übliche Lösung dieses Problems ist ein zweiter zurückgehaltener Datensatz, der *Validierungsdatensatz*. Sie trainieren mehrere Modelle mit unterschiedlichen Hyperparametern auf dem Trainingsdatensatz und wählen das auf dem Validierungsdatensatz am besten abschneidende Modell und die Hyperparameter aus. Erst wenn Sie mit Ihrem Modell zufrieden sind, führen Sie einen einzelnen abschließenden Test mit dem Testdatensatz durch, um den Verallgemeinerungsfehler abzuschätzen.

Um nicht zu viele Trainingsdaten mit dem Validierungsdatensatz zu »vergeuden«, ist die *Kreuzvalidierung* eine verbreitete Technik: Der Trainingsdatensatz wird in komplementäre Untermengen aufgeteilt. Jedes Modell wird mit einer anderen Kombination dieser Untermengen trainiert und mit den restlichen Teilen validiert. Sobald die Art des Modells und die Hyperparameter feststehen, wird ein endgültiges Modell mit den gefundenen Hyperparametern auf dem gesamten Trainingsdatensatz trainiert, und der Verallgemeinerungsfehler wird anhand des Testdatensatzes bestimmt.

Das No-Free-Lunch-Theorem

Ein Modell ist eine vereinfachte Version der Beobachtungen. Die Vereinfachung soll überflüssige Details eliminieren, die sich vermutlich nicht auf neue Datenpunkte verallgemeinern lassen. Um allerdings zu entscheiden, welche Daten zu verwerten und welche beizubehalten sind, müssen Sie *Annahmen* treffen. Beispielsweise trifft ein lineares Modell die Annahme, dass die Daten grundsätzlich linear sind und die Distanz zwischen den Datenpunkten und der Geraden lediglich Rauschen ist, das man ignorieren kann.

In einem berühmten Artikel aus dem Jahr 1996 (<https://goo.gl/dzp946>)¹¹ zeigte David Wolpert, dass es keinen Grund gibt, ein Modell gegenüber einem anderen zu bevorzugen, wenn Sie absolut keine Annahmen über die Daten treffen. Dies nennt man auch das *No-Free-Lunch-(NFL-)Theorem*. Bei einigen Datensätzen ist das beste Modell ein lineares Modell, während bei anderen ein neuronales Netz am besten geeignet ist. Es gibt kein Modell, das garantiert *a priori* besser funktioniert (daher der Name des Theorems). Der einzige Weg, wirklich sicherzugehen, ist, alle möglichen Modelle zu evaluieren. Da dies nicht möglich ist, treffen Sie in der Praxis einige wohlüberlegte Annahmen über die Daten und evaluieren nur einige sinnvoll ausgewählte Modelle. Bei einfachen Aufgaben könnten Sie beispielsweise lineare Modelle mit unterschiedlich starker Regularisierung auswerten, bei einer komplexen Aufgabe hingegen verschiedene neuronale Netze.

11 »The Lack of A Priori Distinctions Between Learning Algorithms«, D. Wolpert (1996).

Übungen

In diesem Kapitel haben wir einige der wichtigsten Begriffe zum Machine Learning behandelt. In den folgenden Kapiteln werden wir uns eingehender damit beschäftigen und mehr Code schreiben, aber zuvor sollten Sie sicherstellen, dass Sie die folgenden Fragen beantworten können:

1. Wie würden Sie Machine Learning definieren?
2. Können Sie vier Arten von Aufgaben nennen, für die Machine Learning gut geeignet ist?
3. Was ist ein gelabelter Trainingsdatensatz?
4. Was sind die zwei verbreitetsten Aufgaben beim überwachten Lernen?
5. Können Sie vier verbreitete Aufgaben für unüberwachtes Lernen nennen?
6. Was für einen Machine-Learning-Algorithmus würden Sie verwenden, um einen Roboter über verschiedene unbekannte Oberflächen laufen zu lassen?
7. Welche Art Algorithmus würden Sie verwenden, um Ihre Kunden in unterschiedliche Gruppen einzuteilen?
8. Würden Sie die Aufgabe, Spam zu erkennen, als überwachte oder unüberwachte Lernaufgabe einstufen?
9. Was ist ein Online-Lernsystem?
10. Was ist Out-of-Core-Lernen?
11. Welche Art Lernalgorithmus beruht auf einem Ähnlichkeitsmaß, um Vorhersagen zu treffen?
12. Was ist der Unterschied zwischen einem Modellparameter und einem Hyperparameter eines Lernalgorithmus?
13. Wonach suchen modellbasierte Lernalgorithmen? Welches ist die häufigste Strategie, die zum Erfolg führt? Wie treffen sie Vorhersagen?
14. Können Sie vier der wichtigsten Herausforderungen beim Machine Learning benennen?
15. Welches Problem liegt vor, wenn Ihr Modell auf den Trainingsdaten eine sehr gute Leistung erbringt, aber schlecht auf neue Daten verallgemeinert? Nennen Sie drei Lösungsansätze.
16. Was ist ein Testdatensatz, und warum sollten man einen verwenden?
17. Was ist der Zweck eines Validierungsdatensatzes?
18. Was kann schiefgehen, wenn Sie Hyperparameter mithilfe der Testdaten einstellen?
19. Was ist Kreuzvalidierung, und warum sollten Sie diese einem Validierungsdatensatz vorziehen?

Lösungen zu diesen Übungsaufgaben finden Sie in Anhang A.

Ein Machine-Learning-Projekt von A bis Z

In diesem Kapitel werden Sie ein Beispielprojekt vom Anfang bis zum Ende erleben. Wir nehmen dazu an, Sie seien ein frisch angeheuerter Data Scientist in einer Immobilienfirma.¹ Folgendes sind die wichtigsten Schritte, die Sie bearbeiten werden:

1. Betrachte das Gesamtbild.
2. Beschaffe die Daten.
3. Erkunde und visualisiere die Daten, um daraus Erkenntnisse zu gewinnen.
4. Bereite die Daten für Machine-Learning-Algorithmen vor.
5. Wähle ein Modell aus und trainiere es.
6. Verfeinere das Modell.
7. Präsentiere die Lösung.
8. Nimm das System in Betrieb, beobachte und warte es.

Der Umgang mit realen Daten

Wenn Sie Machine Learning gerade erst erlernen, experimentieren Sie am besten mit realen Daten, nicht mit künstlich generierten Datensätzen. Glücklicherweise stehen Ihnen Tausende frei verfügbarer Datensätze aus allen möglichen Fachgebieten zur Auswahl. Hier sind einige Quellen, wo Sie passende Daten finden können:

- Beliebte Archive mit frei verfügbaren Daten:
 - UC Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>)
 - Datensätze von Kaggle (<https://www.kaggle.com/datasets>)
 - Datensätze von Amazon AWS (<http://aws.amazon.com/fr/datasets/>)
- Metaseiten (Listen von Datenarchiven):
 - <http://dataportals.org/>

¹ Dieses Beispielprojekt ist komplett erfunden; das Ziel ist, die wichtigsten Schritte eines Machine-Learning-Projekts zu illustrieren, nicht etwas über den Immobilienmarkt zu erfahren.

- <http://opendatamonitor.eu/>
- <http://quandl.com/>
- Andere Seiten, die beliebte offene Datenarchive auflisten:
 - Die Wikipedia-Seite mit Machine-Learning-Datensätzen (<https://goo.gl/SJHN2k>)
 - Fragen auf Quora.com (<http://goo.gl/zDR78y>)
 - subreddit zu Datensätzen (<https://www.reddit.com/r/datasets>)

Für dieses Kapitel suchen wir uns einen Datensatz zu Immobilienpreisen in Kalifornien aus dem StatLib Repository aus (siehe Abbildung 2-1).² Dieser Datensatz basiert auf Informationen aus der kalifornischen Volkszählung von 1990. Er ist nicht gerade aktuell (in der San Francisco Bay konnte man sich damals noch ein nettes Häuschen leisten), bietet aber viele Eigenschaften, anhand derer sich gut lernen lässt. Wir werden deshalb so tun, als wären die Daten aktuell. Wir haben aus didaktischen Gründen auch ein zusätzliches Merkmal hinzugefügt und einige Merkmale entfernt.

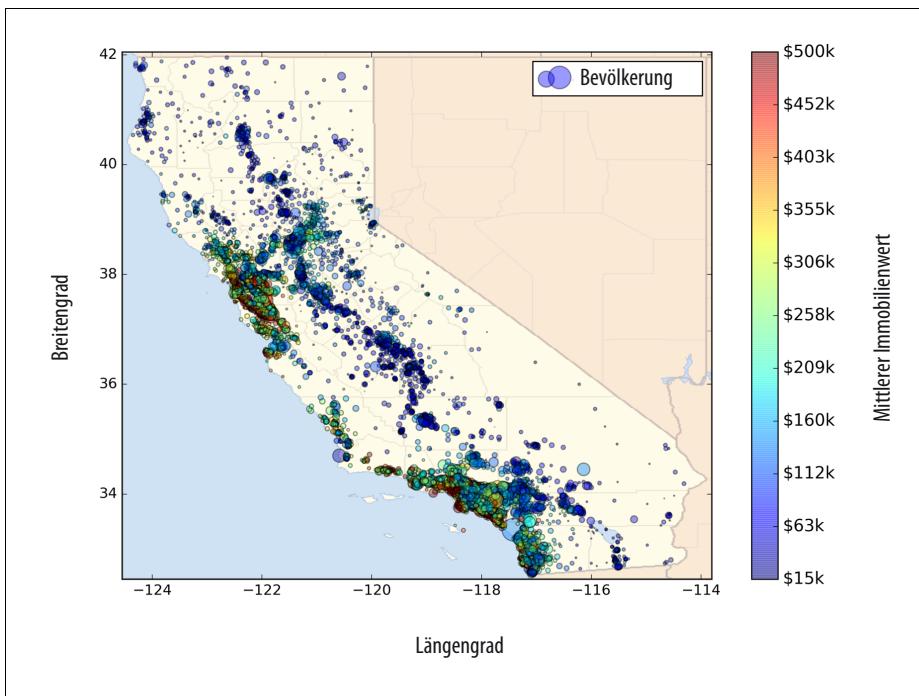


Abbildung 2-1: Immobilienpreise in Kalifornien

2 Der ursprüngliche Datensatz erschien in R. Kelley Pace and Ronald Barry, »Sparse Spatial Autoregressions«, *Statistics & Probability Letters* 33, no. 3 (1997): 291 – 297.

Betrachte das Gesamtbild

Willkommen beim Machine-Learning-Immobilienkonzern! Ihre erste Aufgabe ist, ein Modell der Immobilienpreise in Kalifornien mithilfe der Daten aus der kalifornischen Volkszählung zu erstellen. Diese Daten enthalten für jede Blockgruppe in Kalifornien Metriken wie die Bevölkerung, das mittlere Einkommen, die mittleren Immobilienpreise und so weiter. Blockgruppen sind die kleinste geografische Einheit, für die das US Census Bureau Daten veröffentlicht (eine Blockgruppe hat typischerweise eine Bevölkerung von 600 bis 3000 Menschen). Wir werden diese einfach »Bezirke« nennen.

Ihr Modell sollte aus diesen Daten lernen und in der Lage sein, den mittleren Immobilienpreis in einem beliebigen Bezirk aus allen übrigen Metriken vorherzusagen.



Da Sie ein gut organisierter Data Scientist sein möchten, ist Ihr erster Arbeitsschritt, Ihre Checkliste für Machine-Learning-Projekte zu zücken. Sie können mit der Liste in Anhang B beginnen; diese sollte für die meisten Machine-Learning-Projekte annehmbar funktionieren. Sie sollten sie aber an Ihre Bedürfnisse anpassen. In diesem Kapitel werden wir viele Punkte dieser Checkliste abarbeiten, aber auch einige selbsterklärende oder in späteren Kapiteln besprochene überspringen.

Die Aufgabe abstecken

Die allererste Frage an Ihren Vorgesetzten ist, was denn eigentlich das Geschäftsziel ist; ein Modell zu erstellen, ist vermutlich nicht das eigentliche Ziel. In welcher Weise möchte Ihre Firma das Modell voraussichtlich nutzen und davon profitieren? Von der Antwort hängt ab, wie Sie Ihre Aufgabenstellung formulieren, welche Algorithmen Sie auswählen, welches Qualitätsmaß Sie zum Auswerten des Modells verwenden und wie viel Aufwand Sie in die Optimierung stecken sollten.

Ihr Vorgesetzter antwortet, dass die Ausgabe Ihres Modells (eine Vorhersage des mittleren Immobilienpreises eines Bezirks) zusammen mit anderen *Signalen*³ in ein anderes Machine-Learning-System eingespeist werden soll (siehe Abbildung 2-2). Dieses nachgeschaltete System soll entscheiden, ob sich Investitionen in einer bestimmten Gegend lohnen oder nicht. Die Richtigkeit dieser Entscheidungen wirkt sich direkt auf die Einnahmen aus.

³ Eine in ein Machine-Learning-System eingegebene Information wird oft nach Shannons Informationstheorie als *Signal* bezeichnet: Ein hoher Quotient von Signal und Hintergrundrauschen ist wünschenswert.

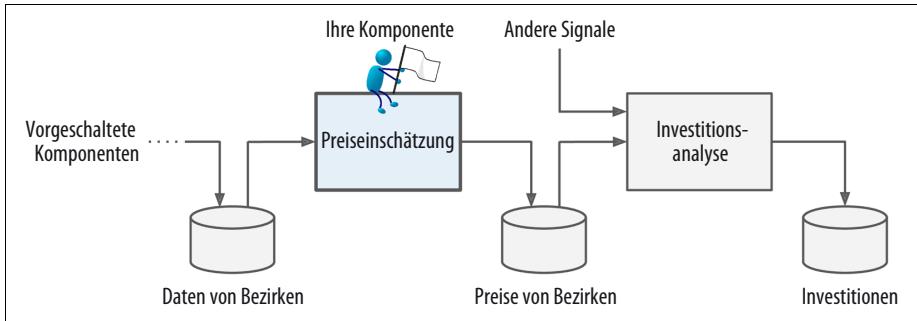


Abbildung 2-2: Eine Machine-Learning-Pipeline für Immobilieninvestitionen

Pipelines

Eine Abfolge von *Komponenten* zur Datenverarbeitung nennt man eine *Pipeline*. Pipelines sind in Machine-Learning-Systemen sehr häufig, weil dabei eine Menge Daten zu bearbeiten und viele Datentransformationen anzuwenden sind.

Diese Komponenten werden üblicherweise asynchron ausgeführt. Jede Komponente liest eine große Datenmenge ein, verarbeitet sie und schiebt die Ergebnisse in einen anderen Datenspeicher. Etwas später liest die nächste Komponente der Pipeline diese Daten ein, produziert ihre eigene Ausgabe und so weiter. Jede Komponente ist einigermaßen eigenständig: Als Schnittstelle zwischen den Komponenten dient der Datenspeicher. Dadurch ist das System recht einfach zu erfassen (mithilfe eines Datenflussdiagramms), und mehrere Teams können sich auf unterschiedliche Komponenten konzentrieren. Wenn außerdem eine Komponente ausfällt, können die nachgeschalteten Komponenten oft normal weiterarbeiten (zumindest für eine Weile), indem sie einfach die letzte Ausgabe der ausgefallenen Komponente verwenden. Dadurch ist diese Architektur recht robust.

Andererseits kann eine ausgefallene Komponente eine ganze Weile unbemerkt bleiben, falls das System nicht angemessen überwacht wird. Die Daten veralten dann, und die Leistung des Gesamtsystems sinkt.

Die nächste Frage ist, was für eine Lösung bereits verwendet wird (falls überhaupt). Häufig erhalten Sie dabei einen Referenzwert für die Leistung sowie Hinweise zum Lösen der Aufgabe. Ihr Vorgesetzter antwortet Ihnen, dass die Immobilienpreise der Bezirke im Moment von Experten manuell geschätzt werden: Ein Team sammelt aktuelle Informationen über einen Bezirk, und wenn es den mittleren Immobilienpreis nicht ermitteln kann, werden diese mithilfe komplexer Regeln geschätzt.

Dieses Verfahren ist sowohl kosten- als auch zeitintensiv, und die Schätzungen sind nicht besonders gut; wenn ein Team den mittleren Immobilienpreis herausfindet, stellt es häufig fest, dass es mit seiner Schätzung mehr als 10% daneben lag. Des-

halb möchte das Unternehmen ein Modell trainieren, um den mittleren Immobilienpreis eines Bezirks aus anderen Angaben über den Bezirk vorherzusagen. Die Daten aus der Volkszählung könnten eine großartige für diesen Zweck nutzbare Datenquelle sein, da sie den mittleren Immobilienpreis und andere Daten für Tausende Bezirke enthalten.

Mit all diesen Informationen sind Sie nun so weit, Ihr System zu entwerfen. Erstens müssen Sie Ihre Aufgabe abstecken: Handelt es sich um überwachtes Lernen, unüberwachtes Lernen oder Reinforcement Learning. Ist es eine Klassifikationsaufgabe, eine Regressionsaufgabe oder etwas anderes? Sollten Sie Techniken aus dem Batch-Learning oder Online-Learning verwenden? Bevor Sie weiterlesen, nehmen Sie sich einen Moment Zeit und versuchen, sich diese Fragen selbst zu beantworten.

Haben Sie die Antworten gefunden? Schauen wir einmal: Es ist ganz klar eine typische überwachte Lernaufgabe, da Ihnen Lernbeispiele mit *Labels* zur Verfügung stehen (jeder Datenpunkt enthält die erwartete Ausgabe, d. h. den mittleren Immobilienpreis eines Bezirks). Es ist außerdem eine typische Regressionsaufgabe, da Sie einen Zahlenwert vorhersagen sollen. Genauer gesagt, handelt es sich um eine *multivariate Regressionsaufgabe*, da das System mehrere Eigenschaften zum Treffen einer Vorhersage heranziehen wird (es wird die Bevölkerung eines Bezirks verwenden, das mittlere Einkommen und so weiter). Im ersten Kapitel haben Sie die Zufriedenheit aus nur einem Merkmal vorhergesagt, dem Pro-Kopf-Bruttoinlandsprodukt, also handelte es sich dabei um eine *univariate Regressionsaufgabe*. Schließlich gibt es keinen kontinuierlichen Strom neuer Daten in das System. Es gibt keinen besonderen Grund, sich auf schnell veränderliche Daten einzustellen, und der Datensatz ist so klein, dass er im Speicher Platz findet. Daher reicht gewöhnliches Batch-Learning völlig aus.



Wenn die Datenmenge riesig wäre, könnten Sie das Batch-Learning entweder auf mehrere Server verteilen (z.B. mit der *MapReduce*-Technik, die wir später noch kennenlernen werden) oder stattdessen eine Technik zum Online-Learning verwenden.

Wähle ein Qualitätsmaß aus

Der nächste Schritt besteht darin, ein geeignetes Qualitätsmaß auszuwählen. Ein typisches Qualitätsmaß für Regressionsaufgaben ist die Wurzel der mittleren quadratischen Abweichung (RMSE). Sie entspricht der Größe des Fehlers, den das System im Mittel bei Vorhersagen macht, wobei großen Fehlern ein höheres Gewicht beigemessen wird. Formel 2-1 zeigt die mathematische Formel zur Berechnung des RMSE.

Formel 2-1: Wurzel der mittleren quadratischen Abweichung (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m \left(h(\mathbf{x}^{(i)}) - y^{(i)} \right)^2}$$

Schreibweisen

In dieser Gleichung werden mehrere im Machine Learning übliche Schreibweisen verwendet, die wir im gesamten Buch wiedersehen werden:

- m ist die Anzahl Datenpunkte im Datensatz, für den der RMSE bestimmt wird.
 - Wenn Sie beispielsweise den RMSE für einen Validierungsdatensatz mit 2000 Bezirken auswerten, dann gilt $m = 2000$.
- $\mathbf{x}^{(i)}$ ist ein Vektor der Werte aller Merkmale (ohne das Label) des i^{ten} Datenpunkts im Datensatz, und $y^{(i)}$ ist das dazugehörige Label (der gewünschte Ausgabewert für diesen Datenpunkt).
 - Wenn der erste Bezirk beispielsweise bei -118.29° Länge und 33.91° nördlicher Breite liegt, 1416 Einwohner mit einem mittleren Einkommen von 38372 USD hat und der mittlere Immobilienpreis 156400 USD beträgt (die übrigen Merkmale ignorieren wir noch), dann gilt:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1416 \\ 38372 \end{pmatrix}$$

und:

$$y^{(1)} = 156400$$

- \mathbf{X} ist eine Matrix mit den Werten sämtlicher Merkmale (ohne Labels) für alle Datenpunkte im Datensatz. Pro Datenpunkt gibt es eine Zeile, und die i^{te} Zeile entspricht der transponierten Form von $\mathbf{x}^{(i)}$, auch als $(\mathbf{x}^{(i)})^T$ geschrieben.⁴
 - Beispielsweise sieht die Matrix \mathbf{X} für den oben beschriebenen ersten Bezirk folgendermaßen aus:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

⁴ Der Transponierungs-Operator wandelt einen Spaltenvektor in einen Zeilenvektor um und umgekehrt.

h ist die Vorhersagefunktion Ihres Systems, auch *Hypothese* genannt. Wenn Ihr System den Merkmalsvektor eines Datenpunkts $\mathbf{x}^{(i)}$ erhält, gibt es den Vorhersagewert $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ für diesen Datenpunkt aus.

- Wenn Ihr System beispielsweise im ersten Bezirk einen mittleren Immobilienpreis von 158400 USD vorhersagt, so ist $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400$. Der Vorhersagefehler für diesen Bezirk ist dann $\hat{y}^{(1)} - y^{(1)} = 2000$.
- RMSE(\mathbf{X}, h) ist die auf den Beispieldaten mit Ihrer Hypothese h gemessene Kostenfunktion.

Wir verwenden kursive Kleinbuchstaben für Skalare (wie m oder $y^{(i)}$) und Funktionen (wie h), fett gedruckte Kleinbuchstaben für Vektoren (wie $\mathbf{x}^{(i)}$) und fett gedruckte Großbuchstaben für Matrizen (wie \mathbf{X}).

Obwohl der RMSE bei Regressionsaufgaben grundsätzlich das Qualitätsmaß erster Wahl ist, ist in manchen Situationen eine andere Funktion vorzuziehen. Nehmen wir an, es gäbe viele Ausreißer unter den Bezirken. In diesem Fall könnten Sie den *mittleren absoluten Fehler* (MAE, auch als mittlere absolute Abweichung bezeichnet; siehe Formel 2-2) berücksichtigen:

Formel 2-2: mittlerer absoluter Fehler

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m \left| h(\mathbf{x}^{(i)}) - y^{(i)} \right|$$

Sowohl RMSE als auch MAE quantifizieren den Abstand zwischen zwei Vektoren: Dem Vektor aller Vorhersagen und dem Vektor mit den Zielwerten. Dabei sind unterschiedliche Abstandsmaße oder Normen möglich:

- Die Wurzel einer quadratischen Summe (RMSE) entspricht dem *Euklidischen Abstand*: Dies ist der Ihnen vertraute Abstand. Sie wird auch als ℓ_2 -Norm oder $\|\cdot\|_2$ (oder einfach $\|\cdot\|$) bezeichnet.
- Allgemeiner ist die ℓ_k -Norm eines Vektors \mathbf{v} mit n Elementen als

$$\|\mathbf{v}\|_k = \left(|v_0|^k + |v_1|^k + \dots + |v_n|^k \right)^{\frac{1}{k}} \text{ definiert. } \ell_0 \text{ gibt einfach nur die Anzahl der}$$

Elemente ungleich null im Vektor wieder, und ℓ_∞ berechnet den größten Absolutwert im Vektor an.

- Je höher der Index einer Norm ist, umso stärker berücksichtigt diese große Werte und vernachlässigt kleinere. Deshalb ist der RMSE empfindlicher für Ausreißer als der MAE. Sind Ausreißer aber exponentiell selten (wie in einer Glockenkurve), so funktioniert der RMSE sehr gut, und ist grundsätzlich vorzuziehen.

- Das Berechnen einer Summe von Absolutwerten (MAE) entspricht der ℓ_1 -Norm, geschrieben als $\|\cdot\|_1$. Diese nennt man bisweilen auch *Manhattan-Metrik*, da sie den Abstand zwischen zwei Punkten in einer Stadt angibt, in der man sich nur entlang rechtwinkliger Häuserblöcke bewegen kann.

Überprüfe die Annahmen

Es ist eine gute Angewohnheit, die bisher (von Ihnen oder von anderen) getroffenen Annahmen aufzuschreiben und zu überprüfen; damit können Sie größere Schwierigkeiten früh erkennen. Die von Ihrem System vorhergesagten Preise für einzelne Bezirke werden in ein nachgeschaltetes Machine-Learning-System eingegeben. Wir nehmen an, dass die Preise als solche verwendet werden. Was aber, wenn das nachgeschaltete System stattdessen die Preise in Kategorien einteilt (z.B. »günstig«, »mittel« oder »teuer«) und anstelle der Preise dann diese Kategorien verwendet? In diesem Fall wäre es überhaupt nicht wichtig, den Preis genau vorherzusagen; Ihr System müsste lediglich die Kategorie richtig bestimmen. Ist dies der Fall, sollte die Aufgabe als Klassifikation beschrieben werden, nicht als Regression. Zu dieser Art Erkenntnisse möchte man nicht erst nach monatelanger Arbeit an einem Regressionsystem gelangen.

Nachdem Sie mit dem für das nachgeschaltete System zuständigen Team gesprochen haben, sind Sie sich glücklicherweise sicher, dass Sie tatsächlich die Preise und keine Kategorien benötigen. Großartig! Damit sind wir gut aufgestellt und haben grünes Licht, um mit dem Programmieren zu beginnen!

Beschaffe die Daten

Es ist Zeit, sich die Hände schmutzig zu machen. Sie können sich gern Ihren Laptop nehmen und die folgenden Codebeispiele in einem Jupyter Notebook nachvollziehen. Das vollständige Jupyter Notebook finden Sie unter <https://github.com/ageron/handson-ml>.

Erstelle eine Arbeitsumgebung

Zuerst benötigen Sie eine Python-Installation. Python ist vermutlich bereits auf Ihrem System installiert. Falls nicht, finden Sie es unter <https://www.python.org/>⁵.

Als Nächstes müssen Sie sich ein Arbeitsverzeichnis für Ihren Machine-Learning-Code und die Datensätze erstellen. Öffnen Sie eine Kommandozeile und geben Sie die folgenden Befehle dort ein (nach dem \$-Prompt):

```
$ export ML_PATH="$HOME/ml"      # Sie können den Pfad ändern, wenn Sie möchten
$ mkdir -p $ML_PATH
```

⁵ Empfohlen ist die neueste Version von Python 3. Python 2.7+ sollte auch funktionieren, ist aber überholt.

Sie benötigen eine Anzahl Python-Module: Jupyter, NumPy, Pandas, Matplotlib und Scikit-Learn. Wenn Jupyter bei Ihnen bereits mit all diesen Modulen läuft, können Sie beruhigt bei Abschnitt »*Die Daten herunterladen*« auf Seite 44 fortfahren. Falls noch etwas fehlt, gibt es mehrere Möglichkeiten, diese Module (und ihre Paketabhängigkeiten) zu installieren. Sie können die Paketverwaltung Ihres Systems verwenden (z.B. apt-get unter Ubuntu oder MacPorts und HomeBrew unter macOS), eine wissenschaftliche Python-Distribution wie Anaconda installieren und dessen Paketverwaltung nutzen oder einfach das in Python eingebaute Paketverwaltungstool pip einsetzen, das (seit Python 2.7.9) standardmäßig Teil der binären Python-Distributionen ist.⁶ Mit dem folgenden Befehl können Sie überprüfen, ob pip installiert ist:

```
$ pip3 --version  
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

Sie sollten sicherstellen, dass Sie eine neuere Version von pip haben, mindestens aber größer 1.4, ab der die Installation von binären Modulen (wheels) möglich ist. Um das pip-Modul zu aktualisieren, geben Sie ein:⁷

```
$ pip3 install --upgrade pip  
Collecting pip  
[...]  
Successfully installed pip-9.0.1
```

Erstellen einer isolierten Umgebung

Falls Sie in einer isolierten Umgebung arbeiten möchten (was sehr empfehlenswert ist, um Konflikte zwischen Modulversionen in unterschiedlichen Projekten zu vermeiden), sollten Sie mit dem folgenden pip-Befehl das Programm virtualenv installieren:

```
$ pip3 install --user --upgrade virtualenv  
Collecting virtualenv  
[...]  
Successfully installed virtualenv
```

Nun können Sie eine isolierte Python-Umgebung erstellen:

```
$ cd $ML_PATH  
$ virtualenv env  
Using base prefix '[...]'  
New python executable in [...]/ml/env/bin/python3.5  
Also creating executable in [...]/ml/env/bin/python  
Installing setuptools, pip, wheel...done.
```

⁶ Wir zeigen hier die Installationsschritte mit pip auf der bash-Konsole eines Linux- oder macOS-Systems. Eventuell müssen Sie die Befehle an Ihr System anpassen. Unter Windows empfehlen wir die Installation von Anaconda.

⁷ Eventuell benötigen Sie Administratorrechte, um diesen Befehl auszuführen; ist dies der Fall, fügen Sie das Präfix sudo hinzu.

Jedes Mal, wenn Sie diese Umgebung aktivieren möchten, öffnen Sie einfach eine Kommandozeile und geben dort ein:

```
$ cd $ML_PATH  
$ source env/bin/activate
```

Solange diese Umgebung aktiv ist, wird jedes von pip installierte Paket in dieser isolierten Umgebung installiert. Python hat nur auf diese Pakete Zugriff (wenn Sie auch die auf dem gesamten System installierten Pakete nutzen möchten, sollten Sie beim Erstellen der Umgebung die Option `--system-site-packages` angeben). Weitere Informationen dazu finden Sie in der Dokumentation zu virtualenv.

Nun können Sie sämtliche benötigten Module und ihre Paketabhängigkeiten mit einem einfachen pip-Befehl installieren:

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn  
Collecting jupyter  
  Downloading jupyter-1.0.0-py2.py3-none-any.whl  
Collecting matplotlib  
  [...]
```

Zur Überprüfung Ihrer Installation versuchen Sie, alle Module folgendermaßen zu importieren:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

Es sollte weder eine Ausgabe noch eine Fehlermeldung zu sehen sein. Nun können Sie mit folgendem Befehl Jupyter starten:

```
$ jupyter notebook  
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml  
[I 15:24 NotebookApp] 0 active kernels  
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/  
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all  
kernels (twice to skip confirmation).
```

Nun läuft in Ihrer Kommandozeile ein Jupyter-Server auf Port 8888. Sie können diesen Server besuchen, indem Sie in Ihrem Browser die Adresse `http://localhost:8888/` eingeben (normalerweise passiert dies automatisch beim Starten des Servers). Sie sollten ein leeres Arbeitsverzeichnis sehen (das lediglich das Verzeichnis `env` enthält, falls Sie die obige Anleitung zum Installieren von virtualenv befolgt haben).

Erstellen Sie nun ein neues Python-Notebook mit dem Button »New« ❶ und wählen die gewünschte Python-Version aus ❷⁸ (siehe Abbildung 2-3).

Dabei passieren drei Dinge: Erstens wird eine neue Datei namens `Untitled.ipynb` für das Notebook in Ihrem Arbeitsverzeichnis angelegt; zweitens wird ein Python-Kernel zum Ausführen des Notebooks gestartet; drittens wird das Notebook in einem

⁸ Jupyter kann mit mehreren Python-Versionen und sogar vielen anderen Sprachen wie R oder Octave umgehen.

neuen Unterfenster geöffnet. Zu Beginn sollten Sie das Notebook zu »Housing« umbenennen **①** (Abbildung 2-4), indem Sie auf »Untitled« klicken und den neuen Namen eingeben (damit wird auch die Datei automatisch zu *Housing.ipynb* umbenannt).

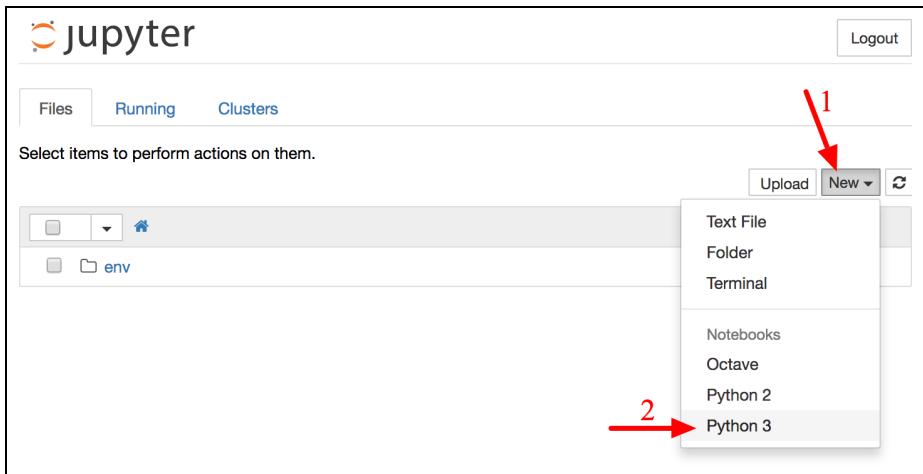


Abbildung 2-3: Ihre Arbeitsumgebung in Jupyter

Ein Notebook besteht aus einer Abfolge von Zellen. Jede Zelle kann ausführbaren Code oder formatierten Text enthalten. Im Moment enthält das Notebook nur eine leere Code-Zelle mit dem Inhalt »In [1]:«. Schreiben Sie in die Zelle `print("Hello world!")` **②** und drücken Sie dann auf den Play-Button **③** **①** (siehe Abbildung 2-4) oder drücken Sie Shift-Enter. Damit wird der Inhalt der aktuellen Zelle an den Python-Kernel des Notebooks geschickt, der diesen ausführt und eine Ausgabe zurückgibt. Das Ergebnis wird unterhalb der Zelle dargestellt, und weil wir uns am unteren Ende des Notebooks befinden, wird automatisch eine neue leere Zelle hinzugefügt. Zum Erlernen weiterer Grundlagen finden Sie im Hilfemenü von Jupyter eine Tour durch die Benutzeroberfläche.

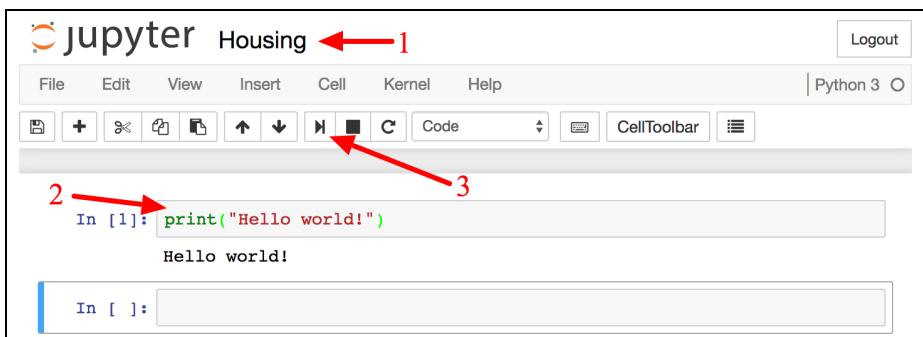


Abbildung 2-4: Python Notebook mit Hello World-Befehl

Die Daten herunterladen

In einer typischen Arbeitsumgebung wären Ihre Daten in einer relationalen Datenbank (oder einem anderen Datenspeicher) und über viele Tabellen/Dokumente/Dateien verteilt. Um auf sie zuzugreifen, müssten Sie zuerst Zugriffsrechte und Passwörter erhalten⁹ und sich mit dem Datenmodell vertraut machen. In diesem Projekt sind die Dinge jedoch deutlich einfacher: Sie laden *housing.tgz* herunter, eine einzelne komprimierte Datei, in der sämtliche Daten als kommaseparierte Datei (CSV) namens *housing.csv* vorliegen.

Sie könnten Ihren Browser zum Herunterladen und anschließend `tar xzf housing.tgz` zum Entpacken und Extrahieren der CSV-Datei verwenden, es ist aber besser, dazu eine kleine Funktion zu schreiben. Dies ist besonders dann nützlich, wenn sich die Daten regelmäßig ändern, weil Sie so die jeweils neuesten Daten mit einem selbst geschriebenen Skript herunterladen können (Sie könnten dieses automatisch in regelmäßigen Abständen ausführen lassen). Den Prozess der Datenbeschaffung zu automatisieren, hilft außerdem, wenn Sie den Datensatz auf mehreren Maschinen installieren möchten.

Mit der folgenden Funktion können Sie die Daten herunterladen:¹⁰

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Wenn Sie nun `fetch_housing_data()` aufrufen, wird das Verzeichnis *datasets/housing* in Ihrer Arbeitsumgebung erstellt, die Datei *housing.tgz* heruntergeladen und die Datei *housing.csv* in dieses Verzeichnis entpackt.

Nun laden wir die Daten mit Pandas. Auch diesmal sollten Sie eine kleine Funktion zum Laden der Daten schreiben:

⁹ Sie müssten eventuell auch gesetzliche Vorgaben berücksichtigen, z.B. Felder mit persönlichen Daten, die niemals in ungeschützte Datenspeicher kopiert werden dürfen.

¹⁰ In einem echten Projekt würden Sie den Code in einer Python-Datei ablegen. Im Moment können Sie ihn aber auch in Ihr Jupyter Notebook schreiben.

```

import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)

```

Diese Funktion liefert ein Pandas-DataFrame-Objekt mit sämtlichen Daten.

Wirf einen kurzen Blick auf die Datenstruktur

Schauen wir uns die ersten fünf Zeilen des DataFrames mit der Methode head() an (siehe Abbildung 2-5).

The screenshot shows a Jupyter Notebook cell with the following content:

```

In [5]: housing = load_housing_data()
housing.head()

Out[5]:
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  population
0 -122.23    37.88      41.0            880.0          129.0        322.0
1 -122.22    37.86      21.0           7099.0         1106.0       2401.0
2 -122.24    37.85      52.0           1467.0          190.0        496.0
3 -122.25    37.85      52.0           1274.0          235.0        558.0
4 -122.25    37.85      52.0           1627.0          280.0        565.0

```

Abbildung 2-5: Die ersten fünf Zeilen im Datensatz

Jede Zeile steht für einen Bezirk. Es gibt zehn Merkmale (Sie können die ersten sechs im Screenshot sehen): longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value und ocean_proximity.

Die Methode info() hilft, schnell eine Beschreibung der Daten zu erhalten. Diese sind insbesondere die Anzahl Zeilen, der Typ jedes Attributs und die Anzahl Werte ungleich null (siehe Abbildung 2-6).

The screenshot shows a Jupyter Notebook cell with the following content:

```

In [6]: housing.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude          20640 non-null float64
latitude           20640 non-null float64
housing_median_age 20640 non-null float64
total_rooms         20640 non-null float64
total_bedrooms     20433 non-null float64
population         20640 non-null float64
households         20640 non-null float64
median_income       20640 non-null float64
median_house_value 20640 non-null float64
ocean_proximity    20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB

```

Abbildung 2-6: Informationen zu Immobilien

Im Datensatz gibt es 20640 Datenpunkte, damit ist er für Machine-Learning-Verhältnisse eher klein. Für den Anfang ist das ausgezeichnet! Beachten Sie, dass das Merkmal `total_bedrooms` nur 20433 Werte ungleich null hat, es gibt also 207 Bezirke ohne diese Angabe. Darum werden wir uns später kümmern müssen.

Bis auf das Feld `ocean_proximity` sind sämtliche Merkmale numerisch. Dessen Typ ist `object`, und es könnte beliebige Python-Objekte enthalten. Da Sie aber diese Daten aus einer CSV-Datei geladen haben, muss es sich dabei natürlich um Text handeln. Beim Betrachten der ersten fünf Zeilen haben Sie möglicherweise bemerkt, dass sich die Werte in der Spalte `ocean_proximity` wiederholen. Es handelt sich dabei also um ein kategorisches Merkmal. Sie können mit der Methode `value_counts()` herausfinden, welche Kategorien es gibt und wie viele Bezirke zu jeder Kategorie gehören:

```
>>> housing["ocean_proximity"].value_counts()  
<1H OCEAN      9136  
INLAND        6551  
NEAR OCEAN     2658  
NEAR BAY       2290  
ISLAND          5  
Name: ocean_proximity, dtype: int64
```

Betrachten wir auch die anderen Spalten. Die Methode `describe()` fasst die numerischen Merkmale zusammen (Abbildung 2-7).

In [8]:	housing.describe()					
Out[8]:		longitude	latitude	housing_median_age	total_rooms	total_bedrooms
	count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
	mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
	std	2.003532	2.135952	12.585558	2181.615252	421.385070
	min	-124.350000	32.540000	1.000000	2.000000	1.000000
	25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
	50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
	75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
	max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Abbildung 2-7: Zusammenfassung aller numerischen Merkmalen

Die Zeilen `count`, `mean`, `min` und `max` sind selbsterklärend. Beachten Sie, dass die leeren Werte ignoriert werden (z.B. beträgt `count` bei `total_bedrooms` 20433, nicht 20640). Die Zeile `std` enthält die *Standardabweichung*, welche die Streuung der Werte angibt.¹¹ Die Zeilen mit 25 %, 50 % und 75 % zeigen die entsprechenden *Perzentile*: Ein Perzentil besagt, dass ein bestimmter prozentualer Anteil der Beobachtungen unterhalb eines Werts liegt. Beispielsweise haben 25 % der Bezirke ein `housing_median_age` unter 18, 50 % liegen unter 29, und 75 % liegen unter 37. Diese

nennt man oft das 25. Perzentil (oder 1. *Quartil*), den Median und das 75. Perzentil (oder 3. Quartil).

Eine andere Möglichkeit, schnell einen Eindruck von den Daten, die wir sehen, zu erhalten, ist, für jedes numerische Merkmal ein Histogramm zu plotten. Ein Histogramm zeigt die Anzahl Datenpunkte (auf der vertikalen Achse), die in einem bestimmten Wertebereich (auf der horizontalen Achse) liegen. Sie können diese entweder für jedes Merkmal einzeln plotten oder die Methode `hist()` für den gesamten Datensatz aufrufen und für jedes numerische Merkmal ein Histogramm erhalten (siehe Abbildung 2-8). Beispielsweise sehen Sie, dass etwas über 800 Bezirke einen `median_house_value` von etwa 100000 USD besitzen.

```
%matplotlib inline # nur im Jupyter Notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

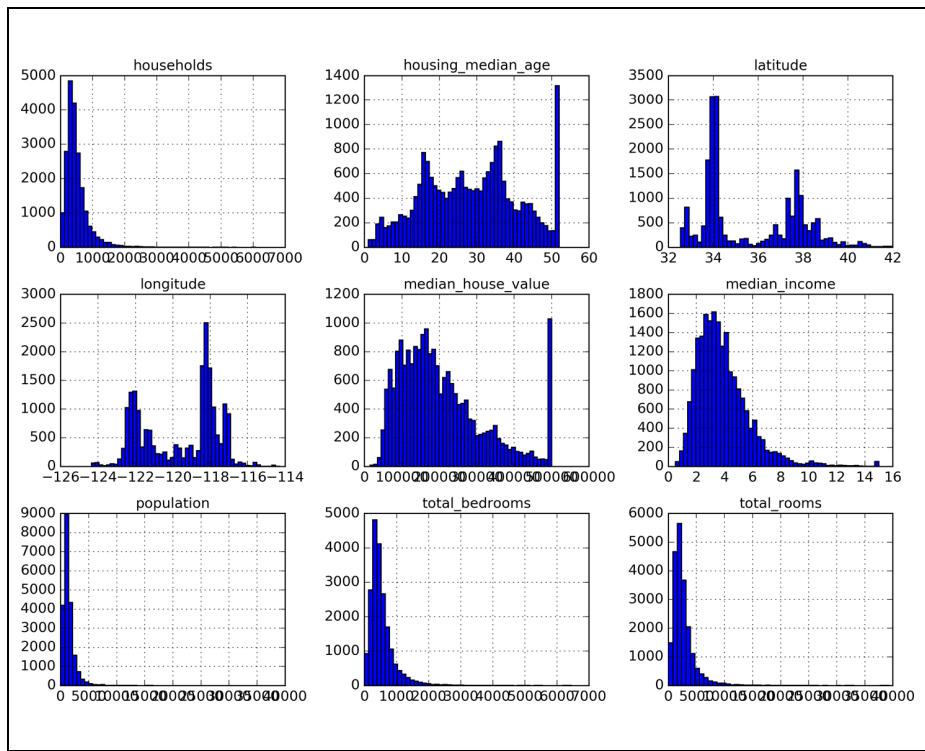


Abbildung 2-8: Ein Histogramm für jedes numerische Attribut

11 Die Standardabweichung wird im Allgemeinen mit σ angegeben (dem griechischen Buchstaben Sigma) und ist die Quadratwurzel der Varianz, der durchschnittlichen quadratischen Abweichung vom Mittelwert. Wenn ein Merkmal der sehr häufigen glockenförmigen *Normalverteilung* folgt (auch *Gaußverteilung* genannt), gilt die »68-95-99.7«-Regel: Etwa 68% der Werte liegen innerhalb von 1σ des Mittelwerts, 95 % innerhalb von 2σ und 99.7% innerhalb von 3σ .



Die Methode `hist()` verwendet Matplotlib, das wiederum ein nutzerabhängiges grafisches Backend zum Zeichnen auf den Bildschirm verwendet. Bevor Sie also etwas plotten können, müssen Sie Matplotlib darüber informieren, welches Backend es verwenden soll. Die einfachste Möglichkeit ist, in Jupyter das magische Kommando `%matplotlib inline` zu verwenden. Dieses weist Jupyter an, Matplotlib zu verwenden, sodass Jupyter als Backend verwendet wird. Die Diagramme werden dann im Notebook selbst dargestellt, wobei Jupyter sie automatisch generiert, sobald eine Zelle ausgeführt wird.

In diesen Histogrammen gibt es einige Dinge zu bemerken:

1. Erstens sieht das mittlere Einkommen nicht nach Werten in US-Dollar aus (USD). Nachdem Sie sich mit dem Team, das die Daten erhoben hat, in Verbindung gesetzt haben, erfahren Sie, dass die Daten skaliert wurden und für höhere mittlere Einkommen nach oben bei 15 (genau 15.0001) und für geringere mittlere Einkommen nach unten bei 0.5 (genau 0.4999) abgeschnitten wurden. Es ist im Machine Learning durchaus üblich, mit solchen vorverarbeiteten Merkmalen zu arbeiten, was nicht notwendigerweise ein Problem darstellt. Sie sollten aber versuchen nachzuvollziehen, wie die Daten berechnet wurden.
2. Das mittlere Alter und der mittlere Wert von Gebäuden wurden ebenfalls gekappt. Letzteres könnte sich als ernstes Problem herausstellen, da Ihre Zielgröße (Ihr Label) betroffen ist. Ihre Machine-Learning-Algorithmen könnten dann lernen, dass es keine Preise jenseits dieser Obergrenze gibt. Sie müssen mit Ihrem Team (dem Team, das die Ausgabe Ihres Systems nutzen möchte) klären, ob dies ein Problem darstellt oder nicht. Wenn Ihnen erklärt wird, dass auch jenseits von 500000 USD präzise Vorhersagen nötig sind, haben Sie zwei Alternativen:
 - a. Für die nach oben begrenzten Bezirke korrekte Labels zu sammeln.
 - b. Die entsprechenden Bezirke aus dem Trainingsdatensatz zu entfernen (auch aus dem Testdatensatz, da Ihr System nicht als schlechter eingestuft werden sollte, wenn es Werte jenseits von 500000 USD vorhersagt).
3. Diese Attribute haben sehr unterschiedliche Wertebereiche. Wir werden dies später im Kapitel besprechen, wenn wir uns dem Skalieren von Merkmalen widmen.
4. Schließlich sind viele der Histogramme *rechtsschief*: Sie erstrecken sich viel weiter vom Median nach rechts als nach links. Dadurch wird das Erkennen von Mustern für einige Machine-Learning-Algorithmen schwieriger. Wir werden später versuchen, diese Merkmale zu einer annähernd glockenförmigen Verteilung zu transformieren.

Hoffentlich haben Sie nun einen besseren Eindruck von den Daten, mit denen Sie sich beschäftigen.



Warten Sie! Bevor Sie sich die Daten weiter ansehen, sollten Sie einen Testdatensatz erstellen, beiseitelegen und nicht hineinschauen.

Erstelle einen Testdatensatz

Es mag sich seltsam anhören, an dieser Stelle einen Teil der Daten freiwillig beiseitezulegen. Schließlich haben wir gerade erst einen kurzen Blick auf die Daten geworfen, und Sie sollten bestimmt noch weiter analysieren, bevor Sie sich für einen Algorithmus entscheiden, oder? Das ist zwar richtig, aber Ihr Gehirn ist ein faszinierendes System zur Mustererkennung. Es ist daher äußerst anfällig für Overfitting: Wenn Sie sich die Testdaten ansehen, könnten Sie auf ein interessantes Muster im Datensatz stoßen, das Sie zur Auswahl eines bestimmten Machine-Learning-Modells veranlasst. Wenn Sie den Fehler der Verallgemeinerung anhand des Testdatensatzes schätzen, wird Ihr Schätzwert zu optimistisch ausfallen, und Sie würden in der Folge ein System starten, das die erwartete Vorhersageleistung nicht erfüllt. Dies nennt man auch den *Data Snooping-Bias*.

Einen Testdatensatz zu erstellen, ist theoretisch sehr einfach: Wählen Sie zufällig einige Datenpunkte aus, meist 20% des Datensatzes, und legen Sie diese beiseite:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Sie können diese Funktion anschließend folgendermaßen verwenden:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "Training +", len(test_set), "Test")
16512 Training + 4128 Test
```

Das funktioniert, ist aber noch nicht perfekt: Wenn Sie dieses Programm erneut ausführen, erzeugt es einen anderen Testdatensatz! Sie (oder Ihre Machine-Learning-Algorithmen) werden mit der Zeit den kompletten Datensatz als Gesamtes sehen, was Sie ja genau vermeiden möchten.

Eine Lösungsmöglichkeit ist, den Testdatensatz beim ersten Durchlauf zu speichern und in späteren Durchläufen zu laden. Eine andere Möglichkeit ist, den Seed-Wert des Zufallsgenerators festzulegen (z.B. mit `np.random.seed(42)`)¹², bevor Sie `np.random.permutation()` aufrufen, sodass jedes Mal die gleichen durchmischten Indizes generiert werden.

¹² Sie werden häufig sehen, dass manche den Seed-Wert auf 42 setzen. Diese Zahl hat keine besondere Bedeutung, außer dass es die Antwort auf die ultimative Frage nach dem Leben, dem Universum und dem ganzen Rest ist.

Allerdings scheitern beide Lösungsansätze, sobald Sie einen aktualisierten Datensatz erhalten. Als Alternative können Sie einen eindeutigen Identifikator verwenden, um zu entscheiden, ob ein Datenpunkt in den Testdatensatz aufgenommen werden soll oder nicht (vorausgesetzt, die Datenpunkte haben eindeutige unveränderliche Identifikatoren). Sie könnten beispielsweise aus dem Identifikator eines Datenpunkts einen Hash berechnen, nur das letzte Byte des Hashes zu verwenden und den Datenpunkt in den Testdatensatz aufzunehmen, falls dieser Wert kleiner oder gleich 51 ist (circa 20% von 256). Damit stellen Sie sicher, dass der Testdatensatz über mehrere Durchläufe konsistent ist, selbst wenn Sie ihn aktualisieren. Ein neuer Datensatz enthält auf diese Weise 20% der neuen Datenpunkte, aber keinen der Datenpunkte, die zuvor im Trainingsdatensatz waren. Hier folgt eine mögliche Implementierung:

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Leider gibt es im Immobiliendatensatz keine Identifikator-Spalte. Die Lösung ist, den Zeilenindex als ID zu verwenden:

```
housing_with_id = housing.reset_index() # fügt die Spalte `index` hinzu
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

Wenn Sie den Zeilenindex als eindeutigen Identifikator verwenden, müssen Sie sicherstellen, dass die neuen Daten am Ende des Datensatzes angehängt werden und nie eine Zeile gelöscht wird. Falls dies nicht möglich ist, können Sie immer noch versuchen, einen eindeutigen Identifikator aus den stabilsten Merkmalen zu entwickeln. Beispielsweise werden die geografische Länge und Breite garantiert für die nächsten paar Millionen Jahre stabil bleiben, daher könnten Sie diese folgendermaßen zu einer ID kombinieren:¹³

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn enthält einige Funktionen, die Datensätze auf unterschiedliche Weise in Teildatensätze aufteilen. Die einfachste Funktion darunter ist `train_test_split`, die so ziemlich das Gleiche tut wie die oben definierte Funktion `split_train_test`, aber einige zusätzliche Optionen bietet. Erstens gibt es den Parameter `random_`

¹³ Die Koordinatenangaben sind recht grob, daher werden viele Bezirke eine identische ID erhalten und somit im gleichen Teildatensatz landen (Test oder Training). Damit haben wir unglücklicherweise ein Bias in der Auswahl.

state, der den Seed-Wert des Zufallszahlengenerators wie oben erklärt festlegt. Zweitens können Sie mehrere Datensätze mit einer identischen Anzahl Zeilen übergeben, diese werden anhand der gleichen Indizes aufgeteilt (dies ist sehr nützlich, z.B., wenn Sie ein separates DataFrame mit den Labels haben):

```
from sklearn.model_selection import train_test_split  
  
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Bisher haben wir ausschließlich zufallsbasierte Methoden zur Stichprobenauswahl betrachtet. Wenn Ihr Datensatz groß genug ist (insbesondere im Vergleich zur Anzahl der Merkmale), ist daran nichts auszusetzen. Ist er aber nicht groß genug, besteht das Risiko, ein erhebliches Stichproben-Bias zu verursachen. Wenn ein Umfrageunternehmen 1000 Personen anruft, um diesen Fragen zu stellen, wählt es nicht einfach nur zufällig 1000 Probanden aus dem Telefonbuch aus. Beispielsweise besteht die Bevölkerung der USA aus 51.3% Frauen und 48.7% Männern, also sollte eine gut aufgebaute Studie in den USA dieses Verhältnis auch in der Stichprobe repräsentieren: 513 Frauen und 487 Männer. Dies bezeichnet man als *stratifizierte Stichprobe*: Die Bevölkerung wird in homogene Untergruppen, die *Strata*, aufgeteilt, und aus jedem Stratum wird die korrekte Anzahl Datenpunkte ausgewählt. Damit ist garantiert, dass der Testdatensatz die Gesamtbevölkerung angemessen repräsentiert. Würde die Stichprobe rein zufällig ausgewählt, gäbe es eine 12%ige Chance, dass die Stichprobe verzerrt ist und entweder weniger als 49% Frauen oder mehr als 54% Frauen im Datensatz enthalten sind. In beiden Fällen wären die Ergebnisse mit einem erheblichen Bias behaftet.

Nehmen wir an, Experten hätten Ihnen in einer Unterhaltung erklärt, dass das mittlere Einkommen ein sehr wichtiges Merkmal zur Vorhersage des mittleren Immobilienpreises ist. Sie möchten sicherstellen, dass der Testdatensatz die unterschiedlichen im Datensatz enthaltenen Einkommensklassen gut repräsentiert. Da das mittlere Einkommen ein stetiges numerisches Merkmal ist, müssen Sie zuerst ein kategorisches Merkmal für das Einkommen generieren. Betrachten wir das Histogramm des Einkommens etwas genauer (siehe Abbildung 2-8): Die meisten mittleren Einkommen liegen bei 20000 – 50000 USD, aber einige mittlere Einkommen liegen deutlich über 60000 USD. Es ist wichtig, dass Ihr Datensatz für jedes Stratum eine genügende Anzahl Datenpunkte enthält, andernfalls liegt ein Bias für die Schätzung der Wichtigkeit des Stratums vor. Das heißt, Sie sollten nicht zu viele Strata haben, und jedes Stratum sollte groß genug sein. Der folgende Code erstellt ein kategorisches Merkmal für das Einkommen, indem er das mittlere Einkommen durch 1.5 teilt (um die Anzahl der Einkommenskategorien zu begrenzen) und mit ceil aufrundet (um diskrete Kategorien zu erhalten). Anschließend werden sämtliche Kategorien über 5 mit Kategorie 5 vereinigt:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)  
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

Diese Einkommenskategorien sind in Abbildung 2-9) dargestellt:

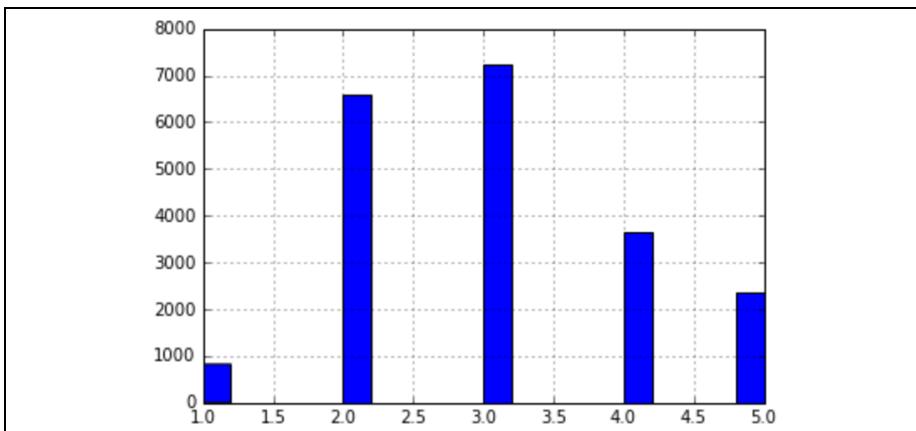


Abbildung 2-9: Histogramm der Einkommenskategorien

Nun sind wir so weit, eine stratifizierte Stichprobe anhand der Einkommenskategorie zu ziehen. Dazu können Sie die Klasse `StratifiedShuffleSplit` aus Scikit-Learn verwenden:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Prüfen wir, ob dies wie erwartet funktioniert hat. Zunächst einmal können Sie sich die Anteile der Einkommenskategorien im gesamten Immobiliendatensatz ansehen:

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

Mit einer ähnlichen Codezeile lassen sich die Anteile der Einkommenskategorien im Testdatensatz bestimmen. In Abbildung 2-10 werden die Anteile der Einkommenskategorien im gesamten Datensatz, im als stratifizierte Stichprobe generierten Testdatensatz und in einem rein zufälligen Testdatensatz miteinander verglichen. Wie Sie sehen, sind die Anteile der Einkommenskategorien in der stratifizierten Stichprobe beinahe die gleichen wie im Gesamt datensatz, während der als zufällige Stichprobe erzeugte Testdatensatz recht verzerrt ist.

Nun sollten Sie das Merkmal `income_cat` entfernen, damit die Daten wieder in ihrem Ursprungszustand sind:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Wir haben uns aus gutem Grund eine Menge Zeit für das Erstellen des Testdatensatzes genommen: Ist es doch ein oft vernachlässigter, aber entscheidender Teil eines Machine-Learning-Projekts. Viele der hier vorgestellten Ideen werden noch nützlich sein, sobald wir die Kreuzvalidierung besprechen. Nun ist es an der Zeit, mit dem nächsten Abschnitt fortzufahren: dem Erkunden der Daten.

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

Abbildung 2-10: Vergleich des Stichproben-Bias einer stratifizierten und einer zufälligen Stichprobe

Erkunde und visualisiere die Daten, um Erkenntnisse zu gewinnen

Bisher haben wir nur einen kurzen Blick auf die Daten geworfen, um ein allgemeines Verständnis von der Art der verarbeiteten Daten zu erhalten. Nun ist unser Ziel, etwas mehr in die Tiefe zu gehen.

Zunächst sollten Sie sicherstellen, dass Sie den Testdatensatz beiseitegelegt haben und nur noch den Trainingsdatensatz erkunden. Wenn Ihr Trainingsdatensatz sehr groß ist, sollten Sie eine Stichprobe ziehen, um sich die Arbeit beim Erkunden zu erleichtern und sie zu beschleunigen. In unserem Fall ist der Datensatz recht klein, sodass Sie direkt mit den vollständigen Daten arbeiten können. Erzeugen wir eine Kopie, mit der Sie experimentieren können, ohne den Trainingsdatensatz zu beschädigen:

```
housing = strat_train_set.copy()
```

Visualisieren geografischer Daten

Weil uns geografische Informationen zur Verfügung stehen (Breite und Länge), sollten wir einen Scatterplot sämtlicher Bezirke erzeugen, um uns die Daten anzusehen (Abbildung 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

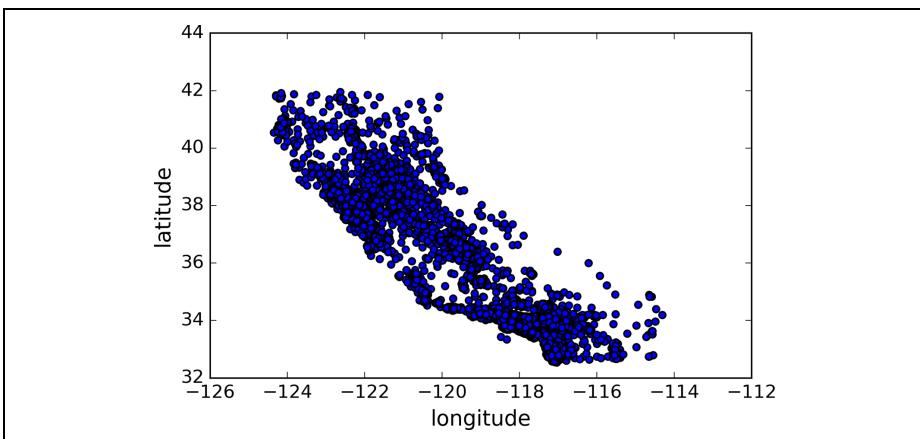


Abbildung 2-11: Ein geografischer Scatterplot der Daten

In Ordnung, die Abbildung sieht wie Kalifornien aus, aber abgesehen davon ist es schwierig, irgendein Muster zu erkennen. Setzen wir den Parameter alpha auf 0.1, wird es viel einfacher, die Orte mit einer hohen Dichte an Datenpunkten zu erkennen (Abbildung 2-12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

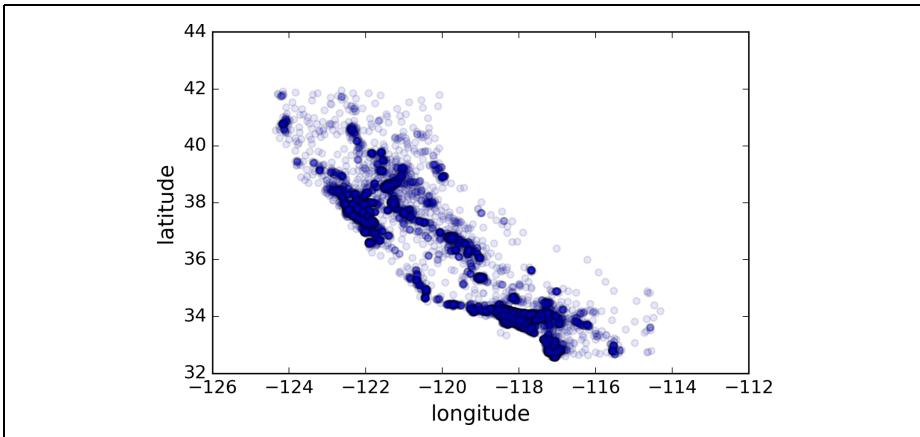


Abbildung 2-12: Eine bessere Darstellung von Gebieten mit hoher Dichte

Das sieht schon viel besser aus: Sie können die Ballungszentren deutlich erkennen, genauer gesagt die Bay Area, den Großraum Los Angeles und San Diego sowie eine lange Reihe einigermaßen hoher Dichte im Central Valley, insbesondere um Sacramento und Fresno.

Im Allgemeinen ist unser Gehirn sehr gut darin, Muster in Bildern zu erkennen. Sie müssen jedoch mit den Parametern zur Visualisierung experimentieren, damit diese Muster deutlich hervortreten.

Betrachten wir nun die Immobilienpreise (Abbildung 2-13). Der Radius jedes Kreises stellt die Bevölkerung eines Bezirks dar (Option s), und die Farbe stellt den Preis dar (Option cmap) namens jet, die von Blau (niedrige Werte) nach Rot (hohe Werte) reicht.¹⁴

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

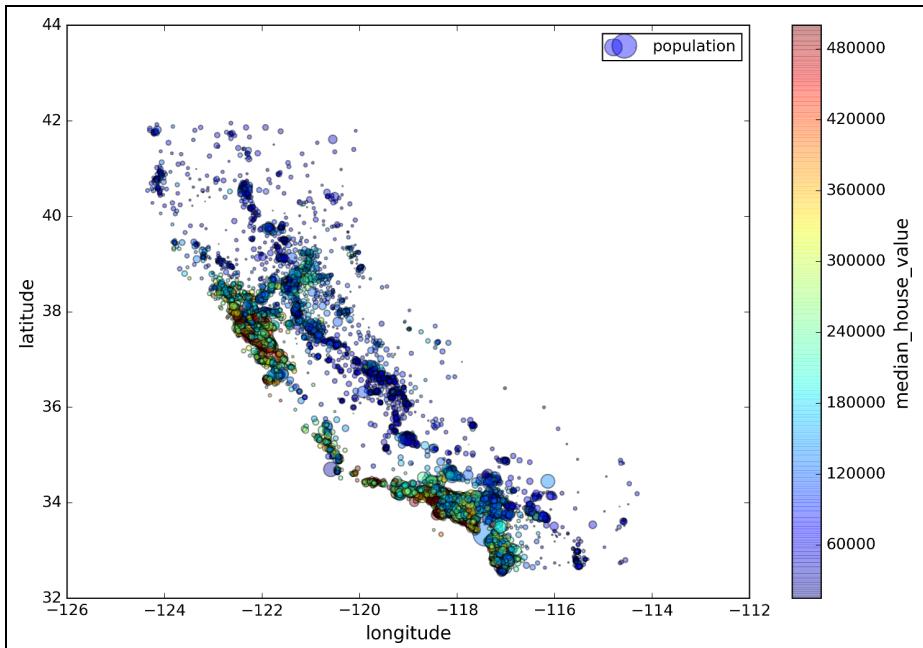


Abbildung 2-13: Immobilienpreise in Kalifornien

In diesem Bild können Sie erkennen, dass die Immobilienpreise stark mit dem Ort (z.B. der Nähe zum Ozean) und der Bevölkerungsdichte zusammenhängen, was Sie vermutlich bereits wussten. Es könnte hilfreich sein, die wichtigsten Cluster mit einem Clustering-Algorithmus zu identifizieren und einige neue Merkmale mit der Entfernung zu den Cluster-Mittelpunkten hinzuzufügen. Die Nähe zum Ozean sollte als Merkmal ebenfalls nützlich sein, obwohl die Immobilienpreise in Nordkalifornien in Küstennähe nicht außerordentlich hoch sind, daher ist dies keine grundsätzliche Regel.

¹⁴ Wenn Sie dieses Buch in Graustufen lesen, schnappen Sie sich einen roten Stift und malen einen Großteil der Küstenlinie von der Bay Area bis nach San Diego aus (wie man erwarten würde). Sie können um Sacramento herum auch ein wenig Gelb hinzufügen.

Suche nach Korrelationen

Da Ihr Datensatz nicht besonders groß ist, können Sie mit der Methode `corr()` den *Korrelationskoeffizienten* (auch *Pearson-Korrelationskoeffizient*) für jedes Merkmalspaar leicht berechnen:

```
corr_matrix = housing.corr()
```

Schauen wir uns einmal an, wie stark jedes Merkmal mit dem mittleren Immobilienwert korreliert:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms          0.135231
housing_median_age   0.114220
households           0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

Der Korrelationskoeffizient liegt zwischen -1 und 1 . Liegt er nahe bei 1 , haben wir eine stark positive Korrelation; z.B. steigt der mittlere Immobilienwert tendenziell mit dem mittleren Einkommen. Ist der Koeffizient nahe bei -1 , liegt eine stark negative Korrelation vor; Sie können eine geringe negative Korrelation zwischen der geografischen Breite und dem mittleren Immobilienwert beobachten (d.h., der Preis sinkt tendenziell ein wenig, wenn Sie sich nach Norden bewegen). Schließlich bedeuten Koeffizienten um null, dass es keine lineare Korrelation gibt. Abbildung 2-14 zeigt unterschiedliche Diagramme und den Korrelationskoeffizienten zwischen der horizontalen und vertikalen Achse.

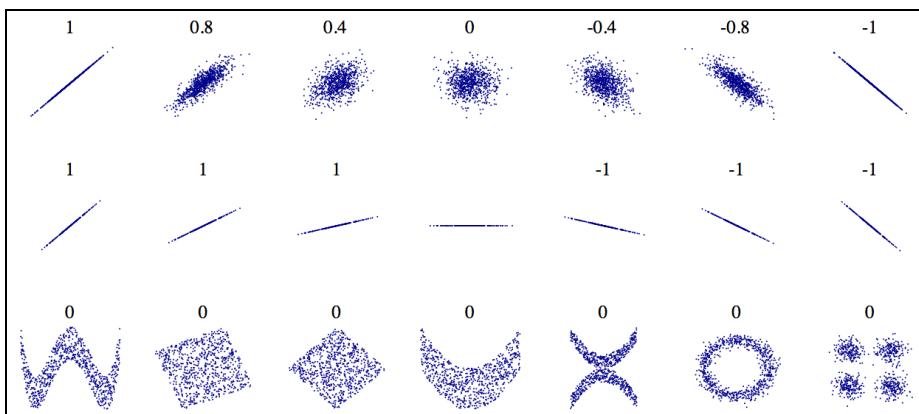


Abbildung 2-14: Korrelationskoeffizienten unterschiedlicher Datensätze
(Quelle: Wikipedia; public domain image)



Der Korrelationskoeffizient erfasst ausschließlich lineare Korrelationen (»wenn x steigt, steigt/sinkt y im Allgemeinen«). Damit können Sie nichtlineare Zusammenhänge vollständig übersehen (z.B. »wenn x nahe null ist, steigt y «). Beachten Sie, dass sämtliche Diagramme in der unteren Reihe einen Korrelationskoeffizienten von null haben, obwohl ihre Achsen ganz klar nicht unabhängig voneinander sind: Dies sind Beispiele für nichtlineare Zusammenhänge. Auch die zweite Reihe zeigt Beispiele, bei denen der Korrelationskoeffizient 1 oder -1 beträgt; beachten Sie, dass dies nichts mit der Steigung zu tun hat. Ihre Körpergröße in Zoll hat beispielsweise sowohl einen Korrelationskoeffizienten von 1 mit Ihrer Körpergröße in Fuß als auch in Nanometern.

Eine andere Möglichkeit, nach Korrelationen zwischen Attributen zu suchen, ist die in Pandas eingebaute Funktion `scatter_matrix`, die jedes numerische Merkmal gegen jedes andere aufträgt. Da es 11 numerische Merkmale gibt, würden Sie $11^2 = 121$ Diagramme erhalten. Da diese nicht auf eine Seite passen, konzentrieren wir uns auf einige vielversprechende Merkmale, die am ehesten mit dem mittleren Immobilienpreis korrelieren (Abbildung 2-15):

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
               "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

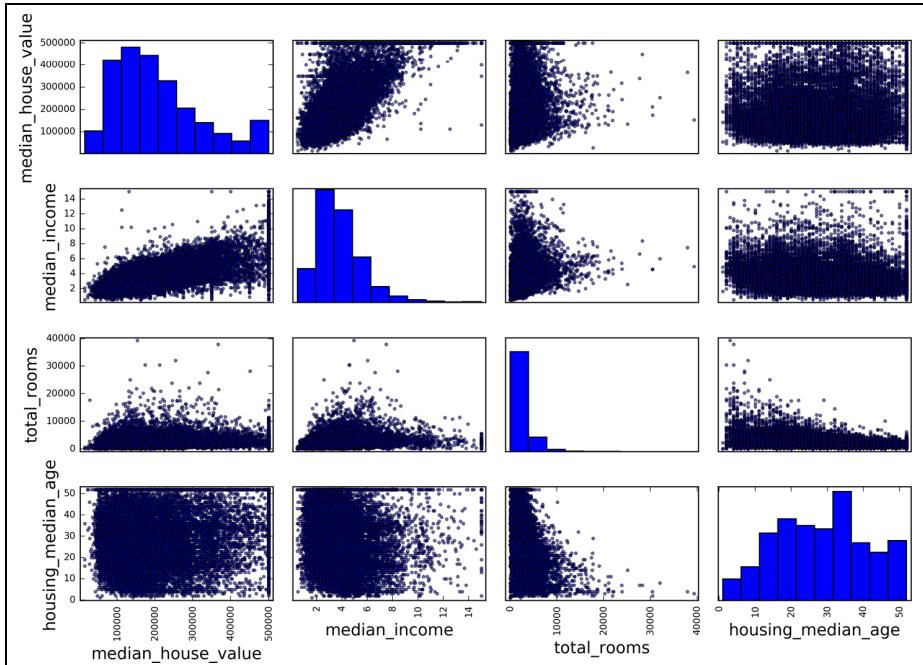


Abbildung 2-15: Scatterplot-Matrix

Die Hauptdiagonale (von oben links nach unten rechts) würde mit lauter geraden Linien gefüllt sein, wenn Pandas jedes Merkmal gegen sich selbst plotten würde. Da dies nicht besonders nützlich wäre, stellt Pandas stattdessen ein Histogramm jedes Merkmals dar (es gibt dazu einige Alternativen; Details finden Sie in der Dokumentation von Pandas).

Das am meisten Erfolg versprechende Merkmal zur Vorhersage des mittleren Immobilienwerts ist das mittlere Einkommen. Daher zoomen wir in den entsprechenden Scatterplot hinein (Abbildung 2-16):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",  
alpha=0.1)
```

Dieses Diagramm verdeutlicht einige Dinge. Erstens ist die Korrelation in der Tat recht stark; Sie können den Trend nach oben klar sehen, und die Punkte sind nicht allzu verstreut. Zweitens erkennen wir die weiter oben beobachtete obere Preisbegrenzung deutlich als horizontale Linie bei 500000 USD. Das Diagramm zeigt aber auch weniger offensichtliche gerade Linien: eine horizontale Linie bei etwa 450000 USD und eine zweite bei 350000 USD, eventuell auch eine bei 280000 USD und darunter noch weitere. Möglicherweise sollten Sie versuchen, die entsprechenden Bezirke aus dem Datensatz zu entfernen, um zu verhindern, dass Ihre Algorithmen diese Artefakte im Datensatz reproduzieren.

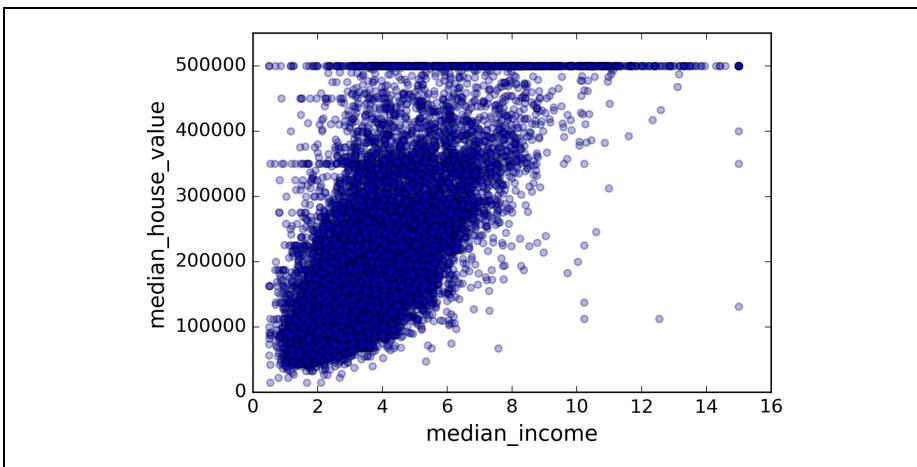


Abbildung 2-16: Mittleres Einkommen gegen mittleren Immobilienwert

Experimentieren mit Kombinationen von Merkmalen

Hoffentlich hat der vorige Abschnitt Ihnen einige Ideen geliefert, mit denen Sie die Daten erkunden und Erkenntnisse gewinnen können. Sie haben einige Artefakte identifiziert, die Sie eventuell entfernen sollten, bevor Sie die Daten in einen Machine-Learning-Algorithmus einspeisen. Sie haben auch einige interessante Kor-

relationen entdeckt, insbesondere mit der Zielgröße. Sie haben außerdem bemerkt, dass einige Merkmale eine rechtsschiefe Verteilung aufweisen. Daher kann es nötig sein, diese zu transformieren (z.B. durch Logarithmieren). Natürlich unterscheidet sich der nötige Aufwand von Projekt zu Projekt beträchtlich, aber die Grundideen sind die gleichen.

Bevor Sie die Daten für Machine-Learning-Algorithmen vorbereiten, sollten wir als letzten Punkt noch einige Kombinationen von Merkmalen ausprobieren. Beispielsweise ist die Anzahl der Räume in einem Distrikt nicht besonders aussagekräftig, wenn Sie nicht wissen, wie viele Haushalte es dort gibt. Was Sie wirklich benötigen, ist die Anzahl Räume pro Haushalt. In ähnlicher Weise ist die Gesamtzahl der Schlafzimmer nicht sehr nützlich: Sie sollten diese mit der Anzahl der Zimmer vergleichen. Auch die Bewohner pro Haushalt könnten eine interessante Kombination von Merkmalen hergeben. Erstellen wir diese neuen Merkmale:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

Nun betrachten wir die Korrelationsmatrix erneut:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income                0.687160
rooms_per_household         0.146285
total_rooms                  0.135097
housing_median_age          0.114110
households                   0.064506
total_bedrooms               0.047689
population_per_household   -0.021985
population                   -0.026920
longitude                     -0.047432
latitude                      -0.142724
bedrooms_per_room            -0.259984
Name: median_house_value, dtype: float64
```

Nicht schlecht! Das neue Merkmal `bedrooms_per_room` korreliert wesentlich stärker mit dem mittleren Immobilienwert als die Anzahl der Zimmer oder Schlafzimmer. Anscheinend sind Häuser mit einem niedrigeren Verhältnis von Schlafzimmern zu Zimmern teurer. Die Anzahl Räume pro Haushalt ist ebenfalls aufschlussreicher als die Gesamtzahl Räume in einem Bezirk – natürlich sind Häuser um so teurer, je größer sie sind.

Dieser Teil der Untersuchung muss nicht extrem gründlich sein; es geht darum, an der richtigen Stelle zu beginnen und schnell einige Erkenntnisse zu sammeln, die für einen halbwegs guten Prototyp ausreichen. Der Prozess ist insgesamt iterativ: Sobald Ihr Prototyp läuft, können Sie dessen Ausgabe untersuchen, weitere Erkenntnisse daraus ziehen und zur Erkundung zurückkehren.

Bereite die Daten für Machine-Learning-Algorithmen vor

Es ist an der Zeit, die Daten für Ihre Machine-Learning-Algorithmen vorzubereiten. Es gibt mehrere gute Gründe, dafür Funktionen zu schreiben, anstatt es manuell zu probieren:

- Sie können die entsprechenden Transformationen leicht auf beliebigen Datensätzen reproduzieren (z.B. auf dem nächsten frischen Datensatz, den Sie bekommen).
- Sie bauen nach und nach eine Bibliothek von Transformationsfunktionen auf, die Sie in zukünftigen Projekten nutzen können.
- Sie können diese Funktionen in Ihrer Produktionsumgebung nutzen, um neue Daten vor der Eingabe in Ihre Algorithmen zu transformieren.
- Dadurch können Sie unterschiedliche Transformationen leichter ausprobieren und prüfen, welche Kombinationen am besten funktionieren.

Aber kehren wir zunächst zu einem sauberen Trainingsdatensatz zurück (indem wir `strat_train_set` noch einmal kopieren) und trennen die Merkmale und Labels voneinander. Wir möchten nämlich nicht unbedingt die gleichen Transformationen auf die Merkmale zur Vorhersage und die Zielwerte anwenden (beachten Sie, dass `drop()` eine Kopie der Daten erzeugt und `strat_train_set` nicht verändert):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Aufbereiten der Daten

Die meisten Machine-Learning-Algorithmen können mit fehlenden Merkmalen nicht umgehen. Deshalb werden wir einige Funktionen schreiben, die sich darum kümmern. Sie haben bereits bemerkt, dass beim Attribut `total_bedrooms` einige Werte fehlen. Um diesen Umstand zu beheben, haben Sie drei Möglichkeiten:

- die entsprechenden Bezirke entfernen
- das Merkmal komplett verwerfen
- die Werte auf einen bestimmten Wert setzen (null, den Median o. Ä.)

Sie können alle drei leicht mit den Methoden `dropna()`, `drop()` und `fillna()` eines DataFrames umsetzen:

```
housing.dropna(subset=["total_bedrooms"])      # Option 1
housing.drop("total_bedrooms", axis=1)          # Option 2
median = housing["total_bedrooms"].median()    # Option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Wenn Sie sich für die dritte Möglichkeit entscheiden, sollten Sie den Median des Trainingsdatensatzes berechnen und die fehlenden Werte mit diesem auffüllen. Sie sollten aber auch daran denken, den berechneten Wert zu sichern. Sie werden ihn

später benötigen, um fehlende Werte im Testdatensatz zu ersetzen, wenn Sie Ihr System evaluieren, und fehlende Werte in neuen Daten zu ersetzen, sobald Ihr System in Betrieb geht.

Scikit-Learn enthält eine nützliche Klasse zum Umgang mit fehlenden Werten: `Imputer`. Sie verwenden diese, indem Sie zuerst eine Instanz von `Imputer` erzeugen und angeben, dass Sie die fehlenden Werte jedes Merkmals mit dessen Median ersetzen möchten:

```
from sklearn.preprocessing import Imputer  
  
imputer = Imputer(strategy="median")
```

Da der Median sich nur bei numerischen Merkmalen berechnen lässt, müssen wir eine Kopie der Daten unter Ausschluss des Text-Merkmals `ocean_proximity` erzeugen:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Anschließend können Sie die `imputer`-Instanz durch Aufrufen der Methode `fit()` an die Trainingsdaten anpassen:

```
imputer.fit(housing_num)
```

Der `imputer` hat einfach den Median jedes Merkmals berechnet und das Ergebnis in Attribut `statistics_` gespeichert. Lediglich beim Merkmal `total_bedrooms` gab es fehlende Daten, aber wir können uns nicht sicher sein, dass die neuen Daten im Betrieb nicht ebenfalls lückenhaft sind. Daher ist es sicherer, den `imputer` auf sämtliche numerischen Merkmale anzuwenden:

```
>>> imputer.statistics_  
array([-118.51, 34.26, 29., 2119.5, 433., 1164., 408., 3.5409])  
>>> housing_num.median().values  
array([-118.51, 34.26, 29., 2119.5, 433., 1164., 408., 3.5409])
```

Sie können nun mit diesem »trainierten« `imputer` den Trainingsdatensatz transformieren, sodass die fehlenden Werte durch die gefundenen Mediane ersetzt werden:

```
X = imputer.transform(housing_num)
```

Das Ergebnis ist ein NumPy-Array mit den transformierten Merkmalen. Dieses wieder in ein Pandas-DataFrame zu überführen, ist ebenfalls einfach:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

Das Design von Scikit-Learn

Das Design der API von Scikit-Learn ist bemerkenswert gut gelungen. Die wichtigsten Designprinzipien (<http://goo.gl/wL10sI>) sind:¹⁵

¹⁵ Details zu den Designprinzipien finden Sie in »API design for machine learning software: experiences from the scikit-learn project«, L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Müller, et al. (2013).

- **Konsistenz.** Alle Objekte besitzen eine konsistente, einfache Schnittstelle:
 - *Estimatoren*. Jedes Objekt, das Parameter anhand eines Datensatzes abschätzen kann, wird *Estimator* genannt (z.B. ist ein Imputer ein Estimator). Das Abschätzen der Parameter wird von der Methode `fit()` durchgeführt, die als Parameter lediglich einen Datensatz benötigt (zwei bei überwachten Lernalgorithmen; der zweite Datensatz enthält in diesem Falle die Labels). Jeder andere für das Abschätzen benötigte Parameter wird als Hyperparameter angesehen (wie `strategy` beim Imputer) und muss als Attribut der Instanz gesetzt werden (für gewöhnlich als Parameter im Konstruktor).
 - *Transformer*. Einige Estimatoren (wie der Imputer) können einen Datensatz außerdem transformieren; diese werden *Transformatoren* genannt. Wieder einmal ist der Aufbau der API recht einfach: Die Methode `transform()` nimmt die Transformation selbst vor, der Datensatz wird als Parameter übergeben. Zurückgegeben wird der transformierte Datensatz. Diese Transformation beruht im Allgemeinen auf den erlernten Parametern, wie es auch beim Imputer der Fall ist. Sämtliche Transformer besitzen außerdem die bequemere Methode `fit_transform()`, die den aufeinanderfolgenden Aufrufen von `fit()` und `transform()` entspricht (manchmal ist `fit_transform()` aber auf höhere Geschwindigkeit optimiert).
 - *Prädiktoren*. Schließlich sind einige Estimatoren in der Lage, auf einem gegebenen Datensatz Vorhersagen zu treffen; diese werden als *Prädiktoren* bezeichnet. Beispielsweise ist das Modell `LinearRegression` aus dem vorigen Kapitel ein Prädiktor: Es sagt die Zufriedenheit aus dem Pro-Kopf-BIP eines Lands vorher. Jeder Prädiktor besitzt die Methode `predict()`, die einen Datensatz mit neuen Datenpunkten annimmt und einen Satz entsprechender Vorhersagen zurückgibt. Jeder Prädiktor besitzt auch die Methode `score()` zum Bestimmen der Vorhersagequalität anhand eines Testdatensatzes (und bei überwachten Lernalgorithmen auch der entsprechenden Labels).¹⁶
- **Inspektion.** Sämtliche Hyperparameter eines Estimators sind direkt als öffentliche Attribute der Instanz abrufbar (z.B. `imputer.strategy`), auch die von einem Estimator erlernten Parameter sind über ein öffentliches Attribut mit einem Unterstrich als Suffix verfügbar (z.B. `imputer.statistics_`).
- **Nicht-proliferierende Klassen.** Datensätze werden in NumPy-Arrays oder Sparse Matrices aus SciPy anstatt in eigenen Klassen abgelegt. Die Hyperparameter sind gewöhnliche Python-Strings oder Zahlen.
- **Komposition.** Existierende Komponenten werden soweit wie möglich wieder verwendet. Beispielsweise können Sie eine Pipeline als beliebige Abfolge von Transformatoren und einem Estimator am Ende definieren und als eigenen Estimator speichern, wie wir noch sehen werden.

¹⁶ Einige Prädiktoren besitzen außerdem Methoden, um die Konfidenz ihrer Vorhersagen zu bestimmen.

- **Sinnvolle Standardwerte.** Die Standardwerte der meisten Parameter in Scikit-Learn sind sinnvoll ausgewählt, sodass Sie schnell ein lauffähiges Grundsystem erstellen können.

Bearbeiten von Text und kategorischen Merkmalen

Wir hatten weiter oben das kategorische Merkmal `ocean_proximity` ausgelassen, weil wir für den enthaltenen Text keinen Median berechnen können:

```
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat.head(10)
17606      <1H OCEAN
18632      <1H OCEAN
14650    NEAR OCEAN
3230          INLAND
3555      <1H OCEAN
19480          INLAND
8879      <1H OCEAN
13685          INLAND
4973      <1H OCEAN
4861      <1H OCEAN
Name: ocean_proximity, dtype: object
```

Die meisten Machine-Learning-Algorithmen bevorzugen ohnehin Zahlen, daher werden wir diese Kategorien von Text zu Zahlen konvertieren. Dazu verwenden wir die Methode `factorize()` aus Pandas, die jeder Kategorie einer anderen Zahl zuordnen:

```
>>> housing_cat_encoded, housing_categories = housing_cat.factorize()
>>> housing_cat_encoded[:10]
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])
```

Nun sieht es besser aus. `housing_cat_encoded` ist nun rein numerisch. Die Methode `factorize()` liefert außerdem eine Liste der Kategorien (»<1H OCEAN« ist der 0 zugeordnet, »NEAR OCEAN« gehört zur 1 und so weiter):

```
>>> housing_categories
Index(['<1H OCEAN', 'NEAR OCEAN', 'INLAND', 'NEAR BAY', 'ISLAND'], dtype='object')
```

Schwierig bei dieser Art der Darstellung ist, dass manche ML-Algorithmen davon ausgehen, zwei benachbarte Werte seien ähnlicher zueinander als zwei weiter entfernte. Dies ist natürlich nicht der Fall (beispielsweise sind die Kategorien 0 und 4 einander ähnlicher als die Kategorien 0 und 1). Um dieses Problem zu beheben, ist es üblich, ein binäres Merkmal pro Kategorie zu erstellen: Ein Merkmal beträgt 1, wenn die Kategorie »<1H OCEAN« ist (und andernfalls 0), ein weiteres Merkmal beträgt 1, wenn die Kategorie »INLAND« ist (und andernfalls 0) und so weiter. Dies nennt man *One-Hot-Codierung*, weil nur jeweils ein Merkmal 1 beträgt (heiß) und die anderen 0 betragen (kalt).

Scikit-Learn stellt uns den OneHotEncoder zur Verfügung, um kategorische Werte von Integerzahlen in One-Hot-Vektoren zu überführen. Codieren wir also unsere Werte als One-Hot-Vektoren:

```
>>> from sklearn.preprocessing import OneHotEncoder  
>>> encoder = OneHotEncoder()  
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))  
>>> housing_cat_1hot  
<16512x5 sparse matrix of type '<class 'numpy.float64'>'  
with 16512 stored elements in Compressed Sparse Row format>
```

Beachten Sie, dass fit_transform() ein 2-D-Array erwartet, aber housing_cat_encoded nur ein eindimensionales Array ist. Daher müssen wir es zuerst umformen¹⁷. Beachten Sie außerdem, dass die Ausgabe kein NumPy-Array, sondern eine Sparse-Matrix aus SciPy ist. Dies ist sehr hilfreich, wenn Sie kategorische Merkmale mit mehreren Tausend Kategorien haben. Nach der One-Hot-Codierung erhalten wir eine Matrix mit Tausenden Spalten, und die gesamte Matrix ist voller Nullen, bis auf eine 1 pro Zeile. Alle Nullen zu speichern, wäre extreme Speicherverschwendungen. Die Sparse-Matrix speichert daher nur die Stellen der Elemente ungleich null. Sie können sie mehr oder weniger wie ein gewöhnliches 2-D-Array verwenden,¹⁸ aber wenn Sie sie wirklich in ein (dichtes) NumPy-Array umwandeln möchten, können Sie die Methode toarray() aufrufen:

```
>>> housing_cat_1hot.toarray()  
array([[ 1.,  0.,  0.,  0.,  0.],  
       [ 1.,  0.,  0.,  0.,  0.],  
       [ 0.,  1.,  0.,  0.,  0.],  
       ...,  
       [ 0.,  0.,  1.,  0.,  0.],  
       [ 1.,  0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  1.,  0.]])
```

Beide Transformationen (von textbasierten Kategorien zu Integer-Kategorien, anschließend von Integer-Kategorien zu One-Hot-Vektoren) lassen sich in einem Schritt mit der Klasse CategoricalEncoder durchführen. Diese ist kein Teil von Scikit-Learn 0.19.0 und früher, aber wird in Kürze hinzugefügt. Sie sollte also verfügbar sein, sobald Sie diese Zeilen lesen. Falls nicht, können Sie sich den Code aus dem Jupyter-Notebook für dieses Kapitel abholen (der Code wurde aus PullRequest #9151 kopiert). So lässt er sich verwenden:

```
>>> from sklearn.preprocessing import CategoricalEncoder # or get from notebook  
>>> cat_encoder = CategoricalEncoder()  
>>> housing_cat_reshaped = housing_cat.values.reshape(-1, 1)  
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)  
>>> housing_cat_1hot  
<16512x5 sparse matrix of type '<class 'numpy.float64'>'  
with 16512 stored elements in Compressed Sparse Row format>
```

¹⁷ Die Funktion reshape() in NumPy ermöglicht, dass eine Dimension -1 beträgt, was für »nicht festgelegt« steht: Der Wert ergibt sich aus der Länge des Arrays und den übrigen Dimensionen.

¹⁸ Details finden Sie in der Dokumentation von SciPy.

Standardmäßig gibt der CategoricalEncoder eine Sparse-Matrix aus, Sie können die Kodierung aber auf "onehot-dense" setzen, wenn Sie eine dichtere Matrix bevorzugen:

```
>>> cat_encoder = CategoricalEncoder(encoding="onehot-dense")
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
>>> housing_cat_1hot
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Sie können die Liste der Kategorien über die Instanzvariable des Encoders categories_einsehen. Sie ist eine Liste mit einem 1-D-Array von Kategorien für jedes kategorische Merkmal (in diesem Fall eine Liste mit einem einzelnen Array, da es nur ein kategorisches Merkmal gibt):

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)]
```



Wenn ein kategorisches Merkmal eine große Anzahl möglicher Kategorien aufweist (z.B. Ländercode, Beruf, Spezies und so weiter), führt die One-Hot-Kodierung zu einer großen Zahl Eingabemerkmale. Dies kann das Trainieren verlangsamen und die Leistung verringern. In diesem Fall sollten Sie eine dichtere Repräsentation namens Embedding verwenden. Dazu benötigen Sie ein gutes Verständnis der neuronalen Netzen (Details finden Sie in Kapitel 14).

Eigene Transformer

Obwohl Scikit-Learn viele nützliche Transformer bereitstellt, werden Sie für bestimmte Aufgaben bei der Datenaufbereitung oder zum Kombinieren bestimmter Merkmale Ihre eigenen schreiben müssen. Ihre Transformer sollten nahtlos mit den übrigen Funktionen von Scikit-Learn zusammenarbeiten (z.B. Pipelines), und da Scikit-Learn auf Duck Typing (anstelle von Vererbung) aufbaut, müssen Sie lediglich eine Klasse definieren und drei Methoden implementieren: fit() (welche self zurückgibt), transform() und fit_transform(). Sie können die letzte automatisch erhalten, indem Sie als Oberklasse TransformerMixin wählen. Wenn Sie außerdem BaseEstimator als Oberklasse wählen (und die Verwendung der Parametern *args und **kargs im Konstruktor vermeiden), erhalten Sie die zusätzlichen Methoden (get_params() und set_params()), die beim automatischen Einstellen der Hyperparameter hilfreich sind. Als Beispiel ist hier eine kleine Transformer-Klasse angegeben, die die zuvor besprochenen kombinierten Merkmale hinzufügt:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6
```

```

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # weder *args noch **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # sonst nichts zu tun
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                        bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)

```

In diesem Beispiel besitzt der Transformer einen Hyperparameter, `add_bedrooms_per_room`, der standardmäßig auf `True` gesetzt ist (es hilft oft, sinnvolle Standardwerte anzugeben). Mit diesem Hyperparameter können Sie leicht herausfinden, ob das Hinzufügen dieses Merkmals einen Machine-Learning-Algorithmus verbessert oder nicht. Im Allgemeinen können Sie für jeden Aufbereitungsschritt, dessen Sie sich nicht zu 100% sicher sind, einen Hyperparameter hinzufügen. Je stärker Sie die Schritte zur Datenaufbereitung automatisieren, desto mehr Kombinationen können Sie später automatisch ausprobieren lassen. Dadurch wird es wahrscheinlicher, dass Sie eine gute Kombination finden (und eine Menge Zeit einsparen).

Skalieren von Merkmalen

Das *Skalieren von Merkmalen* ist eine der wichtigsten Arten von Transformationen, die Sie werden anwenden müssen. Mit wenigen Ausnahmen können Machine-Learning-Algorithmen nicht besonders gut mit numerischen Eingabedaten auf unterschiedlichen Skalen arbeiten. Dies ist bei den Immobiliendaten der Fall: Die Gesamtzahl der Zimmer reicht von etwa 6 bis 39320, während das mittlere Einkommen nur von 0 bis 15 reicht. Eine Skalierung der Zielgröße ist dagegen im Allgemeinen nicht erforderlich.

Es gibt zwei übliche Verfahren, um sämtliche Merkmale auf die gleiche Skala zu bringen: die *Min-Max-Skalierung* und die *Standardisierung*.

Die Min-Max-Skalierung (viele nennen dies *Normalisieren*) ist schnell erklärt: Die Werte werden verschoben und so umskaliert, dass sie hinterher von 0 bis 1 reichen. Wir erreichen dies, indem wir den kleinsten Wert abziehen und anschließend durch die Differenz von Minimal- und Maximalwert teilen. Scikit-Learn enthält für diesen Zweck den Transformer `MinMaxScaler`. Über den Hyperparameter `feature_range` können Sie den Wertebereich einstellen, falls Sie aus irgendeinem Grunde nicht 0 – 1 erhalten möchten.

Die Standardisierung funktioniert deutlich anders: Bei dieser wird zuerst der Mittelwert abgezogen (sodass standardisierte Werte stets den Mittelwert Null besitzen), anschließend wird durch die Varianz geteilt, sodass die entstehende Verteilung eine Varianz von 1 hat. Im Gegensatz zur Min-Max-Skalierung sind die Werte bei der Standardisierung nicht an einen bestimmten Wertebereich gebunden. Für einige Algorithmen stellt dies ein Problem dar (z.B. erwarten neuronale Netze meist Eingabewerte zwischen 0 und 1). Dafür ist die Standardisierung wesentlich weniger anfällig für Ausreißer. Stellen Sie sich vor, ein Bezirk hätte durch einen Datenfehler ein mittleres Einkommen von 100. Die Min-Max-Skalierung würde alle übrigen Werte von 0 – 15 in den Bereich 0 – 0.15 quetschen, während sich bei der Standardisierung nicht viel ändert. Scikit-Learn enthält zur Standardisierung einen Transformer namens `StandardScaler`.



Wie bei allen Transformationen ist es wichtig, das Skalierungsverfahren nur mit den Trainingsdaten anzupassen, nicht mit dem vollständigen Datensatz (der die Testdaten enthält). Nur dann dürfen Sie diese zum Transformieren des Trainingsdatensatzes und des Testdatensatzes (und neuer Daten) verwenden.

Pipelines zur Transformation

Wie Sie sehen, sind viele Transformationsschritte in einer bestimmten Reihenfolge nötig. Glücklicherweise hilft die Klasse `Pipeline` in Scikit-Learn dabei, solche Abfolgen von Transformationen zu organisieren. Hier folgt eine kleine Pipeline für die numerischen Attribute:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Der Konstruktor von `Pipeline` benötigt eine Liste von Name-/Estimator-Paaren, wodurch die Abfolge der einzelnen Schritte definiert wird. Bis auf den letzten Estimator müssen sämtliche Estimatoren Transformer sein (d.h. die Methode `fit_transform()` besitzen). Die Namen können Sie beliebig wählen (solange sie keine doppelten Unterstriche enthalten »`_`«).

Wenn Sie die Methode `fit()` dieser Pipeline aufrufen, wird für jeden Transformer nacheinander `fit_transform()` aufgerufen, wobei die Ausgabe jedes Aufrufs automatisch als Eingabe für den nächsten Schritt verwendet wird. Beim letzten Estimator wird lediglich die Methode `fit()` aufgerufen.

Die Pipeline stellt die gleichen Methoden wie der letzte Estimator zur Verfügung. In diesem Falle ist der letzte Estimator ein `StandardScaler` und damit ein Transformer, sodass die Pipeline die Methode `transform()` besitzt, mit der sich sämtliche Transformationsschritte auf einen Datensatz anwenden lassen (sie besitzt auch die Methode `fit_transform`, die wir anstelle der Aufrufe von `fit()` und `transform()` ebenfalls nutzen könnten).

Es wäre schön, ein Pandas-DataFrame direkt in unsere Pipeline einzugeben, anstatt die numerischen Spalten zuerst in ein NumPy-Array umzuwandeln. Es gibt in Scikit-Learn keine Möglichkeit zum Verarbeiten von Pandas-DatFrames,¹⁹ aber wir können uns für diese Aufgabe einen eigenen Transformer schreiben:

```
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Unser `DataFrameSelector` transformiert die Daten, indem er die gewünschten Merkmale auswählt, den Rest verwirft und das resultierende DataFrame in ein NumPy-Array umwandelt. So können Sie leicht eine Pipeline schreiben, mit der Sie die numerischen Merkmale aus einem Pandas-DataFrame entnehmen können: Die Pipeline würde mit einem `DataFrameSelector` beginnen, der die numerischen Merkmale auswählt, anschließend könnten die zuvor besprochenen Vorverarbeitungsschritte folgen. Sie könnten ebenso eine zweite Pipeline für die kategorischen Merkmale schreiben, indem Sie diese mit einem `DataFrameSelector` auswählen und anschließend einen `LabelBinarizer` darauf anwenden.

```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('label_binarizer', LabelBinarizer()),
])
```

¹⁹ Pull Request #3886 könnte eine Klasse `ColumnTransformer` einführen, die merkmalspezifische Transformationen erleichtern soll. Sie könnten auch `pip3 install sklearn-pandas` ausführen, um die Klasse `DataFrameMapper` mit dem gleichen Ziel zu erhalten.

Aber wie lassen sich diese zwei Pipelines zu einer einzigen Pipeline verbinden? Die Antwort ist die Scikit-Learn-Klasse FeatureUnion. Dieser übergeben Sie eine Liste von Transformern (die wiederum ganze Transformationspipelines enthalten können); sobald deren Methode `transform()` aufgerufen wird, ruft sie die Methode `transform()` für jeden Transformer parallel auf, wartet auf die Ausgaben, hängt diese aneinander und gibt das Ergebnis zurück (und natürlich führt ein Aufruf der Methode `fit()` dazu, dass `fit()` für jeden Transformer aufgerufen wird). Eine vollständige Pipeline, die sowohl die numerischen als auch die kategorischen Attribute verarbeitet, könnte folgendermaßen aussehen:

```
from sklearn.pipeline import FeatureUnion

full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])
```

Sie können die gesamte Pipeline mit folgendem einfachen Befehl ausführen:

```
>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
       0.        ,  0.        ],
      [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
       0.        ,  0.        ],
      [...]
     >>> housing_prepared.shape
(16512, 16)
```

Wähle ein Modell aus und trainiere es

Endlich! Sie haben Ihre Aufgabe abgesteckt, die Daten erhalten und untersucht, Sie haben einen Trainings- und einen Testdatensatz erstellt, und Sie haben Pipelines zur Transformation geschrieben, um Ihre Daten automatisiert aufzuräumen und auf Machine-Learning-Algorithmen vorzubereiten. Nun sind Sie bereit, ein Machine-Learning-Modell auszuwählen und zu trainieren.

Trainieren und Auswerten auf dem Trainingsdatensatz

Die gute Nachricht ist, dass es dank der vorherigen Schritte nun wesentlich einfacher wird, als Sie vielleicht denken. Wir trainieren zuerst ein lineares Regressionsmodell wie im vorigen Kapitel:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Fertig! Sie haben nun ein funktionsfähiges lineares Regressionsmodell. Probieren wir es mit einigen Datenpunkten aus dem Trainingsdatensatz aus:

```

>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Vorhersagen:", lin_reg.predict(some_data_prepared))
Vorhersagen: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]

```

Es funktioniert, auch wenn die Vorhersagen nicht unbedingt sehr genau sind (z.B. liegt die erste Vorhersage um etwa 40 % daneben!). Bestimmen wir den RMSE dieses Regressionsmodells für den gesamten Trainingsdatensatz mithilfe der Funktion `mean_squared_error` aus Scikit-Learn:

```

>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.198198489219

```

In Ordnung, dieser RMSE ist besser als gar nichts, aber bestimmt nicht großartig: Der Wert `median_housing_values` liegt in den meisten Bezirken zwischen 120000 und 265000 USD. Damit ist eine typische Abweichung von 68628 bei der Vorhersage nicht sehr zufriedenstellend. Dies ist ein Beispiel für ein Modell, das die Trainingsdaten underfittet. Das kann bedeuten, dass die Merkmale nicht genügend Information für eine gute Vorhersage liefern oder dass das Modell nicht mächtig genug ist. Wie wir im vorigen Kapitel gesehen haben, sind die wichtigsten Gegenmaßnahmen bei Underfitting die Auswahl eines mächtigeren Modells, das Bereitstellen besserer Merkmale für den Trainingsalgorithmus und das Verringern von Restriktionen im Modell. Dieses Modell ist allerdings nicht regularisiert, und damit fällt die dritte Möglichkeit aus. Sie könnten versuchen, weitere Merkmale hinzuzufügen (beispielsweise den Logarithmus der Bevölkerung). Wir probieren aber erst einmal ein komplexeres Modell aus.

Wir trainieren nun einen `DecisionTreeRegressor`. Dies ist ein mächtiges Modell, das komplexe nichtlineare Zusammenhänge in den Daten erfassen kann (Entscheidungsbäume werden im Detail in Kapitel 6 vorgestellt). Der Code dazu sollte Ihnen mittlerweile bekannt vorkommen:

```

from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)

```

Das trainierte Modell können wir wieder auf dem Trainingsdatensatz auswerten:

```

>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0

```

Was bitte!? Überhaupt kein Fehler? Kann dieses Modell wirklich perfekt sein? Natürlich ist es wesentlich wahrscheinlicher, dass das Modell einem immensen Overfitting der Daten zum Opfer gefallen ist. Wie können wir uns dessen sicher sein? Wie wir weiter oben gesehen haben, sollten wir den Testdatensatz nicht anfassen, bis wir ein betriebsbereites Modell haben, dessen Qualität wir zuversichtlich einschätzen. Wir müssen daher einen Teil des Trainingsdatensatzes zum Trainieren verwenden und einen anderen Teil zur Validierung des Modells.

Bessere Auswertung mittels Kreuzvalidierung

Wir könnten den Entscheidungsbaum evaluieren, indem wir den Trainingsdatensatz mit der Funktion `train_test_split` in einen kleineren Trainingsdatensatz und einen Validierungsdatensatz aufteilen, dann das Modell mit dem kleineren Trainingsdatensatz trainieren und schließlich mit dem Validierungsdatensatz auswerten. Dies macht ein wenig Arbeit, ist aber nicht schwer und funktioniert recht gut.

Eine hervorragende Alternative dazu ist die in Scikit-Learn eingebaute *Kreuzvalidierung*. Der folgende Code führt eine *k-fache Kreuzvalidierung* durch: Er spaltet den Trainingsdatensatz zufällig in zehn unterschiedliche Teilmengen, genannt *Folds* auf. Anschließend trainiert und evaluiert er den Entscheidungsbaum zehnmal hintereinander, wobei jedes Mal ein anderer Fold zur Evaluierung genutzt wird, während auf den übrigen neun Folds trainiert wird. Das Ergebnis ist ein Array mit zehn Scores aus der Evaluierung:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```



Die Kreuzvalidierung in Scikit-Learn erwartet eine Nutzenfunktion (größer ist besser) anstatt einer Kostenfunktion (kleiner ist besser). Daher ist die Scoring-Funktion das Gegenteil des MSE (also ein negativer Wert). Deshalb steht im Code der Ausdruck `-scores` vor dem Berechnen der Quadratwurzel.

Betrachten wir das Ergebnis:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mittelwert:", scores.mean())
...     print("Standardabweichung:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 70232.0136482  66828.46839892  72444.08721003  70761.50186201
         71125.52697653  75581.29319857  70169.59286164  70055.37863456
         75370.49116773  71222.39081244]
Mittelwert: 71379.0744771
Standardabweichung: 2458.31882043
```

Nun steht der Entscheidungsbaum nicht mehr so gut da wie zuvor. Tatsächlich scheint er schlechter als das lineare Regressionsmodell abzuschneiden! Beachten Sie, dass wir mit der Kreuzvalidierung nicht nur eine Schätzung der Leistung unseres Modells erhalten, sondern auch eine Angabe darüber, wie präzise diese Schätzung ist (d.h. die Standardabweichung). Der Entscheidungsbaum hat einen RMSE von etwa $71379, \pm 2458$. Nur mit einem Validierungsdatensatz würden Sie diese Information nicht erhalten. Allerdings erfordert die Kreuzvalidierung, dass das Modell mehrmals trainiert wird. Deshalb ist sie nicht immer praktikabel.

Berechnen wir, um auf Nummer sicher zu gehen, die gleichen Scores für das lineare Regressionsmodell:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                                scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 66760.97371572  66962.61914244  70349.94853401  74757.02629506
         68031.13388938  71193.84183426  64968.13706527  68261.95557897
         71527.64217874  67665.10082067]
Mittelwert: 69047.8379055
Standardabweichung: 2735.51074287
```

Unsere Vermutung war richtig: Das Overfitting im Entscheidungsbaum ist so stark, dass dieses Modell ungenauer ist als die lineare Regression.

Probieren wir noch ein letztes Modell aus: den `RandomForestRegressor`. Wie wir in Kapitel 7 sehen werden, trainiert ein Random Forest viele Entscheidungsbäume auf zufällig ausgewählten Teilmengen der Merkmale und mittelt deren Vorhersagen. Ein Modell aus vielen anderen Modellen zusammenzusetzen, nennt man *Ensemble Learning*. Es ist oft eine gute Möglichkeit, ML-Algorithmen noch besser zu nutzen. Wir werden uns mit dem Code nicht groß beschäftigen, da er größtenteils der gleiche ist wie bei den anderen beiden Modellen:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
...
>>> forest_rmse
21941.911027380233
>>> display_scores(forest_rmse_scores)
Scores: [ 51650.94405471  48920.80645498  52979.16096752  54412.74042021
         50861.29381163  56488.55699727  51866.90120786  49752.24599537
         55399.50713191  53309.74548294]
Mittelwert: 52564.1902524
Standardabweichung: 2301.87380392
```

Wow, das ist viel besser: Random Forests wirken sehr vielversprechend. Allerdings ist der Score auf dem Trainingsdatensatz noch immer viel geringer als auf den Validierungsdatensätzen. Dies deutet darauf hin, dass das Modell die Trainingsdaten noch immer overfittet. Gegenmaßnahmen beim Overfitting sind, das Modell zu

vereinfachen, Restriktionen einzuführen (es zu regularisieren) oder deutlich mehr Trainingsdaten zu beschaffen. Bevor Sie sich aber eingehender mit Random Forests beschäftigen, sollten Sie mehr Modelle aus anderen Familien von Machine-Learning-Algorithmen ausprobieren (mehrere Support Vector Machines mit unterschiedlichen Kernels, ein neuronales Netz und so weiter), ohne jedoch zu viel Zeit mit dem Einstellen der Hyperparameter zu verbringen. Das Ziel ist, eine engere Auswahl (zwei bis fünf) der vielversprechendsten Modelle zu treffen.



Sie sollten jedes Modell, mit dem Sie experimentieren, abspeichern, sodass Sie später zu jedem beliebigen Modell zurückkehren können. Stellen Sie sicher, dass Sie sowohl die Hyperparameter als auch die trainierten Parameter abspeichern, ebenso die Scores der Kreuzvalidierung und eventuell sogar die Vorhersagen. Damit können Sie leichter Vergleiche der Scores und der Arten der Fehler zwischen unterschiedlichen Modellen ziehen. Sie können in Scikit-Learn erstellte Modelle mit dem Python-Modul `pickle` oder dem Modul `sklearn.externals.joblib` speichern, wobei Letzteres große NumPy-Arrays effizienter serialisiert:

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# und später...
my_model_loaded = joblib.load("my_model.pkl")
```

Optimiere das Modell

Nehmen wir an, Sie hätten inzwischen eine engere Auswahl Erfolg versprechender Modelle. Nun müssen Sie diese optimieren. Betrachten wir dazu einige Alternativen.

Gittersuche

Eine Möglichkeit wäre, von Hand an den Hyperparametern herumzubasteln, bis Sie eine gute Kombination finden. Dies wäre sehr mühselig, und Sie hätten nicht die Zeit, viele Kombinationen auszuprobieren.

Stattdessen sollten Sie die Scikit-Learn-Klasse `GridSearchCV` die Suche für Sie erledigen lassen. Sie müssen ihr lediglich sagen, mit welchen Hyperparametern Sie experimentieren möchten und welche Werte ausprobiert werden sollen. Dann werden alle möglichen Kombinationen von Hyperparametern über eine Kreuzvalidierung evaluiert. Der folgende Code sucht die beste Kombination der Hyperparameter für den `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
```

```

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)

```



Wenn Sie keine Ahnung haben, welchen Wert ein Hyperparameter haben soll, können Sie einfach Zehnerpotenzen ausprobieren (oder für eine feinkörnigere Suche Potenzen einer kleineren Zahl, wie im Beispiel beim Hyperparameter `n_estimators`).

Das `param_grid` weist Scikit-Learn an, zuerst alle Kombinationen von $3 \times 4 = 12$ der Hyperparameter `n_estimators` und `max_features` mit den im ersten dict angegebenen Werten auszuprobieren (keine Sorge, wenn Sie die Bedeutung der Hyperparameter noch nicht kennen; diese werden in Kapitel 7 erklärt). Anschließend werden alle Kombinationen $2 \times 3 = 6$ der Hyperparameter im zweiten dict ausprobiert, diesmal ist jedoch der Hyperparameter `bootstrap` auf `False` statt `True` (den Standardwert) gesetzt.

Insgesamt probiert die Gittersuche Kombinationen von $12 + 6 = 18$ der Hyperparameter mit dem `RandomForestRegressor` aus. Jedes Modell wird fünf Mal trainiert (weil wir eine fünffache Kreuzvalidierung verwenden). Anders gesagt, gibt es Trainingsrunden der Art $18 \times 5 = 90!$ Es kann eine ganze Weile dauern, aber schließlich können Sie die beste Parameterkombination wie folgt abfragen:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```



Da 8 und 30 die maximalen ausprobierten Werte sind, sollten wir noch einmal mit höheren Werten suchen, da der Score vielleicht noch besser wird.

Sie können auch direkt auf den besten Estimator zugreifen:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features=8, max_leaf_nodes=None, min_impurity_split=1e-07,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=1,
                      oob_score=False, random_state=42, verbose=0, warm_start=False)
```



Falls `GridSearchCV` mit `refit=True` initialisiert wird (der Standardeinstellung), wird der beste über Kreuzvalidierung gefundene Estimator noch einmal mit dem gesamten Trainingsdatensatz trainiert. Dies ist grundsätzlich eine gute Idee, da mehr Daten voraussichtlich die Genauigkeit erhöhen.

Natürlich sind auch die Scores der Evaluation verfügbar:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63825.0479302 {'max_features': 2, 'n_estimators': 3}
55643.8429091 {'max_features': 2, 'n_estimators': 10}
53380.6566859 {'max_features': 2, 'n_estimators': 30}
60959.1388585 {'max_features': 4, 'n_estimators': 3}
52740.5841667 {'max_features': 4, 'n_estimators': 10}
50374.1421461 {'max_features': 4, 'n_estimators': 30}
58661.2866462 {'max_features': 6, 'n_estimators': 3}
52009.9739798 {'max_features': 6, 'n_estimators': 10}
50154.1177737 {'max_features': 6, 'n_estimators': 30}
57865.3616801 {'max_features': 8, 'n_estimators': 3}
51730.0755087 {'max_features': 8, 'n_estimators': 10}
49694.8514333 {'max_features': 8, 'n_estimators': 30}
62874.4073931 {'max_features': 2, 'n_estimators': 3, 'bootstrap': False}
54561.9398157 {'max_features': 2, 'n_estimators': 10, 'bootstrap': False}
59416.6463145 {'max_features': 3, 'n_estimators': 3, 'bootstrap': False}
52660.245911 {'max_features': 3, 'n_estimators': 10, 'bootstrap': False}
57490.0168279 {'max_features': 4, 'n_estimators': 3, 'bootstrap': False}
51093.9059428 {'max_features': 4, 'n_estimators': 10, 'bootstrap': False}
```

In diesem Beispiel erhalten wir die beste Lösung, indem wir den Hyperparameter `max_features` auf 8 setzen und den Hyperparameter `n_estimators` auf 30. Der RMSE beträgt bei dieser Kombination 49694, was etwas besser als der zuvor mit den voreingestellten Hyperparametern berechnete Wert ist (dieser betrug 52564). Herzlichen Glückwunsch, Sie haben Ihr bestes Modell erfolgreich optimiert!



Vergessen Sie nicht, dass Sie auch einige der Vorverarbeitungsschritte als Hyperparameter ansehen können. Die Gittersuche kann automatisch herausfinden, ob Sie ein Merkmal hinzufügen sollten, dessen Sie sich nicht sicher sind (z.B. der Hyperparameter `add_bedrooms_per_room` Hyperparameter Ihres Transformers `CombinedAttributesAdder`). In ähnlicher Weise kann die Gittersuche die beste Möglichkeit finden, mit Ausreißern und fehlenden Werten umzugehen, Merkmale auszuwählen und mehr.

Zufällige Suche

Die Gittersuche ist als Ansatz gut geeignet, wenn Sie wie im Beispiel oben relativ wenige Kombinationen durchsuchen möchten. Ist der *Suchraum* für die Hyperparameter jedoch groß, ist stattdessen `RandomizedSearchCV` vorzuziehen. Diese Klasse lässt sich genauso wie `GridSearchCV` verwenden, aber anstatt alle möglichen Kombinationen auszuprobieren, wird eine gegebene Anzahl zufälliger Kombinationen evaluiert, indem in jedem Durchlauf ein zufälliger Wert für jeden Hyperparameter gebildet wird. Dieser Ansatz hat zwei Vorteile:

- Wenn Sie die zufällige Suche für 1000 Iterationen laufen lassen, werden für jeden Hyperparameter 1000 unterschiedliche Werte ausprobiert (anstatt nur einige Werte pro Hyperparameter wie bei der Gittersuche).
- Sie haben eine stärkere Kontrolle über die Rechenzeit, die Sie der Suche nach Hyperparametern zuteilen möchten, indem Sie einfach die Anzahl Iterationen festlegen.

Ensemble-Methoden

Eine weitere Möglichkeit zur Optimierung Ihres Systems ist, die Modelle mit der besten Leistung miteinander zu kombinieren. Die Gruppe (oder das »Ensemble«) schneidet oft besser ab als das beste Modell für sich allein (genauso wie ein Random Forest mehr leistet als ein einzelner Entscheidungsbaum), insbesondere wenn die einzelnen Modelle unterschiedliche Arten von Fehlern machen. Mit diesem Thema werden wir uns in Kapitel 7 beschäftigen.

Analysiere die besten Modelle und ihre Fehler

Oft lassen sich Erkenntnisse über die Aufgabe durch Inspizieren der besten Modelle gewinnen. Beispielsweise zeigt der RandomForestRegressor die relative Wichtigkeit jedes Merkmals für genaue Vorhersagen:

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.33442355e-02,   6.29090705e-02,   4.11437985e-02,
       1.46726854e-02,   1.41064835e-02,   1.48742809e-02,
       1.42575993e-02,   3.66158981e-01,   5.64191792e-02,
       1.08792957e-01,   5.33510773e-02,   1.03114883e-02,
       1.64780994e-01,   6.02803867e-05,   1.96041560e-03,
       2.85647464e-03])
```

Stellen wir die Scores für die Wichtigkeit der Merkmale gemeinsam mit ihren Namen dar:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_one_hot_attribs = list(encoder.classes_)
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
```

```
(0.014106483453584102, 'total_bedrooms'),  
(0.010311488326303787, '<1H OCEAN'),  
(0.0028564746373201579, 'NEAR OCEAN'),  
(0.0019604155994780701, 'NEAR BAY'),  
(6.0280386727365991e-05, 'ISLAND'))]
```

Mit dieser Information könnten Sie einige der weniger nützlichen Merkmale entfernen (z. B. ist offenbar nur eine Kategorie für die Nähe zum Ozean wirklich nützlich, Sie könnten versuchen, die übrigen wegzulassen).

Sie sollten ebenfalls versuchen, die Fehler Ihres Systems zu betrachten und zu verstehen, warum es diese begeht und wie sie behoben werden könnten (z. B. Hinzufügen von zusätzlichen Merkmalen oder im Gegenteil Entfernen von überflüssigen, von Ausreißern und so weiter).

Evaluiere das System auf dem Testdatensatz

Nachdem Sie Ihre Modelle eine Weile lang optimiert haben, haben Sie ein System, das ausreichend gut funktioniert. Nun ist es an der Zeit, das endgültige Modell mit dem Testdatensatz zu evaluieren. Daran ist nichts Besonderes; Sie nehmen die Prädiktoren und Labels Ihres Testdatensatzes, transformieren die Daten mit `full_pipeline` (rufen Sie `transform()` auf, *nicht fit_transform()!*) und evaluieren das endgültige Modell mit den Testdaten:

```
final_model = grid_search.best_estimator_  
  
X_test = strat_test_set.drop("median_house_value", axis=1)  
y_test = strat_test_set["median_house_value"].copy()  
  
X_test_prepared = full_pipeline.transform(X_test)  
  
final_predictions = final_model.predict(X_test_prepared)  
  
final_mse = mean_squared_error(y_test, final_predictions)  
final_rmse = np.sqrt(final_mse) # => evaluiert zu 47766.0
```

Wenn Sie eine Menge Hyperparameter optimiert haben, ist die Qualität der Vorhersage normalerweise etwas schlechter als die mit der Kreuzvalidierung bestimmte (weil Ihr System dann auf die Validierungsdaten optimiert ist und nicht ganz so gut auf unbekannten Daten abschneidet). In diesem Beispiel ist das nicht der Fall, aber wenn es passiert, müssen Sie der Versuchung widerstehen, Ihre Hyperparameter noch einmal zu ändern, um die Zahlen gut aussehen zu lassen; die Verbesserungen werden vermutlich nicht gut auf neue Daten verallgemeinerbar sein.

Es folgt die Phase vor der Inbetriebnahme; Sie müssen Ihre Lösung präsentieren (und hervorheben, was Sie herausgefunden haben, was funktioniert hat und was nicht, welche Annahmen getroffen wurden und was die Grenzen Ihres Systems sind), dokumentieren und ansprechendes Präsentationsmaterial mit klaren Visua-

lisierungen und eingängigen Aussagen (z.B. »das mittlere Einkommen ist der beste Prädiktor des Immobilienpreises«) anbieten.

Nimm das System in Betrieb, überwache und warte es

Sie haben die Freigabe für die Inbetriebnahme! Sie müssen Ihre Lösung nun fit machen für die Produktivumgebung, indem Sie die Datenquellen für den Betrieb anschließen und Tests schreiben.

Sie müssen auch Code zur Überwachung schreiben, damit Sie die Leistung des laufenden Systems in regelmäßigen Abständen prüfen können und ein Alarm ausgelöst wird, wenn sie abfällt. Dies ist wichtig, damit Sie nicht nur plötzliche Ausfälle, sondern auch einen allmählichen Leistungsverfall erfassen. Letztere Situation ist häufig, weil sich Daten mit der Zeit weiterentwickeln und Modelle deshalb »verrotten«, es sei denn, sie werden regelmäßig mit frischen Daten trainiert.

Zum Evaluieren der Leistung Ihres Systems müssen Sie eine Stichprobe aus den Vorhersagen Ihres Systems ziehen und diese auswerten. Diese Beobachtung sollte grundsätzlich ein Mensch vornehmen. Die Analysten können Experten des Fachgebiets sein oder Arbeiter auf einer Plattform für Crowdsourcing wie Amazon Mechanical Turk oder CrowdFlower. In beiden Fällen müssen Sie die menschliche Pipeline zum Auswerten in Ihr System einbauen.

Sie sollten auch die Qualität der Eingabedaten evaluieren. Manchmal fällt die Leistung wegen eines schlechten Signals leicht ab (z.B. ein Sensor mit Fehlfunktion, der zufällige Werte sendet, oder eine altbackene Ausgabe eines anderen Teams), es kann aber lange dauern, bis Ihre Überwachung deswegen einen Alarm auslöst. Wenn Sie die Eingabedaten des Systems überwachen, können Sie dies früh feststellen. Ein Überwachen der Eingabe ist beim Online-Learning besonders wichtig.

Schließlich sollten Sie Ihre Modelle regelmäßig mit neuen Daten trainieren. Soweit möglich, sollten Sie diesen Prozess automatisieren. Wenn Sie dies nicht tun, werden Sie das Modell wahrscheinlich (im besten Fall) alle sechs Monate aktualisieren müssen, und die Leistung wird daher mit der Zeit schwanken. Falls Ihr System ein Online-Learning-System ist, sollten Sie dessen Zustand in regelmäßigen Abständen sichern, damit Sie leicht zu einem funktionsfähigen Zustand zurückkehren können.

Probieren Sie es aus!

Dieses Kapitel hat Ihnen hoffentlich einen Eindruck verschafft, worin ein Machine-Learning-Projekt besteht, und Ihnen die Werkzeuge zum Trainieren eines guten Systems nähergebracht. Wie Sie sehen, besteht ein Großteil der Arbeit aus der Vorbereitung der Daten, dem Bauen von Überwachungswerkzeugen, dem Aufbau

einer von Menschen gestützten Pipeline und dem Automatisieren des regelmäßigen Trainings Ihrer Modelle. Die Algorithmen zum Machine Learning sind natürlich ebenfalls wichtig, aber es ist günstiger, den Prozess insgesamt gut zu beherrschen und drei oder vier Algorithmen gut zu kennen, anstatt Ihre gesamte Zeit mit dem Untersuchen ausgefeilter Algorithmen zu verbringen und zu wenig Zeit für den Gesamtprozess übrigzuhaben.

Wenn Sie es also noch nicht ohnehin getan haben, ist nun ein guter Moment, den Laptop aufzuklappen, einen interessanten Datensatz auszusuchen und den gesamten Prozess von A bis Z durchzuarbeiten. Ein guter Ausgangspunkt ist eine Webseite für Wettbewerbe wie <http://kaggle.com/>: Dort finden Sie Datensätze zum Ausprobieren, klare Zielvorgaben und können Erfahrungen austauschen.

Übungen

Verwenden Sie den Immobilien-Datensatz aus diesem Kapitel für folgende Aufgaben:

1. Probieren Sie einen Support Vector Machine Regressor (`sklearn.svm.SVR`) mit unterschiedlichen Hyperparametern aus, beispielsweise `kernel="linear"` (mit unterschiedlichen Werten für den Hyperparameter `C`) oder `kernel="rbf"` (mit unterschiedlichen Werten für die Hyperparameter `C` und `gamma`). Kümmern Sie sich im Moment nicht zu sehr darum, was Hyperparameter überhaupt sind. Wie gut schneidet der beste SVR-Prädiktor ab?
2. Ersetzen Sie `GridSearchCV` durch `RandomizedSearchCV`.
3. Fügen Sie der vorbereitenden Pipeline einen Transformer hinzu, um nur die wichtigsten Attribute auszuwählen.
4. Erstellen Sie eine einzige Pipeline, die die vollständige Vorverarbeitung der Daten und die endgültige Vorhersage durchführt.
5. Probieren Sie einige der Optionen bei der Vorverarbeitung mit `GridSearchCV` aus.

Lösungen zu diesen Übungsaufgaben finden Sie online in den Jupyter-Notebooks auf <https://github.com/ageron/handson-ml>.

KAPITEL 3

Klassifikation

In Kapitel 1 haben wir erwähnt, dass die häufigsten Aufgaben beim überwachten Lernen Regression (die Vorhersage von Werten) und Klassifikation (die Vorhersage von Kategorien) sind. In Kapitel 2 haben wir uns eine Regressionsaufgabe genauer angesehen, bei der wir den Wert von Immobilien mit unterschiedlichen Algorithmen wie linearer Regression, Entscheidungsbäumen und Random Forests vorhergesagt haben (diese Methoden werden in späteren Kapiteln genauer vorgestellt). Wenden wir uns nun Systemen zur Klassifikation zu.

MNIST

In diesem Kapitel werden wir den MNIST-Datensatz verwenden, eine Sammlung von 70000 kleinen Bildern handschriftlicher Ziffern, die von Oberschülern und Mitarbeitern des US Census Bureaus aufgeschrieben wurden. Jedes Bild ist mit der dargestellten Ziffer gelabelt. Dieser Datensatz ist so intensiv untersucht worden, dass er oft als »Hello World« des Machine Learning bezeichnet wird: Wann immer jemand ein neues Klassifikationsverfahren entwickelt, möchte man wissen, wie es auf MNIST abschneidet. Jeder, der Machine Learning lernt, beschäftigt sich früher oder später mit MNIST.

Scikit-Learn enthält viele Hilfsfunktionen zum Herunterladen verbreiteter Datensätze. MNIST ist einer davon. Der folgende Code besorgt den MNIST-Datensatz:¹

```
>>> from sklearn.datasets import fetch_mldata  
>>> mnist = fetch_mldata('MNIST original')  
>>> mnist  
{'COL_NAMES': ['label', 'data'],  
 'DESCR': 'mldata.org dataset: mnist-original',  
 'data': array([[0, 0, 0, ..., 0, 0, 0],  
 [0, 0, 0, ..., 0, 0, 0],  
 [0, 0, 0, ..., 0, 0, 0],
```

¹ Scikit-Learn speichert heruntergeladene Daten standardmäßig im Verzeichnis \$HOME/scikit_learn_data.

```

...,
[0, 0, 0, ..., 0, 0, 0],
[0, 0, 0, ..., 0, 0, 0],
[0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
'target': array([ 0.,  0.,  0., ...,  9.,  9.])}

```

Die von Scikit-Learn heruntergeladenen Datensätze sind für gewöhnlich Dictionaries mit einer ähnlichen Struktur, bestehend aus folgenden Schlüsseln:

- Der Schlüssel DESCR beschreibt den Datensatz.
- Der Schlüssel data enthält ein Array mit einer Zeile pro Datenpunkt und einer Spalte pro Merkmal.
- Der Schlüssel target enthält ein Array mit den Labels.

Betrachten wir die beiden Arrays:

```

>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)

```

Es gibt 70000 Bilder, jedes davon hat 768 Merkmale. Das liegt daran, dass jedes Bild aus 28 x 28 Pixeln besteht und jedes Merkmal einfach die Intensität eines Pixels von 0 (weiß) bis 255 (schwarz) enthält. Betrachten wir eine Ziffer aus dem Datensatz. Dazu müssen wir lediglich den Merkmalsvektor eines Datenpunkts herausgreifen, zu einem Array mit den Abmessungen 28 x 28 umformatieren und mit der Funktion imshow() aus Matplotlib darstellen:

```

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off")
plt.show()

```



Dieses Bild sieht wie eine 5 aus, was uns das Label auch bestätigt:

```

>>> y[36000]
5.0

```

Abbildung 3-1 zeigt einige weitere Bilder aus dem MNIST-Datensatz, um Ihnen ein Gefühl für die Komplexität dieser Klassifikationsaufgabe zu geben.



Abbildung 3-1: Einige Ziffern aus dem MNIST-Datensatz

Einen Moment noch! Sie sollten stets einen Testdatensatz erstellen und vor dem genaueren Betrachten der Daten beiseitelegen. Der MNIST-Datensatz ist bereits in Trainingsdaten (die ersten 60000 Bilder) und Testdaten (die letzten 10000 Bilder) unterteilt:

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Wir mischen außerdem die Trainingsdaten; damit stellen wir sicher, dass bei der Kreuzvalidierung sämtliche Folds einander ähnlich sind (Sie möchten nicht, dass einige Ziffern in einem Fold fehlen). Außerdem reagieren einige Lernalgorithmen sensibel auf die Reihenfolge der Trainingsdatenpunkte und schneiden schlechter ab, wenn sie viele ähnliche Datenpunkte nacheinander erhalten. Das Mischen des Datensatzes sorgt dafür, dass dies nicht passiert:²

```
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

Trainieren eines binären Klassifikators

Für den Anfang werden wir die Aufgabe vereinfachen und lediglich versuchen, eine Ziffer zu erkennen – beispielsweise die Ziffer 5. Dieser »5en-Detektor« ist ein Bei-

² In einigen Fällen ist das Mischen keine gute Idee – wenn Sie beispielsweise mit Zeitreihen arbeiten (wie Aktienkursen oder Wetterdaten). Diese Fälle werden wir in den nächsten Kapiteln betrachten.

spiel für einen *binären Klassifikator*, mit dem sich genau zwei Kategorien unterscheiden lassen, 5 und nicht-5. Erstellen wir also die Zielvektoren für diese Klassifikationsaufgabe:

```
y_train_5 = (y_train == 5) # True bei allen 5en, False bei allen anderen Ziffern.  
y_test_5 = (y_test == 5)
```

Nun wählen wir einen Klassifikator aus und trainieren diesen. Ein guter Ausgangspunkt ist der Klassifikator für das *stochastische Gradientenverfahren* (SGD), dem die Klasse `SGDClassifier` in Scikit-Learn entspricht. Dieser Klassifikator hat den Vorteil, sehr große Datensätze effizient zu bearbeiten. Dies liegt daran, dass SGD die Trainingsdatenpunkte einzeln und nacheinander abarbeitet (wodurch SGD außerdem für *Online-Learning* gut geeignet ist). Erstellen wir zunächst einen `SGDClassifier` und trainieren diesen auf dem gesamten Trainingsdatensatz:

```
from sklearn.linear_model import SGDClassifier  
  
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train_5)
```



Der `SGDClassifier` arbeitet beim Trainieren zufallsbasiert (daher der Name »stochastisch«). Wenn Sie reproduzierbare Ergebnisse erhalten möchten, sollten Sie den Parameter `random_state` setzen.

Nun können Sie mit dem Klassifikator Bilder mit der Nummer 5 erkennen:

```
>>> sgd_clf.predict([some_digit])  
array([ True], dtype=bool)
```

Der Klassifikator meint, dass dieses Bild eine 5 darstellt (`True`). Es sieht so aus, als hätte er in diesem Fall richtig geraten! Werten wir nun die Qualität dieses Modells aus.

Qualitätsmaße

Einen Klassifikator auszuwerten, ist oft deutlich verzwickter, als einen Regressor auszuwerten, daher werden wir einen Großteil dieses Kapitels mit diesem Thema zubringen. Es gibt viele unterschiedliche Qualitätsmaße. Schnappen Sie sich also noch einen Kaffee und seien Sie bereit, viele neue Begriffe und Abkürzungen kennenzulernen!

Messen der Genauigkeit über Kreuzvalidierung

Kreuzvalidierung ist eine gute Möglichkeit zum Auswerten von Modellen, wie Sie bereits in Kapitel 2 gesehen haben.

Implementierung der Kreuzvalidierung

Bisweilen benötigen Sie eine genauere Kontrolle über den Prozess der Kreuzvalidierung, als sie von Scikit-Learn angeboten wird. In diesen Fällen können Sie die Kreuzvalidierung selbst implementieren, was eigentlich recht einfach ist. Das folgende Codebeispiel tut in etwa das Gleiche wie die Scikit-Learn-Funktion `cross_val_score()` und liefert das gleiche Ergebnis:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # gibt 0.9502, 0.96565 und 0.96495 aus
```

Die Klasse `StratifiedKFold` bildet eine stratifizierte Stichprobe (wie in Kapitel 2 erklärt), um Folds mit einer repräsentativen Anzahl Punkte aus jeder Kategorie zu ermitteln. Bei jeder Iteration erstellt der Code einen Klon des Klassifikators, trainiert diesen auf den zum Trainieren abgestellten Folds und führt eine Vorhersage für den Fold zum Testen durch. Anschließend ermittelt er die Anzahl korrekter Vorhersagen und gibt deren Anteil aus.

Wir verwenden die Funktion `cross_val_score()`, um unser `SGDClassifier`-Modell mit k-facher Kreuzvalidierung und drei Folds auszuwerten. Bei der k-fachen Kreuzvalidierung wird der Trainingsdatensatz bekanntlich in k Folds aufgeteilt (drei in diesem Fall), anschließend werden auf jedem Fold Vorhersagen getroffen und ausgewertet, wobei das Modell mit den jeweils restlichen Folds trainiert wird (siehe Kapitel 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502,  0.96565,  0.96495])
```

Wow! Eine *Genauigkeit* über 95% (Anteil korrekter Vorhersagen) auf sämtlichen Folds der Kreuzvalidierung? Das sieht fantastisch aus, nicht wahr? Bevor Sie sich aber vom Ergebnis mitreißen lassen, betrachten wir zum Vergleich einen sehr primitiven Klassifikator, der einfach jedes Bild der Kategorie »nicht-5« zuordnet:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
```

```

def fit(self, X, y=None):
    pass
def predict(self, X):
    return np.zeros((len(X), 1), dtype=bool)

```

Können Sie die Genauigkeit dieses Modells erraten? Finden wir es heraus:

```

>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 ,  0.90715,  0.9128 ])

```

Richtig, es hat eine Genauigkeit von über 90%! Dies liegt einfach daran, dass nur etwa 10% der Bilder 5en sind. Wenn Sie also jedes Mal darauf tippen, dass ein Bild *keine 5* enthält, werden Sie in 90% der Fälle richtig liegen. Damit schlagen Sie sogar Nostradamus.

Dies zeigt, warum die Genauigkeit bei Klassifikatoren für gewöhnlich nicht das Qualitätsmaß der Wahl ist, besonders wenn Sie es mit *unbalancierten Datensätzen* zu tun haben (also solchen, bei denen einige Kategorien viel häufiger als andere auftreten).

Konfusionsmatrix

Eine weitaus bessere Möglichkeit zum Auswerten der Vorhersageleistung eines Klassifikators ist das Betrachten einer *Konfusionsmatrix*. Die Idee dabei ist, die Datenpunkte auszuzählen, bei denen Kategorie A als Kategorie B klassifiziert wird. Um zum Beispiel zu sehen, wie oft der Klassifikator das Bild einer 5 für eine 3 gehalten hat, würden Sie in der 5. Zeile und 3. Spalte der Konfusionsmatrix nachschauen.

Um eine Konfusionsmatrix zu berechnen, benötigen Sie einen Satz Vorhersagen, die sich dann mit den korrekten Zielwerten vergleichen lassen. Sie könnten auch auf dem Testdatensatz Vorhersagen vornehmen, wir lassen es aber im Moment beiseite (denken Sie daran, dass Sie die Testdaten erst ganz am Ende Ihres Projekts verwenden sollten, sobald Sie einen einsatzbereiten Klassifikator haben). Stattdessen können Sie die Funktion `cross_val_predict()` verwenden:

```

from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

```

Wie die Funktion `cross_val_score()` führt auch `cross_val_predict()` eine k-fache Kreuzvalidierung durch, aber anstatt Genauigkeiten zurückzugeben, liefert sie die für jeden Test-Fold berechneten Vorhersagen. Damit erhalten Sie für jeden Datenpunkt im Trainingsdatensatz eine saubere Vorhersage (»sauber« bedeutet, dass das zur Vorhersage eingesetzte Modell diese Daten beim Trainieren nicht gesehen hat).

Nun sind wir bereit, die Konfusionsmatrix mit der Funktion `confusion_matrix()` zu berechnen. Wir übergeben dieser einfach die Zielkategorien (`y_train_5`) und die vorhergesagten Kategorien (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
       [1077, 4344]])
```

Jede Zeile in einer Konfusionsmatrix steht für eine *tatsächliche Kategorie*, während jede Spalte eine *vorhergesagte Kategorie* darstellt. Die erste Zeile dieser Matrix beinhaltet nicht-5-Bilder (die *negative Kategorie*): 53272 von diesen wurden korrekt als Nicht-5en klassifiziert (man nennt diese *richtig Negative*), die übrigen 1307 wurden fälschlicherweise als 5en klassifiziert (*falsch Positive*). Die zweite Zeile betrachtet die Bilder mit 5en (die *positive Kategorie*): 1077 Bilder wurden fälschlicherweise als Nicht-5en vorhergesagt (*falsch Negative*), die restlichen 4344 wurden korrekt als 5en vorhergesagt (*richtig Positive*). Ein perfekter Klassifikator würde lediglich richtig Positive und richtig Negative produzieren, sodass die Konfusionsmatrix nur auf der Hauptdiagonalen (von links oben nach rechts unten) Werte ungleich null enthielte:

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
       [0, 5421]])
```

Die Konfusionsmatrix gibt uns eine Menge Informationen, aber manchmal wünscht man sich ein etwas komakteres Qualitätsmaß. Ein interessantes Maß ist die Genauigkeit der positiven Vorhersagen; dies nennt man auch die *Relevanz* (engl. precision) des Klassifikators (Formel 3-1).

Formel 3-1: Relevanz

$$\text{Relevanz} = \frac{RP}{RP + FP}$$

RP ist dabei die Anzahl richtig Positiver und FP die Anzahl falsch Positiver.

Eine perfekte Relevanz lässt sich trivialerweise erhalten, indem man eine einzige positive Vorhersage trifft und sicherstellt, dass sie richtig ist (Relevanz = 1/1 = 100 %). Dies wäre nicht besonders nützlich, da der Klassifikator dann alle außer einem positiven Datenpunkt ignorieren würde. Daher geht die Relevanz üblicherweise mit einem zweiten Maß namens *Sensitivität* (engl. recall) einher, das auch als *Trefferquote* oder *Richtig-positiv-Rate (TPR)* bezeichnet wird: dieses ist der Anteil positiver Datenpunkte, die vom Klassifikator entdeckt wurden (Formel 3-2).

Formel 3-2: Sensitivität

$$\text{Sensitivität} = \frac{RP}{RP + FN}$$

RN ist dabei natürlich die Anzahl falsch Negativer.

Falls Sie die Konfusionsmatrix verwirrt, hilft Ihnen womöglich Abbildung 3-2.

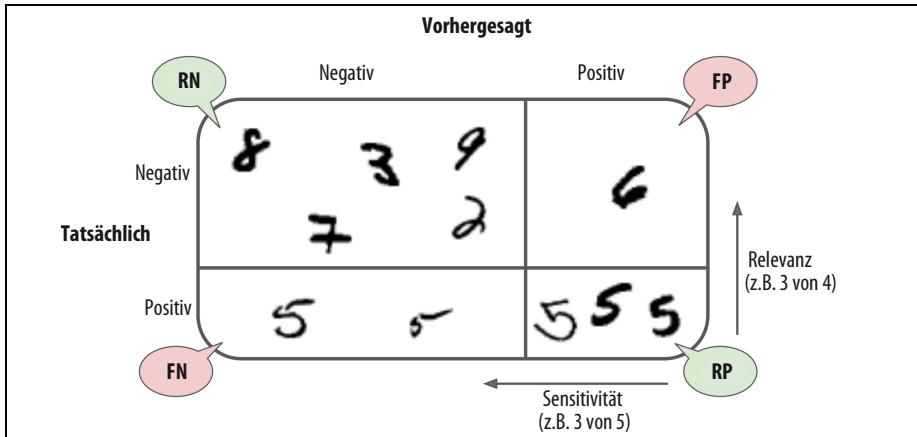


Abbildung 3-2: Eine illustrierte Konfusionsmatrix

Relevanz und Sensitivität

Scikit-Learn enthält mehrere Funktionen zum Berechnen von Klassifikationsmetriken, darunter Relevanz und Sensitivität:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)      # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred)    # == 4344 / (4344 + 1077)
0.79136690647482011
```

Damit glänzt Ihr 5en-Detektor nicht mehr so stark, wie es die Genauigkeit vermuten ließ. Wenn er erklärt, dass ein Bild eine 5 enthält, liegt er nur in 77% der Fälle richtig. Außerdem findet er nur 79% der vorhandenen 5en.

Es ist oft bequem, Relevanz und Sensitivität zu einer einzigen Metrik zu kombinieren, dem F_1 -Score, insbesondere, wenn Sie zwei Klassifikatoren miteinander vergleichen möchten. Der F_1 -Score ist der *harmonische Mittelwert* von Relevanz und Sensitivität (Formel 3-3). Während der gewöhnliche Mittelwert alle Werte gleich behandelt, erhalten beim harmonischen Mittelwert niedrigere Mittelwerte ein weit aus höheres Gewicht. Daher erhält ein Klassifikator nur dann einen hohen F_1 -Score, wenn sowohl Relevanz als auch Sensitivität hoch sind.

Formel 3-3: F_1 -Score

$$F_1 = \frac{2}{\frac{1}{\text{Relevanz}} + \frac{1}{\text{Sensitivität}}} = 2 \times \frac{\text{Relevanz} \times \text{Sensitivität}}{\text{Relevanz} + \text{Sensitivität}} = \frac{RP}{RP + \frac{FN+FP}{2}}$$

Der F_1 -Score, lässt sich mit der Funktion `f1_score()` berechnen:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_pred)
0.78468208092485547
```

Der F_1 -Score begünstigt Klassifikatoren mit ähnlicher Relevanz und Sensitivität. Das ist nicht immer erwünscht: Bei manchen Anwendungen ist Relevanz wichtiger, in anderen ist die Sensitivität von herausragender Bedeutung. Wenn Sie beispielsweise einen Klassifikator trainieren, der für Kinder geeignete Videos erkennt, werden Sie vermutlich einen Klassifikator bevorzugen, der zwar viele gute Videos verwirft (niedrige Sensitivität), aber wirklich nur geeignete anzeigt (hohe Relevanz). Ablehnen werden Sie hingegen den Klassifikator, der zwar eine hohe Sensitivität aufweist, dafür aber ein paar wirklich ungeeignete Videos in Ihrem Produkt anzeigt (in solchen Fällen könnten Sie sogar erwägen, die Videoauswahl des Klassifikators von Hand zu überprüfen). Wenn Sie andererseits Einkaufsdiebstähle auf den Bildern einer Überwachungskamera erkennen möchten, ist es vermutlich in Ordnung, wenn Ihr Klassifikator eine Relevanz von nur 30% erzielt, dafür aber eine Sensitivität von 99% aufweist (das Sicherheitspersonal wird dann einige Fehlalarme erhalten, aber fast alle Diebe werden geschnappt).

Leider können Sie nicht beides haben: Ein Erhöhen der Relevanz senkt die Sensitivität und umgekehrt. Dies nennt man auch die *Wechselbeziehung zwischen Relevanz und Sensitivität*.

Die Wechselbeziehung zwischen Relevanz und Sensitivität

Um diese Wechselbeziehung besser zu verstehen, betrachten wir, wie der SGDClassifier bei der Klassifikation Entscheidungen trifft. Für jeden Datenpunkt wird anhand einer *Entscheidungsfunktion* ein Score berechnet, und falls dieser Score über einem Schwellenwert liegt, wird der Datenpunkt der positiven Kategorie zugeordnet, andernfalls der negativen Kategorie. Abbildung 3-3 zeigt einige Ziffern, mit dem niedrigsten Score auf der linken und dem höchsten Score auf der rechten Seite. Nehmen wir an, die *Entscheidungsgrenze* läge beim Pfeil in der Mitte (zwischen den zwei 5en): Sie erhalten dann vier richtig Positive (echte 5en) auf der rechten Seite des Schwellenwerts und einen falsch Positiven (eine 6).

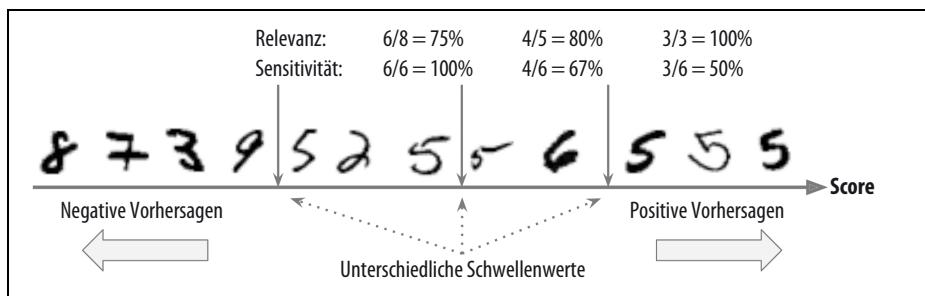


Abbildung 3-3: Schwellenwert bei der Entscheidung und Wechselbeziehung zwischen Relevanz und Sensitivität

Mit diesem Schwellenwert läge die Relevanz bei 80% (vier von fünf). Allerdings entdeckt der Klassifikator nur vier der insgesamt sechs vorhandenen 5en, daher

liegt die Sensitivität bei 67% (vier von sechs). Erhöhen wir nun den Schwellenwert (bis zum Pfeil auf der rechten Seite), würde der falsch Positive (die 6) zu einem richtig Negativen werden. Damit würde sich die Relevanz erhöhen (auf 100%), aber im Gegenzug würde ein richtig Positiver zu einem falsch Negativen werden. Damit würde die Sensitivität auf 50% sinken. Umgekehrt führt Absenken des Schwellenwerts zu einer höheren Sensitivität und geringerer Relevanz.

In Scikit-Learn können Sie den Schwellenwert nicht direkt festlegen, aber die zur Vorhersage verwendeten Scores der Entscheidungsfunktion sind verfügbar. Anstatt die Methode `predict()` eines Klassifikators aufzurufen, können Sie die Methode `decision_function()` verwenden, die für jeden Datenpunkt einen Score liefert, auf dessen Grundlage Sie Vorhersagen mit einem beliebigen Schwellenwert treffen können:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

Der `SGDClassifier` verwendet als Schwellenwert 0, sodass der obige Code das gleiche Ergebnis wie die Methode `predict()` liefert (nämlich `True`). Erhöhen wir nun diesen Schwellenwert:

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

Dies bestätigt uns, dass ein Erhöhen des Schwellenwerts die Sensitivität verringert. Das Bild enthält tatsächlich eine 5, was der Klassifikator bei einem Schwellenwert von 0 auch korrekt erkennt. Er scheitert aber, wenn wir den Schwellenwert auf 200000 erhöhen.

Wie sollen wir uns also für einen Schwellenwert entscheiden? Dazu müssen Sie zunächst wieder die Scores sämtlicher Datenpunkte im Trainingsdatensatz mit der Funktion `cross_val_predict()` ermitteln, diesmal aber mit der Angabe, dass Sie die Entscheidungswerte anstelle der Vorhersagen erhalten möchten:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Mit diesen Scores können Sie Relevanz und Sensitivität für alle möglichen Schwellenwerte mithilfe der Funktion `precision_recall_curve()` berechnen:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Als letzten Schritt verwenden Sie Matplotlib, um Relevanz und Sensitivität als Funktion des Schwellenwerts darzustellen (Abbildung 3-4):

```

def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Relevanz")
    plt.plot(thresholds, recalls[:-1], "g-", label="Sensitivität")
    plt.xlabel("Schwellenwert")
    plt.legend(loc='center left')
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()

```

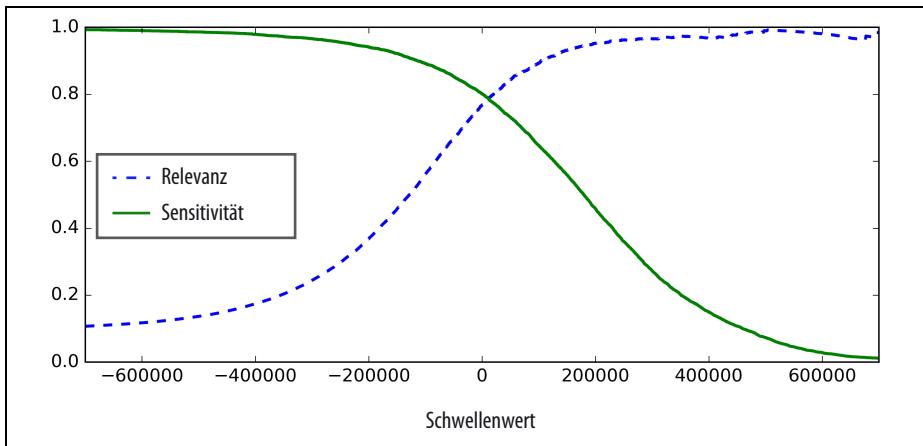


Abbildung 3-4: Relevanz und Sensitivität über dem Schwellenwert zur Entscheidung



Sie fragen sich vielleicht, warum die Verlaufskurve der Relevanz in Abbildung 3-4 unregelmäßiger ist als die der Sensitivität. Das liegt daran, dass die Relevanz manchmal beim Erhöhen des Schwellenwerts sinkt (auch wenn sie im Allgemeinen steigt). Um dies nachzuvollziehen, betrachten Sie noch einmal Abbildung 3-3. Wenn Sie beim Schwellenwert in der Mitte beginnen und sich nur eine Ziffer nach rechts bewegen, sinkt die Relevanz von $\frac{4}{5}$ (80%) auf $\frac{3}{4}$ (75%). Andererseits kann die Sensitivität nur sinken, wenn sich der Schwellenwert erhöht, was die glattere Form der Kurve erklärt.

Nun können wir einen Schwellenwert auswählen, der das für unsere Aufgabe optimale Verhältnis von Relevanz und Sensitivität liefert. Alternativ dazu ist ein Kompromiss zwischen Relevanz und Sensitivität auch möglich, indem Sie die Relevanz gegen die Sensitivität plotten, wie in Abbildung 3-5 gezeigt.

Wie Sie sehen, fällt die Relevanz etwa bei einer Sensitivität von 80 % scharf ab. Vermutlich sollten Sie einen Kompromiss zwischen Relevanz und Sensitivität kurz vor diesem Abfall auswählen – beispielsweise bei einer Sensitivität von etwa 60 %. Aber natürlich hängt das auch von Ihrem Projekt ab.

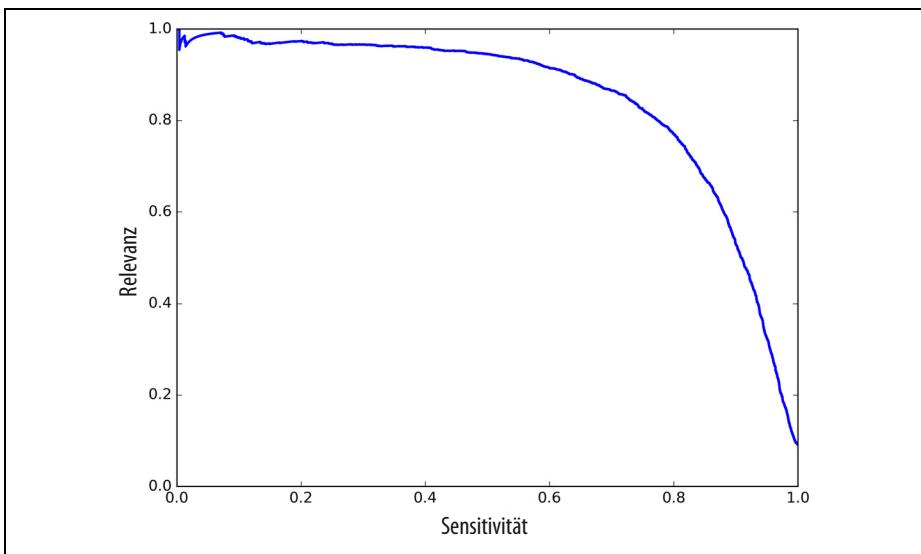


Abbildung 3-5: Relevanz gegen Sensitivität

Nehmen wir an, Sie möchten eine Relevanz von 90% erhalten. Sie schauen im ersten Diagramm nach (zoomen ein wenig hinein) und finden heraus, dass Sie einen Schwellenwert von etwa 70000 benötigen. Um Vorhersagen zu treffen (vorerst noch auf den Trainingsdaten), führen Sie anstelle der Methode `predict()` des Klassifikators den folgenden Code aus:

```
y_train_pred_90 = (y_scores > 70000)
```

Überprüfen wir Relevanz und Sensitivität dieser Vorhersagen:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.8998702983138781
>>> recall_score(y_train_5, y_train_pred_90)
0.63991883416343853
```

Großartig, Sie haben nun einen Klassifikator mit einer Relevanz von 90% (Sie sind zumindest nah genug dran)! Wie Sie sehen, ist es recht einfach, einen Klassifikator mit einer praktisch beliebigen Relevanz zu erstellen: Sie setzen einfach den Schwellenwert hoch genug und sind fertig. Moment einmal. Ein Klassifikator mit einer hohen Relevanz ist nicht sehr nützlich, wenn dessen Sensitivität zu niedrig ist!



Wenn jemand sagt »Lass uns eine Relevanz von 99% erreichen«, sollten Sie fragen »Bei welcher Sensitivität?«

Die ROC-Kurve

Die ROC-Kurve (*Receiver Operating Characteristic*) ist ein weiteres verbreitetes Hilfsmittel bei der binären Klassifikation. Sie ist der Relevanz-/Sensitivitäts-Kurve

sehr ähnlich, aber anstatt die Relevanz gegen die Sensitivität aufzutragen, zeigt die ROC-Kurve die *Richtig-positiv-Rate* (TNR) (ein anderer Name für Sensitivität) gegen die *Falsch-positiv-Rate* (FPR). Die FPR ist der Anteil negativer Datenpunkte, die fälschlicherweise als positiv eingestuft werden. Sie ist eins minus der *Richtig-negativ-Rate*, dem Anteil der korrekt als negativ eingestuften Datenpunkte. Die TNR wird auch als *Spezifität* bezeichnet. Also wird bei der ROC-Kurve die *Sensitivität* gegen die *1-Spezifität* aufgetragen.

Um eine ROC-Kurve zu plotten, müssen Sie zunächst mit der Funktion `roc_curve()` die TPR und FPR bei unterschiedlichen Schwellenwerten berechnen:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Anschließend plotten Sie mit Matplotlib die FPR gegen die TPR. Der folgende Code stellt die Abbildung in Abbildung 3-6 her:

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('Falsch-positiv-Rate')
    plt.ylabel('Richtig-positiv-Rate')

plot_roc_curve(fpr, tpr)
plt.show()
```

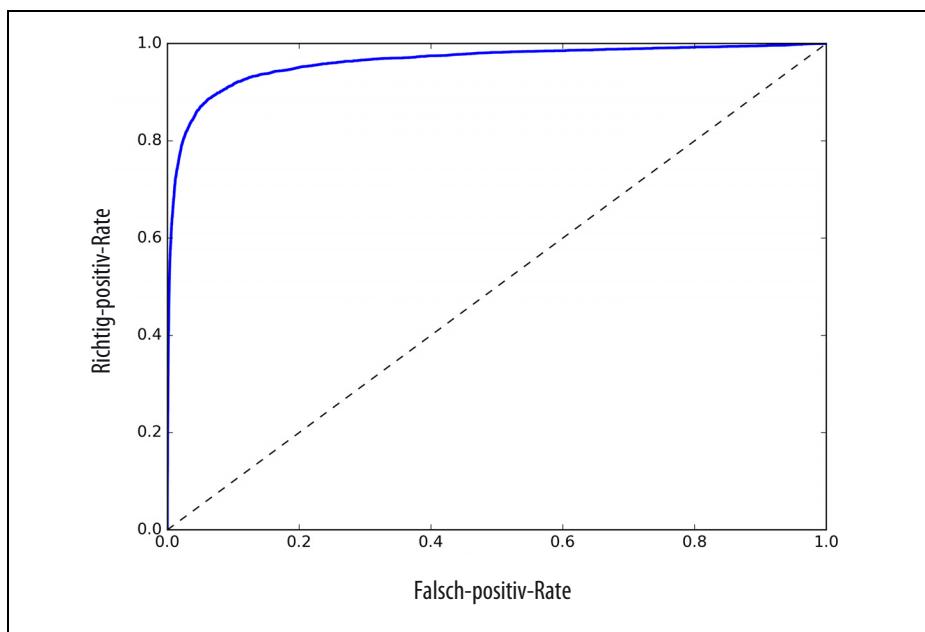


Abbildung 3-6: ROC-Kurve

Auch hier gilt es, einen Kompromiss zu schließen: Je höher die Sensitivität (TPR), desto mehr falsch Positive (FPR) liefert der Klassifikator. Die gestrichelte Linie stellt die ROC-Kurve eines völlig zufälligen Klassifikators dar; ein guter Klassifikator entfernt sich so weit wie möglich von dieser Linie (in Richtung der linken oberen Ecke).

Klassifikatoren lassen sich vergleichen, indem man die *Area under the Curve* (AUC) bestimmt. Ein perfekter Klassifikator hat eine *ROC AUC* von genau 1, ein völlig zufälliger dagegen hat eine *ROC AUC* von 0.5. In Scikit-Learn gibt es eine Funktion zum Berechnen der *ROC AUC*:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.97061072797174941
```



Da die ROC-Kurve der Relevanz-/Sensitivitäts-Kurve (oder auch PR-Kurve, engl. *precision/recall*) recht ähnlich ist, fragen Sie sich vielleicht, welche Sie denn nun benutzen sollen. Als Faustregel sollten Sie die PR-Kurve immer dann bevorzugen, wenn die positive Kategorie selten ist oder wenn die falsch Positiven wichtiger als die falsch Negativen sind. Andernfalls verwenden Sie die ROC-Kurve. Beim Betrachten der obigen ROC-Kurve (und des ROC-AUC-Scores) könnten Sie denken, dass der Klassifikator wirklich gut ist. Aber das liegt vor allem daran, dass es nur wenige Positive (5en) im Vergleich zu den Negativen (Nicht-5en) gibt. Die PR-Kurve dagegen macht deutlich, dass unser Klassifikator noch Luft nach oben hat (die Kurve sollte näher an der oberen rechten Ecke liegen).

Trainieren wir nun einen `RandomForestClassifier` und vergleichen wir dessen ROC-Kurve und ROC-AUC-Score mit dem `SGDClassifier`. Zunächst benötigen wir die Scores für jeden Datenpunkt im Trainingsdatensatz. Allerdings besitzt der `RandomForestClassifier` aufgrund seiner Funktionsweise (siehe Kapitel 7) keine Methode `decision_function()`. Stattdessen besitzt er die Methode `predict_proba()`. Klassifikatoren in Scikit-Learn haben im Allgemeinen entweder die eine oder die andere. Die Methode `predict_proba()` liefert ein Array mit einer Zeile pro Datenpunkt und einer Spalte pro Kategorie. Es enthält die Wahrscheinlichkeiten, dass ein bestimmter Datenpunkt einer bestimmten Kategorie angehört (z.B. eine 70%ige Chance, dass das Bild eine 5 darstellt):

```
from sklearn.ensemble import RandomForestClassifier  
  
forest_clf = RandomForestClassifier(random_state=42)  
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,  
                                     method="predict_proba")
```

Um aber eine ROC-Kurve zu zeichnen, benötigen Sie Scores, keine Wahrscheinlichkeiten. Eine einfache Lösung ist, die Wahrscheinlichkeit der positiven Kategorie als Score zu verwenden:

```

y_scores_forest = y_probas_forest[:, 1]    # score = Wahrsch.
der positiven Kategorie
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)

```

Nun sind Sie soweit, die ROC-Kurve zu plotten. Dabei ist es hilfreich, zum Vergleich auch die erste ROC-Kurve zu plotten (Abbildung 3-7):

```

plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()

```

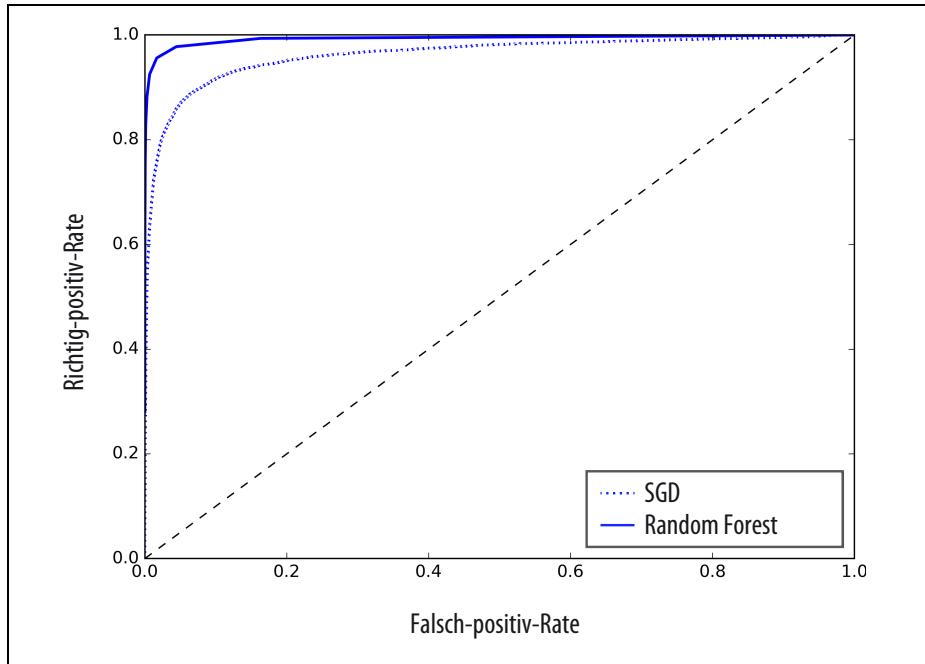


Abbildung 3-7: Vergleich zweier ROC-Kurven

Wie in Abbildung 3-7 gezeigt, sieht die Kurve für den RandomForestClassifier viel besser aus als die für den SGDClassifier: Sie kommt deutlich näher an die linke obere Ecke heran. Dementsprechend ist auch der ROC-AUC-Score deutlich besser:

```

>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291

```

Versuchen Sie, Relevanz und Sensitivität zu berechnen: Sie sollten eine Relevanz von 98.5% und eine Sensitivität von 82.8% erhalten. Gar nicht schlecht!

Sie wissen nun hoffentlich, wie Sie binäre Klassifikatoren trainieren, ein für Ihre Aufgabe geeignetes Qualitätsmaß auswählen, Ihre Klassifikatoren mithilfe einer Kreuzvalidierung auswerten, einen Ihren Anforderungen entsprechenden Kompromiss zwischen Relevanz und Sensitivität finden und unterschiedliche Modelle über

ROC-Kurven und ROC-AUC-Scores vergleichen können. Versuchen wir als Nächstes, mehr als nur die 5en zu erkennen.

Klassifikatoren mit mehreren Kategorien

Während binäre Klassifikatoren zwischen zwei Kategorien unterscheiden, können *Klassifikatoren mit mehreren Kategorien* (auch *multinomiale Klassifikatoren*) mehr als zwei Kategorien unterscheiden.

Einige Algorithmen (wie die Random-Forest-Klassifikatoren oder naive Bayes-Klassifikatoren) sind in der Lage, direkt mehrere Kategorien zu berücksichtigen. Andere (wie Support-Vector-Machine-Klassifikatoren oder lineare Klassifikatoren) sind stets binäre Klassifikatoren. Es gibt jedoch einige Strategien, mit denen sich mit mehreren binären Klassifikatoren mehrere Kategorien zuordnen lassen.

Sie könnten beispielsweise ein System zum Einteilen der Bilder von Ziffern in zehn Kategorien (von 0 bis 9) entwickeln, indem Sie zehn binäre Klassifikatoren trainieren, einen für jede Ziffer (also einen 0-Detektor, einen 1-Detektor, einen 2-Detektor und so weiter). Wenn Sie anschließend ein Bild klassifizieren möchten, ermitteln Sie für jeden dieser Klassifikatoren den Entscheidungs-Score und wählen die Kategorie aus, deren Klassifikator den höchsten Score lieferte. Dies bezeichnet man als *One-versus-All*-(OvA-)Strategie (oder *One-versus-the-Rest*).

Eine weitere Strategie ist, einen binären Klassifikator pro Zahlenpaar zu trainieren: einen zum Unterscheiden von 0 und 1, einen weiteren zum Unterscheiden von 0 und 2, einen für 1 und 2 und so weiter. Dies bezeichnet man als *One-versus-One*-(OvO-)Strategie. Wenn es N Kategorien gibt, müssen Sie $N \times (N - 1) / 2$ Klassifikatoren trainieren. Bei der MNIST-Aufgabe müsste man also 45 binäre Klassifikatoren trainieren! Wenn Sie ein Bild klassifizieren möchten, müssten Sie das Bild alle 45 Klassifikatoren durchlaufen lassen und prüfen, welche Kategorie am häufigsten vorkommt. Der Hauptvorteil von OvO ist, dass jeder Klassifikator nur auf dem Teil der Trainingsdaten mit den beiden zu unterscheidenden Kategorien trainiert werden muss.

Einige Algorithmen (wie Support-Vector-Machine-Klassifikatoren) skalieren schlecht mit der Größe des Trainingsdatensatzes. Bei diesen ist also OvO zu bevorzugen, weil sich viele Klassifikatoren mit kleinen Trainingsdatensätzen schneller trainieren lassen als wenige Klassifikatoren auf großen Trainingsdatensätzen. Bei den meisten binären Klassifikationsalgorithmen ist jedoch OvA vorzuziehen.

Scikit-Learn erkennt, wenn Sie einen binären Klassifikationsalgorithmus für eine Aufgabe mit mehreren Kategorien einsetzen, und verwendet automatisch OvA (außer bei SVM-Klassifikatoren, dort wird OvO eingesetzt). Probieren wir dies mit dem SGDClassifier aus:

```
>>> sgd_clf.fit(X_train, y_train) # y_train, nicht y_train_5  
>>> sgd_clf.predict([some_digit])  
array([ 5.])
```

Das war leicht! Der Code trainiert den `SGDClassifier` auf den Trainingsdaten mit den ursprünglichen Zielkategorien von 0 bis 9 (`y_train`) anstatt mit den Zielkategorien 5-gegen-alle (`y_train_5`). Anschließend wird eine Vorhersage getroffen (in diesem Fall eine richtige). Im Hintergrund trainiert Scikit-Learn zehn binäre Klassifikatoren, ermittelt deren Entscheidungswerte für das Bild und wählt die Kategorie mit dem höchsten Wert aus.

Um nachzuweisen, dass dies tatsächlich der Fall ist, können Sie die Methode `decision_function()` aufrufen. Statt einem Score pro Datenpunkt erhalten Sie nun zehn Werte, einen pro Kategorie:

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-311402.62954431, -363517.28355739, -446449.5306454 ,
       -183226.61023518, -414337.15339485,  161855.74572176,
       -452576.39616343, -471957.14962573, -518542.33997148,
       -536774.63961222]])
```

Der höchste Wert ist in der Tat derjenige für die Kategorie 5:

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> sgd_clf.classes_[5]
5.0
```



Beim Trainieren eines Klassifikators wird die Liste der Zielkategorien im Attribut `classes_` nach Werten sortiert gespeichert. In diesem Fall entspricht der Index jeder Kategorie im Array `classes_` bequemerweise der Kategorie selbst (beispielsweise ist die Kategorie beim Index 5 auch die Kategorie 5), aber normalerweise ist dies nicht so.

Wenn Sie Scikit-Learn dazu bringen möchten, das One-versus-One- oder One-versus-All-Verfahren zu verwenden, verwenden Sie dazu die Klassen `OneVsOneClassifier` bzw. `OneVsRestClassifier`. Erstellen Sie einfach eine Instanz, der Sie im Konstruktor einen binären Klassifikator übergeben. Der folgende Code erzeugt beispielsweise einen Klassifikator mit mehreren Kategorien mithilfe der OvO-Strategie und einem `SGDClassifier`:

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

Einen `RandomForestClassifier` zu verwenden ist genauso einfach:

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

Diesmal musste Scikit-Learn weder OvA noch OvO anwenden, da ein Random-Forest-Klassifikatoren Datenpunkte direkt mehreren Kategorien zuordnen können. Sie können `predict_proba()` aufrufen, um eine Liste der vom Klassifikator ermittelten Wahrscheinlichkeiten für jeden Datenpunkt und jede Kategorie zu erhalten:

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ]])
```

Sie sehen auch, dass der Klassifikator sich seiner Vorhersage sehr sicher ist: Die 0.8 beim 5. Index im Array bedeutet, dass das Modell das Bild mit 80%iger Wahrscheinlichkeit für eine 5 hält. Es denkt außerdem, dass das Bild stattdessen auch eine 0 oder 3 sein könnte (mit einer Wahrscheinlichkeit von jeweils 10%).

Diese Klassifikatoren sollten wir nun natürlich auch auswerten. Wie gewöhnlich setzen wir die Kreuzvalidierung ein. Beurteilen wir die Genauigkeit des SGDClassifer mit der Funktion `cross_val_score()`:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187,  0.84899245,  0.86652998])
```

In sämtlichen Test-Folds erhalten wir mehr als 84%. Mit einem zufälligen Klassifikator hätten wir eine Genauigkeit von 10% erhalten. Dies ist also kein schlechtes Ergebnis. Es geht aber noch viel besser. Beispielsweise erhöht ein einfaches Skalieren der Eingabedaten (wie in Kapitel 2 besprochen) die Genauigkeit auf über 90%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([ 0.91011798,  0.90874544,  0.906636 ])
```

Fehleranalyse

In einem echten Projekt würden Sie an dieser Stelle natürlich die Schritte auf der Checkliste für Machine-Learning-Projekte abarbeiten (siehe Anhang B): Optionen zur Datenaufarbeitung abwägen, mehrere Modelle ausprobieren, die besten in eine engere Auswahl ziehen, deren Hyperparameter mit `GridSearchCV` optimieren und so viel wie möglich automatisieren, wie Sie es im vorigen Kapitel getan haben. Wir nehmen an dieser Stelle an, dass Sie ein vielversprechendes Modell gefunden haben und nach Verbesserungsmöglichkeiten suchen. Eine Möglichkeit ist, die Arten von Fehlern zu untersuchen, die das Modell begeht.

Sie können sich zunächst die Konfusionsmatrix ansehen. Dazu müssen Sie als Erstes über die Funktion `cross_val_predict()` Vorhersagen treffen und anschließend die bereits gezeigte Funktion `confusion_matrix()` aufrufen:

```

>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5725,     3,   24,    9,   10,   49,   50,   10,   39,    4],
       [  2, 6493,   43,   25,    7,   40,    5,   10, 109,    8],
       [ 51,   41, 5321,  104,   89,   26,   87,   60, 166,   13],
       [ 47,   46, 141, 5342,    1, 231,   40,   50, 141,  92],
       [ 19,   29,   41,   10, 5366,    9,   56,   37,   86, 189],
       [ 73,   45,   36, 193,   64, 4582, 111,   30, 193,  94],
       [ 29,   34,   44,    2,   42,   85, 5627,   10,   45,    0],
       [ 25,   24,   74,   32,   54,   12,    6, 5787,   15, 236],
       [ 52, 161,   73, 156,   10, 163,   61,   25, 5027, 123],
       [ 43,   35,   26,   92, 178,   28,    2, 223,   82, 5240]])

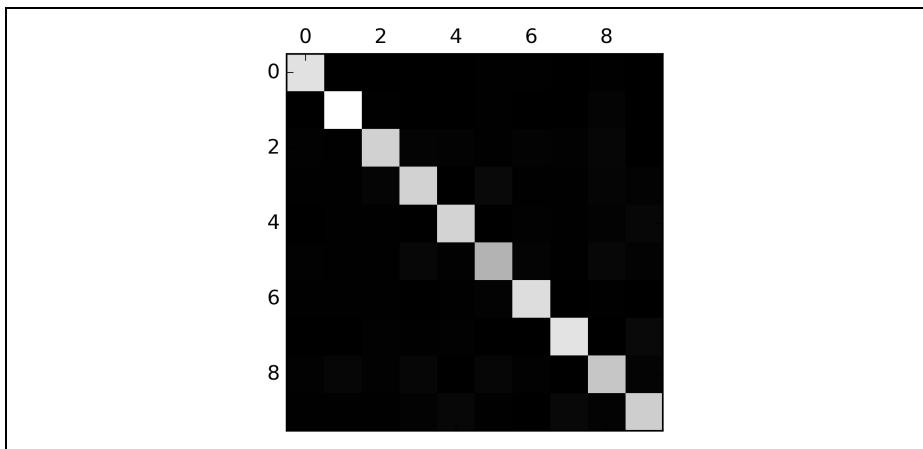
```

Dies sind eine Menge Zahlen. Es ist oft bequemer, sich eine Konfusionsmatrix über die Matplotlib-Funktion `matshow()` als Diagramm anzeigen zu lassen:

```

plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()

```



Diese Konfusionsmatrix sieht recht gut aus, da die meisten Bilder auf der Hauptdiagonalen liegen. Dies bedeutet, dass sie korrekt zugeordnet wurden. Die 5en sehen etwas dunkler als die anderen Ziffern aus. Dies könnte bedeuten, dass es weniger Bilder von 5en im Datensatz gibt oder dass der Klassifikator bei den 5en nicht so gut wie bei den übrigen Ziffern abschneidet. Es lässt sich zeigen, dass hier beides der Fall ist.

Heben wir im Diagramm nun die Fehler hervor. Zunächst müssen Sie jeden Wert in der Konfusionsmatrix durch die Anzahl der Bilder in der entsprechenden Kategorie teilen, sodass Sie den Anteil der Fehler anstatt der absoluten Anzahl Fehler vergleichen können (Letzteres würde die häufigen Kategorien übertrieben schlecht aussehen lassen):

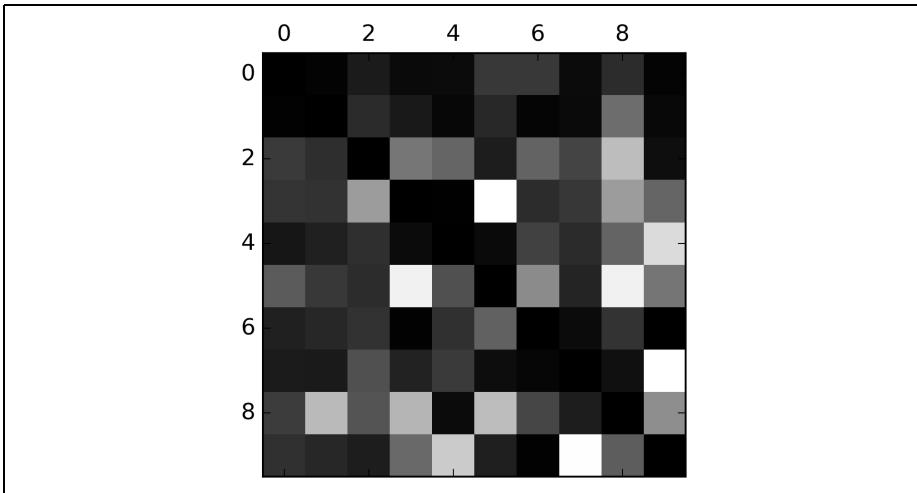
```

row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

```

Nun füllen wir die Diagonale mit Nullen auf, um nur die Fehler zu betrachten, und plotten das Ergebnis:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



Nun sehen Sie deutlich, welche Arten Fehler der Klassifikator begeht. Wie gesagt, stehen die Zeilen für die tatsächlichen Kategorien und Spalten für die vorhergesagten. Die Spalten der Kategorien 8 und 9 sind recht hell, es wurden also recht viele Bilder fälschlicherweise als 8 oder 9 zugeordnet. In ähnlicher Weise sind auch die Zeilen 8 und 9 recht hell, beide Ziffern wurden also oft mit anderen Ziffern verwechselt. Umgekehrt sind einige Zeilen recht dunkel, z.B. Zeile 1. Dies bedeutet, dass die meisten 1en korrekt erkannt wurden (einige wurden für 8en gehalten, aber das war es auch schon). Beachten Sie, dass die Fehler nicht perfekt symmetrisch sind, es gibt mehr 5en, die als 8en vorhergesagt wurden als umgekehrt.

Eine Analyse der Konfusionsmatrix gibt Ihnen häufig Aufschluss darüber, wie sich Ihr Klassifikator verbessern lässt. Aus diesem Diagramm sehen Sie, dass sich Ihre Mühe auf das Verbessern der Klassifikation der 8 und 9 und das Verwechseln der 3 und 5 konzentrieren sollte. Sie könnten also versuchen, zusätzliche Trainingsdaten für diese Ziffern zu sammeln oder neue Merkmale zu ermitteln, die dem Klassifikator helfen können – beispielsweise ein Algorithmus, der die geschlossenen Schleifen zählt (eine 8 hat zwei, eine 6 hat eine, eine 5 keine). Oder Sie könnten die Bilder vorverarbeiten (z.B. mit Scikit-Image, Pillow oder OpenCV), um vorhandene Muster wie die Schleifen besser hervorzuheben.

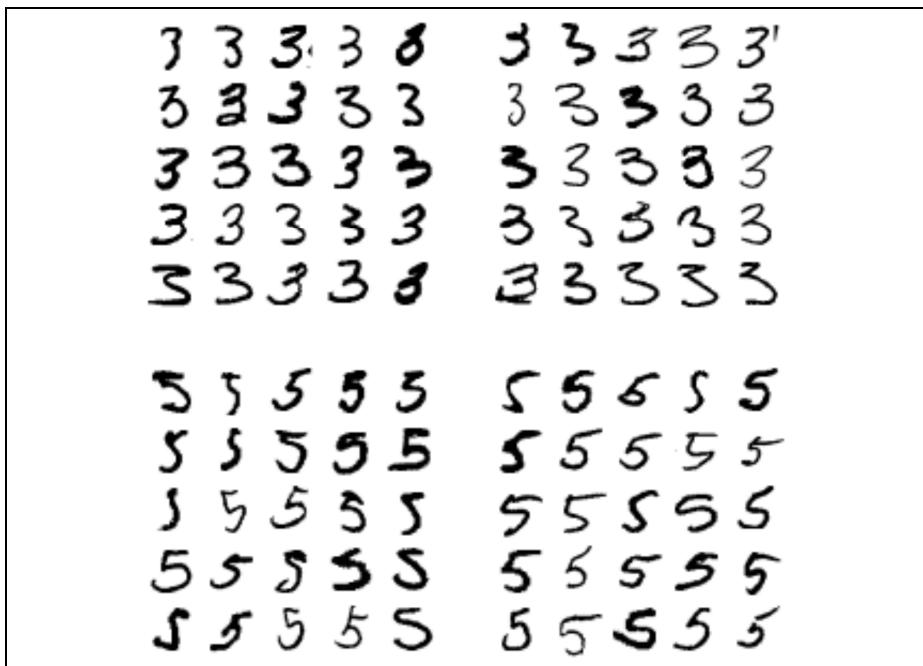
Das Analysieren einzelner Fehler hilft auch dabei zu erkennen, was Ihr Klassifikator eigentlich tut und warum er scheitert. Dies ist aber schwieriger und zeitlich aufwendiger. Zum Beispiel könnten wir Beispiele von 3en und 5en plotten:

```

cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()

```



Die zwei 5-x-5-Blöcke auf der linken Seite zeigen als 3en klassifizierte Ziffern und die zwei 5-x-5-Blöcke auf der rechten Seite zeigen als 5en klassifizierte Ziffern. Einige der falsch zugeordneten Ziffern (z.B. in den Blöcken links unten und rechts oben) sind so unleserlich, dass sogar ein Mensch bei der Zuordnung Schwierigkeiten hätte (z.B. sieht die 5 in der 8. Zeile und der 1. Spalte wirklich wie eine 3 aus).³ Es ist schwer nachzuvollziehen, warum der Klassifikator diese Fehler gemacht hat.⁴

-
- 3 Die Funktion `plot_digits()` verwendet die Matplotlib-Funktion `imshow()`, Details finden Sie im Jupyter Notebook zu diesem Kapitel)
 - 4 Bedenken Sie aber, dass unser Gehirn ein ausgezeichnetes System zur Mustererkennung ist und unser Gesichtssinn eine Menge komplexer Vorverarbeitung vornimmt, bevor Information in unser Bewusstsein vordringt. Deshalb bedeutet der Umstand, dass etwas leicht aussieht, nicht gleichzeitig auch, dass es leicht ist.

Der Grund ist, dass wir einen einfachen `SGDClassifier` verwendet haben, ein lineares Modell. Dieses ordnet in jeder Kategorie ein Gewicht pro Pixel zu und zählt bei einem neuen Bild einfach nur die gewichteten Intensitäten der Pixel zusammen, um einen Score für jede Kategorie zu berechnen. Da sich die 3en und 5en nur um einige Pixel unterscheiden, kommt das Modell bei diesen Ziffern leicht durcheinander.

Der Hauptunterschied zwischen 3en und 5en ist die Stellung der kurzen Linie, welche die obere Linie mit dem unteren Bogen verbindet. Wenn Sie eine 3 zeichnen und diese Verbindungsleitung ein Stück weiter links platzieren, könnte der Klassifikator das Bild leicht für eine 5 halten und umgekehrt. Anders gesagt reagiert unser Klassifikator sehr sensibel auf Verschiebungen und Rotationen des Bilds. Ein Möglichkeit, der Verwechslung von 3 und 5 entgegenzuwirken, wäre daher, die Bilder so vorzuverarbeiten, dass sie alle zentriert und wenig gedreht sind. Vermutlich ließen sich auf diese Weise auch weitere Fehler vermeiden.

Klassifikation mit mehreren Labels

Bisher wurde jeder Datenpunkt stets genau einer Kategorie zugeordnet. In einigen Fällen kann es sein, dass Sie Ihren Klassifikator mehrere Kategorien für jeden Datenpunkt ausgeben lassen möchten. Betrachten Sie beispielsweise einen Klassifikator zur Gesichtserkennung: Was sollte dieser tun, wenn er mehrere Personen im gleichen Bild findet? Natürlich sollte er jeder erkannten Person ein Label zuordnen. Sagen wir einmal, der Klassifikator sei auf drei Gesichtern trainiert worden: Alice, Bob und Charlie. Wenn er anschließend ein Bild von Alice und Bob gezeigt bekommt, sollte er `[1, 0, 1]` ausgeben (was bedeutet: »Alice ja, Bob nein, Charlie ja«). Solch einen Klassifikator, der mehrere binäre Labels ausgibt, nennt man ein *Klassifikationssystem mit mehreren Labels*.

Wir beschäftigen uns an dieser Stelle noch nicht mit Gesichtserkennung. Stattdessen betrachten wir zur Veranschaulichung ein einfacheres Beispiel:

```
from sklearn.neighbors import KNeighborsClassifier  
  
y_train_large = (y_train >= 7)  
y_train_odd = (y_train % 2 == 1)  
y_multilabel = np.c_[y_train_large, y_train_odd]  
  
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train, y_multilabel)
```

Dieses Codebeispiel erstellt das Array `y_multilabel` mit zwei Labels als Ziel für jedes Ziffernbild: Das erste besagt, ob die Ziffer groß ist (7, 8 oder 9), und das zweite, ob sie ungerade ist. Die folgenden Zeilen erstellen eine Instanz von `KNeighborsClassifier` (diese unterstützt die Klassifikation mehrerer Labels, was nicht alle Klassifikatoren tun). Wir trainieren diesen mit dem Array mit mehreren Zielwerten. Nun können Sie eine Vorhersage vornehmen und bemerken, dass zwei Labels ausgegeben werden:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]], dtype=bool)
```

Und das Ergebnis ist richtig! Die Ziffer 5 ist tatsächlich nicht groß (False) und ungerade (True).

Es gibt viele Möglichkeiten, einen Klassifikator mit mehreren Labels auszuwerten, und die Wahl der richtigen Metrik hängt von Ihrem Projekt ab. Ein Ansatz ist beispielsweise, den F_1 -Score für jedes Label einzeln zu bestimmen (oder eine andere der oben besprochenen binären Klassifikationsmetriken) und anschließend einfach deren Mittelwert zu berechnen. Der folgende Code berechnet den mittleren F_1 -Score über sämtliche Labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average='macro')
0.96845540180280221
```

Dabei nehmen wir an, dass alle Labels gleich wichtig sind, was nicht der Fall sein muss. Wenn Sie zum Beispiel viel mehr Bilder von Alice als von Bob oder Charlie haben, sollten Sie dem Score eines Klassifikators bei Alice ein höheres Gewicht verleihen. Eine einfache Möglichkeit ist dabei, jedem Label ein Gewicht entsprechend der Anzahl Datenpunkte mit diesem Label (dem *Support*) zu verleihen. Um dies zu erreichen, setzen Sie im obigen Codebeispiel `average="weighted"`.⁵

Klassifikation mit mehreren Ausgaben

Die letzte Art hier betrachteter Klassifikationsaufgaben nennt man *Klassifikation mit mehreren Kategorien und mehreren Ausgaben* (oder einfach *Klassifikation mit mehreren Ausgaben*). Sie ist eine Verallgemeinerung der Klassifikation mit mehreren Labels, wobei jedes Label mehrere Kategorien beinhalten kann (also mehr als zwei mögliche Werte haben kann).

Um dies zu veranschaulichen, erstellen wir ein System, das Rauschen aus Bildern entfernt. Als Eingabe erhält es ein verrausches Bild einer Ziffer und wird (hoffentlich) ein sauberes Bild einer Ziffer als Array von Pixelintensitäten ähnlich den MNIST-Bildern ausgeben. Beachten Sie dabei, dass die Ausgabe des Klassifikators mehrere Labels aufweist (ein Label pro Bildpunkt) und jedes Label mehrere Werte annehmen kann (Pixelintensitäten von 0 bis 255). Daher ist dies ein Beispiel für ein Klassifikationssystem mit mehreren Ausgaben.



Die Unterscheidung zwischen Klassifikation und Regression ist wie in diesem Beispiel manchmal unscharf. Es lässt sich argumentieren, dass die Vorhersage der Intensität von Pixeln mehr mit Regression als mit Klassifikation zu tun hat. Außerdem sind Systeme mit mehreren Ausgaben nicht auf Klassifikationsaufgaben

⁵ Scikit-Learn stellt einige weitere Optionen zur Mittelwertbildung und Metriken bei mehreren Labels zur Verfügung; schlagen Sie bitte die Details in der Dokumentation nach.

beschränkt; sie könnten sogar ein System entwickeln, das mehrere Labels pro Datenpunkt ausgibt, darunter Kategorien und Werte.

Erstellen wir zunächst Trainings- und Testdatensätze, indem wir den Intensitäten der Pixel in den MNIST-Bildern mit der Funktion `randint()` aus NumPy Rauschen hinzufügen. Die Zielbilder sind dann die Originalbilder:

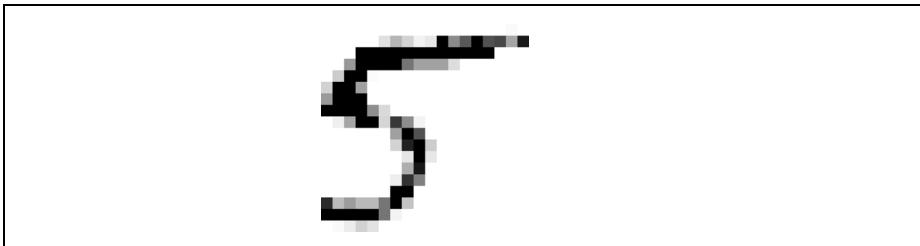
```
noise = rnd.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = rnd.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Schauen wir uns ein Bild aus dem Testdatensatz an (ja, wir schnüffeln an dieser Stelle in den Testdaten herum, Sie sollten an dieser Stelle entrüstet seufzen):



Auf der linken Seite ist das verrauschte Eingabebild, und auf der rechten Seite das saubere Zielbild. Trainieren wir nun den Klassifikator, um dieses Bild zu säubern:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



Wir sind nah genug am Ziel! Damit beenden wir unsere Tour der Klassifikationsverfahren. Sie wissen nun hoffentlich, wie Sie Metriken für Klassifikationsaufgaben auswählen, einen guten Kompromiss zwischen Relevanz und Sensitivität eingehen, Klassifikatoren vergleichen und ganz allgemein gute Systeme für unterschiedliche Klassifikationsaufgaben konstruieren.

Übungsaufgaben

1. Versuchen Sie, einen Klassifikator für den MNIST-Datensatz zu entwickeln, der auf den Testdaten eine Genauigkeit von mehr als 97% erzielt. Hinweis: Der `KNeighborsClassifier` funktioniert bei dieser Aufgabe recht gut; Sie müssen aber geeignete Hyperparameter finden (probieren Sie eine Gittersuche auf den Hyperparametern `weights` und `n_neighbors`).
2. Schreiben Sie eine Funktion, mit der sich ein MNIST-Bild in jeder Richtung (links, rechts, oben, unten) um ein Pixel verschieben lässt.⁶ Erstellen Sie anschließend aus sämtlichen Bildern im Trainingsdatensatz vier verschobene Kopien (eine pro Richtung) und fügen Sie diese den Trainingsdaten hinzu. Trainieren Sie anschließend Ihr bestes Modell auf dem so erweiterten Trainingsdatensatz und messen Sie die Genauigkeit auf den Testdaten. Sie sollten beobachten, dass das Modell noch besser wird! Diese Technik, einen Trainingsdatensatz künstlich zu vergrößern, nennt man *Data Augmentation* oder *Erweitern des Trainingsdatensatzes*.
3. Versuchen Sie sich am *Titanic*-Datensatz. Ein geeigneter Ort, dies zu probieren, ist Kaggle (<https://www.kaggle.com/c/titanic>).
4. Entwickeln Sie einen Spamfilter (eine fortgeschrittene Übung):
 - Laden Sie Beispiele für Spam und Ham aus dem öffentlichen Datensatz von Apache SpamAssassin (<http://spamassassin.apache.org/old/publiccorpus/>) herunter.
 - Entpacken Sie die Datensätze und machen Sie sich mit dem Format der Daten vertraut.
 - Unterteilen Sie die Datensätze in Trainings- und Testdaten.
 - Schreiben Sie eine Pipeline zur Vorverarbeitung der Daten, um jede E-Mail in einen Merkmalsvektor zu überführen. Ihre Pipeline sollte eine E-Mail in einen (dünn besetzten) Vektor transformieren, der das Vorhandensein jedes möglichen Worts beinhaltet. Wenn beispielsweise sämtliche E-Mails nur die vier Wörter »Hello«, »how«, »are«, »you«, enthalten, würde aus der E-Mail »Hello you Hello Hello you« der Vektor [1, 0, 0, 1] entstehen ([»Hello« ist vorhanden, »how« und »are« nicht, »you« ist vorhanden]) oder [3, 0, 0, 2], falls Sie die Häufigkeit jedes Worts auszählen möchten.
 - Sie können Ihrer Pipeline Hyperparameter hinzufügen, um einzustellen, ob der Header der E-Mails verworfen werden soll oder nicht. Sie können jede E-Mail in Kleinbuchstaben umwandeln, Interpunktionszeichen entfernen, alle URLs durch »URL« ersetzen, alle Zahlen durch »NUMBER« ersetzen

⁶ Sie können dazu die Funktion `shift()` aus dem Modul `scipy.ndimage.interpolation` verwenden. Beispielsweise verschiebt `shift(image, [2, 1], cval=0)` das Bild um 2 Pixel nach unten und 1 Pixel nach rechts.

oder sogar *Stemming* einsetzen (also die Endungen von Wörtern entfernen; es gibt für diesen Zweck Python-Bibliotheken).

- Probieren Sie anschließend mehrere Klassifikatoren aus und prüfen Sie, ob Sie einen guten Spamfilter bauen können, der eine hohe Sensitivität und Relevanz aufweist.

Lösungen zu diesen Übungen finden Sie in den Jupyter Notebooks auf <https://github.com/ageron/handson-ml>.

KAPITEL 4

Trainieren von Modellen

Bisher haben wir Modelle zum Machine Learning und deren Trainingsalgorithmen mehr oder weniger als Blackbox behandelt. Wenn Sie sich mit den Übungen in den ersten Kapiteln beschäftigt haben, war es vielleicht überraschend für Sie, wie viel Sie erreichen können, ohne etwas über die Funktionsweise der Modelle zu wissen: Sie haben ein System zur Regression optimiert, Sie haben einen Klassifikator für Ziffern verbessert und sogar einen Spamfilter aufgebaut – alles ohne zu wissen, wie diese eigentlich funktionieren. Tatsächlich brauchen Sie in den meisten Fällen die Details der Implementierung nicht zu kennen.

Ein Grundverständnis der Funktionsweise der Modelle hilft Ihnen allerdings dabei, sich schnell auf ein geeignetes Modell, das richtige Trainingsverfahren und einen guten Satz Hyperparameter für Ihre Aufgabe einzuschließen. Die Hintergründe zu verstehen, hilft Ihnen auch bei der Fehlersuche und erlaubt eine effizientere Fehleranalyse. Schließlich sind die meisten in diesem Kapitel besprochenen Themen eine Voraussetzung für Verständnis, Aufbau und Training von neuronalen Netzen (mit denen wir uns in Teil II dieses Buchs befassen werden).

In diesem Kapitel betrachten wir Modelle zur linearen Regression, einem der einfachsten Modelle überhaupt. Wir werden zwei unterschiedliche Ansätze zum Trainieren diskutieren:

- Verwenden einer Gleichung mit »geschlossener Form«, die die für den Trainingsdatensatz idealen Modellparameter direkt berechnet (also die Modellparameter, die eine Kostenfunktion über die Trainingsdaten minimieren).
- Verwenden eines iterativen Optimierungsverfahrens, des Gradientenverfahrens (GD), bei dem die Modellparameter schrittweise angepasst werden, um die Kostenfunktion über die Trainingsdaten zu minimieren und dabei möglicherweise die gleichen Parameter wie beim ersten Ansatz zu erhalten. Wir werden einige Varianten des Gradientenverfahrens betrachten, die uns bei den neuronalen Netzen in Teil II wieder und wieder begegnen werden: das Batch-Gradientenverfahren, das Mini-Batch-Gradientenverfahren und das stochastische Gradientenverfahren.

Anschließend werden wir einen Blick auf die polynomische Regression werfen, ein komplexeres Modell, das sich auch für nichtlineare Daten eignet. Da es bei diesem Modell mehr Parameter als bei der linearen Regression gibt, ist es anfälliger für Overfitting der Trainingsdaten. Wir werden uns daher ansehen, wie sich eine Überanpassung mit Lernkurven erkennen lässt, und betrachten anschließend einige Techniken zur Regularisierung, mit denen sich die Gefahr einer Überanpassung an die Trainingsdaten senken lässt.

Schließlich werden wir zwei weitere Modelle anschauen, die häufig für Klassifikationsaufgaben eingesetzt werden: die logistische Regression und die Softmax-Regression.



Dieses Kapitel enthält einige mathematische Formeln, die Begriffe aus der linearen Algebra und Analysis verwenden. Um diese Formeln zu verstehen, müssen Sie wissen, was Vektoren und Matrizen sind, wie sich diese transponieren lassen, was ein Skalarprodukt ist, was eine inverse Matrix ist und was partielle Ableitungen sind. Wenn Sie mit diesen Begriffen nicht vertraut sind, gehen Sie bitte die als Jupyter Notebooks verfügbaren einführenden Tutorials zu linearer Algebra und Analysis in den Online-Materialien durch. Diejenigen mit einer ausgeprägten Mathe-Allergie sollten dieses Kapitel dennoch durchgehen und die Formeln überspringen; ich hoffe, der Text hilft Ihnen, einen Großteil der Begriffe zu verstehen.

Lineare Regression

In Kapitel 1, haben wir ein einfaches Regressionsmodell der Zufriedenheit mit dem Leben betrachtet: $Zufriedenheit = \theta_0 + \theta_1 \times BIP_pro_Kopf$.

Dieses Modell ist nichts weiter als eine lineare Funktion des Eingabewerts BIP_pro_Kopf . θ_0 und θ_1 sind die Parameter des Modells.

Allgemeiner formuliert, trifft ein lineares Modell eine Vorhersage, indem es eine gewichtete Summe der Eingabemerkmale berechnet und eine Konstante namens *Bias-Term* (oder *Achsenabschnitt*) hinzuaddiert, wie in Formel 4-1 zu sehen ist.

Formel 4-1: Lineares Regressionsmodell zur Vorhersage

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

- \hat{y} ist der vorhergesagte Wert.
- n ist die Anzahl Merkmale.
- x_i ist der i^{te} Wert des Merkmals.
- θ_j ist der j^{te} Modellparameter (inklusive des Bias-Terms θ_0 und den Gewichten der Merkmale $\theta_1, \theta_2, \dots, \theta_n$).

In Vektorschreibweise lässt sich dies deutlich kompakter ausdrücken, wie Sie in Formel 4-2 sehen.

Formel 4-2: Lineares Regressionsmodell zur Vorhersage (Vektorschreibweise)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- θ ist der *Parametervektor* des Modells mit Bias-Term und den θ_0 Gewichte der Merkmale θ_1 bis θ_n .
- θ^T ist die transponierte Form von θ (einem Vektor von Zeilen anstatt von Spalten).
- \mathbf{x} ist der *Merkmalsvektor* eines Datenpunkts mit den Werten x_0 bis x_n , wobei x_0 stets 1 beträgt.
- $\theta^T \cdot \mathbf{x}$ ist das Skalarprodukt von θ^T und \mathbf{x} .
- h_{θ} ist die Hypothesenfunktion unter Verwendung der Modellparameter θ .

Dies ist also ein lineares Regressionsmodell. Wie sollen wir dieses trainieren? Wir erinnern uns, dass wir beim Trainieren eines Modells dessen Parameter so einstellen, dass das Modell so gut wie möglich an die Trainingsdaten angepasst ist. Dazu benötigen wir zuerst ein Qualitätsmaß für die Anpassung des Modells an die Trainingsdaten. In Kapitel 2 haben wir gesehen, dass das häufigste Gütekriterium bei einem Regressionsmodell die Wurzel der mittleren quadratischen Abweichung oder der Root Mean Square Error (RMSE) (Formel 2-1) ist. Um ein lineares Regressionsmodell zu trainieren, müssen wir daher den Wert für θ finden, für den der RMSE minimal wird. In der Praxis ist es einfacher, die mittlere quadratische Abweichung anstelle des RMSE zu berechnen. Dabei erhalten wir das gleiche Ergebnis (weil ein Wert, der eine Funktion minimiert auch dessen Quadratwurzel minimiert).¹

Der mittlere quadratische Fehler (MSE) der Hypothese einer linearen Regression h_{θ} lässt sich auf dem Trainingsdatensatz X mithilfe von Formel 4-3 berechnen.

Formel 4-3: MSE-basierte Kostenfunktion für ein lineares Regressionsmodell

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

Ein Großteil der Notation wurde bereits in Kapitel 2 vorgestellt (siehe »Schreibweisen« auf Seite 38). Der einzige Unterschied ist, dass wir h_{θ} anstelle von h schreiben, um deutlich zu machen, dass das Modell durch den Vektor θ parametrisiert wird.

1 Lernverfahren versuchen häufig, eine andere Funktion als das eigentliche zur Auswertung des fertigen Modells verwendete Qualitätsmaß zu optimieren. Dies liegt meist an der einfacheren Berechenbarkeit, weil sich diese Funktion im Gegensatz zum Qualitätsmaß leichter differenzieren lässt oder weil wir beim Training zusätzliche Nebenbedingungen hinzufügen möchten. Letzteres werden wir bei der Regularisierung sehen.

Um die Notation zu vereinfachen, werden wir im Folgenden einfach nur $\text{MSE}(\theta)$ anstelle von $\text{MSE}(\mathbf{X}, h_\theta)$ schreiben.

Die Normalengleichung

Um einen Wert für θ zu finden, der die Kostenfunktion minimiert, gibt es eine Lösung mit geschlossener Form – anders ausgedrückt, eine mathematische Gleichung, die uns das Ergebnis direkt liefert. Diese wird auch als die *Normalengleichung* bezeichnet (Formel 4-4).²

Formel 4-4: Normalengleichung

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ ist der Wert von θ für den die Kostenfunktion minimal wird.
- \mathbf{y} ist ein Vektor der Zielwerte von $y^{(1)}$ bis $y^{(m)}$.

Erstellen wir einige annähernd lineare Daten, um diese Gleichung auszuprobieren (Abbildung 4-1):

```
import numpy as np  
  
X = 2 * np.random.rand(100, 1)  
y = 4 + 3 * X + np.random.randn(100, 1)
```

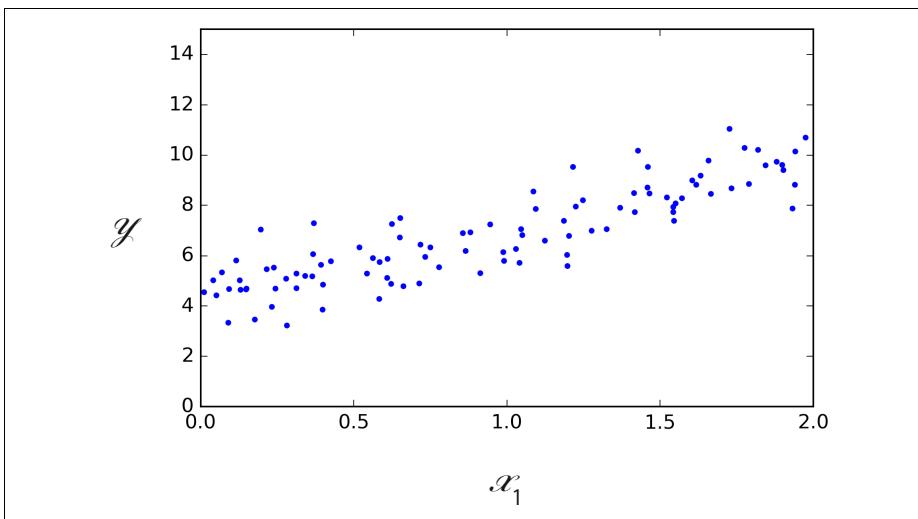


Abbildung 4-1: Zufällig generierter linearer Datensatz

² Der Nachweis, dass diese Gleichung tatsächlich den Wert von θ liefert, der die Kostenfunktion minimiert, liegt außerhalb des Rahmens dieses Buchs.

Berechnen wir nun $\hat{\theta}$ mithilfe der Normalengleichung. Wir verwenden die Funktion `inv()` aus dem in NumPy enthaltenen Modul für lineare Algebra (`np.linalg`), um die Inversion der Matrix und die Methode `dot()` zur Matrizenmultiplikation zu berechnen:

```
X_b = np.c_[np.ones((100, 1)), X] # füge x0 = 1 zu jedem Datenpunkt hinzu
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Die eigentliche Funktion zum Generieren der Daten ist $y = 4 + 3x_1 + \text{normalverteiltes Rauschen}$. Sehen wir einmal, was die Gleichung herausgefunden hat:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Wir hatten gehofft, $\theta_0 = 4$ und $\theta_1 = 3$ anstelle von $\theta_0 = 4.215$ und $\theta_1 = 2.770$ zu erhalten. Das ist nah dran, aber durch das Rauschen wurde es unmöglich, die Parameter der ursprünglichen Funktion exakt zu reproduzieren.

Nun können Sie mithilfe von $\hat{\theta}$ Vorhersagen treffen:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_
new] # füge x0 = 1 zu jedem Datenpunkt hinzu
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Nun plotten wir die Vorhersagen dieses Modells (Abbildung 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

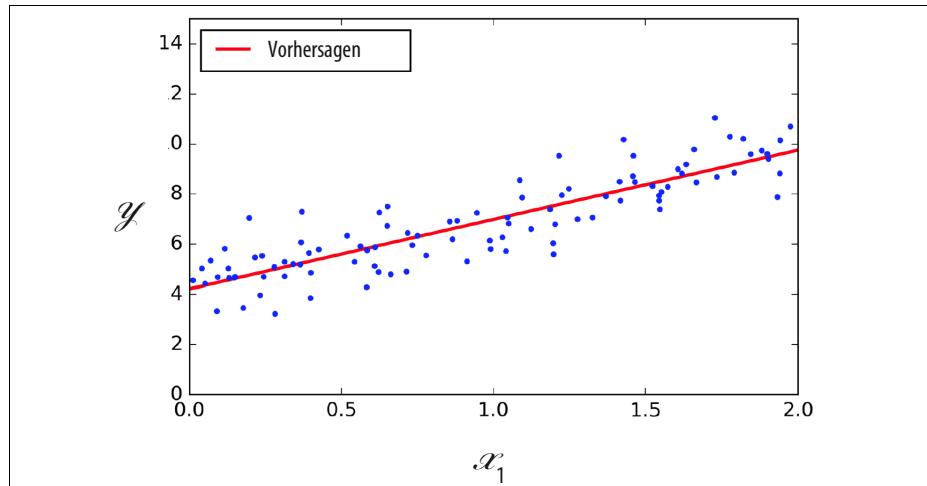


Abbildung 4-2: Vorhersagen des linearen Regressionsmodells

Eine äquivalente Implementierung mit Scikit-Learn sieht folgendermaßen aus:³

```
>>> from sklearn.linear_model import LinearRegression  
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([-4.21509616]), array([[ 2.77011339]]))  
>>> lin_reg.predict(X_new)  
array([[ -4.21509616],  
       [ 9.75532293]])
```

Komplexität der Berechnung

Die Normalengleichung berechnet die Inversion von $\mathbf{X}^T \cdot \mathbf{X}$, eine $n \times n$ -Matrix (wobei n die Anzahl Merkmale ist). Die *Komplexität der Berechnung* dieser Matrixinversion beträgt typischerweise etwa $O(n^{2.4})$ bis $O(n^3)$ (depending on the implementation). Anders ausgedrückt, wenn Sie die Anzahl Merkmale verdoppeln, erhöht sich die Rechenzeit etwa um den Faktor $2^{2.4} = 5.3$ bis $2^3 = 8$.



Die Normalengleichung wird sehr langsam, wenn die Anzahl Merkmale groß wird (z.B. 100000).

Als Vorteil ist zu nennen, dass sich die Gleichung linear in Bezug auf die Anzahl Datenpunkte im Trainingsdatensatz verhält (die Komplexität beträgt $O(m)$), sie behandelt große Datensätze also effizient, vorausgesetzt, sie passen in den Speicher. Wenn Sie Ihr lineares Regressionsmodell erst einmal trainiert haben (mithilfe der Normalengleichung oder einem anderen Algorithmus), sind die Vorhersagen sehr schnell: Die Komplexität der Berechnung verhält sich sowohl in Bezug auf die Anzahl vorherzusagender Datenpunkte als auch zur Anzahl Merkmale linear. Anders ausgedrückt dauert das Treffen einer Vorhersage für doppelt so viele Datenpunkte (oder doppelt so viele Merkmale) nur etwa doppelt so lange.

Wir werden uns nun völlig andere Verfahren zum Trainieren eines linearen Regressionsmodells ansehen, die sich besser für Fälle eignen, bei denen es eine große Anzahl Merkmale oder zu viele Trainingsdaten gibt, als dass sie sich im Speicher unterbringen ließen.

Das Gradientenverfahren

Das *Gradientenverfahren* ist ein sehr allgemeiner Algorithmus zur Optimierung, der optimale Lösungen für eine Vielzahl von Fragestellungen ermitteln kann. Der Grundgedanke beim Gradientenverfahren ist, die Parameter iterativ so zu verändern, dass eine Kostenfunktion minimiert wird.

³ Beachten Sie, dass Scikit-Learn den Bias-Term (`intercept_`) von den Gewichten getrennt ablegt (`coef_`).

Stellen Sie sich einmal vor, Sie hätten sich in dichtem Nebel in den Bergen verlaufen; Sie können nur die Neigung des Bodens unter Ihren Füßen fühlen. Um schnell ins Tal zu gelangen, wäre eine geeignete Strategie, sich stets in Richtung der steilsten Neigung bergab zu bewegen. Genau dies tut das Gradientenverfahren: Es berechnet den lokalen Gradienten der Fehlerfunktion in Abhängigkeit vom Parametervektor θ und bewegt sich in Richtung eines abfallenden Gradienten. Sobald der Gradient null wird, haben Sie ein Minimum gefunden!

Zu Beginn befüllen Sie θ mit Zufallszahlen (dies nennt man *zufällige Initialisierung*). Dann verbessern Sie die Parameter nach und nach in ganz kleinen Schritten, wobei Sie bei jedem Schritt versuchen, die Kostenfunktion zu senken (z.B. den MSE), bis der Algorithmus bei einem Minimum *konvergiert* (siehe Abbildung 4-3).

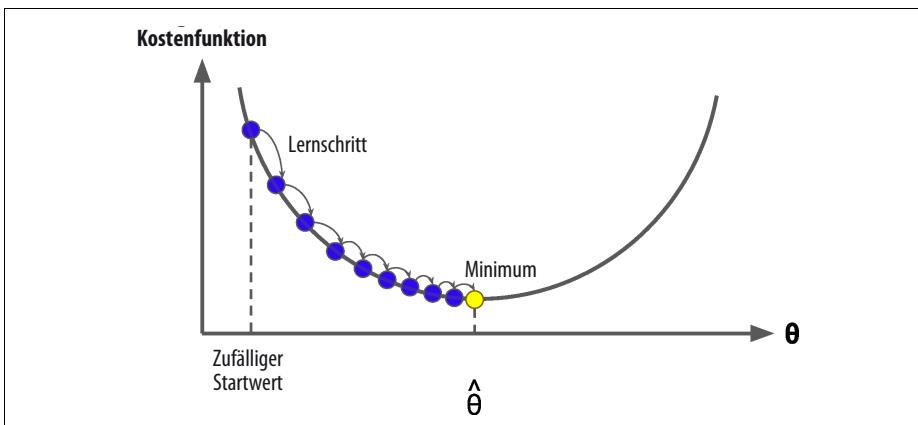


Abbildung 4-3: Gradientenverfahren

Ein wichtiger Parameter beim Gradientenverfahren ist die Größe der Schritte, die durch die *Lernrate*, einen Hyperparameter, festgelegt wird. Ist die Lernrate zu klein, muss der Algorithmus viele Iterationen durchlaufen, bevor er konvergiert. Das dauert natürlich sehr lange (siehe Abbildung 4-4).

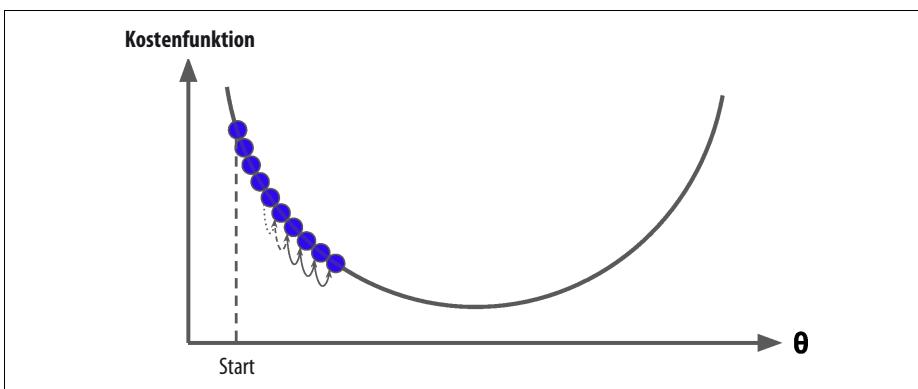


Abbildung 4-4: Zu geringe Lernrate

Wenn die Lernrate dagegen zu groß ist, kann es passieren, dass Sie die Täler überspringen und auf der anderen Seite landen, möglicherweise sogar höher als zuvor. Dadurch kann der Algorithmus divergieren, also immer größere Werte erzeugen und überhaupt keine gute Lösung finden (siehe Abbildung 4-5).

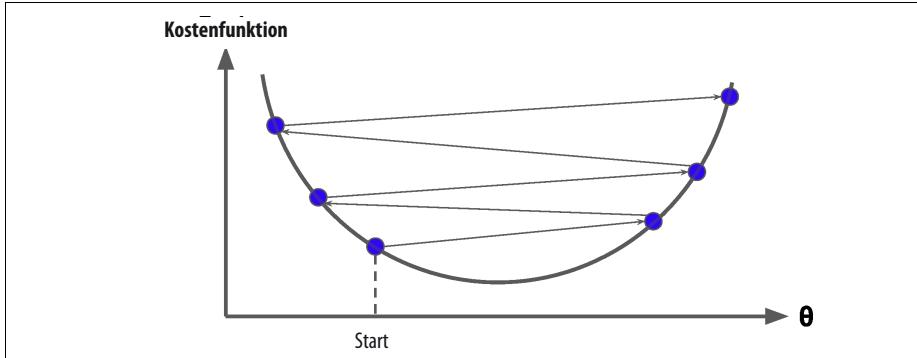


Abbildung 4-5: Zu große Lernrate

Schließlich sehen nicht alle Kostenfunktionen wie hübsche, regelmäßige Schüsseln aus. Es kann Täler, Grate, Plateaus und alle möglichen Arten unregelmäßiger Landschaften geben, die das Konvergieren beim Minimum deutlich erschweren. Abbildung 4-6 zeigt die zwei wichtigsten Herausforderungen beim Gradientenverfahren: Wenn die zufällige Initialisierung den Algorithmus auf der linken Seite startet, dann wird er bei einem *lokalem Minimum* konvergieren, das weniger gut als das *globale Minimum* ist. Wenn Sie auf der rechten Seite starten, dauert es sehr lange, das Plateau zu überqueren. Wenn Sie den Algorithmus zu früh beenden, werden Sie nie das globale Minimum erreichen.

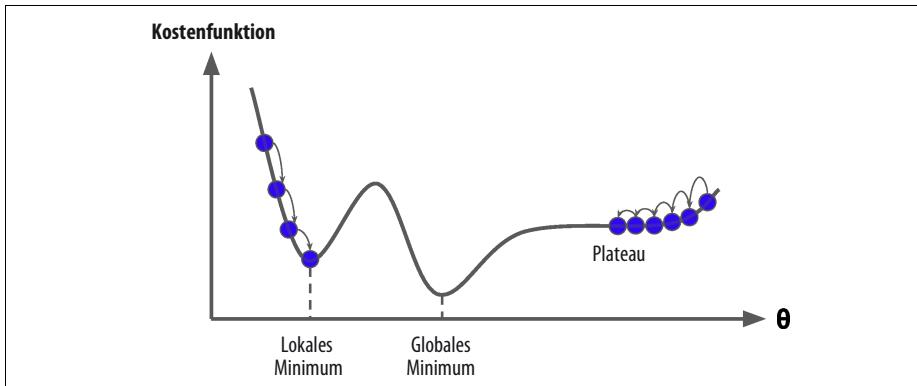


Abbildung 4-6: Fallstricke beim Gradientenverfahren

Glücklicherweise ist MSE als Kostenfunktion eines linearen Regressionsmodells eine *konvexe Funktion*. Das bedeutet, wenn Sie zwei beliebige Punkte auf der Kurve auswählen, schneidet die lineare Verbindung zwischen diesen beiden niemals die

Kurve. Das impliziert, dass es keine lokalen Minima gibt, nur ein globales Minimum. Sie ist auch eine stetige Funktion mit einer Steigung, die sich niemals abrupt ändert.⁴ Diese zwei Umstände haben eine wichtige Konsequenz: Mit dem Gradientenverfahren kann man sich dem globalen Minimum beliebig annähern (wenn Sie lange genug warten und die Lernrate nicht zu groß ist).

Die Kostenfunktion hat also die Form einer Schüssel. Wenn die Merkmale sehr unterschiedlich skaliert sind, kann es aber eine längliche Schüssel sein. Abbildung 4-7 illustriert das Gradientenverfahren für einen Trainingsdatensatz, bei dem die Merkmale 1 und 2 gleich skaliert sind (links), und für einen Trainingsdatensatz, bei dem die Beträge von Merkmal 1 sehr viel geringer sind als die von Merkmal 2 (rechts).⁵

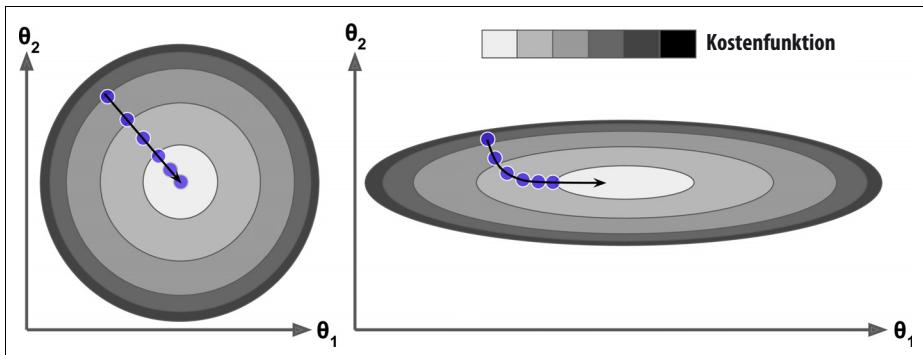


Abbildung 4-7: Gradientenverfahren mit und ohne Skalierung von Merkmalen

Wie Sie sehen, bewegt sich auf der linken Seite das Gradientenverfahren direkt auf das Minimum zu und erreicht es schnell. Auf der rechten Seite bewegt es sich zunächst beinahe orthogonal zur Richtung des globalen Minimums. Hier muss der Algorithmus ein mehr oder weniger flaches Tal komplett durchwandern. Auch hier wird das Minimum erreicht, dies dauert aber seine Zeit.



Wenn Sie das Gradientenverfahren verwenden, sollten Sie sicherstellen, dass sämtliche Merkmale ähnlich skaliert sind (z.B. über die Klasse `StandardScaler` in Scikit-Learn). Andernfalls wird deutlich mehr Zeit vergehen, bis der Algorithmus konvergiert.

Dieses Diagramm verdeutlicht außerdem, dass beim Trainieren eines Modells eine Kombination von Modellparametern gesucht wird, die eine Kostenfunktion (über die Trainingsdaten) minimieren. Es ist eine Suche im *Parameterraum* des Modells: Je mehr Parameter das Modell besitzt, desto mehr Dimensionen hat dieser Raum, und umso schwieriger ist die Suche: In einem 300-dimensionalen Heuhaufen nach

⁴ Genauer gesagt ist ihre Ableitung Lipschitz-stetig.

⁵ Da Merkmal 1 kleiner ist, ist eine stärkere Änderung von θ_1 nötig, um sich auf die Kostenfunktion auszuwirken. Deshalb ist die Schüssel entlang der θ_1 -Achse länglich.

einer Nadel zu suchen, ist viel komplizierter als in drei Dimensionen. Da die Kostenfunktion konvex ist, liegt die Nadel im Falle der linearen Regression glücklicherweise immer auf dem Boden einer Schüssel.

Batch-Gradientenverfahren

Um das Gradientenverfahren zu implementieren, müssen Sie den Gradienten der Kostenfunktion nach jedem Modellparameter θ_j berechnen. Anders ausgedrückt müssen Sie berechnen, wie stark sich die Kostenfunktion ändert, wenn Sie θ_j ein wenig verändern. Dies nennt man eine *partielle Ableitung*. Sie verhält sich wie die Frage »Wie ist die Neigung des Bergs unter meinen Füßen, wenn ich mich nach Osten wende?«, um anschließend die gleiche Frage nach Norden gerichtet zu stellen (ebenso bei allen anderen Dimensionen, falls Sie sich ein Universum mit mehr als drei Dimensionen vorstellen können). Formel 4-5 berechnet die partielle Ableitung der Kostenfunktion nach dem Parameter θ_j , was sich auch als

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) \text{ schreiben lässt.}$$

Formel 4-5: Partielle Ableitung der Kostenfunktion

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Anstatt partielle Ableitungen einzeln zu berechnen, können Sie Formel 4-6 verwenden, um alle zeitgleich zu berechnen. Der Gradientenvektor $\nabla_{\theta} \text{MSE}(\theta)$ enthält sämtliche partiellen Ableitungen der Kostenfunktion (eine für jeden Modellparameter).

Formel 4-6: Gradientenvektor der Kostenfunktion

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$



Beachten Sie, dass diese Formel bei jedem Schritt im Gradientenverfahren Berechnungen über den gesamten Trainingsdatensatz \mathbf{X} vornimmt. Daher bezeichnet man diesen Algorithmus auch als *Batch-Gradientenverfahren*: Dieses Verfahren verwendet bei jedem Schritt den gesamten Stapel Trainingsdaten und ist daher bei sehr großen Trainingsdatensätzen auffällig langsam (wir werden aber gleich einen viel schnelleren Algorithmus für das Gradientenverfahren kennenlernen). Das Gradientenverfahren skaliert dafür gut mit der Anzahl Merkmale; das Trainieren eines linearen Regressionsmodells mit Hunderttausenden Merkmalen ist mit dem Gradientenverfahren sehr viel schneller als mit der Normalengleichung.

Sobald Sie den Gradientenvektor ermittelt haben, der bergauf weist, gehen Sie einfach in die entgegengesetzte Richtung, um sich bergab zu bewegen. Dazu müssen Sie $\nabla_{\theta}\text{MSE}(\theta)$ von θ abziehen. An dieser Stelle kommt die Lernrate η ins Spiel:⁶ Multiplizieren Sie den Gradientenvektor mit η , um die Größe des Schritts bergab zu ermitteln (Formel 4-7).

Formel 4-7: Schritt im Gradientenverfahren

$$\theta^{(\text{nächster Schritt})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Betrachten wir eine schnelle Implementierung dieses Algorithmus:

```
eta = 0.1 # Lernrate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # zufällige Initialisierung

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

Das war nicht allzu schwer! Schauen wir uns einmal das berechnete theta an:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

Hey, das entspricht exakt dem von der Normalengleichung gefundenen Ergebnis! Das Gradientenverfahren hat perfekt funktioniert. Aber was wäre passiert, wenn wir eine andere Lernrate eta verwendet hätten? Abbildung 4-8 zeigt die ersten zehn Schritte im Gradientenverfahren mit drei unterschiedlichen Lernraten (die gestrichelte Linie steht für den Ausgangspunkt).

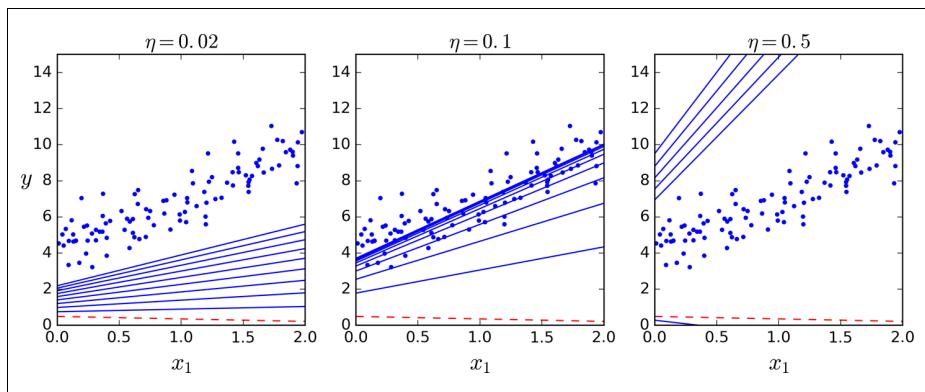


Abbildung 4-8: Gradientenverfahren mit unterschiedlichen Lernraten

⁶ Eta (η) ist der 7. Buchstabe im griechischen Alphabet.

Auf der linken Seite ist die Lernrate zu gering: Der Algorithmus findet zwar irgendwann eine Lösung, aber es wird lange dauern. In der Mitte sieht die Lernrate recht gut aus: Innerhalb weniger Iterationen ist der Algorithmus auf der Lösung konvergiert. Auf der rechten Seite ist die Lernrate zu hoch: Der Algorithmus divergiert, springt von einer Seite zur anderen und entfernt sich dabei immer weiter von der Lösung. Um eine geeignete Lernrate zu finden, können Sie eine Gittersuche durchführen (siehe Kapitel 2). Allerdings sollten Sie dabei die Anzahl der Iterationen begrenzen, sodass die Gittersuche Modelle eliminieren kann, bei denen das Konvergieren zu lange dauert.

Sie fragen sich vielleicht, wie man die Anzahl Iterationen bestimmen soll. Wenn diese zu gering ist, werden Sie beim Anhalten des Algorithmus noch immer weit von der optimalen Lösung entfernt sein. Ist sie aber zu hoch, verschwenden Sie Zeit, während sich die Modellparameter nicht mehr verändern. Eine einfache Lösung ist, die Anzahl Iterationen auf einen sehr großen Wert zu setzen, aber den Algorithmus anzuhalten, sobald der Gradientenvektor winzig klein wird – denn das passiert, wenn das Gradientenverfahren das Minimum (beinahe) erreicht hat. Der Gradientenvektor wird dann so winzig, wenn sein Betrag kleiner als eine sehr kleine Zahl x wird, was man auch als *Toleranz* bezeichnet.

Konvergenzrate

Wenn die Kostenfunktion konvex ist und ihre Steigung sich nicht abrupt ändert (wie es bei der MSE-Kostenfunktion der Fall ist), lässt sich nachweisen, dass das Batch-Gradientenverfahren mit einer gegebenen Lernrate eine *Konvergenzrate* von $O(\frac{1}{\text{Iterationen}})$ besitzt. Anders ausgedrückt, wenn Sie die Toleranz ϵ durch 10 teilen (um eine genauere Lösung erhalten), muss der Algorithmus etwa 10 Mal mehr Iterationen durchführen.

Stochastisches Gradientenverfahren

Das Hauptproblem beim Batch-Gradientenverfahren ist, dass es bei jedem Schritt den gesamten Trainingsdatensatz zum Berechnen der Gradienten verwendet, wodurch es bei großen Trainingsdatensätzen sehr langsam wird. Das andere Extrem ist das *stochastische Gradientenverfahren* (SGD), das bei jedem Schritt nur einen Datenpunkt zufällig auswählt und nur für diesen Punkt die Gradienten berechnet. Natürlich wird dadurch der Algorithmus viel schneller, da in jeder Iteration nur sehr wenige Daten verändert werden müssen. Damit ist das Trainieren auf riesigen Datensätzen möglich, da pro Iteration nur ein Datenpunkt verändert werden muss (SGD lässt sich auch als Out-of-Core-Algorithmus implementieren).⁷

⁷ Out-of-Core-Algorithmen werden in Kapitel 1 besprochen.

Andererseits ist dieser Algorithmus wegen seiner stochastischen (d.h. zufälligen) Arbeitsweise wesentlich unregelmäßiger als das Batch-Gradientenverfahren: Anstatt sanft zum Minimum hinabzugeilen, hüpfst die Kostenfunktion auf und ab und sinkt nur im Mittel. Mit der Zeit landet sie sehr nah am Minimum, springt dort aber weiter umher und kommt nie zur Ruhe (siehe Abbildung 4-9). Sobald der Algorithmus anhält, werden die endgültigen Parameter also gut, aber nicht optimal sein.

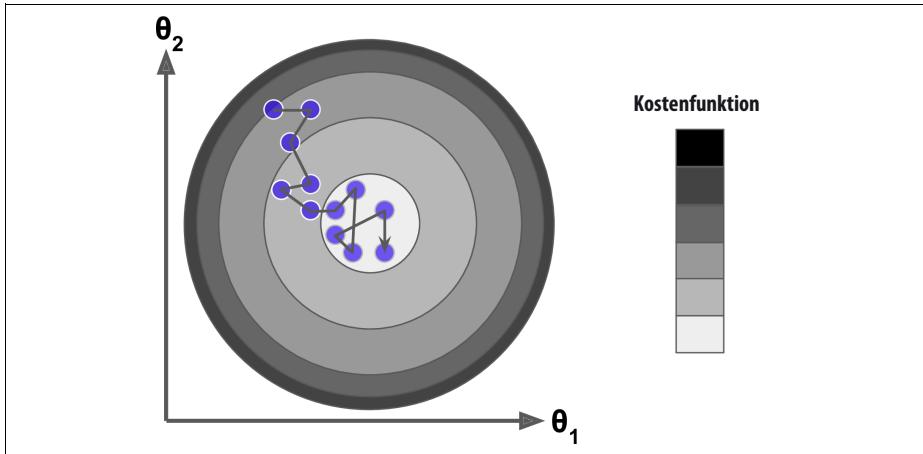


Abbildung 4-9: Stochastisches Gradientenverfahren

Bei einer sehr unregelmäßigen Kostenfunktion (wie in Abbildung 4-6) kann dies dem Algorithmus helfen, aus lokalen Minima wieder herauszuspringen. Daher hat das stochastische Gradientenverfahren im Vergleich zum Batch-Gradientenverfahren eine höhere Chance, das globale Minimum zu finden.

Die Zufälligkeit ist also gut, um lokalen Minima zu entfliehen, aber schlecht, da der Algorithmus beim Minimum nie zur Ruhe kommt. Eine Lösung dieses Dilemmas ist, die Lernrate schrittweise zu verringern. Die Schritte sind zu Beginn groß (was zu schnellen Fortschritten führt und beim Verlassen lokaler Minima hilft) und werden dann immer kleiner, sodass der Algorithmus beim globalen Minimum zur Ruhe kommt. Diesen Prozess bezeichnet man als *Simulated Annealing*, da es dem Ausglühen in der Metallurgie ähnelt, bei dem geschmolzenes Metall langsam abkühlt. Die Funktion zum Festlegen der Lernrate bezeichnet man als *Learning Schedule*. Wenn die Lernrate zu schnell reduziert wird, bleiben Sie in einem lokalen Minimum stecken oder sogar auf halber Strecke zum Minimum. Wird die Lernrate zu langsam gesenkt, springen Sie sehr lange um das Minimum herum und erhalten eine suboptimale Lösung, wenn Sie das Trainieren zu früh anhalten.

Im folgenden Codebeispiel ist das stochastische Gradientenverfahren mit einem einfachen Learning Schedule implementiert:

```
n_epochs = 50
t0, t1 = 5, 50 # Hyperparameter für den Learning Schedule
```

```

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # zufällige Initialisierung

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients

```

Standardmäßig iterieren wir in Runden mit je m Iterationen; jede Runde nennt man *Epoche*. Der Code für das Batch-Gradientenverfahren hat den gesamten Trainingsdatensatz 1000 Mal durchlaufen. Dieser Code durchläuft die Trainingsdaten nur 50 Mal und erzielt eine recht gute Lösung:

```

>>> theta
array([[ 4.21076011],
       [ 2.74856079]])

```

Abbildung 4-10 zeigt die ersten zehn Schritte beim Trainieren (achten Sie darauf, wie unregelmäßig die Schritte sind).

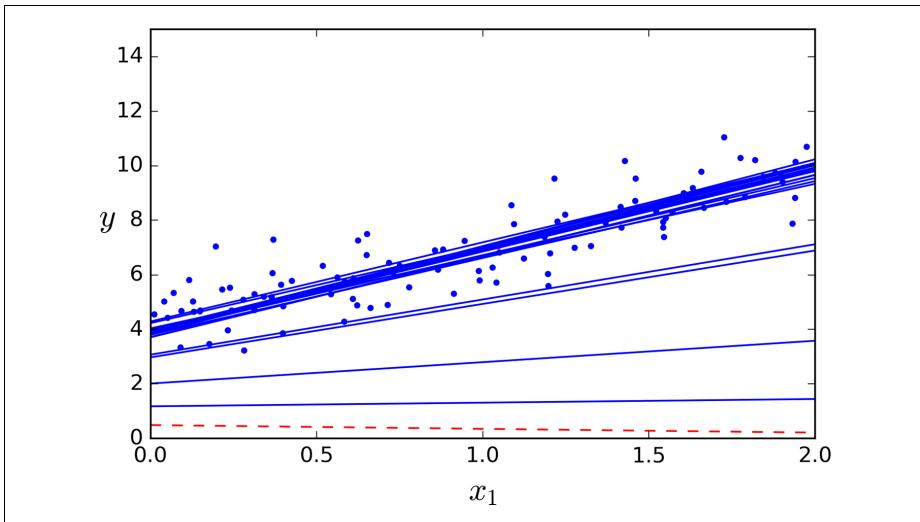


Abbildung 4-10: Die ersten zehn Schritte des stochastischen Gradientenverfahrens

Da die Datenpunkte zufällig ausgewählt werden, werden manche Datenpunkte innerhalb einer Epoche mehrmals selektiert, andere dagegen gar nicht. Wenn Sie sichergehen möchten, dass jeder Datenpunkt in jeder Epoche abgearbeitet wird, können Sie die Trainingsdaten durchmischen und dann Eintrag für Eintrag durch-

gehen. Anschließend mischen Sie die Daten erneut und so weiter. Allerdings konvergiert dieses Verfahren im Allgemeinen langsamer.

Um eine lineare Regression mit dem stochastischen Gradientenverfahren in Scikit-Learn durchzuführen, verwenden Sie die Klasse `SGDRegressor`, die den quadratischen Fehler als Kostenfunktion minimiert. Das folgende Codebeispiel führt 50 Epochen aus, beginnt mit einer Lernrate von 0.1 (`eta0=0.1`), verwendet den voreingestellten Learning Schedule (einen anderen als den oben vorgestellten) und keinerlei Regularisierung (`penalty=None`; Details dazu folgen in Kürze):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Die gefundene Lösung liegt wieder nah an der von der Normalengleichung gefundenen:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.16782089]), array([ 2.72603052]))
```

Mini-Batch-Gradientenverfahren

Als letzten Algorithmus unter den Gradientenverfahren sehen wir uns das *Mini-Batch Gradientenverfahren* an. Es ist recht einfach nachzuvollziehen, wenn Sie mit dem Batch- und dem stochastischen Gradientenverfahren vertraut sind: Anstatt die Gradienten bei jedem Schritt auf dem gesamten Trainingsdatensatz (wie beim Batch-Gradientenverfahren) oder nur auf einem Datenpunkt (wie beim stochastischen Gradientenverfahren) zu berechnen, berechnet das Mini-Batch-Gradientenverfahren die Gradienten auf kleinen, zufälligen Teilmengen, den *Mini-Batches*. Der Hauptvorteil des Mini-Batch-Gradientenverfahrens gegenüber dem stochastischen Verfahren ist, dass Sie die Leistung durch für Matrizenoperationen optimierte Hardware steigern können, besonders beim Verwenden von GPUs.

Die Fortschritte des Algorithmus im Parameterraum sind weniger abrupt als beim SGD, besonders bei größeren Mini-Batches. Daher wandert das Mini-Batch-Gradientenverfahren etwas näher um das Minimum herum als das SGD. Andererseits kann es schwieriger sein, lokale Minima zu verlassen (im Falle von Aufgaben, bei denen lokale Minima eine Rolle spielen; lineare Regression gehört nicht dazu). Abbildung 4-11 zeigt die Pfade durch den Parameterraum beim Trainieren mit den drei Algorithmen. Alle erreichen das Minimum, aber das Batch-Gradientenverfahren hält dort auch an, während sich sowohl das stochastische als auch das Mini-Batch-Gradientenverfahren weiter um das Minimum herum bewegen. Allerdings benötigt das Batch-Gradientenverfahren für jeden Schritt eine Menge Zeit, und auch das stochastische und das Mini-Batch-Gradientenverfahren würden mit einem guten Learning Schedule das Minimum erreichen.

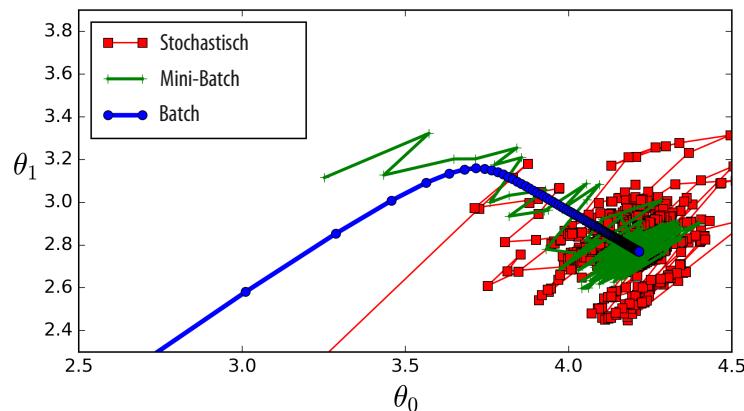


Abbildung 4-11: Pfade von Gradientenverfahren im Parameterraum

Vergleichen wir die bisher besprochenen Algorithmen zur linearen Regression⁸ (dabei ist m die Anzahl der Trainingsdatenpunkte und n die Anzahl der Merkmale); siehe Tabelle 4-1.

Tabelle 4-1: Vergleich von Algorithmen zur linearen Regression

Algorithmus	großes m	Out-of-Core unterstützt	großes n	Hyperparameter	Skalieren nötig	Scikit-Learn
Normalengleichung	schnell	Nein	langsam	0	Nein	LinearRegression
Batch-GD	langsam	Nein	schnell	2	Ja	k. A.
stochastisches GD	schnell	Ja	schnell	≥ 2	Ja	SGDRegressor
Mini-Batch GD	schnell	Ja	schnell	≥ 2	Ja	k. A.



Nach dem Trainieren gibt es kaum noch einen Unterschied: Alle diese Algorithmen führen zu sehr ähnlichen Modellen und treffen Vorhersagen in der gleichen Art und Weise.

Polynomiale Regression

Wie sieht es aus, wenn Ihre Daten komplexer als eine gerade Linie sind? Überraschenderweise können wir auch nichtlineare Daten mit einem linearen Modell fit-

⁸ Die Normalengleichung lässt sich nur zur linearen Regression einsetzen, die Algorithmen für das Gradientenverfahren lassen sich auch zum Trainieren vieler anderer Modelle einsetzen, wie wir noch sehen werden.

ten. Dazu können wir einfach Potenzen jedes Merkmals als neue Merkmale hinzufügen und dann ein lineare Modell auf diesem erweiterten Merkmalssatz trainieren. Diese Technik nennt man *polynomiale Regression*.

Betrachten wir ein Beispiel. Zunächst generieren wir einige nichtlineare Daten anhand einer einfachen *quadratischen Gleichung*⁹ (zuzüglich etwas Rauschen; siehe Abbildung 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

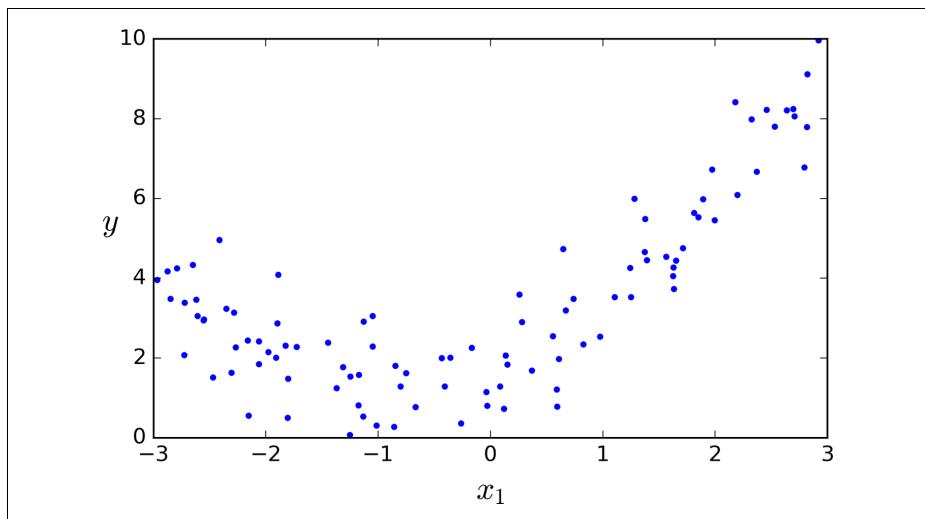


Abbildung 4-12: Generierte nichtlineare, verrauschte Daten

Offensichtlich lassen sich diese Daten nicht angemessen mit einer Geraden fitten. Daher verwenden wir die Scikit-Learn-Klasse `PolynomialFeatures`, um unsere Trainingsdaten zu transformieren und zu jedem Merkmal in den Trainingsdaten dessen Quadrat (das Polynom 2. Grades) als neues Merkmal hinzuzufügen (in diesem Fall gibt es nur ein Merkmal):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

X_{poly} enthält jetzt das ursprüngliche Merkmal aus X sowie das Quadrat dieses Merkmals. Nun können Sie ein mit `LinearRegression` erstelltes Modell auf diese erweiterten Trainingsdaten anwenden (Abbildung 4-13):

⁹ Eine quadratische Gleichung entspricht der Form $y = ax^2 + bx + c$.

```

>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))

```

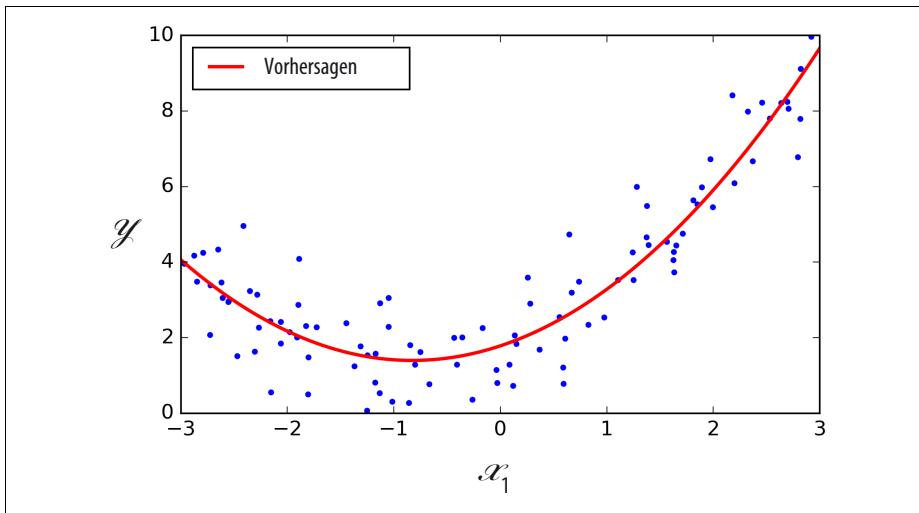


Abbildung 4-13: Vorhersagen eines polynomiellen Regressionsmodells

Nicht schlecht: Das Modell schätzt $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$, während die ursprüngliche Funktion $y = 0.5x_1^2 + 1.0x_1 + 2.0$ normalverteiltes Rauschen lautete.

Wenn es mehrere Merkmale gibt, ist die polynomiale Regression in der Lage, Wechselbeziehungen zwischen Merkmalen zu finden (was ein einfaches lineares Regressionsmodell nicht kann). Dies ist dadurch möglich, dass `PolynomialFeatures` bis zu einem bestimmten Grad sämtliche Kombinationen von Merkmalen hinzufügt. Wenn es beispielsweise zwei Merkmale a und b gibt, erzeugt `PolynomialFeatures` mit `degree=3` nicht nur die Merkmale a^2 , a^3 , b^2 und b^3 , sondern auch die Kombinationen ab , a^2b und ab^2 .



`PolynomialFeatures(degree=d)` transformiert ein Array mit n Merkmalen in ein Array mit $\frac{(n+d)!}{d! n!}$ Merkmalen, wobei $n!$ die Fakultät von n ist, also $1 \times 2 \times 3 \times \dots \times n$. Hüten Sie sich vor der kombinatorischen Explosion bei der Anzahl der Merkmale!

Lernkurven

Wenn Sie höhergradige polynomiale Regressionen durchführen, werden Sie die Trainingsdaten deutlich genauer fitten als mit einer gewöhnlichen linearen Regression. Beispielsweise wird in Abbildung 4-14 ein polynomielles Modell 300. Grades

auf die obigen Trainingsdaten angewandt und das Ergebnis mit einem gewöhnlichen linearen Modell und einem quadratischen Modell (einem Polynom 2. Grades) verglichen. Achten Sie darauf, wie das Polynom 3. Grades hin und her schwankt, um so nah wie möglich an die Trainingsdatenpunkte zu gelangen.

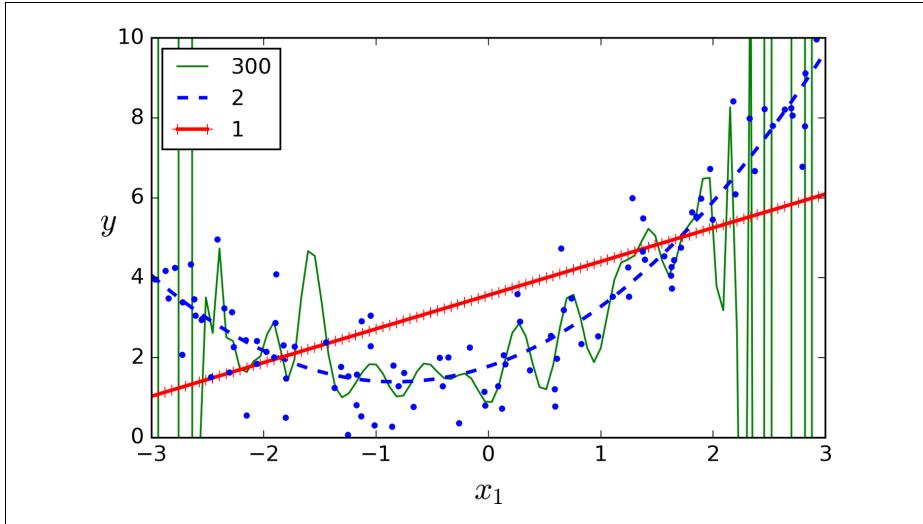


Abbildung 4-14: Regression mit einem höhergradigen Polynom

Natürlich liegt beim Modell mit der höhergradigen Regression sehr starkes Overfitting vor, während beim linearen Modell Underfitting vorliegt. In diesem Fall verallgemeinert das quadratische Modell am besten. Dies ist sinnvoll, weil ja auch den Daten ein quadratisches Modell zugrunde liegt, aber im Allgemeinen wissen Sie nicht, mit welcher Funktion die Daten generiert wurden. Wie also sollen Sie entscheiden, wie komplex Ihr Modell sein soll? Woher wissen Sie, ob Overfitting oder Underfitting der Daten vorliegt?

In Kapitel 2 haben Sie eine Kreuzvalidierung durchgeführt, um die Verallgemeinerungsleistung des Modells abzuschätzen. Wenn ein Modell auf den Trainingsdaten gut, aber bei der Kreuzvalidierung schlecht abschneidet, dann liegt in Ihrem Modell Overfitting vor. Wenn es bei beiden eine schlechte Leistung erbringt, ist es Underfitting. Auf diese Weise können Sie herausfinden, ob Ihr Modell zu einfach oder zu komplex ist.

Eine Alternative dazu ist, sich die *Lernkurven* anzusehen: Diese Diagramme zeigen die Leistung des Modells auf den Trainings- und den Validierungsdaten über der Größe des Trainingsdatensatzes. Um diese Plots zu erzeugen, trainieren Sie einfach das Modell mehrmals auf unterschiedlich großen Teilmengen des Trainingsdatensatzes. Der folgende Code definiert eine Funktion, die die Lernkurve eines Modells anhand der Trainingsdaten plottet:

```

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")

```

Schauen wir uns die Lernkurven des einfachen linearen Regressionsmodells an (einer Geraden; Abbildung 4-15):

```

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)

```

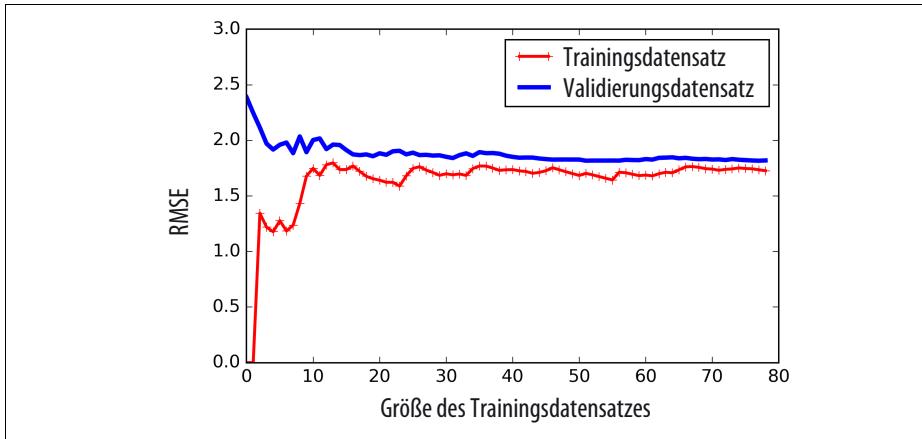


Abbildung 4-15: Lernkurven

An dieser Stelle sind einige Erklärungen angebracht. Zuerst betrachten wir die Voraussageleistung auf den Trainingsdaten: Wenn der Trainingsdatensatz nur aus ein oder zwei Datenpunkten besteht, kann das Modell diese perfekt fitten. Deshalb beginnt die Kurve bei null. Aber sobald neue Datenpunkte hinzukommen, wird die perfekte Anpassung unmöglich, weil die Daten Rauschen enthalten und weil sie überhaupt nicht linear sind. Daher steigt der Fehler auf den Trainingsdaten, bis er ein Plateau erreicht, bei dem zusätzliche Daten die durchschnittliche Abweichung weder verbessern noch verschlechtern. Schauen wir uns nun die Leistung des Modells auf den Validierungsdaten an. Wenn das Modell auf sehr wenigen Datenpunkten trainiert wird, kann es nicht anständig verallgemeinern, weswegen der Validierungsfehler zu Beginn recht groß ist. Sobald das Modell weitere Trainingsdaten kennenternt, sinkt der Validierungsfehler allmählich. Allerdings kann auch

hier eine Gerade irgendwann die Daten nicht gut modellieren, daher erreicht der Fehler ein Plateau in der Nähe der zweiten Kurve.

Diese Lernkurven sind für ein Modell typisch, bei dem Underfitting vorliegt. Beide Kurven erreichen ein Plateau; sie liegen nah beieinander und recht weit oben.



Bei Underfitting der Trainingsdaten verbessert sich Ihr Modell durch zusätzliche Trainingsdaten nicht. Sie benötigen ein komplexeres Modell oder müssen bessere Merkmale finden.

Betrachten wir nun die Lernkurven eines polynomiellen Modells 10. Grades auf den gleichen Daten (Abbildung 4-16):

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("lin_reg", LinearRegression()),
])
plot_learning_curves(polynomial_regression, X, y)
```

Diese Lernkurven erinnern ein wenig an die vorigen, es gibt aber zwei sehr wichtige Unterschiede:

- Der Fehler auf den Trainingsdaten ist viel geringer als beim Modell mit der linearen Regression.
- Es gibt eine Lücke zwischen den Kurven. Das bedeutet, dass das Modell auf den Trainingsdaten deutlich besser abschneidet als auf den Validierungsdaten. Dies ist die Handschrift eines Modells mit Overfitting. Wenn Sie allerdings einen deutlich größeren Datensatz verwendeten, würden sich die zwei Kurven weiter annähern.

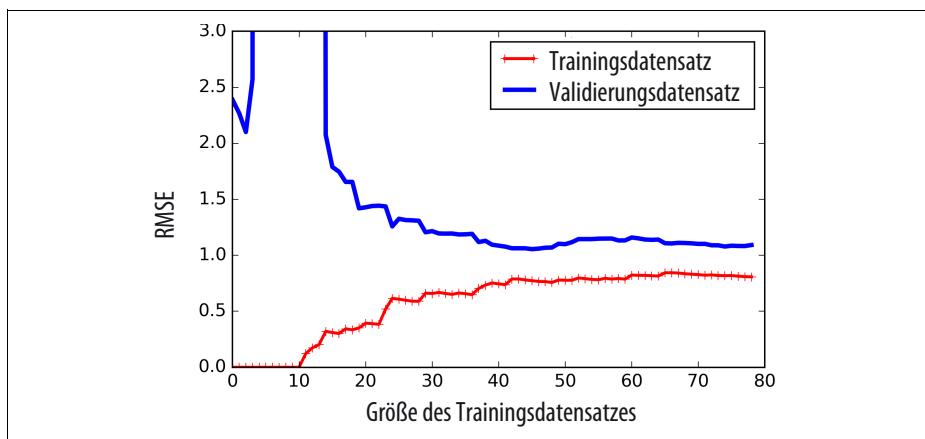


Abbildung 4-16: Lernkurven für ein polynomielles Modell



Eine Möglichkeit, ein Modell mit Overfitting zu verbessern, ist, so lange zusätzliche Trainingsdaten bereitzustellen, bis der Validierungsfehler den Trainingsfehler erreicht.

Das Gleichgewicht zwischen Bias und Varianz

Ein wichtiges theoretisches Ergebnis aus der Statistik und dem Machine Learning ist, dass sich der Verallgemeinerungsfehler eines Modells als Summe dreier sehr unterschiedlicher Fehler ausdrücken lässt:

Bias

Dieser Teil des Verallgemeinerungsfehlers wird durch falsche Annahmen verursacht, etwa die Annahme, dass die Daten linear sind, obwohl sie sich quadratisch verhalten. Ein Modell mit hohem Bias wird die Trainingsdaten vermutlich underfitten.¹⁰

Varianz

Dieser Teil kommt durch übermäßige Empfindlichkeit des Modells für kleine Variationen in den Trainingsdaten zustande. Ein Modell mit vielen Freiheitsgraden (wie etwa ein höhergradiges Polynom) hat vermutlich eine hohe Varianz und overfittet daher die Trainingsdaten leichter.

Nicht reduzierbare Fehler

Dieser Teil ist durch das Rauschen in den Daten selbst bedingt. Die einzige Möglichkeit, diesen Fehleranteil zu verringern, ist, die Daten zu säubern (z.B. die Datenquellen zu reparieren (wie etwa beschädigte Sensoren) oder Ausreißer zu erkennen und zu entfernen).

Das Steigern der Komplexität eines Modells erhöht meistens dessen Varianz und senkt dessen Bias. Umgekehrt erhöht eine geringere Komplexität des Modells dessen Bias und senkt die Varianz. Deshalb nennt man dies ein Gleichgewicht.

Regularisierte lineare Modelle

Wie wir bereits in den Kapiteln Kapitel 1 und Kapitel 2 gesehen haben, ist Regularisierung des Modells (es einzuschränken) eine sinnvolle Möglichkeit, um Overfitting zu vermeiden: Je weniger Freiheitsgrade das Modell hat, desto schwieriger wird es, die Daten zu overfitten. Beispielsweise lässt sich ein polynomielles Modell einfach regularisieren, indem man den Grad des Polynoms verringert.

Bei einem linearen Modell wird die Regularisierung normalerweise in Form von Nebenbedingungen auf den Gewichten des Modells umgesetzt. Wir werden nun drei unterschiedliche Arten von Nebenbedingungen betrachten: Ridge-Regression, Lasso-Regression und Elastic Net.

10 Diese Art Bias ist nicht mit dem Bias-Term linearer Modelle zu verwechseln.

Ridge-Regression

Die *Ridge-Regression* (auch *Regularisierung nach Tikhonov* genannt) ist eine regulierte Variante der linearen Regression: Zur Kostenfunktion wird der *Regularisierungsterm* $\alpha \sum_{i=1}^n \theta_i^2$ addiert. Dieser zwingt den Lernalgorithmus, nicht nur die Daten zu fitten, sondern auch die Gewichte des Modells so klein wie möglich zu halten. Ein Regularisierungsterm sollte nur beim Trainieren zur Kostenfunktion addiert werden. Ist das Modell erst einmal trainiert, sollten Sie die Vorhersageleistung des Modells mit dem nicht regularisierten Leistungsmaß evaluieren.



Es ist recht häufig, dass die Kostenfunktion beim Trainieren sich vom Qualitätsmaß beim Testen unterscheidet. Neben der Regularisierung ist ein weiterer Grund, dass eine Kostenfunktion beim Trainieren leicht optimierbare Ableitungen haben sollte, während das Leistungsmaß beim Testen möglichst nah am eigentlichen Ziel sein sollte. Ein gutes Beispiel hierfür ist ein Klassifikator, der mit einer Kostenfunktion wie (dem in Kürze besprochenen) Log Loss trainiert wird, aber mit Relevanz und Sensitivität evaluiert wird.

Der Hyperparameter α steuert, wie stark Sie das Modell regularisieren möchten. Bei $\alpha = 0$ entspricht die Ridge-Regression exakt der linearen Regression. Wenn α sehr groß ist, werden sämtliche Gewichte annähernd null, und es ergibt sich eine horizontale Linie durch den Mittelwert der Daten. Formel 4-8 zeigt die Kostenfunktion bei der Ridge-Regression.¹¹

Formel 4-8: Kostenfunktion bei der Ridge-Regression

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Der Bias-Term θ_0 ist nicht regularisiert (die Summe beginnt bei $i = 1$, nicht bei 0). Wenn wir \mathbf{w} als Gewichtsvektor der Merkmale definieren (θ_1 bis θ_n), so ist der Regularisierungsterm einfach gleich $\frac{1}{2} (\|\mathbf{w}\|_2)^2$, wobei $\|\cdot\|_2$ für die ℓ_2 -Norm des Gewichtsvektors steht.¹² Beim Gradientenverfahren, addieren Sie einfach $\alpha \mathbf{w}$ zum MSE-Gradientenvektor (Formel 4-8).



Es ist wichtig, die Daten zu skalieren (z.B. den StandardScaler zu verwenden), bevor Sie eine Ridge-Regression durchführen, da diese sensibel auf die Skala der Eingabemerkmale reagiert. Dies ist bei den meisten regularisierten Modellen der Fall.

¹¹ Die Notation $J(\theta)$ ist für Kostenfunktionen üblich, die keinen kurzen Namen besitzen; wir werden dieser Schreibweise häufiger im weiteren Verlauf des Buchs begegnen. Aus dem Kontext wird ersichtlich sein, um welche Kostenfunktion es geht.

¹² Normen werden in Kapitel 2 besprochen.

Abbildung 4-17 zeigt mehrere auf linearen Daten trainierte Ridge-Modelle mit unterschiedlichen Werten für α . Auf der linken Seite wurden einfache Ridge-Modelle verwendet, die zu linearen Vorhersagen führen. Auf der rechten Seite wurden die Daten zunächst mit `PolynomialFeatures(degree=10)` um polynomiale Merkmale erweitert, anschließend mit dem `StandardScaler` skaliert und schließlich die Ridge-Regression auf die fertigen Merkmale angewendet: Dies ist eine polynomiale Regression mit Ridge-Regularisierung. Beachten Sie, wie ein Erhöhen von α zu flacheren (d.h. weniger extremen, vernünftigeren) Vorhersagen führt; die Varianz des Modells sinkt, aber sein Bias steigt dafür.

Wie die lineare Regression können wir auch die Ridge-Regression entweder als geschlossene Gleichung oder durch das Gradientenverfahren berechnen. Die Vor- und Nachteile sind die gleichen. Formel 4-9 zeigt die Lösung der geschlossenen Form (wobei \mathbf{A} eine $n \times n$ -Identitätsmatrix¹³ ist, nur dass die linke obere Ecke eine 0 für den Bias-Term enthält).

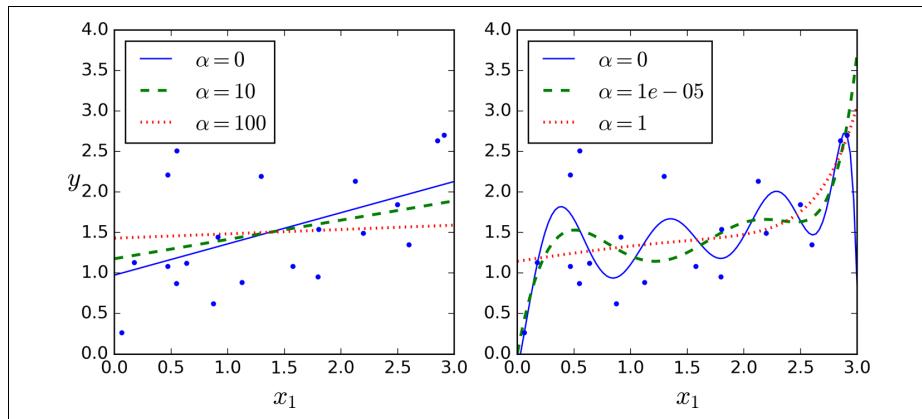


Abbildung 4-17: Ridge-Regression

Formel 4-9: Lösung der geschlossenen Form bei der Ridge-Regression

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Die Ridge-Regression lässt sich mit Scikit-Learn in der geschlossenen Form folgendermaßen (mit einer Variante von Formel 4-9 nach einer Technik zur Matrizenfaktorisierung von André-Louis Cholesky) lösen:

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([ 1.55071465])
```

¹³ Eine quadratische Matrix voller Nullen, die Einsen auf der Hauptdiagonalen enthält (von links oben nach rechts unten).

Über das stochastisches Gradientenverfahren:¹⁴

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([ 1.13500145])
```

Der Hyperparameter `penalty` legt die Art des Regularisierungsterms fest. Über die Angabe "l2" fügen Sie zur Kostenfunktion einen Regularisierungsterm in Höhe des halben Quadrats der ℓ_2 -Norm des Gewichtsvektors hinzu: Dies entspricht der Ridge-Regression.

Lasso-Regression

Das Verfahren *Least Absolute Shrinkage and Selection Operator Regression* (kurz: *Lasso-Regression*) ist eine weitere regularisierte Variante der linearen Regresssion: Wie die Ridge-Regression fügt sie zur Kostenfunktion einen Regularisierungsterm hinzu, dieser verwendet aber die ℓ_1 -Norm des Gewichtsvektors anstatt des halbier-ten Quadrats der ℓ_2 -Norm (siehe Formel 4-10).

Formel 4-10: Kostenfunktion bei der Lasso-Regression

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

Abbildung 4-18 zeigt das Gleiche wie Abbildung 4-17, aber ersetzt die Ridge-Modelle durch Lasso-Modelle und verwendet kleinere Werte für α .

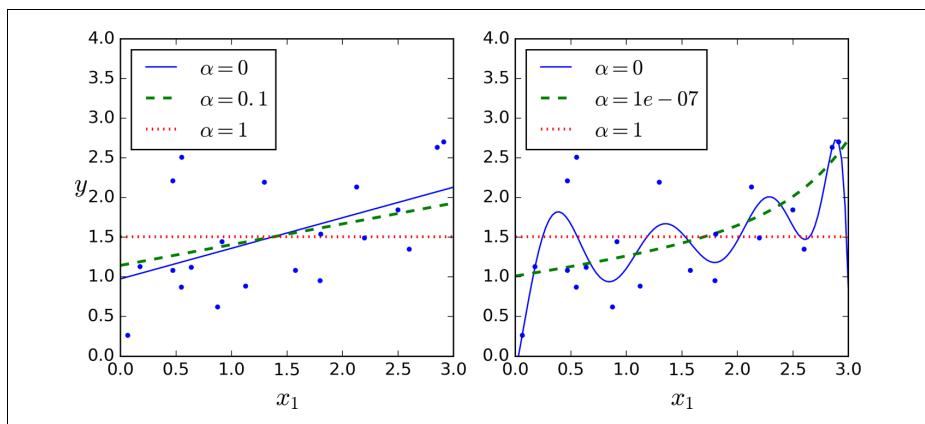


Abbildung 4-18: Lasso-Regression

¹⁴ Alternativ können Sie auch die Klasse `Ridge` mit dem Solver "sag" verwenden. Das Stochastic-Average-Gradientenverfahren ist eine Variante des SGD. Details finden Sie in der Präsentation »Minimizing Finite Sums with the Stochastic Average Gradient Algorithm« (<http://goo.gl/vxVya2>) von Mark Schmidt et al. von der University of British Columbia.

Eine wichtige Eigenschaft der Lasso-Regression ist, dass sie die Gewichte der unwichtigsten Merkmale vollständig eliminiert (d.h. diese auf null setzt). Beispielsweise sieht die gestrichelte Linie im Diagramm auf der rechten Seite von Abbildung 4-18 (mit $\alpha = 10^{-7}$) quadratisch oder fast schon linear aus: Sämtliche Gewichte der höhergradigen polynomiellen Merkmale sind auf null gesetzt. Anders ausgedrückt führt die Lasso-Regression eine automatische Merkmalsauswahl durch und gibt ein *spärliches Modell* aus (d.h. mit wenigen Gewichten ungleich null).

Warum das so ist, können Sie anhand von Abbildung 4-19 sehen: Im Diagramm oben links stellen die Symbole im Hintergrund (Ellipsen) eine unregularisierte MSE-Kostenfunktion dar ($\alpha = 0$), die weißen Kreise zeigen den Pfad des Batch-Gradientenverfahrens mit dieser Kostenfunktion. Die Symbole im Vordergrund (Rhomben) zeigen den ℓ_1 -Strafterm, und die Dreiecke stellen den Pfad des Batch-Gradientenverfahrens nur für diesen Strafterm dar ($\alpha \rightarrow \infty$). Achten Sie darauf, wie der Pfad zunächst $\theta_1 = 0$ erreicht, anschließend einen Abhang hinabrollt, bis er $\theta_2 = 0$ erreicht. Im Diagramm oben rechts stehen die Symbole für die gleiche Kostenfunktion zuzüglich eines ℓ_1 -Strafterms mit $\alpha = 0.5$. Das globale Minimum liegt auf der Achse $\theta_2 = 0$. Das Batch-Gradientenverfahren erreicht zunächst $\theta_2 = 0$ und rollt dann den Abhang bis zum globalen Minimum hinab. Die zwei Diagramme unten zeigen das Gleiche mit einem ℓ_2 -Strafterm. Das regularisierte Minimum liegt näher bei $\theta = 0$ als das unregularisierte Minimum, aber die Gewichte werden nicht vollständig eliminiert.

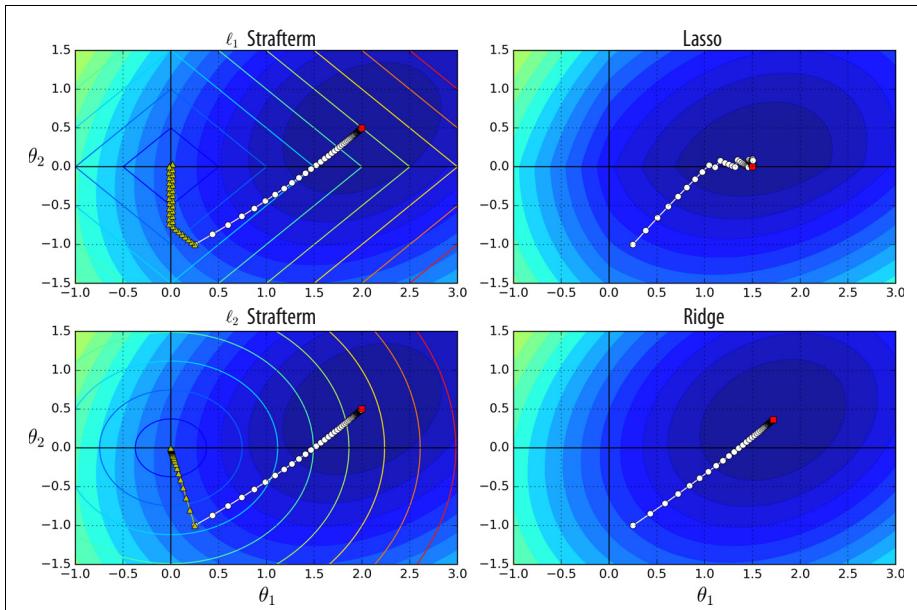


Abbildung 4-19: Lasso im Vergleich zur Ridge-Regularisierung



Bei der Lasso-Kostenfunktion neigt der Pfad beim Batch-Gradientenverfahren dazu, zum Ende hin über das Tal hinwegzuspringen. Dies liegt daran, dass sich die Steigung bei $\theta_2 = 0$ abrupt ändert. Sie müssen die Lernrate schrittweise senken, um beim globalen Minimum zu konvergieren.

Die Lasso-Kostenfunktion ist bei $\theta_i = 0$ (mit $i = 1, 2, \dots, n$) nicht differenzierbar, das Gradientenverfahren funktioniert trotzdem, wenn Sie einen *Subgradientenvektor* \mathbf{g} verwenden, wenn irgendein $\theta_i = 0$ beträgt.¹⁵ Formel 4-11 zeigt einen Subgradientenvektor, der sich für das Gradientenverfahren mit der Lasso-Kostenfunktion einsetzen lässt.

Formel 4-11: Subgradientenvektor für die Lasso-Regression

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{wobei } \text{sign}(\theta_i) = \begin{cases} -1 & \text{wenn } \theta_i < 0 \\ 0 & \text{wenn } \theta_i = 0 \\ +1 & \text{wenn } \theta_i > 0 \end{cases}$$

Hier folgt ein kleines Scikit-Learn-Beispiel für das Verwenden der Klasse Lasso. Sie könnten stattdessen auch die Klasse SGDRegressor(penalty="l1") verwenden.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

Elastic Net

Elastic Net liegt auf halber Strecke zwischen der Ridge-Regression und der Lasso-Regression. Der Regularisierungsterm ist eine Mischung aus den Regularisierungstermen von Ridge und Lasso, und Sie können das Mischverhältnis r bestimmen. Bei $r = 0$ ist Elastic Net äquivalent zur Ridge-Regression, und bei $r = 1$ entspricht es der Lasso-Regression (siehe Formel 4-12).

Formel 4-12: Kostenfunktion von Elastic Net

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

Wann sollten Sie also die einfache lineare Regression (also ohne jegliche Regularisierung) Ridge, Lasso oder Elastic Net verwenden? Ein wenig Regularisierung ist fast immer vorzuziehen, also sollten Sie die einfache lineare Regression vermeiden.

¹⁵ Sie können sich einen Subgradientenvektor bei einem nicht differenzierbaren Punkt als mittleren Vektor zwischen den Gradientenvektoren um diesen Punkt herum vorstellen.

Ridge ist ein guter Ausgangspunkt, aber wenn Sie vermuten, dass nur einige Merkmale wichtig sind, sollten Sie Lasso oder Elastic Net verwenden, da diese tendenziell die Gewichte nutzloser Merkmale auf null reduzieren. Im Allgemeinen ist Elastic Net gegenüber Lasso vorzuziehen, da sich Lasso sprunghaft verhalten kann, wenn die Anzahl der Merkmale größer als die Anzahl der Trainingsdatenpunkte ist oder wenn mehrere Merkmale stark miteinander korrelieren.

Hier folgt ein kurzes Anwendungsbeispiel für die Scikit-Learn-Klasse ElasticNet (`l1_ratio` entspricht dem Mischverhältnis r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

Early Stopping

Ein völlig anderes Verfahren zum Regularisieren iterativer Lernalgorithmen wie des Gradientenverfahrens ist, das Training zu unterbrechen, sobald der Validierungsfehler ein Minimum erreicht. Dies bezeichnet man als *Early Stopping*. Abbildung 4-20 zeigt ein komplexes Modell (in diesem Fall ein höhergradiges polynomielles Regressionsmodell), das mit dem Batch-Gradientenverfahren trainiert wird. Von Epoche zu Epoche lernt der Algorithmus, und der Vorhersagefehler (RMSE) auf dem Trainingsdatensatz sinkt dabei ebenso wie der Vorhersagefehler auf dem Validierungsdatensatz. Nach einer Weile hört der Validierungsfehler aber auf zu sinken und steigt wieder an. Dies weist darauf hin, dass das Modell angefangen hat, die Trainingsdaten zu overfitten. Mit Early Stopping beenden Sie das Training, sobald der Validierungsfehler das Minimum erreicht. Diese Regularisierungstechnik ist derart einfach und effizient, dass Geoffrey Hinton sie ein »beautiful free lunch« genannt hat.

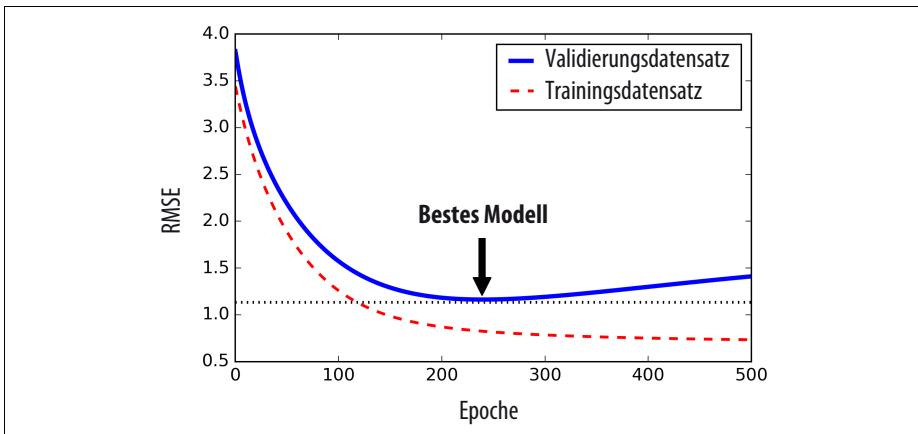


Abbildung 4-20: Regularisierung mit Early Stopping

TIPP: Beim stochastischen und dem Mini-Batch-Gradientenverfahren sind die Kurvenverläufe nicht ganz so glatt, und es ist manchmal schwierig zu erkennen, ob sie das Minimum erreicht haben oder nicht. Eine Möglichkeit ist, nur dann anzuhalten, wenn der Validierungsfehler eine Weile oberhalb des Minimums liegt (wenn Sie sich sicher sind, dass das Modell es nicht besser kann). Anschließend setzen Sie die Modellparameter wieder auf den Punkt zurück, an dem der Validierungsfehler minimal war.

Hier sehen Sie eine einfache Implementierung des Early Stopping:

```
from sklearn.base import clone

sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # macht weiter, wo es aufgehört hat
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Mit dem Aufruf der Methode `fit()` mit `warm_start=True` wird das Trainieren beim letzten Stand fortgesetzt, anstatt neu zu starten.

Logistische Regression

Wie in Kapitel 1 besprochen, lassen sich einige Regressionsalgorithmen auch zur Klassifikation einsetzen (und umgekehrt). Die *logistische Regression* (auch *Logit Regression* genannt) wird häufig zum Abschätzen der Wahrscheinlichkeit eingesetzt, dass ein Datenpunkt einer bestimmten Kategorie angehört (z.B. wie hoch ist die Wahrscheinlichkeit, dass diese E-Mail Spam enthält?). Wenn die geschätzte Wahrscheinlichkeit mehr als 50% beträgt, sagt das Modell vorher, dass der Datenpunkt zu dieser Kategorie gehört (der positiven Kategorie mit dem Label »1«), ansonsten wird das Gegenteil vorhergesagt (d.h., der Punkt gehört der negativen Kategorie mit dem Label »0« an). Damit ist dies ein binärer Klassifikator.

Abschätzen von Wahrscheinlichkeiten

Wie funktioniert die logistische Regression? Genauso wie ein lineares Regressionsmodell wird eine gewichtete Summe der Eingabemerkmale (und eines Bias-Terms) berechnet, aber anstatt wie bei der linearen Regression das Ergebnis direkt auszugeben, wird die *logistische Funktion* des Ergebnisses berechnet (siehe Formel 4-13).

Formel 4-13: Geschätzte Wahrscheinlichkeit bei einem logistischen Regressionsmodell (Vektorschreibweise)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \cdot \mathbf{x})$$

Die logistische Funktion – auch *logit* genannt – und als $\sigma(\cdot)$ geschrieben – ist eine *sigmoide Funktion* (d. h. eine S-förmige), die eine Zahl zwischen 0 und 1 ausgibt. Sie ist in Formel 4-14 definiert und in Abbildung 4-21 dargestellt.

Formel 4-14: Logistische Funktion

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

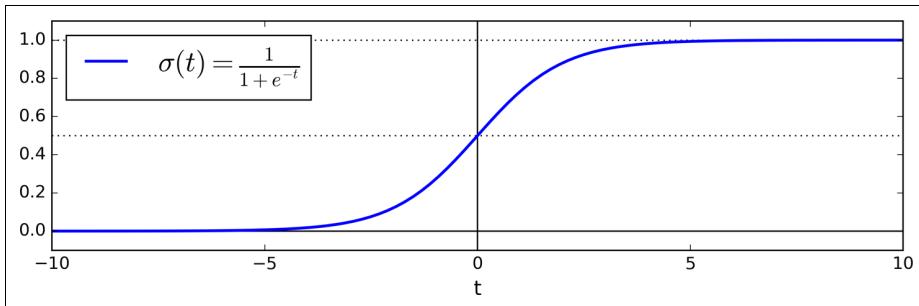


Abbildung 4-21: Logistische Funktion

Hat das logistische Regressionsmodell erst einmal die Wahrscheinlichkeit $\hat{p} = h_{\theta}(\mathbf{x})$ abgeschätzt, dass ein Datenpunkt \mathbf{x} der positiven Kategorie angehört, kann es leicht eine Vorhersage \hat{y} treffen (siehe Formel 4-15).

Formel 4-15: Vorhersage eines logistischen Regressionsmodells

$$\hat{y} = \begin{cases} 0 & \text{bei } \hat{p} < 0.5, \\ 1 & \text{bei } \hat{p} \geq 0.5. \end{cases}$$

Bei $t < 0$ gilt $\sigma(t) < 0.5$ und bei $t \geq 0$ gilt $\sigma(t) \geq 0.5$. Ein logistisches Regressionsmodell sagt also 1 vorher, wenn $\boldsymbol{\theta}^T \cdot \mathbf{x}$ positiv ist und 0 falls es negativ ist.

Trainieren und Kostenfunktion

Gut, Sie wissen nun, wie ein logistisches Regressionsmodell Wahrscheinlichkeiten abschätzt und Vorhersagen trifft. Aber wie lässt es sich trainieren? Das Trainingsziel ist, den Parametervektor $\boldsymbol{\theta}$ so zu setzen, dass das Modell bei positiven Datenpunkten ($y = 1$) hohe Wahrscheinlichkeiten und bei negativen Datenpunkten ($y = 0$) geringe Wahrscheinlichkeiten abschätzt. Diese Idee ist in der Kostenfunktion in Formel 4-16 für einen einzelnen Trainingsdatenpunkt \mathbf{x} ausgedrückt.

Formel 4-16: Kostenfunktion eines einzelnen Trainingsdatenpunkts

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{bei } y = 1, \\ -\log(1 - \hat{p}) & \text{bei } y = 0. \end{cases}$$

Diese Kostenfunktion ist sinnvoll, weil $-\log(t)$ sehr groß wird, sobald t sich 0 nähert. Daher sind die Kosten hoch, wenn das Modell bei einem positiven Datenpunkt eine Wahrscheinlichkeit nahe 0 schätzt, ebenso wenn das Modell bei einem negativen Datenpunkt eine Wahrscheinlichkeit nahe 1 schätzt. Andererseits ist $-\log(t)$ nahe 0, wenn t nahe 1 ist, sodass die Kosten auf 0 zugehen, wenn die geschätzte Wahrscheinlichkeit bei einem negativen Datenpunkt nahe bei 0 oder bei einem positiven Datenpunkt nahe bei 1 liegt. Genau das benötigen wir.

Die Kostenfunktion über den gesamten Trainingsdatensatz entspricht den durchschnittlichen Kosten über sämtliche Trainingsdatenpunkte. Diese lässt sich wie in Formel 4-17 ausdrücken (wie Sie leicht überprüfen können), die man als *Log Loss* bezeichnet.

Formel 4-17: Kostenfunktion bei der logistischen Regression (Log Loss)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

Die schlechte Nachricht dabei ist, dass es keine bekannte Gleichung mit geschlossener Form gibt, mit der sich ein Wert für θ berechnen ließe, der diese Kostenfunktion minimiert (es gibt kein Äquivalent zur Normalengleichung). Die gute Nachricht ist, dass diese Funktion konvex ist, sodass das Gradientenverfahren (oder ein anderer Optimierungsalgorithmus) garantiert das globale Optimum findet (wenn die Lernrate nicht zu groß ist und Sie lange genug warten). Die partiellen Ableitungen der Kostenfunktion nach dem j ten Modellparameter θ_j sind in Formel 4-18 angegeben.

Formel 4-18: Partielle Ableitungen der logistischen Kostenfunktion

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

Diese Gleichung ist Formel 4-5 sehr ähnlich: Für jeden Datenpunkt wird der Vorhersagefehler berechnet und mit dem j ten Merkmalswert multipliziert. Anschließend wird daraus der Mittelwert aller Trainingsdatenpunkte berechnet. Der Gradientenvektor aus sämtlichen partiellen Ableitungen lässt sich im Batch-Gradientenverfahren verwenden. Das ist alles: Sie wissen nun, wie ein logistisches Regressionsmodell trainiert wird. Für das stochastische Gradientenverfahren würden Sie natürlich nur genau einen Datenpunkt und beim Mini-Batch-Gradientenverfahren ebenfalls nur genau einen Mini-Batch verwenden.

Entscheidungsgrenzen

Verwenden wir den Iris-Datensatz, um die logistische Regression zu veranschaulichen. Es handelt sich hierbei um einen bekannten Datensatz, der die Länge und Breite der Kelchblätter (engl. sepal) und Kronblätter (petal) von 150 Iris-Blüten aus drei Unterarten enthält: Iris-Setosa, Iris-Versicolor und Iris-Virginica (siehe Abbildung 4-22).

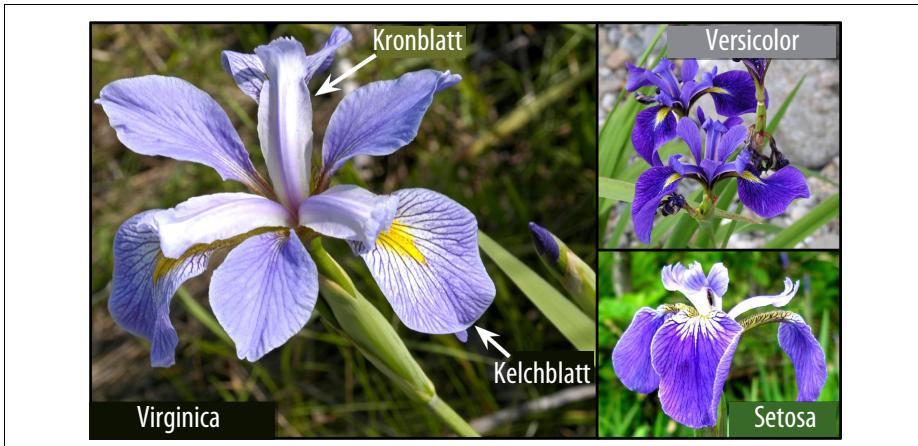


Abbildung 4-22: Blüten dreier Arten von Iris-Pflanzen¹⁶

Versuchen wir, einen Klassifikator zu erstellen, mit dem sich anhand der Breite der Kronblätter die Spezies Iris-Virginica erkennen lässt. Laden wir zunächst die Daten:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> list(iris.keys())  
['data', 'target_names', 'feature_names', 'target', 'DESCR']  
>>> X = iris["data"][:, 3:] # Breite der Kronblätter  
>>> y = (iris["target"] == 2).astype(np.int) # 1 bei Iris-Virginica, sonst 0
```

Nun trainieren wir ein logistisches Regressionsmodell:

```
from sklearn.linear_model import LogisticRegression  
  
log_reg = LogisticRegression()  
log_reg.fit(X, y)
```

Betrachten wir die geschätzten Wahrscheinlichkeiten bei Blüten mit Breiten der Kronblätter von 0 bis 3 cm (Abbildung 4-23):

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  
y_proba = log_reg.predict_proba(X_new)
```

¹⁶ Die Bilder wurden von den entsprechenden Wikipedia-Seiten reproduziert. Foto einer Iris-Virginica von Frank Mayfield (Creative Commons BY-SA 2.0, <https://creativecommons.org/licenses/by-sa/2.0/>), Foto einer Iris-Versicolor von D. Gordon E. Robertson (Creative Commons BY-SA 3.0, <https://creativecommons.org/licenses/by-sa/3.0/>), und das Foto der Iris-Setosa ist Public Domain.

```

plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Nicht Iris-Virginica")
# + weiterer Matplotlib-Code, um das Bild ansprechender zu gestalten

```

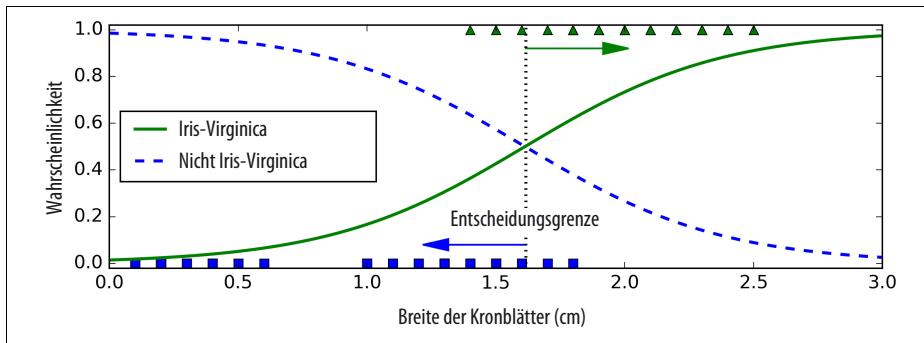


Abbildung 4-23: Geschätzte Wahrscheinlichkeiten und Entscheidungsgrenze

Die Breite der Kronblätter der Iris-Virginica-Blüten (als Dreiecke dargestellt) liegt zwischen 1.4 cm und 2.5 cm, während die anderen Iris-Blüten (die Quadrate) schmalere Kronblätter zwischen 0.1 cm und 1.8 cm haben. Diese Bereiche überlappen einander ein wenig. Oberhalb von 2 cm ist sich der Klassifikator sicher, dass die Blüte eine Iris-Virginica ist (für diese Kategorie wird eine hohe Wahrscheinlichkeit ausgegeben), unterhalb von 1 cm ist er sich sicher, dass es keine Iris-Virginica ist (hohe Wahrscheinlichkeit für die Kategorie »Nicht Iris-Virginica«). Im Übergangsbereich zwischen diesen Extremen ist sich der Klassifikator nicht sicher. Wenn Sie aber die Kategorie vorhersagen (mit der Methode `predict()` anstatt mit `predict_proba()`), wird die jeweils wahrscheinlichere Kategorie ausgegeben. Es gibt also eine *Entscheidungsgrenze* bei etwa 1.6 cm, bei der beide Wahrscheinlichkeiten mit 50% gleich groß sind: Wenn die Kronblätter breiter als 1.6 cm sind, sagt der Klassifikator eine Iris-Virginica vorher, andernfalls, dass es keine ist (selbst wenn er sich nicht besonders sicher ist):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Abbildung 4-24 zeigt den gleichen Datensatz, aber diesmal mit zwei Merkmalen: der Länge und Breite der Kronblätter. Ein Klassifikator mit logistischer Regression kann nach dem Trainieren anhand dieser zwei Merkmale die Wahrscheinlichkeit abschätzen, dass eine neue Blüte eine Iris-Virginica ist. Die gestrichelte Linie markiert die Punkte, an denen das Modell eine Wahrscheinlichkeit von 50% schätzt: Dies ist die Entscheidungsgrenze des Modells. Beachten Sie, dass diese Grenze eine Gerade ist.¹⁷ Jede der parallelen Linien steht für die Punkte, an denen das Modell eine bestimmte Wahrscheinlichkeit liefert, von 15% (unten links) bis 90% (oben rechts).

¹⁷ Die Menge aller Punkte \mathbf{x} , für die $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ gilt, ergibt eine gerade Linie.

rechts). Sämtliche Blüten jenseits der Linie oben rechts sind laut dem Modell mit mehr als 90%iger Wahrscheinlichkeit Iris-Virginica.

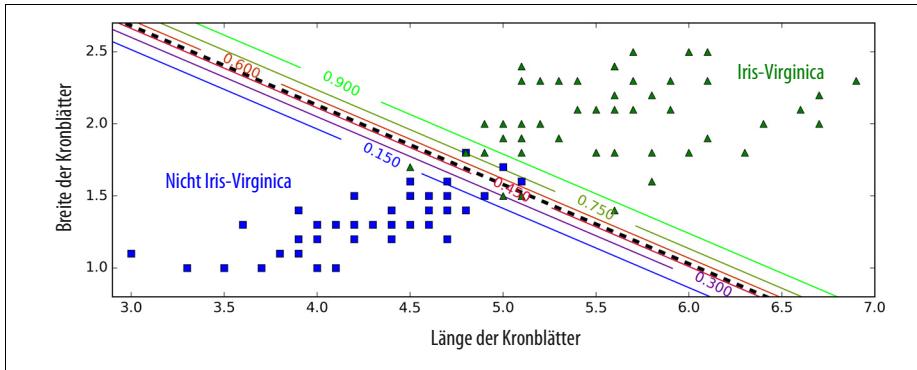


Abbildung 4-24: Lineare Entscheidungsgrenze

Wie andere lineare Modelle lässt sich auch die logistische Regression mit ℓ_1 - oder ℓ_2 -Straftermen regularisieren. Scikit-Learn fügt standardmäßig einen ℓ_2 -Strafterm hinzu.



Der Hyperparameter zum Steuern der Regularisierung eines Logistic-Regression-Modells in Scikit-Learn ist nicht `alpha` (wie bei anderen linearen Modellen), sondern dessen Kehrwert C . Je höher der Wert von C , desto *weniger stark* ist das Modell regularisiert.

Softmax-Regression

Das logistische Regressionsmodell lässt sich direkt auf mehrere Kategorien verallgemeinern, ohne dass man mehrere binäre Klassifikatoren trainieren und miteinander kombinieren muss (wie in Kapitel 3 besprochen). Dies nennt man *Softmax-Regression* oder *multinomiale logistische Regression*.

Die Grundidee dabei ist recht einfach: Die Softmax-Regression berechnet für einen Datenpunkt x zunächst den Score $s_k(x)$ für jede Kategorie k und schätzt anschließend durch Berechnen der *Softmax-Funktion* (auch *normalisierte Exponentialfunktion* genannt) die Wahrscheinlichkeit jeder Kategorie ab. Die Formel zum Berechnen von $s_k(x)$ aus den Scores sollte Ihnen bekannt vorkommen, da sie der Formel für die Vorhersage einer linearen Regression ähnelt (siehe Formel 4-19).

Formel 4-19: Softmax-Score für Kategorie k

$$s_k(x) = (\theta^{(k)})^T \cdot x$$

Dabei hat jede Kategorie ihren eigenen Parametervektor $\theta^{(k)}$. Diese Vektoren werden üblicherweise als Zeilen einer *Parametmatrix* Θ abgelegt.

Haben Sie erst einmal den Score jeder Kategorie für den Datenpunkt x berechnet, können Sie die Wahrscheinlichkeit \hat{p}_k für die Zugehörigkeit des Datenpunkts zur

Kategorie k abschätzen, indem Sie die Softmax-Funktion (Formel 4-20) auf die Scores anwenden: Sie berechnet die Exponentialfunktion aus jedem Score und normalisiert diese (durch Teilen durch die Summe aller Potenzen).

Formel 4-20: Softmax-Funktion

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

- K ist dabei die Anzahl der Kategorien.
- $\mathbf{s}(\mathbf{x})$ ist ein Vektor, der die Scores jeder Kategorie für den Datenpunkt \mathbf{x} enthält.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ ist die anhand der Scores geschätzte Wahrscheinlichkeit dafür, dass der Datenpunkt \mathbf{x} zur Kategorie k gehört.

Wie bei der Klassifikation mit logistischer Regression sagt die Softmax-Regression die Kategorie mit der höchsten geschätzten Wahrscheinlichkeit vorher (die einfach die Kategorie mit dem höchsten Score ist), wie Formel 4-21 zeigt.

Formel 4-21: Vorhersage eines Klassifikators mit Softmax-Regression

$$\hat{y} = \underset{k}{\operatorname{argmax}} \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}} s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \left((\boldsymbol{\theta}^{(k)})^T \cdot \mathbf{x} \right)$$

- Der Operator *argmax* liefert den Wert einer Variable, der eine Funktion maximiert. In dieser Gleichung liefert er denjenigen Wert für k , für den die geschätzte Wahrscheinlichkeit $\sigma(\mathbf{s}(\mathbf{x}))_k$ maximal wird.



Die Klassifikation mit Softmax-Regression sagt zeitgleich nur eine Kategorie vorher (sie arbeitet mit mehreren Kategorien, nicht mehreren Ausgaben). Sie sollte also nur verwendet werden, wenn sich die Kategorien gegenseitig ausschließen, wie etwa bei Pflanzenarten. Sie können sie also nicht dazu verwenden, um mehrere Personen im gleichen Bild zu erkennen.

Da Sie nun wissen, wie das Modell Wahrscheinlichkeiten schätzt und Vorhersagen trifft, werfen wir einen Blick auf das Trainieren. Das Ziel ist ein Modell, das die Zielkategorie mit einer hohen Wahrscheinlichkeit schätzt (und konsequenterweise eine niedrige Wahrscheinlichkeit für die übrigen Kategorien). Die als *Kreuzentropie* bezeichnete Kostenfunktion in Formel 4-22 zu minimieren, sollte dazu beitragen, da sie das Modell für niedrige Wahrscheinlichkeiten der Zielkategorie abstrafft. Mit der Kreuzentropie wird häufig gemessen, wie gut die Wahrscheinlichkeiten der Kategorien mit den Zielkategorien übereinstimmen (wir werden ihr in den nächsten Kapiteln noch mehrmals begegnen).

Formel 4-22: Die Kreuzentropie als Kostenfunktion

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

- $y_k^{(i)}$ ist gleich 1, wenn die Zielkategorie des i^{ten} Datenpunkts k ist; andernfalls beträgt es 0.

Wenn es nur zwei Kategorien gibt ($K = 2$), entspricht diese Kostenfunktion der Kostenfunktion der logistischen Regression (Log Loss; siehe Formel 4-17).

Kreuzentropie

Die Kreuzentropie stammt aus der Informationstheorie. Nehmen Sie einmal an, Sie möchten jeden Tag Informationen über das Wetter effizient übermitteln. Wenn es acht Möglichkeiten gibt (Sonne, Regen und so weiter), könnten Sie jede Möglichkeit durch 3 Bits codieren, da $2^3 = 8$ ergibt. Wenn Sie allerdings wissen, dass es fast jeden Tag sonnig ist, wäre es viel effizienter, »Sonne« als ein Bit (0) zu codieren und die anderen sieben Möglichkeiten durch je vier Bits auszudrücken (die alle mit 1 beginnen). Die Kreuzentropie bestimmt die durchschnittliche Anzahl Bits, die Sie pro Möglichkeit übermitteln. Wenn Ihre Annahme über das Wetter perfekt ist, entspricht die Kreuzentropie der Entropie des Wetters (d.h. dessen intrinsischer Unvorhersagbarkeit). Wenn aber Ihre Annahmen falsch sind (z.B. weil es häufig regnet), erhöht sich die Kreuzentropie um einen Betrag, den man als *Kullback-Leibler-Divergenz* bezeichnet.

Die Kreuzentropie zweier Wahrscheinlichkeitsverteilungen p und q ist als

$$H(p, q) = - \sum_x p(x) \log q(x)$$

definiert (zumindest wenn die Verteilungen diskret sind).

Der Gradientenvektor dieser Kostenfunktion nach $\theta^{(k)}$ ist in Formel 4-23 geschrieben:

Formel 4-23: Gradientenvektor der Kreuzentropie für Kategorie k

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

Nun können Sie den Gradientenvektor für jede Kategorie berechnen und über das Gradientenverfahren (oder einen anderen Optimierungsalgorithmus) die Parametermatrix Θ ermitteln, für die die Kostenfunktion minimal wird.

Verwenden wir die Softmax-Regression, um die Iris-Blüten in alle drei Kategorien einzuteilen. Die Scikit-Learn-Klasse `LogisticRegression` verwendet standardmäßig

die One-versus-All-Strategie, wenn Sie diese mit mehr als zwei Kategorien trainieren. Sie können aber den Hyperparameter `multi_class` auf "multinomial" setzen, um stattdessen die Softmax-Regression zu verwenden. Sie müssen außerdem einen Solver angeben, der die Softmax-Regression unterstützt, beispielsweise den Solver "`lbfgs`" (Details dazu finden Sie in der Dokumentation von Scikit-Learn). Dieser verwendet automatisch eine ℓ_2 -Regularisierung, die Sie über den Hyperparameter `C` steuern können.

```
X = iris["data"][:, (2, 3)] # Länge und Breite der Kronblätter
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Das nächste Mal, wenn Sie eine Iris-Blüte mit 5 cm langen und 2 cm breiten Kronblättern finden, können Sie Ihr Modell fragen, welche Art Iris dies ist. Es sollte als Antwort Iris-Virginica (Kategorie 2) mit einer Wahrscheinlichkeit von 94.2 % geben (oder Iris-Versicolor mit einer Wahrscheinlichkeit von 5.8 %):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07,  5.75276067e-02,  9.42471760e-01]])
```

Abbildung 4-5 zeigt die sich dabei ergebenden Entscheidungsgrenzen als Hintergrundfarben. Die Entscheidungsgrenzen zwischen zwei Kategorien sind stets linear. Die Abbildung zeigt auch die Wahrscheinlichkeiten für die Kategorie Iris-Versicolor als geschwungene Linien (z.B. die mit 0.450 beschriftete Linie steht für eine Wahrscheinlichkeit von 45%). Das Modell kann eine Kategorie vorhersagen, die eine geschätzte Wahrscheinlichkeit unter 50 % hat. Beispielsweise ist am Punkt, an dem sich alle drei Linien treffen, die Wahrscheinlichkeit für alle Kategorien die gleiche, nämlich 33 %.

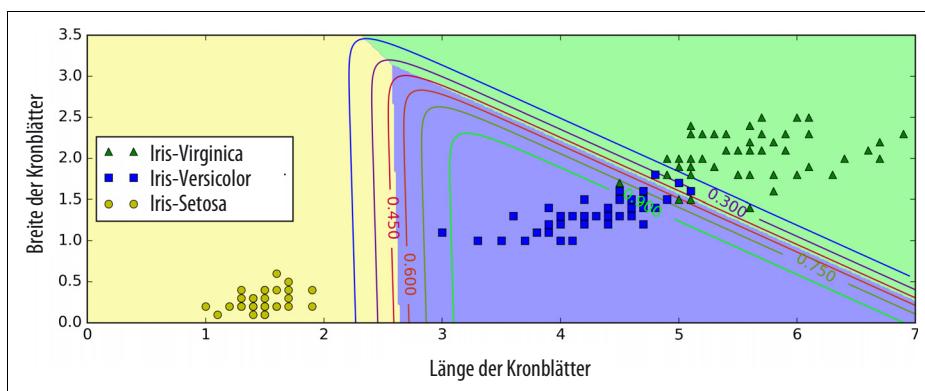


Abbildung 4-25: Entscheidungsgrenzen bei der Softmax-Regression

Übungen

1. Welchen Trainingsalgorithmus für die lineare Regression können Sie verwenden, wenn Sie einen Trainingsdatensatz mit Millionen Merkmalen haben?
2. Nehmen Sie an, dass die Merkmale in Ihrem Trainingsdatensatz unterschiedlich skaliert sind. Welche Algorithmen würden dadurch in Mitleidenschaft gezogen und in welcher Weise? Was können Sie dagegen tun?
3. Kann das Gradientenverfahren bei einem logistischen Regressionsmodell in einem lokalen Minimum stecken bleiben?
4. Führen alle Algorithmen für das Gradientenverfahren zum gleichen Modell, vorausgesetzt, sie laufen lange genug?
5. Nehmen Sie an, Sie verwenden das Batch-Gradientenverfahren und plotten den Validierungsfehler in jeder Epoche. Was passiert vermutlich, wenn der Validierungsfehler ständig steigt? Wie können Sie dies beheben?
6. Ist es eine gute Idee, das Mini-Batch-Gradientenverfahren sofort zu unterbrechen, sobald der Validierungsfehler steigt?
7. Welcher der besprochenen Algorithmen für das Gradientenverfahren erreicht die Umgebung der optimalen Lösung am schnellsten? Welcher konvergiert? Wie können Sie auch die übrigen konvergieren lassen?
8. Sie verwenden eine polynomische Regression, plotten die Lernkurven und bemerken, dass es zwischen dem Trainingsfehler und dem Validierungsfehler einen großen Unterschied gibt. Was passiert? Nennen Sie drei Möglichkeiten, dies zu beheben.
9. Bei der Ridge-Regression bemerken Sie, dass der Trainingsfehler und der Validierungsfehler beinahe gleich und recht hoch sind. Krankt dieses Modell an einem hohen Bias oder an einer hohen Varianz? Sollten Sie den Regularisierungsparameter α erhöhen oder senken?
10. Welche Gründe sprechen für folgende Verfahren?
 - Ridge-Regression anstatt einer einfachen linearen Regression (d.h. ohne Regularisierung)?
 - Lasso anstelle einer Ridge-Regression?
 - Elastic Net anstelle von Lasso?
11. Angenommen, Sie möchten Bilder als innen/außen und Tag/Nacht klassifizieren. Sollten Sie zwei Klassifikatoren mit logistischer Regression oder einen Klassifikator mit Softmax-Regression erstellen?
12. Implementieren Sie das Batch-Gradientenverfahren mit Early Stopping für die Softmax-Regression (ohne Scikit-Learn).

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

Support Vector Machines

Eine *Support Vector Machine* (SVM) ist ein sehr mächtiges und flexibles Machine-Learning-Modell, mit dem Sie sowohl lineare als auch nichtlineare Klassifikationsaufgaben, Regression und sogar die Erkennung von Ausreißern bewältigen können. Es gehört zu den beliebtesten Modellen, und deshalb sollte jeder mit etwas Interesse an Machine Learning die SVM in seinem Repertoire haben. SVMs sind besonders zur Klassifikation komplexer Datensätze kleiner oder mittlerer Größe geeignet.

In diesem Kapitel werden Grundbegriffe zu SVMs, wie man sie verwendet und ihre Funktionsweise erklärt.

Lineare Klassifikation mit SVMs

Der SVMs zugrunde liegende Gedanke lässt sich am besten anhand einiger Bilder erläutern. Abbildung 5-1 zeigt einen Teil des Iris-Datensatzes, der am Ende von Kapitel 4 erstmals erwähnt wurde. Die zwei Kategorien lassen sich sehr leicht mit einer Geraden voneinander trennen (sie sind *linear separierbar*). Das Diagramm auf der linken Seite zeigt die Entscheidungsgrenzen dreier möglicher linearer Klassifikatoren. Das als gestrichelte Linie dargestellte Modell ist so schlecht, dass es die Kategorien nicht einmal ordentlich voneinander trennt. Die zwei übrigen Modelle funktionieren auf dem Trainingsdatensatz ausgezeichnet, aber ihre Entscheidungsgrenzen befinden sich sehr nah an den Datenpunkten, sodass diese Modelle bei neuen Daten vermutlich nicht besonders gut abschneiden werden. Die durchgezogene Linie im Diagramm auf der rechten Seite dagegen stellt die Entscheidungsgrenze eines SVM-Klassifikators dar; diese Linie separiert die beiden Kategorien nicht nur, sie hält auch den größtmöglichen Abstand zu den Trainingsdatenpunkten ein. Sie können sich einen SVM-Klassifikator als die breiteste mögliche Straße zwischen den Kategorien vorstellen (hier als parallele gestrichelte Linien dargestellt). Dies bezeichnet man auch als *Large-Margin-Klassifikation*.

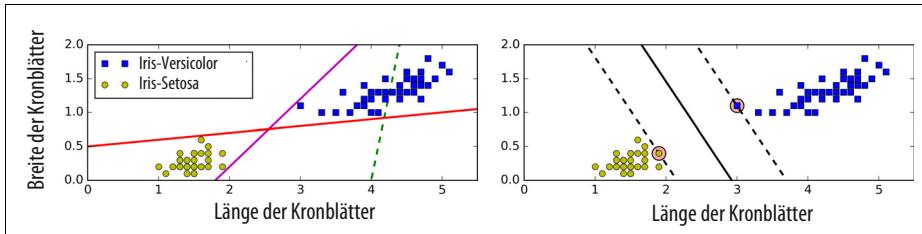


Abbildung 5-1: Large-Margin-Klassifikation

Beachten Sie, dass das Hinzufügen neuer Trainingsdaten »abseits der Straße« die Entscheidungsgrenze überhaupt nicht beeinflusst: Sie wird ausschließlich durch die Datenpunkte am Rande der Straße determiniert (oder »gestützt«). Diese Datenpunkte nennt man deshalb auch die *Stützvektoren* (engl. *support vectors*) (diese sind in Abbildung 5-1 durch Kreise markiert).



SVMs reagieren empfindlich auf die Skalierung der Merkmale, wie Sie in Abbildung 5-2 sehen: Im linken Diagramm ist die vertikale Skala deutlich größer als die horizontale. Daher ist die breiteste mögliche Straße nahezu horizontal. Nach Skalieren der Merkmale (z.B. mit dem StandardScaler in Scikit-Learn), sieht die Entscheidungsgrenze deutlich besser aus (auf der rechten Seite).

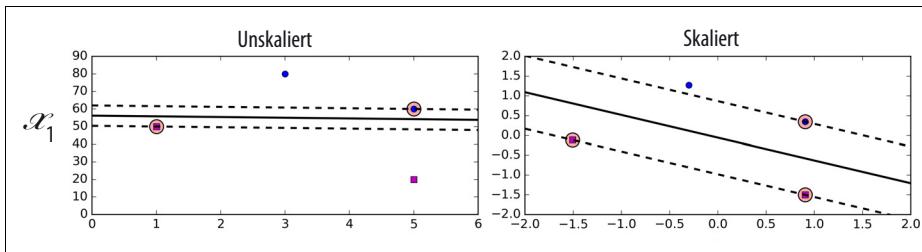


Abbildung 5-2: Empfindlichkeit für Skalierung der Merkmale

Soft-Margin-Klassifikation

Wenn wir voraussetzen, dass sich sämtliche Datenpunkte abseits der Straße und auf der richtigen Straßenseite befinden, nennt man dies *Hard-Margin-Klassifikation*. Bei der Hard-Margin-Klassifikation treten aber zwei Probleme auf: Erstens funktioniert sie nur, wenn die Daten linear separierbar sind. Zweitens ist sie recht anfällig für Ausreißer. Abbildung 5-3 zeigt den Iris-Datensatz mit nur einem zusätzlichen Ausreißer: Auf der linken Seite ist das Finden eines strengen Margin unmöglich, auf der rechten führt der zusätzliche Punkt zu einer deutlich anderen Trennlinie als der in Abbildung 5-1 ohne den Ausreißer, und das Modell verallgemeinert deshalb nicht so gut.

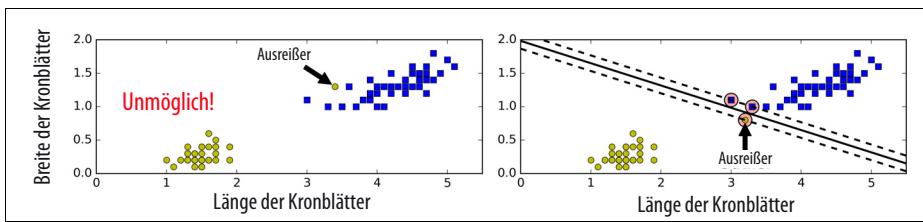


Abbildung 5-3: Anfälligkeit der Hard-Margin-Klassifikation für Ausreißer

Um diesen Schwierigkeiten aus dem Weg zu gehen, ist ein etwas flexibleres Modell vorzuziehen. Dabei gilt es, eine gute Balance zwischen der größtmöglichen Straßenbreite und einer begrenzten Anzahl von *Margin-Verletzungen* (also Datenpunkten, die mitten auf der Straße oder sogar auf der falschen Seite landen) herzustellen. Dies bezeichnet man als *Soft-Margin-Klassifikation*.

In den SVM-Klassen in Scikit-Learn können Sie dieses Verhalten über den Hyperparameter C steuern: Ein kleinerer Wert für C führt zu einer breiteren Straße, aber mehr Margin-Verletzungen. Abbildung 5-4 zeigt die Entscheidungsgrenzen und Margins für zwei Soft-Margin-SVM-Klassifikatoren auf einem nichtlinear separierbaren Datensatz. Auf der linken Seite führt ein hoher Wert für C dazu, dass der Klassifikator weniger Verletzungen der Margin zulässt, aber dafür einen schmalen Margin in Kauf nimmt. Auf der rechten Seite wird der Margin dank eines niedrigeren Werts für C viel breiter, aber viele Datenpunkte landen auf der Straße. Dennoch sieht es so aus, als wenn der zweite Klassifikator besser verallgemeinert: Tatsächlich unterlaufen ihm sogar auf diesem Trainingsdatensatz weniger Fehler, da nämlich die meisten Margin-Verletzungen auf der richtigen Seite der Entscheidungsgrenze liegen.

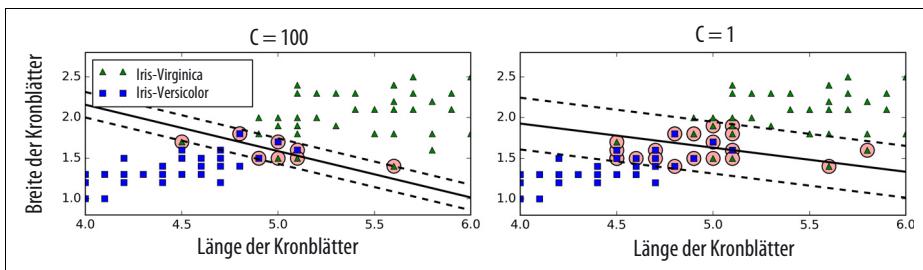


Abbildung 5-4: Weniger Margin-Verletzungen gegenüber einem breiteren Margin



Wenn Ihr SVM-Modell overfittet, können Sie es durch Senken des C-Werts regularisieren.

Das folgende Codebeispiel mit Scikit-Learn lädt den Iris-Datensatz, skaliert die Merkmale und trainiert anschließend ein lineares SVM-Modell (mithilfe der Klasse

`LinearSVC` und $C = 1$ sowie der in Kürze erklärten Funktion *hinge loss*), um Blumen der Art Iris-Virginica zu erkennen. Das dabei erhaltene Modell ist auf der rechten Seite von Abbildung 5-4 dargestellt.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # Länge, Breite der Kronblätter
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])
svm_clf.fit(X, y)
```

Anschließend können Sie mit dem Modell wie bisher Vorhersagen vornehmen:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```



Im Gegensatz zu Klassifikatoren, die mit der logistischen Regression arbeiten, geben SVM-Klassifikatoren keine Wahrscheinlichkeiten für die einzelnen Kategorien aus.

Alternativ könnten Sie auch die Klasse `SVC` mit `SVC(kernel="linear", C=1)` verwenden. Dies ist aber wesentlich langsamer, besonders bei großen Trainingsdatensätzen, und daher nicht empfehlenswert. Eine weitere Möglichkeit bietet die Klasse `SGDClassifier` mit `SGDClassifier(loss="hinge", alpha=1/(m*C))`. Diese verwendet das stochastische Gradientenverfahren (siehe Kapitel 4), um einen linearen SVM-Klassifikator zu trainieren. Sie konvergiert nicht so schnell wie die Klasse `LinearSVC`, ist dafür aber beim Verarbeiten riesiger Datenmengen, die sich nicht im Speicher unterbringen lassen (Out-of-Core-Training), sowie bei Online-Klassifikationsaufgaben nützlich.



Die Klasse `LinearSVC` regularisiert den Bias-Term, daher sollten Sie den Trainingsdatensatz durch Subtrahieren des Mittelwerts zunächst zentrieren. Dies geschieht automatisch, wenn Sie die Daten mit dem `StandardScaler` skalieren. Außerdem sollten Sie dafür sorgen, dass der Hyperparameter `loss` auf "hinge" gesetzt ist, da dies nicht der Standardeinstellung entspricht. Um schließlich eine kürzere Rechenzeit zu erreichen, sollten Sie den Hyperparameter `dual` auf `False` setzen, es sei denn, es gibt mehr Merkmale als Trainingsdatenpunkte (wir werden uns in diesem Kapitel noch mit Dualität beschäftigen).

Nichtlineare SVM-Klassifikation

Auch wenn lineare SVM-Klassifikatoren sehr effizient sind und in vielen Fällen überraschend gut funktionieren, sind viele Datensätze nicht einmal annähernd linear separierbar. Eine Möglichkeit zum Umgang mit nichtlinearen Datensätzen ist das Hinzufügen zusätzlicher Merkmale, wie etwa polynomiale Merkmale (wie in Kapitel 4); in manchen Fällen erhalten Sie dabei einen linear separierbaren Datensatz. Betrachten Sie das Diagramm auf der linken Seite von Abbildung 5-5: Es stellt einen einfachen Datensatz mit einem einzigen Merkmal x_1 dar. Dieser Datensatz ist, wie Sie sehen, nichtlinear separierbar. Wenn Sie aber ein zweites Merkmal $x_2 = (x_1)^2$ hinzufügen, ist der sich dabei ergebende zweidimensionale Datensatz ausgezeichnet linear separierbar.

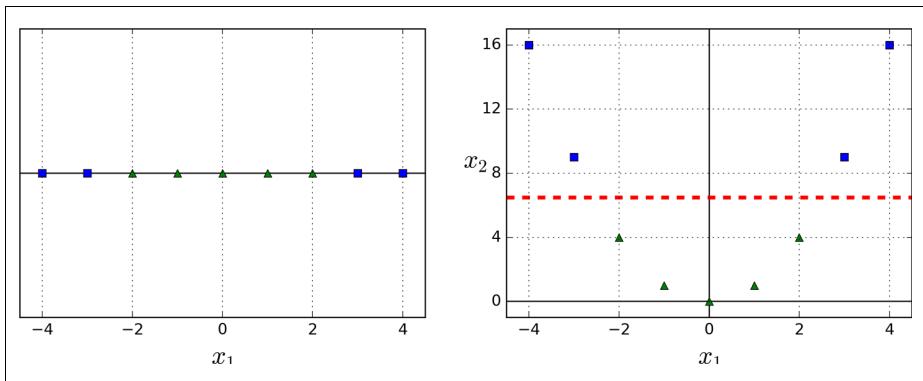


Abbildung 5-5: Hinzufügen von Merkmalen, um einen linear separierbaren Datensatz zu erhalten

Dieser Ansatz lässt sich mit Scikit-Learn umsetzen, indem Sie eine Pipeline erstellen, die aus einem `PolynomialFeatures`-Transformer (wie in Abschnitt »*Polynomiale Regression*« auf Seite 122 besprochen) und anschließend einem `StandardScaler` und einem `LinearSVC` besteht. Probieren wir dies anhand des Datensatzes `moons` aus (siehe Abbildung 5-6):

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```

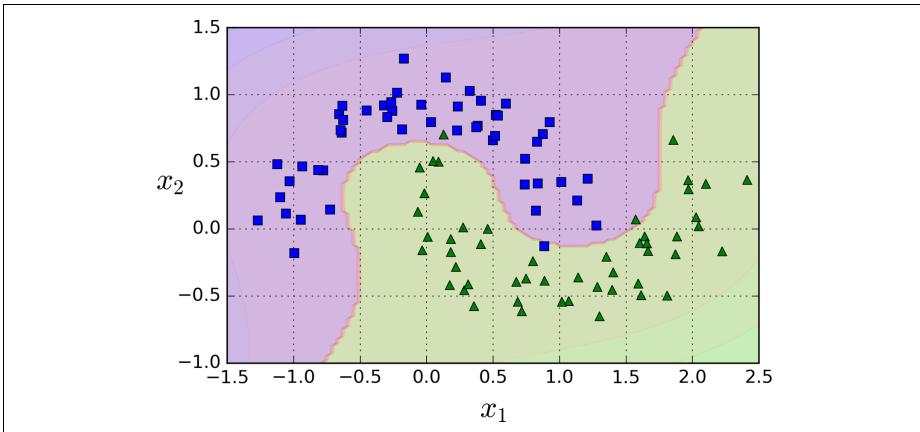


Abbildung 5-6: Linearer SVM-Klassifikator mit polynomiellen Merkmalen

Polynomieller Kernel

Das Hinzufügen polynomieller Merkmale lässt sich einfach umsetzen und funktioniert bei vielen Machine-Learning-Algorithmen sehr gut (nicht nur SVMs). Allerdings können Polynome niedrigen Grades nicht gut mit komplexen Datensätzen umgehen, höhergradige Polynome dagegen erzeugen eine riesige Anzahl Merkmale, wodurch das Modell zu langsam wird.

Glücklicherweise können wir bei SVMs eine beinahe magische mathematische Technik einsetzen, den *Kernel-Trick* (diesen stellen wir gleich vor). Dieser ermöglicht es, das gleiche Ergebnis wie beim Hinzufügen vieler polynomieller Merkmale zu erhalten, ohne diese explizit hinzuzufügen – das funktioniert auch für Polynome sehr hohen Grades. Dadurch umgehen wir die kombinatorische Explosion der Merkmalsanzahl, da wir überhaupt keine Merkmale hinzufügen müssen. Die Klasse SVC verwendet diesen Trick. Probieren wir ihn auf dem Datensatz moons aus:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

Dieser Code trainiert einen SVM-Klassifikator mit einem polynomiellen Kernel 3. Grades. Es ist auf der linken Seite von Abbildung 5-7 dargestellt. Auf der rechten Seite ist ein weiterer SVM-Klassifikator mit einem polynomiellen Kernel 10. Grades gezeigt. Natürlich müssen Sie den Grad der Polynome senken, falls Ihr Modell zum Overfitting neigt. Dementsprechend müssen Sie den Grad der Polynome erhöhen, falls Underfitting vorliegt. Der Hyperparameter `coef0` steuert, wie stark das Modell von den höhergradigen im Gegensatz zu den niedriggradigen Polynomialternen beeinflusst wird.

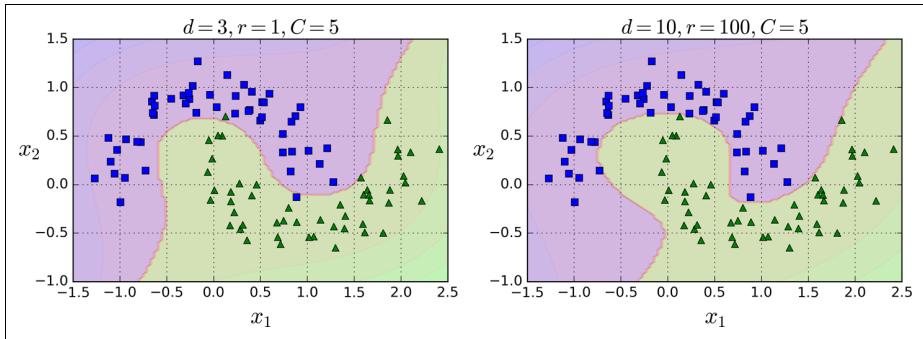


Abbildung 5-7: SVM-Klassifikatoren mit polynomiellem Kernel



Ein häufiger Ansatz zum Finden der richtigen Werte der Hyperparameter ist eine Gittersuche (siehe Kapitel 2). Es ist meist schneller, zunächst eine sehr grobe Gittersuche durchzuführen und anschließend die besten gefundenen Werte mit einer zweiten Gittersuche zu verfeinern. Ein gutes Gefühl dafür, was jeder der Hyperparameter bewirkt, hilft Ihnen außerdem dabei, den richtigen Teil des Hyperparameter-Raums zu durchkämmen.

Hinzufügen von Ähnlichkeitsbasierten Merkmalen

Eine weitere Technik zum Umgang mit nichtlinearen Aufgaben ist, mit einer *Ähnlichkeitsfunktion* berechnete Merkmale hinzuzufügen. Diese Funktion misst, wie ähnlich ein Datenpunkt zu einem bestimmten Orientierungspunkt, der *Landmarke*, ist. Betrachten wir beispielsweise den zuvor besprochenen eindimensionalen Datensatz und fügen wir zwei Landmarken bei $x_1 = -2$ und $x_1 = 1$ hinzu (siehe das linke Diagramm in Abbildung 5-8). Anschließend definieren wir die Ähnlichkeitsfunktion als Gaußsche *radiale Basisfunktion* (*RBF*) mit $\gamma = 0.3$ (siehe Formel 5-1).

Formel 5-1: Gaußsche RBF

$$\phi\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

Dies ist eine glockenförmige Funktion, die zwischen 0 (sehr weit von der Landmarke entfernt) und 1 (genau bei der Landmarke) liegt. Damit können wir die neuen Merkmale berechnen. Betrachten wir beispielsweise den Datenpunkt $x_1 = -1$: Er liegt im Abstand 1 zur ersten und im Abstand 2 zur zweiten Landmarke. Daher sind seine neuen Merkmale $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ und $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. Das Diagramm auf der rechten Seite von Abbildung 5-8 zeigt den transformierten Datensatz (ohne die ursprünglichen Merkmale). Wie Sie sehen, ist er nun linear separierbar.

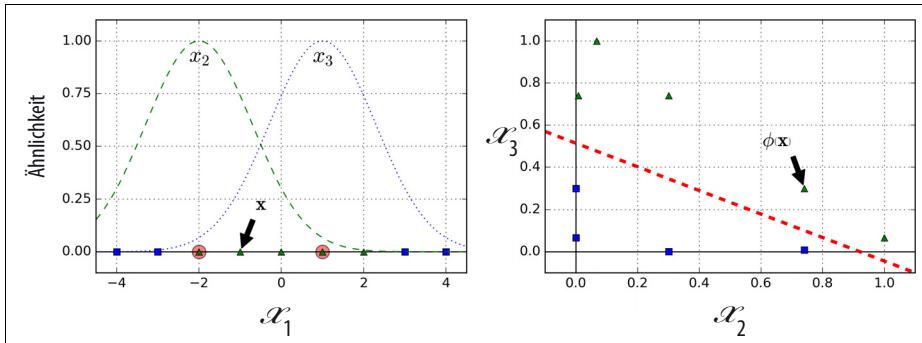


Abbildung 5-8: Ähnlichkeit von Merkmalen, berechnet mit der Gaußschen RBF

Sie fragen sich vielleicht, wie die Landmarken ausgesucht werden. Die einfachste Möglichkeit ist, einfach bei jedem einzelnen Datenpunkt eine Landmarke zu erzeugen. Dadurch entstehen sehr viele Dimensionen, und die Chance erhöht sich, dass der transformierte Trainingsdatensatz linear separierbar ist. Der Nachteil dieser Methode ist, dass ein Datensatz mit m Datenpunkten und n Merkmalen in einen Trainingsdatensatz mit m Datenpunkten und m Merkmalen umgewandelt wird (vorausgesetzt, Sie verwerfen die ursprünglichen Merkmale). Wenn Ihr Trainingsdatensatz sehr groß ist, erhalten Sie eine dementsprechend große Anzahl Merkmale.

Der Gaußsche RBF-Kernel

Wie die polynomiellen Merkmale können auch die ähnlichkeitsbasierten Merkmale bei jedem Machine-Learning-Algorithmus nützlich sein, es kann aber sehr rechenintensiv sein, alle zusätzlichen Merkmale zu berechnen, besonders wenn der Trainingsdatensatz sehr umfangreich ist. Auch hier kommt uns der magische Kernel-Trick der SVMs zur Hilfe: Er ermöglicht es, ein zu vielen ähnlichkeitsbasierten Merkmalen äquivalentes Ergebnis zu erhalten, ohne diese Merkmale hinzuzufügen. Probieren wir den Gaußschen RBF-Kernel mit der Klasse SVC aus:

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
])
rbf_kernel_svm_clf.fit(X, y)
```

Dieses Modell ist links unten in Abbildung 5-9 dargestellt. Die übrigen Diagramme zeigen mit anderen Einstellungen der Hyperparameter γ und C berechnete Modelle. Durch das Erhöhen von γ werden die glockenförmigen Kurven schmäler (linkes Diagramm in Abbildung 5-9). Dadurch wird der Einfluss jedes Datenpunkts geringer: Die Entscheidungsgrenze wird unregelmäßiger und schlängelt sich um einzelne Datenpunkte herum. Ein kleiner Wert für γ dagegen verbreitert die Glockenkurve, sodass die Datenpunkte einen großen Einflussbereich haben und die Entscheidungsgrenze weicher wird. Damit verhält sich γ wie ein

Regularisierungsparameter: Wenn Ihr Modell zum Overfitting neigt, sollten Sie diesen Wert verringern, wenn es zum Underfitting neigt, sollten Sie ihn erhöhen (ähnlich zum Hyperparameter C).

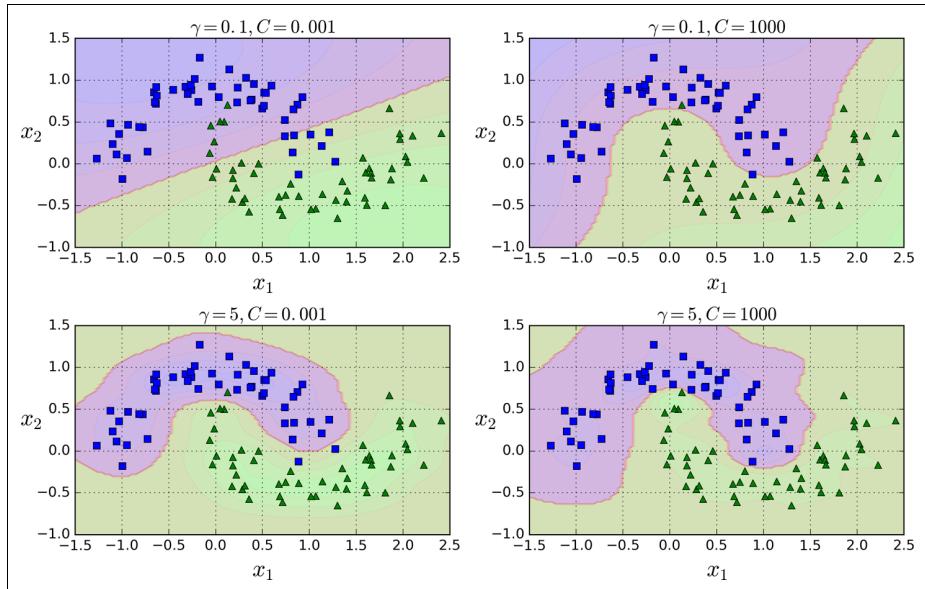


Abbildung 5-9: SVM-Klassifikatoren mit RBF-Kernel

Es gibt noch weitere Kernels, aber diese kommen weitaus seltener zum Einsatz. Beispielsweise sind manche dieser Kernels auf bestimmte Datenstrukturen spezialisiert. *String Kernels* finden sich bisweilen bei der Klassifikation von Textdokumenten oder DNA-Sequenzen (z. B. dem *string subsequence-Kernel* oder Kernels, die auf der *Levenshtein-Distanz* aufbauen).



Wie sollen Sie sich bei so vielen möglichen Kernels für einen entscheiden? Als Faustregel gilt: Sie sollten immer zuerst den linearen Kernel ausprobieren (`LinearSVC`). Dieser ist viel schneller als `SVC(kernel="linear")`), besonders bei sehr umfangreichen Trainingsdaten oder sehr vielen Merkmalen. Wenn der Trainingsdatensatz nicht zu groß ist, sollten Sie auch den Gaußschen RBF-Kernel ausprobieren; er funktioniert in den meisten Fällen. Wenn Sie dann noch Zeit und Rechenkapazität übrig haben, können Sie mit einigen anderen Kernels, Kreuzvalidierung und Gittersuche experimentieren, insbesondere wenn es für Ihre Datenstruktur spezialisierte Kernels gibt.

Komplexität der Berechnung

Die Klasse `LinearSVC` verwendet die Bibliothek *liblinear*, die einen optimierten Algorithmus (<http://goo.gl/R635CH>) für lineare SVMs enthält.¹ Sie unterstützt den Kernel-

Trick nicht, skaliert aber annähernd linear mit der Größe des Trainingsdatensatzes und der Anzahl Merkmale: Seine zeitliche Komplexität ist in etwa $O(m \times n)$.

Der Algorithmus benötigt länger, wenn eine sehr hohe Präzision erforderlich ist. Dies lässt sich über den Toleranz-Hyperparameter ϵ (tol in Scikit-Learn) einstellen. Für die meisten Klassifikationsaufgaben genügt die voreingestellte Toleranz.

Die Klasse SVC dagegen verwendet die Bibliothek *libsvm*, deren Algorithmus (<http://goo.gl/a8HkE3>) den Kernel-Trick unterstützt.² Die zeitliche Komplexität liegt meist zwischen $O(m^2 \times n)$ und $O(m^3 \times n)$. Leider bedeutet dies, dass der Algorithmus bei großen Trainingsdatensätzen (z.B. Hunderttausenden Datenpunkten) unsäglich langsam wird. Dieser Algorithmus ist bestens für komplexe, aber kleine und mittelgroße Trainingsdatensätze geeignet. Er skaliert jedoch gut mit der Anzahl Merkmale, insbesondere bei *dünn besetzten Merkmalen* (wenn also jeder Datenpunkt nur wenige Merkmale ungleich null aufweist). In diesem Fall skaliert der Algorithmus in etwa mit der durchschnittlichen Anzahl Nicht-Null-Merkmale pro Datenpunkt. Tabelle 5-1 vergleicht die Klassen zur SVM-Klassifikation in Scikit-Learn miteinander.

Tabelle 5-1: Vergleich der Klassen zur SVM-Klassifikation in Scikit-Learn

Klasse	zeitl. Komplexität	Out-of-Core möglich	Scaling nötig	Kernel-Trick
LinearSVC	$O(m \times n)$	Nein	Ja	Nein
SGDClassifier	$O(m \times n)$	Ja	Ja	Nein
SVC	$O(m^2 \times n)$ bis $O(m^3 \times n)$	Nein	Ja	Ja

SVM-Regression

Wie bereits erwähnt, ist der SVM-Algorithmus recht flexibel: Er ermöglicht nicht nur die lineare und nichtlineare Klassifikation, sondern auch lineare und nichtlineare Regression. Dabei wird ein umgekehrtes Ziel verfolgt: Anstatt die breitestmögliche Straße zwischen zwei Kategorien zu fitten und Verletzungen dieser Grenze zu minimieren, versucht die SVM-Regression, so viele Datenpunkte wie möglich *auf* der Straße zu platzieren und Grenzverletzungen (diesmal Datenpunkte *abseits* der Straße) zu minimieren. Die Breite der Straße wird über den Hyperparameter ϵ gesteuert. Abbildung 5-10 zeigt zwei lineare SVM-Regressionsmodelle, die mit zufälligen linearen Daten trainiert wurden, das eine mit breitem Margin ($\epsilon = 1.5$) und das andere mit schmalem Margin ($\epsilon = 0.5$).

Das Hinzufügen von weiteren Trainingsdatenpunkten innerhalb des Margin beeinflusst die Vorhersagen des Modells nicht; man bezeichnet dieses Modell daher als ϵ -insensitiv.

1 »A Dual Coordinate Descent Method for Large-scale Linear SVM«, Lin et al. (2008).

2 »Sequential Minimal Optimization (SMO)«, J. Platt (1998).

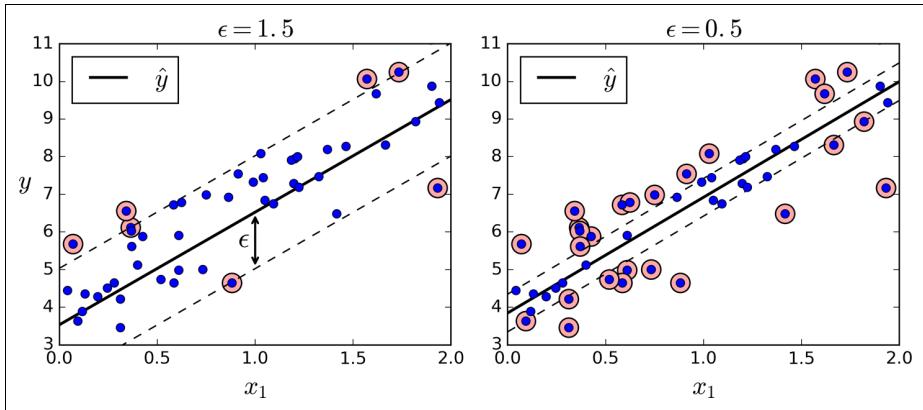


Abbildung 5-10: SVM-Regression

Sie können die lineare SVM-Regression mit der Scikit-Learn-Klasse `LinearSVR` durchführen. Der folgende Code erstellt das auf der linken Seite von Abbildung 5-10 dargestellte Modell (die Trainingsdaten müssen dazu skaliert und zentriert sein):

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Zum Bearbeiten nichtlinearer Regressionsaufgaben können Sie ein Kernel-SVM-Modell verwenden. Beispielsweise zeigt Abbildung 5-11 eine SVM-Regression mit einem polynomiellen Kernel 2. Grades auf einem zufälligen quadratischen Trainingsdatensatz. Im linken Diagramm gibt es wenig Regularisierung (ein großer Wert für C) und im rechten Diagramm sehr viel mehr Regularisierung (ein kleiner Wert für C).

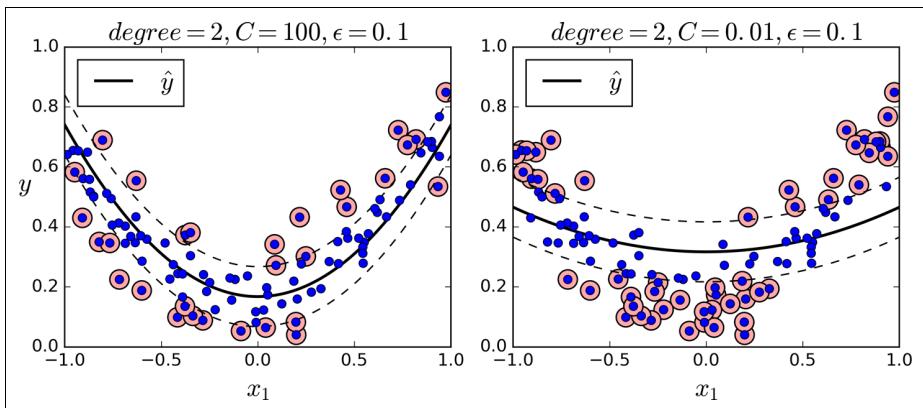


Abbildung 5-11: SVM-Regression mit einem polynomiellen Kernel 2. Grades

Der folgende Code erstellt das auf der linken Seite von Abbildung 5-11 dargestellte Modell. Dazu wird die Scikit-Learn-Klasse SVR verwendet (die den Kernel-Trick unterstützt). Die Klasse SVR ist das Pendant der Klasse SVC für Regressionsaufgaben, und die Klasse LinearSVR ist das Pendant zur Klasse LinearSVC. Die Klasse LinearSVR skaliert linear mit der Größe des Trainingsdatensatzes (wie die Klasse LinearSVC), die Klasse SVR wird dagegen mit einem wachsenden Trainingsdatensatz sehr langsam (wie die Klasse SVC).

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



SVMs lassen sich auch zum Erkennen von Ausreißern einsetzen; Details dazu finden Sie in der Dokumentation zu Scikit-Learn.

Hinter den Kulissen

In diesem Abschnitt erklären wir, wie SVMs Vorhersagen treffen und wie ihr Trainingsalgorithmus funktioniert. Wir beginnen mit linearen SVM-Klassifikatoren. Wenn Sie Machine Learning einfach nur ausprobieren möchten, können Sie diesen Abschnitt ruhig überspringen, mit den Übungen am Ende des Kapitels weitermachen und hierher zurückkehren, sobald Sie SVMs besser verstehen möchten.

Zu Beginn müssen wir einige Schreibweisen klären: In Kapitel 4 haben wir alle Modellparameter im Vektor θ platziert, darunter den Bias-Term θ_0 und die Gewichte der Eingabemerkmale θ_1 bis θ_n und haben allen Datenpunkten die Bias-Eingabe $x_0 = 1$ hinzugefügt. In diesem Kapitel werden wir eine andere, bei SVMs bequemere (und übliche) Notation verwenden: Den Bias-Term bezeichnen wir als b und den Vektor mit den Gewichten der Merkmale als w . Wir müssen diesmal zum Merkmalsvektor kein Bias-Merkmal hinzufügen.

Entscheidungsfunktion und Vorhersagen

Der lineare SVM-Klassifikator sagt die Kategorie eines neuen Datenpunkts x vorher, indem er einfach die Entscheidungsfunktion $w^T \cdot x + b = w_1 x_1 + \dots + w_n x_n + b$ berechnet: Ist das Ergebnis positiv, wird als Ergebnis \hat{y} die positive Kategorie (1) vorhergesagt, andernfalls die negative Kategorie (0); siehe Formel 5-2.

Formel 5-2: Vorhersage eines linearen SVM-Klassifikators

$$\hat{y} = \begin{cases} 0 & \text{wenn } w^T \cdot x + b < 0, \\ 1 & \text{wenn } w^T \cdot x + b \geq 0 \end{cases}$$

Abbildung 5-12 zeigt die Entscheidungsfunktion, die dem Modell auf der rechten Seite von Abbildung 5-4 entspricht: Es ist eine zweidimensionale Ebene, da dieser Datensatz zwei Merkmale besitzt (Breite und Länge der Kronblätter). Die Entscheidungsgrenze ist die Menge aller Punkte, bei denen die Entscheidungsfunktion gleich 0 ist: Sie ist die Schnittmenge zweier Ebenen und damit eine Gerade (als dicke durchgezogene Linie dargestellt).³

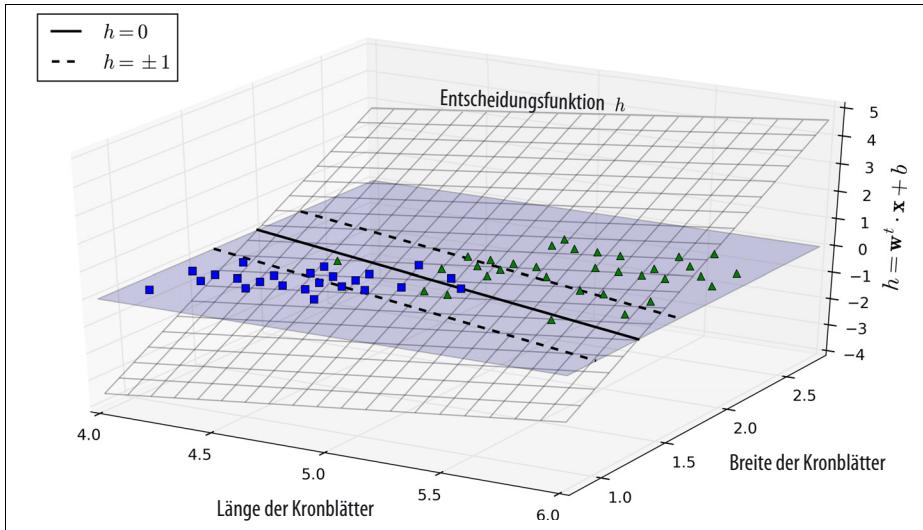


Abbildung 5-12: Entscheidungsfunktion für den Iris-Datensatz

Die gestrichelten Linien stehen für Punkte, bei denen die Entscheidungsfunktion 1 oder -1 beträgt: Sie befinden sich parallel und im gleichen Abstand zur Entscheidungsgrenze und bilden einen Rand (Margin) um sie herum. Beim Trainieren eines linearen SVM-Klassifikators wird nach einem Wert für w und b gesucht, für den dieser Margin so breit wie möglich wird und gleichzeitig Überschreitungen vermieden werden (Hard-Margin) oder nur in begrenztem Maße auftreten (Soft-Margin).

Zielfunktionen beim Trainieren

Betrachten wir die Steigung der Entscheidungsfunktion: Sie entspricht der Norm des Gewichtsvektors $\|w\|$. Wenn wir diese Steigung durch 2 teilen, so sind die Punkte, bei denen die Entscheidungsfunktion ± 1 beträgt, doppelt so weit von der Entscheidungsgrenze entfernt. Anders ausgedrückt führt das Teilen der Steigung durch 2 zu einer Multiplikation des Margin mit dem Faktor 2. Dies lässt sich etwas besser in 2-D veranschaulichen, wie in Abbildung 5-13 gezeigt. Je kleiner der Gewichtsvektor w , umso größer wird der Margin.

³ Wenn es n Merkmale gibt, ist die Entscheidungsfunktion allgemein eine n -dimensionale Hyperebene, und die Entscheidungsgrenze ist eine $(n - 1)$ -dimensionale Hyperebene.

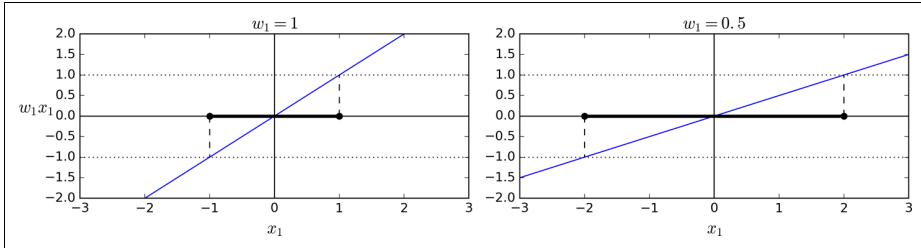


Abbildung 5-13: Ein kleinerer Gewichtsvektor führt zu einem größeren Margin

Wir möchten also $\|\mathbf{w}\|$ minimieren, um einen großen Margin zu erhalten. Wenn wir allerdings jegliche Verletzungen des Margin vermeiden möchten (Hard-Margin), muss die Entscheidungsfunktion für alle positiven Trainingsdatenpunkte größer als 1 sein und für alle negativen Punkte kleiner als -1. Wenn wir für negative Datenpunkte $t^{(i)} = -1$ (wenn $y^{(i)} = 0$ gilt) und für positive Datenpunkte $t^{(i)} = 1$ festlegen (wenn $y^{(i)} = 1$ gilt), so können wir diese Bedingung für alle Datenpunkte durch $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ ausdrücken.

Daher lässt sich die Zielfunktion eines linearen SVM-Klassifikators mit Hard-Margin durch das *Optimierungsproblem mit Nebenbedingungen* in Formel 5-3 schreiben.

Formel 5-3: Zielfunktion eines linearen SVM-Klassifikators mit Hard-Margin

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimiere}} \quad \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \\ & \text{unter der Bedingung} \quad t^{(i)} \left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b \right) \geq 1 \quad \text{für } i = 1, 2, \dots, m \end{aligned}$$



Wir minimieren $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$, was gleich $\frac{1}{2} \|\mathbf{w}\|^2$ ist, anstatt $\|\mathbf{w}\|$ zu minimieren. Dies würde zum gleichen Ergebnis führen (da die Werte von \mathbf{w} und b , die einen Wert minimieren, auch dessen halbes Quadrat minimieren), aber $\frac{1}{2} \|\mathbf{w}\|^2$ besitzt eine einfachere Ableitung (sie beträgt einfach nur \mathbf{w}), während $\|\mathbf{w}\|$ bei $\mathbf{w} = 0$ nicht differenzierbar ist. Algorithmen zur Optimierung laufen mit differenzierbaren Funktionen viel besser.

Um die Zielfunktion für Soft-Margin zu erhalten, müssen wir für jeden Datenpunkt eine *Slack-Variable* $\zeta^{(i)} \geq 0$ einführen⁴: $\zeta^{(i)}$ bestimmt, inwieweit der i te Datenpunkt den Margin verletzen darf. Wir haben nun zwei gegenläufige Ziele: die Slack-Variablen so klein wie möglich zu machen, um Verletzungen des Margin zu verringern, und $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$ so klein wie möglich zu machen, um den Margin zu vergrößern. An dieser Stelle kommt der Hyperparameter C ins Spiel: Er erlaubt uns, die Balance zwischen diesen beiden Zielen festzulegen. Damit erhalten wir das Optimierungsproblem in Formel 5-4.

4 Ζeta (ζ) ist der 8. Buchstabe des griechischen Alphabets.

Formel 5-4: Zielfunktion eines linearen SVM-Klassifikators mit Soft-Margin

$$\begin{aligned} \text{minimiere } & \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ \text{unter der Bedingung } & t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{und} \quad \zeta^{(i)} \geq 0 \quad \text{für } i = 1, 2, \dots, m \end{aligned}$$

Quadratische Programme

Sowohl das Hard-Margin- als auch das Soft-Margin-Problem gehören zu den konvexen quadratischen Optimierungsproblemen mit linearen Nebenbedingungen. Solche Probleme bezeichnet man als *Quadratische Programme* (QP). Es sind zahlreiche Solver erhältlich, die QP-Probleme mit verschiedenen Techniken lösen können. Diese aufzuführen, sprengt den Rahmen dieses Buchs.⁵ Die allgemeine Formulierung des Problems ist durch Formel 5-5 gegeben.

Formel 5-5: Formulierung Quadratischer Programme

$$\begin{aligned} \underset{\mathbf{p}}{\text{Minimiere}} \quad & \frac{1}{2} \mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ \text{unter der Bedingung } & \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \\ \text{wobei } & \left\{ \begin{array}{ll} \mathbf{p} & \text{ein } n_p\text{-dimensionaler Vektor } (n_p = \text{Anzahl Parameter}), \\ \mathbf{H} & \text{eine } n_p \times n_p\text{-Matrix}, \\ \mathbf{f} & \text{ein } n_p\text{-dimensionaler Vektor}, \\ \mathbf{A} & \text{eine } n_c \times n_p\text{-Matrix } (n_c = \text{Anzahl Nebenbedingungen}), \\ \mathbf{b} & \text{ein } n_c\text{-dimensionaler Vektor ist.} \end{array} \right. \end{aligned}$$

Beachten Sie, dass der Ausdruck $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ genau n_c Nebenbedingungen definiert: $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ mit $i = 1, 2, \dots, n_c$, wobei $\mathbf{a}^{(i)}$ ein Vektor mit den Elementen der i^{ten} Zeile von \mathbf{A} und $b^{(i)}$ das i^{te} Element von \mathbf{b} ist.

Sie können leicht nachweisen, dass Sie mit den folgenden QP-Parametern die Zielfunktion für einen linearen Hard-Margin-SVM-Klassifikator erhalten:

- $n_p = n + 1$, wobei n die Anzahl Merkmale ist (das +1 steht für den Bias-Term).
- $n_c = m$, wobei m der Anzahl Trainingsdatenpunkte entspricht.
- \mathbf{H} ist eine $n_p \times n_p$ -Identitätsmatrix, außer dass die Zelle in der linken oberen Ecke eine Null enthält (um den Bias-Term zu ignorieren).
- $\mathbf{f} = \mathbf{0}$, ein n_p -dimensionaler Vektor voller Nullen.
- $\mathbf{b} = \mathbf{1}$, ein n_c -dimensionaler Vektor voller Einsen.
- $\mathbf{a}^{(i)} = -t^{(i)} \hat{\mathbf{x}}^{(i)}$, wobei $\hat{\mathbf{x}}^{(i)}$ gleich $\mathbf{x}^{(i)}$ mit dem zusätzlichen Bias-Merkmal $\hat{\mathbf{x}}_0 = 1$ ist.

Demnach können Sie einen linearen Hard-Margin-SVM-Klassifikator trainieren, indem Sie einen QP-Solver von der Stange verwenden und ihm die oben angegebene

⁵ Um mehr über Quadratische Programme zu erfahren, können Sie Stephen Boyd und Lieven Vandenberghe lesen, *Convex Optimization* (<http://goo.gl/FGXuLw>, Cambridge, UK: Cambridge University Press, 2004) oder sich eine Reihe Vorlesungsvideos (<http://goo.gl/rTo3Af>) von Richard Brown ansehen.

nen Parameter übergeben. Der als Ergebnis erhaltene Vektor \mathbf{p} enthält den Bias-Term $b = p_0$ und die Gewichte der Merkmale $w_i = p_i$ mit $i = 1, 2, \dots, m$. In ähnlicher Weise können Sie einen QP-Solver einsetzen, um ein Soft-Margin-Problem zu lösen (Übungen dazu finden Sie am Ende dieses Kapitels).

Um jedoch den Kernel-Trick zu verwenden, werden wir uns eine andere Art Optimierungsproblem mit Nebenbedingungen ansehen.

Das duale Problem

Bei einem als *primales Problem* bekannten Optimierungsproblem ist es möglich, dieses als ein eng verwandtes Problem, nämlich dessen *duales Problem* zu formulieren. Die Lösung des dualen Problems legt normalerweise eine Untergrenze für die Lösung des primalen Problems fest, aber unter gewissen Umständen kann es auch die gleichen Lösungen wie das primale Problem haben. Glücklicherweise ist dies beim Optimierungsproblem einer SVM der Fall,⁶ sodass Sie sich aussuchen können, ob Sie das primale Problem oder das duale Problem lösen möchten; beide haben die gleiche Lösung. Formel 5-6 zeigt die duale Form der Zielfunktion für eine lineare SVM (wenn Sie an der Herleitung des dualen Problems aus dem primalen Problem interessiert sind, sehen Sie sich Anhang C an).

Formel 5-6: Duale Form der Zielfunktion eines linearen SVM

$$\text{minimiere}_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)}^T \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

unter der Bedingung $\alpha^{(i)} \geq 0 \quad \text{für } i = 1, 2, \dots, m$

Wenn Sie erst einmal den Vektor $\hat{\alpha}$ gefunden haben, der diese Gleichung minimiert (mit einem QP-Solver), können Sie $\hat{\mathbf{w}}$ und \hat{b} berechnen, die das primale Problem über Formel 5-7 minimieren.

Formel 5-7: Von der dualen Lösung zur primalen Lösung

$$\begin{aligned}\hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{i=1}^m \left(1 - t^{(i)} \left(\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)} \right) \right) \\ \hat{\alpha}^{(i)} &> 0\end{aligned}$$

Das duale Problem lässt sich schneller als das primale lösen, wenn die Anzahl Trainingsdatenpunkte kleiner als die Anzahl der Merkmale ist. Bedeutender ist aber,

⁶ Die Zielfunktion ist konvex, und die Nebenbedingungen sind stetig differenzierbare und konvexe Funktionen.

dass es den Kernel-Trick ermöglicht, was mit dem primalen nicht funktioniert. Was also ist dieser Kernel-Trick überhaupt?

Kernel-SVM

Nehmen wir an, Sie möchten mit zweidimensionalen Trainingsdaten (wie dem Datensatz moons) eine polynomische Transformation 2. Ordnung durchführen und anschließend einen linearen SVM-Klassifikator auf den transformierten Daten trainieren. Formel 5-8 zeigt die zu verwendende polynomische Zuordnungsfunktion 2. Grades ϕ .

Formel 5-8: polynomische Zuordnung 2. Grades

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Beachten Sie, dass der transformierte Vektor drei statt der ursprünglichen zwei Dimensionen besitzt. Betrachten wir nun, was bei einer Anzahl zweidimensionaler Vektoren \mathbf{a} und \mathbf{b} passiert, wenn wir diese polynomische Zuordnung 2. Grades anwenden und anschließend das Skalarprodukt der transformierten Vektoren berechnen (siehe Formel 5-9).

Formel 5-9: Kernel-Trick bei einer polynomischen Zuordnung 2. Grades

$$\begin{aligned} \phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2 \end{aligned}$$

Was halten Sie davon? Das Skalarprodukt der transformierten Vektoren entspricht dem Quadrat des Skalarprodukts der ursprünglichen Vektoren: $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$.

An dieser Stelle stellt sich die entscheidende Erkenntnis ein: Wenn Sie die Transformation ϕ auf alle Trainingsdatenpunkte anwenden, enthält das duale Problem (siehe Formel 5-6) das Skalarprodukt $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$. Wenn aber ϕ die in Formel 5-8 definierte polynomische Transformation 2. Ordnung ist, dann können Sie dieses Skalarprodukt der transformierten Vektoren einfach durch $(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^2$ ersetzen. Also brauchen Sie die Trainingsdaten überhaupt nicht zu transformieren: Ersetzen Sie einfach das Skalarprodukt in Formel 5-6 durch sein Quadrat. Das Ergebnis ist

exakt das gleiche, als hätten Sie sich die Mühe gemacht, die Trainingsdaten tatsächlich zu transformieren und anschließend einen linearen SVM-Algorithmus auszuführen. Mit diesem Trick wird der gesamte Prozess rechnerisch wesentlich effizienter. Darin besteht die Essenz des Kernel-Tricks.

Die Funktion $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ nennt man einen *polynomiellen Kernel* 2. Grades. Beim Machine Learning versteht man unter einem *Kernel* eine Funktion, mit der sich das Skalarprodukt $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$ lediglich aus den ursprünglichen Vektoren \mathbf{a} und \mathbf{b} berechnen lässt, ohne dass die Transformation ϕ berechnet werden (oder überhaupt bekannt sein) muss. In Formel 5-10 sind einige der gebräuchlichsten Kernels aufgelistet.

Formel 5-10: Gebräuchliche Kernels

$$\text{Linear: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomiel: } K(\mathbf{a}, \mathbf{b}) = \left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r \right)^d$$

$$\text{Gaußsche RBF: } K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$$

$$\text{Sigmoid: } K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)$$

Mercers Theorem

Laut *Mercers Theorem* muss, wenn eine Funktion $K(\mathbf{a}, \mathbf{b})$ einige mathematische Bedingungen, die *Mercer-Bedingungen* (K muss stetig sein, seine Parameter symmetrisch, sodass gilt $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$ und so weiter), erfüllt, auch eine Funktion ϕ existieren, die \mathbf{a} und \mathbf{b} in einen anderen Raum abbildet (der möglicherweise sehr viel mehr Dimensionen aufweist), sodass gilt: $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$. Damit können Sie K als Kernel einsetzen, da Sie ja wissen, dass ϕ existiert, selbst wenn Sie ϕ nicht genau kennen. Im Falle des Gaußschen RBF-Kernels lässt sich nachweisen, dass ϕ jeden Trainingsdatenpunkt in einen Raum mit unendlich vielen Dimensionen transformiert, es ist also gut, dass Sie die Zuordnung nicht vornehmen müssen!

Einige häufig eingesetzte Kernels (wie der sigmoide Kernel) erfüllen nicht alle Mercer-Bedingungen. In der Praxis funktionieren sie dennoch gut.

Um ein loses Ende müssen wir uns noch kümmern. Formel 5-7 zeigt, wie wir im Falle eines linearen SVM-Klassifikators von einer dualen Lösung zur primalen Lösung gelangen. Wenn Sie aber den Kernel-Trick anwenden, erhalten Sie Gleichungen mit $\phi(x^{(i)})$. Tatsächlich muss $\hat{\mathbf{w}}$ die gleiche Anzahl Dimensionen wie $\phi(x^{(i)})$ aufweisen. Diese Zahl kann sehr groß oder sogar unendlich sein und ist daher nicht berechenbar. Wie aber können wir Vorhersagen treffen, ohne $\hat{\mathbf{w}}$ zu kennen? Die gute Nachricht ist, dass wir hier die Formel für $\hat{\mathbf{w}}$ aus Formel 5-7 in die Entscheidungsfunktion für einen neuen Datenpunkt $\mathbf{x}^{(n)}$ einsetzen können und eine Formel

erhalten, die ausschließlich aus Skalarprodukten zwischen den Eingabevektoren besteht. Damit ist wieder der Kernel-Trick einsetzbar (Formel 5-11).

Formel 5-11: Vorhersagen mit einem Kernel-SVM treffen

$$\begin{aligned}
 h_{\hat{\mathbf{w}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \hat{\mathbf{w}}^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{j=1}^m \hat{\alpha}^{(i)} t^{(i)} (\phi(\mathbf{x}^{(i)})) \right)^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} \\
 &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \left(\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(n)}) \right) + \hat{b} \\
 &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b} \\
 \hat{\alpha}^{(i)} > 0
 \end{aligned}$$

Weil $\alpha^{(i)} \neq 0$ nur für die Stützvektoren gilt, muss für Vorhersagen das Skalarprodukt zwischen dem neuen Eingabevektor $\mathbf{x}^{(n)}$ und den Stützvektoren anstatt mit sämtlichen Trainingsdatenpunkten berechnet werden. Natürlich müssen Sie auch hier den Bias-Term \hat{b} mit dem gleichen Trick berechnen (Formel 5-12).

Formel 5-12: Berechnen des Bias-Terms mithilfe des Kernel-Tricks

$$\begin{aligned}
 \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \hat{\mathbf{w}}^T \cdot \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \cdot \phi(\mathbf{x}^{(i)}) \right) \\
 &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)
 \end{aligned}$$

Falls Sie hiervon Kopfschmerzen bekommen, ist das völlig normal: Dies ist eine Nebenwirkung des Kernel-Tricks.

Online-SVMs

Bevor wir dieses Kapitel beenden, schauen wir uns noch kurz Online-SVM-Klassifikatoren an (zur Erinnerung: Online-Learning bedeutet, inkrementell zu lernen, üblicherweise beim Eintreffen neuer Daten).

Bei linearen SVM-Klassifikatoren lässt sich das Gradientenverfahren einsetzen (z. B. mit der Klasse `SGDClassifier`), um die vom primalen Problem abgeleitete Kostenfunktion in Formel 5-13 zu minimieren. Leider konvergiert diese wesentlich langsamer als die auf QP basierenden Methoden.

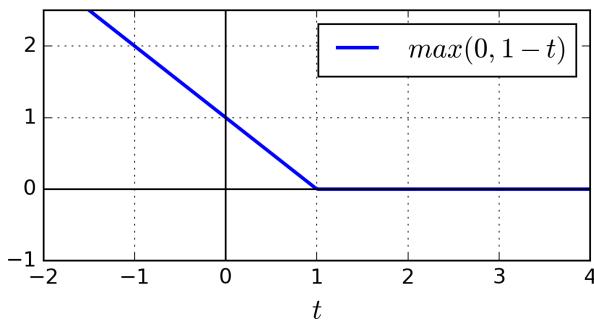
Formel 5-13: Kostenfunktion eines linearen SVM-Klassifikators

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max\left(0, 1 - t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b)\right)$$

Die erste Summe in dieser Kostenfunktion verleiht dem Modell einen kleinen Gewichtsvektor \mathbf{w} , der den Margin vergrößert. Die zweite Summe berechnet die Gesamtheit sämtlicher Verletzungen des Margin. Die Verletzung des Margin ist für einen Datenpunkt dann gleich 0, wenn er abseits der Straße und auf der richtigen Seite liegt, anderenfalls ist der Verletzungsbetrag proportional zur Entfernung von der richtigen Straßenseite. Das Minimieren dieses Terms stellt sicher, dass das Modell so wenige und so kleine Verletzungen wie möglich zulässt.

Hinge Loss

Die Funktion $\max(0, 1 - t)$ nennt man die *Hinge Loss*-Funktion (siehe unten). Sie ist gleich 0 für $t \geq 1$. Ihre Ableitung (Steigung) beträgt -1 für $t < 1$ und 0 für $t > 1$. Sie ist bei $t = 1$ nicht differenzierbar, aber wie bei der Lasso-Regression (siehe Abschnitt »Lasso-Regression« auf Seite 131) können Sie dennoch das Gradientenverfahren verwenden, indem Sie ein beliebiges *Subdifferential* bei $t = 1$ berechnen (d.h. einen beliebigen Wert zwischen -1 und 0).



Es ist ebenfalls möglich, Online-Kernel-SVMs zu implementieren – beispielsweise mit den Methoden »Incremental and Decremental SVM Learning« (<http://goo.gl/JEqVui>)⁷ oder »Fast Kernel Classifiers with Online and Active Learning« (<https://goo.gl/hsoUHA>)⁸. Diese sind allerdings in Matlab und C++ implementiert. Für größere nichtlineare Aufgaben sollten Sie stattdessen neuronale Netze in Betracht ziehen (siehe Teil II).

⁷ »Incremental and Decremental Support Vector Machine Learning«, G. Cauwenberghs, T. Poggio (2001).

⁸ »Fast Kernel Classifiers with Online and Active Learning«, A. Bordes, S. Ertekin, J. Weston, L. Bottou (2005).

Übungen

1. Was ist die Support Vector Machines zugrunde liegende Idee?
2. Was ist ein Stützvektor?
3. Warum ist es wichtig, beim Verwenden von SVMs die Eingabedaten zu skalieren?
4. Kann ein SVM-Klassifikator einen Konfidenzwert ausgeben, wenn er einen Datenpunkt klassifiziert? Wie sieht es mit einer Wahrscheinlichkeit aus?
5. Sollten Sie die primale oder die duale Form des SVM-Problems verwenden, um ein Modell mit Millionen Datenpunkten und Hunderten Merkmalen zu trainieren?
6. Nehmen wir an, Sie hätten einen SVM-Klassifikator mit RBF-Kernel trainiert. Es sieht so aus, als würde Underfitting der Trainingsdaten vorliegen: Sollten Sie γ (gamma) erhöhen oder senken? Wie sieht es mit C aus?
7. Wie sollten Sie die QP-Parameter (\mathbf{H} , \mathbf{f} , \mathbf{A} und \mathbf{b}) setzen, um ein lineares SVM-Klassifikationsproblem mit Soft-Margin mit einem herkömmlichen QP-Solver zu lösen?
8. Trainieren Sie einen `LinearSVC` auf linear separierbaren Daten. Trainieren Sie anschließend einen `SVC` und einen `SGDClassifier` auf dem gleichen Datensatz. Schauen Sie, ob Sie beide dazu bringen können, ein in etwa gleiches Modell zu berechnen.
9. Trainieren Sie einen SVM-Klassifikator auf dem MNIST-Datensatz. Da SVM-Klassifikatoren binäre Klassifikatoren sind, müssen Sie die One-versus-All-Strategie einsetzen, um alle zehn Ziffern zu klassifizieren. Sie müssen eventuell zur Optimierung der Hyperparameter kleinere Datensätze zur Validierung verwenden, um den Vorgang zu beschleunigen. Was für eine Genauigkeit erreichen Sie?
10. Trainieren Sie einen SVM-Regressor auf dem Datensatz zu Immobilienpreisen in Kalifornien.

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

KAPITEL 6

Entscheidungsbäume

Wie SVMs sind auch *Entscheidungsbäume* sehr flexible Machine-Learning-Algorithmen, die sich sowohl für Klassifikations- als auch Regressionsaufgaben eignen. Sogar Aufgaben mit multiplen Ausgaben lassen sich mit ihnen lösen. Es sind sehr mächtige Algorithmen, mit denen sich komplexe Datensätze fitten lassen. Beispielsweise haben Sie in Kapitel 2 ein Modell mit dem `DecisionTreeRegressor` trainiert und an den Datensatz zu kalifornischen Immobilien perfekt angepasst (genauer gesagt, overfittet).

Entscheidungsbäume sind außerdem die funktionelle Komponente von Random Forests (siehe Kapitel 7), die zu den mächtigsten heute verfügbaren Machine-Learning-Algorithmen gehören.

In diesem Kapitel werden wir besprechen, wie sich Entscheidungsbäume trainieren, visualisieren und für Vorhersagen einsetzen lassen. Anschließend werden wir den in Scikit-Learn verwendeten CART-Trainingsalgorithmus durchgehen. Wir werden betrachten, wie sich Entscheidungsbäume regularisieren und für Regressionsaufgaben einsetzen lassen. Schließlich werden wir einige Einschränkungen von Entscheidungsbäumen kennenlernen.

Trainieren und Visualisieren eines Entscheidungsbaums

Um Entscheidungsbäume zu verstehen, werden wir zunächst einen erstellen und uns ansehen, wie dieser Vorhersagen trifft. Der folgende Code trainiert einen `DecisionTreeClassifier` auf dem Iris-Datensatz (siehe Kapitel 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # Länge und Breite der Kronblätter
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Sie können den trainierten Entscheidungsbaum visualisieren, indem Sie durch Aufrufen der Methode `export_graphviz()` eine Datei namens `iris_tree.dot` mit einer Repräsentation als Graph erzeugen:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

Diese `.dot`-Datei lässt sich anschließend mit dem Kommandozeilenprogramm `dot` aus dem Paket `graphviz` in verschiedene Formate wie PDF oder PNG umwandeln.¹ Der folgende Konsolenbefehl wandelt die `.dot`-Datei in ein `.png`-Bild um:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Ihr erster Entscheidungsbaum ist der in Abbildung 6-1 dargestellte.

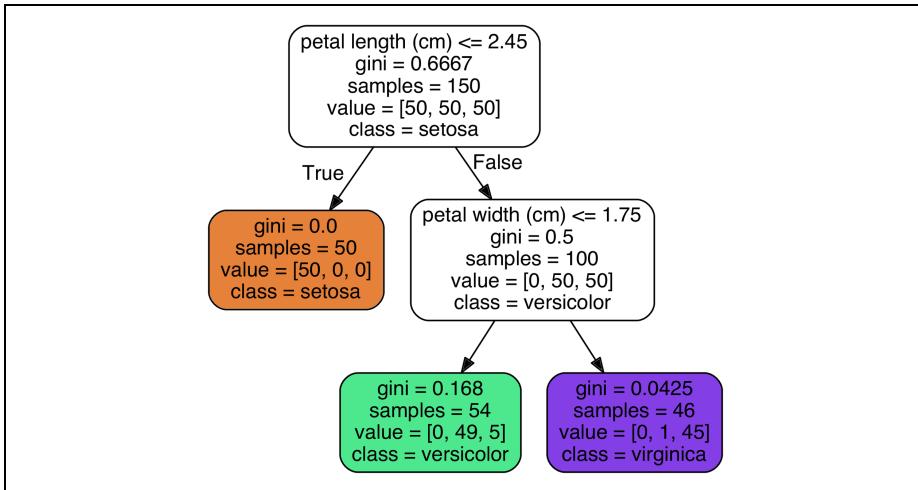


Abbildung 6-1: Iris-Entscheidungsbaum

Vorhersagen treffen

Betrachten wir nun, wie der in Abbildung 6-1 dargestellte Baum Vorhersagen trifft. Nehmen wir an, Sie finden eine Iris-Blüte und möchten diese klassifizieren. Sie beginnen an der *Wurzel* des Baums (bei depth 0, ganz oben): Dieser Knoten stellt

¹ Graphviz ist ein Open-Source-Softwarepaket zur Visualisierung von Graphen und unter <http://www.graphviz.org/> verfügbar.

die Frage, ob das Kronblatt der Blüte kürzer als 2.45 cm ist. Ist dies der Fall, fahren Sie mit dem Kind auf der linken Seite fort (depth 1, links). In diesem Fall ist der Knoten ein *Blatt* (es besitzt keine weiteren Kinder) und stellt keine weiteren Fragen: Sie übernehmen einfach die von diesem Knoten vorhergesagte Kategorie, somit sagt unser Entscheidungsbaum vorher, dass Ihre Blüte eine Iris-Setosa (class=setosa) ist.

Nehmen wir an, Sie finden eine weiter Blüte, bei der das Kronblatt diesmal länger als 2.45 cm ist. Sie fahren mit dem rechten Kind der Wurzel fort (bei depth 1, rechts). Dieser Knoten ist kein Blatt, sondern stellt eine weitere Frage: Ist das Kronblatt schmäler als 1.75 cm? Ist dies der Fall, ist Ihre Blüte vermutlich eine Iris-Versicolor (bei depth 2, links). Wenn nicht, ist sie vermutlich eine Iris-Virginica (bei depth 2, rechts). Es ist tatsächlich so einfach.



Einer der vielen Vorteile von Entscheidungsbäumen ist, dass sie sehr wenig Vorbereitung der Daten erfordern. Insbesondere ist keinerlei Skalierung oder Zentrierung von Merkmalen notwendig.

Das Attribut `samples` eines Knotens zählt, für wie viele Trainingsdatenpunkte dieser gültig ist. Beispielsweise haben 100 Datenpunkte ein Kronblatt mit einer Länge von mindestens 2.45 cm (bei depth 1, rechts), davon haben 54 ein Kronblatt mit einer Breite von weniger als 1.75 cm (bei depth 2, links). Das Attribut `value` eines Knotens verrät uns, wie viele Trainingsdatenpunkte der Knoten in jeder Kategorie enthält: Beispielsweise enthält der Knoten rechts unten 0 Exemplare von Iris-Setosa, 1 Iris-Versicolor und 45 Iris-Virginica. Schließlich misst das Attribut `gini` die *Unreinheit* eines Knotens: Ein Knoten gilt als »rein« (`gini=0`), wenn sämtliche enthaltenen Datenpunkte der gleichen Kategorie angehören. Da beispielsweise der linke Knoten bei depth-1 ausschließlich Instanzen von Iris-Setosa enthält, ist er rein und besitzt einen `gini`-Score von 0. Formel 6-1 stellt dar, wie der Trainingsalgorithmus den `gini`-Score G_i für den i^{ten} Knoten berechnet. Beispielsweise ist der `gini`-Score für den linken Knoten bei depth-2 gleich $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Wir werden in Kürze ein weiteres Maß für *Unreinheit* kennenlernen.

Formel 6-1: *Gini-Unreinheit*

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

- $p_{i,k}$ ist dabei der Anteil von Instanzen der Kategorie k an den Datenpunkten im Knoten i .



Scikit-Learn verwendet den CART-Algorithmus, der ausschließlich *Binärbäume* erzeugt: Innere Knoten haben stets zwei Kinder (d.h., die Fragen lassen sich nur mit Ja oder Nein beantworten). Es existieren aber auch andere Algorithmen wie ID3, die Entscheidungsbäume erzeugen, deren Knoten mehr als zwei Kinder haben können.

Abbildung 6-2 zeigt die Entscheidungsgrenzen dieses Entscheidungsbaums. Die dicke vertikale Linie steht für die Entscheidungsgrenze der Wurzel (depth 0): petal length = 2.45 cm. Da das Gebiet auf der linken Seite rein ist (ausschließlich Iris-Setosa), lässt es sich nicht weiter aufteilen. Das Gebiet auf der rechten Seite ist dagegen unrein, daher teilt es der Knoten bei depth-1 auf der Höhe petal width = 1.75 cm auf (als gestrichelte Linie dargestellt). Da `max_depth` auf 2 gesetzt wurde, endet der Entscheidungsbaum an dieser Stelle. Wenn Sie allerdings `max_depth` auf 3 setzen, würden die zwei Knoten bei depth-2 jeweils eine zusätzliche Entscheidungsgrenze produzieren (hier durch die gepunkteten Linien dargestellt).

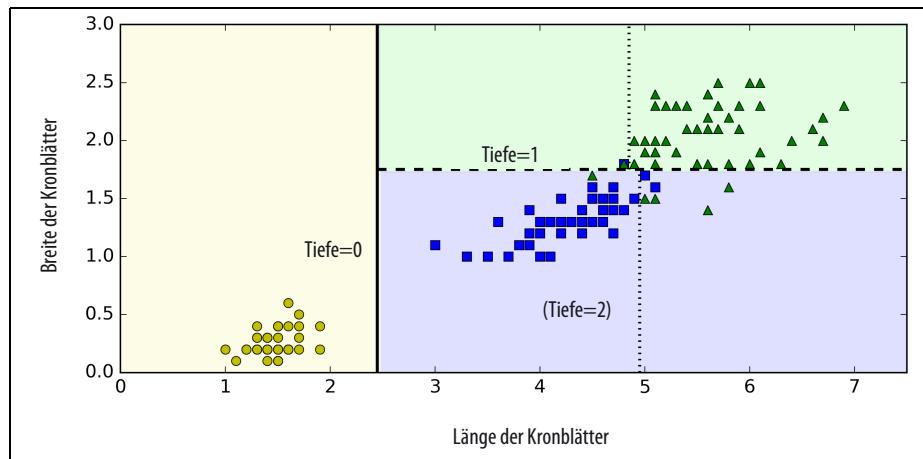


Abbildung 6-2: Entscheidungsgrenzen in einem Entscheidungsbaum

Interpretation von Modellen: White Box im Vergleich zu Blackbox

Wie Sie sehen, sind Entscheidungsbäume recht einfach nachzuvollziehen und ihre Entscheidungen leicht interpretierbar. Solche Modelle werden oft als *White-Box-Modelle* bezeichnet. Im Gegensatz dazu gehören, wie wir noch sehen werden, Random Forests oder neuronale Netze im Allgemeinen zu den *Blackbox-Modellen*. Sie treffen zwar ausgezeichnete Vorhersagen, und Sie können die dazu durchgeföhrten Berechnungen leicht nachprüfen; trotzdem ist es in der Regel schwierig, in wenigen Worten zu erklären, warum die Vorhersagen in einer bestimmten Weise getroffen wurden. Wenn beispielsweise ein neuronales Netzwerk behauptet, dass eine bestimmte Person auf einem Bild zu sehen ist, kann man nur schwer erkennen, worauf sich diese Vorhersage stützt: Hat das Modell die Augen der Person erkannt? Den Mund? Die Nase? Die Schuhe? Oder gar das Sofa, auf dem die Person saß? Umgekehrt geben uns Entscheidungsbäume einfache, klare Regeln zur Klassifikation, die notfalls sogar von Hand durchgeführt werden können (z.B. bei der Klassifikation von Blüten).

Schätzen von Wahrscheinlichkeiten für Kategorien

Ein Entscheidungsbaum kann auch die Wahrscheinlichkeit für die Zugehörigkeit eines Datenpunkts zu einer Kategorie k abschätzen: Zuerst schreitet das Verfahren den Baum ab, um das Blatt für diesen Datenpunkt zu finden, und gibt dann den Anteil der Trainingsdatenpunkte der Kategorie k in diesem Knoten zurück. Nehmen wir an, Sie hätten eine Blüte mit 5 cm langen und 1.5 cm breiten Kronblättern entdeckt. Der dazu passende Knoten im Entscheidungsbaum ist der Knoten bei depth-2 auf der linken Seite, daher sollte der Entscheidungsbaum die folgenden Wahrscheinlichkeiten ausgeben: 0% für Iris-Setosa (0/54), 90.7% für Iris-Versicolor (49/54) und 9.3% für Iris-Virginica (5/54). Wenn Sie nach einer Vorhersage der Kategorie fragen, erhalten Sie natürlich Iris-Versicolor (Kategorie 1), da diese die höchste Wahrscheinlichkeit hat. Überprüfen wir dies:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfekt! Beachten Sie, dass die geschätzten Wahrscheinlichkeiten überall in der unteren rechten Ecke von Abbildung 6-2 die gleichen sind – sogar wenn die Kronblätter beispielsweise 6 cm lang und 1.5 cm breit wären (obwohl es sich dann höchstwahrscheinlich um eine Iris-Virginica handeln würde).

Der CART-Trainings-Algorithmus

Scikit-Learn verwendet den *Classification and Regression Tree*-Algorithmus (CART-Algorithmus), um Entscheidungsbäume zu trainieren (oder »anzubauen«). Er folgt einer sehr einfachen Grundidee: Der Algorithmus teilt die Trainingsdaten zunächst anhand eines Merkmals k und eines Schwellenwerts t_k in zwei Untermengen auf (z. B. »petal length ≤ 2.45 cm«). Wie werden k und t_k ausgewählt? Der Algorithmus sucht nach dem Paar (k, t_k) , das die reinsten (nach deren Größe gewichteten) Untermengen hervorbringt. Dabei versucht der Algorithmus, die Kostenfunktion in Formel 6-2 zu minimieren.

Formel 6-2: Kostenfunktion des CART-Algorithmus zur Klassifikation

$$J(k, t_k) = \frac{m_{\text{links}}}{m} G_{\text{links}} + \frac{m_{\text{rechts}}}{m} G_{\text{rechts}}$$

wobei $\begin{cases} G_{\text{links/rechts}} & \text{die Unreinheit der linken/rechten Untermenge misst,} \\ m_{\text{links/rechts}} & \text{die Anzahl Datenpunkte in der linken/rechten Untermenge ist.} \end{cases}$

Sobald der Trainingsdatensatz erfolgreich zweigeteilt wurde, werden die Untermengen nach dem gleichen Verfahren weiter aufgeteilt. Dies setzt sich rekursiv fort, bis die (durch den Hyperparameter `max_depth` angegebene) maximale Tiefe erreicht wurde oder keine Aufteilung gefunden werden kann, die die Unreinheit weiter

reduziert. Andere Hyperparameter steuern weitere Abbruchbedingungen (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` und `max_leaf_nodes`). Diese erklären wir in Kürze.



Wie Sie sehen, gehört der CART-Algorithmus zu den Greedy-Algorithmen: Er sucht gierig nach einer optimalen Aufteilung auf der höchsten möglichen Ebene und setzt diesen Vorgang auf jeder Stufe fort. Es wird nicht geprüft, ob eine Aufteilung einige Schritte weiter zur höchsten möglichen Unreinheit führt. Ein Greedy-Algorithmus liefert oft eine recht gute Lösung, diese muss aber nicht die bestmögliche sein.

Leider gehört das Finden des optimalen Baums zu den *NP-vollständigen* Problemen:² Es erfordert eine Zeit von $O(\exp(m))$, wodurch das Problem selbst für eher kleine Datensätze praktisch unlösbar wird. Daher müssen wir uns mit einer »annehmbar guten« Lösung zufriedengeben.

Komplexität der Berechnung

Das Treffen von Vorhersagen erfordert das Abschreiten eines Entscheidungsbaums von der Wurzel zu einem Blatt. Entscheidungsbäume sind halbwegs ausbalanciert, daher erfordert das Abschreiten etwa $O(\log_2(m))$ Knoten.³ Da bei jedem Knoten nur ein Merkmal geprüft werden muss, ist die Komplexität der Vorhersage lediglich $O(\log_2(m))$, egal wie viele Merkmale es gibt. Die Vorhersagen sind also auch für große Trainingsdatensätze sehr schnell.

Allerdings vergleicht der Trainingsalgorithmus bei jedem Knoten sämtliche Merkmale (falls `max_features` gesetzt ist auch weniger) aller Datenpunkte miteinander. Dies führt zu einer Komplexität von $O(n \times m \log(m))$ beim Trainieren. Bei kleinen Trainingsdatensätzen (weniger als einige Tausend Datenpunkte) kann Scikit-Learn das Trainieren beschleunigen, indem es die Daten vorsortiert (mit `presort=True`). Bei größeren Trainingsdatensätzen verlangsamst dies das Trainieren aber deutlich.

2 P ist die Menge der in polynomieller Zeit lösbar Probleme. NP ist die Menge der Probleme, deren Lösungen sich in polynomieller Zeit überprüfen lassen. Ein NP-schweres Problem ist ein Problem, zu dem sich jedes NP-Problem in polynomieller Zeit reduzieren lässt. Ein NP-vollständiges Problem ist sowohl NP als auch NP-schwer. Ob P = NP gilt, ist eine wichtige mathematische Frage. Falls P ≠ NP gilt (was wahrscheinlich erscheint), kann es für kein NP-vollständiges Problem jemals einen polynomiellen Algorithmus geben (außer vielleicht auf einem Quantencomputer).

3 \log_2 ist der binäre Logarithmus. Er entspricht $\log_2(m) = \log(m) / \log(2)$.

Gini-Unreinheit oder Entropie?

Standardmäßig wird die Gini-Unreinheit verwendet, Sie können aber stattdessen auch die *Entropie* als Maß für die Unreinheit auswählen, indem Sie den Hyperparameter criterion auf "entropy" setzen. Der aus der Thermodynamik stammende Begriff Entropie beschreibt Unordnung auf molekularer Ebene: Die Entropie nähert sich null, wenn Moleküle unbeweglich und wohlgeordnet sind. Dieses Konzept hat sich später in andere Fachgebiete ausgebreitet, darunter Shannons *Informationstheorie*, wo sie den durchschnittlichen Informationsgehalt einer Nachricht misst:⁴ Wenn alle Nachrichten identisch sind, beträgt die Entropie null. Im Machine Learning wird die Entropie oft als Maß für die Unreinheit eingesetzt: Die Entropie einer Menge ist null, wenn Sie nur aus Datenpunkten einer Kategorie besteht. Formel 6-3 zeigt die Definition der Entropie des i^{ten} Knotens. Beispielsweise beträgt die Entropie für den linken Knoten bei depth-2 in Abbildung 6-1 genau $-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31$.

Formel 6-3: Entropie

$$H_i = - \sum_{k=1}^n p_{i,k} \log(p_{i,k})$$

$p_{i,k} \neq 0$

Sollten Sie also die Gini-Unreinheit oder die Entropie verwenden? Tatsächlich macht es meist keinen großen Unterschied: Beide ergeben ähnliche Bäume. Die Gini-Unreinheit lässt sich ein wenig schneller berechnen und eignet sich daher als Standardwert. Wenn beide Maße voneinander abweichen, neigt die Gini-Unreinheit dazu, die häufigste Kategorie in einem eigenen Ast des Baums abzusondern, wohingegen die Entropie etwas ausbalanciertere Bäume erzeugt.⁵

Hyperparameter zur Regularisierung

Entscheidungsbäume treffen sehr wenige Annahmen über die Trainingsdaten (im Gegensatz zu beispielsweise linearen Modellen, die offensichtlich annehmen, dass sich die Daten linear verhalten). Sich selbst überlassen, passt sich die Struktur des Baums sehr genau an die Trainingsdaten an und führt höchstwahrscheinlich zu Overfitting. Solch ein Modell wird auch als *parameterfreies Modell* bezeichnet, nicht weil es keine Parameter gäbe (meist gibt es viele), sondern weil die Anzahl der Parameter nicht vor dem Trainieren festgelegt wird. Daher ist es dem Modell über-

4 Ein Reduzieren der Entropie wird oft als *Zugewinn an Information* bezeichnet.

5 Details finden Sie in einer interessanten Analyse von Sebastian Raschka (<http://goo.gl/UndTrO>).

lassen, sich eng an die Daten anzupassen. Im Gegensatz dazu besitzt ein *parametrisches Modell* wie etwa ein lineares Modell eine im Voraus festgelegte Anzahl Parameter. Es verfügt daher über eine begrenzte Anzahl Freiheitsgrade, wodurch es weniger zu Overfitting neigt (aber dafür ein höheres Risiko für Underfitting besteht).

Um ein Overfitting der Trainingsdaten zu vermeiden, müssen Sie die Freiheitsgrade eines Entscheidungsbaums beim Trainieren einschränken. Wie Sie inzwischen wissen, nennt man dies Regularisierung. Die Hyperparameter zur Regularisierung hängen vom verwendeten Algorithmus ab. Im Allgemeinen können Sie aber mindestens die maximale Tiefe des Entscheidungsbaums begrenzen. In Scikit-Learn lässt sich dies über den Hyperparameter `max_depth` erreichen (die Voreinstellung ist `None`, eine unbegrenzte Tiefe). Ein Reduzieren von `max_depth` regularisiert das Modell und verringert damit das Risiko einer Überanpassung.

Die Klasse `DecisionTreeClassifier` bietet einige weitere Parameter, die die Form des Entscheidungsbaums in ähnlicher Weise einschränken: `min_samples_split` (die minimale Anzahl von Datenpunkten, die ein Knoten aufweisen muss, damit er aufgeteilt werden kann), `min_samples_leaf` (die minimale Anzahl Datenpunkte, die ein Blatt haben muss), `min_weight_fraction_leaf` (wie `min_samples_leaf`, aber als Anteil der gesamten gewichteten Datenpunkte), `max_leaf_nodes` (maximale Anzahl Blätter) und `max_features` (maximale Anzahl beim Aufteilen eines Knotens berücksichtigter Merkmale). Ein Erhöhen der `min_*`-Hyperparameter und ein Senken der `max_*`-Hyperparameter regularisiert das Modell.



Andere Algorithmen trainieren Entscheidungsbäume zunächst ohne Einschränkungen, entfernen aber anschließend überflüssige Knoten (*Pruning*). Ein Knoten wird als überflüssig angesehen, wenn seine Kinder ausschließlich Blätter sind und der durch ihn erbrachte Zugewinn an Reinheit nicht *statistisch signifikant* ist. Standardisierte statistische Tests wie der χ^2 -Test schätzen die Wahrscheinlichkeit ab, dass eine Verbesserung rein zufällig erfolgt ist (dies bezeichnet man als die *Nullhypothese*). Wenn diese Wahrscheinlichkeit, genannt *p-Wert*, höher als ein eingestellter Schwellenwert ist (typischerweise 5%, durch einen Hyperparameter steuerbar), dann wird der Knoten als überflüssig erachtet und seine Kinder gelöscht. Dieses Pruning wird fortgesetzt, bis alle überflüssigen Knoten entfernt worden sind.

Abbildung 6-3 zeigt zwei auf dem Datensatz `moons` (aus Kapitel 5) trainierten Entscheidungsbäume. Der Entscheidungsbaum auf der linken Seite wurde mit den voreingestellten Hyperparametern trainiert (also ohne Restriktionen), der auf der rechten Seite mit `min_samples_leaf=4`. Es wird schnell deutlich, dass das Modell auf der linken Seite overfittet ist und das Modell auf der rechten Seite vermutlich besser verallgemeinert.

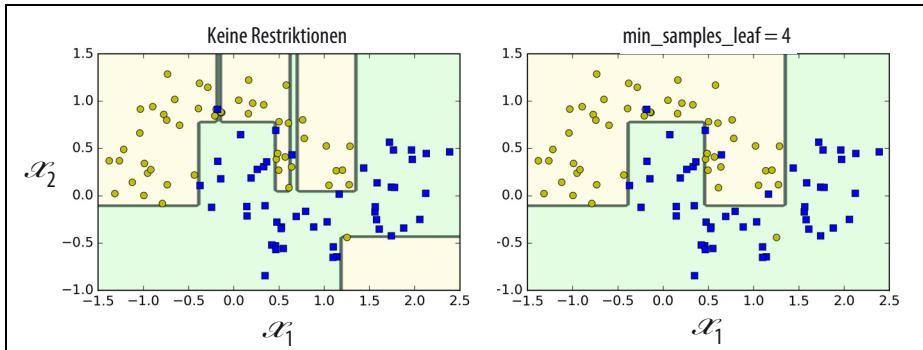


Abbildung 6-3: Regularisierung mit `min_samples_leaf`

Regression

Entscheidungsbäume können auch Regressionsaufgaben bewältigen. Erstellen wir mit der Klasse `DecisionTreeRegressor` aus Scikit-Learn einen Regressionsbaum und trainieren diesen auf einem verrauschten quadratischen Datensatz mit `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

Der dabei erhaltene Baum ist in Abbildung 6-4 dargestellt.

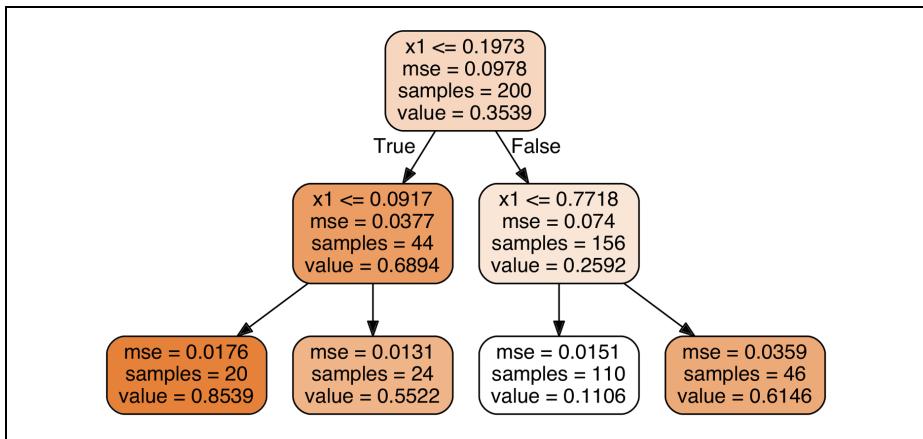


Abbildung 6-4: Ein Entscheidungsbaum zur Regression

Dieser Baum sieht dem zuvor zur Klassifikation erstellten Baum sehr ähnlich. Der Hauptunterschied ist, dass er anstelle einer Kategorie in jedem Knoten einen Wert vorhersagt. Wenn Sie beispielsweise eine Vorhersage für einen neuen Datenpunkt

bei $x_1 = 0.6$ treffen möchten, schreiten Sie den Baum von der Wurzel ausgehend ab. Sie erreichen irgendwann das Blatt mit der Vorhersage $\text{value}=0.1106$. Diese Vorhersage ist nichts weiter als der durchschnittliche Zielwert der 110 Trainingsdatenpunkte in diesem Blatt. Diese Vorhersage führt zu einem mittleren quadratischen Fehler (MSE) von 0.0151 in diesen 110 Datenpunkten.

Die Vorhersagen dieses Modells sind auf der linken Seite von Abbildung 6-5 dargestellt. Wenn Sie `max_depth=3` setzen, erhalten Sie die auf der rechten Seite dargestellten Vorhersagen. Beachten Sie, dass der vorhergesagte Wert in jedem Abschnitt dem durchschnittlichen Zielwert der Datenpunkte in diesem Abschnitt entspricht. Der Algorithmus teilt jeden Abschnitt so auf, dass möglichst viele Trainingsdatenpunkte so nah wie möglich am vorhergesagten Wert liegen.

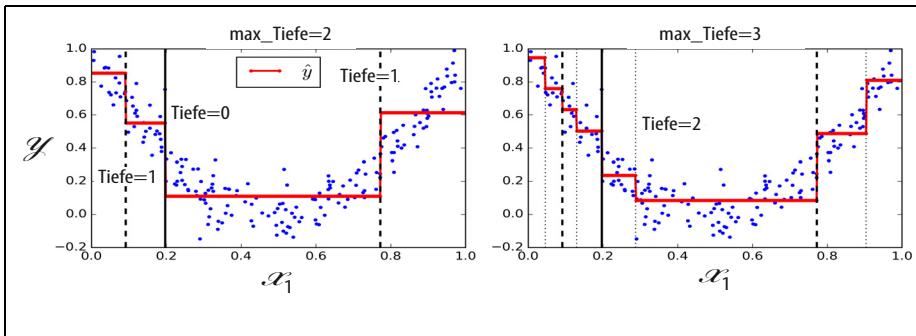


Abbildung 6-5: Vorhersagen zweier Entscheidungsbäume zur Regression

Der CART-Algorithmus funktioniert weitgehend wie oben vorgestellt. Der einzige Unterschied besteht darin, dass der Trainingsdatensatz so aufgeteilt wird, dass der MSE anstatt der Unreinheit minimiert wird. Formel 6-4 zeigt die vom Algorithmus minimierte Kostenfunktion.

Formel 6-4: CART-Kostenfunktion für die Regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{links}} + \frac{m_{\text{rechts}}}{m} \text{MSE}_{\text{rechts}} \quad \text{wobei} \quad \begin{cases} \text{MSE}_{\text{Knoten}} = \sum_{i \in \text{Knoten}} (\hat{y}_{\text{Knoten}} - y^{(i)})^2 \\ \hat{y}_{\text{Knoten}} = \frac{1}{m_{\text{Knoten}}} \sum_{i \in \text{Knoten}} y^{(i)} \end{cases}$$

Wie bei Klassifikationsaufgaben sind auch Entscheidungsbäume für Regressionsaufgaben anfällig für Overfitting. Ohne jegliche Regularisierung (z.B. mit den voreingestellten Hyperparametern) erhalten Sie die Vorhersagen auf der linken Seite von Abbildung 6-6. Dies ist offensichtlich ein schwerwiegendes Overfitting der Trainingsdaten. Durch Setzen von `min_samples_leaf=10` erhalten Sie ein weitaus sinnvollereres Modell, das Sie auf der rechten Seite von Abbildung 6-6 sehen.

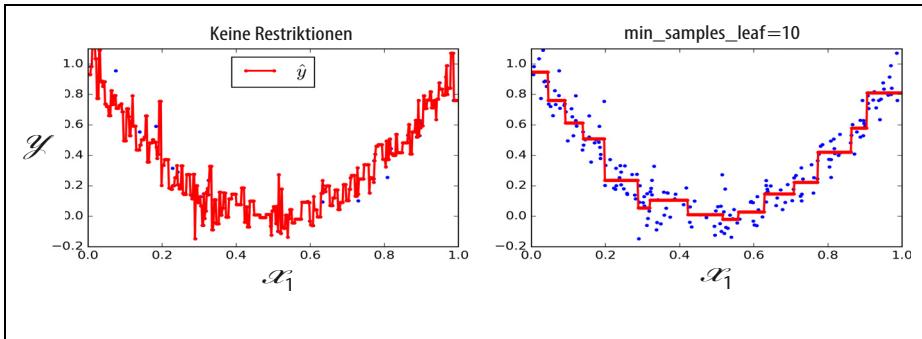


Abbildung 6-6: Regularisierung eines Regressionsbaums

Instabilität

Sie sind hoffentlich mittlerweile davon überzeugt, dass sehr vieles für Entscheidungsbäume spricht: Sie sind einfach zu verstehen und zu interpretieren, leicht anwendbar und mächtig. Sie haben allerdings auch einige Schwachstellen: Erstens haben Sie womöglich bereits festgestellt, dass Entscheidungsbäume orthogonale Entscheidungsgrenzen lieben (alle Unterteilungen stehen im rechten Winkel zu einer Achse). Dadurch reagieren sie empfindlich auf Rotationen der Trainingsdaten. Als Beispiel zeigt Abbildung 6-7 einen einfachen linear separierbaren Datensatz: Auf der linken Seite kann der Entscheidungsbau diese einfacher unterteilen. Auf der rechten Seite wirkt die Entscheidungsgrenze nach einer Drehung des Datensatzes um 45° unnötig verschachtelt. Obwohl beide Entscheidungsbäume die Trainingsdaten perfekt abbilden, verallgemeinert das Modell auf der rechten Seite vermutlich nicht besonders gut. Dieses Problem lässt sich über eine Hauptkomponentenzerlegung angehen (siehe Kapitel 8), die häufig zu einer besseren Ausrichtung der Trainingsdaten führt.

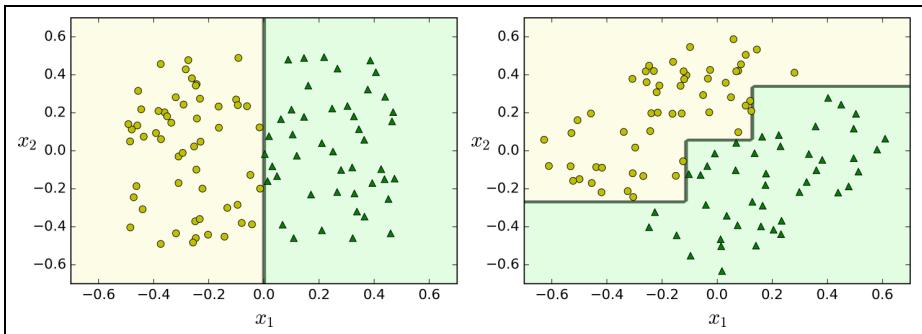


Abbildung 6-7: Empfindlichkeit für Rotation der Trainingsdaten

Allgemeiner gesprochen ist das Hauptproblem bei Entscheidungsbäumen, dass sie sehr empfindlich auf kleine Variationen in den Trainingsdaten reagieren. Wenn Sie beispielsweise einfach die breiteste Iris-Versicolor aus dem Iris-Trainingsdatensatz entfernen (diejenige mit 4.8 cm langen und 1.8 cm breiten Kronblättern) und einen neuen Entscheidungsbaum trainieren, erhalten Sie das in Abbildung 6-8 gezeigte Modell. Wie Sie sehen, unterscheidet es sich sehr stark vom vorigen Entscheidungsbaum (Abbildung 6-2). Genauer gesagt, da der von Scikit-Learn verwendete Trainingsalgorithmus stochastisch⁶ arbeitet, können Sie sogar mit den gleichen Trainingsdaten sehr unterschiedliche Modelle erhalten (es sei denn, Sie setzen den Hyperparameter `random_state`).

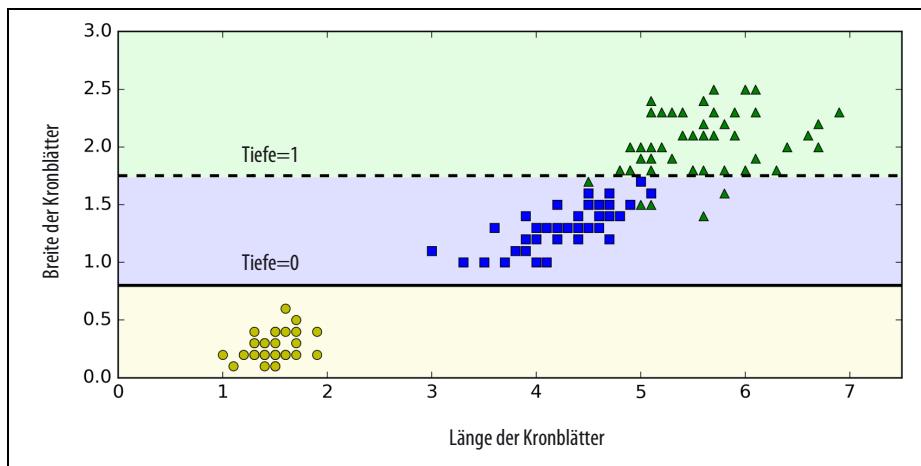


Abbildung 6-8: Anfälligkeit für Einzelheiten der Trainingsdaten

Random Forests wirken dieser Instabilität entgegen, indem sie die Vorhersagen vieler Bäume mitteln, wie wir im nächsten Kapitel sehen werden.

Übungen

- Was ist die ungefähre Tiefe eines mit 1 Million Datenpunkten (ohne Restriktionen) trainierten Entscheidungsbaums?
- Ist die Gini-Unreinheit eines Knotens im Allgemeinen geringer oder größer als die seines Elternteils? Ist sie *im Allgemeinen* kleiner/größer oder *immer* kleiner/größer?
- Sollte man versuchen, `max_depth` zu senken, wenn ein Entscheidungsbaum einen Trainingsdatensatz overfittet?

⁶ Die bei jedem Knoten zu evaluierenden Merkmale werden zufällig ausgewählt.

4. Sollte man versuchen, die Eingabemerkmale zu skalieren, wenn ein Entscheidungsbaum die Trainingsdaten underfittet?
5. Wenn es eine Stunde dauert, einen Entscheidungsbaum mit 1 Million Datenpunkten zu trainieren, wie lange etwa wird das Trainieren eines weiteren Baums mit 10 Millionen Datenpunkten dauern?
6. Wenn Ihr Trainingsdatensatz aus 100000 Datenpunkten besteht, beschleunigt das Setzen von `presort=True` das Trainieren?
7. Trainieren und optimieren Sie einen Entscheidungsbaum für den Datensatz `moons`.
 - a. Erzeugen Sie einen `moons`-Datensatz mit `make_moons(n_samples=10000, noise=0.4)`.
 - b. Teilen Sie ihn mit `train_test_split()` in einen Trainings- und einen Testdatensatz auf.
 - c. Verwenden Sie die Gittersuche mit Kreuzvalidierung (mithilfe der Klasse `GridSearchCV`), um gute Einstellungen für die Hyperparameter eines `DecisionTreeClassifier` zu finden. Hinweis: Probieren Sie unterschiedliche Werte für `max_leaf_nodes`.
 - d. Trainieren Sie den Baum mit den vollständigen Trainingsdaten und bestimmen Sie die Qualität Ihres Modells auf den Testdaten. Sie sollten eine Genauigkeit zwischen 85 % und 87% erhalten.
8. Züchten Sie einen Wald.
 - a. Erzeugen Sie im Anschluss an die vorige Aufgabe 1000 Untermengen Ihres Trainingsdatensatzes mit jeweils 100 zufällig ausgewählten Datenpunkten. Hinweis: Sie können dazu die Klasse `ShuffleSplit` aus Scikit-Learn verwenden.
 - b. Trainieren Sie auf jedem der Teildatensätze einen Entscheidungsbaum mit den besten oben gefundenen Hyperparametern. Werten Sie diese 1000 Entscheidungsbäume auf den Testdaten aus. Da sie mit kleineren Datensätzen trainiert wurden, schneiden diese Bäume mit nur ca. 80% voraussichtlich schlechter als der erste Entscheidungsbaum ab.
 - c. Nun kommt der Zaubertrick. Generieren Sie für jeden Testdatenpunkt die Vorhersagen der 1000 Entscheidungsbäume und heben Sie ausschließlich die häufigste Vorhersage auf (Sie können dazu die Funktion `mode()` aus SciPy verwenden). Damit erhalten Sie *Mehrheitsvorhersagen* für Ihren Testdatensatz.
 - d. Werten Sie diese Vorhersagen mit dem Testdatensatz aus: Sie sollten eine etwas höhere Genauigkeit als beim ersten Modell erhalten (etwa 0.5 bis 1.5% höher). Herzlichen Glückwunsch, Sie haben soeben einen Random-Forest-Klassifikator trainiert!

Lösungen zu diesen Übungen finden Sie in Anhang A.

Ensemble Learning und Random Forests

Nehmen Sie einmal an, Sie würden Tausenden zufällig ausgewählten Leuten eine komplexe Frage stellen und dann deren Antworten zusammenfassen. In vielen Fällen ist diese gesammelte Antwort der eines Experten überlegen. Dies bezeichnet man als *Schwarmintelligenz*. Wenn Sie analog dazu die Vorhersagen einer Gruppe Prädiktoren (z.B. Klassifikatoren oder Regressoren) zusammenfassen, erhalten Sie oft bessere Vorhersagen als mit dem besten einzelnen Vorhersagmodell. Eine solche Gruppe Prädiktoren nennt man auch ein *Ensemble*; daher bezeichnet man diese Technik als *Ensemble Learning* und einen Algorithmus zum Ensemble Learning als *Ensemble-Methode*.

Sie können beispielsweise eine Gruppe Entscheidungsbäume auf jeweils unterschiedlichen, zufällig ausgewählten Teilmengen des Trainingsdatensatzes trainieren. Um eine Vorhersage zu treffen, sammeln Sie die Vorhersagen aller einzelnen Bäume und sagen dann die Kategorie vorher, die dabei die meisten Stimmen erhält (wie in der letzten Übung in Kapitel 6). Solch ein Ensemble von Entscheidungsbäumen nennt man einen *Random Forest*, trotz seiner Einfachheit einer der mächtigsten bekannten Machine-Learning-Algorithmen.

Wie in Kapitel 2 besprochen, kommen Ensemble-Methoden eher gegen Ende eines Projekts zum Einsatz. Wenn Sie bereits einige gute Prädiktoren erstellt haben, lassen sich diese zu einem noch besseren Vorhersagmodell vereinen. Unter den Gewinnern bei Machine-Learning-Wettbewerben finden sich häufig mehrere Ensemble-Methoden (wie etwa bei der *Netflix Prize Competition* (<http://netflixprize.com/>)).

In diesem Kapitel werden wir die beliebtesten Ensemble-Methoden vorstellen, darunter *bagging*, *boosting*, *stacking* und einige andere. Wir werden uns auch mit Random Forests beschäftigen.

Abstimmverfahren unter Klassifikatoren

Stellen Sie sich vor, Sie hätten einige Klassifikatoren trainiert, die alle eine Genauigkeit von jeweils 80% erzielen. Darunter könnte ein Klassifikator mit logistischer

Regression sein, ein SVM-Klassifikator, ein Random-Forest-Klassifikator, ein k-nächste-Nachbarn-Klassifikator und vielleicht noch weitere (siehe Abbildung 7-1).

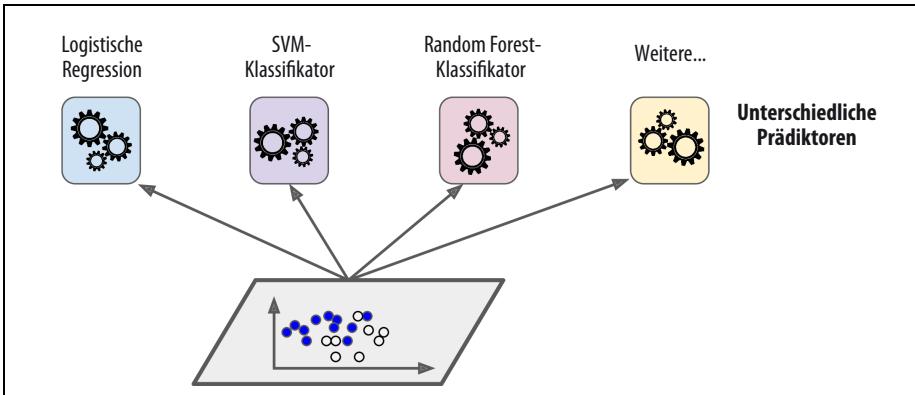


Abbildung 7-1: Trainieren unterschiedlicher Klassifikatoren

Die Vorhersagen lassen sich sehr einfach zu einem noch besseren Klassifikator zusammenfassen, indem Sie die Vorhersagen aller Klassifikatoren zusammenfassen und die Kategorie mit den meisten Stimmen vorhersagen. Einen solchen mehrheitsbasierten Klassifikator bezeichnet man als *Hard-Voting-Klassifikator* (siehe Abbildung 7-2).

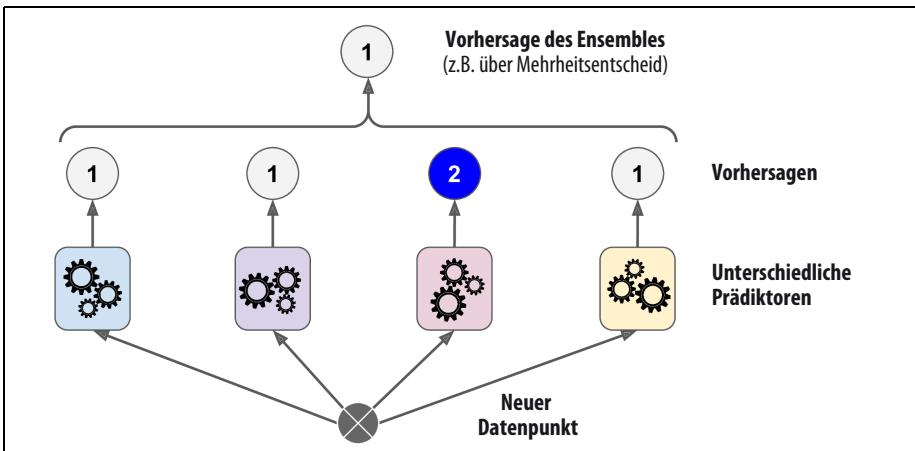


Abbildung 7-2: Vorhersagen eines Hard-Voting-Klassifikators

Es mag etwas überraschend erscheinen, dass dieser abstimmungsbasierte Klassifikator häufig eine höhere Genauigkeit erzielt als der beste Klassifikator im Ensemble. Tatsächlich kann das Ensemble sogar dann ein *starker Lerner* sein (also eine hohe Genauigkeit erreichen), wenn jeder Klassifikator ein *schwacher Lerner* ist (also nur geringfügig besser als zufälliges Raten ist). Dies setzt aber voraus, dass es genügend viele schwache Lerner gibt und dass diese sich genügend voneinander unterscheiden.

Wie ist dies möglich? Vielleicht kann die folgende Analogie dieses Rätsel ein wenig erhellern. Nehmen wir an, Sie hätten eine etwas unausgewogene Münze, die beim Werfen in 51% der Fälle Kopf und in 49% der Fälle Zahl zeigt. Wenn Sie diese Münze 1000 Mal werfen, erhalten Sie etwa 510 Mal Kopf und 490 Mal Zahl und damit eine Mehrheit für Kopf. Wenn Sie genau nachrechnen, finden Sie heraus, dass der Kopf-Wurf mit einer Wahrscheinlichkeit von ca. 75% häufiger ist. Je öfter Sie die Münze werfen, umso höher wird diese Wahrscheinlichkeit (z.B. mit 10000 Würfen steigt sie auf über 97%). Dies liegt am *Gesetz der großen Zahl*: Während Sie die Münze immer öfter werfen, nähert sich der Anteil von Kopf-Würfen immer näher an die tatsächliche Wahrscheinlichkeit an (51%). Abbildung 7-3 zeigt zehn Serien solcher unausgewogener Münzwürfe. Sie können dort sehen, wie der Anteil von Kopf sich mit steigender Anzahl Würfe 51% annähert. Irgendwann nähern sich alle zehn Serien so nah an 51%, dass sie kontinuierlich über 50% liegen.

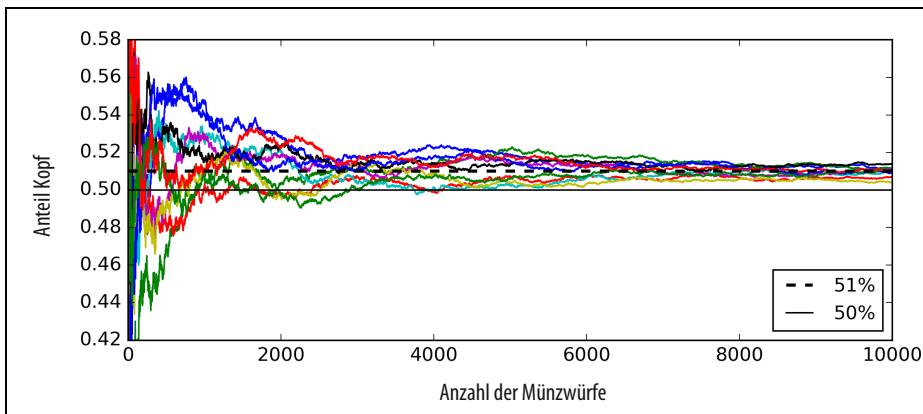


Abbildung 7-3: Das Gesetz der großen Zahl

In ähnlicher Weise können Sie ein Ensemble aus 1000 Klassifikatoren konstruieren, die für sich allein in jeweils nur 51% der Fälle richtig liegen (also kaum besser als der Zufall). Wenn Sie die von der Mehrheit gewählte Kategorie vorhersagen, können Sie auf eine Genauigkeit von 75% hoffen! Dies ist aber nur der Fall, wenn die Klassifikatoren voneinander vollständig unabhängig sind und nicht miteinander korrelierende Fehler begehen. Natürlich ist dies nicht der Fall, wenn sie auf den gleichen Daten trainiert sind. Dann neigen sie dazu, die gleiche Art Fehler zu begehen, und deshalb gibt es viele Mehrheitsentscheidungen für die falsche Kategorie, wodurch die Genauigkeit des Ensembles sinkt.



Ensemble-Methoden funktionieren am besten, wenn die Prädiktoren so unterschiedlich wie möglich sind. Sie können verschiedene Klassifikatoren erhalten, indem Sie sehr unterschiedliche Algorithmen verwenden. Damit erhöhen Sie die Chance, dass sie sehr differenzierte Arten von Fehlern begehen, was die Genauigkeit des Ensembles erhöht.

Der folgende Code erstellt und trainiert einen Klassifikator in Scikit-Learn, in dem drei unterschiedliche Klassifikatoren abstimmen (der Trainingsdatensatz ist der in Kapitel 5 vorgestellte Datensatz moons):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Betrachten wir die Genauigkeit jedes einzelnen Klassifikators auf den Testdaten:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

Da haben Sie es! Der abstimmungsbasierte Klassifikator ist eine Spur besser als alle übrigen Klassifikatoren. Wenn sämtliche Klassifikatoren in der Lage sind, Wahrscheinlichkeiten für die Kategorien zu berechnen (d.h. die Methode `predict_proba()` besitzen), dann können Sie Scikit-Learn anweisen, die Kategorie mit der höchsten über alle einzelnen Klassifikatoren gemittelten Wahrscheinlichkeit vorherzusagen. Dies wird auch als *Soft Voting* bezeichnet. Damit erzielen Sie häufig eine bessere Vorhersageleistung als mit Hard Voting, weil den zuverlässigeren Stimmen mehr Gewicht beigegeben wird. Sie müssen dazu lediglich `voting="hard"` mit `voting="soft"` ersetzen und sicherstellen, dass sämtliche Klassifikatoren Wahrscheinlichkeiten abschätzen können. Dies ist bei der Klasse `SVC` nicht automatisch der Fall. Sie müssen daher den Hyperparameter `probability` auf `True` setzen (dadurch führt die Klasse `SVC` eine Kreuzvalidierung zum Abschätzen der Wahrscheinlichkeiten durch). Das Training dauert dabei länger, und Sie erhalten die Methode `predict_proba()`). Wenn Sie den obigen Code auf Soft Voting umstellen, sollten Sie herausfinden, dass das Abstimmverfahren eine Genauigkeit von mehr als 91% erzielt!

Bagging und Pasting

Um einen möglichst diversen Satz Klassifikatoren zu erhalten, können wir wie oben besprochen unterschiedliche Algorithmen einsetzen. Wir können aber auch bei jedem Prädiktor den gleichen Trainingsalgorithmus verwenden, aber mit einer jeweils anderen, zufällig ausgewählten Teilmenge der Trainingsdaten trainieren. Werden diese Teilmengen *mit Zurücklegen* gebildet, bezeichnet man die Methode als *Bagging* (<http://goo.gl/o42tml>)¹ (eine Kurzform von *Bootstrap-Aggregation*)². Werden die Teilmengen *ohne Zurücklegen* gebildet, bezeichnet man dies als *Pasting* (<http://goo.gl/BXm0pm>).³

Anders ausgedrückt ist es sowohl beim Bagging als auch beim Pasting möglich, dass ein Datenpunkt in mehreren Prädiktoren zum Training verwendet wird, aber nur beim Bagging kann ein und derselbe Datenpunkt mehrfach für den gleichen Prädiktor ausgewählt werden. In Abbildung 7-4 ist dieser Vorgang dargestellt.

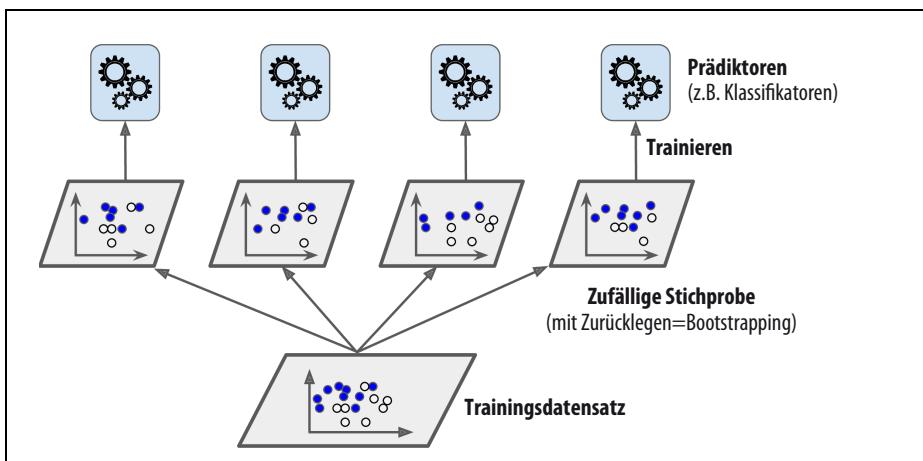


Abbildung 7-4: Auswahl von Trainingsdatensätzen mit Pasting/Bagging und Trainieren

Sobald sämtliche Prädiktoren trainiert wurden, kann das Ensemble eine Vorhersage für einen neuen Datenpunkt treffen, indem es einfach die Vorhersagen sämtlicher Prädiktoren zusammenfasst. Die dazu verwendete Aggregatfunktion ist bei einer Klassifikation typischerweise der *Modalwert* (also wie bei einem Hard-Voting-Klassifikator die häufigste Vorhersage) und bei einer Regression der Mittelwert. Jeder einzelne Prädiktor hat ein höheres Bias, als wenn er auf dem ursprünglichen Trainingsdatensatz trainiert würde, aber durch die Aggregation werden sowohl Bias als auch Varianz gesenkt.⁴ Das Gesamtergebnis ist, dass das Ensemble ein ähn-

1 »Bagging Predictors«, L. Breiman (1996).

2 In der Statistik nennt man das Bilden von Stichproben mit Zurücklegen auch *Bootstrapping*.

3 »Pasting small votes for classification in large databases and on-line«, L. Breiman (1999).

4 Bias und Varianz wurden in Kapitel 4 vorgestellt.

liches Bias, aber eine niedrigere Varianz aufweist als ein einzelner auf dem gesamten ursprünglichen Trainingsdatensatz trainierter Prädiktor.

Wie Sie Abbildung 7-4 entnehmen können, lassen sich alle Prädiktoren parallel auf unterschiedlichen CPU-Cores oder sogar unterschiedlichen Servern trainieren. In ähnlicher Weise lassen sich auch die Vorhersagen parallelisieren. Dies ist einer der Gründe, aus denen Bagging und Pasting ausgesprochen beliebt sind: Sie skalieren sehr gut.

Bagging und Pasting in Scikit-Learn

Scikit-Learn enthält mit der Klasse `BaggingClassifier` (oder zur Regression die Klasse `BaggingRegressor`) eine einfache Schnittstelle für sowohl Bagging als auch Pasting. Das folgende Codebeispiel trainiert ein Ensemble von 500 Entscheidungsbäumen,⁵ die mit jeweils 100 zufällig aus den Trainingsdaten mit Zurücklegen ausgewählten Datenpunkten trainiert werden (dies ist ein Beispiel für Bagging; falls Sie stattdessen Pasting verwenden möchten, setzen Sie `bootstrap=False`). Der Parameter `n_jobs` stellt die Anzahl von Scikit-Learn bei Training und Vorhersage zu verwendender CPU-Cores ein (-1 instruiert Scikit-Learn, sämtliche verfügbaren Cores zu nutzen):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```



Der `BaggingClassifier` führt automatisch ein Soft Voting anstelle von Hard Voting durch, falls die zugrunde liegenden Klassifikatoren Wahrscheinlichkeiten für Kategorien abschätzen können (also die Methode `predict_proba()` besitzen). Bei unseren Entscheidungsbäumen ist dies der Fall.

Abbildung 7-5 vergleicht die Entscheidungsgrenze eines einzelnen Entscheidungsbäums mit der Entscheidungsgrenze des Ensembles mit 500 Bäumen und Bagging (aus dem obigen Codebeispiel). Beide Modelle wurden auf dem Datensatz moons trainiert. Die Vorhersagen des Ensembles verallgemeinern deutlich besser als die Vorhersagen des einzelnen Entscheidungsbäums: Das Ensemble weist ein vergleichbares Bias auf, hat aber eine geringere Varianz (ihm unterlaufen auf den Trainingsdaten etwa genauso viele Fehler, aber die Entscheidungsgrenze ist regelmäßiger).

⁵ Für `max_samples` lässt sich auch eine Fließkommazahl zwischen 0.0 und 1.0 einsetzen. In diesem Fall ist die auszuwählende Anzahl Datenpunkte gleich der mit `max_samples` multiplizierten Größe des Trainingsdatensatzes.

Mit Bootstrapping werden die Teildatensätze zum Trainieren der einzelnen Prädiktoren ein wenig diverser, daher ist das Bias beim Bagging etwas höher als beim Pasting. Das heißt aber auch, dass die Prädiktoren weniger miteinander korrelieren, und die Varianz des Ensembles reduziert sich dadurch. Insgesamt führt Bagging häufig zu besseren Modellen und wird deshalb meist bevorzugt. Wenn Sie aber Zeit und Rechenkapazität übrig haben, können Sie eine Kreuzvalidierung durchführen, um Bagging und Pasting miteinander zu vergleichen, und die bessere der beiden Methoden auswählen.

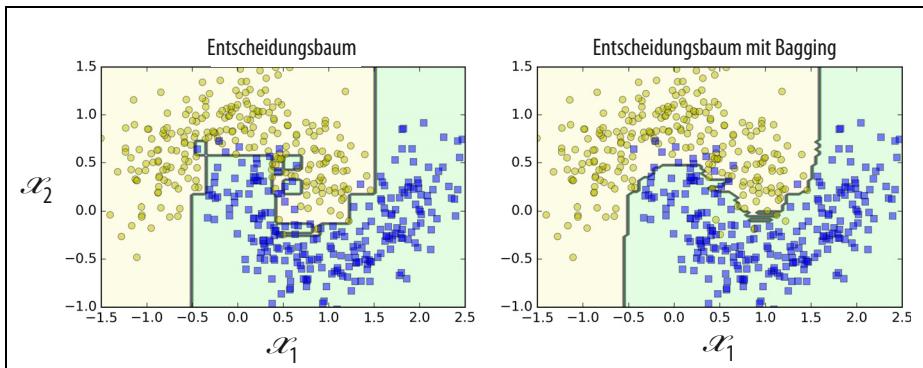


Abbildung 7-5: Ein einzelner Entscheidungsbaum im Vergleich zu einem Ensemble von 500 Bäumen mit Bagging

Out-of-Bag-Evaluation

Beim Bagging können einige Datenpunkte mehrmals einem bestimmten Prädiktor zugewiesen werden, während andere überhaupt nicht ausgewählt werden. Der `BaggingClassifier` wählt mit den Standardeinstellungen m Trainingsdatenpunkte mit Zurücklegen aus (`bootstrap=True`), wobei m der Größe des Trainingsdatensatzes entspricht. Damit werden im Durchschnitt nur etwa 63 % der Trainingsdatenpunkte für jeden Prädiktor ausgewählt.⁶ Die übrigen 37 % nicht ausgewählten Trainingsdatenpunkte bezeichnet man als *Out-of-Bag*-OOB-Datenpunkte. Beachten Sie, dass es nicht bei allen Prädiktoren die gleichen 37 % sind.

Da ein Prädiktor beim Trainieren niemals die OOB-Datenpunkte sieht, lässt er sich mit diesen Datenpunkten auswerten, ohne dass ein separater Validierungsdatensatz oder Kreuzvalidierung nötig wären. Sie können das Ensemble selbst evaluieren, indem Sie den Mittelwert der OOB-Evaluationen der einzelnen Prädiktoren bilden.

Mit Scikit-Learn können Sie nach dem Trainieren automatisch eine OOB-Evaluation durchführen, indem Sie beim Erstellen eines `BaggingClassifier` den Wert `oob_score=True` setzen. Dies ist im folgenden Codebeispiel gezeigt. Der Score der dabei durchgeführten Auswertung ist als Attribut `oob_score_` verfügbar:

⁶ Bei steigendem m nähert sich dieser Anteil $1 - \exp(-1) \approx 63.212\%$.

```

>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9013333333333332

```

Laut der OOB-Evaluation erzielt der BaggingClassifier auf dem Testdatensatz eine Genauigkeit von voraussichtlich etwa 90.1 %. Das prüfen wir:

```

>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9120000000000003

```

Wir erhalten eine Genauigkeit von 91.2 % auf den Testdaten – das ist nah genug!

Die OOB-Entscheidungsfunktion jedes Trainingsdatenpunkts ist auch als Attribut `oob_decision_function_` verfügbar. In diesem Fall liefert diese Entscheidungsfunktion die Wahrscheinlichkeiten der Kategorien für jeden Trainingsdatenpunkt (da der zugrunde liegende Estimator die Methode `predict_proba()` enthält). Beispielsweise schätzt die OOB-Evaluation, dass der zweite Trainingsdatenpunkt mit 68.25 %iger Wahrscheinlichkeit der positiven Kategorie angehört (und mit 31.75 % der negativen Kategorie):

```

>>> bag_clf.oob_decision_function_
array([[ 0.31746032,  0.68253968],
       [ 0.34117647,  0.65882353],
       [ 1.          ,  0.          ],
       ...
       [ 1.          ,  0.          ],
       [ 0.03108808,  0.96891192],
       [ 0.57291667,  0.42708333]])

```

Zufällige Patches und Subräume

Die Klasse `BaggingClassifier` ermöglicht auch das Auswählen von Merkmalen. Dies wird über zwei weitere Hyperparameter gesteuert: `max_features` und `bootstrap_features`. Diese funktionieren genauso wie `max_samples` und `bootstrap`, wählen aber Merkmale anstelle von Datenpunkten aus. So wird jeder Prädiktor mit einer zufälligen Teilmenge der Eingabemerkmale trainiert.

Dies ist besonders bei hochdimensionalen Eingabedaten nützlich (z.B. Bildern). Das zufällige Auswählen sowohl von Trainingsdatenpunkten als auch von Merkmalen bezeichnet man als die Methode der *zufälligen Patches* (<http://goo.gl/B2EcM2>).⁷ Werden sämtliche Trainingsdatenpunkte verwendet (z.B. mit `bootstrap=False` und `max_samples=1.0`), aber Merkmale ausgewählt (z.B. `bootstrap_features=True` und/

⁷ »Ensembles on Random Patches«, G. Louppe and P. Geurts (2012).

oder `max_features` kleiner als 1.0), so nennt man dies die *Methode der zufälligen Subräume* (<http://goo.gl/NPi5vH>).⁸

Das Auswählen von Merkmalen führt zu noch mehr Diversität unter den Prädiktoren, und es wird etwas mehr Bias gegen eine geringere Varianz eingetauscht.

Random Forests

Wie bereits besprochen, ist ein *Random Forest* (<http://goo.gl/zVOGQ1>)⁹ ein Ensemble von Entscheidungsbäumen, die nach der Bagging-Methode trainiert werden (seltener auch mit Pasting). Typischerweise entspricht dabei `max_samples` der Größe des Trainingsdatensatzes. Anstatt einen `BaggingClassifier` zu erstellen und diesem einen `DecisionTreeClassifier` zu übergeben, können Sie die bequemere Klasse `RandomForestClassifier` verwenden, die für Entscheidungsbäume optimiert worden ist¹⁰ (es existiert auch die Klasse `RandomForestRegressor` für Regressionsaufgaben). Das folgende Codebeispiel trainiert einen Random-Forest-Klassifikator mit 500 Bäumen (jeder mit maximal 16 Knoten) und verwendet dazu alle verfügbaren CPU-Cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Mit wenigen Ausnahmen enthält ein `RandomForestClassifier` sämtliche Hyperparameter eines `DecisionTreeClassifier` (um das Erzeugen der Bäume zu steuern) und sämtliche Hyperparameter eines `BaggingClassifier`, um das Ensemble selbst zu kontrollieren.¹¹

Der Random-Forest-Algorithmus verwendet beim Erzeugen der Bäume ein zusätzliches Zufallselement; anstatt beim Aufteilen eines Knotens das bestmögliche Merkmal zu ermitteln (siehe Kapitel 6), wird das beste Merkmal in einer zufälligen Untermenge von Merkmalen identifiziert. Dies führt zu höherer Diversität unter den Bäumen, wobei (schon wieder) ein höheres Bias für eine geringere Varianz in Kauf genommen wird. Dadurch wird das Modell in der Regel insgesamt besser. Der folgende `BaggingClassifier` ist mit dem obigen `RandomForestClassifier` nahezu identisch:

⁸ »The random subspace method for constructing decision forests«, Tin Kam Ho (1998).

⁹ »Random Decision Forests«, T. Ho (1995).

¹⁰ Die Klasse `BaggingClassifier` ist weiter nützlich, wenn Sie ein Ensemble von anderen Modellen als Entscheidungsbäumen erstellen möchten.

¹¹ Es gibt einige bemerkenswerte Ausnahmen: `splitter` fehlt (es wird automatisch "random" verwendet), `presort` fehlt (automatisch `False`), `max_samples` fehlt (automatisch 1.0) und `base_estimator` fehlt ebenfalls (es wird ein `DecisionTreeClassifier` mit den gegebenen Hyperparametern verwendet).

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),  
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Extra-Trees

Wenn Sie in einem Random Forest einen Baum erzeugen, wird bei der Teilung an jedem Knoten nur eine zufällige Auswahl der Merkmale berücksichtigt (wie oben erwähnt). Diese Bäume lassen sich noch zufälliger gestalten, indem die Schwellenwerte für jedes Merkmal ebenfalls zufällig bestimmt werden, anstatt (wie in gewöhnlichen Entscheidungsbäumen) nach dem bestmöglichen Schwellenwert zu suchen.

Einen Wald solch extrem zufallsabhängiger Bäume bezeichnet man einfach als *Extremely Randomized Trees*¹² (<http://goo.gl/RHGEA4>) (oder kurz *Extra-Trees*). Wieder einmal wird hier Bias gegen eine geringere Varianz getauscht. Extra-Trees lassen sich außerdem viel schneller trainieren als gewöhnliche Random Forests, da das Finden des bestmöglichen Schwellenwerts für jedes Merkmal und jeden inneren Knoten einer der zeitaufwendigsten Schritte beim Erzeugen eines Baums ist.

Sie können einen Extra-Trees-Klassifikator in Scikit-Learn mit der Klasse `ExtraTreesClassifier` erstellen. Deren Schnittstelle ist mit der Klasse `RandomForestClassifier` identisch. Analog dazu hat die Klasse `ExtraTreesRegressor` die gleiche Schnittstelle wie die Klasse `RandomForestRegressor`.



Es ist schwer im Voraus zu sagen, ob ein `RandomForestClassifier` bessere oder schlechtere Vorhersagen trifft als ein `ExtraTreesClassifier`. Im Allgemeinen lässt sich das nur durch Ausprobieren beider Varianten und einen direkten Vergleich durch Kreuzvalidierung herausfinden (sowie durch Optimieren der Hyperparameter mittels Gittersuche).

Wichtigkeit von Merkmalen

Eine weitere großartige Eigenschaft von Random Forests ist, dass sich mit ihnen die relative Wichtigkeit jedes Merkmals bestimmen lässt. Scikit-Learn bestimmt die Wichtigkeit von Merkmalen darüber, wie stark ein bestimmtes Merkmal die Unreinheit von Knoten, die dieses Merkmal verwenden, im Durchschnitt reduziert (über sämtliche Bäume im Forest). Genauer gesagt, ist dies ein gewichteter Mittelwert, wobei das Gewicht jedes Knotens gleich der Anzahl dort verwendeter Trainingsdatenpunkte ist (siehe Kapitel 6).

Scikit-Learn berechnet diesen Score nach dem Trainieren automatisch für jedes Merkmal und skaliert das Ergebnis anschließend so, dass die Summe aller Wichtigkeiten 1 ergibt. Sie können das Ergebnis über das Attribut `feature_importances_` inspizieren. Beispielsweise trainiert der folgende Code einen `RandomForestClassi-`

12 »Extremely randomized trees«, P. Geurts, D. Ernst, L. Wehenkel (2005).

fier auf dem Iris-Datensatz (aus Kapitel 4) und gibt die Wichtigkeit jedes Merkmals aus. Es sieht so aus, als wären die wichtigsten Merkmale die Länge (44%) und Breite (42%) der Kronblätter, während Länge und Breite der Kelchblätter im Vergleich dazu eher uninteressant sind (jeweils 11% und 2%).

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Wenn Sie einen Random-Forest-Klassifikator auf dem MNIST-Datensatz (aus Kapitel 3) trainieren und die Wichtigkeit jedes einzelnen Pixels plotten, erhalten Sie das Muster in Abbildung 7-6.

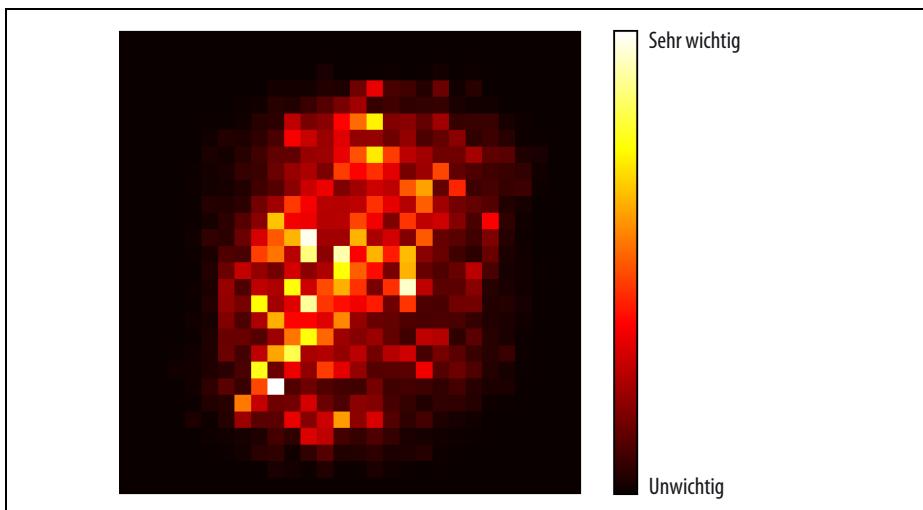


Abbildung 7-6: Wichtigkeit der Pixel in MNIST (laut einem Random-Forest-Klassifikator)

Random Forests sind sehr nützlich, um einen schnellen Überblick darüber zu erhalten, welche Merkmale wirklich wichtig sind, besonders wenn Sie gezwungen sind, Merkmale auszuwählen.

Boosting

Boosting (ursprünglich *Hypothesis Boosting*) bezeichnet eine beliebige Ensemble-Methode, bei der sich mehrere schwache Lerner zu einem starken Lerner kombinieren lassen. Die Grundidee der meisten Methoden zum Boosting ist, die Prädik-

toren nacheinander zu trainieren, sodass jeder bestrebt ist, die Fehler seines Vorgängers zu beheben. Es gibt zahlreiche Boosting-Verfahren; die bei Weitem beliebtesten sind jedoch AdaBoost (<http://goo.gl/OIduRW>)¹³ – kurz für Adaptive Boosting – und Gradient Boosting. Beginnen wir mit AdaBoost.

AdaBoost

Ein Prädiktor kann seinen Vorgänger korrigieren, indem er den vom Vorgänger nicht abgedeckten Trainingsdatenpunkten etwas mehr Aufmerksamkeit widmet. Damit ergeben sich neue Prädiktoren, die sich mehr und mehr auf die schwierigen Fälle konzentrieren. Dies ist die von AdaBoost verwendete Technik.

Um beispielsweise einen AdaBoost-Klassifikator zu entwickeln, wird ein erster Klassifikator erstellt (z.B. ein Entscheidungsbaum), trainiert und für Vorhersagen auf den Trainingsdaten eingesetzt. Das relative Gewicht der falsch vorhergesagten Trainingsdatenpunkte wird anschließend erhöht. Nun wird ein zweiter Klassifikator mit den aktualisierten Gewichten trainiert, es werden abermals Vorhersagen auf den Trainingsdaten getroffen, die Gewichte aktualisiert und so weiter (siehe Abbildung 7-7).

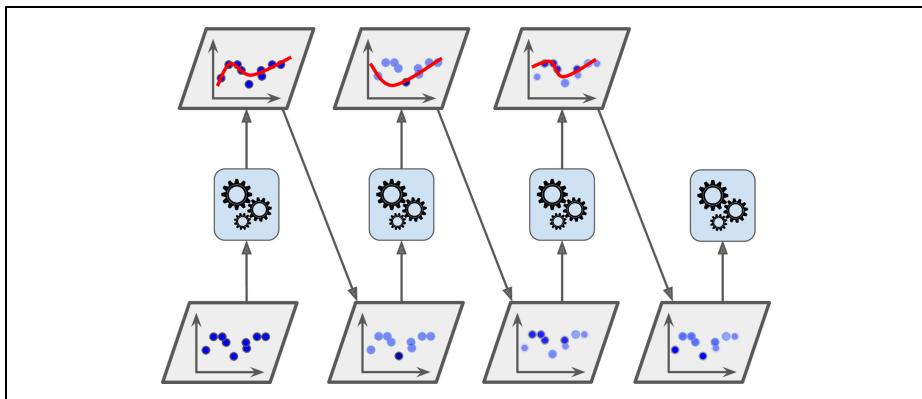


Abbildung 7-7: Sequenzielles Training mit aktualisierten Gewichten der Datenpunkte bei AdaBoost

Abbildung 7-8 zeigt die Entscheidungsgrenzen von fünf aufeinanderfolgenden Prädiktoren auf dem Datensatz moons (in diesem Beispiel ist jeder Prädiktor ein in hohem Maße regularisierter SVM-Klassifikator mit einem RBF-Kernel¹⁴). Der erste Klassifikator liegt bei vielen Datenpunkten falsch, also werden deren Gewichte erhöht. Deshalb schneidet der zweite Klassifikator bei diesen Datenpunkten besser

¹³ »A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting«, Yoav Freund, Robert E. Schapire (1997).

¹⁴ Dies geschieht nur zur Veranschaulichung. SVMs sind grundsätzlich keine guten Prädiktoren für AdaBoost, weil sie langsam sind und bei AdaBoost zur Instabilität neigen.

ab und so weiter. Das Diagramm auf der rechten Seite stellt die gleiche Folge von Prädiktoren dar, nur dass die Lernrate halbiert ist (also die Gewichte der falsch klassifizierten Datenpunkte bei jedem Durchlauf nur um die Hälfte erhöht werden). Wie Sie sehen, hat diese sequenzielle Lerntechnik eine gewisse Ähnlichkeit mit dem Gradientenverfahren. Anstatt aber die Parameter eines einzelnen Prädiktors zum Minimieren einer Kostenfunktion zu verändern, fügt AdaBoost Prädiktoren zum Ensemble hinzu, um es nach und nach zu verbessern.

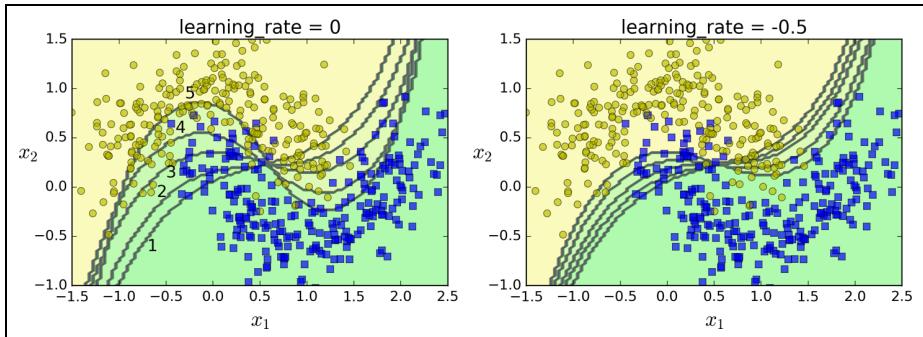


Abbildung 7-8: Entscheidungsgrenzen aufeinanderfolgender Prädiktoren

Sobald sämtliche Prädiktoren trainiert wurden, trifft das Ensemble ähnlich wie beim Bagging oder Pasting Vorhersagen, die Prädiktoren haben aber je nach ihrer Genauigkeit auf den gewichteten Trainingsdaten unterschiedliches Gewicht.



Diese sequenzielle Lernmethode hat einen wichtigen Nachteil: Sie lässt sich nicht parallelisieren (höchstens teilweise). Jeder Prädiktor lässt sich erst trainieren, sobald der vorige Prädiktor trainiert und ausgewertet wurde. Daher skaliert dieses Verfahren nicht so gut wie Bagging oder Pasting.

Betrachten wir den Algorithmus hinter AdaBoost etwas genauer. Das Gewicht für jeden Datenpunkt $w^{(i)}$ wird zu Beginn auf $\frac{1}{m}$ gesetzt. Ein erster Prädiktor wird trainiert und seine gewichtete Fehlerquote r_1 auf den Trainingsdaten berechnet; siehe Formel 7-1.

Formel 7-1: Gewichtete Fehlerquote für den j^{ten} Prädiktor

$$r_j = \frac{\sum_{i=1}^m w^{(i)} \hat{y}_j^{(i)} \neq y^{(i)}}{\sum_{i=1}^m w^{(i)}}$$

wobei $\hat{y}_j^{(i)}$ die Vorhersage des j^{ten} Prädiktors für den i^{ten} Datenpunkt ist.

Das Gewicht des Prädiktors α_j wird anschließend mit Formel 7-2 berechnet, wobei der Hyperparameter η die Lernrate ist (voreingestellt ist 1).¹⁵

Je genauer der Prädiktor ist, desto höher ist auch sein Gewicht. Wenn er nur zufällig rät, ist sein Gewicht nahezu null. Liegt er jedoch häufiger falsch als richtig (also ungenauer als zufälliges Raten), wird ihm ein negatives Gewicht zugewiesen.

Formel 7-2: Gewicht eines Prädiktors

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Anschließend werden die Gewichte der Datenpunkte mit Formel 7-3 aktualisiert: Die falsch klassifizierten Datenpunkte werden *geboostet*.

Formel 7-3: Regel zum Aktualisieren der Gewichte

für $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{wenn } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{wenn } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Anschließend werden die Gewichte aller Datenpunkte normiert (durch $\sum_{i=1}^m w^{(i)}$ geteilt).

Schließlich wird mit den aktualisierten Gewichten ein neuer Prädiktor trainiert, und der ganze Vorgang wird wiederholt (das Gewicht des neuen Prädiktors wird berechnet, die Gewichte der Datenpunkte werden aktualisiert, ein neuer Prädiktor wird trainiert und so weiter). Der Algorithmus hält an, sobald eine gewünschte Anzahl Prädiktoren erreicht oder ein perfekter Prädiktor gefunden wird.

Beim Treffen von Vorhersagen berechnet AdaBoost einfach die Vorhersagen sämtlicher Prädiktoren und gewichtet sie nach den Gewichten der Prädiktoren α_j . Die vorhergesagte Kategorie ist diejenige, die eine Mehrheit der gewichteten Stimmen erhält (siehe Formel 7-4).

Formel 7-4: Vorhersagen mit AdaBoost

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j \quad \text{wobei } N \text{ die Anzahl Prädiktoren ist.}$$

$$\hat{y}_j(\mathbf{x}) = k$$

Scikit-Learn verwendet eine für mehrere Kategorien geeignete Version von AdaBoost namens *SAMME* (<http://goo.gl/Eji2vR>)¹⁶, was für *Stagewise Additive Modeling*

15 Der ursprüngliche AdaBoost-Algorithmus verwendet keine Lernrate als Hyperparameter.

16 Details können Sie in »Multi-Class AdaBoost«, J. Zhu et al. (2006) nachlesen.

using a Multiclass Exponential loss function steht. Bei nur zwei Kategorien ist SAMME zu AdaBoost äquivalent. Wenn die Prädiktoren außerdem in der Lage sind, Wahrscheinlichkeiten für die Kategorien anzugeben (also die Methode `predict_proba()` besitzen), kann Scikit-Learn eine Variante von SAMME namens `SAMME.R` einsetzen (das R steht für »Real«), die sich auf die Wahrscheinlichkeiten anstelle der Vorhersagen stützt und im Allgemeinen besser funktioniert.

Der folgende Code trainiert einen AdaBoost-Klassifikator mit 200 *Decision Stumps*. Er verwendet dazu die Klasse `AdaBoostClassifier` aus Scikit-Learn (wie Sie sich vielleicht denken, gibt es auch eine Klasse `AdaBoostRegressor`). Ein Decision Stump ist ein Entscheidungsbaum mit `max_depth=1` – es ist also ein Baum, der aus einem einzigen inneren Knoten und zwei Blättern besteht. Dies ist der standardmäßig in der Klasse `AdaBoostClassifier` eingestellte Estimator:

```
from sklearn.ensemble import AdaBoostClassifier  
  
ada_clf = AdaBoostClassifier(  
    DecisionTreeClassifier(max_depth=1), n_estimators=200,  
    algorithm="SAMME.R", learning_rate=0.5)  
ada_clf.fit(X_train, y_train)
```



Wenn Ihr AdaBoost-Ensemble zum Overfitting der Trainingsdaten neigt, können Sie die Anzahl der Estimatoren reduzieren oder den zugrunde liegenden Estimator stärker regularisieren.

Gradient Boosting

Ein zweiter sehr beliebter Boosting-Algorithmus ist *Gradient Boosting* (<http://goo.gl/Ezw4jL>).¹⁷ Wie AdaBoost fügt auch Gradient Boosting die Prädiktoren nacheinander einem Ensemble hinzu, wobei jeder seinen Vorgänger korrigiert. Anstatt jedoch bei jedem Durchlauf wie bei AdaBoost die Gewichte der Datenpunkte zu verändern, werden bei dieser Methode die neuen Prädiktoren an die vom vorigen Prädiktor begangenen *Restfehler* angepasst.

Betrachten wir ein einfaches Regressionsbeispiel mit Entscheidungsbäumen als zugrunde liegendem Prädiktor (natürlich funktioniert Gradient Boosting auch bei Regressionsaufgaben ausgezeichnet). Dies bezeichnet man als *Gradient Tree Boosting* oder *Gradient Boosted Regression Trees (GBRT)*. Zunächst passen wir einen `DecisionTreeRegressor` an die Trainingsdaten an (beispielsweise einen verrauschten quadratischen Trainingsdatensatz):

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg1 = DecisionTreeRegressor(max_depth=2)  
tree_reg1.fit(X, y)
```

¹⁷ Erstmals in »Arcing the Edge«, L. Breiman (1997) vorgestellt.

Anschließend trainieren wir einen zweiten DecisionTreeRegressor auf den vom ersten Prädiktor verursachten Restfehlern:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Schließlich trainieren wir noch einen dritten Regressor auf den Restfehlern des zweiten Prädiktors:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Damit haben wir ein Ensemble aus drei Bäumen. Wir können Vorhersagen für einen neuen Datenpunkt treffen, indem wir einfach die Vorhersagen aller drei Bäume addieren:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Abbildung 7-9 zeigt in der linken Spalte die Vorhersagen dieser drei Bäume und in der rechten Spalte die Vorhersagen des Ensembles. In der ersten Zeile besteht das Ensemble aus nur einem Baum, daher sind seine Vorhersagen exakt die gleichen wie die des ersten Baums. In der zweiten Zeile wird ein zweiter Baum auf den Restfehlern des ersten Baums trainiert. Auf der rechten Seite können Sie sehen, dass die Vorhersage des Ensembles der Summe der Vorhersagen der ersten beiden Bäume entspricht.

In ähnlicher Weise wird in der dritten Zeile ein weiterer Baum auf den Restfehlern des zweiten Baums trainiert. Sie sehen, dass sich die Vorhersagen des Ensembles nach und nach verbessern, während wir dem Ensemble zusätzliche Bäume hinzufügen.

Ein GBRT-Ensemble lässt sich einfacher mit der Klasse GradientBoostingRegressor in Scikit-Learn trainieren. Wie die Klasse RandomForestRegressor enthält auch diese Hyperparameter, die das Wachstum der Entscheidungsbäume steuern (z.B. `max_depth`, `min_samples_leaf` und so weiter), und Hyperparameter, die das Training des Ensembles insgesamt beeinflussen, z.B. die Anzahl der Bäume (`n_estimators`). Der folgende Code erzeugt das gleiche Ensemble wie oben:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

Der Hyperparameter `learning_rate` skaliert den Beitrag jedes einzelnen Baums. Wenn Sie ihn auf einen niedrigen Wert wie 0.1 setzen, benötigen Sie mehr Bäume, um das Ensemble an die Trainingsdaten anzupassen. Dafür verallgemeinern die Vorhersagen in der Regel besser. Dies ist eine als *Shrinkage* bezeichnete Regularisierungstechnik. Abbildung 7-10 zeigt zwei GBRT-Ensembles, die mit einer niedrigen Lernrate trainiert wurden: Das Ensemble auf der linken Seite enthält nicht

genug Bäume, um die Trainingsdaten abzubilden, das auf der rechten Seite enthält zu viele Bäume und verursacht ein Overfitting des Trainingsdatensatzes.

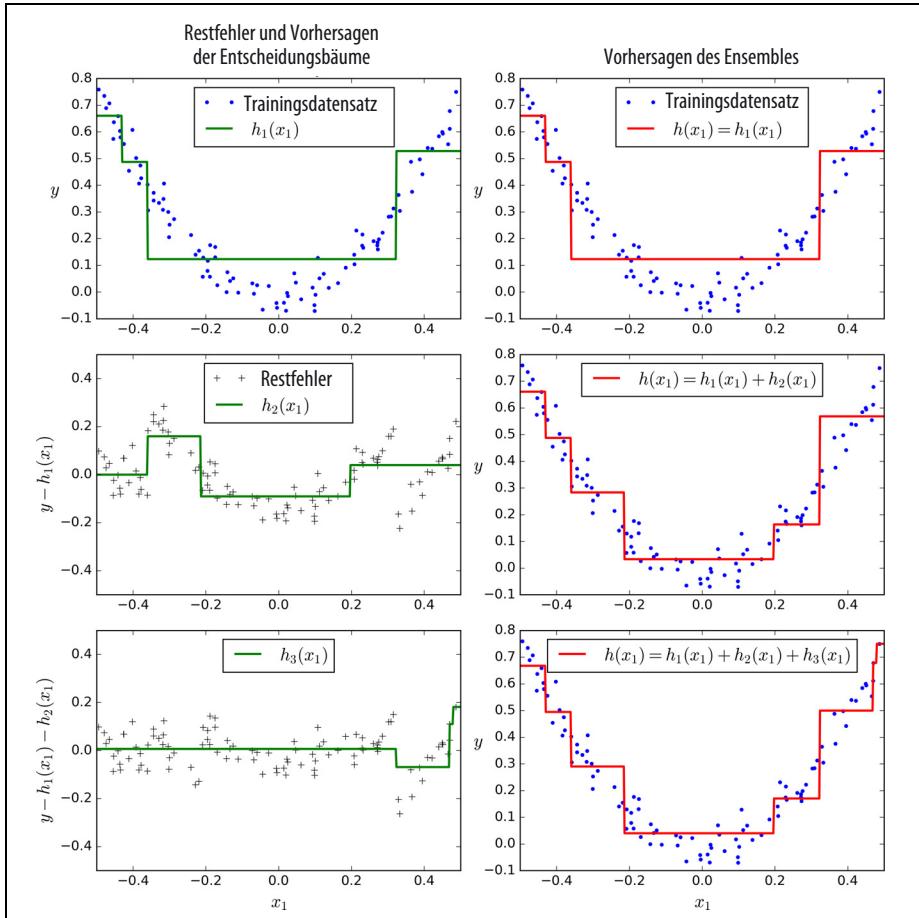


Abbildung 7-9: Gradient Boosting

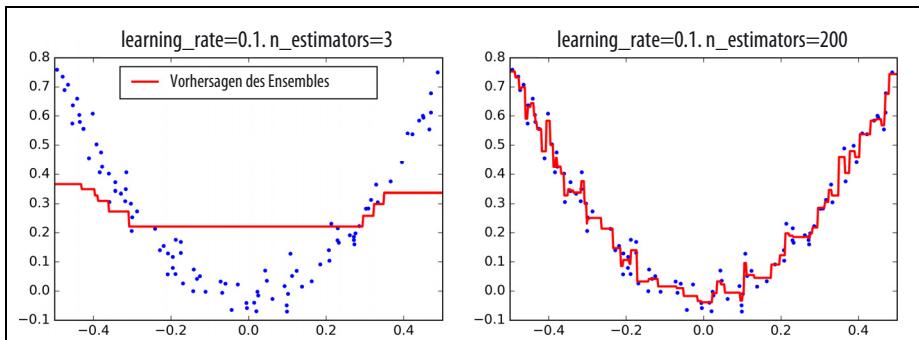


Abbildung 7-10: GBRT-Ensembles mit zu wenigen (links) und zu vielen Prädiktoren (rechts)

Um die optimale Anzahl Bäume zu ermitteln, können Sie *Early Stopping* verwenden (siehe Kapitel 4), leicht umzusetzen mit der Methode `staged_predict()`: Sie liefert einen Iterator über die vom Ensemble in jedem Trainingsabschnitt (mit einem Baum, zwei Bäumen, und so weiter) getroffenen Vorhersagen. Der folgende Code trainiert ein GBRT-Ensemble mit 120 Bäumen, legt zur Bestimmung der optimalen Anzahl Bäume den Validierungsfehler in jedem Trainingsabschnitt fest und trainiert schließlich ein weiteres GBRT-Ensemble mit der optimalen Anzahl Bäume:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

Die Validierungsfehler werden auf der linken Seite von Abbildung 7-11 und die Vorhersagen des optimalen Modells auf der rechten Seite gezeigt.

Das frühe Anhalten lässt sich auch implementieren, indem Sie den Trainingsprozess verfrüh beenden (anstatt zuerst eine größere Anzahl Bäume zu trainieren und dann im Rückblick die optimale Anzahl zu ermitteln). Über den Parameter `warm_start=True` lässt sich ein inkrementelles Training veranlassen, wobei Scikit-Learn bereits bestehende Bäume beim Aufruf von `fit()` wieder verwendet. Der folgende Code beendet das Training, sobald sich der Validierungsfehler in fünf aufeinanderfolgenden Iterationen nicht verbessert:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
    if error_going_up == 5:
        break # Early Stopping
```

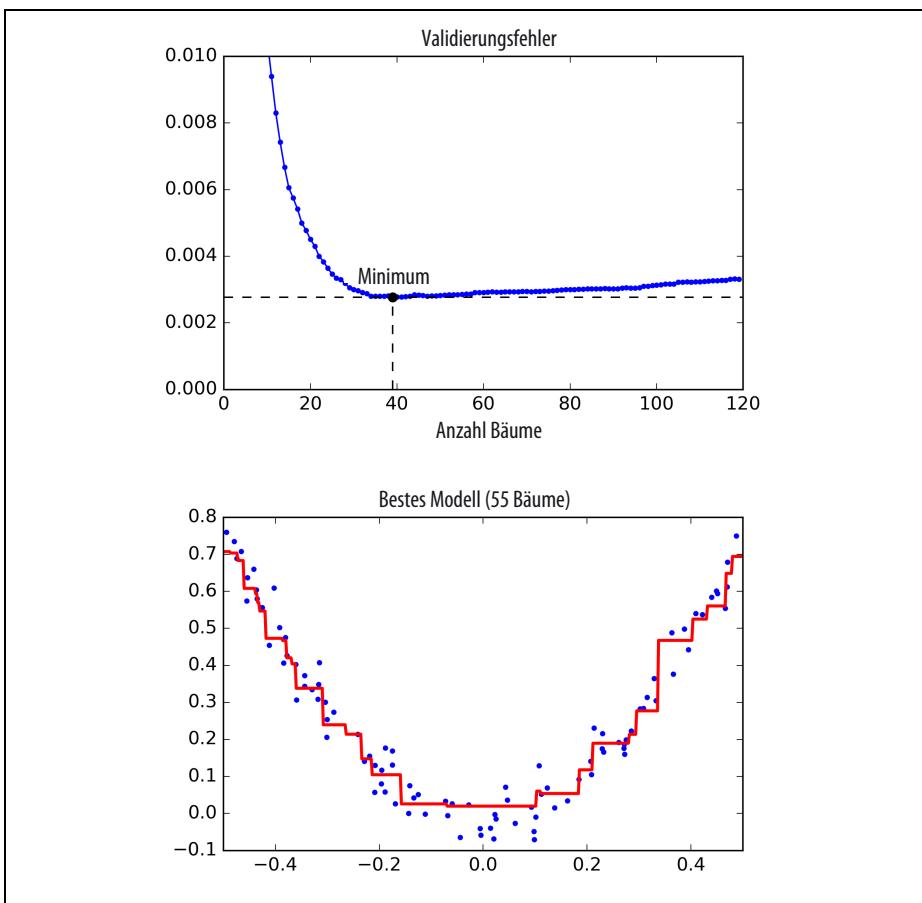


Abbildung 7-11: Optimieren der Anzahl Bäume durch Early Stopping

Die Klasse `GradientBoostingRegressor` unterstützt auch den Hyperparameter `subsample`, der den Anteil der zum Trainieren jedes Baums auszuwählenden Trainingsdatenpunkte festlegt. Beispielsweise wird mit `subsample=0.25` jeder Baum auf zufällig ausgewählten 25 % der Trainingsdatenpunkte trainiert. Wie Sie sich sicher denken können, wird hiermit ein höheres Bias gegen eine niedrigere Varianz eingetauscht. Diese Technik nennt man auch *stochastisches Gradient Boosting*.



Gradient Boosting lässt sich auch mit anderen Kostenfunktionen verwenden. Dies können Sie über den Hyperparameter `loss` einstellen (Details können Sie der Dokumentation von Scikit-Learn entnehmen).

Stacking

Die letzte in diesem Kapitel besprochene Ensemble-Methode nennt man *Stacking* (eine Kurzform von *Stacked Generalization* (<http://goo.gl/9I2NBw>)).¹⁸ Diese Methode beruht auf einer einfachen Idee: Anstatt triviale Funktionen (wie Hard Voting) zum Zusammenfassen der Vorhersagen aller Prädiktoren in einem Ensemble zu verwenden, trainieren wir ein Modell, das diese Zusammenfassung durchführt. Abbildung 7-12 zeigt ein Ensemble, das eine Regressionsaufgabe für einen neuen Datenpunkt durchführt. Jeder der drei unteren Prädiktoren sagt einen anderen Wert vorher (3.1, 2.7 und 2.9), und der letzte Prädiktor (den man als *Blender* oder *Meta-Lerner* bezeichnet) nimmt diese Vorhersagen als Eingabe und trifft daraus die endgültige Vorhersage (3.0).

Ein häufig verwendetes Ansatz zum Trainieren des Blenders ist, einen Hold-out-Datensatz zu verwenden.¹⁹ Sehen wir uns dies genauer an. Zuerst wird der Trainingsdatensatz in zwei Untermengen aufgeteilt. Die erste Teilmenge verwenden wir, um die Prädiktoren der ersten Stufe zu trainieren (siehe Abbildung 7-13).

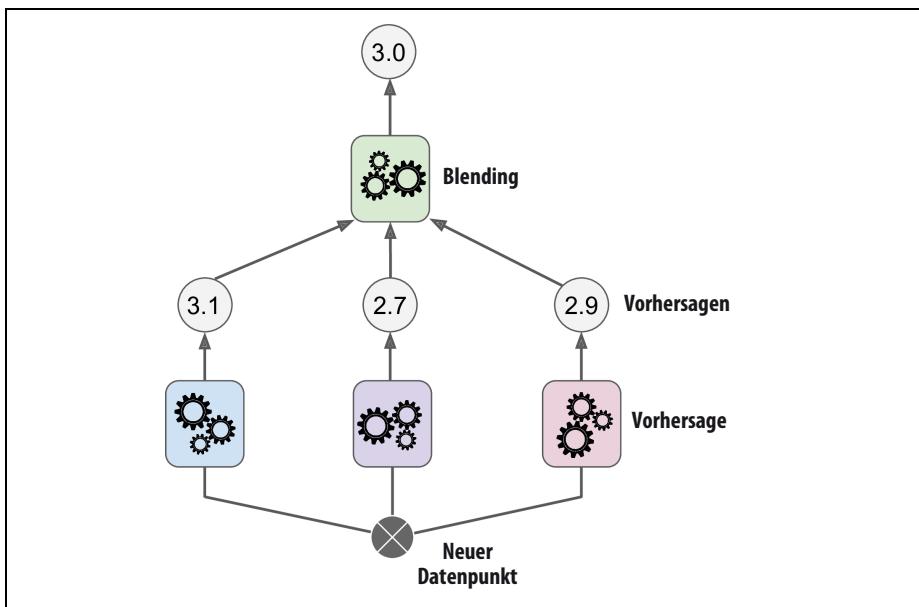


Abbildung 7-12: Aggregieren von Vorhersagen mit einem Blender-Prädiktor

18 »Stacked Generalization«, D. Wolpert (1992).

19 Alternativ kann man auch Out-of-Fold-Vorhersagen verwenden. In einigen Situationen bezeichnet man dies als *Stacking*, während man den Hold-out-Datensatz *Blending* nennt. Die meisten Programmierer verwenden diese Begriffe aber synonym.

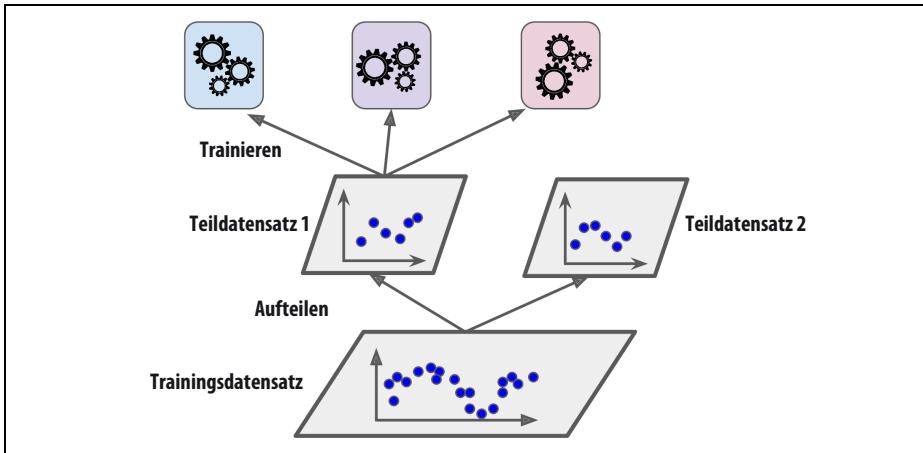


Abbildung 7-13: Trainieren der ersten Stufe

Anschließend verwenden wir diese Prädiktoren, um Vorhersagen auf dem zweiten (beiseitegelegten) Teildatensatz zu treffen (siehe Abbildung 7-14). Dadurch stellen wir sicher, dass die Vorhersagen »sauber« sind, da die Prädiktoren keinen dieser Datenpunkte beim Training gesehen haben.

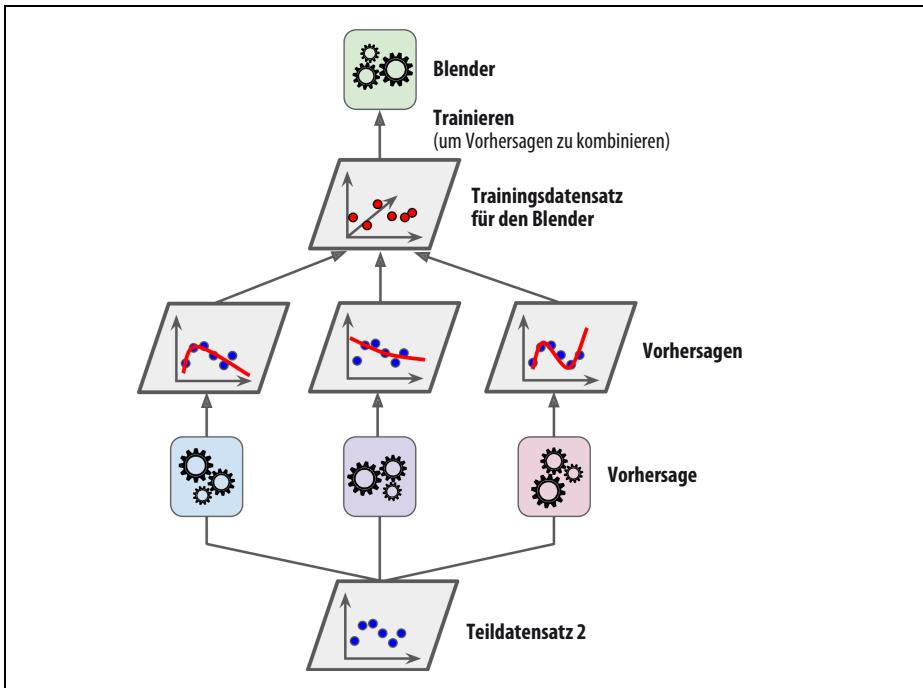


Abbildung 7-14: Trainieren eines Blenders

Nun gibt es für jeden Datenpunkt im Hold-out-Datensatz drei vorhergesagte Werte. Diese vorhergesagten Werte können wir als Eingabedaten verwenden (wodurch unser neuer Trainingsdatensatz drei Dimensionen erhält). Die Zielwerte verwenden wir so, wie sie sind. Der Blender wird auf diesem neuen Datensatz trainiert, sodass er lernt, die Zielwerte aus den Vorhersagen der ersten Stufe vorherzusagen.

Es ist möglich, mehrere unterschiedliche Blender auf diese Weise zu trainieren (z.B. einen mit linearer Regression, einen zweiten als Random-Forest-Regressor und so weiter): Wir erhalten dabei eine Stufe mit Blendern. Der Trick dabei ist, die Trainingsdaten in drei Untermengen aufzuteilen: Mit der ersten wird die erste Stufe trainiert, mit der zweiten wird der Trainingsdatensatz zum Trainieren der zweiten Stufe erstellt (mithilfe der Vorhersagen aus der ersten Stufe), und mit der dritten Teilmenge trainieren wir eine dritte Stufe (mit den Vorhersagen der Prädiktoren der zweiten Stufe). Haben wir das geschafft, können wir Vorhersagen für neue Datenpunkte treffen, indem wir jede Stufe wie in Abbildung 7-15 gezeigt nacheinander abarbeiten.

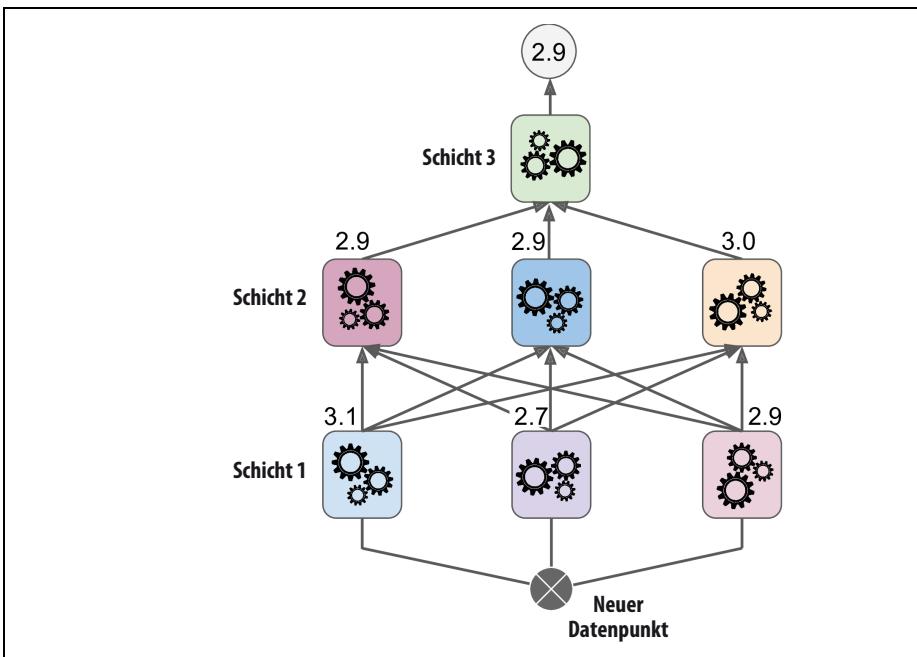


Abbildung 7-15: Vorhersagen in einem mehrschichtigem Stacking-Ensemble

Leider wird Stacking nicht unmittelbar von Scikit-Learn unterstützt. Es ist aber nicht besonders schwer, es selbst zu implementieren (siehe folgende Übungen). Alternativ lässt sich auch eine Open-Source-Implementierung wie brew verwenden (verfügbar unter <https://github.com/viisar/brew>).

Übungen

1. Wenn Sie fünf unterschiedliche Modelle auf den exakt gleichen Trainingsdaten trainiert haben und für alle eine Relevanz von 95% erzielen, lassen sich diese Modelle kombinieren, um ein noch besseres Ergebnis zu erhalten? Begründen Sie Ihre Antwort.
2. Worin unterscheiden sich Klassifikatoren mit Hard und Soft Voting?
3. Ist es möglich, das Trainieren eines Ensembles mit Bagging zu beschleunigen, indem man es auf mehrere Server verteilt? Wie sieht es bei Ensembles mit Pasting, Ensembles mit Boosting, Random Forests oder Ensembles mit Stacking aus?
4. Welchen Vorteil bietet die Out-of-Bag-Evaluation?
5. Wodurch werden Extra-Trees zufälliger als gewöhnliche Random Forests? Wobei hilft dieses zusätzliche Zufallselement? Sind Extra-Trees langsamer oder schneller als gewöhnliche Random Forests?
6. Falls Ihr AdaBoost-Ensemble die Trainingsdaten underfittet, welche Hyperparameter sollten Sie in welcher Weise verändern?
7. Wenn Ihr Gradient-Boosting-Ensemble die Trainingsdaten overfittet, sollten Sie dann die Lernrate erhöhen oder verringern?
8. Laden Sie den MNIST-Datensatz (siehe Kapitel 3) und teilen Sie diesen in Datensätze zum Training, zur Validierung und zum Testen auf (z.B. 40000 Datenpunkte zum Trainieren, 10000 zur Validierung und 10000 zum Testen). Trainieren Sie anschließend unterschiedliche Klassifikatoren, z.B. einen Random-Forest-Klassifikator, einen Extra-Trees-Klassifikator und ein SVM. Versuchen Sie danach, diese zu einem Ensemble zu kombinieren, das besser ist als alle Klassifikatoren auf den Validierungsdaten. Verwenden Sie dazu einen Klassifikator mit Soft oder Hard Voting. Sobald Sie einen gefunden haben, probieren Sie diesen auf dem Testdatensatz aus. Wie viel besser ist das Ensemble im Vergleich zu den einzelnen Klassifikatoren?
9. Führen Sie die einzelnen Klassifikatoren aus der vorigen Übung aus, um Vorhersagen auf den Validierungsdaten zu treffen. Erstellen Sie einen neuen Trainingsdatensatz mit den sich daraus ergebenden Vorhersagen: Jeder Trainingsdatenpunkt ist ein Vektor mit den Vorhersagen sämtlicher Klassifikatoren für ein und dasselbe Bild, und die Zielgröße ist die Kategorie des Bilds. Herzlichen Glückwunsch, Sie haben soeben einen Blender trainiert, der zusammen mit den Klassifikatoren ein Stacking-Ensemble bildet! Werten Sie das Ensemble mit dem Testdatensatz aus. Erstellen Sie für jedes Bild im Testdatensatz mit allen Klassifikatoren Vorhersagen und füttern Sie den Blender mit diesen Vorhersagen, um eine Vorhersage für das Ensemble zu erhalten. Wie schneidet es im Vergleich zum zuvor trainierten abstimmungsbasierten Klassifikator ab?

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

KAPITEL 8

Dimensionsreduktion

Viele Aufgaben beim Machine Learning enthalten für jeden Datenpunkt Tausende oder sogar Millionen Merkmale. Das Training wird dadurch nicht nur extrem langsam, auch das Finden einer geeigneten Lösung wird dadurch erschwert, wie wir gleich sehen werden. Dieses Problem wird bisweilen als *Fluch der Dimensionalität* bezeichnet.

Glücklicherweise lässt sich bei realen Aufgaben die Anzahl der Merkmale beträchtlich reduzieren, sodass sich eine nicht zu bewältigende Aufgabe bearbeiten lässt. Betrachten Sie beispielsweise die MNIST-Bilder (aus Kapitel 3): Die Pixel an den Bildrändern sind so gut wie immer weiß, daher könnten Sie diese Pixel aus dem Trainingsdatensatz entfernen, ohne viel Information zu verlieren. Abbildung 7-6 bestätigt, dass diese Pixel für die Klassifikationsaufgabe völlig bedeutungslos sind. Außerdem korrelieren zwei benachbarte Pixel oft miteinander: Wenn Sie diese zu einem einzelnen Pixel vereinigen (z.B. über den Mittelwert der beiden Intensitäten), verlieren Sie nicht viel Information.



Das Reduzieren der Dimensionen geht mit einem Verlust an Information einher (wie das Komprimieren eines Bilds zum JPEG-Format, dessen Qualität abnehmen kann). Obwohl das Trainieren dadurch beschleunigt wird, kann es die Leistung Ihres Systems ein wenig schmälern. Außerdem werden Ihre Pipelines dadurch etwas komplexer und somit schwieriger zu warten. Sie sollten also Ihr System zuerst mit den ursprünglichen Daten trainieren, bevor Sie im Falle eines zu langsamen Trainings eine Dimensionsreduktion in Betracht ziehen. In einigen Fällen filtert die Dimensionsreduktion der Trainingsdaten allerdings Rauschen und unnötige Details heraus und führt dadurch zu einer höheren Vorhersagequalität (dies ist aber die Ausnahme; in der Regel wird das Training einfach nur schneller).

Außer der Beschleunigung des Trainings ist die Dimensionsreduktion auch für die Datenvisualisierung (oder *DataViz*) äußerst nützlich. Das Reduzieren auf zwei (oder drei) Dimensionen ermöglicht das Plotten eines hochdimensionalen Daten-

satzes als Diagramm, wodurch man häufig wichtige Einblicke in Form von visuell erkennbaren Mustern, z. B. Clustern, erhält.

In diesem Kapitel werden wir den Fluch der Dimensionalität besprechen und einen Eindruck davon erhalten, was in hochdimensionalen Räumen passiert. Anschließend werden wir zwei Ansätze zur Dimensionsreduktion vorstellen (Projektion und Manifold Learning) und drei der beliebtesten Techniken zur Dimensionsreduktion ausprobieren: PCA, Kernel PCA und LLE.

Der Fluch der Dimensionalität

Wir sind so sehr an das Leben in drei Dimensionen gewöhnt¹, dass unsere Vorstellungskraft an höher dimensionalen Räumen scheitert. Selbst einen einfachen 4-D-Hyperwürfel kann man sich nur unglaublich schwer vorstellen (siehe Abbildung 8-1), von einem 200-dimensionalen Ellipsoid, der in einen 1000-dimensionalen Raum gekrümmkt ist, einmal ganz zu schweigen.

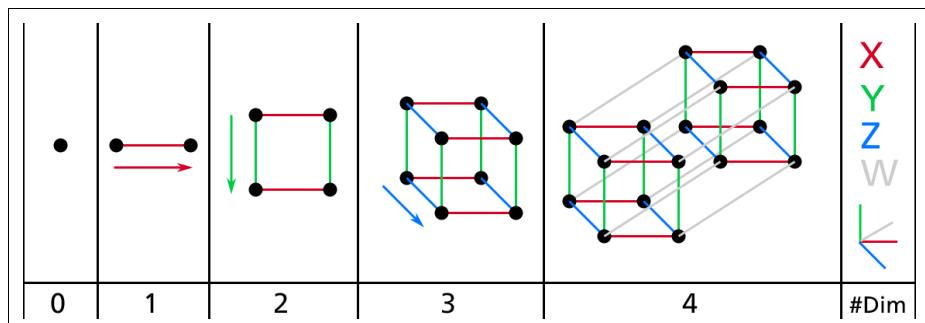


Abbildung 8-1: Punkt, Strecke, Quadrat, Würfel und Tesserakt (0-D bis 4-D Hyperwürfel)²

Es stellt sich heraus, dass sich in höher dimensionalen Räumen viele Dinge anders verhalten. Wenn Sie beispielsweise einen zufälligen Punkt in einem Einheitsquadrat (ein Quadrat mit den Abmessungen 1×1) betrachten, so beträgt die Wahrscheinlichkeit, näher als 0.001 an einer Kante zu sein, weniger als 0.4% (anders gesagt, ist es sehr unwahrscheinlich, dass ein zufälliger Punkt »extrem« in Bezug auf eine der Dimensionen ist). Aber in einem 10000-dimensionalen Einheitshyperwürfel (ein Würfel mit den Kantenlängen $1 \times 1 \times \dots \times 1$ und insgesamt 10000 Ecken) steigt diese Wahrscheinlichkeit auf mehr als 99.999999% an. Die meisten Punkte in einem hochdimensionalen Hyperwürfel liegen sehr nah am Rand.³

¹ In Ordnung, vier Dimensionen, wenn Sie die Zeit dazu zählen, und einige mehr, wenn Sie sich mit der Stringtheorie beschäftigen.

² Einen in den 3-D-Raum projizierten rotierenden Tesserakt finden Sie auf <http://goo.gl/OM7ktJ>. Bild vom Wikipedia-Nutzer NerdBoy1392 (Creative Commons BY-SA 3.0, <https://creativecommons.org/licenses/by-sa/3.0/>). Reproduziert von <https://en.wikipedia.org/wiki/Tesseract>.

Hier ist ein weitaus lästigerer Unterschied: Wenn Sie in einem Einheitsquadrat zwei zufällige Punkte auswählen, beträgt der Abstand zwischen diesen Punkten durchschnittlich 0.52. Wenn Sie in einem 3-D-Würfel zwei zufällige Punkte bestimmen, beträgt der Abstand zwischen diesen Punkten durchschnittlich 0.66. Wie aber sieht es bei zwei zufällig ausgewählten Punkten in einem 1000000-dimensionalen Hyperwürfel aus? Deren Abstand beträgt, ob Sie es glauben oder nicht, im Durchschnitt etwa 408.25 (etwa $\sqrt{1000000/6}$)! Dies ist wenig intuitiv: Wie können zwei Punkte so weit auseinander liegen, wenn sie beide im gleichen Einheitshyperwürfel liegen? Diese Tatsache deutet darauf hin, dass hochdimensionale Datensätze tendenziell sehr dünn besetzt sind: Die meisten Trainingsdatenpunkte liegen mit hoher Wahrscheinlichkeit weit voneinander entfernt. Das bedeutet auch, dass jeder neue Datenpunkt vermutlich weit weg von allen Trainingsdatenpunkten liegt, wodurch Vorhersagen weitaus weniger zuverlässig als bei wenigen Dimensionen sind, da ihnen größere Extrapolationen zugrunde liegen. Kurz, je mehr Dimensionen ein Trainingsdatensatz besitzt, umso größer ist die Gefahr des Overfitting.

Theoretisch wäre eine Lösung zum Fluch der Dimensionalität, die Größe des Trainingsdatensatzes zu vergrößern, bis die Dichte der Trainingsdatenpunkte ausreichend ist. In der Praxis wächst die Anzahl der für eine bestimmte Dichte nötigen Datenpunkte exponentiell mit der Anzahl Dimensionen. Mit nur 100 Merkmalen (viel weniger als bei der MNIST-Aufgabe) würden Sie mehr Trainingsdatenpunkte benötigen, als es Atome im beobachtbaren Universum gibt, damit die Punkte im Abstand von durchschnittlich 0.1 zueinander liegen, sofern sie einheitlich über alle Dimensionen verteilt sind.

Die wichtigsten Ansätze zur Dimensionsreduktion

Bevor wir uns mit einzelnen Algorithmen zur Dimensionsreduktion auseinander setzen, betrachten wir die zwei wichtigsten Ansätze zum Reduzieren von Dimensionen: Projektion und Manifold Learning.

Projektion

In den meisten Aufgaben im wirklichen Leben sind die Trainingsdaten *nicht* gleichmäßig über alle Dimensionen verteilt. Viele Merkmale sind annähernd konstant, andere sind in hohem Maße miteinander korreliert (wie anhand des MNIST-Beispiels besprochen). Deshalb liegen sämtliche Trainingsdatenpunkte innerhalb (oder nahe bei) eines viel niedriger dimensionalen *Subraums* des höher dimensionalen Raums. Da dies sehr abstrakt klingt, sehen wir uns ein Beispiel an. In Abbildung 8-2 erkennen Sie einen durch Kreise dargestellten 3-D-Datensatz.

3 Wenn Sie genug Dimensionen berücksichtigen, vertritt jeder Ihrer Bekannten vermutlich in mindestens einer Dimension extreme Ansichten (z.B. wie viel Zucker sie in ihren Kaffee tun).

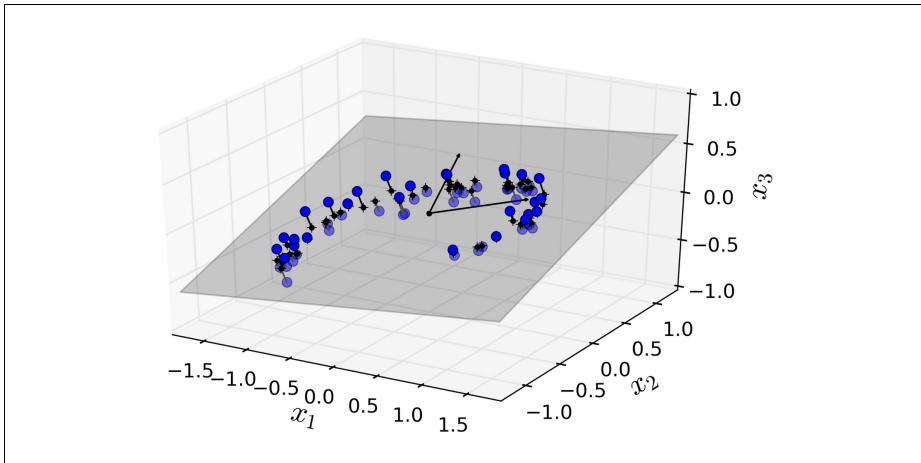


Abbildung 8-2: Ein 3-D-Datensatz in der Nähe eines 2-D-Subraums

Beachten Sie, dass sämtliche Trainingsdatenpunkte nahe einer Ebene liegen: Dies ist der niedriger dimensionale (2-D-)Subraum des höher dimensionalen (3-D-)Raums. Wenn wir nun jeden Trainingsdatenpunkt lotrecht auf diesen Subraum fallen lassen (wie durch die kurzen Linien zwischen Datenpunkten und der Ebene angezeigt), erhalten wir die in Abbildung 8-3 dargestellten neuen Datensatz. Ta-da! Wir haben soeben die Dimensionen des Datensatzes von 3-D nach 2-D reduziert. Allerdings entsprechen die Achsen nun den neuen Merkmalen z_1 und z_2 (den Koordinaten der Projektion auf die Ebene).

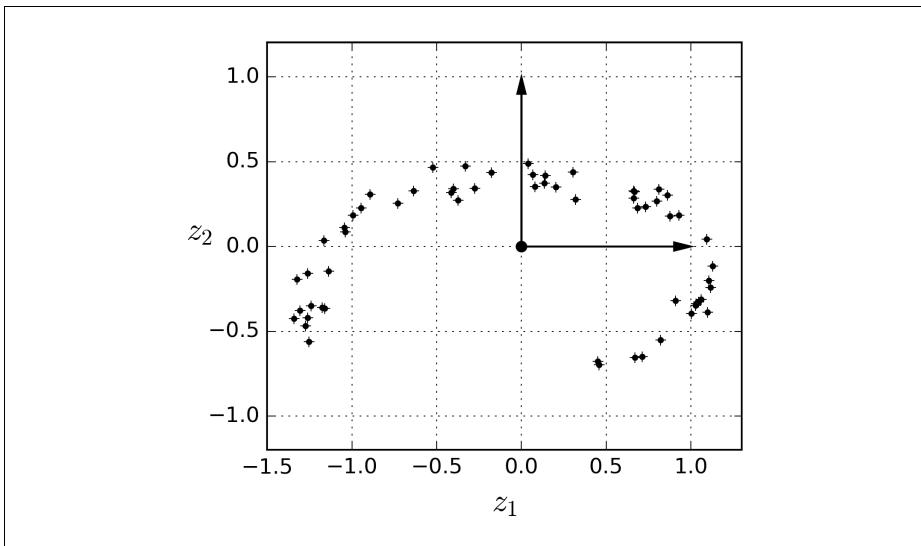


Abbildung 8-3: Der neue 2-D-Datensatz nach der Projektion

Eine Projektion ist jedoch nicht immer die beste Lösung zur Dimensionsreduktion. In vielen Fällen ist der Subraum gekrümmt und verdreht, wie im Falle des berühmten Swiss Roll-Datensatzes in Abbildung 8-4.

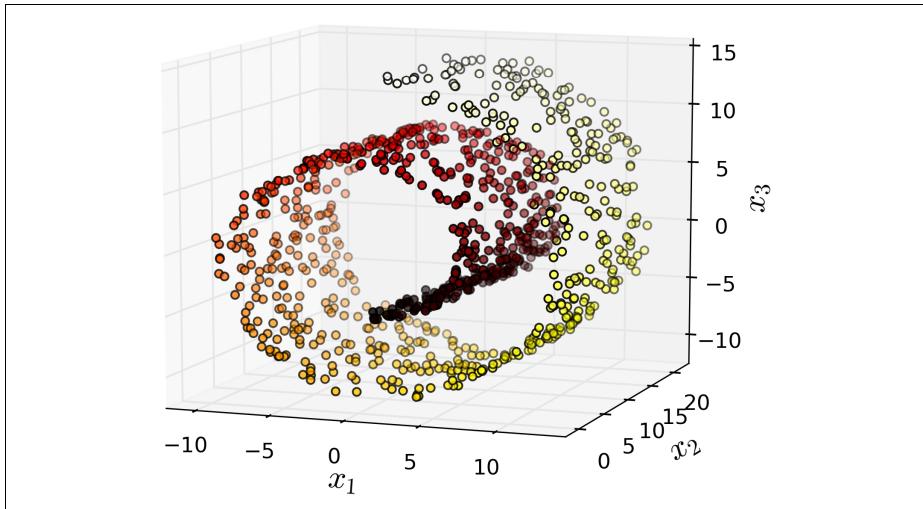


Abbildung 8-4: Swiss-Roll-Datensatz

Eine einfache Projektion auf eine Ebene (z.B. durch Weglassen von x_3) würde unterschiedliche Ebenen der Swiss Roll zusammenquetschen, wie die linke Seite von Abbildung 8-5 zeigt. Was Sie stattdessen wirklich benötigen, ist, die Swiss Roll aufzurollen, um den 2-D-Datensatz auf der rechten Seite von Abbildung 8-5 zu erhalten.

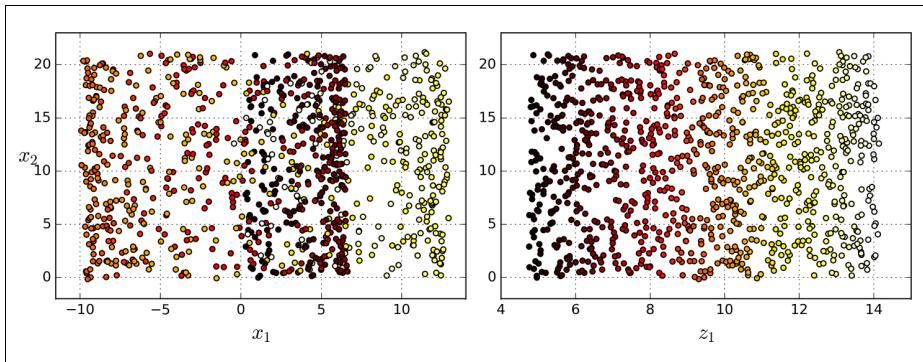


Abbildung 8-5: Eine auf eine Ebene gequetschte Projektion (links) im Vergleich zum Aufrollen der Swiss-Roll-Daten (rechts)

Manifold Learning

Der Swiss-Roll-Datensatz ist ein Beispiel für ein *2-D-Manifold*. Einfach ausgedrückt ist ein 2-D-Manifold eine zweidimensionale Form, die in einem höher dimensionierten Raum verzerrt und verdreht ist. Allgemeiner ist ein d -dimensionaler Manifold Teil eines n -dimensionalen Raums (wobei $d < n$), der lokal einer d -dimensionalen Hyperebene ähnelt. Im Falle der Swiss-Roll-Daten, gilt $d = 2$ und $n = 3$: Lokal sind sie eine 2-D-Ebene, die in der dritten Dimension aufgerollt ist.

Viele Algorithmen zur Dimensionsreduktion modellieren das *Manifold*, auf dem die Trainingsdatenpunkte liegen; dies nennt man *Manifold Learning*. Ihm liegt die *Manifold-Annahme* oder *Manifold-Hypothese* zugrunde, die besagt, dass die meisten höher dimensionalen Datensätze in der Nähe eines Manifold mit deutlich weniger Dimensionen liegen. Diese Annahme ist sehr häufig empirisch bestätigt worden.

Erinnern Sie sich noch einmal an den MNIST-Datensatz: Alle handgeschriebenen Ziffern weisen einige Ähnlichkeiten auf. Sie bestehen aus verbundenen Linien, die Ränder sind weiß, sie sind mehr oder weniger zentriert und so weiter. Wenn Sie zufällig Bilder erzeugen würden, würde nur ein lächerlich geringer Bruchteil nach handgeschriebenen Ziffern aussehen. Anders ausgedrückt, sind die Freiheitsgrade beim Erzeugen des Bilds einer Ziffer sehr viel geringer als beim Erzeugen eines beliebigen Bilds. Diese Beschränkungen zwingen den Datensatz in ein niedriger dimensioniertes Manifold.

Eine zweite, implizite Annahme geht häufig mit der Manifold-Annahme einher: Die zu lösende Aufgabe (z.B. Klassifikation oder Regression) ist mit einem durch weniger Dimensionen ausgedrückten Manifold einfacher zu lösen. Beispielsweise sind die Swiss-Roll-Daten in der oberen Zeile von Abbildung 8-6 in zwei Kategorien geteilt: Im 3-D-Raum (links) wäre die Entscheidungsgrenze sehr komplex, aber im aufgerollten 2-D-Manifold (rechts) ist die Entscheidungsgrenze eine einfache Gerade.

Allerdings trifft diese Annahme nicht immer zu. Beispielsweise liegt die Entscheidungsgrenze in der unteren Zeile von Abbildung 8-6 bei $x_1 = 5$. Diese Entscheidungsgrenze sieht im ursprünglichen 3-D-Raum sehr einfach aus (eine vertikale Ebene), im aufgerollten Manifold ist sie jedoch deutlich komplexer (eine Menge von vier unabhängigen Strecken).

Kurz, wenn Sie die Dimensionalität Ihres Trainingsdatensatzes vor dem Trainieren eines Modells reduzieren, wird das Training definitiv beschleunigt. Allerdings wird die Lösung dadurch nicht unbedingt besser oder einfacher; dies hängt vollständig von den Daten ab.

Sie haben nun hoffentlich einen Eindruck davon, was der Fluch der Dimensionalität ist und wie Algorithmen zur Dimensionsreduktion diesem entgegenwirken kön-

nen, besonders wenn die Manifold-Annahme gilt. Im restlichen Kapitel werden wir uns einige der beliebtesten Algorithmen anschauen.

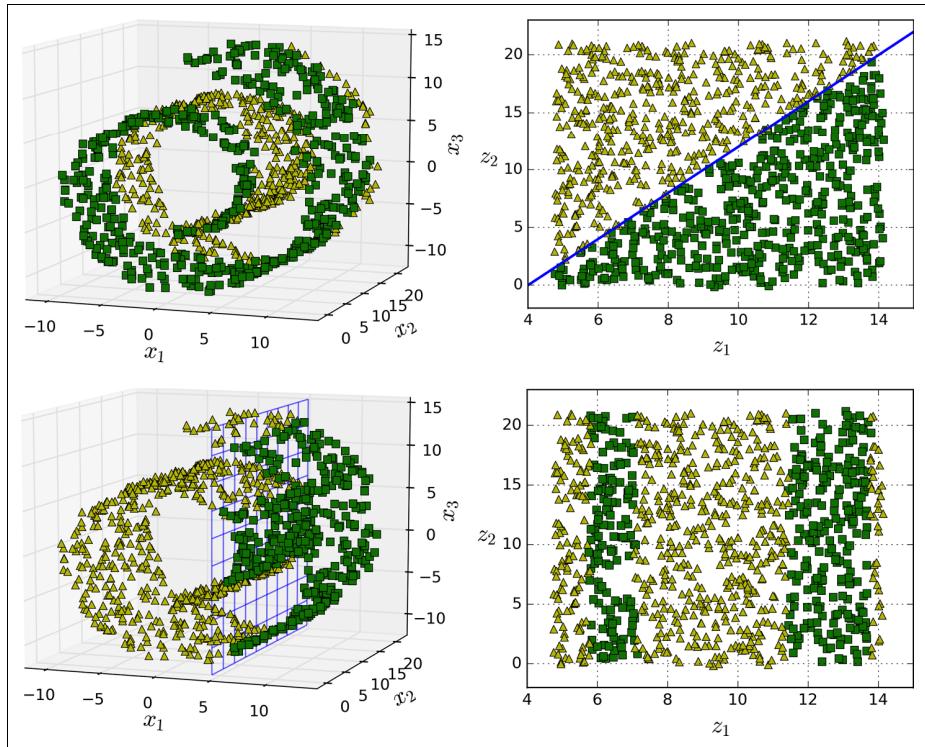


Abbildung 8-6: Die Entscheidungsgrenze wird bei weniger Dimensionen nicht immer einfacher.

Hauptkomponentenzerlegung (PCA)

Die *Hauptkomponentenzerlegung* (PCA) ist das bei Weitem beliebteste Verfahren zur Dimensionsreduktion. Sie findet zunächst diejenige Hyperebene, die den Daten am nächsten liegt, und projiziert die Daten anschließend darauf.

Erhalten der Varianz

Bevor Sie die Trainingsdaten auf eine niedriger dimensionale Hyperebene projizieren können, müssen Sie zunächst die richtige Hyperebene auswählen. Als Beispiel ist in Abbildung 8-7 auf der linken Seite ein einfacher 2-D-Datensatz mit drei unterschiedlichen Achsen (d.h. eindimensionalen Hyperebenen) dargestellt. Auf der rechten Seite sehen Sie die Projektionen des Datensatzes auf jede dieser Achsen. Wie Sie sehen, bleibt bei der Projektion auf die durchgezogene Linie ein Maximum

an Varianz erhalten, während bei der Projektion auf die gepunktete Linie nur sehr wenig Varianz übrig ist. Die Projektion auf die gestichelte Linie liegt irgendwo dazwischen.

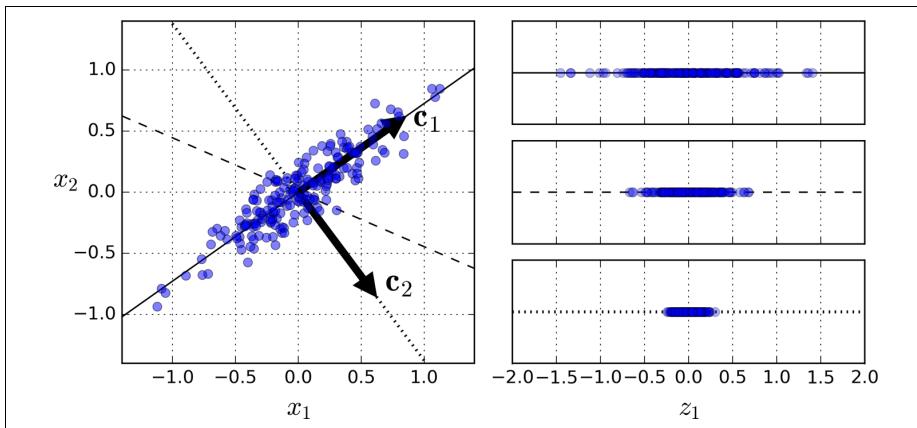


Abbildung 8-7: Auswählen eines Subraums für die Projektion

Es scheint sinnvoll, die Achse so auszuwählen, dass ein Maximum an Varianz erhalten bleibt, weil so weniger Information verloren geht als bei den übrigen Projektionen. Die Auswahl lässt sich auch damit begründen, dass der mittlere quadratische Abstand zwischen dem ursprünglichen Datensatz und der Projektion minimal ist. Diese einfache Grundidee ist die Essenz der *Hauptkomponentenzerlegung* (<http://goo.gl/gbNo1D>).⁴

Hauptkomponenten

Bei der PCA wird die Achse gesucht, auf der die größte Varianz der Trainingsdaten liegt. In Abbildung 8-7 ist dies die durchgezogene Linie. Auch eine Achse, zur ersten Achse orthogonal, wird gefunden, die der größten verbliebenen Varianz entspricht. In diesem 2-D-Beispiel gibt es dabei keine Wahl: Es ist die gepunktete Linie. In einem höher dimensionalen Datensatz würde die PCA auch eine dritte Achse, zu den beiden vorigen Achsen orthogonal, finden, dann eine vierte, eine fünfte und so weiter – so viele Achsen, wie der Datensatz Dimensionen besitzt.

Der Einheitsvektor, der die i^{te} Achse definiert, wird die i^{te} *Hauptkomponente* (PC) genannt. In Abbildung 8-7 ist die 1. Hauptkomponente c_1 und die zweite c_2 . In Abbildung 8-2 sind die ersten zwei Hauptkomponenten als orthogonale Pfeile in der Ebene dargestellt. Die dritte Hauptkomponente wäre orthogonal zur Ebene (nach oben oder unten).

⁴ »On Lines and Planes of Closest Fit to Systems of Points in Space«, K. Pearson (1901).



Die Richtung der Hauptkomponenten ist nicht stabil: Wenn Sie den Trainingsdatensatz leicht durchmischen und die PCA erneut durchführen, zeigen einige Hauptkomponenten in die entgegengesetzte Richtung. Allerdings liegen sie noch immer auf den gleichen Achsen. In einigen Fällen können einige Hauptkomponenten sogar rotieren oder die Plätze tauschen, aber die von ihnen aufgespannte Ebene bleibt die gleiche.

Wie lassen sich die Hauptkomponenten für einen Trainingsdatensatz finden? Glücklicherweise gibt es eine Standardtechnik zur Matrizenfaktorisierung, die *Singular Value Decomposition* (SVD), mit der Sie die Matrix \mathbf{X} mit den Trainingsdaten in das Skalaproduct der drei Matrizen $\mathbf{U} \cdot \Sigma \cdot \mathbf{V}$ zerlegen können, wobei \mathbf{V} sämtliche gesuchten Hauptkomponenten enthält, wie in Formel 8-1 dargestellt.

Formel 8-1: Matrix der Hauptkomponenten

$$\mathbf{V} = \begin{pmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \dots & \mathbf{c}_n \\ | & | & | \end{pmatrix}$$

Der folgende Python-Code verwendet die NumPy-Funktion `svd()`, um sämtliche Hauptkomponenten für den Trainingsdatensatz zu berechnen, und extrahiert dann die ersten zwei:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```



Die PCA geht davon aus, dass der Datensatz um den Koordinatenursprung zentriert ist. Wie wir sehen werden, kümmern sich die PCA-Klassen in Scikit-Learn um das Zentrieren der Daten. Wenn Sie allerdings die PCA selbst implementieren (wie im obigen Beispiel) oder wenn Sie andere Bibliotheken verwenden, sollten Sie das Zentrieren der Daten nicht vergessen.

Die Projektion auf d Dimensionen

Haben Sie erst einmal sämtliche Hauptkomponenten gefunden, können Sie die Dimensionen Ihres Datensatzes auf d Dimensionen reduzieren, indem Sie ihn auf die von den ersten d Hauptkomponenten aufgespannte Hyperebene projizieren. Die Auswahl dieser Hyperebene stellt sicher, dass die Projektion so viel Varianz wie möglich bewahrt. Beispielsweise wird in Abbildung 8-2 ein 3-D-Datensatz auf die von den ersten zwei Hauptkomponenten definierte 2-D-Ebene projiziert, wobei ein großer Teil der Varianz im Datensatz erhalten bleibt. Als Ergebnis ist die 2-D-Projektion dem ursprünglichen 3-D-Datensatz sehr ähnlich.

Um den Trainingsdatensatz auf die Hyperebene zu projizieren, können Sie einfach das Skalarprodukt aus der Matrix \mathbf{X} mit den Trainingsdaten und der Matrix \mathbf{W}_d aus den ersten d Hauptkomponenten bilden (d.h. einer Matrix aus den ersten d Spalten von \mathbf{V}), wie Formel 8-2 zeigt.

Formel 8-2: Projektion des Trainingsdatensatzes auf d Dimensionen

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \cdot \mathbf{W}_d$$

Der folgende Python-Code projiziert den Trainingsdatensatz auf eine durch die ersten zwei Hauptkomponenten aufgespannte Ebene:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

Da haben Sie es! Sie wissen nun, wie Sie die Dimensionalität eines Datensatzes auf eine beliebige Anzahl Dimensionen reduzieren können und dabei so viel Varianz wie möglich beibehalten.

Verwenden von Scikit-Learn

Die Scikit-Learn-Klasse PCA implementiert die PCA mit dem zuvor vorgestellten Verfahren der SVD-Zerlegung. Das folgende Codebeispiel wendet die PCA an, um die Dimensionalität des Datensatzes auf zwei Dimensionen zu reduzieren (beachten Sie, dass die Daten automatisch zentriert werden):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

Nach dem Anpassen des PCA-Transformers auf den Datensatz können Sie über das Attribut `components_` auf die Hauptkomponenten zugreifen (es enthält die Hauptkomponenten als horizontale Vektoren, sodass `pca.components_.T[:, 0]` der ersten Hauptkomponente entspricht).

Der Anteil erklärter Varianz

Eine weitere sehr nützliche Information ist der *Anteil erklärter Varianz* jeder Hauptkomponente, der im Attribut `explained_variance_ratio_` verfügbar ist. Dieses zeigt an, welcher Teil der Varianz auf der Achse jeder Hauptkomponente liegt. Betrachten wir als Beispiel den Anteil der Varianz der ersten zwei Hauptkomponenten des in Abbildung 8-2 dargestellten 3-D-Datensatzes:

```
>>> pca.explained_variance_ratio_
array([ 0.84248607,  0.14631839])
```

Dies sagt uns, dass 84.2 % der Varianz im Datensatz entlang der ersten Achse liegen und 14.6 % entlang der zweiten Achse. Damit bleiben weniger als 1.2 % für die dritte Achse übrig, man darf also annehmen, dass diese nur wenig Information enthält.

Auswählen der richtigen Anzahl Dimensionen

Anstatt die Anzahl verbleibender Dimensionen willkürlich auszuwählen, sollte man grundsätzlich die Anzahl Dimensionen so wählen, dass diese kumulativ einen ausreichend großen Anteil der Varianz abdecken (z.B. 95%). Es sei denn, Sie möchten die Dimensionen zur Visualisierung reduzieren – in diesem Falle sollten Sie die Anzahl Dimensionen auf 2 oder 3 reduzieren.

Der folgende Code berechnet eine PCA ohne Reduktion der Dimensionalität und berechnet anschließend die minimale Anzahl Dimensionen, um 95 % der Varianz im Trainingsdatensatz abzudecken:

```
pca = PCA()  
pca.fit(X_train)  
cumsum = np.cumsum(pca.explained_variance_ratio_ )  
d = np.argmax(cumsum >= 0.95) + 1
```

Sie könnten anschließend die Hauptkomponentenzerlegung erneut mit `n_components=d` durchführen. Es gibt jedoch eine weitaus bessere Alternative: Statt die Anzahl der Hauptkomponenten, die abgedeckt werden sollen, festzulegen, können Sie auch den Wert `n_components` auf eine Zahl zwischen 0.0 und 1.0 setzen, was dann als Anteil der abzudeckenden Varianz interpretiert wird:

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X_train)
```

Noch eine Möglichkeit wäre, die erklärte Varianz als Funktion der Anzahl Dimensionen aufzutragen (durch Plotten von `cumsum`; siehe Abbildung 8-8). Meist gibt es in der Kurve einen Knick, ab dem die Varianz nicht mehr so schnell ansteigt. Sie können dies als intrinsische Dimensionalität des Datensatzes ansehen. In diesem Fall sehen Sie, dass bei einer Reduktion auf 100 Dimensionen nur wenig der erklärten Varianz verloren ginge.

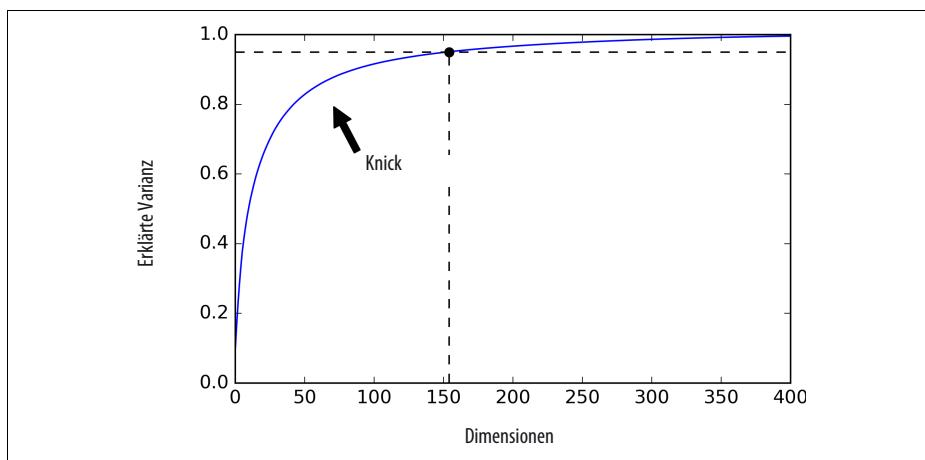


Abbildung 8-8: Erklärte Varianz als Funktion der Anzahl Dimensionen

PCA als Komprimierungsverfahren

Offensichtlich nimmt ein Trainingsdatensatz nach einer Dimensionsreduktion deutlich weniger Platz ein. Versuchen Sie beispielsweise, eine PCA auf den MNIST-Datensatz unter Beibehalten von 95 % der Varianz anzuwenden. Sie sollten herausfinden, dass jeder Datenpunkt nur aus etwas mehr als 150 Merkmalen anstatt der ursprünglichen 784 Merkmale besteht. Während wir also einen Großteil der Varianz beibehalten, ist der Datensatz auf weniger als 20 % der ursprünglichen Größe geschrumpft! Dies ist eine anständige Komprimierungsrate, was einen Klassifikationsalgorithmus (z.B. einen SVM-Klassifikator) erheblich beschleunigen kann.

Es ist auch möglich, den reduzierten Datensatz zurück auf die 784 Dimensionen zu projizieren, indem man die zur PCA-Projektion inverse Transformation durchführt. Natürlich erhalten Sie dabei nicht die Originaldaten, da bei der Projektion ein wenig Information verloren ging (die restlichen 5 % der Varianz), aber wir werden vermutlich recht nah an den Ursprungsdaten liegen. Den mittleren quadratischen Abstand zwischen den rekonstruierten Daten (komprimiert und anschließend dekomprimiert) nennt man den *Rekonstruktionsfehler*. Als Beispiel komprimiert der folgende Code den MNIST-Datensatz auf 154 Dimensionen und verwendet anschließend die Methode `inverse_transform()`, um ihn wieder auf 784 Dimensionen zu dekomprimieren. Abbildung 8-9 zeigt einige Ziffern aus dem ursprünglichen Datensatz (links) und die dazugehörigen Ziffern nach Komprimierung und Dekomprimierung. Wie Sie sehen, kommt es zu einem geringen Verlust an Bildqualität, aber die Ziffern sind größtenteils intakt.

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

Original	Komprimiert
1 4 0 1 7	1 4 0 1 7
8 3 0 5 1	8 3 0 5 1
0 2 7 7 6	0 2 7 7 6
2 2 0 6 2	2 2 0 6 2
4 5 8 3 4	4 5 8 3 4

Abbildung 8-9: Komprimierung der MNIST-Daten unter Beibehaltung von 95 % der Varianz

Die Gleichung der Rücktransformation ist in Formel 8-3 angegeben.

Formel 8-3: Rücktransformation der Hauptkomponenten zur ursprünglichen Anzahl Dimensionen

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d-\text{proj}} \cdot \mathbf{W}_d^T$$

Inkrementelle PCA

Eine Schwierigkeit bei der obigen Implementierung der PCA ist, dass der gesamte Trainingsdatensatz in den Speicher passen muss, damit der SVD-Algorithmus ausgeführt werden kann. Glücklicherweise wurden *inkrementelle PCA*-(IPCA)-Algorithmen entwickelt: Sie können den Trainingsdatensatz in kleinere Portionen aufteilen und einen IPCA-Algorithmus mit einer Portion nach der anderen füttern. Dies ist bei großen Trainingsdatensätzen und beim Verwenden von PCA in Online-Umgebungen hilfreich (d.h., sobald neue Datenpunkte eintreffen).

Der folgende Code teilt den MNIST-Datensatz in 100 Portionen auf (mit der Funktion `array_split()` aus NumPy) und speist sie in die Scikit-Learn-Klasse `IncrementalPCA` (<http://goo.gl/FmdhUP>) ein⁵, um die Dimensionalität des MNIST-Datensatzes auf 154 Dimensionen zu reduzieren (wie zuvor). Beachten Sie, dass Sie bei jedem Teildatensatz die Methode `partial_fit()` anstelle von `fit()` mit den gesamten Trainingsdaten aufrufen müssen:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternativ können Sie die NumPy-Klasse `memmap` verwenden, mit der Sie ein großes als Binärdatei auf der Festplatte gespeichertes Array so manipulieren können, als wäre es komplett im Speicher; die Klasse lädt lediglich die jeweils benötigten Daten in den Speicher. Da die Klasse `IncrementalPCA` zu jedem Zeitpunkt nur einen kleinen Teil der Daten verwendet, hält sich der Speicherverbrauch in Grenzen. Damit ist es möglich, die Methode `fit()` wie gewohnt aufzurufen:

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

Randomisierte PCA

Scikit-Learn stellt eine weitere Art von Hauptkomponentenzerlegung zur Verfügung, die *randomisierte PCA*. Diese basiert auf einem stochastischen Algorithmus, der schnell eine Approximation der ersten d Hauptkomponenten findet. Er besitzt eine Komplexität der Rechenzeit von $O(m \times d^2) + O(d^3)$ anstatt von $O(m \times n^2) + O(n^3)$ und ist damit den vorigen Algorithmen weit überlegen, sofern d deutlich kleiner als n ist.

⁵ Scikit-Learn verwendet den in »Incremental Learning for Robust Visual Tracking«, D. Ross et al. (2007) beschriebenen Algorithmus.

```

rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)

```

Kernel PCA

In Kapitel 5 haben wir den Kernel-Trick kennengelernt, ein mathematisches Instrument zur impliziten Zuordnung von Datenpunkten in einen sehr hochdimensionalen Raum (den *Merkmalsraum*), das nichtlineare Klassifikation und Regression mit Support Vector Machines ermöglicht. Einer linearen Entscheidungsgrenze im hochdimensionalen Merkmalsraum entspricht eine komplexe nichtlineare Entscheidungsgrenze im *ursprünglichen Raum*.

Der gleiche Trick lässt sich auch bei der PCA anwenden, wodurch sich komplexe nichtlineare Projektionen für die Dimensionsreduktion einsetzen lassen. Dies bezeichnet man als *Kernel PCA (kPCA)* (<http://goo.gl/5lQT5Q>).⁶ Sie ist dazu geeignet, bei der Projektion Cluster von Datenpunkten zusammenzuhalten oder bisweilen sogar Datensätze in der Nähe eines verdrehten Manifolds aufzurollen.

Als Beispiel verwendet der folgende Code die Scikit-Learn-Klasse KernelPCA, um die kPCA mit einem RBF-Kernel durchzuführen (Details über den RBF-Kernel und andere Kernels finden Sie in Kapitel 5):

```

from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)

```

Abbildung 8-10 zeigt den auf zwei Dimensionen reduzierten Swiss-Roll-Datensatz, einmal mit einem linearen Kernel (äquivalent zur Klasse PCA), mit einem RBF-Kernel, und mit einem sigmoiden Kernel (logistisch).

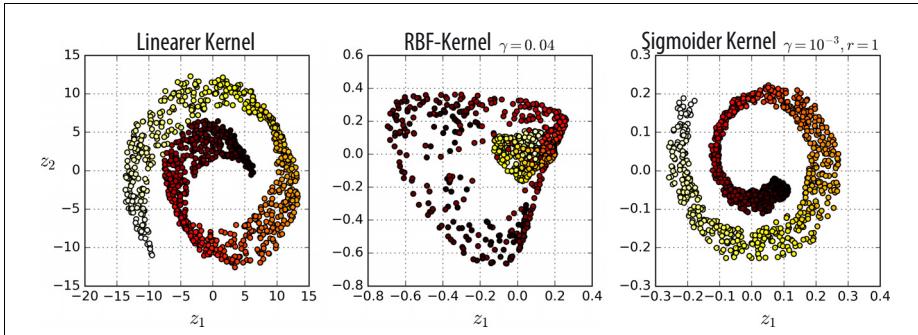


Abbildung 8-10: Mit kPCA auf 2-D reduzierte Swiss-Roll-Daten unter Verwendung unterschiedlicher Kernel

⁶ »Kernel Principal Component Analysis«, B. Schölkopf, A. Smola, K. Müller (1999).

Auswahl eines Kernels und Optimierung der Hyperparameter

Da die kPCA ein unüberwachter Lernalgorithmus ist, gibt es kein offensichtliches Qualitätsmaß, das uns bei der Auswahl des besten Kernels und der Hyperparameter behilflich ist. Allerdings ist die Dimensionsreduktion oft ein Vorverarbeitungsschritt für überwachte Lernaufgaben (z.B. Klassifikation). Sie können also eine Gittersuche einsetzen, um Kernel und Hyperparameter so zu wählen, dass die bestmögliche Leistung bei der Vorhersage erzielt wird. Beispielsweise erstellt der folgende Code eine aus zwei Schritten bestehende Pipeline, in der zuerst die Dimensionalität mittels kPCA auf zwei Dimensionen reduziert wird. Anschließend wird zur Klassifikation eine logistische Regression durchgeführt. Wir verwenden GridSearchCV, um Kernel und Wert für gamma zu finden, die am Ende der Pipeline die genaueste Klassifikation liefern:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("k pca", KernelPCA(n_components=2)),
    ("log reg", LogisticRegression())
])

param_grid = [
    {"k pca_gamma": np.linspace(0.03, 0.05, 10),
     "k pca_kernel": ["rbf", "sigmoid"]}
]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Der beste Kernel und die Hyperparameter sind im Attribut `best_params_` gespeichert:

```
>>> print(grid_search.best_params_)
{'k pca_gamma': 0.04333333333333335, 'k pca_kernel': 'rbf'}
```

Ein anderer, diesmal vollständig unüberwachter Ansatz ist, Kernel und Hyperparameter so zu wählen, dass der Fehler bei der Rekonstruktion minimal wird. Dies ist jedoch nicht so einfach wie bei der linearen PCA. Und zwar aus folgendem Grund: Abbildung 8-11 zeigt den ursprünglichen Swiss-Roll-Datensatz in 3-D (links oben) und den 2-D-Datensatz nach kPCA mit einem RBF-Kernel (rechts oben). Dank dem Kernel-Trick entspricht dies mathematisch der Zuordnung der Trainingsdaten zu einem Merkmalsraum mit unendlich vielen Dimensionen (unten rechts) über die *Merkmalszuordnung* φ mit anschließender Projektion der transformierten Trainingsdaten zu 2-D über lineare PCA. Wenn wir den linearen PCA-Schritt für einen gegebenen Datenpunkt im reduzierten Raum umkehren könnten, würde der rekonstruierte Punkt im Merkmalsraum liegen, nicht im Ursprungsräum (wie z.B. der im Diagramm durch ein x gekennzeichnete). Da der Merkmalsraum unendlich viele Dimensionen hat, ist der rekonstruierte Punkt nicht berechenbar, und deshalb

können wir den Fehler der Rekonstruktion auch nicht bestimmen. Glücklicherweise lässt sich ein Punkt im Ursprungsraum finden, der sich dem rekonstruierten Punkt zuordnen lässt. Dies bezeichnet man als Reconstruction *Pre-Image*. Sie können den quadratischen Abstand vom Pre-Image zum ursprünglichen Datenpunkt bestimmen und anschließend Kernel und Hyperparameter so wählen, dass die Abweichung vom Reconstruction Pre-Image minimal wird.

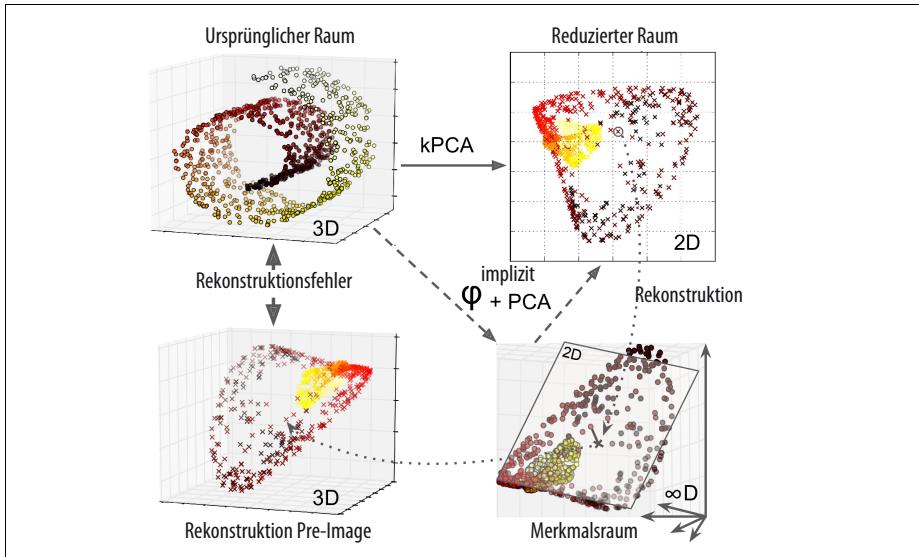


Abbildung 8-11: Kernel PCA und die Abweichung vom Reconstruction Pre-Image

Sie fragen sich womöglich, wie sich diese Rekonstruktion praktisch durchführen lässt. Eine Möglichkeit ist, ein überwachtes Regressionsmodell mit den projizierten Datenpunkten als Trainingsdatensatz und mit den ursprünglichen Datenpunkten als Zielgröße zu trainieren. Scikit-Learn tut dies automatisch, wenn Sie den Parameter `fit_inverse_transform=True` setzen, wie das folgende Codebeispiel demonstriert:⁷

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                     fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```



Standardmäßig gibt es mit `fit_inverse_transform=False` und `KernelPCA` keine Methode `inverse_transform()`. Diese Methode wird nur mit `fit_inverse_transform=True` erstellt.

⁷ Scikit-Learn verwendet den Kernel-Ridge-Regressionsalgorithmus, beschrieben in Gokhan H. Bakir, Jason Weston, and Bernhard Scholkopf, »Learning to Find Pre-images« (<http://goo.gl/d0ydY6>, Tübingen, Germany: Max Planck Institute for Biological Cybernetics, 2004).

Danach können Sie die Abweichung vom Reconstruction Pre-Image berechnen:

```
>>> from sklearn.metrics import mean_squared_error  
>>> mean_squared_error(X, X_preimage)  
32.786308795766132
```

Nun kann die Gittersuche mit Kreuzvalidierung zum Finden des Kernels und der Hyperparameter zum Minimieren der Abweichung vom Reconstruction Pre-Image zum Einsatz kommen.

LLE

Locally Linear Embedding (<https://goo.gl/iA9bns>) (LLE)⁸ ist eine weitere sehr mächtige Methode zur *nichtlinearen Dimensionsreduktion* (NLDR). Sie ist eine Manifold-Learning-Technik, die im Gegensatz zu den oben vorgestellten Algorithmen nicht auf Projektionen beruht. Zusammengefasst bestimmt LLE die linearen Zusammenhänge jedes Datenpunkts zu seinen nächsten Nachbarn und sucht anschließend nach einer niedriger dimensionalen Repräsentation des Trainingsdatensatzes, bei der diese lokalen Beziehungen so gut wie möglich erhalten bleiben (Details in Kürze). Damit ist das Verfahren besonders gut zum Aufrollen verdrehter Manifolds geeignet, besonders wenn es nicht allzu viel Rauschen gibt.

Als Beispiel verwendet der folgende Code die Scikit-Learn-Klasse `LocallyLinearEmbedding`, um die Swiss-Roll-Daten aufzurichten. Der erhaltene 2-D-Datensatz ist in Abbildung 8-12 gezeigt.

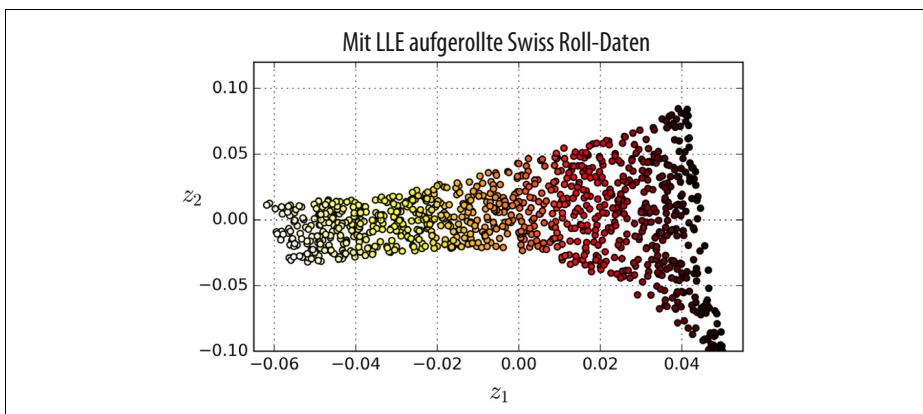


Abbildung 8-12: Mit LLE aufgerollte Swiss-Roll-Daten

Wie Sie sehen, wird die Swiss Roll vollständig aufgerollt, und die Abstände zwischen den Datenpunkten sind lokal gut erhalten. Allerdings bleiben die Abstände im größeren Maßstab nicht erhalten: Der linke Teil der aufgerollten Daten ist

8 »Nonlinear Dimensionsreduktion by Locally Linear Embedding«, S. Roweis, L. Saul (2000).

zusammengequetscht, der rechte Teil auseinandergezogen. Dennoch hat LLE beim Modellieren des Manifold gute Arbeit geleistet.

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

LLE funktioniert folgendermaßen: Zuerst werden die k -nächsten Nachbarn zu jedem Trainingsdatenpunkt $\mathbf{x}^{(i)}$ identifiziert (im obigen Code mit $k = 10$), anschließend wird $\mathbf{x}^{(i)}$ als lineare Funktion dieser Nachbarn rekonstruiert. Genauer gesagt, werden die Gewichte $w_{i,j}$ so gewählt, dass der quadratische Abstand zwischen $\mathbf{x}^{(i)}$ und $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$ so klein wie möglich ist, wobei $w_{i,j} = 0$ gilt, wenn $\mathbf{x}^{(j)}$ keiner der k -nächsten Nachbarn von $\mathbf{x}^{(i)}$ ist. Damit ist der erste Schritt beim LLE-Verfahren das in Formel 8-4 angegebene Optimierungsproblem, wobei \mathbf{W} eine Matrix aus sämtlichen Gewichten $w_{i,j}$ ist. Die zweite Nebenbedingung normalisiert die Gewichte für jeden Trainingsdatenpunkt $\mathbf{x}^{(i)}$.

Formel 8-4: LLE Schritt 1: Ein lineares Modell der lokalen Nachbarschaftsbeziehungen

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left\| \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right\|^2$$

unter der Bedingung $\begin{cases} w_{i,j} = 0 & \text{wenn } \mathbf{x}^{(j)} \text{ nicht zu den } k \text{ nächsten Nachbarn von } \mathbf{x}^{(i)} \text{ gehört} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{für } i = 1, 2, \dots, m \end{cases}$

Nach diesem Schritt codiert die Gewichtsmatrix $\hat{\mathbf{W}}$ (mit den Gewichten $\hat{w}_{i,j}$) die lokalen linearen Nachbarschaftsbeziehungen zwischen den Trainingsdatenpunkten. Im zweiten Schritt werden die Trainingsdatenpunkte in einen d -dimensionalen Raum (mit $d < n$) abgebildet, wobei die Nachbarschaftsbeziehungen so gut wie möglich beibehalten werden. Wenn $\mathbf{z}^{(i)}$ das Abbild von $\mathbf{x}^{(i)}$ in diesem d -dimensionalen Raum ist, möchten wir, dass der quadratische Abstand zwischen $\mathbf{z}^{(i)}$ und $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$ so klein wie möglich ist. Aus dieser Grundidee ergibt sich das in Formel 8-5 formulierte Optimierungsproblem. Es sieht dem ersten Schritt sehr ähnlich, aber anstatt die optimalen Gewichte bei festen Datenpunkten zu ermitteln, machen wir nun das Gegenteil: Die Gewichte stehen fest, und wir suchen die optimale Position für das Abbild eines Datenpunkts im niedriger dimensionalen Raum. Beachten Sie, dass \mathbf{Z} eine Matrix sämtlicher $\mathbf{z}^{(i)}$ ist.

Formel 8-5: LLE Schritt 2: Reduzieren der Dimensionalität unter Beibehaltung von Nachbarschaftsbeziehungen

$$\hat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left\| \mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)} \right\|^2$$

Die Scikit-Learn-Implementierung des LLE-Verfahrens besitzt folgende Komplexität der Berechnung: $O(m \log(m) n \log(k))$, um die k -nächsten Nachbarn zu finden,

$O(mnk^3)$, um die Gewichte zu optimieren, und $O(dm^2)$, um die niedriger dimensionale Abbildung zu berechnen. Leider sorgt das m^2 im letzten Term dafür, dass dieser Algorithmus schlecht auf sehr große Datensätze skaliert.

Weitere Techniken zur Dimensionsreduktion

Es gibt viele weitere Techniken zur Dimensionsreduktion. Einige davon sind in Scikit-Learn enthalten. Hier sind die beliebtesten:

- *Multidimensionale Skalierung* (MDS) reduziert die Dimensionen unter Beibehaltung der Abstände zwischen den Datenpunkten (siehe Abbildung 8-13).

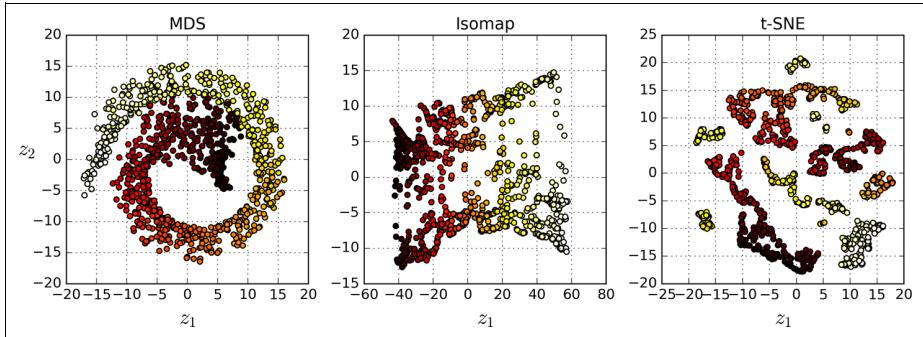


Abbildung 8-13: Reduzieren der Swiss-Roll-Daten zu einer 2-D-Darstellung mit unterschiedlichen Techniken

- *Isomap* erstellt einen Graphen, indem es jeden Datenpunkt mit seinen nächsten Nachbarn verbindet. Anschließend werden die Dimensionen unter Beibehaltung der geodätischen Distanz⁹ zwischen den Datenpunkten verringert.
- *t-verteiltes Stochastic Neighbor Embedding* (t-SNE) reduziert die Dimensionalität und versucht, ähnliche Datenpunkte nahe beieinander und unähnliche voneinander entfernt zu halten. Das Verfahren wird vor allem zur Visualisierung eingesetzt, insbesondere für Cluster von Datenpunkten in hochdimensionalen Räumen (z. B. um die MNIST-Bilder in 2-D darzustellen).
- *Lineare Diskriminantenanalyse* (LDA) ist eigentlich ein Klassifikationsverfahren, das aber beim Training die am stärksten zwischen den Kategorien unterscheidenden Achsen erlernt. Über diese Achsen lässt sich eine Hyperebene zur Projektion der Daten definieren. Der Vorteil dabei ist, dass die Projektion Kategorien soweit wie möglich voneinander entfernt hält. Damit ist LDA zur Dimensionsreduktion geeignet, bevor ein Klassifikationsalgorithmus wie ein SVM-Klassifikator ausgeführt wird.

⁹ Die geodätische Distanz zwischen zwei Knoten eines Graphen ist die Anzahl der Knoten auf dem kürzesten Pfad zwischen diesen Knoten.

Übungen

1. Welches sind die wichtigsten Gründe, die Dimensionen eines Datensatzes zu verringern? Welches sind die wichtigsten Nachteile?
2. Was ist der Fluch der Dimensionalität?
3. Ist die Dimensionalität eines Datensatzes erst einmal reduziert, ist es möglich, die Operation umzukehren? Falls ja, wie? Falls nein, warum nicht?
4. Lässt sich die PCA einsetzen, um die Dimensionen eines hochgradig nichtlinearen Datensatzes zu verringern?
5. Sie führen eine PCA auf einem 1000-dimensionalen Datensatz durch und legen den Anteil der erklärten Streuung auf 95 % fest. Wie viele Dimensionen hat der sich daraus ergebende Datensatz?
6. In welchen Fällen würden Sie eine reine Hauptkomponentenzerlegung durchführen, wann eine inkrementelle PCA, eine randomisierte PCA oder eine Kernel PCA?
7. Wie können Sie die Leistung eines Algorithmus zur Dimensionsreduktion auf Ihrem Datensatz bestimmen?
8. Ist es sinnvoll, zwei unterschiedliche Algorithmen zur Dimensionsreduktion hintereinanderzuschalten?
9. Laden Sie den MNIST-Datensatz (aus Kapitel 3) und teilen Sie diesen in einen Trainingsdatensatz und einen Testdatensatz auf (verwenden Sie die ersten 60000 Datenpunkte zum Trainieren und die übrigen 10000 zum Testen). Trainieren Sie einen Random-Forest-Klassifikator auf dem Datensatz und messen dessen Laufzeit. Evaluieren Sie das entstandene Modell anhand der Testdaten. Führen Sie anschließend eine Hauptkomponentenzerlegung mit einem Anteil erklärter Streuung von 95 % durch, um die Dimensionalität des Datensatzes zu verringern. Trainieren Sie einen neuen Random-Forest-Klassifikator auf dem reduzierten Datensatz und messen Sie wiederum die Laufzeit. War das Trainieren viel schneller? Evaluieren Sie auch diesen Klassifikator auf den Testdaten: Wie schneidet er im Vergleich zum ersten Klassifikator ab?
10. Verwenden Sie t-SNE, um den MNIST-Datensatz auf zwei Dimensionen zu reduzieren, und visualisieren Sie das Ergebnis mit Matplotlib. Sie können einen Scatterplot mit zehn unterschiedlichen Farben verwenden, um die Zielkategorien jedes Bilds darzustellen. Alternativ können Sie farbige Ziffern an die Orte jedes Datenpunkts schreiben oder sogar verkleinerte Versionen der Ziffernbilder selbst (wenn Sie alle Ziffern darstellen, wird das Diagramm voll und unübersichtlich sein. Sie sollten also eine zufällige Stichprobe ziehen oder einen Datenpunkt nur dann zeichnen, falls noch kein anderer Punkt in der Umgebung gezeichnet wurde). Verwenden Sie andere Algorithmen zur Dimensionsreduktion wie die Hauptkomponentenzerlegung, LLE oder MDS und vergleichen Sie die entstehenden Diagramme.

Lösungen zu diesen Übungen finden Sie in Anhang A.

TEIL II

Neuronale Netze und Deep Learning

Einsatzbereit mit TensorFlow

TensorFlow ist eine mächtige Open-Source-Bibliothek für numerische Berechnungen, die sich besonders für Machine Learning in großem Stil eignet. Ihr Grundprinzip ist einfach: Sie definieren einen Graphen von Berechnungen in Python (beispielsweise den in Abbildung 9-1), anschließend führt TensorFlow diesen Graphen mithilfe von optimiertem C++-Code aus.

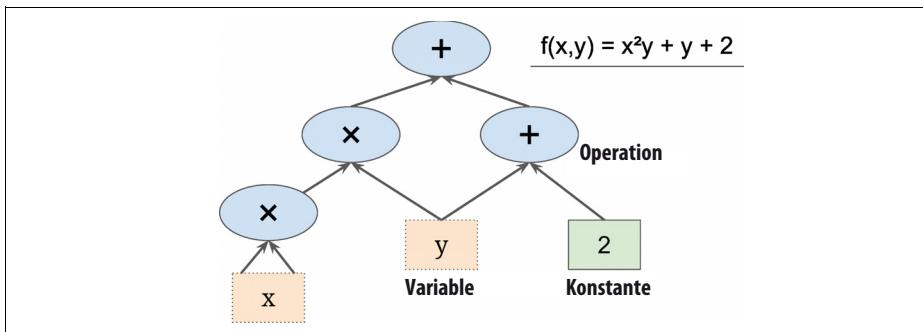


Abbildung 9-1: Ein einfacher Berechnungsgraph

Vor allem aber lässt sich ein Graph in mehrere Abschnitte aufteilen, die auf mehreren CPUs oder GPUs ausgeführt werden können (wie in Abbildung 9-2 dargestellt). TensorFlow unterstützt auch verteiltes Rechnen, sodass Sie gigantische neuronale Netze auf riesigen Trainingsdatensätzen trainieren können, indem Sie die Berechnungen auf Hunderte Server verteilen (siehe Kapitel 12). TensorFlow ist in der Lage, Netze mit Millionen Parametern auf einem Trainingsdatensatz mit Milliarden Datenpunkten und Millionen Merkmalen zu trainieren. Dies ist nicht überraschend, da TensorFlow vom Team hinter Google Brain entwickelt wurde und viele der großen Google-Dienste wie Google Cloud Speech, Google Photos und Google Search mit TensorFlow betrieben werden.

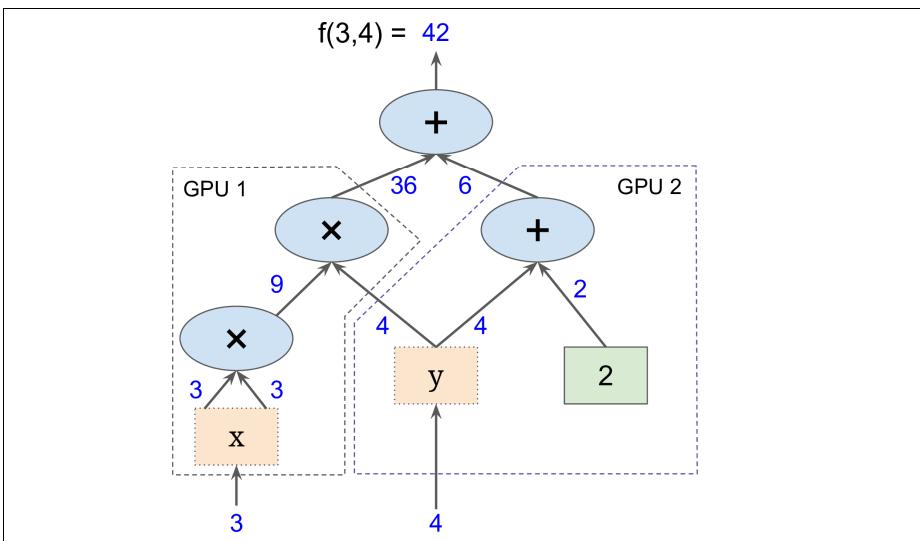


Abbildung 9-2: Parallele Berechnung auf mehreren CPUs/GPUs/Servern

Als TensorFlow im November 2015 als Open-Source-Software veröffentlicht wurde, gab es bereits zahlreiche Open-Source-Bibliotheken für Deep Learning (einige sind in Tabelle 9-1 aufgezählt). Fairerweise muss man sagen, dass es die meisten Features von TensorFlow bereits in der einen oder anderen Bibliothek gab. Dennoch haben es das klare Design, die Skalierbarkeit, die Flexibilität¹ und die gute Dokumentation von TensorFlow (und nicht zuletzt der Name Google) schnell auf die oberen Tabellenplätze gebracht. Kurz, TensorFlow wurde als flexibles, skalierbares und sofort einsatzbereites Framework entworfen, wobei die bereits existierenden Frameworks nur zwei dieser drei Punkte abdeckten. Hier sind einige Highlights von TensorFlow:

- Es läuft nicht nur auf Windows, Linux und macOS, sondern auch auf mobilen Endgeräten, darunter sowohl iOS und Android.
- Es stellt eine sehr einfache Python-API namens *TF.Learn* zur Verfügung² (`tensorflow.contrib.learn`), die zu Scikit-Learn kompatibel ist. Wie Sie sehen werden, können Sie damit verschiedene Arten neuronaler Netze mit wenigen Codezeilen trainieren. Dies war ursprünglich ein unabhängiges Projekt mit dem Titel *Scikit Flow* (oder *skflow*).
- Es stellt eine weitere einfache API namens *TF-slim* (`tensorflow.contrib.slim`) bereit, um Aufbau, Training und Evaluierung neuronaler Netze zu vereinfachen.

¹ TensorFlow ist nicht auf neuronale Netze oder Machine Learning beschränkt; Sie können quantenmechanische Simulationen darauf laufen lassen.

² Diese ist nicht mit der Bibliothek *TFLearn* zu verwechseln, einem unabhängigen Projekt.

- Weitere unabhängig entwickelte APIs wie *Keras* (<http://keras.io>) (inzwischen als `tensorflow.contrib.keras` verfügbar) oder *Pretty Tensor* (<https://github.com/google/prettytensor/>) bauen direkt auf TensorFlow auf.
- Die eigentliche Python-Schnittstelle ist flexibel genug (zum Preis höherer Komplexität), um alle erdenklichen Berechnungen durchzuführen, darunter jede beliebige Architektur neuronaler Netze.
- Viele ML-Vorgänge sind hochgradig effizient in C++ implementiert, besonders die zum Aufbau neuronaler Netze. Es gibt auch eine C++-Schnittstelle zum Definieren eigener leistungsfähiger Operationen.
- Einige fortgeschrittene Optimierungsknoten erlauben die Suche nach Parametern zum Minimieren einer Kostenfunktion. Diese Knoten sind sehr einfach einzusetzen, da TensorFlow die Gradienten der von Ihnen definierten Funktionen automatisch berechnet, was als *automatische Differenzierung* (oder *Autodiff*) bezeichnet wird.
- Es enthält das großartige Visualisierungstool *TensorBoard*, mit dem Sie den Berechnungsgraphen untersuchen, Lernkurven betrachten und vieles mehr tun können.
- Google hat einen Cloud-Dienst gestartet, um TensorFlow-Graphen auszuführen (<https://cloud.google.com/ml>).
- Schließlich gibt es ein passioniertes und hilfreiches Entwicklerteam und eine wachsende Community, die an Verbesserungen arbeitet. Es gehört zu den beliebtesten Open-Source-Projekten auf GitHub, und es werden ständig weitere Projekte mit TensorFlow entwickelt (schauen Sie sich beispielsweise die Ressourcen-Seiten auf <https://www.tensorflow.org/> oder <https://github.com/jtoy/awesome-tensorflow> an). Um technische Fragen zu stellen, sollten Sie <http://stackoverflow.com/> verwenden und Ihre Frage mit "tensorflow" taggen. Sie können Bugs und Wünsche nach Features über GitHub äußern. Die allgemeine Diskussion zum Thema findet in einer Google-Gruppe (<http://goo.gl/N7kRF9>) statt.

In diesem Kapitel werden wir die Grundlagen von TensorFlow bearbeiten. Dazu werden wir die Installation durchgehen und einfache Berechnungsgraphen erstellen, ausführen, speichern und visualisieren. Es ist wichtig, diese Grundlagen zu meistern, bevor Sie Ihr erstes neuronales Netz bauen (was wir uns im nächsten Kapitel vornehmen).

Tabelle 9-1: Open-Source-Bibliotheken für Deep Learning (unvollständige Liste)

Bibliothek	API	Plattformen	gestartet von	Jahr
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	A. Gibson, J. Patterson	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014

Tabelle 9-1: Open-Source-Bibliotheken für Deep Learning (unvollständige Liste) (Fortsetzung)

Bibliothek	API	Plattformen	gestartet von	Jahr
MXNet	Python, C++, andere	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	University of Montreal	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

Installation

Fangen wir an! Wir gehen davon aus, dass Sie Jupyter und Scikit-Learn bereits entsprechend der Anleitung in Kapitel 2 installiert haben. Zum Installieren von TensorFlow genügt pip. Wenn Sie mit virtualenv eine isolierte Umgebung erstellt haben, sollten Sie diese jetzt aktivieren:

```
$ cd $ML_PATH          # Ihr ML-Arbeitsverzeichnis (z.B. $HOME/ml)
$ source env/bin/activate
```

Als Nächstes installieren Sie TensorFlow:

```
$ pip3 install --upgrade tensorflow
```



Für die Unterstützung von GPUs benötigen Sie `tensorflow-gpu` anstelle von `tensorflow`. Details finden Sie in Kapitel 12.

Um Ihre Installation zu testen, geben Sie den folgenden Befehl ein. Er gibt Ihnen die installierte TensorFlow-Version aus.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)'
1.0.0
```

Erstellen und Ausführen eines ersten Graphen

Der folgende Code erstellt den in Abbildung 9-1 dargestellten Graphen:

```
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

Das ist alles! Für das Verständnis ist es hierbei wichtig zu verstehen, dass dieser Code überhaupt keine Berechnung ausführt, auch wenn es danach aussieht (besonders in der letzten Zeile). Es wird lediglich ein Berechnungsgraph erzeugt. Genau

genommen werden noch nicht einmal die Variablen initialisiert. Um diesen Graphen zu evaluieren, müssen Sie eine *Session* in TensorFlow öffnen und mit dieser die Variablen initialisieren und *f* evaluieren. Eine TensorFlow-Session kümmert sich darum, die Operationen auf *Geräte* wie CPUs und GPUs zu verteilen und diese auszuführen. Sie enthält auch sämtliche Werte von Variablen.³ Der folgende Code erstellt eine Session, initialisiert die Variablen, evaluiert *f* und schließt dann die Session (zur Freigabe von Ressourcen):

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Es ist etwas mühselig, ständig *sess.run()* zu schreiben, es gibt aber glücklicherweise eine bessere Alternative:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Innerhalb des *with*-Blocks ist die Session automatisch als Standard eingestellt. Der Aufruf *x.initializer.run()* entspricht *tf.get_default_session().run(x.initializer)*, und in ähnlicher Weise ist *f.eval()* äquivalent zum Aufruf *tf.get_default_session().run(f)*. Der Code ist dadurch einfacher zu lesen. Außerdem wird die Session am Ende des Codeabschnitts automatisch geschlossen.

Anstatt die Initialisierung jeder einzelnen Variablen explizit auszuführen, können Sie auch die Funktion *global_variables_initializer()* verwenden. Beachten Sie, dass diese Funktion die Initialisierung nicht sofort vornimmt, sondern einen Knoten im Graphen erstellt, der beim Ausführen sämtliche Variablen initialisiert:

```
init = tf.global_variables_initializer() # erstelle Knoten zum Initialisieren

with tf.Session() as sess:
    init.run() # initialisiert die Variablen
    result = f.eval()
```

Innerhalb von Jupyter oder einer Python-Shell sollten Sie lieber eine *InteractiveSession* erstellen. Der einzige Unterschied zur gewöhnlichen Session ist, dass eine *InteractiveSession* beim Erstellen automatisch als Standardsession gesetzt wird. Damit benötigen Sie den *with*-Block nicht (Sie müssen aber die Session von Hand schließen, sobald Sie fertig sind):

```
>>> sess = tf.InteractiveSession()
>>> init.run()
```

³ Beim Einsatz von TensorFlow für verteiltes Rechnen werden die Variablen auf den Servern anstatt in der Session gespeichert, wie wir in Kapitel 12 sehen werden.

```
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()
```

Ein TensorFlow-Programm besteht normalerweise aus zwei Teilen: Im ersten Teil wird ein Berechnungsgraph erstellt (dies bezeichnet man als *Konstruktionsphase*), und im zweiten Teil wird dieser ausgeführt (dies bezeichnet man als *Ausführungsphase*). In der Konstruktionsphase wird der Berechnungsgraph mit dem ML-Modell und den zum Trainieren nötigen Berechnungen ausgestattet. In der Ausführungsphase wird für gewöhnlich ein Trainingsschritt über eine Schleife wiederholt (beispielsweise ein Schritt pro Mini-Batch), sodass sich die Modellparameter schrittweise verbessern. Wir werden hierfür in Kürze ein Beispiel kennenlernen.

Graphen verwalten

Jeder erstellte Knoten wird automatisch dem Standardgraphen hinzugefügt:

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

In den meisten Fällen reicht dies aus. Manchmal möchte man aber mehrere Graphen unabhängig bearbeiten. Dazu können Sie ein neues Graph-Objekt erstellen und dieses innerhalb eines with-Blocks temporär zum Standardgraphen machen:

```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```



In Jupyter (oder einer Python-Shell) ist es üblich, beim Herumexperimentieren die gleichen Befehle mehrfach auszuführen. Als Folge davon kann der Standardgraph viele Knoten doppelt enthalten. Eine Lösung wäre, den Jupyter-Kernel (oder die Python-Shell) erneut zu starten, bequemer ist es jedoch, den Standardgraphen mit der Funktion `tf.reset_default_graph()` zurückzusetzen.

Lebenszyklus des Werts von Knoten

Wenn Sie einen Knoten evaluieren, ermittelt TensorFlow automatisch die Menge der Knoten, von denen dieser Knoten abhängig ist, und evaluiert zunächst diese Knoten. Betrachten Sie als Beispiel den folgenden Code:

```
w = tf.constant(3)
x = w + 2
```

```

y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval()) # 10
    print(z.eval()) # 15

```

Zuerst definiert der Code einen sehr einfachen Graphen. Dann wird eine Session gestartet und der Graph zum Evaluieren von y ausgeführt: TensorFlow findet automatisch heraus, dass y von x abhängt, das wiederum von w abhängt. Daher wird zuerst w ausgewertet, dann x , anschließend y , und zuletzt wird der Wert von y ausgegeben. Als letzten Befehl wird der Graph bei z ausgewertet. Wieder findet TensorFlow heraus, dass w und x zuerst ausgewertet werden müssen. Es ist an dieser Stelle wichtig, hervorzuheben, dass die Ergebnisse der vorigen Evaluierung von w und x *nicht* wiederverwendet werden. Kurz, w und x werden im obigen Code zweimal ausgewertet.

Die Werte von Knoten werden zwischen den einzelnen Auswertungen eines Graphen verworfen, nur die Werte von Variablen bleiben über die gesamte Session erhalten (Queues und Readers enthalten ebenfalls einen dauerhaften Zustand, wie wir in Kapitel 12 sehen werden). Eine Variable beginnt ihren Lebenszyklus, sobald ihr Initializer ausgeführt wird, und beendet diesen, sobald die Session geschlossen wird.

Wenn Sie y und z effizient auswerten möchten, ohne w und x wie im obigen Beispiel doppelt zu evaluieren, müssen Sie TensorFlow bitten, sowohl y als auch z in einem Ausführungsschritt zu berechnen. Das demonstriert der folgende Code:

```

with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
    print(y_val) # 10
    print(z_val) # 15

```



Wird TensorFlow in einem Prozess ausgeführt, wird kein Zustand von einer Session zur nächsten übertragen, selbst wenn diese den gleichen Graphen verwenden (jede Session würde eine eigene Kopie jeder Variablen enthalten). Im verteilten Betrieb von TensorFlow (siehe Kapitel 12) wird der Zustand von Variablen auf den Servern anstatt in den Sessions gespeichert. Somit können dort mehrere Sessions auf die gleichen Variablen zugreifen.

Lineare Regression mit TensorFlow

Operationen in TensorFlow (auch kurz als *Ops* bezeichnet) können eine beliebige Anzahl Eingabewerte annehmen und eine beliebige Anzahl Ausgabewerte produzieren. Beispielsweise nehmen die Ops zum Addieren und Multiplizieren jeweils zwei Eingabewerte an und produzieren einen Ausgabewert. Konstanten und Variablen nehmen keine Eingabewerte an (diese werden als *Source Ops* bezeichnet).

Ein- und Ausgaben sind mehrdimensionale Arrays oder *Tensoren* (daher der Name »TensorFlow«). Wie Arrays in NumPy haben Tensoren einen Typ und ein shape-Attribut. In der Python-API werden Tensoren tatsächlich einfach als ndarrays aus NumPy gespeichert. Sie enthalten normalerweise floats, können aber auch zum Ablegen von Strings (oder beliebigen Bytefolgen) verwendet werden.

In den bisherigen Beispielen enthielten die Tensoren lediglich einen einzelnen Skalar, aber Sie können natürlich Berechnungen auf Arrays beliebiger Form ausführen. Beispielsweise führt der folgende Code eine lineare Regression auf 2-D-Arrays mit dem kalifornischen Immobiliendatensatz (aus Kapitel 2) durch. Zunächst beschafft er sich den Datensatz; anschließend fügt er allen Trainingsdatenpunkten ein zusätzliches Eingabemerkmals als Bias hinzu ($x_0 = 1$, dies wird unmittelbar mit NumPy durchgeführt); anschließend werden zwei konstante Knoten X und y in TensorFlow erzeugt, um diese Daten und die Zielgröße aufzunehmen⁴. Zur Definition von theta werden einige von TensorFlow bereitgestellte Matrizenoperationen verwendet. Diese Matrizenoperationen – `transpose()`, `matmul()` und `matrix_inverse()` – sind selbst erklärend, sie führen aber wie gewöhnlich keine unmittelbaren Berechnungen aus; stattdessen erzeugen sie Knoten im Graphen, und beim Ausführen des Graphen werden die Berechnungen ausgeführt. Sie erkennen womöglich, dass die Definition von theta mit der Normalengleichung übereinstimmt ($\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$; siehe Kapitel 4). Abschließend erzeugt der Code eine Session und verwendet diese zur Auswertung von theta.

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

Der Hauptvorteil dieses Codes im Gegensatz zur direkten Berechnung der Normalengleichung mit NumPy ist, dass TensorFlow ihn direkt auf Ihrer GPU ausführen kann (vorausgesetzt, Sie besitzen eine und haben TensorFlow mit GPU-Unterstützung installiert; Details dazu finden Sie in Kapitel 12).

⁴ Beachten Sie, dass `housing.target` ein 1-D-Array ist, wir es aber für die Berechnung von `theta` in einen Spaltenvektor überführen müssen. Die NumPy-Funktion `reshape()` akzeptiert für eine der Dimensionen `-1` (was für »nicht spezifiziert« steht): Diese Dimension wird anhand der Länge des Arrays und der verbleibenden Dimensionen berechnet.

Implementieren des Gradientenverfahrens

Versuchen wir nun, anstelle der Normalengleichung das Batch-Gradientenverfahren (aus Kapitel 4) zu verwenden. Dazu werden wir zunächst die Gradienten von Hand ausrechnen und anschließend das Autodiff-Feature von TensorFlow nutzen, mit dem TensorFlow die Gradienten automatisch berechnet. Als Letztes werden wir einige der in TensorFlow eingebauten Optimierungsverfahren verwenden.



Beim Verwenden des Gradientenverfahrens ist es wichtig, die Merkmalsvektoren mit den Eingabedaten zunächst zu normalisieren, andernfalls wird das Training deutlich langsamer. Sie können dies mit TensorFlow, NumPy, dem StandardScaler aus Scikit-Learn oder einer beliebigen anderen Lösung tun. Der folgende Code geht davon aus, dass die Normalisierung bereits vorgenommen wurde.

Manuelle Berechnung der Gradienten

Der folgende Code sollte bis auf einige neue Elemente weitgehend selbsterklärend sein:

- Die Funktion `random_uniform()` erstellt einen Knoten im Graphen, der einen Tensor mit Zufallswerten mit den gegebenen Abmessungen und dem gegebenen Wertebereich erzeugt, in etwa wie die NumPy-Funktion `rand()`.
- Die Funktion `assign()` erstellt einen Knoten, der einer Variablen einen neuen Wert zuweist. In diesem Fall wird damit folgender Schritt des Batch-Gradientenverfahrens umgesetzt: $\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$.
- Die Hauptschleife wiederholt den Trainingsschritt (`n_epochs` Mal) und gibt alle 100 Iterationen die aktuelle mittlere quadratische Abweichung (`mse`) aus. Sie sollten beobachten, dass die MSE bei jeder Iteration sinkt.

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
```

```

if epoch % 100 == 0:
    print("Epoch", epoch, "MSE =", mse.eval())
    sess.run(training_op)

best_theta = theta.eval()

```

Verwenden von Autodiff

Der obige Code funktioniert ausgezeichnet, dazu müssen jedoch die Gradienten mathematisch aus der Kostenfunktion (MSE) abgeleitet werden. Bei der linearen Regression ist dies noch vergleichsweise einfach, aber wenn Sie das bei einem komplexen neuronalen Netz versuchen, würde dies einige Kopfschmerzen verursachen: Es wäre sowohl mühselig als auch fehleranfällig. Sie könnten die *symbolische Differenzierung* verwenden, um die Gleichungen für die partiellen Ableitungen automatisch zu finden, aber der dabei entstehende Code wäre wahrscheinlich nicht besonders effizient.

Um dies besser nachzuvollziehen, betrachten wir die Funktion $f(x) = \exp(\exp(\exp(x)))$. Wenn Sie sich mit Analysis auskennen, können Sie die Ableitung $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$ bilden. Wenn Sie $f(x)$ und $f'(x)$ separat und exakt ihrer Schreibweise entsprechend implementieren, wird Ihr Code nicht maximal effizient arbeiten. Eine effizientere Lösung wäre, eine Funktion zu schreiben, die zuerst $\exp(x)$ berechnet, anschließend $\exp(\exp(x))$, dann $\exp(\exp(\exp(x)))$ und am Ende alle drei Ergebnisse zurückgibt. Damit erhalten Sie $f(x)$ direkt (den dritten Term). Wenn Sie dann die Ableitung benötigen, könnten Sie einfach alle drei Terme miteinander multiplizieren und sind fertig. Beim naiven Ansatz müssten Sie die Funktion \exp insgesamt neunmal aufrufen, um sowohl $f(x)$ als auch $f'(x)$ zu berechnen. Mit diesem Ansatz müssen Sie sie nur noch dreimal aufrufen.

Wenn Ihre Funktion beliebigen Code enthält, wird es sogar noch unangenehmer. Können Sie die Gleichung (oder den Code) finden, mit der sich die partiellen Ableitungen der folgenden Funktion finden lassen? Unser Tipp: Probieren Sie es besser nicht.

```

def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z

```

Glücklicherweise kommt an dieser Stelle das Autodiff-Feature von TensorFlow ins Spiel: Es berechnet Ihnen die Gradienten automatisch und effizient. Sie müssen nur die Zeile `gradients = ...` im obigen Code für das Gradientenverfahren mit der folgenden Zeile ersetzen, und der Code funktioniert weiter:

```
gradients = tf.gradients(mse, [theta])[0]
```

Die Funktion `gradients()` akzeptiert eine Operation (in diesem Falle `mse`) und eine Liste von Variablen (in diesem Falle nur `theta`) und erstellt eine Liste von Operati-

onen (eine pro Variable) zum Berechnen der Ableitungen der Operation nach jeder der Variablen. Der Knoten gradients berechnet daher den Gradientenvektor der MSE nach theta.

Es gibt vier wichtige Ansätze zum automatischen Berechnen von Ableitungen. Diese sind in Tabelle 9-2 zusammengefasst. TensorFlow verwendet *Autodiff im Reverse-Modus*, was bei vielen Eingabewerten und wenigen Ausgabewerten perfekt (effizient und genau) geeignet ist. Bei den meisten neuronalen Netzen ist dies der Fall. Das Verfahren berechnet sämtliche partiellen Ableitungen der Ausgabe nach allen Eingabegröße innerhalb von nur $n_{\text{Ausgaben}} + 1$ Durchläufen durch den Graphen.

Tabelle 9-2: Wichtigste Verfahren zum automatischen Berechnen von Gradienten

Technik	Anzahl Durchläufe durch den Graphen zur Berechnung aller Gradienten	Genauigkeit	unterstützt beliebigen Code	Kommentar
Numerische Differenzierung	$n_{\text{Eingaben}} + 1$	niedrig	ja	trivial zu implementieren
Symbolische Differenzierung	N/A	hoch	nein	erstellt einen völlig unterschiedlichen Graphen
Autodiff im Forward-Modus	n_{Eingaben}	hoch	ja	verwendet <i>duale Zahlen</i>
Autodiff im Reverse-Modus	$n_{\text{Ausgaben}} + 1$	hoch	ja	in TensorFlow implementiert

Wenn Sie sich dafür interessieren, wie dieses magische Verfahren funktioniert, lesen Sie Anhang D.

Verwenden von Optimierungsverfahren

TensorFlow berechnet also die Ableitungen für Sie. Es wird aber noch einfacher: Es gibt bereits einige vorimplementierte Optimierungsverfahren, darunter das Gradientenverfahren. Sie können im obigen Code die Zeilen `gradients = ...` und `training_op = ...` einfach mit dem folgenden Code ersetzen:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

Wenn Sie ein anderes Optimierungsverfahren verwenden möchten, müssen Sie nur eine Zeile ändern. Sie können beispielsweise mit folgender Einstellung einen Momentum Optimizer verwenden (der oft viel schneller als das Gradientenverfahren konvergiert; siehe Kapitel 11):

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

Daten in den Trainingsalgorithmus einspeisen

Modifizieren wir nun den Code, um das Mini-Batch-Gradientenverfahren zu implementieren. Dazu müssen wir irgendwie X und y in jeder Iteration mit dem jeweils nächsten Mini-Batch ersetzen. Die einfachste Möglichkeit hierzu bieten Platzhalter-Knoten. Diese Knoten sind etwas Besonderes, weil sie überhaupt keine Berechnung durchführen, sondern beim Ausführen lediglich die gewünschten Daten ausgeben. Man verwendet sie normalerweise dazu, die Trainingsdaten während des Trainings an TensorFlow zu übergeben. Wenn Sie keinen Wert für einen Platzhalter angeben, erhalten Sie beim Ausführen einen Ausnahmefehler.

Platzhalter-Knoten lassen sich mit der Funktion `placeholder()` erstellen, wobei Sie den Datentyp des ausgegebenen Tensors angeben müssen. Sie können wahlweise auch die Abmessungen des Tensors angeben, falls diese obligatorisch sein sollen. Wenn Sie eine Dimension mit `None` angeben, steht dies für »beliebige Größe.« Beispielsweise erstellt der folgende Code einen Platzhalter-Knoten `A` sowie den Knoten `B = A + 5`. Beim Auswerten von `B` übergeben wir der Methode `eval()` den Wert `feed_dict`, mit dem `A` festgelegt wird. Beachten Sie, dass `A` zweidimensional sein muss und es drei Spalten geben sollte (andernfalls wird ein Ausnahmefehler erzeugt). Die Anzahl Zeilen ist jedoch beliebig.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
 [12. 13. 14.]]
```



Sie können die Ausgabe *beliebiger* Operationen als Eingabe verwenden, nicht nur Platzhalter. Im Falle von Platzhaltern versucht TensorFlow aber nicht, die Operation auszuwerten; es werden einfach die übergebenen Werte verwendet.

Um das Mini-Batch-Gradientenverfahren zu implementieren, müssen wir den bestehenden Code nur wenige verändern. Zuerst ändern wir die Definition von X und y in der Konstruktionsphase zu Platzhalter-Knoten:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Anschließend definieren wir die Größe der Batches und berechnen die Gesamtzahl der Batches:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

In der Ausführungsphase lesen wir schließlich die Mini-Batches einen nach dem anderen ein und geben die Werte von X und y über den Parameter feed_dict an, wobei das Auswerten eines Knotens von beiden abhängt.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # lade die Daten von der Festplatte
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
```



Wir müssen die Werte für X und y beim Auswerten von theta nicht übergeben, da der Knoten von keinem der beiden abhängt.

Modelle speichern und wiederherstellen

Haben Sie Ihr Modell erst einmal trainiert, sollten Sie dessen Parameter auf der Festplatte sichern, sodass Sie es zu einem beliebigen späteren Zeitpunkt laden, in einem anderen Programm verwenden oder mit anderen Modellen vergleichen können. Beim Training sollten Sie außerdem in regelmäßigen Abständen einen Zwischenstand speichern, sodass Sie bei einem Computerabsturz das Training einfach beim zuletzt gespeicherten Zwischenstand fortsetzen können.

TensorFlow erleichtert das Speichern und Wiederherstellen von Modellen erheblich. Erstellen Sie einfach am Ende der Konstruktionsphase einen Saver-Knoten (nach Erstellen aller Knoten mit Variablen); in der Ausführungsphase rufen Sie einfach dessen Methode save() auf, wann immer Sie das Modell sichern möchten, und übergeben dieser die Session und den Pfad zur Sicherungsdatei:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # Sicherungskopie alle 100 Iterationen
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
```

```

sess.run(training_op)

best_theta = theta.eval()
save_path = saver.save(sess, "/tmp/my_model_final.ckpt")

```

Das Wiederherstellen eines Modells ist ebenso einfach: Wie zuvor erstellen Sie am Ende der Konstruktionsphase einen Saver, zu Beginn der Ausführungsphase müssen Sie die Methode `restore()` des Saver-Objekts aufrufen, anstatt die Variablen zu initialisieren:

```

with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]

```

Standardmäßig speichert und lädt ein Saver sämtliche Variablen mit ihren jeweiligen Namen, Sie können aber auch genau angeben, welche Variablen gespeichert oder geladen werden sollen und welche Namen für sie zu verwenden sind. Im folgenden Codebeispiel speichert der Saver lediglich die Variable `theta` unter dem Namen `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

Die Methode `save()` speichert standardmäßig auch die Struktur des Graphen in einer zweiten Datei mit dem gleichen Namen und der Endung `.meta` ab. Sie können diese Graphenstruktur mit der Funktion `tf.train.import_meta_graph()` laden. Damit wird der Graph dem Standardgraphen hinzugefügt, und Sie erhalten ein Saver-Objekt, mit dem Sie anschließend denn Zustand des Graphen (d.h. die Werte der Variablen) wiederherstellen können:

```

saver = tf.train.import_meta_graph("/tmp/my_model_final.ckpt.meta")

with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]

```

Dies stellt das gespeicherte Modell vollständig wieder her, inklusive der Struktur des Graphen und der Werte von Variablen. So müssen Sie nicht den Code zum Erstellen des Graphen suchen.

Graphen und Lernkurven mit TensorBoard visualisieren

Wir haben nun einen Berechnungsgraphen, der ein lineares Regressionsmodell mithilfe des Mini-Batch-Gradientenverfahrens erstellt. Wir speichern in regelmäßigen Abständen einen Zwischenstand. Dies klingt anspruchsvoll, oder? Allerdings verwenden wir noch immer die Funktion `print()`, um den Fortschritt beim Trainieren darzustellen. Es gibt eine bessere Möglichkeit: TensorBoard. Wenn Sie diesem einige Trainingsdatensätze übergeben, erhalten Sie eine ansprechende interaktive Visualisierung der Modelleigenschaften in Ihrem Browser (z.B. Lernkurven). Sie

können auch die Definition des Graphen übergeben und erhalten eine Oberfläche, um diesen zu untersuchen. Damit können Sie besser nach Fehlern im Graphen suchen, Engpässe finden und so weiter.

Als ersten Schritt passen wir Ihr Programm ein wenig an, sodass es die Definition des Graphen und einige Daten zum Training in ein Log-Verzeichnis schreibt – beispielsweise die Abweichung beim Training (MSE) – TensorBoard liest die Daten aus diesem Verzeichnis. Sie benötigen bei jedem Programmdurchlauf ein anderes Log-Verzeichnis, sonst verschmilzt TensorBoard Angaben aus unterschiedlichen Durchläufen und bringt die komplette Visualisierung durcheinander. Die einfachste Lösung ist, den Namen des Log-Verzeichnisses mit einem Zeitstempel zu versehen. Der folgende Code am Anfang Ihres Programms kümmert sich darum:

```
from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}{}/run-{}".format(root_logdir, now)
```

Als Nächstes fügen Sie den folgenden Code am Ende der Konstruktionsphase hinzu:

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

Die erste Zeile erstellt einen Knoten im Graphen, der den MSE-Wert ermittelt und ihn in einen TensorBoard-kompatiblen Binärstring namens *summary* schreibt. Die zweite Zeile erstellt einen `FileWriter`, mit dem Sie die Zusammenfassungen in Log-Dateien im Log-Verzeichnis schreiben können. Der erste Parameter gibt den Pfad des Log-Verzeichnisses an (in diesem Falle ein relatives Verzeichnis wie `tf_logs/run-20160906091959/`). Der zweite (optionale) Parameter ist der zu visualisierende Graph. Beim Instanziieren erzeugt der `FileWriter` das Log-Verzeichnis, falls es noch nicht existiert (und falls nötig auch übergeordnete Verzeichnisse), und schreibt die Definition des Graphen in eine binäre Logdatei, die man als *events-Datei* bezeichnet.

Als Nächstes sollten Sie die Ausführungsphase anpassen, sodass der Knoten `mse_summary` beim Trainieren regelmäßig abgearbeitet wird (z.B. alle zehn Mini-Batches). Dies generiert eine Zusammenfassung, die Sie mit dem `file_writer` in die events-Datei schreiben können. Hier folgt der aktualisierte Code:

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```



Vermeiden Sie das Schreiben von Informationen über das Trainieren bei jedem einzelnen Schritt, da das Trainieren dadurch deutlich langsamer würde.

Schließlich sollten Sie am Ende des Programms den `FileWriter` schließen:

```
file_writer.close()
```

Führen Sie dieses Programm nun aus: Es erstellt das Log-Verzeichnis und schreibt eine events-Datei mit der Definition des Graphen und den MSE-Werten in dieses Verzeichnis. Öffnen Sie eine Kommandozeile und wechseln Sie in Ihr Arbeitsverzeichnis. Schreiben Sie anschließend `ls -l tf_logs/run*`, um den Inhalt des Log-Verzeichnisses auszugeben:

```
$ cd $ML_PATH          # Ihr ML-Arbeitsverzeichnis (z.B. $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep  6 11:10 events.out.tfevents.1472553182.mymac
```

Wenn Sie das Programm ein zweites Mal ausführen, sollten Sie ein zweites Unter-Verzeichnis im Verzeichnis `tf_logs/` sehen:

```
$ ls -l tf_logs/
total 0
drwxr-xr-x 3 ageron staff 102 Sep  6 10:07 run-20160906091959
drwxr-xr-x 3 ageron staff 102 Sep  6 10:22 run-20160906092202
```

Großartig! Nun ist es an der Zeit, den TensorBoard-Server zu starten. Sie müssen dazu Ihre virtualenv-Umgebung starten, falls Sie eine erstellt haben. Anschließend starten Sie den Server mit dem Befehl `tensorboard` und übergeben das übergeordnete Log-Verzeichnis. Damit starten Sie den TensorBoard-Webserver auf Port 6006 (was sich kopfüber als »goog« lesen lässt):

```
$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard on port 6006
(You can navigate to http://0.0.0.0:6006)
```

Öffnen Sie nun einen Browser und gehen Sie zur Adresse `http://0.0.0.0:6006/` (oder `http://localhost:6006/`). Willkommen bei TensorBoard! Auf der Events-Karte sollten Sie auf der rechten Seite MSE sehen. Wenn Sie darauf klicken, sehen Sie ein Diagramm mit dem MSE über den Trainingszeitraum beider Durchläufe (Abbildung 9-3). Sie können die dargestellten Durchläufe an- und ausschalten, hinein- und herauszoomen, über der Kurve schweben, um Details zu erfahren und so weiter.

Klicken Sie nun auf die Registerkarte *Graphs*. Sie sollten das Diagramm in Abbildung 9-4 sehen.

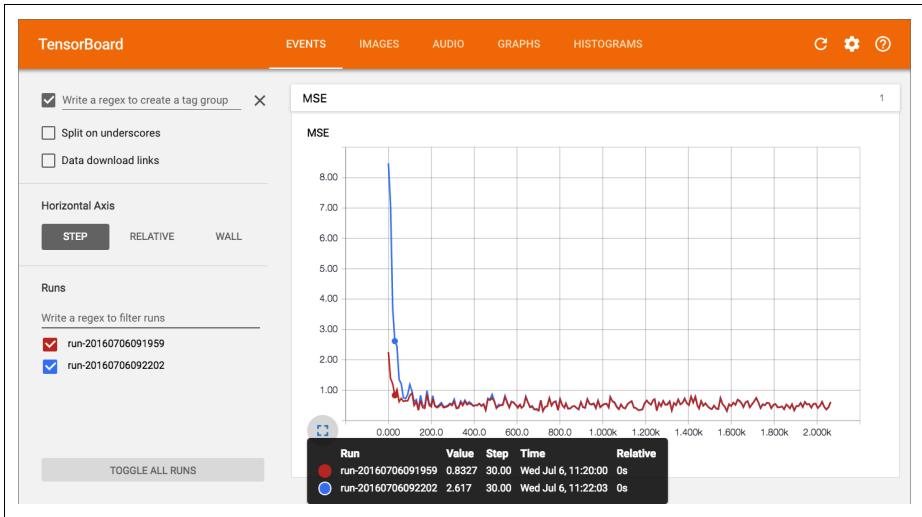


Abbildung 9-3: Visualisieren von Trainingsdurchläufen mit TensorBoard

Zum Verbessern der Übersichtlichkeit sind Knoten mit vielen *Kanten* (d. h. Verbindungen zu anderen Knoten) im Hilfsfenster auf der rechten Seite separat dargestellt (Sie können einen Knoten durch Rechtsklick zwischen dem Hauptgraphen und dem Hilfsfenster hin- und herbewegen). Einige Teile des Graphen sind von Beginn an zusammengefaltet. Bewegen Sie sich beispielsweise über den Knoten gradients und klicken Sie dort auf das \oplus -Symbol, um diesen Subgraphen zu entfalten. In diesem Subgraphen können Sie anschließend den Subgraphen `mse_grad` auspacken.

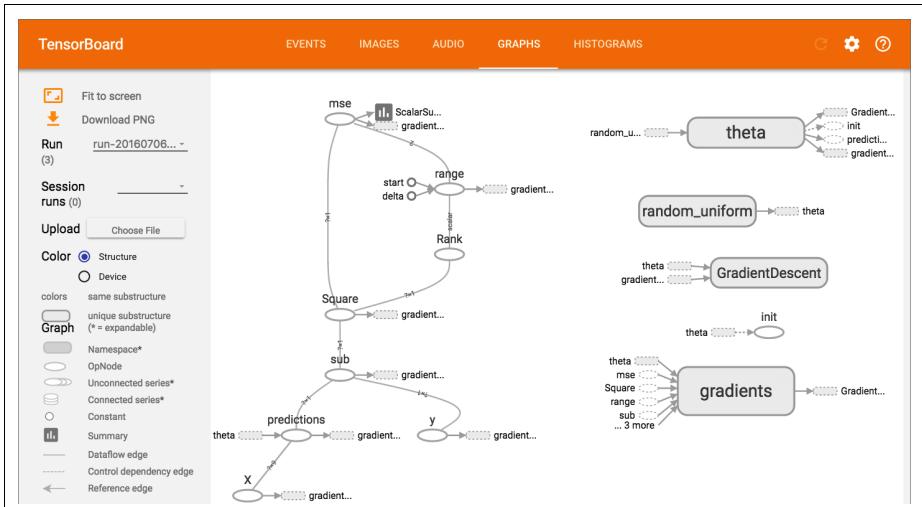


Abbildung 9-4: Visualisieren eines Graphen mit TensorBoard



Wenn Sie in Jupyter einen Blick auf den Graphen werfen möchten, sollten Sie sich die Funktion `show_graph()` aus dem Notebook für dieses Kapitel ansehen. Sie wurde ursprünglich von A. Mordvintsev für sein großartiges *Deepdream Tutorial Notebook* (<http://goo.gl/EtCWUc>) geschrieben. Eine andere Möglichkeit ist, das *TensorFlow Debugger Tool* (<https://github.com/ericjang/tdb>) von E. Jang zu installieren, in dem eine Jupyter-Erweiterung zum Visualisieren von Graphen (und mehr) enthalten ist.

Name Scopes

Bei komplexeren Modellen wie neuronalen Netzen wird ein Graph mit Tausenden Knoten schnell unübersichtlich. Um dies zu vermeiden, können Sie *Name Scopes* oder Bezugsrahmen erzeugen, um ähnliche Knoten zu gruppieren. Als Beispiel modifizieren wir den obigen Code, um die Operationen `error` und `mse` einem Scope mit dem Namen "loss" zuzuordnen:

```
with tf.name_scope("loss") as scope:  
    error = y_pred - y  
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

Der Name jeder im Scope definierten Operation erhält nun das Präfix "loss/":

```
>>> print(error.op.name)  
loss/sub  
>>> print(mse.op.name)  
loss/mse
```

In TensorBoard erscheinen die Knoten `mse` und `error` nun innerhalb des Namensraums `loss`, der automatisch zusammengeklappt erscheint (Abbildung 9-5).

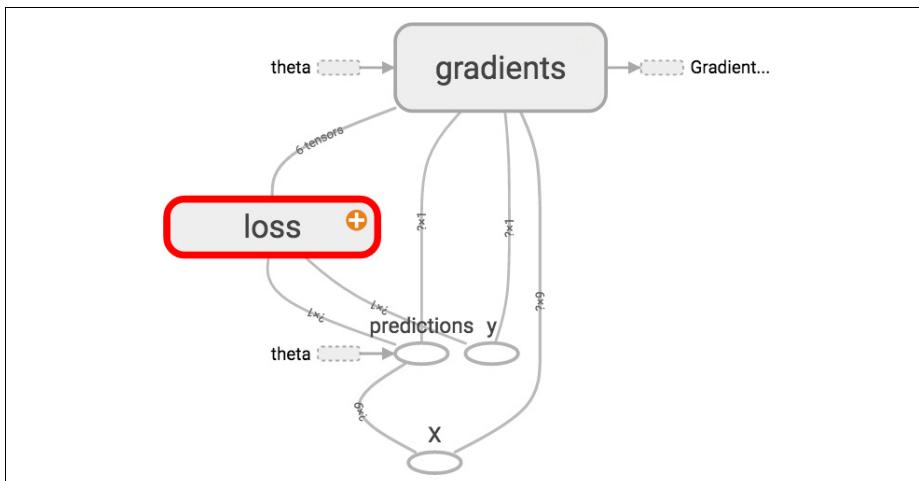


Abbildung 9-5: Ein zusammengeklappter Scope in TensorBoard

Modularität

Angenommen, Sie möchten einen Graphen erstellen, der die Ausgabe von zwei *Rectified Linear Units* (ReLU) hinzufügt. Ein ReLU berechnet eine lineare Funktion der Eingabedaten und gibt das Ergebnis aus, falls es positiv ist, andernfalls 0. Siehe dazu auch Formel 9-1.

Formel 9-1: Rectified Linear Unit

$$h_{\mathbf{w}, b}(\mathbf{X}) = \max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

Der folgende Code erledigt die Aufgabe, enthält aber zahlreiche Redundanzen:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```

Derart repetitiver Code ist schwer zu warten und fehleranfällig (tatsächlich enthält dieser Code einen beim Einfügen entstandenen Fehler. Entdecken Sie ihn?). Mit weiteren ReLUs würde es noch schwieriger werden. Glücklicherweise können Sie mit TensorFlow die DRY-Regel (Don't Repeat Yourself) beherzigen: Erstellen Sie einfach eine Funktion zum Erstellen eines ReLU. Der folgende Code erstellt fünf ReLUs und gibt deren Summe aus (beachten Sie, dass `add_n()` eine Operation zum Berechnen einer Summe von Tensoren aus einer Liste erstellt):

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

TensorFlow prüft beim Erstellen eines Knotens, ob dessen Name bereits existiert. Ist dies der Fall, fügt es dem Namen einen Unterstrich gefolgt von einem Index hinzu, um einen eindeutigen Namen zu erzeugen. Daher enthält die erste ReLU

Knoten mit den Namen "weights", "bias", "z" und "relu" (und viele andere Knoten mit Standardnamen wie "MatMul"); die zweite ReLU enthält Knoten mit den Namen "weights_1", "bias_1" und so weiter; die dritte ReLU enthält Knoten mit den Namen "weights_2", "bias_2" und so weiter. TensorBoard erkennt solche Serien und klappt diese zusammen, um die Übersicht zu erhöhen (siehe Abbildung 9-6).

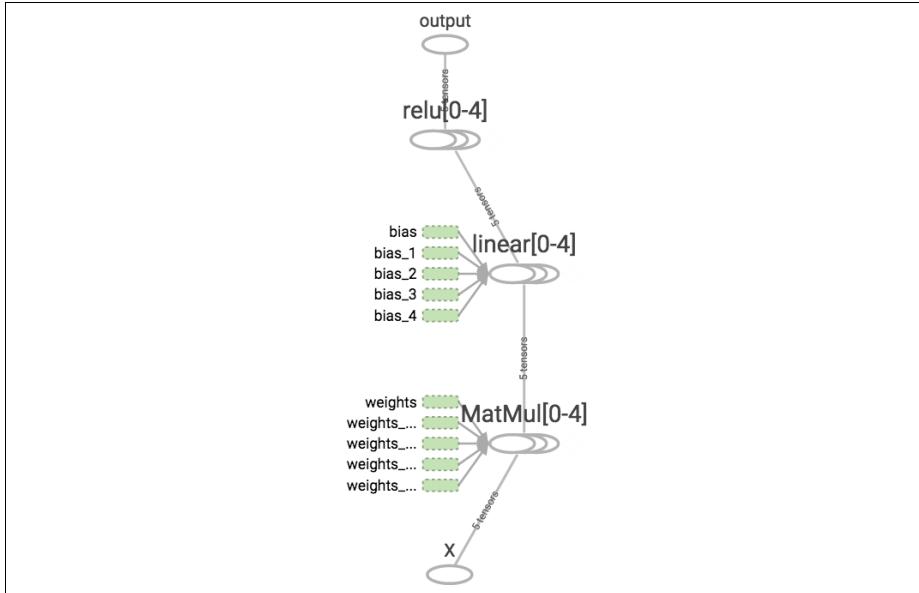


Abbildung 9-6: Serie zusammengeklappter Knoten

Mit Scopes können Sie den Graphen deutlich sauberer gestalten. Sie können einfach den gesamten Inhalt der Funktion `relu()` in einen Scope verschieben. Abbildung 9-7 zeigt den entstehenden Graphen. Dabei gibt TensorFlow auch den Scopes selbst eindeutige Namen, indem `_1`, `_2` und so weiter angehängt wird.

```
def relu(X):
    with tf.name_scope("relu"):
        [...]
```

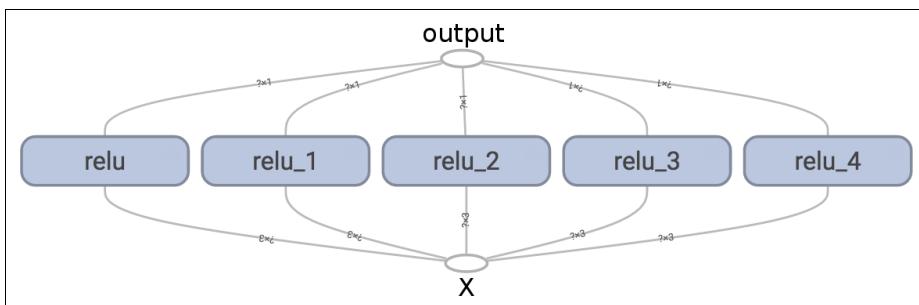


Abbildung 9-7: Ein aufgeräumter Graph mit Scopes für einzelne Units

Teilen von Variablen

Wenn Sie eine Variable mehreren Komponenten Ihres Graphen zur Verfügung stellen möchten, können Sie diese zuerst erstellen und dann als Parameter den benötigten Funktionen übergeben. Nehmen wir beispielsweise an, Sie möchten den ReLU-Schwellenwert (momentan auf 0 festgelegt) allen ReLUs als Variable `threshold` bereitstellen. Sie könnten diese Variable zuerst erstellen und dann der Funktion `relu()` übergeben:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Das funktioniert hervorragend: Sie können durch Setzen der Variablen `threshold` den Schwellenwert sämtlicher ReLUs mit dieser Variablen einstellen. Wenn es jedoch viele gemeinsame Parameter wie diesen gibt, müssen Sie jedes Mal sämtliche Parameter einzeln übergeben. Häufig erstellt man stattdessen ein Python-Dictionary mit allen Variablen des Modells und übergibt dies jeder Funktion. Manchmal erstellt man auch eine Klasse für jedes Modul (z.B. eine Klasse `ReLU`) und verwendet Klassenattribute zum Verwalten der geteilten Parameter. Eine weitere Möglichkeit besteht darin, beim ersten Aufruf die geteilte Variable als Attribut der Funktion `relu()` zuzuweisen:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow bietet eine weitere Alternative, die sauberer und modularen Code eher als die obigen Lösungen begünstigt.⁵ Diese ist zunächst etwas schwerer nachzuvollziehen, wird aber in TensorFlow oft verwendet. Deshalb lohnt es sich, etwas mehr ins Detail zu gehen. Die Grundidee ist, die Funktion `get_variable()` zu verwenden, die die Variable beim ersten Mal erstellt und sie wiederverwendet, falls sie bereits existiert. Das gewünschte Verhalten (neu erstellen oder wiederverwenden) wird über ein Attribut des aktuellen `variable_scope()` kontrolliert. Das folgende Codebeispiel erstellt eine Variable namens "`relu/threshold`" (als Skalar, da `shape=()` gegeben ist, mit dem Startwert 0.0):

⁵ Das Erstellen einer Klasse `ReLU` ist vermutlich die sauberste Lösungsmöglichkeit, aber sie ist eher schwergängig.

```

with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))

```

Beachten Sie, dass dieser Code einen Ausnahmefehler verursacht, falls ein früherer Aufruf von `get_variable()` diese Variable bereits erstellt hat. Damit wird vermieden, dass eine Variable versehentlich wiederverwendet wird. Wenn Sie eine Variable wiederverwenden möchten, müssen Sie dies explizit angeben, indem Sie den Parameter `reuse` auf `True` setzen (in diesem Fall müssen Sie die Abmessungen oder den Startwert nicht angeben):

```

with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")

```

Dieser Code greift auf die bestehende Variable "relu/threshold" zu oder löst einen Ausnahmefehler auf, falls sie nicht existiert oder nicht mit `get_variable()` erstellt wurde. Alternativ dazu können Sie das Attribut `reuse` auf `True` setzen, indem Sie die Methode `reuse_variables()` aufrufen:

```

with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")

```



Ist `reuse` erst einmal auf `True` gesetzt, lässt es sich im gleichen Codeabschnitt nicht wieder zurück auf `False` setzen. Wenn Sie außerdem weitere Variablen-Kontexte innerhalb dieses Abschnitts definieren, erben diese automatisch den Parameter `reuse=True`. Nur mit `get_variable()` erstellte Variablen lassen sich auf diese Weise wiederverwenden.

Sie haben nun alle Teile beisammen, um die Variable `threshold` der Funktion `relu()` zur Verfügung zu stellen, ohne sie als Parameter zu übergeben:

```

def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold") # verwende bestehende Variable
        [...]
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"): # erstelle die Variable
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    relus = [relu(X) for relu_index in range(5)]
    output = tf.add_n(relus, name="output")

```

Im Code wird zunächst die Funktion `relu()` definiert, anschließend wird die Variable `relu/threshold` (als später mit 0.0 initialisierter Skalar) erstellt. Durch Aufruf der Funktion `relu()` werden dann fünf ReLUs erstellt. Die Funktion `relu()` verwendet die Variable `relu/threshold` und erstellt die übrigen ReLU-Knoten.



Mit `get_variable()` erstellte Variablen erhalten stets ihren `variable_scope` als Präfix (z.B. "relu/threshold"). Bei allen anderen Knoten (darunter auch mit `tf.Variable()` erstellte Variablen) verhält sich der Kontext der Variablen wie ein neuer Namenskontext. Falls bereits ein Namenskontext mit einem identischen Namen erstellt wurde, wird der Name durch ein Suffix eindeutig gemacht. Beispielsweise haben alle im obigen Code erstellten Knoten (außer der Variablen `threshold`) Namen mit den Präfixen "relu_1/" bis "relu_5/", wie Abbildung 9-8 zeigt.

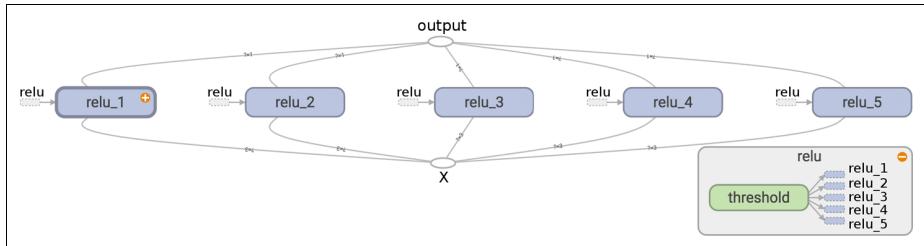


Abbildung 9-8: Fünf ReLUs, die sich die Variable `threshold` teilen

Es ist etwas unglücklich, dass die Variable `threshold` außerhalb der Funktion `relu()` definiert werden muss, wo der restliche ReLU-Code liegt. Um dies zu beheben, erstellen wir im folgenden Codebeispiel die Variable `threshold` beim ersten Aufruf der Funktion `relu()` und verwenden sie in den folgenden Aufrufen. Damit muss sich die Funktion `relu()` nicht mehr um Namensräume oder das Teilen von Variablen sorgen: Sie ruft einfach `get_variable()` auf, wodurch die Variable `threshold` erstellt oder gesetzt wird (sie muss nicht wissen, was davon zutrifft). Der übrige Code ruft `relu()` fünfmal auf, wobei beim ersten Aufruf `reuse=False` und bei den übrigen `reuse=True` gesetzt ist.

```
def relu(x):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(tf.relu(X))
    output = tf.add_n(relus, name="output")
```

Der sich daraus ergebende Graph sieht etwas anders aus, da die geteilte Variable innerhalb des ersten ReLU liegt (siehe Abbildung 9-9).

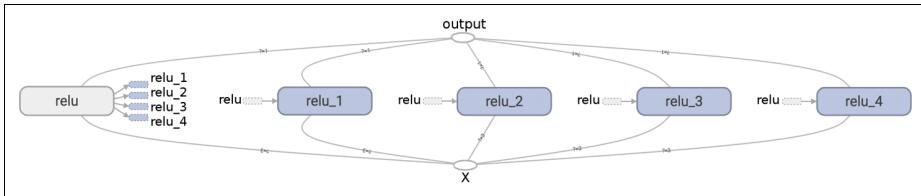


Abbildung 9-9: Fünf ReLUs teilen sich die Variable threshold

Damit beschließen wir unsere Einführung in TensorFlow. Wir werden weitere fortgeschrittene Themen in den folgenden Kapiteln besprechen, insbesondere neuronale Netzwerke für Deep Learning, Convolutional Neural Networks und rekurrente neuronale Netze, außerdem das Skalieren von TensorFlow durch Multithreading, Queues, mehrere GPUs und mehrere Server.

Übungen

- Was sind die wesentlichen Vorteile, die ein Berechnungsgraph gegenüber dem direkten Ausführen der Berechnungen bietet? Was sind die Nachteile?
- Ist der Ausdruck `a_val = a.eval(session=sess)` äquivalent zu `a_val = sess.run(a)`?
- Ist der Ausdruck `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` äquivalent zu `a_val, b_val = sess.run([a, b])`?
- Können Sie zwei Graphen in derselben Session ausführen?
- Wenn Sie einen Graphen `g` mit der Variablen `w` erstellen, dann zwei Threads starten und in jedem Thread eine Session mit dem gleichen Graphen `g` erstellen, enthält dann jede Session eine eigene Kopie der Variablen `w` oder wird sie geteilt?
- Wann wird eine Variable initialisiert? Wann wird sie zerstört?
- Was ist der Unterschied zwischen einem Platzhalter und einer Variablen?
- Was passiert, wenn Sie den Graphen eine von einem Platzhalter abhängige Operation ausführen lassen, diesem aber keinen Wert zuweisen? Was passiert, wenn die Operation nicht vom Platzhalter abhängig ist?
- Können Sie beim Ausführen eines Graphen den Ausgabewert einer beliebigen Operation angeben oder nur den Wert von Platzhaltern?
- Wie können Sie einer Variablen einen beliebigen Wert zuweisen (während der Ausführungsphase)?
- Wie oft muss Autodiff im Reverse-Modus einen Graphen durchlaufen, um die Ableitungen einer Kostenfunktion nach zehn Variablen zu berechnen? Wie sieht es bei Autodiff im Forward-Modus aus? Und bei der symbolischen Differenzierung?

12. Implementieren Sie die logistische Regression mit dem Mini-Batch-Gradientenverfahren in TensorFlow. Trainieren Sie es und werten Sie es auf dem Datensatz moons aus (siehe Kapitel 5). Versuchen Sie, sämtliche Besonderheiten hinzuzufügen:

- Definieren Sie den Graphen innerhalb einer Funktion `logistic_regression()`, die sich leicht wiederverwenden lässt.
- Speichern Sie in regelmäßigen Abständen einen Zwischenstand beim Trainieren mit einem `Saver`. Speichern Sie auch das endgültige Modell nach dem Training.
- Stellen Sie den zuletzt gespeicherten Zwischenstand wieder her, falls das Training unterbrochen wurde.
- Definieren Sie den Graphen mit sinnvoll gewählten Scopes, sodass der Graph in TensorBoard gut aussieht.
- Fügen Sie Zusammenfassungen hinzu, um die Lernkurven in TensorBoard darzustellen.
- Optimieren Sie einige Hyperparameter wie die Lernrate oder die Größe der Mini-Batches und betrachten Sie die Gestalt der Lernkurve.

Lösungen zu diesen Übungen finden Sie in Anhang A.

Einführung in künstliche neuronale Netze

Vögel haben uns zum Fliegen inspiriert, die Klette zu Klettverschlüssen, und viele weitere Erfindungen sind ebenfalls von der Natur inspiriert. Es erscheint also nur logisch, in der Architektur des Gehirns nach Inspirationen zum Erschaffen intelligenter Maschinen zu suchen. Dies ist die Grundidee bei *künstlichen neuronalen Netzen* (engl. ANNs). Allerdings müssen Flugzeuge, wenn sie auch von Vögeln inspiriert sind, nicht mit den Flügeln schlagen. In ähnlicher Weise haben sich auch ANNs nach und nach recht weit von ihren biologischen Cousins entfernt. Einige Forscher sind sogar der Meinung, dass wir die biologische Analogie komplett außer Acht lassen sollten (z.B. indem wir von »Einheiten« anstatt von »Neuronen« sprechen), um unsere Kreativität nicht auf biologisch plausible Systeme zu beschränken.¹

ANNs sind die Kernkomponente des Deep Learning. Sie sind flexibel, mächtig und skalierbar, was sie ideal für große und hochgradig komplexe Machine-Learning-Aufgaben einsetzbar macht, wie beispielsweise die Klassifizierung von Milliarden Bildern (Google Images), Spracherkennung (Apples Siri), Videoempfehlungen für Hunderte Millionen Nutzer pro Tag (YouTube) oder den Weltmeister im Brettspiel *Go* durch die Analyse von Millionen Partien und anschließendes Spielen gegen sich selbst zu schlagen (*AlphaGo* von DeepMind).

In diesem Kapitel werden wir künstliche neuronale Netze kennenlernen. Wir beginnen mit einem kurzen Überblick der ersten Architekturen von ANNs. Dann werden wir *mehrschichtige Perzeptrons* (MLPs) vorstellen und eines mithilfe von TensorFlow implementieren, um die Klassifikation der MNIST-Ziffern anzugehen (die bereits aus Kapitel 3 bekannt ist).

¹ Wenn Sie sich von der Biologie inspirieren lassen, ohne sich vor biologisch unrealistischen Modellen zu fürchten, erhalten Sie das Beste aus beiden Welten.

Von biologischen zu künstlichen Neuronen

Überraschenderweise gibt es ANNs schon eine ganze Weile: Das erste Mal wurden Sie 1943 vom Neurophysiologen Warren McCulloch und vom Mathematiker Walter Pitts erwähnt. In ihrem wegweisenden Artikel (<https://goo.gl/Ul4mxW>)² »A Logical Calculus of Ideas Immanent in Nervous Activity« stellen McCulloch und Pitts ein vereinfachtes rechnerisches Modell vor, nach dem biologische Neuronen im Gehirn von Tieren zusammenarbeiten könnten, um komplexe Berechnungen mithilfe von *Aussagenlogik* durchzuführen. Dies war die erste Architektur eines künstlichen neuronalen Netzwerks. Seitdem wurden viele weitere Architekturen erfunden, wie wir noch sehen werden.

Die frühen Erfolge von ANNs bis zu den 1960ern führten zur verbreiteten Annahme, dass wir uns schon bald mit wirklich intelligenten Maschinen unterhalten würden. Als klar wurde, dass dieses Versprechen nicht eingelöst werden würde (zumindest für eine lange Zeit), flossen Forschungsgelder in andere Richtungen, und für ANNs brach ein langes, dunkles Zeitalter an. In den frühen 1980ern lebte das Interesse an ANNs mit der Entdeckung neuer Netzwerkarchitekturen wieder auf. Aber in den 1990ern bevorzugten die meisten Forscher andere mächtige Machine-Learning-Verfahren wie Support Vector Machines (siehe Kapitel 5), da diese bessere Ergebnisse und ein solideres theoretisches Fundament versprachen. Schließlich erleben wir heute eine weitere Renaissance der ANNs. Wird auch diese Welle wie die vorigen wieder verebben? Es gibt gute Gründe anzunehmen, dass es diesmal anders ist und der Einfluss auf unseren Alltag beträchtlich sein wird:

- Heute sind gewaltige Datenmengen zum Trainieren von neuronalen Netzen verfügbar, und ANNs schneiden bei großen und komplexen Aufgaben häufig besser als andere ML-Verfahren ab.
- Der erhebliche Zuwachs an Rechenkapazität seit den 1990ern ermöglicht das Trainieren großer neuronaler Netze innerhalb eines sinnvollen Zeitraums. Teils liegt dies an Moores Law, teils an der Spieleindustrie, der wir leistungsfähige Grafikprozessoren verdanken.
- Die Algorithmen zum Trainieren sind verbessert worden. Eigentlich sind sie nur ein wenig anders als die in den 1990ern verwendeten, aber diese kleinen Änderungen haben zu sehr großen Verbesserungen geführt.
- Einige theoretische Einschränkungen von ANNs haben sich als Vorteile herausgestellt. Beispielsweise ging man davon aus, dass die Algorithmen zum Trainieren von ANNs wegen ihrer Neigung zu lokalen Optima zum Scheitern verurteilt wären. Diese sind aber in der Praxis selten (oder liegen zumindest nahe genug am globalen Optimum).
- ANNs befinden sich in einem förderlichen Kreislauf von finanzieller Unterstützung und Fortschritt. Faszinierende auf ANNs basierende Produkte schaf-

² »A Logical Calculus of Ideas Immanent in Nervous Activity«, W. McCulloch and W. Pitts (1943).

fen es regelmäßig in die Schlagzeilen, wodurch sie weitere Aufmerksamkeit und Förderung anziehen, was wiederum weitere Fortschritte und noch faszinierende Produkte nach sich zieht.

Biologische Neuronen

Bevor wir künstliche Neuronen besprechen, schauen wir uns kurz ein biologisches Neuron an (dargestellt in Abbildung 10-1). Es ist eine ungewöhnlich aussehende Zelle, wie man sie vor allem im zerebralen Kortex von Tieren antrifft (z.B. in Ihrem Gehirn). Sie besteht aus einem *Zellkörper* mit dem Zellkern und den meisten komplexen Bestandteilen einer Zelle, vielen verzweigten Auswüchsen, den *Dendriten*, sowie einem sehr langen Auswuchs, dem *Axon*. Die Länge des Axons kann einigen oder bis zu Zehntausenden Längen des Zellkörpers entsprechen. Am Ende teilt sich das Axon in feine Verästelungen, die *Telodendria* auf. An der Spitze dieser Verästelungen befinden sich winzige Strukturen, die *synaptischen Verbindungen* (oder einfach *Synapsen*), die mit den Dendriten (oder Zellkörpern) anderer Neuronen verbunden sind. Biologische Neuronen erhalten über die Synapsen kurze elektrische Impulse oder *Signale* von anderen Neuronen. Erhält ein Neuron innerhalb weniger Millisekunden eine ausreichende Anzahl Signale, feuert es sein eigenes Signal ab.

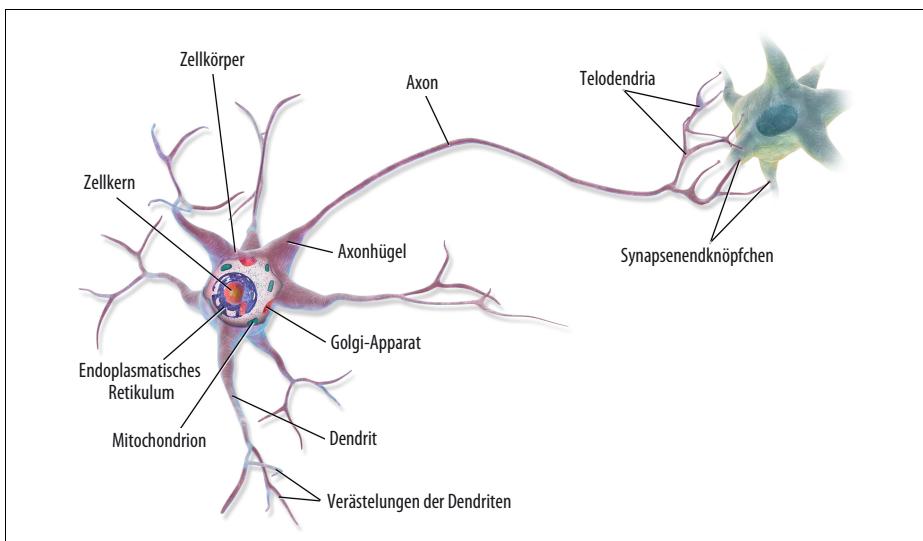


Abbildung 10-1: Biologisches Neuron³

Biologische Neuronen verhalten sich also auf scheinbar einfache Weise. Sie sind aber als riesiges Netzwerk mit Milliarden Neuronen organisiert, in dem jedes Neu-

³ Bild von Bruce Blaus (Creative Commons 3.0, <https://creativecommons.org/licenses/by/3.0/>). Reproduziert nach <https://en.wikipedia.org/wiki/Neuron>.

ron mit Tausenden anderen Neuronen verbunden ist. Mit solch einem riesigen Netzwerk recht einfacher Neuronen lassen sich hoch komplexe Berechnungen durchführen, wie auch ein komplexer Ameisenstaat aus der Zusammenarbeit einfacher Ameisen entsteht. Die Architektur von biologischen neuronalen Netzen (BNNs)⁴ wird noch immer untersucht. Einige Teile des Gehirns sind aber kartiert worden, und es sieht danach aus, dass Neuronen häufig in aufeinanderfolgenden Schichten zu finden sind, wie Abbildung 10-2 zeigt.

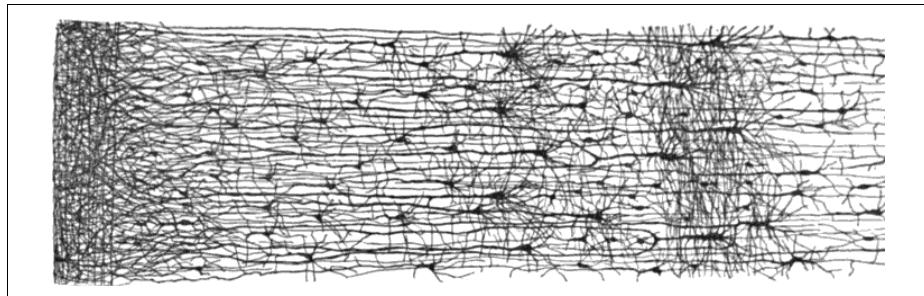


Abbildung 10-2: Mehrere Schichten eines biologischen neuronalen Netzes (menschlicher Kortex)⁵

Logische Berechnungen mit Neuronen

Warren McCulloch und Walter Pitts haben ein sehr einfaches Modell eines biologischen Neurons vorgeschlagen, das später als *künstliches Neuron* bekannt wurde: Es besitzt eine oder mehrere binäre (ein/aus) Eingabeleitungen und eine binäre Ausgabeleitung. Das künstliche Neuron wird einfach dann aktiviert, wenn eine bestimmte Anzahl Eingaben aktiv sind. McCulloch und Pitts haben nachgewiesen, dass sich selbst mit einem derart einfachen Modell ein neuronales Netz konstruieren lässt, das jeden möglichen logischen Ausdruck berechnet. Als Beispiel werden wir einige ANNs konstruieren, die verschiedene logische Berechnungen durchführen (siehe Abbildung 10-3). Dabei wird ein Neuron aktiviert, wenn mindestens zwei seiner Eingabeleitungen aktiv sind.

- Das erste Netz auf der linken Seite ist eine Identitätsfunktion: Ist Neuron A aktiviert, wird auch Neuron C aktiviert (da es zwei Signale von Neuron A erhält). Ist Neuron A aber inaktiv, ist auch Neuron C inaktiv.
- Das zweite Netz führt ein logisches AND durch: Neuron C wird nur aktiviert, wenn sowohl Neuron A als auch Neuron B aktiviert sind (ein einzelnes Eingabesignal reicht nicht aus, um Neuron C zu aktivieren).

⁴ Im Zusammenhang mit Machine Learning sind mit dem Begriff »neuronales Netz« grundsätzlich ANNs, nicht BNNs gemeint.

⁵ Drawing of a cortical lamination von S. Ramon y Cajal (public domain). Reproduziert nach https://en.wikipedia.org/wiki/Cerebral_cortex.

- Das dritte Netz führt ein logisches OR durch: Neuron C wird aktiviert, wenn entweder Neuron A oder Neuron B aktiviert ist (oder beide).
- Wenn wir schließlich annehmen, dass eine Eingabeleitung die Aktivität eines Neurons unterbinden kann (was bei biologischen Neuronen der Fall ist), berechnet das vierte Netz einen etwas komplexeren logischen Ausdruck: Neuron C wird nur aktiviert, wenn Neuron A aktiv und Neuron B inaktiv ist. Wenn Neuron A ständig aktiv ist, erhalten Sie ein logisches NOT: Neuron C ist aktiv, wenn Neuron B inaktiv ist und umgekehrt.

Sie können sich sicher vorstellen, wie sich diese Netze zur Berechnung komplexer logischer Ausdrücke kombinieren lassen (siehe Übungen am Ende des Kapitels).

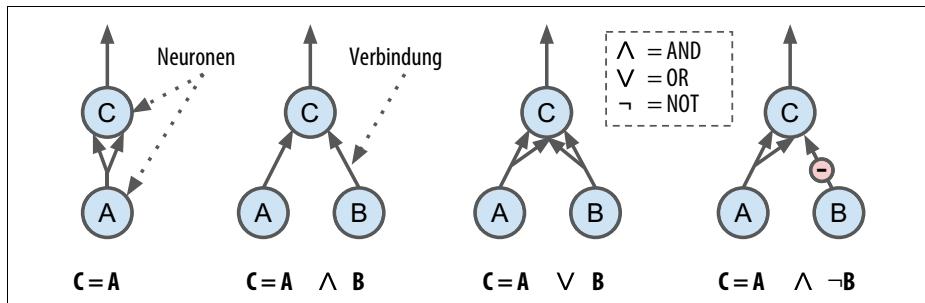


Abbildung 10-3: ANNs für einfache logische Berechnungen

Das Perzeptron

Das 1957 von Frank Rosenblatt erfundene *Perzeptron* gehört zu den einfachsten Architekturen neuronaler Netze. Es baut auf einem leicht unterschiedlichen künstlichen Neuron auf, (siehe Abbildung 10-4) der *Linear Threshold Unit* (LTU): Die Ein- und Ausgaben sind nun Zahlen (anstatt binäre Ein-/Aus-Werte), und zu jeder Eingabeleitung gehört ein Gewicht. Die LTU berechnet eine gewichtete Summe ihrer Eingaben ($z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$) und wendet dann eine *Aktivierungsfunktion* auf diese Summe an und gibt das Ergebnis aus: $h_{\mathbf{w}}(\mathbf{x}) = \text{Aktivierungsfunktion}(\mathbf{w}^T \cdot \mathbf{x})$.

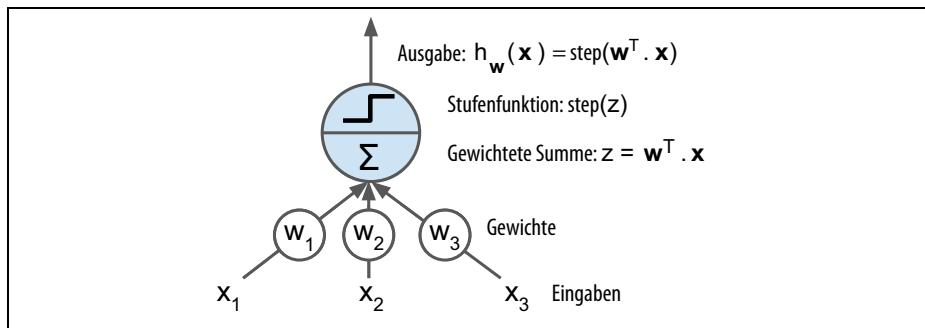


Abbildung 10-4: Linear Threshold Unit

Die im Perzeptron am häufigsten verwendete Aktivierungsfunktion ist die *Heaviside-Funktion* (siehe Formel 10-1). Manchmal wird stattdessen die Vorzeichenfunktion verwendet.

Formel 10-1: Häufig in Perzeptrons eingesetzte Aktivierungsfunktionen

$$\text{heaviside}(z) = \begin{cases} 0 & \text{wenn } z < 0 \\ 1 & \text{wenn } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{wenn } z < 0 \\ 0 & \text{wenn } z = 0 \\ +1 & \text{wenn } z > 0 \end{cases}$$

Eine einzelne LTU lässt sich für einfache lineare binäre Klassifikationsaufgaben einsetzen. Sie berechnet eine Linearkombination der Eingabewerte und gibt die positive Kategorie aus, falls das Ergebnis einen Schwellenwert überschreitet, und andernfalls die negative Kategorie (wie bei der logistischen Regression oder einer linearen SVM). Sie könnten beispielsweise eine einzelne LTU zum Klassifizieren von Iris-Blüten anhand der Länge und Breite der Kronblätter einsetzen (wie in den vorigen Kapiteln mit einem zusätzlichen Bias-Merkmal $x_0 = 1$). Beim Trainieren einer LTU gilt es, die richtigen Werte für w_0 , w_1 und w_2 zu finden (den Algorithmus zum Trainieren besprechen wir gleich).

Ein Perzeptron besteht einfach aus einer einzelnen Schicht LTUs,⁶ wobei jedes Neuron mit allen Eingabewerten verbunden ist. Diese Verbindungen werden oft als spezielle Neuronen zum Durchreichen der Information repräsentiert, den *Eingabeneuronen*: Diese geben einfach nur die erhaltenen Eingabedaten aus. Außerdem wird üblicherweise ein zusätzliches Bias-Merkmal hinzugefügt ($x_0 = 1$). Dieses Bias-Merkmal wird durch ein weiteres spezielles Neuron repräsentiert, dem *Bias-Neuron*, das stets 1 ausgibt.

In Abbildung 10-5 ist ein Perzeptron mit zwei Eingaben und drei Ausgaben dargestellt. Dieses Perzeptron kann Datenpunkte gleichzeitig drei unterschiedlichen binären Kategorien zuordnen, wodurch es zu einem Klassifikator mit mehreren Ausgaben wird.

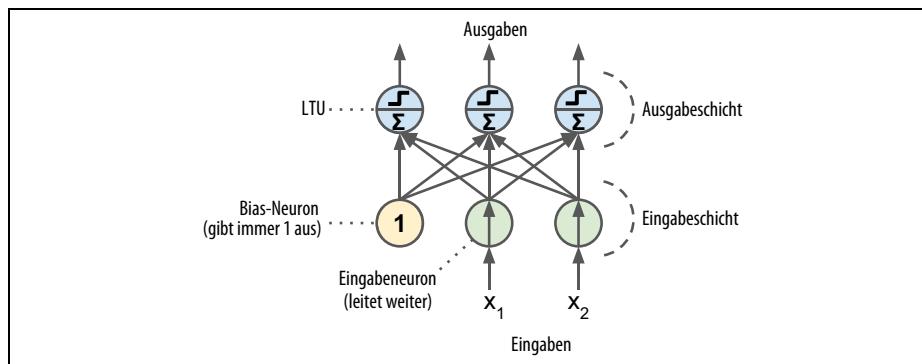


Abbildung 10-5: Perzeptron als Diagramm

6 Der Name *Perzeptron* wird manchmal für ein winziges Netz mit einer einzelnen LTU verwendet.

Wie aber lässt sich ein Perzeptron trainieren? Der von Frank Rosenblatt vorgeschlagene Algorithmus zum Trainieren von Perzeptrons ist in weiten Teilen von der *Hebb'schen Lernregel* inspiriert. In seinem 1949 veröffentlichten Buch *The Organization of Behavior* stellte Donald Hebb eine These vor, nach der die Verbindung zweier biologischer Neuronen stärker werde, wenn eines das andere häufig aktiviert. Siegrid Löwel hat diesen Gedanken später als knackigen Satz formuliert: »Cells that fire together, wire together.« Dieser Grundsatz wurde später als Hebb'sche Lernregel (oder *Hebb'sches Lernen*) bekannt; dabei wird die Verbindung zweier Neuronen immer dann verstärkt, wenn beide die gleiche Ausgabe haben. Perzeptrons lassen sich mit einer Variante dieser Regel trainieren, die die vom Netz begangenen Fehler berücksichtigt; Verbindungen, die zu einer falschen Ausgabe führen, werden nicht verstärkt. Beim Trainieren wird dem Perzeptron ein einzelner Trainingsdatenpunkt vorgestellt und eine Vorhersage getroffen. Bei jedem Ausgabeneuron, das eine falsche Vorhersage trifft, werden die Gewichte der Verbindungen verstärkt, die zu einer korrekten Vorhersage beigetragen hätten. Diese Regel ist in Formel 10-2 dargestellt.

Formel 10-2: Lernregel für Perzeptrons (Aktualisieren von Gewichten)

$$w_{i,j} \text{ (nächster Schritt)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

- $w_{i,j}$ ist dabei das Gewicht der Verbindung zwischen dem i^{ten} Neuron und dem j^{ten} Ausgabeneuron.
- x_i ist der i^{te} Wert des aktuellen Trainingsdatenpunkts.
- y_j ist die Ausgabe des j^{ten} Ausgabeneurons für den aktuellen Trainingsdatenpunkt.
- \hat{y}_j ist die Zielausgabe des j^{ten} Ausgabeneurons für den aktuellen Trainingsdatenpunkt.
- η ist die Lernrate.

Die Entscheidungsgrenze jedes Ausgabeneurons ist linear. Damit sind Perzeptrons nicht in der Lage, komplexe Muster zu erlernen (wie auf logistischer Regression aufbauende Klassifikatoren). Wenn sich die Trainingsdatenpunkte aber linear separieren lassen, konvergiert dieser Algorithmus laut Rosenblatt zu einer Lösung.⁷ Dies nennt man das *Perzeptron-Konvergenztheorem*.

Scikit-Learn enthält die Klasse `Perceptron`, die ein einzelnes LTU-Netz implementiert. Diese funktioniert wie erwartet – beispielsweise auf dem Iris-Datensatz (aus Kapitel 4):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
```

⁷ Diese Lösung ist nicht eindeutig: Wenn die Daten linear separierbar sind, gibt es eine unendliche Menge Hyperebenen, die diese separieren.

```

iris = load_iris()
X = iris.data[:, (2, 3)] # Länge, Breite von Kronblättern
y = (iris.target == 0).astype(np.int) # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])

```

Sie haben vielleicht bemerkt, dass der Lernalgorithmus für Perzeptrons stark an das stochastische Gradientenverfahren erinnert. Tatsächlich ist die Scikit-Learn-Klasse Perceptron mit den folgenden Hyperparametern zum SGDClassifier äquivalent: loss="perceptron", learning_rate="constant", eta0=1 (die Lernrate) und penalty=None (keine Regularisierung).

Im Gegensatz zur logistischen Regression geben Perzeptrons keine Wahrscheinlichkeit für die Kategorien aus; vielmehr treffen sie anhand eines festen Schwellenwerts Vorhersagen. Dies ist ein guter Grund, die logistische Regression dem Perzepton vorzuziehen.

In einer Monografie aus dem Jahr 1969 mit dem Titel *Perceptrons* hoben Marvin Minsky und Seymour Papert eine Reihe ernster Nachteile von Perzeptrons hervor, insbesondere ihr Versagen bei einer Reihe trivialer Probleme (z. B. das *exklusive OR* (XOR) als Klassifikationsaufgabe; dargestellt auf der linken Seite von Abbildung 10-6). Natürlich gilt dies auch für jedes andere lineare Klassifikationsmodell (wie die logistische Regression), aber die Wissenschaft hatte wesentlich größere Hoffnungen in Perzeptrons gesetzt. Daher war die Enträuschung groß, und in der Folge wandten sich viele Wissenschaftler vom *Konnektionismus* insgesamt ab (d. h. dem Studium neuronaler Netze), um sich übergeordneten Aufgabenstellungen wie Logik, Problemlösung und Suche zuzuwenden.

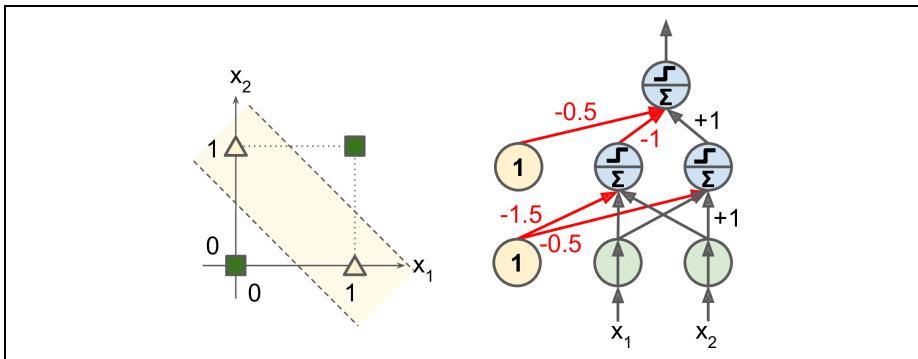


Abbildung 10-6: XOR-Klassifikationsproblem und ein MLP zu dessen Lösung

Es stellt sich aber heraus, dass sich einige dieser Beschränkungen aufheben lassen, indem man mehrere Perzeptrons in Reihe schaltet. Das dabei entstehende ANN bezeichnet man als *mehrschichtiges Perzepton* (MLP). Insbesondere ist ein MLP in der Lage, das XOR-Problem zu bewältigen, wie Sie durch Nachrechnen der Ausgabe des MLP auf der rechten Seite von Abbildung 10-6 mit allen möglichen Einga-

ben prüfen können: Mit den Eingaben $(0, 0)$ oder $(1, 1)$ gibt das Netzwerk 0 aus, und mit den Eingaben $(0, 1)$ oder $(1, 0)$ gibt es 1 aus.

Mehrschichtiges Perzeptron und Backpropagation

Ein MLP setzt sich aus einer Eingabeschicht (zum Durchreichen) und einer oder mehreren Schichten von LTUs zusammen, den *verborgenen Schichten*, und einer letzten Schicht LTUs, der *Ausgabeschicht* (siehe Abbildung 10-7). Bis auf die Ausgabeschicht enthält jede Schicht ein Bias-Neuron und ist mit der nächsten Schicht vollständig verbunden. Wenn ein ANN aus zwei oder mehr verborgenen Schichten besteht, bezeichnet man es auch als *Deep-Learning-Netz* (DNN).

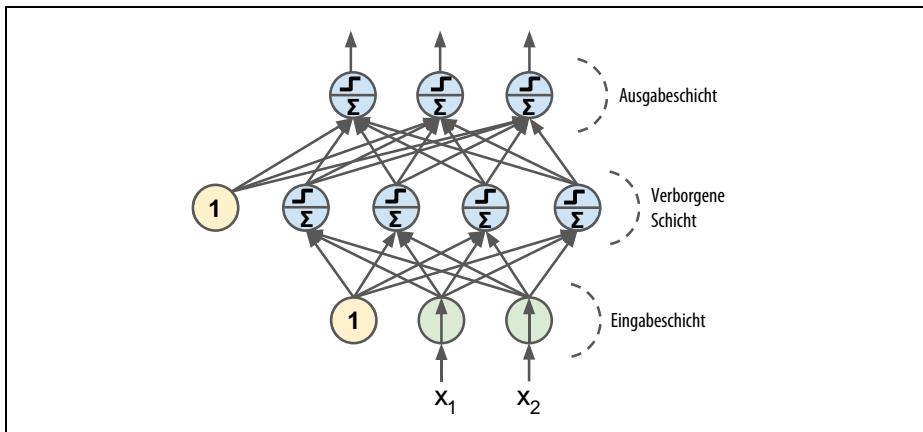


Abbildung 10-7: Mehrschichtiges Perzeptron

Viele Jahre lang haben Forscher vergeblich nach einer Möglichkeit zum Trainieren von MLPs gesucht. Im Jahr 1986 aber publizierten D. E. Rumelhart et al. einen wegweisenden Artikel (<https://goo.gl/Wl7Xyc>)⁸, der den *Backpropagation*-Algorithmus bekannt machte.⁹ Heute würden wir ihn als Gradientenverfahren mit Autodiff im Reverse-Modus beschreiben (das Gradientenverfahren wurde in Kapitel 4 besprochen, Autodiff in Kapitel 9).

Der Algorithmus speist jeden Trainingsdatenpunkt in das Netz ein und berechnet die Ausgabe jedes Neurons in jeder aufeinanderfolgenden Schicht (dies ist ein vorwärts gerichteter Durchlauf wie beim Treffen von Vorhersagen). Anschließend wird der Fehler der Ausgabe des Netzes gemessen (d.h. die Differenz zwischen der gewünschten und der tatsächlichen Ausgabe). Für jedes Neuron in der letzten verborgenen Schicht wird dann bestimmt, wie stark es zum Fehler der Ausgabe bei-

8 »Learning Internal Representations by Error Propagation«, D. Rumelhart, G. Hinton, R. Williams (1986).

9 Dieser Algorithmus wurde von mehreren Wissenschaftlern in unterschiedlichen Fachgebieten unabhängig voneinander entdeckt, darunter P. Werbos im Jahr 1974.

trug. Anschließend wird berechnet, welcher Teil dieser Fehlerbeiträge auf jedes Neuron in der vorigen verborgenen Schicht entfiel – und so weiter, bis der Algorithmus die Eingabeschicht erreicht. In diesem rückwärtigen Durchlauf wird der Fehlergradient über sämtliche Gewichte im Netz zurückverfolgt (daher der Name Backpropagation). Wenn Sie den Algorithmus für Autodiff im Reverse-Modus in Anhang D untersuchen, erkennen Sie, dass die beiden Durchläufe (vorwärts und rückwärts) des Backpropagation-Verfahrens identisch mit Autodiff im Reverse-Modus sind. Der letzte Schritt des Backpropagation-Algorithmus ist ein Schritt im Gradientenverfahren auf allen Gewichten im Netz unter Verwendung des zuvor bestimmten Gradienten.

Verkürzen wir dies noch weiter: Bei jedem Trainingsdatenpunkt trifft der Backpropagation-Algorithmus zuerst eine Vorhersage (im Vorwärtsdurchlauf), bestimmt den Fehler, durchläuft dann rückwärts jede Schicht, um den Fehlerbeitrag jeder Verbindung zu ermitteln (im Rückwärtsdurchlauf), und verändert schließlich die Gewichte der Verbindungen, um den Fehler zu verringern (als Schritt im Gradientenverfahren).

Damit dieser Algorithmus gut funktioniert, nahmen die Autoren eine wichtige Änderung an der Architektur des MLP vor: Sie ersetzten die Stufenfunktion mit der logistischen Funktion $\sigma(z) = 1 / (1 + \exp(-z))$. Dies erwies sich als entscheidend, weil die Stufenfunktion nur flache Abschnitte enthält und es daher keinen Gradienten gibt (die Gradientenmethode kann sich auf einer flachen Oberfläche nicht bewegen), wohingegen die Ableitung der logistischen Funktion überall ungleich null ist. Damit kann die Gradientenmethode an jeder Stelle ein wenig voranschreiten. Der Backpropagation-Algorithmus lässt sich auch mit anderen *Aktivierungsfunktionen* als der logistischen Funktion einsetzen. Zwei weitere beliebte Aktivierungsfunktionen sind:

Der Tangens hyperbolicus $\tanh(z) = 2\sigma(2z) - 1$

Wie die logistische Funktion ist der Tangens S-förmig, stetig und differenzierbar, aber die Ausgabewerte liegen im Bereich zwischen -1 und 1 (anstatt von 0 bis 1 bei der logistischen Funktion). Damit wird die Ausgabe jeder Schicht zu Beginn des Trainings tendenziell normalisiert (d.h. auf 0 zentriert). Dies beschleunigt bisweilen die Konvergenz.

Die ReLU-Funktion $\text{ReLU}(z) = \max(0, z)$

(*Erstmals in Kapitel 9 erwähnt.*) Diese Funktion ist stetig, aber bei $z = 0$ leider nicht differenzierbar (die Steigung ändert sich abrupt, wodurch die Gradientenmethode umherspringen kann). In der Praxis funktioniert diese Funktion aber sehr gut und ist außerdem schnell berechenbar. Wichtiger ist, dass sie keinen maximalen Ausgabewert besitzt, was einige Probleme des Gradientenverfahrens umgeht (wir kommen in Kapitel 11 hierauf zu sprechen).

Diese beliebten Aktivierungsfunktionen und ihre Ableitungen sind in Abbildung 10-8 dargestellt.

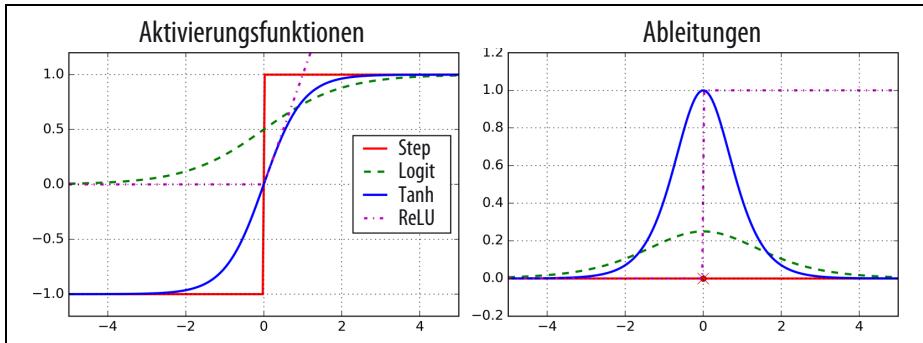


Abbildung 10-8: Aktivierungsfunktionen und ihre Ableitungen

Ein MLP wird häufig zur Klassifikation eingesetzt, wobei jede Ausgabe einer anderen binären Kategorie entspricht (z. B. Spam/Ham, dringend/nicht dringend und so weiter). Falls die Kategorien sich gegenseitig ausschließen (z. B. Kategorien von 0 bis 9 bei der Klassifikation der Bilder von Ziffern), werden in der Ausgabeschicht normalerweise die einzelnen Aktivierungsfunktionen durch eine gemeinsame *Softmax*-Funktion ersetzt (siehe Abbildung 10-9). Die Softmax-Funktion wurde in Kapitel 4 vorgestellt. Die Ausgabe jedes Neurons entspricht dann der geschätzten Wahrscheinlichkeit der entsprechenden Kategorie. Beachten Sie, dass das Signal nur in eine Richtung fließt (von der Ein- zur Ausgabe). Daher ist diese Architektur ein Beispiel für ein *Feed-Forward-Netz* (engl. Feedforward Neural Networks, FNN).

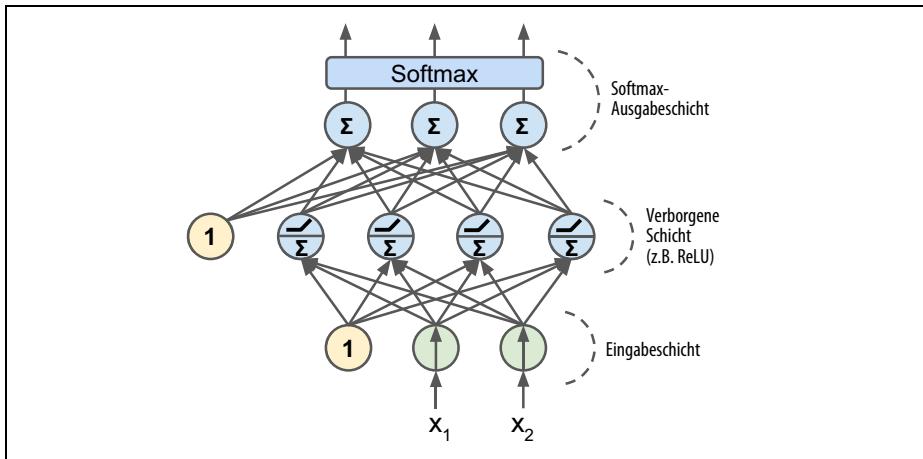


Abbildung 10-9: Ein modernes MLP (mit ReLU und Softmax) zur Klassifikation



Biologische Neuronen scheinen eine in etwa sigmoidale (S-förmige) Aktivierungsfunktion zu implementieren, daher hatte sich die Wissenschaft sehr lange auf Sigmoidalfunktionen eingeschossen. In ANNs funktioniert ReLU als Aktivierungsfunktion jedoch meist besser. In diesem Fall war die biologische Analogie irreführend.

Ein MLP mit der TensorFlow-API trainieren

Die einfachste Möglichkeit, ein MLP mit TensorFlow zu trainieren, ist, die abstrakte API `TF.Learn` zu verwenden, die eine zu Scikit-Learn kompatible API bereitstellt. Die Klasse `DNNClassifier` vereinfacht das Trainieren eines tiefen neuronalen Netzes mit einer beliebigen Anzahl verborgener Schichten und einer Softmax-Ausgabeschicht, um die geschätzten Wahrscheinlichkeiten der Kategorien auszugeben. Beispielsweise trainiert der folgende Code ein DNN zur Klassifikation, das aus zwei verborgenen Schichten (eine mit 300, eine mit 100 Neuronen) und einer Softmax-Ausgabeschicht mit 10 Neuronen besteht:

```
import tensorflow as tf

feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                         feature_columns=feature_cols)
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # wenn TensorFlow >= 1.1
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

Der Code erstellt zuerst einige Spalten mit Realzahlen aus dem Trainingsdatensatz (es gibt auch andere Arten von Spalten, z.B. für Kategorien). Anschließend erstellen wir einen `DNNClassifier` und stecken diesen in einen Adapter für die Kompatibilität mit Scikit-Learn. Schließlich führen wir 40000 Iterationen zum Trainieren von Batches mit jeweils 50 Datenpunkten aus.

Wenn Sie diesen Code auf dem MNIST-Datensatz ausführen, (nach dem Skalieren z.B. mit dem `StandardScaler` aus Scikit-Learn), erzielt das resultierende Modell auf dem Testdatensatz eine Genauigkeit von etwa 98.2%! Das ist besser als das beste in Kapitel 3 trainierte Modell:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = dnn_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred['classes'])
0.9825000000000004
```



Das Paket `tensorflow.contrib` enthält viele nützliche Funktionen, ist aber auch ein Ort für experimentellen Code, der sich noch nicht als Teil der Kern-API von TensorFlow qualifiziert hat. Daher kann sich die Klasse `DNNClassifier` (und jeder andere Code in `contrib`) in der Zukunft ohne Vorwarnung ändern.

Hinter den Kulissen erstellt der `DNNClassifier` sämtliche Schichten mit Neuronen mit der ReLU-Aktivierungsfunktion (wir können dies über den Hyperparameter `activation_fn` festlegen). Die Ausgabeschicht verwendet die Funktion Softmax, und die Kostenfunktion ist die (in Kapitel 4 vorgestellte) Kreuzentropie.

Ein DNN direkt mit TensorFlow trainieren

Wenn Sie sich mehr Kontrolle über die Architektur des Netzes wünschen, können Sie die (in Kapitel 9 vorgestellte) niederschwellige Python-API von TensorFlow nutzen. In diesem Abschnitt werden wir mit dieser API das obige Modell nachbauen, das Mini-Batch-Gradientenverfahren implementieren und auf dem MNIST-Datensatz trainieren. Der erste Schritt ist die Konstruktionsphase, in der wir den TensorFlow-Graphen aufbauen. Der zweite Schritt ist die Ausführungsphase, in der Sie den Graphen zum Trainieren des Modells ausführen.

Konstruktionsphase

Fangen wir an. Zuerst müssen wir die Bibliothek tensorflow importieren. Anschließend gilt es, die Anzahl der Ein- und Ausgabewerte zu definieren und die Anzahl verborgener Neuronen in jeder Schicht festzulegen:

```
import tensorflow as tf

n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Nun können wir wie in Kapitel 9 Platzhalter-Knoten verwenden, um die Trainingsdaten und die Zielgröße zu repräsentieren. Die Abmessungen von X sind dabei nur teilweise definiert. Wir wissen, dass es ein 2-D-Tensor (also eine Matrix) sein wird, wobei die Datenpunkte entlang der ersten Dimension und die Merkmale entlang der zweiten Dimension liegen. Wir wissen auch, dass es 28 x 28 Merkmale gibt (ein Merkmal pro Pixel). Wir wissen aber noch nicht, wie viele Datenpunkte jeder Batch zum Trainieren enthält, und setzen daher die Abmessungen von X auf (None, n_inputs). Analog dazu wissen wir, dass y ein 1-D-Tensor mit einem Eintrag pro Datenpunkt sein wird, aber auch hier kennen wir die genaue Anzahl der Datenpunkte noch nicht. Daher setzen wir die Abmessung auf (None).

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Erstellen wir nun das eigentliche neuronale Netz. Der Platzhalter X wird als Eingabeschicht verwendet; während der Ausführungsphase wird er durch jeweils einem Batch Trainingsdaten ersetzt (beachten Sie, dass das neuronale Netz alle Datenpunkte eines Batches simultan verarbeitet). Nun müssen Sie die zwei verborgenen Schichten und die Ausgabeschicht erstellen. Die zwei verborgenen Schichten sind beinahe identisch: Sie unterscheiden sich nur in den mit ihnen verbundenen Eingaben und in der Anzahl enthaltener Neuronen. Die Ausgabeschicht ist ebenfalls sehr ähnlich, verwendet statt ReLU aber Softmax als Aktivierungsfunktion. Schreiben wir also eine Funktion `neuron_layer()`, mit der wir eine Schicht nach der anderen erstellen können. Sie benötigt Parameter zum Spezifizieren der Eingaben, der Anzahl Neuronen, der Aktivierungsfunktion und den Namen der Schicht:

```

def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs + n_neurons)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z

```

Gehen wir dieses Codebeispiel Zeile für Zeile durch:

1. Zuerst erstellen wir einen Scope mit dem Namen der Schicht: Dieser enthält sämtliche Knoten einer Schicht. Dieser Schritt ist optional, aber der Graph sieht damit in TensorBoard deutlich aufgeräumter aus.
2. Anschließend beschaffen wir die Anzahl der Eingabewerte, indem wir die Abmessungen der Eingabematrix nehmen und die Größe der zweiten Dimension verwenden (die erste Dimension ist die Anzahl der Datenpunkte).
3. Die nächsten drei Zeilen erstellen die Variable W , in der wir die Gewichtsmatrix speichern (auch als *Kernel* einer Schicht bezeichnet). Diese ist ein 2-D-Tensor mit den Gewichten aller Verbindungen zwischen jeder Eingabe und jedem Neuron; daher sind die Abmessungen der Matrix (n_{inputs} , n_{neurons}). Sie wird mit Zufallswerten initialisiert, die einer gestutzten Normalverteilung¹⁰ (Gaußverteilung) mit einer Standardabweichung von $2 / \sqrt{n_{\text{inputs}} + n_{\text{neurons}}}$ folgen. Diese Standardabweichung hilft dem Algorithmus dabei, schneller zu konvergieren (wir werden das in Kapitel 11 genauer besprechen; dies ist eine dieser kleinen Änderungen, die einen gewaltigen Einfluss auf die Effizienz neuronaler Netze hatten). Es ist wichtig, die Gewichte aller verborgenen Schichten mit Zufallszahlen zu initialisieren, um Symmetrien zu vermeiden, die das Gradientenverfahren nicht auflösen könnte.¹¹
4. Die nächste Zeile erstellt die Variable b für das Bias, die mit 0 initialisiert wird (hier gibt es kein Symmetrieproblem), sodass es pro Neuron einen Bias-Parameter gibt.
5. Dann erstellen wir einen Subgraphen, um $Z = X \cdot W + b$ zu berechnen. Die vektorisierte Implementierung berechnet effizienterweise die gewichteten

¹⁰ Die gestutzte anstelle der gewöhnlichen Normalverteilung zu verwenden, sorgt dafür, dass es keine sehr großen Gewichte gibt, die das Training verlangsamen können.

¹¹ Wenn Sie beispielsweise alle Gewichte auf 0 setzen, geben sämtliche Neuronen 0 aus, und der Fehlergradient ist für alle Neuronen einer Schicht der gleiche. Das Gradientenverfahren verändert dann alle Gewichte in genau der gleichen Weise, sodass sie weiter gleich bleiben. Anders ausgedrückt, würde sich das Modell wie ein einziges Neuron pro Schicht verhalten, selbst wenn jede Schicht aus Hunderten Neuronen bestünde.

Summen aller Eingaben und des Bias-Terms für jedes Neuron einer Schicht und für alle Datenpunkte eines Batches zeitgleich.

6. Schließlich gibt der Code `activation(Z)` zurück, falls der Parameter `activation` angegeben wurde, etwa mit `tf.nn.relu` (d.h. `max(0, Z)`), andernfalls wird einfach nur `Z` zurückgegeben.

Sie haben nun eine Funktion zum Erstellen einer Schicht Neuronen. Erstellen wir damit ein Deep-Learning-Netz! Die erste Schicht erhält `X` als Eingabe. Die zweite erhält die Ausgabe der ersten verborgenen Schicht als Eingabe. Schließlich erhält die Ausgabeschicht die Ausgabe der zweiten verborgenen Schicht als Eingabe.

```
with tf.name_scope("dnn"):  
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",  
                           activation=tf.nn.relu)  
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",  
                           activation=tf.nn.relu)  
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

Beachten Sie, dass wir wieder einmal um der Namensklarheit willen einen Scope verwenden. Außerdem ist `logits` die Ausgabe des neuronalen Netzes, *bevor* es die Softmax-Aktivierungsfunktion durchläuft: Aus Gründen der Optimierung werden wir Softmax später berechnen.

Wie Sie sich denken können, enthält TensorFlow viele nützliche Funktionen zum Erstellen von Standard-Schichten neuronaler Netze. Daher müssen Sie meist nicht Ihre eigene `neuron_layer()`-Funktion schreiben, wie wir es soeben getan haben. Beispielsweise erstellt die TensorFlow-Funktion `tf.layers.dense()` (vormals `tf.contrib.layers.fully_connected()`) eine vollständig verbundene Schicht, in der sämtliche Eingaben mit allen Neuronen in der Schicht verbunden sind. Sie kümmert sich um das Erstellen der Gewichte und Bias-Variablen unter den Namen `kernel` und `bias` sowie um eine angemessene Strategie zum Initialisieren der Werte. Über den Parameter `activation` legen Sie die Aktivierungsfunktion fest. Wie wir in Kapitel 11 sehen werden, unterstützt sie auch Regularisierungsparameter. Verändern wir den obigen Code so, dass er die Funktion `dense()` anstelle unserer Funktion `neuron_layer()` verwendet. Ersetzen Sie einfach den Abschnitt zum Erstellen von `dnn` durch folgenden Code:

```
with tf.name_scope("dnn"):  
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",  
                           activation=tf.nn.relu)  
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",  
                           activation=tf.nn.relu)  
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Unser neuronales Netz ist nun vollständig. Wir müssen anschließend eine Kostenfunktion definieren, um es zu trainieren. Wir verwenden dazu die Kreuzentropie wie bei der Softmax-Regression in Kapitel 4. Wie bereits erwähnt, bestraft die Kreuzentropie Modelle mit einer geringen Wahrscheinlichkeit für die Zielkategorie. TensorFlow enthält mehrere Funktionen zum Berechnen der Kreuzentropie. Wir

verwenden hier `sparse_softmax_cross_entropy_with_logits()`: Sie berechnet die Kreuzentropie aus den »logits« (d.h. der Ausgabe des Netzes vor der Softmax-Aktivierungsfunktion) und benötigt Labels als Integerzahlen von 0 bis zur Anzahl Kategorien minus 1 (in unserem Fall von 0 bis 9). Dabei erhalten wir einen 1-D-Tensor mit der Kreuzentropie für jeden Datenpunkt. Wir können dann die mittlere Kreuzentropie über sämtliche Datenpunkte mit der TensorFlow-Funktion `reduce_mean()` berechnen.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                               logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```



Die Funktion `sparse_softmax_cross_entropy_with_logits()` verwendet einfach die Softmax-Aktivierungsfunktion an und berechnet anschließend die Kreuzentropie, ist aber effizienter und behandelt Grenzfälle wie logits von 0 korrekt. Deshalb haben wir oben die Softmax-Aktivierungsfunktion nicht explizit verwendet. Es gibt eine weitere Funktion namens `softmax_cross_entropy_with_logits()`, die Labels als One-Hot-Vektoren annimmt (anstelle der Integerzahlen von 0 bis der Klassenanzahl minus 1).

Wir haben das neuronale Netz als Modell, wir haben die Kostenfunktion, nun benötigen wir noch einen `GradientDescentOptimizer`, der die Modellparameter verändert, um die Kostenfunktion zu minimieren. Dabei gibt es nichts Neues; wir tun das Gleiche wie in Kapitel 9:

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

Der letzte wichtige Schritt der Konstruktionsphase ist, anzugeben, wie das Modell evaluiert werden soll. Wir werden als Qualitätsmaß einfach die Genauigkeit verwenden. Zuerst bestimmen wir für jeden Datenpunkt, ob die Vorhersage des neuronalen Netzes korrekt ist, indem wir den höchsten logit-Wert mit der Zielkategorie vergleichen. Dazu können Sie die Funktion `in_top_k()` verwenden. Diese liefert uns einen 1-D-Tensor mit booleschen Werten, die wir in Floats umwandeln und daraus den Mittelwert bestimmen. Damit erhalten wir die Genauigkeit des gesamten Netzes.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Wie üblich müssen wir einen Knoten zum Initialisieren sämtlicher Variablen erstellen. Wir erstellen auch einen Saver, um die trainierten Parameter in einer Datei zu sichern:

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Geschafft! Damit beenden wir die Konstruktionsphase. Insgesamt waren es weniger als 40 Codezeilen, der Vorgang war dennoch sehr intensiv: Wir haben Platzhalter für Eingabedaten und Zielgröße erstellt, eine Funktion zum Erstellen einer Schicht Neuronen geschrieben, mit dieser Funktion ein DNN generiert, die Kostenfunktion definiert, einen Optimierer erstellt und schließlich das Qualitätsmaß definiert. Weiter geht es mit der Ausführungsphase.

Ausführungsphase

Dieser Teil ist viel kürzer und einfacher. Zuerst laden wir den MNIST-Datensatz. Wir könnten dazu wie in den vorigen Kapiteln Scikit-Learn verwenden, aber TensorFlow bietet zu diesem Zweck eine eigene Hilfsfunktionen an, die die Daten beschafft, skaliert (zwischen 0 und 1), durchmischt und eine einfache Funktion zum Laden eines einzelnen Mini-Batches anbietet. Verwenden wir also statt SciKit-Learn die Hilfefunktion von TensorFlow:

```
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("/tmp/data/")
```

Nun definieren wir die Anzahl auszuführender Epochen und die Größe der Mini-Batches:

```
n_epochs = 40  
batch_size = 50
```

Damit sind wir bereit, das Modell zu trainieren:

```
with tf.Session() as sess:  
    init.run()  
    for epoch in range(n_epochs):  
        for iteration in range(mnist.train.num_examples // batch_size):  
            X_batch, y_batch = mnist.train.next_batch(batch_size)  
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})  
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})  
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,  
                                            y: mnist.test.labels})  
        print(epoch, "Genauigkeit Training:", acc_train, "Genauigkeit Test:", acc_test)  
  
    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Dieser Code öffnet eine TensorFlow-Session und führt den init-Knoten aus, der sämtliche Variablen initialisiert. Anschließend führt er die Hauptschleife zum Trainieren aus: In jeder Epoche iteriert der Code über eine Anzahl Mini-Batches, die der Größe des Trainingsdatensatzes entspricht. Jeder Mini-Batch wird über die Methode `next_batch()` geholt, anschließend führt der Code einfach den Trainingsvorgang aus, indem er die Eingabedaten und Zielwerte des aktuellen Mini-Batches einspeist. Am Ende jeder Epoche wertet der Code das Modell mit dem letzten Mini-Batch und dem gesamten Testdatensatz aus und gibt das Ergebnis aus. Am Ende werden die Modellparameter in eine Datei geschrieben.

Verwenden des neuronalen Netzes

Das fertig trainierte neuronale Netz lässt sich nun für Vorhersagen einsetzen. Dazu können Sie die Konstruktionsphase wiederverwenden, aber die Ausführungsphase ändert sich folgendermaßen:

```
with tf.Session() as sess:  
    saver.restore(sess, "./my_model_final.ckpt")  
    X_new_scaled = [...] # einige neue Bilder (auf 0 bis 1 skaliert)  
    Z = logits.eval(feed_dict={X: X_new_scaled})  
    y_pred = np.argmax(Z, axis=1)
```

Zuerst lädt der Code die Modellparameter aus der Datei. Dann lädt er einige zu klassifizierende Bilder. Denken Sie daran, die gleiche Skalierung von Merkmalen wie bei den Trainingsdaten zu verwenden (in diesem Falle skalieren wir von 0 bis 1). Dann wertet der Code den Knoten `logits` aus. Wenn Sie die geschätzten Wahrscheinlichkeiten aller Kategorien wissen möchten, müssten Sie die Funktion `softmax()` auf die `logits` anwenden. Wenn Sie aber lediglich eine Kategorie vorhersagen möchten, wählen Sie einfach die Kategorie mit dem höchsten logit-Wert aus (dies erledigt die Funktion `argmax()`).

Feinabstimmung der Hyperparameter eines neuronalen Netzes

Die Flexibilität neuronaler Netze ist auch einer ihrer Hauptnachteile: Es gibt viele Hyperparameter, an denen Sie drehen können. Sie können jede erdenkliche *Netzwerktopologie* (wie die Neuronen miteinander verbunden sind) verwenden. Sogar in einem einfachen MLP können Sie die Anzahl der Schichten, die Anzahl der Neuronen pro Schicht, die Aktivierungsfunktion in jeder Schicht, die Strategie zur Initialisierung der Gewichte und vieles mehr verändern. Woher sollen Sie wissen, welche Kombination von Hyperparametern für Ihre Aufgabe geeignet ist?

Natürlich können Sie wie in den vorigen Kapiteln eine Gittersuche mit Kreuzvalidierung einsetzen, um die richtigen Hyperparameter zu finden. Weil es aber viele veränderbare Hyperparameter gibt und das Trainieren eines neuronalen Netzes auf einem großen Datensatz lange dauert, können Sie auf diese Weise nur einen sehr kleinen Teil des Hyperparameterraums in sinnvoller Zeit durchsuchen. Es ist viel günstiger, eine Zufallssuche (<https://goo.gl/QFjMKu>) zu verwenden, wie in Kapitel 2 erwähnt. Eine weitere Alternative ist ein Tool wie *Oscar* (<http://oscar.caldeck.ai/>), das komplexere Algorithmen bereitstellt, die Ihnen beim schnellen Finden guter Hyperparameter behilflich sind.

Eine grobe Vorstellung davon zu haben, welche Werte für jeden Hyperparameter sinnvoll sind, ist hilfreich, wenn Sie den Suchraum verkleinern möchten. Beginnen wir mit der Anzahl verborgener Schichten.

Anzahl verborgener Schichten

Bei vielen Aufgaben können Sie mit einer einzelnen verborgenen Schicht beginnen und passable Ergebnisse erhalten. Es wurde gezeigt, dass ein MLP mit nur einer verborgenen Schicht auch die komplexesten Funktionen modellieren kann, genug Neuronen vorausgesetzt. Für eine lange Zeit waren Wissenschaftler deshalb überzeugt, dass man tiefere neuronale Netze nicht untersuchen müsse. Sie übersahen aber, dass tiefere Netze eine weitaus höhere *Parametereffizienz* als flache besitzen: Sie können komplexe Funktionen mit exponentiell weniger Neuronen als flache Netze modellieren, wodurch sie sich schneller trainieren lassen.

Um dies nachzuvollziehen, nehmen Sie einmal an, Sie sollten mit einem Zeichenprogramm einen Wald zeichnen, ohne jedoch Copy-and-paste zu verwenden. Sie müssten jeden Baum einzeln zeichnen, Ast für Ast, Blatt für Blatt. Wenn Sie stattdessen ein Blatt zeichnen, es mehrfach kopieren und einen Ast erhalten, diesen mehrfach kopieren und einen Baum erzeugen und schließlich diesen Baum mehrfach kopieren, sodass ein Wald entsteht, wären Sie innerhalb kurzer Zeit fertig. Reale Daten sind häufig hierarchisch strukturiert, und DNNs machen sich diesen Umstand zunutze: Die ersten verborgenen Schichten modellieren einfache Strukturen (z.B. Linien mit unterschiedlicher Form und Orientierung), die verborgenen Schichten in der Mitte kombinieren diese Grundstrukturen zu Zwischenstrukturen (z.B. Quadrate und Kreise), und die letzten verborgenen Schichten sowie die Ausgabeschicht kombinieren diese Strukturen, um komplexe Strukturen zu modellieren (z.B. Gesichter).

Diese hierarchische Architektur sorgt nicht nur dafür, dass DNNs schneller zu einer annehmbaren Lösung konvergieren, sie erhöht auch deren Fähigkeit zur Verallgemeinerung. Wenn Sie beispielsweise bereits ein Modell zur Gesichtserkennung in Bildern trainiert haben und nun ein neues neuronales Netz zum Erkennen von Frisuren trainieren möchten, können Sie die ersten Schichten des ersten Netzes wiederverwenden. Anstatt die Gewichte und Bias der ersten Schichten zufällig zu initialisieren, können Sie die Gewichte und Bias des bestehenden Netzes einsetzen. Dadurch muss das Netz nicht sämtliche kleinteiligen Strukturen, die in den meisten Bildern vorkommen, von Neuem erlernen; es muss nur noch die komplexeren Strukturen erlernen (z.B. Frisuren).

Zusammengefasst funktionieren eine oder zwei verborgene Schichten bei den meisten Aufgaben am Anfang gut (z.B. können Sie mit einer Schicht und einigen Hundert Neuronen leicht eine Genauigkeit von 97% auf dem MNIST-Datensatz erhalten, und über 98% in etwa der gleichen Zeit, wenn Sie die gleiche Anzahl Neuronen auf zwei verborgene Schichten verteilen). Bei komplexeren Aufgaben können Sie die Anzahl der verborgenen Schichten schrittweise erhöhen, bis Sie die Trainingsdaten overfitten. Bei sehr komplexen Aufgaben wie der Klassifikation von Bildern oder Spracherkennung sind meist Netze mit Dutzenden Schichten (oder Hunderten, die aber nicht vollständig verbunden sind, wie wir noch in Kapitel 13

sehen werden) und riesige Datenmengen zum Trainieren nötig. Allerdings müssen Sie solche Netze selten von Anfang an trainieren: Es ist sehr verbreitet, Teile eines erstklassigen vortrainierten Netzes für eine ähnliche Aufgabe wiederzuverwenden. Das Trainieren ist so deutlich schneller und benötigt viel weniger Daten (dies werden wir in Kapitel 11 besprechen).

Anzahl Neuronen pro verborgene Schicht

Offensichtlich hängt die Anzahl der Neuronen in den Ein- und Ausgabeschichten von der für Ihre Aufgabe benötigten Ein- und Ausgabe ab. Beispielsweise sind für die MNIST-Aufgabe $28 \times 28 = 784$ Eingabeneuronen und 10 Ausgabeneuronen erforderlich. Bei den verborgenen Schichten ist es üblich, diese trichterförmig aufzubauen, sodass jede folgende Schicht immer weniger Neuronen enthält – der Grund hierfür ist, dass viele einfache Merkmale sich zu deutlich weniger komplexen Merkmalen entwickeln können. Ein typisches neuronales Netz für MNIST könnte aus zwei verborgenen Schichten mit jeweils 300 und 100 Neuronen bestehen. Allerdings ist dies keine allgemeine Praxis, und Sie könnten auch für alle verborgenen Schichten die gleiche Größe einstellen – z.B. 150 Neuronen pro verborgene Schicht: Damit müssen Sie nur einen Hyperparameter anstatt einen pro Schicht optimieren. Wie die Anzahl der Schichten können Sie auch die Anzahl Neuronen nach und nach erhöhen, bis das Netz overfittet. Im Allgemeinen entwickelt das Erhöhen der Anzahl Schichten mehr Durchschlagskraft als das Erhöhen der Anzahl Neuronen pro Schicht. Wie hieraus ersichtlich wird, ist beim Finden der perfekten Anzahl Neuronen leider ein wenig schwarze Kunst im Spiel.

Ein einfacherer Ansatz ist, ein Modell mit mehr Schichten und Neuronen als tatsächlich benötigt zu wählen und durch Early Stopping die Gefahr von Overfitting zu verhindern (ebenso eignen sich andere Regularisierungstechniken, besonders *Drop-out*, das wir in Kapitel 11 näher betrachten). Dieser Ansatz wurde »Stretch Pants« getauft:¹². Anstatt viel Zeit mit der Suche nach der perfekt sitzenden Hose zu vergeuden, verwenden Sie eine große elastische Hose, die automatisch auf die richtige Größe zusammenschrumpft.

Aktivierungsfunktionen

In den meisten Fällen können Sie ReLU als Aktivierungsfunktion in den verborgenen Schichten einsetzen (oder eine ihrer Varianten aus Kapitel 11). Sie lässt sich etwas schneller als andere Aktivierungsfunktionen berechnen, und die Gradientenmethode hält sich nicht lange auf Plateaus auf, da die Funktion bei hohen Eingabewerten keine Sättigung erreicht (im Gegensatz zur logistischen Funktion oder dem Tangens hyperbolicus, der sich 1 annähert).

12 Von Vincent Vanhoucke in seinem Deep Learning-Kurs (<https://goo.gl/Y5TFqz>) auf Udacity.com.

In der Ausgabeschicht ist die Softmax-Aktivierungsfunktion im Allgemeinen eine gute Wahl für Klassifikationsaufgaben (wenn sich die Kategorien gegenseitig ausschließen). Bei Regressionsaufgaben brauchen Sie überhaupt keine Aktivierungsfunktion zu verwenden.

Damit beenden wir unsere Einführung in künstliche neuronale Netze. In den folgenden Kapiteln werden wir Techniken zum Trainieren von Deep-Learning-Netzen besprechen und das Training auf mehrere Server und GPUs verteilen. Anschließend werden wir einige weitere beliebte Architekturen neuronaler Netze kennenlernen: Convolutional Neural Networks, rekurrente neuronale Netze und Autoencoder.¹³

Übungen

1. Zeichnen Sie ein ANN mit den ursprünglichen künstlichen Neuronen (ähnlich denen in Abbildung 10-3), das $A \oplus B$ ausrechnet (wobei \oplus für das logische XOR steht). Hinweis: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
2. Warum ist eine logistische Regression dem klassischen Perzeptron (einer einzelnen Schicht von Linear Threshold Units, die mit dem Perzeptron-Trainingsverfahren trainiert werden) grundsätzlich vorzuziehen? Wie können Sie ein Perzeptron so verändern, dass es einer Klassifikation mittels logistischer Regression gleichwertig ist?
3. Warum war die logistische Aktivierungsfunktion eine Schlüsselkomponente beim Trainieren der ersten MLPs?
4. Nennen Sie drei verbreitete Aktivierungsfunktionen. Können Sie diese zeichnen?
5. Angenommen Sie haben ein MLP mit einer Eingabeschicht aus 10 Eingabeneuronen, gefolgt von einer verborgenen Schicht mit 50 künstlichen Neuronen und schließlich eine Ausgabeschicht mit 3 künstlichen Neuronen. Sämtliche Neuronen verwenden ReLU als Aktivierungsfunktion.
 - Welche Abmessungen hat die Eingabematrix \mathbf{X} ?
 - Welche Abmessungen haben der Gewichtsvektor der verborgenen Schicht \mathbf{W}_h und ihr Bias-Vektor \mathbf{b}_h ?
 - Welche Abmessungen haben der Gewichtsvektor der Ausgabeschicht \mathbf{W}_o und ihr Bias-Vektor \mathbf{b}_o ?
 - Welche Abmessungen hat die Ausgabematrix des Netzes \mathbf{Y} ?
 - Schreiben Sie die Gleichung zum Berechnen der Ausgabematrix \mathbf{Y} als Funktion von $\mathbf{X}, \mathbf{W}_h, \mathbf{b}_h, \mathbf{W}_o$ und \mathbf{b}_o .

¹³ Einige zusätzliche Architekturen von ANNs werden in Anhang E vorgestellt.

6. Wie viele Neuronen benötigen Sie in der Ausgabeschicht, wenn Sie E-Mails als Spam oder Ham klassifizieren möchten? Welche Aktivierungsfunktion sollten Sie in der Ausgabeschicht verwenden? Wenn Sie stattdessen das MNIST-Problem lösen möchten, wie viele Neuronen benötigen Sie dann in der Ausgabeschicht und welche Aktivierungsfunktion? Beantworten Sie die gleiche Frage für die Vorhersage von Immobilienpreisen aus Kapitel 2.
7. Was ist das Backpropagation-Verfahren und wie funktioniert es? Was ist der Unterschied zwischen Backpropagation und Autodiff im Reverse-Modus?
8. Können Sie alle Hyperparameter aufzählen, die Sie in einem MLP verändern können? Wenn das MLP die Trainingsdaten overfittet, wie könnten Sie diese Hyperparameter verändern, um das Problem zu beheben?
9. Trainieren Sie ein Deep-Learning-MLP auf dem MNIST-Datensatz und schauen Sie, ob Sie eine Relevanz von mehr als 98% erzielen. Versuchen Sie wie in der letzten Übung von Kapitel 9, alle Register zu ziehen (den Zwischenstand zu speichern und zu laden, Zusammenfassungen auszugeben, Lernkurven mit TensorBoard zu zeichnen und so weiter).

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

Trainieren von Deep-Learning-Netzen

In Kapitel 10 haben wir künstliche neuronale Netze besprochen und unser erstes Deep-Learning-Netz trainiert. Es war aber ein sehr flaches DNN mit nur zwei verborgenen Schichten. Wie lässt sich eine sehr komplexe Aufgabe wie das Erkennen Hunderter Gegenstände in hochauflösenden Bildern angehen? Dazu müssen Sie ein weitaus tieferes DNN mit (z.B.) 10 Schichten aus jeweils Hunderten Neuronen und Hunderttausenden Verbindungen trainieren. Das wird kein Spaziergang:

- Erstens tritt bei Deep-Learning-Netzen das Problem der *schwindenden Gradienten* auf (oder das verwandte Problem der *explodierenden Gradienten*), was das Trainieren der ersten Schichten erheblich erschwert.
- Zweitens wird das Trainieren bei einem derart großen Netz extrem langsam.
- Drittens besteht bei einem Modell mit Millionen Parametern eine ernste Gefahr, die Trainingsdaten zu overfitten.

In diesem Kapitel wenden wir uns nacheinander jedem dieser Probleme zu und stellen Techniken zu deren Lösung vor. Wir beginnen mit dem Problem schwindender Gradienten und probieren einige der beliebtesten Lösungsstrategien aus. Als Nächstes werden wir unterschiedlicher Optimierer betrachten, die bei großen Modellen den Trainingsvorgang im Vergleich zum gewöhnlichen Gradientenverfahren erheblich beschleunigen. Schließlich werden wir einige bei großen neuronalen Netzen verbreitete Regularisierungstechniken behandeln.

Mit diesen Werkzeugen werden Sie in der Lage sein, sehr tiefe Netze zu trainieren: Willkommen beim Deep Learning!

Das Problem schwindender/explodierender Gradienten

Wie in Kapitel 10 besprochen, arbeitet sich der Backpropagation-Algorithmus von der Ausgabeschicht zur Eingabeschicht vor und berechnet unterwegs den Fehlergradienten. Hat der Algorithmus erst einmal den Fehlergradienten der Kostenfunk-

tion nach jedem Parameter im Netz bestimmt, werden diese Gradienten zum Aktualisieren jedes Parameters im Netz verwendet.

Leider werden die Gradienten mit diesem Algorithmus zu den niedrigeren Schichten hin immer kleiner und kleiner. In der Folge ändert das Gradientenverfahren die Gewichte der Verbindungen in den ersten Schichten kaum, und das Training konvergiert nie zu einer annehmbaren Lösung. Dies bezeichnet man als das Problem der *schwindenden Gradienten*. In einigen Fällen kann auch das Gegenteil passieren: Die Gradienten werden größer und größer, sodass die Gewichte vieler Schichten eine extrem große Änderung erfahren und der Algorithmus divergiert. Dies bezeichnet man als das Problem der *explodierenden Gradienten*, das vor allem in rekurrenten neuronalen Netzen auftritt (siehe Kapitel 14). Allgemeiner ausgedrückt sind die Gradienten in Deep-Learning-Netzen instabil; die Lerngeschwindigkeiten unterschiedlicher Schichten weichen stark voneinander ab.

Obwohl dieses ungünstige Verhalten schon vor einer Weile beobachtet wurde (dies war einer der Gründe, aus dem man Deep-Learning-Netze für eine lange Zeit beiseitegelegt hatte), konnte es erst um 2010 deutlich besser verstanden werden. Ein Artikel mit dem Titel »*Understanding the Difficulty of Training Deep Feedforward Neural Networks*« (<http://goo.gl/1rhAef>) von Xavier Glorot und Yoshua Bengio¹ fand einige Hauptverdächtige, darunter die Kombination der beliebten logistischen Aktivierungsfunktion und die damals verbreitete zufällige Initialisierung der Gewichte, genauer die zufällige Initialisierung mit einer Normalverteilung mit dem Mittelwert 0 und einer Standardabweichung von 1.

Kurz, die Autoren wiesen nach, dass mit diesem Muster aus Aktivierungsfunktion und Initialisierung die Varianz der Ausgaben jeder Schicht höher als die Varianz der Eingaben wird. Beim Durchschreiten des Netzes erhöht sich die Varianz von Schicht zu Schicht, bis die Aktivierungsfunktion in den späteren Schichten gesättigt ist. Das Problem, dass die logistische Funktion einen Mittelwert von 0.5 anstatt 0 hat, wird dadurch verschlimmert (der Tangens hyperbolicus besitzt einen Mittelwert von 0 und verhält sich in Deep-Learning-Netzen etwas besser als die logistische Funktion).

Wenn Sie sich die logistische Aktivierungsfunktion ansehen (siehe Abbildung 11-1), erkennen Sie, dass die Funktion bei großen Eingabewerten (negativ oder positiv) eine Sättigung bei 0 oder 1 erreicht und ihre Ableitung sehr nah bei 0 liegt. Beim Backpropagation-Verfahren gibt es daher praktisch keinen Gradienten, der sich durch das Netzwerk propagieren ließe, und das bisschen Gradient wird in den späteren Schichten auch noch ausgedünnt. Es bleibt also für die ersten Schichten wirklich nichts übrig.

¹ »*Understanding the Difficulty of Training Deep Feedforward Neural Networks*«, X. Glorot, Y. Bengio (2010).

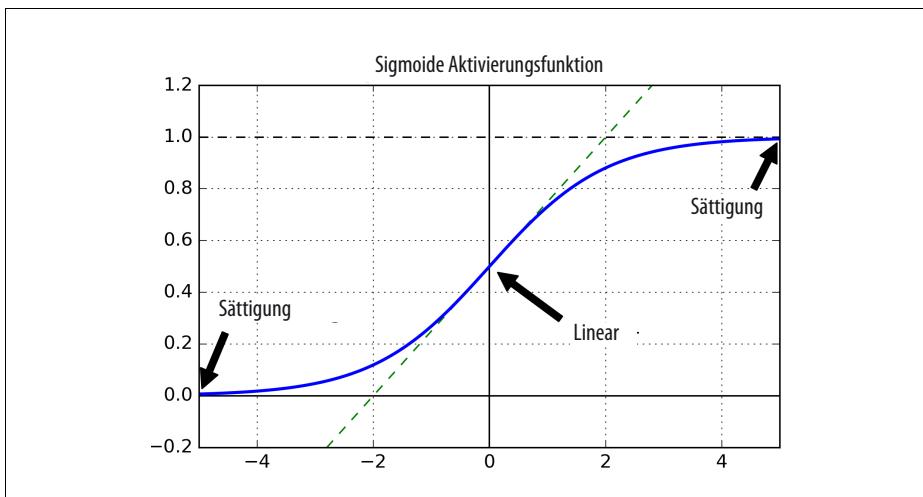


Abbildung 11-1: Sättigung der logistischen Aktivierungsfunktion

Initialisierung nach Xavier und He

In ihrem Artikel empfehlen Glorot und Bengio eine deutliche Verbesserung. Das Signal muss in beide Richtungen fließen können: bei der Vorhersage vorwärts und beim Propagieren der Gradienten rückwärts. Wir möchten weder, dass das Signal unterwegs verhungert, noch dass es explodiert und zu einer Sättigung führt. Die Autoren argumentieren, dass für einen guten Signalfluss die Varianz der Ausgaben und die der Eingaben jeder Schicht gleich sein müssen.² Außerdem müssen die Gradienten bei der Backpropagation vor und nach einer Schicht die gleiche Varianz besitzen (bitte lesen Sie den Artikel, falls Sie an den mathematischen Details interessiert sind). Beides ist nicht möglich, es sei denn, eine Schicht hat gleich viele eingehende und ausgehende Verbindungen, aber die Autoren schlugen einen praktisch gut funktionierenden Kompromiss vor: Die Gewichte der Verbindungen müssen zufällig gesetzt werden wie in Formel 11-1 beschrieben, wobei n_{Eingaben} und n_{Ausgaben} die Anzahl der Ein- und Ausgabeverbindungen der zu initialisierenden Schicht sind (auch *Fan-in* und *Fan-out* genannt). Diese Initialisierungsstrategie wird oft *Initialisierung nach Xavier* (nach dem Vornamen des Autors), bisweilen auch *Initialisierung nach Glorot*.

² Hier besteht eine Analogie: Wenn Sie den Verstärkerknopf eines Mikrofons zu nah an die Null stellen, hört niemand Ihre Stimme, aber wenn Sie ihn zu nah an das Maximum stellen, ist Ihre Stimme gesättigt, und niemand versteht, was Sie sagen. Stellen Sie sich nun eine Kette mehrerer solcher Verstärker vor: Alle müssen richtig eingestellt sein, damit Ihre Stimme am Ende der Kette laut und deutlich zu verstehen ist. Ihre Stimme muss aus jedem Verstärker mit der gleichen Amplitude herauskommen, mit der sie hereinkommt.

Formel 11-1: Initialisierung nach Xavier (beim Verwenden der logistischen Aktivierungsfunktion)

$$\text{Normalverteilung mit Mittelwert 0 und Standardabweichung } \sigma = \sqrt{\frac{2}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$$

$$\text{Oder eine Gleichverteilung zwischen } -r \text{ und } +r, \text{ mit } r = \sqrt{\frac{6}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$$

Wenn es etwa gleich viele Eingabe- wie Ausgabeverbindungen gibt, erhalten Sie einfache Gleichungen (z.B. $\sigma = 1 / \sqrt{n_{\text{Eingaben}}}$ oder $r = \sqrt{3} / \sqrt{n_{\text{Eingaben}}}$). Wir haben diese vereinfachte Strategie in Kapitel 10 verwendet.³

Die Initialisierung nach Xavier kann das Training erheblich beschleunigen und gehört zu den Tricks, die zum gegenwärtigen Erfolg von Deep Learning führten. Einige neuere Artikel (<http://goo.gl/VHP3pB>)⁴ haben ähnliche Strategien für andere Aktivierungsfunktionen entwickelt, wie Tabelle 11-1 zeigt. Die Initialisierungsstrategie für die ReLU-Aktivierungsfunktion (und ihre Varianten wie die in Kürze beschriebene ELU-Aktivierung) bezeichnet man bisweilen als *Initialisierung nach He* (nach dem Nachnamen ihres Autors).

Tabelle 11-1: Initialisierungsparameter für Aktivierungsfunktionen

Aktivierungsfunktion	Gleichverteilung $[-r, r]$	Normalverteilung
logistisch	$r = \sqrt{\frac{6}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$
Tangens hyperbolicus	$r = 4 \sqrt{\frac{6}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$	$\sigma = 4 \sqrt{\frac{2}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$
ReLU (und Varianten)	$r = \sqrt{2} \sqrt{\frac{6}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$	$\sigma = \sqrt{2} \sqrt{\frac{2}{n_{\text{Eingaben}} + n_{\text{Ausgaben}}}}$

Die Funktion `tf.layers.dense()` (vorgestellt in Kapitel 10) verwendet standardmäßig die Initialisierung nach Xavier (mit einer Gleichverteilung). Sie können diese mit `variance_scaling_initializer()` auf die Initialisierung nach He umstellen:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                         kernel_initializer=he_init, name="hidden1")
```

³ Diese vereinfachte Strategie wurde bereits viel früher vorgestellt – beispielsweise im Buch *Neural Networks: Tricks of the Trade* von Genevieve Orr und Klaus-Robert Müller (Springer) aus dem Jahr 1998.

⁴ [Beispielsweise »Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification«, K. He et al. (2015).]



Die Initialisierung nach He berücksichtigt lediglich das Fan-in, nicht den Durchschnitt von Fan-in und Fan-out wie die Initialisierung nach Xavier. Dies ist die Standardeinstellung für die Funktion `variance_scaling_initializer()`, Sie können sie aber über den Parameter `mode="FAN_AVG"` ändern.

Nicht sättigende Aktivierungsfunktionen

Eine Erkenntnis des Artikels von Glorot und Bengio aus dem Jahr 2010 war, dass die Auswahl einer ungeeigneten Aktivierungsfunktion eine Teilursache des Problems schwindender/explodierender Gradienten war. Bis dahin hatten die meisten Leute angenommen, dass sigmoide Aktivierungsfunktionen eine ausgezeichnete Wahl sein müssten, zumal Mutter Natur in etwa diese in biologischen Neuronen verwendet. Wie sich aber herausstellte, verhalten sich in Deep-Learning-Netzen andere Aktivierungsfunktionen viel günstiger, besonders die ReLU-Aktivierungsfunktion, vor allem, weil sie bei positiven Werten keine Sättigung erreicht (und weil sie sich schnell berechnen lässt).

Leider ist auch die ReLU-Aktivierungsfunktion nicht perfekt. Sie krankt an einem Problem, das als *sterbende ReLUs* bekannt ist: Beim Trainieren sterben einige Neuronen praktisch ab, d.h., sie geben nichts anderes als 0 aus. In einigen Fällen kommt es vor, dass die Hälfte der Neuronen im Netzwerk tot sind, besonders wenn Sie eine große Lernrate eingestellt haben. Wenn beim Trainieren die Gewichte eines Neurons so aktualisiert werden, dass die gewichtete Summe der Eingaben negativ ist, gibt dieses Neuron eine 0 aus. Es ist unwahrscheinlich, dass das entsprechende Neuron danach wieder zum Leben erwacht, weil der Gradient der ReLU-Funktion für negative Eingaben 0 beträgt.

Um dieses Problem zu lösen, können Sie eine Variante der ReLU-Funktion verwenden, z.B. *Leaky ReLU*. Diese Funktion ist definiert als $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (siehe Abbildung 11-2). Der Hyperparameter α definiert, wie stark die Funktion »leckt«: Er entspricht der Steigung der Funktion bei $z < 0$ und beträgt normalerweise 0.01. Diese geringe Steigung stellt sicher, dass Leaky ReLUs niemals sterben; sie können in ein langes Koma fallen, können aber wieder erwachen. Ein kürzlich erschienener Artikel (<https://goo.gl/B1xhKn>)⁵ verglich mehrere Varianten der ReLU-Aktivierungsfunktion und kam zu dem Schluss, dass die Leaky-Varianten der ursprünglichen ReLU-Aktivierungsfunktion stets überlegen sind. Das Setzen von $\alpha = 0.2$ (ein riesiges Leck) schien zu einer höheren Leistung als $\alpha = 0.01$ (ein kleines Leck) zu führen. Die Autoren werteten auch die *randomisierte Leaky ReLU* (RReLU) aus, bei der α beim Trainieren aus einem vorgegebenen Bereich zufällig ausgewählt wird und beim Testen auf einen Durchschnittswert gesetzt wird. Auch diese Funktion schneidet recht gut ab und schien als Regularisierung zu fungieren (das Risiko für Overfitting der Trainingsdaten zu senken). Schließlich wurde auch die *parametri-*

5 »Empirical Evaluation of Rectified Activations in Convolutional Network«, B. Xu et al. (2015).

sierte Leaky ReLU (PReLU) betrachtet, bei der α beim Training erlernt wird (diese PReLU ist kein Hyperparameter, sondern wird ein Parameter, der vom Backpropagation-Verfahren modifiziert werden kann wie jeder andere Parameter). Dieses Verfahren schnitt auf großen Bilddatensätzen sehr viel besser als ReLU ab, neigte bei kleineren Datensätzen aber zum Overfitting.

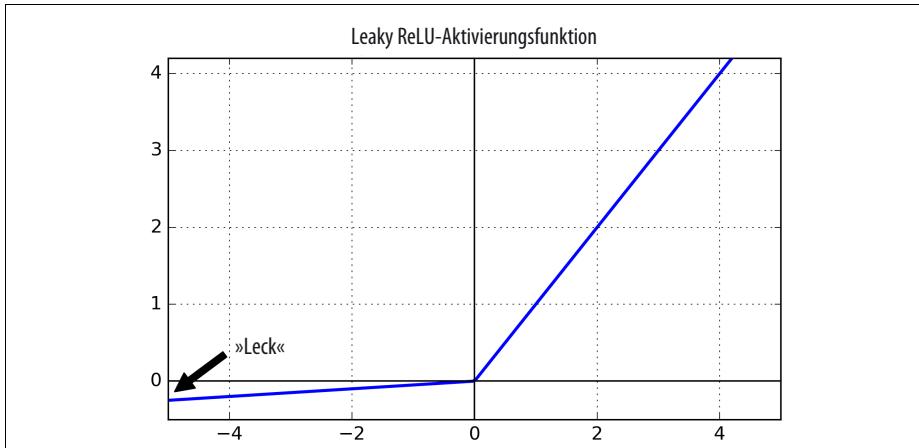


Abbildung 11-2: Leaky ReLU

Schließlich schlug ein Artikel aus dem Jahr 2015 (<http://goo.gl/Sdl2P7>) von Djork-Arné Clevert et al.⁶ eine neue Aktivierungsfunktion namens *Exponential Linear Unit* (ELU) vor, die im Experiment sämtliche ReLU-Varianten aus dem Felde schlug: Die Trainingszeit verringerte sich, und das neuronale Netz erzielte auf den Testdaten eine höhere Leistung. Die Funktion ist in Abbildung 11-3 dargestellt und ihre Definition in Formel 11-2 ausgeschrieben.

Formel 11-2: ELU Aktivierungsfunktion

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{wenn } z < 0 \\ z & \text{wenn } z \geq 0 \end{cases}$$

Sie sieht der ReLU-Funktion sehr ähnlich, weist aber einige große Unterschiede auf:

- Erstens nimmt sie bei $z < 0$ negative Werte an, wodurch das Neuron eine durchschnittliche Ausgabe um 0 haben kann. Dies bekämpft das Problem schwundender Gradienten. Der Hyperparameter α definiert den Wert, dem sich die ELU-Funktion bei stark negativen Werten von z annähert. Er wird normalerweise auf 1 gesetzt, aber Sie können ihn wie jeden anderen Hyperparameter verändern.

⁶ »Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)«, D. Clevert, T. Unterthiner, S. Hochreiter (2015).

- Zweitens ist die Ableitung für $z < 0$ ungleich 0, was das Problem sterbender Neuronen vermeidet.
- Drittens ist die Funktion an allen Stellen glatt, auch bei $z = 0$, was das Gradientenverfahren beschleunigt, da es links und rechts von $z = 0$ weniger umherspringt.

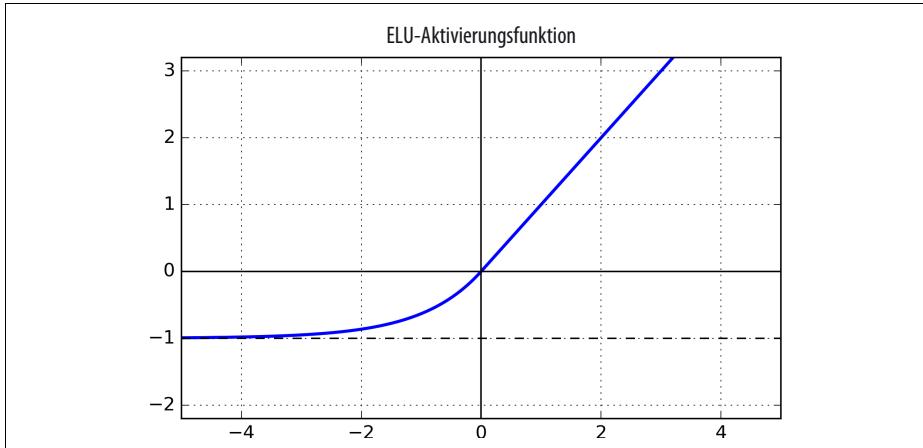


Abbildung 11-3: ELU Aktivierungsfunktion

Der Hauptnachteil der ELU-Aktivierungsfunktion ist, dass sie sich langsamer als ReLU und seine Varianten berechnen lässt (wegen der Exponentialfunktion), aber beim Trainieren wird dies durch die schnellere Konvergenz kompensiert. Beim Testen ist ein ELU-Netz jedoch langsamer als ein ReLU-Netz.



Welche Aktivierungsfunktion sollten Sie also in den verborgenen Schichten Ihrer Deep-Learning-Netze verwenden? Auch wenn es im Einzelfall Unterschiede gibt, gilt als Faustregel $\text{ELU} > \text{Leaky ReLU}$ (und Varianten) $> \text{ReLU} > \tanh >$ logistisch. Wenn es Ihnen sehr auf Geschwindigkeit zur Laufzeit ankommt, sollten Sie Leaky ReLUs den Vorzug vor ELUs geben. Wenn Sie sich nicht mit noch einem Hyperparameter befassen möchten, können Sie die oben empfohlenen Standardwerte für α verwenden (0.01 für Leaky ReLU und 1 für ELU). Wenn Sie zusätzliche Zeit und Rechenkapazität übrig haben, können Sie weitere Aktivierungsfunktionen über Kreuzvalidierung mit einbeziehen, insbesondere RReLU, falls Ihr Netz zu Overfitting neigt, und PReLU, wenn Ihr Trainingsdatensatz sehr groß ist.

TensorFlow enthält die Funktion `elu()`, mit der Sie Ihr neuronales Netz aufbauen können. Setzen Sie einfach den Parameter `activation` beim Aufruf der Funktion `dense()` wie folgt:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

In TensorFlow gibt es keine vordefinierte Funktion für Leaky ReLUs, aber Sie können sie sich recht leicht definieren:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = tf.layers.dense(X, n_hidden1, activation=leaky_relu, name="hidden1")
```

Batch-Normalisierung

Obwohl die Initialisierung nach He zusammen mit der ELU (oder einer Variante der ReLU) das Problem der schwindenden/explodierenden Gradienten zu Beginn des Trainierens deutlich reduzieren kann, sind sie keine Garantie dafür, dass die Probleme nicht später zurückkehren.

In einem Artikel aus dem Jahr 2015 (<https://goo.gl/gA4GSP>)⁷ schlagen Sergey Ioffe und Christian Szegedy eine Technik namens *Batch-Normalisierung* (BN) vor, um das Problem schwindender/explodierender Gradienten anzugehen sowie das allgemeinere Problem der Verteilung von Änderungen der Eingaben beim Trainieren aufgrund von Änderungen der Parameter der vorigen Schichten (das die Autoren als *Internal Covariate Shift*-Problem bezeichnen).

Die Technik besteht aus dem Hinzufügen einer Operation kurz vor der Aktivierungsfunktion in jeder Schicht des Modells, die einfach die Eingaben auf null zentriert und normalisiert und das Ergebnis anschließend mit zwei neuen Parametern pro Schicht skaliert und verschiebt (ein Parameter zum Skalieren, einer zum Verschieben). Anders ausgedrückt, kann das Modell durch diesen Vorgang die optimale Skalierung und den Mittelwert für die Eingaben jeder Schicht erlernen.

Um die Eingaben auf null zu zentrieren und zu normalisieren, muss der Algorithmus Mittelwert und Standardabweichung der Eingaben schätzen. Dazu werden Mittelwert und Standardabweichung der Eingaben aus dem aktuellen Mini-Batch ausgewertet (daher der Name »Batch-Normalisierung«). Die gesamte Prozedur ist in Formel 11-3 zusammengefasst.

Formel 11-3: Algorithmus zur Batch-Normalisierung

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$

⁷ »Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift«, S. Ioffe and C. Szegedy (2015).

- μ_B ist der empirische Mittelwert für den gesamten Mini-Batch B .
- σ_B ist die empirische Standardabweichung, ebenfalls für den gesamten Mini-Batch bestimmt.
- m_B ist die Anzahl Datenpunkte im Mini-Batch.
- $\hat{\mathbf{x}}^{(i)}$ ist die auf null zentrierte und normalisierte Eingabe.
- γ ist der Parameter zum Skalieren der Schicht.
- β ist der Parameter zum Verschieben der Schicht (Offset).
- ϵ ist eine kleine Zahl, zum Vermeiden einer Division durch null (normalerweise 10^{-5}). Diese wird als *Smoothing-Term* bezeichnet.
- $\mathbf{z}^{(i)}$ ist die Ausgabe der BN-Operation: Sie ist eine skalierte und verschobene Version der Eingaben.

Beim Testen gibt es keine Mini-Batches, mit denen sich Mittelwert und Standardabweichung empirisch bestimmen ließen. Stattdessen können Sie einfach Mittelwert und Standardabweichung des gesamten Trainingsdatensatzes verwenden. Diese lassen sich beim Trainieren einfach als gleitender Durchschnitt bestimmen. Für jede Batch-normalisierte Schicht werden also vier Parameter bestimmt: γ (Skalierung), β (Offset), μ (Mittelwert) und σ (Standardabweichung).

Die Autoren konnten zeigen, dass diese Technik sämtliche untersuchten Deep-Learning-Netze erheblich verbesserte. Das Problem der schwindenden Gradienten verkleinerte sich so weit, dass Aktivierungsfunktionen mit Sättigung wie tanh und sogar die logistische Aktivierungsfunktion zum Einsatz kommen konnten. Die Netze reagierten weitaus weniger sensibel auf die Initialisierung der Gewichte. Es konnten auch wesentlich höhere Lernraten eingesetzt werden, was den Lernprozess beschleunigte. Die Autoren betonen: »Auf ein Modell zur Bildklassifikation erster Güte angewandt, erzielt die Batch-Normalisierung die gleiche Genauigkeit bei 14 Mal weniger Trainingsschritten und schlägt das ursprüngliche Modell mit signifikantem Abstand. [...] Mit einem Ensemble Batch-normalisierter Netze konnten wir das beste publizierte Ergebnis bei der ImageNet-Klassifikation verbessern: Ein Top-5-Validierungsfehler von 4.9% (und 4.8% im Testdatensatz) lässt die Genauigkeit menschlicher Klassifikation hinter sich.« Schließlich fungiert die Batch-Normalisierung einer unerschöpflichen Quelle gleich sogar als Regularisierung und macht andere Regularisierungstechniken (wie das später im Kapitel beschriebene Drop-out) unnötig.

Durch die Batch-Normalisierung wird das Modell allerdings komplexer (auch wenn die Eingabedaten nicht mehr normalisiert werden müssen, da die erste Batch-normalisierte verborgene Schicht sich nun darum kümmert). Es gibt außerdem einen Nachteil zur Laufzeit: Das neuronale Netz trifft seine Vorhersagen aufgrund der zusätzlichen Berechnungen in jeder Schicht langsamer. Wenn Sie also blitzschnelle Vorhersagen benötigen, sollten Sie prüfen, wie schnell ELU und Initialisierung nach He abschneiden, bevor Sie die Batch-Normalisierung in Betracht ziehen.



Sie mögen feststellen, dass das Trainieren zunächst langsam ist, während das Gradientenverfahren nach der optimalen Skalierung und den Offsets sucht, aber es beschleunigt sich, sobald halbwegs gute Werte gefunden wurden.

Implementieren der Batch-Normalisierung mit TensorFlow

TensorFlow enthält die Funktion `tf.nn.batch_normalization()`, die die Eingaben zentriert und normalisiert. Sie müssen aber Mittelwert und Standardabweichung selbst berechnen (wie erwähnt beim Trainieren anhand der Daten aus dem Mini-Batch, beim Testen aus dem gesamten Datensatz) und dieser Funktion als Parameter übergeben. Sie müssen auch den Skalierungsparameter und Offset selbst erstellen (und dieser Funktion übergeben). Dies ist machbar, aber nicht besonders komfortabel. Die Funktion `tf.layers.batch_normalization()` ist besser geeignet, sie nimmt Ihnen all diese Dinge ab, wie folgender Code zeigt:

```
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

training = tf.placeholder_with_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn, training=training,
                                       momentum=0.9)
```

Gehen wir diesen Code im Einzelnen durch. Die ersten Zeilen sind halbwegs selbsterklärend, bis wir den Platzhalter `training` definieren: Wir setzen diesen beim Trainieren auf `True`, ansonsten ist er auf `False` voreingestellt. Damit sagen wir der Funktion `tf.layers.batch_normalization()`, ob sie Mittelwert und Standardabweichung aus dem aktuellen Mini-Batch verwenden soll (beim Trainieren) oder Mittelwert und Standardabweichung des gesamten Trainingsdatensatzes (beim Testen).

Danach alternieren vollständig verbundene Schichten und Schichten mit Batch-Normalisierung: Die vollständig verbundenen Schichten werden mit der Funktion `tf.layers.dense()` wie in Kapitel 10 erstellt. Beachten Sie, dass wir bei den vollständig verbundenen Schichten keine Aktivierungsfunktion angeben, weil wir die Aktivierungsfunktion nach der jeweiligen Schicht zur Batch-Normalisierung anwenden

möchten.⁸ Wir erstellen die Schicht zur Batch-Normalisierung mit der Funktion `tf.layers.batch_normalization()` und setzen ihre Parameter `training` und `momentum`. Der BN-Algorithmus verwendet *exponentiellen Zerfall* zur Berechnung des gleitenden Durchschnitts, weswegen wir den Parameter `momentum` benötigen: Für einen neuen Wert v wird der gleitende Durchschnitt \hat{v} durch folgende Gleichung aktualisiert:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Ein guter Wert für `momentum` liegt normalerweise um 1 – beispielsweise 0.9, 0.99 oder 0.999 (bei großen Datensätzen und kleinen Mini-Batches sollten Sie zusätzliche 9en hinzufügen).

Sie haben vielleicht bemerkt, dass der Code einige Wiederholungen enthält und die gleichen Parameter zur Batch-Normalisierung wieder und wieder auftauchen. Um diese Wiederholungen zu vermeiden, können Sie die Funktion `partial()` aus dem Modul `functools` verwenden (in der Python-Standardbibliothek enthalten). Sie erstellt einen leichtgewichtigen Wrapper um eine Funktion und erlaubt das Festlegen von Standardwerten für einige Parameter. Das Erstellen der Schichten im Netz lässt sich damit folgendermaßen modifizieren:

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                             training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

Dieses kleine Beispiel sieht vielleicht nicht sehr viel besser aus als vorher, aber wenn Sie 10 Schichten haben und jeweils die gleiche Aktivierungsfunktion, den gleichen Initialisierer und Regularisierer und so weiter verwenden möchten, wird Ihr Code durch diesen Trick sehr viel lesbarer.

Die restliche Konstruktionsphase ist die gleiche wie in Kapitel 10: Sie definieren die Kostenfunktion, erstellen einen Optimierer, weisen ihn an, die Kostenfunktion zu minimieren, definieren die Operationen zum Evaluieren, erstellen einen Saver und so weiter.

Die Ausführungsphase ist ebenfalls im Wesentlichen die gleiche, es gibt aber zwei Ausnahmen. Erstens müssen Sie beim Trainieren immer dann, wenn eine Schicht

⁸ Viele Wissenschaftler sprechen sich dafür aus, die Batch-Normalisierung nach (anstatt vor) der Aktivierung zu platzieren.

von der `batch_normalization()`-Schicht abhängt, den Platzhalter `training` auf `True` setzen. Zweitens erstellt die Funktion `batch_normalization()` einige Operationen, die in jedem Trainingsschritt ausgewertet werden müssen, um den gleitenden Durchschnitt zu berechnen (dieser ist zur Berechnen von Mittelwert und Standardabweichung des Trainingsdatensatzes nötig). Diese Operationen werden automatisch zur Kollektion `UPDATE_OPS` hinzugefügt, daher müssen wir uns lediglich eine Liste aller Operationen in dieser Kollektion beschaffen und sie in jedem Trainingsschritt ausführen:

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run([training_op, extra_update_ops],
                    feed_dict={training: True, X: X_batch, y: y_batch})
            accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                    y: mnist.test.labels})
            print(epoch, "Test Genauigkeit:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Das ist alles! Es ist unwahrscheinlich, dass die Batch-Normalisierung in diesem winzigen Beispiel einen sonderlich positiven Einfluss hat, aber bei tieferen Netzen macht es einen gewaltigen Unterschied.

Gradient Clipping

Eine beliebte Technik, um das Problem der explodierenden Gradienten zu entschärfen, ist, die Gradienten während der Backpropagation einfach zu kappen, sodass sie niemals einen festgelegten Schwellenwert überschreiten (hilfreich vor allem bei rekurrenten neuronalen Netzen; siehe Kapitel 14). Man nennt diese Technik *Gradient Clipping* (<http://goo.gl/dRDAaf>).⁹ Die meisten Leute bevorzugen heutzutage die Batch-Normalisierung, aber es ist dennoch nützlich, über Gradient Clipping Bescheid zu wissen.

In TensorFlow kümmert sich die Funktion `minimize()` des Optimizers sowohl um das Berechnen des Gradienten als auch um seine Anwendung. Daher müssen Sie stattdessen zunächst die Methode `compute_gradients()` aufrufen, dann eine Operation zum Kappen des Gradienten mit der Funktion `clip_by_value()` und zum Schluss eine Operation zum Anwenden des gekappten Gradienten mit der Methode `apply_gradients()` des Optimizers erstellen:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
```

⁹ »On the difficulty of training recurrent neural networks«, R. Pascanu et al. (2013).

```

grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)

```

Anschließend führen Sie wie gewohnt `training_op` bei jedem Trainingsschritt aus. Dies berechnet die Gradienten, schneidet sie zwischen -1.0 und 1.0 ab und wendet diese an. Der Schwellenwert ist ein Hyperparameter, den Sie optimieren können.

Wiederverwenden vortrainierter Schichten

Es ist im Allgemeinen keine gute Idee, ein sehr großes DNN von Anfang an zu trainieren: Stattdessen sollten Sie stets versuchen, ein existierendes neuronales Netz zu finden, das eine ähnliche Aufgabe erledigt. Dann können Sie die ersten Schichten dieses Netzes wiederverwenden: Man bezeichnet dies als *Transfer Learning*, und es beschleunigt nicht nur das Trainieren erheblich, sondern erfordert auch wesentlich weniger Trainingsdaten.

Nehmen Sie beispielsweise an, Ihnen stünde ein DNN zur Verfügung, das zur Klassifizierung von Bildern in 100 unterschiedliche Kategorien wie Tiere, Pflanzen, Fahrzeuge und Alltagsgegenstände trainiert wurde. Sie möchten ein DNN trainieren, mit dem sich bestimmte Arten von Fahrzeugen klassifizieren lassen. Diese Aufgaben sind einander sehr ähnlich, daher sollten Sie Teile des ersten Netzes wiederverwenden (siehe Abbildung 11-4).

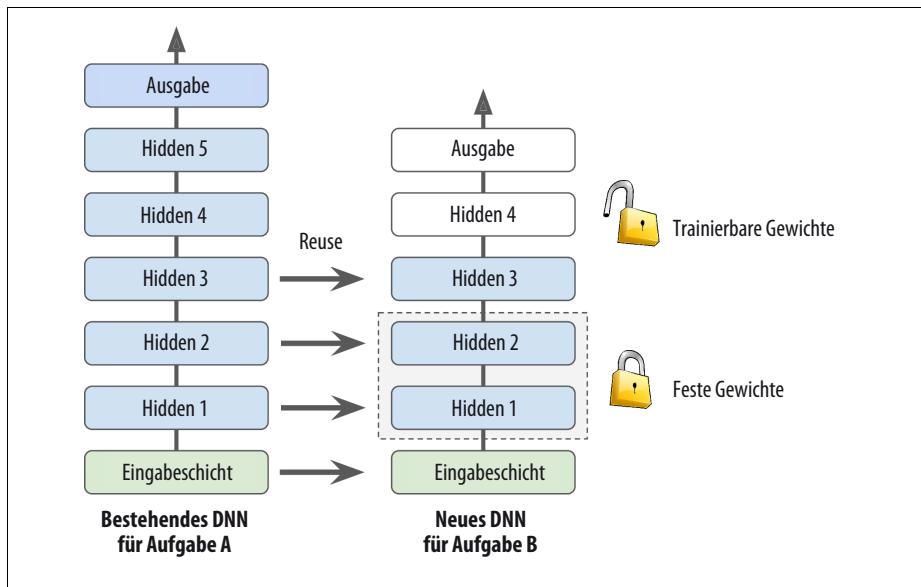


Abbildung 11-4: Wiederverwenden vortrainierter Schichten



Wenn die Bilder bei Ihrer neuen Aufgabe nicht die gleiche Größe haben wie die in der ursprünglichen Aufgabe verwendeten, müssen Sie einen Vorverarbeitungsschritt durchführen, der die Bilder auf die vom ursprünglichen Modell erwartete Größe skaliert. Im Allgemeinen funktioniert Transfer Learning nur, wenn die Eingabedaten auf niedriger Ebene ähnliche Eigenschaften haben.

Wiederverwenden eines TensorFlow-Modells

Wenn das ursprüngliche Modell mit TensorFlow trainiert wurde, können Sie es einfach wiederherstellen und auf der neuen Aufgabe trainieren. Wie in Kapitel 9 besprochen, lassen sich mit der Funktion `import_meta_graph()` Operationen in den Standardgraphen importieren. Sie erhalten einen Saver, den Sie später zum Laden des Modells verwenden können:

```
saver = tf.train.import_meta_graph("./my_model_final.ckpt.meta")
```

Sie benötigen dann ein Handle zu den Operationen und Tensoren zum Trainieren. Dabei helfen die Methoden `get_operation_by_name()` und `get_tensor_by_name()` des Graphen. Der Name eines Tensors ist der Name der Operation, die ihn ausgibt, gefolgt von einer `:0` (oder `:1` falls es die zweite Ausgabe ist, `:2` bei der dritten und so weiter):

```
X = tf.get_default_graph().get_tensor_by_name("X:0")
y = tf.get_default_graph().get_tensor_by_name("y:0")
accuracy = tf.get_default_graph().get_tensor_by_name("eval/accuracy:0")
training_op = tf.get_default_graph().get_operation_by_name("GradientDescent")
```

Falls das vortrainierte Modell nicht gut dokumentiert ist, müssen Sie den Graphen untersuchen, um die Namen der benötigten Operationen herauszubekommen. In diesem Fall sollten Sie entweder den Graphen mit TensorBoard betrachten (dazu müssen Sie ihn zuerst mit einem `FileWriter` exportieren, wie in Kapitel 9 beschrieben) oder über die Methode `get_operations()` sämtliche Operationen des Graphen ausgeben:

```
for op in tf.get_default_graph().get_operations():
    print(op.name)
```

Wenn Sie selbst der Autor des ursprünglichen Modells sind, können Sie Menschen, die Ihr Modell verwenden möchten, das Leben leichter machen, indem Sie Ihrem Modell sehr klare Namen verleihen und diese dokumentieren. Eine Alternative ist, eine Kollektion zu erstellen, in der sämtliche für andere eventuell wichtigen Operationen enthalten sind:

```
for op in (X, y, accuracy, training_op):
    tf.add_to_collection("meine_wichtigen_ops", op)
```

Auf diese Weise kann jemand, der Ihr Modell wiederverwenden möchte, einfach schreiben:

```
X, y, accuracy, training_op = tf.get_collection("meine_wichtigen_ops")
```

Sie können anschließend den Zustand des Modells mit dem Saver wiederherstellen und das Trainieren mit Ihren eigenen Daten fortsetzen.

```
with tf.Session() as sess:  
    saver.restore(sess, "./my_model_final.ckpt")  
    [...] # das Modell mit Ihren eigenen Daten trainieren
```

Als Alternative können Sie auch den Python-Code zum Aufbau des ursprünglichen Graphen anstelle von `import_meta_graph()` verwenden, falls Ihnen dieser zur Verfügung steht.

Allgemein sollten Sie nur einen Teil des ursprünglichen Modells verwenden, normalerweise die ersten Schichten. Die Funktion `import_meta_graph()` lädt den gesamten ursprünglichen Graphen, es hindert Sie aber nichts daran, einige unerwünschte Schichten zu ignorieren. Beispielsweise könnten Sie einige neue Schichten auf den vortrainierten erstellen (z.B. eine verborgene und eine Ausgabeschicht auf der bestehenden verborgenen Schicht 3), wie in Abbildung 11-4 zu sehen ist. Sie müssten dann die Verlustfunktion für diese neue Ausgabe berechnen und einen Optimierer erstellen, um den Verlust zu minimieren.

Wenn Sie den Python-Code des vortrainierten Graphen haben, können Sie einfach die gewünschten Teile verwenden und den Rest löschen. In diesem Fall müssen Sie aber das vortrainierte Modell mit einem Saver wiederherstellen (und die wiederherstellenden Variablen angeben; andernfalls beschwert sich TensorFlow darüber, dass die Graphen nicht übereinstimmen) und das neue Modell mit einem zweiten Saver sichern. Beispielsweise stellt der folgende Code lediglich die Schichten 1, 2 und 3 wieder her:

```
[...] # erstelle das neue Modell mit den verborgenen Schichten 1-3  
  
reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,  
                               scope="hidden[123]") # regulärer Ausdruck  
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])  
restore_saver = tf.train.Saver(reuse_vars_dict) # lädt die Schichten 1-3  
  
init = tf.global_variables_initializer() # initialisiert alte und neue Variablen  
saver = tf.train.Saver() # speichert das neue Modell  
  
with tf.Session() as sess:  
    init.run()  
    restore_saver.restore(sess, "./my_model_final.ckpt")  
    [...] # trainiere das Modell  
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

Zuerst erstellen wir das neue Modell und kopieren die verborgenen Schichten 1 bis 3 aus dem bestehenden. Dann beschaffen wir uns über den regulären Ausdruck "hidden[123]" eine Liste sämtlicher Variablen in den verborgenen Schichten 1 bis 3. Anschließend erstellen wir ein Dictionary, das die Namen jeder Variable im ursprünglichen Modell zu seinem Namen im neuen Modell abbildet (die Namen sollten in der Regel die gleichen sein). Als Nächstes erstellen wir einen Saver, der nur diese Variablen wiederherstellt. Wir erstellen auch eine Operation zum Initia-

lisieren aller Variablen (alte und neue) und einen zweiten Saver, um das gesamte neue Modell und nicht nur die Schichten 1 bis 3 abzuspeichern. Wir starten eine Session, initialisieren sämtliche Variablen des Modells, stellen die Variablenwerte aus Schicht 1 bis 3 des ursprünglichen Modells wieder her, trainieren das Modell für die neue Aufgabe und speichern es schließlich.



Je ähnlicher die Aufgaben sind, desto mehr Schichten können Sie wiederverwenden (beginnend mit den ersten Schichten). Bei sehr ähnlichen Aufgaben können Sie probieren, alle verborgenen Schichten beizubehalten und nur die Ausgabeschicht zu ersetzen.

Modelle aus anderen Frameworks wiederverwenden

Falls das Modell mit einem anderen Framework trainiert wurde, müssen Sie die Modellparameter manuell laden (d.h. mit Code unter Theano, falls es unter Theano trainiert wurde) und diese dann den entsprechenden Variablen zuweisen. Dies kann recht mühselig werden. Das folgende Codebeispiel zeigt, wie Sie Gewichte und Bias aus der ersten verborgenen Schicht eines mit einem anderen Framework trainierten Modells übertragen können:

```
original_w = [...] # Lade die Gewichte aus dem anderen Framework
original_b = [...] # Lade die Biases aus dem anderen Framework
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
[...] # Erstelle das restliche Modell

# Beschaffe die Knoten für die Variablen in hidden1
graph = tf.get_default_graph()
assign_kernel = graph.get_operation_by_name("hidden1/kernel/Assign")
assign_bias = graph.get_operation_by_name("hidden1/bias/Assign")
init_kernel = assign_kernel.inputs[1]
init_bias = assign_bias.inputs[1]

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init, feed_dict={init_kernel: original_w, init_bias: original_b})
    # [...] Trainiere das Modell auf der neuen Aufgabe
```

In dieser Implementierung laden wir zuerst das vortrainierte Modell mit dem anderen Framework (hier nicht gezeigt). Wir extrahieren die gewünschten Modellparameter. Anschließend erstellen wir unser TensorFlow-Modell wie üblich. Dann kommt der schwierige Teil: Zu jeder Variablen in TensorFlow gehört eine Zuweisungsoperation, mit der sie initialisiert wird. Wir beschaffen uns zunächst ein Handle auf diese Zuweisungsoperationen (diese haben den gleichen Namen wie die Variable gefolgt von "/Assign"). Wir benötigen auch ein Handle für die zweite Eingabe jeder Zuweisungsoperation: Die zweite Eingabe ist ein der Variablen zugewiesener Wert, in diesem Fall der Startwert dieser Variablen. Wenn die Session startet, werden die gewöhnlichen Initialisierungsvorgänge ausgeführt, hier aber geben wir

die wiederzuverwendenden Variablen an. Alternativ dazu könnten wir Werte der Variablen mit neuen Zuweisungen und Platzhaltern auch nach der Initialisierung erstellen. Aber warum sollten wir neue Knoten im Graphen erstellen, wenn alles, was wir brauchen, dort bereits vorhanden ist?

Einfrieren der unteren Schichten

Die ersten Schichten des bestehenden DNN haben vermutlich gelernt, kleinteilige Eigenschaften von Bildern zu erkennen, die in beiden Bildklassifikationen nützlich sind. Sie können daher diese Schichten unverändert einsetzen. Es lohnt sich, deren Gewichte beim Trainieren des neuen DNN einzufrieren: Wenn die unteren Schichten feststehen, sind die Gewichte der übrigen Schichten leichter zu trainieren (weil sie sich nicht an ein bewegliches Ziel anpassen müssen). Um die unteren Schichten beim Trainieren einzufrieren, können Sie dem Optimierer eine Liste zu trainierender Variablen übergeben und dabei die Variablen der ersten Schichten auslassen:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

Die erste Zeile generiert eine Liste aller trainierbaren Variablen der Schichten 3 und 4 sowie der Ausgabeschicht. Damit lassen wir die Variablen der verborgenen Schichten 1 und 2 aus. Anschließend übergeben wir die so eingeschränkte Liste trainierbarer Variablen an die Funktion `minimize()` des Optimierers. Ta-da! Die Schichten 1 und 2 sind nun eingefroren: Sie rühren sich beim Trainieren nicht (man nennt diese auch *Frozen Layers*).

Eine andere Möglichkeit ist, mit `stop_gradient()` eine Schicht in den Graphen einzufügen, die sämtliche Schichten darunter einfriert:

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                            name="hidden1") # wiederverwendet, eingefroren
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                            name="hidden2") # wiederverwendet, eingefroren
    hidden2_stop = tf.stop_gradient(hidden2)
    hidden3 = tf.layers.dense(hidden2_stop, n_hidden3, activation=tf.nn.relu,
                            name="hidden3") # wiederverwendet, nicht eingefroren
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu,
                            name="hidden4") # neu!
    logits = tf.layers.dense(hidden4, n_outputs, name="outputs") # neu!
```

Caching der eingefrorenen Schichten

Da sich die eingefrorenen Schichten nicht verändern, lässt sich die Ausgabe der obersten eingefrorenen Schicht für jeden Datenpunkt in einem Cache speichern. Da beim Trainieren der gesamte Datensatz mehrmals durchlaufen wird, führt dies zu einem bedeutenden Geschwindigkeitsgewinn, da Sie zum Cachen die eingefrorenen Schichten nur einmal pro Trainingsdatenpunkt durchlaufen müssen (anstatt

einmal pro Epoche). Sie könnten beispielsweise den gesamten Trainingsdatensatz mit den ersten Schichten verarbeiten (genug RAM vorausgesetzt), dann beim Trainieren Batches mit Ausgabewerten der verborgenen Schicht 2 anstatt Batches mit Trainingsdatenpunkten verwenden, und diese der Operation zum Trainieren übergeben:

```
import numpy as np

n_batches = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")

    h2_cache = sess.run(hidden2, feed_dict={X: mnist.train.images})

    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(mnist.train.num_examples)
        hidden2_batches = np.array_split(h2_cache[shuffled_idx], n_batches)
        y_batches = np.array_split(mnist.train.labels[shuffled_idx], n_batches)
        for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
            sess.run(training_op, feed_dict={hidden2:hidden2_batch, y:y_batch})

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

Die letzte Zeile der Trainingsschleife führt die weiter oben definierte Trainingsoperation aus (die die Schichten 1 und 2 nicht betrifft) und übergibt einen Batch Ausgabewerte aus der zweiten verborgenen Schicht (und die dazugehörigen Zielwerte). Da wir TensorFlow die Ausgabe der verborgenen Schicht 2 übergeben, werden diese nicht ausgewertet (oder die von ihr abhängigen Knoten).

Verändern, Auslassen oder Ersetzen der oberen Schichten

Die Ausgabeschicht des ursprünglichen Modells sollte in der Regel ersetzt werden, da sie meist für die neue Aufgabe überhaupt nicht nützlich ist und womöglich ohnehin nicht die richtige Anzahl Ausgaben hat.

Auch die oberen verborgenen Schichten des ursprünglichen Modells sind eher selten so nützlich wie die unteren Schichten, da die komplexeren Merkmale bei der neuen Aufgabe beträchtlich von denen der ursprünglichen Aufgabe abweichen können. Es gilt also, die richtige Anzahl zu verwendender Schichten zu finden.

Frieren Sie zunächst sämtliche kopierten Schichten ein. Dann trainieren Sie das Modell und prüfen dessen Leistungsfähigkeit. Schalten Sie anschließend eine oder zwei der oberen verborgenen Schichten hinzu, sodass die Backpropagation diese verändern kann, und prüfen Sie, ob sich die Leistung verbessert. Je mehr Trainingsdaten Sie haben, desto mehr nicht eingefrorene Schichten können Sie hinzuziehen.

Wenn Sie noch immer keine gute Leistung erzielen und über wenige Trainingsdaten verfügen, können Sie die oberste(n) verborgene(n) Schichten ganz verwerfen

und die verbliebenen Schichten wieder einfrieren. Diesen Vorgang wiederholen Sie, bis Sie die richtige Anzahl Schichten gefunden haben. Wenn Sie viele Trainingsdaten haben, können Sie versuchen, die obersten verborgenen Schichten zu ersetzen, anstatt sie zu verwerfen, oder sogar noch mehr Schichten hinzufügen.

Modell-Zoos

Wo können Sie ein zu Ihrer Aufgabe passendes bereits trainiertes neuronales Netz finden? Die erste Anlaufstelle ist selbstverständlich Ihre eigene Modellsammlung. Es lohnt sich, alle Ihre Modelle zu speichern und so zu organisieren, dass Sie diese später leicht wiederfinden. Alternativ dazu können Sie in einem *Modell-Zoo* suchen. Viele Data Scientists trainieren Machine-Learning-Modelle für unterschiedliche Zwecke und stellen ihre vortrainierten Modelle freundlicherweise der Öffentlichkeit zur Verfügung.

Für TensorFlow existiert ein eigener Modell-Zoo unter <https://github.com/tensorflow/models>. Insbesondere enthält es die meisten hoch entwickelten neuronalen Netze zur Bildklassifizierung wie VGG, Inception und ResNet (siehe Kapitel 13, sehen Sie im Verzeichnis *models/slim* nach), darunter den Code, die vortrainierten Modelle und Hilfsmittel zum Herunterladen beliebter Bilddatensätze.

Ein weiterer beliebter Modell-Zoo ist der Caffe *Model Zoo* (<https://goo.gl/XI02X3>). Dort sind viele Modelle zur Bildverarbeitung enthalten (wie LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, inception), die auf unterschiedlichen Datensätzen trainiert wurden (z.B. ImageNet, Places Database, CIFAR10 und so weiter). Saumitro Dasgupta hat einen Konverter geschrieben, der unter <https://github.com/ethereon/caffe-tensorflow> verfügbar ist.

Unüberwachtes Vortrainieren

Nehmen Sie einmal an, Sie möchten eine komplexe Aufgabe bearbeiten, und es stehen Ihnen dafür nicht sehr viele gelabelte Trainingsdaten zur Verfügung. Leider finden Sie kein für eine ähnliche Aufgabe trainiertes Modell. Geben Sie nicht gleich auf! Zuerst sollten Sie natürlich versuchen, weitere gelabelte Trainingsdaten zu finden, aber falls dies zu schwierig oder zu teuer ist, können Sie immer noch *unüberwachtes Vortrainieren* durchführen (siehe Abbildung 11-5). Wenn Ihnen also reichlich ungelabelte Trainingsdaten zur Verfügung stehen, können Sie mit diesen eine Schicht nach der anderen trainieren. Sie beginnen bei der ersten Schicht und arbeiten sich mit einem unüberwachten Verfahren zur Erkennung von Merkmalen wie *Restricted Boltzmann Machines* (RBMs; siehe Anhang E) oder Autoencodern (siehe Kapitel 15) vor. Jede Schicht wird auf der Ausgabe der zuvor trainierten Schichten trainiert (sämtliche Schichten außer der gerade trainierten sind dazu eingefroren). Sobald Sie auf diese Weise alle Schichten trainiert haben, können Sie durch überwachtes Lernen eine Feinabstimmung des Netzes vornehmen (z.B. mittels Backpropagation).

Dies ist ein langer und mühevoller Prozess, der aber oft gut funktioniert; tatsächlich wurde diese Technik von Geoffrey Hinton und seinem Team im Jahr 2006 eingesetzt, was das Interesse an neuronalen Netzen wiederaufleben ließ und zum Siegeszug von Deep Learning führte. Bis 2010 war unüberwachtes Vortrainieren (üblicherweise mit RBMs) bei Deep-Learning-Netzen der Normalfall, und erst nachdem das Problem der schwindenden Gradienten behoben werden konnte, wurden allein auf Backpropagation basierende DNNs häufiger. Allerdings ist das unüberwachte Vortrainieren (heutzutage eher mit Autoencodern als mit RBMs) eine sinnvolle Option, wenn Sie es mit einer komplexen Aufgabe zu tun haben, Ihnen kein verwertbares Modell zur Verfügung steht und Sie wenige gelabelte, dafür aber viele ungelabelte Trainingsdaten haben.¹⁰

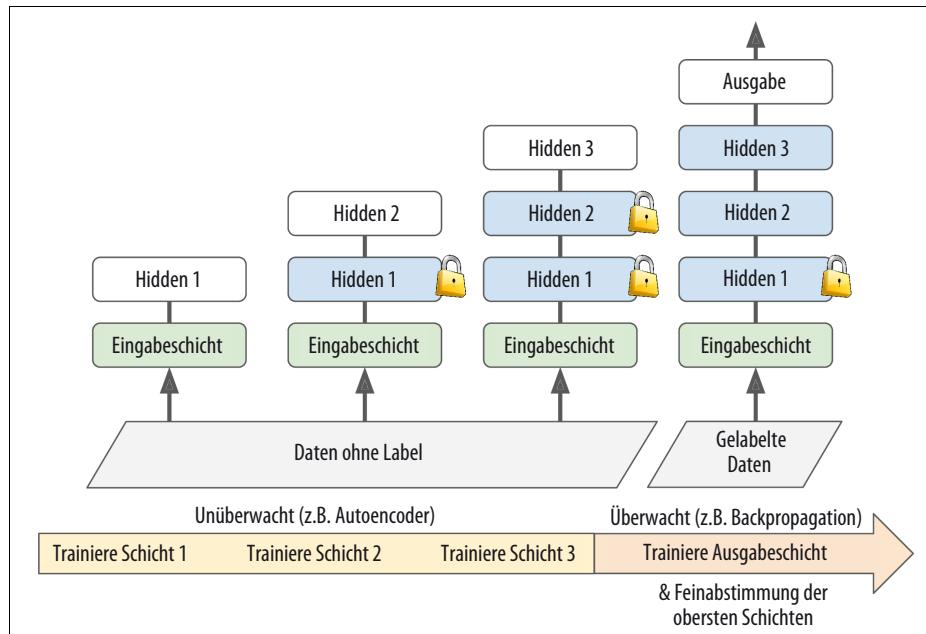


Abbildung 11-5: Unüberwachtes Vortrainieren

Vortrainieren anhand einer Hilfsaufgabe

Abschließend stellen wir die Möglichkeit vor, zuerst ein neuronales Netz auf einer Hilfsaufgabe zu trainieren, für das wir gelabelte Trainingsdaten leicht erhalten oder generieren können. Anschließend verwenden wir die unteren Schichten dieses Net-

¹⁰ Eine Alternative ist, eine überwachte Lernaufgabe zu entwickeln, bei der Sie leichter an gelabelte Trainingsdaten herankommen. Wenn Sie beispielsweise ein Modell trainieren möchten, um Ihre Freunde in Bildern zu identifizieren, könnten Sie Millionen Bilder mit Gesichtern aus dem Internet herunterladen. Damit trainieren Sie einen Klassifikator, der entscheidet, ob zwei Gesichter identisch sind oder nicht, um anschließend neue Bilder mit allen Bildern Ihrer Freunde zu vergleichen.

zes für die eigentliche Aufgabe. Die unteren Schichten des ersten Netzes lernen, Merkmale zu erkennen, die vom zweiten Netz genutzt werden können.

Beispielsweise stehen Ihnen beim Konstruieren eines Systems zur Gesichtserkennung nur wenige Bilder jeder Einzelperson zur Verfügung – sicher nicht genug, um einen guten Klassifikator zu trainieren. Hunderte Bilder jeder Person zu sammeln, wäre sicher nicht praktikabel. Sie könnten allerdings eine Menge Bilder zufällig ausgewählter Personen im Internet sammeln und ein erstes neuronales Netz darauf trainieren, ob zwei Bilder die gleiche Person enthalten. Solch ein Netz würde gute Merkmale von Gesichtern erlernen, sodass sich mit den unteren Schichten auch mit wenigen Trainingsdaten ein guter Klassifikator trainieren ließe.

Trainingsbeispiele ohne Labels lassen sich meist kostengünstig sammeln, diese mit Labels zu versehen, ist dagegen recht teuer. In diesem Fall ist es üblich, alle Trainingsbeispiele als »gut« zu labeln, anschließend durch Verzerren dieser guten Daten neue Trainingsbeispiele zu erzeugen und diese als »schlecht« zu labeln. Damit können Sie ein erstes neuronales Netz trainieren, um Datenpunkte als gut oder schlecht einzustufen. Sie könnten beispielsweise Millionen Sätze herunterladen, diese als »gut« labeln, dann in jedem Satz zufällig ein Wort ändern und die geänderten Sätze als »schlecht« labeln. Wenn ein neuronales Netz erkennen kann, dass »Der Hund schläft« ein guter Satz ist, »Der Hund sie« jedoch ein schlechter, weiß es vermutlich schon eine Menge über Sprache. Die unteren Schichten dieses Netzes wiederzuverwenden, wird voraussichtlich bei vielen Aufgaben in der Sprachverarbeitung hilfreich sein.

Sie können auch ein erstes Netz trainieren, um für jeden Trainingsdatenpunkt einen Score auszugeben, und eine Kostenfunktion verwenden, die sicherstellt, dass der Score eines guten Datenpunkts mindestens um einen bestimmten Betrag höher ist als der Score eines schlechten. Dies bezeichnet man als *Max Margin Learning*.

Schnellere Optimierer

Das Trainieren eines sehr großen Deep-Learning-Netzes kann nervtötend langsam sein. Wir haben bisher vier Möglichkeiten betrachtet, um das Trainieren zu beschleunigen (und die Lösung zu verbessern): eine geeignete Initialisierungsstrategie für die Gewichte der Verbindungen zu wählen, die Aktivierungsfunktion sinnvoll zu wählen, Batch-Normalisierung zu verwenden und Teile eines vortrainierten Netzes zu verwenden. Eine weitere deutliche Beschleunigung lässt sich durch Verwenden eines schnelleren Optimierers anstelle des gewöhnlichen Gradientenverfahrens erzielen. In diesem Abschnitt stellen wir die verbreitetsten Verfahren vor: Momentum Optimization, Accelerated Gradient nach Nesterov, AdaGrad, RMSProp und schließlich die Adam-Optimierung.

Momentum Optimization

Stellen Sie sich eine Bowlingkugel vor, die eine leicht abschüssige glatte Oberfläche herunterrollt: Sie beginnt langsam, beschleunigt aber bald, bis sie eine Endgeschwindigkeit erreicht (falls es Reibung und Luftwiderstand gibt). Dies ist die einfache Idee bei der von Boris Polyak im Jahr 1964 vorgeschlagenen *Momentum Optimization* (<https://goo.gl/F1SE8c>).¹¹ Im Gegensatz dazu führt das Gradientenverfahren auf einer schießen Ebene einfach viele gleich große Schritte nach unten aus, sodass es insgesamt eine längere Zeit bis zum unteren Ende benötigt.

Wie erwähnt aktualisiert das Gradientenverfahren die Gewichte θ , indem es den Gradienten von der nach den Gewichten ($\nabla_{\theta}J(\theta)$) abgeleiteten Kostenfunktion $J(\theta)$, multipliziert mit der Lernrate η , direkt abzieht. Die Gleichung hierfür lautet: $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$. Sie kümmert sich nicht darum, wie groß frühere Gradienten waren. Wenn der lokale Gradient klein ist, arbeitet das Verfahren sehr langsam.

Momentum Optimization berücksichtigt die früheren Gradienten: Bei jedem Schritt fügt es den lokalen Gradienten zum *Momentvektor* \mathbf{m} hinzu (multipliziert mit der Lernrate η) und aktualisiert die Gewichte, indem es diesen Momentvektor einfach abzieht (siehe Formel 11-4). Anders ausgedrückt wird der Gradient als Beschleunigung, nicht als Geschwindigkeit interpretiert. Um Reibung nachzubilden und zu verhindern, dass das Moment zu groß wird, gibt es im Verfahren den zusätzlichen Hyperparameter β , das *Moment*, der zwischen 0 (hohe Reibung) und 1 (keine Reibung) liegen muss. Ein typisches Moment beträgt 0.9.

Formel 11-4: *Moment-Algorithmus*

1. $\mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla_{\theta}J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

Sie können leicht nachweisen, dass die Endgeschwindigkeit (der maximale Betrag der Gewichtsveränderung) bei einem konstanten Gradienten gleich dem Produkt aus Gradient, Lernrate η und $\frac{1}{1-\beta}$ ist. Wenn beispielsweise $\beta = 0.9$ gilt, so beträgt die Endgeschwindigkeit 10 mal Gradient mal Lernrate. Die Momentum Optimization bewegt sich also 10 Mal schneller als das Gradientenverfahren! Dadurch kann die Momentum Optimization schneller als das Gradientenverfahren aus Plateaus entkommen. Wir haben in Kapitel 4 gesehen, dass die Kostenfunktion bei unterschiedlich skalierten Eingaben wie eine breite Schüssel aussieht (siehe Abbildung 4-7). Das Gradientenverfahren steigt die steilen Wände schnell herab, benötigt dann aber eine lange Zeit bis ins Tal. Die Momentum Optimization dagegen rollt immer schneller und schneller das Tal entlang, bis sie den tiefsten Punkt (das Optimum) erreicht. Bei Deep-Learning-Netzen ohne Batch-Normalisierung haben die oberen Schichten häufig Eingaben mit sehr unterschiedlichen Wertebereichen. In

¹¹ »Some methods of speeding up the convergence of iteration methods«, B. Polyak (1964).

dieser Situation ist die Momentum Optimization sehr hilfreich. Sie hilft auch dabei, an lokalen Optima vorbeizurollen.



Wegen des Moments kann der Optimierer ein wenig über das Ziel hinausschießen, zurückkehren, sich wieder zu weit entfernen und so mehrmals oszillieren, bevor er sich beim Minimum stabilisiert. Aus diesem Grund ist es gut, ein wenig Reibung im System zu haben: Sie eliminiert diese Oszillation und beschleunigt daher die Konvergenz.

Das Implementieren der Momentum Optimization in TensorFlow ist ein Klacks: Ersetzen Sie einfach den `GradientDescentOptimizer` durch den `MomentumOptimizer` und lehnen Sie sich anschließend zurück!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

Der Hauptnachteil der Momentum Optimization ist, dass wir uns um einen weiteren Hyperparameter kümmern müssen. In der Praxis funktioniert der Wert 0.9 für das Moment meist gut und ist fast immer schneller als das Gradientenverfahren.

Beschleunigter Gradient nach Nesterov

Eine kleine Variante der Momentum Optimization, die von Yurii Nesterov im Jahr 1983 (<https://goo.gl/V011vD>) vorgestellt wurde,¹² ist so gut wie immer schneller als die reine Momentum Optimization. Die Idee bei der *Momentum Optimization nach Nesterov* oder dem *beschleunigten Gradienten nach Nesterov* (NAG) ist, den Gradienten der Kostenfunktion nicht an der aktuellen Position zu bestimmen, sondern ein wenig weiter vorwärts in Richtung des Moments (siehe Formel 11-5). Der einzige Unterschied zur ursprünglichen Momentum Optimization ist, dass der Gradient bei $\theta + \beta\mathbf{m}$ anstatt bei θ bestimmt wird.

Formel 11-5: Algorithmus des beschleunigten Gradienten nach Nesterov

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$

Diese kleine Modifikation funktioniert, weil der Momentvektor im Allgemeinen in die richtige Richtung zeigt (also zum Optimum hin). Daher ist das Verwenden des Gradienten ein Stück weiter in dieser Richtung etwas genauer als an der ursprünglichen Position, wie Sie Abbildung 11-6 entnehmen können (wobei ∇_1 für den Gradienten der am Ausgangspunkt θ bestimmten Kostenfunktion steht und ∇_2 für den Gradienten am Punkt $\theta + \beta\mathbf{m}$).

¹² »A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$ «, Yurii Nesterov (1983).

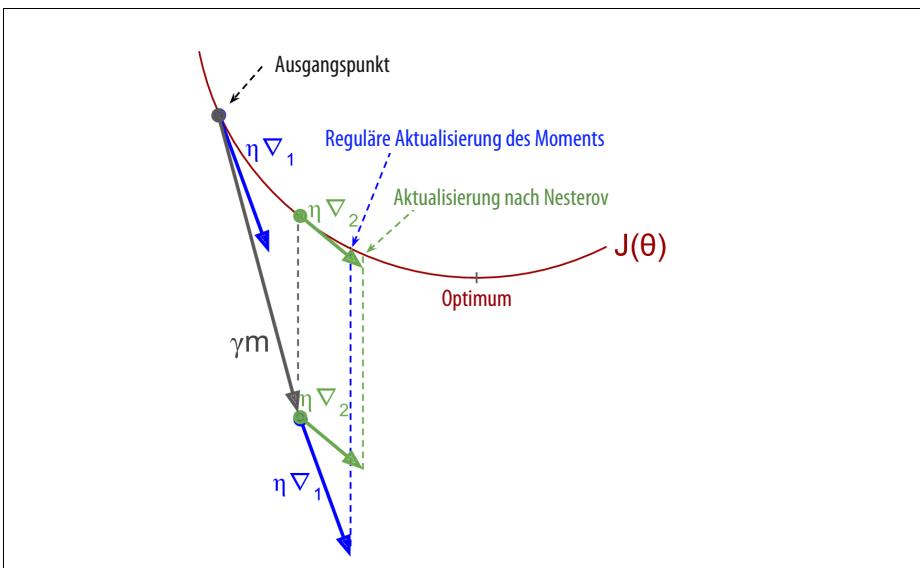


Abbildung 11-6: Gewöhnliche Momentum Optimization im Vergleich zur Variante nach Nesterov

Wie Sie sehen, führt die Änderung nach Nesterov etwas näher ans Optimum heran. Nach einer Weile addieren sich diese kleinen Verbesserungen auf, wodurch NAG deutlich schneller als die gewöhnliche Momentum Optimization ist. Wenn zudem das Moment die Gewichte durch ein Tal zieht, drückt ∇_1 sie weiter durch das Tal voran, während ∇_2 sie in Richtung der Talsohle schiebt. Dadurch wird Oszillation unterbunden und die Konvergenz beschleunigt.

NAG beschleunigt das Trainieren ähnlich wie gewöhnliche Momentum Optimization. Um sie zu verwenden, setzen Sie einfach beim Erstellen eines MomentumOptimizer den Parameter `use_nesterov=True`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

Betrachten wir noch einmal das Problem der länglichen Schüssel: Das Gradientenverfahren bewegt sich schnell entlang der steilsten Wand und geht dann langsam bis zur Talsohle weiter. Es wäre schön, wenn der Algorithmus dies früh erkennen könnte, um sich ein wenig mehr in Richtung des globalen Optimums auszurichten.

Der *AdaGrad-Algorithmus* (<http://goo.gl/4Tyd4j>)¹³ arbeitet genau so, er skaliert den Gradientenvektor entlang der steilsten Dimensionen herunter (siehe Formel 11-6):

¹³ »Adaptive Subgradient Methods for Online Learning and Stochastic Optimization«, J. Duchi et al. (2011).

Formel 11-6: AdaGrad – Algorithmus

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

Der erste Schritt akkumuliert das Quadrat der Gradienten in den Vektor \mathbf{s} (das Symbol \otimes steht für die Multiplikation der einzelnen Elemente). Diese vektorisierte Form entspricht der Berechnung von $s_i \leftarrow s_i + (\partial / \partial \theta_i J(\theta))^2$ für jedes Element s_i des Vektors \mathbf{s} ; anders ausgedrückt akkumuliert jedes s_i die Quadrate der partiellen Ableitung der Kostenfunktion nach dem Parameter θ_i . Ist die Kostenfunktion entlang der i^{ten} Dimension steil, wird s_i von Iteration zu Iteration immer größer.

Der zweite Schritt ist zum Gradientenverfahren beinahe identisch, es gibt aber einen großen Unterschied: Der Gradientenvektor wird um den Faktor $\sqrt{\mathbf{s} + \epsilon}$ herunterskaliert (das Symbol \oslash steht für die elementweise Division, und ϵ ist ein Glättungsterm, um Divisionen durch null zu vermeiden, und beträgt normalerweise 10^{-10}). Diese Vektorschreibweise entspricht der (gleichzeitigen) Berechnung von $\theta_i \leftarrow \theta_i - \eta \partial / \partial \theta_i J(\theta) / \sqrt{s_i + \epsilon}$ für alle Parameter θ_i .

Kurz, dieser Algorithmus baut die Lernrate nach und nach ab, bei steilen Dimensionen erfolgt der Abbau aber schneller als bei Dimensionen mit geringeren Steigungen. Dies bezeichnet man auch als *adaptive Lernrate*. Die daraus entstehenden Aktualisierungen weisen eher in Richtung des globalen Optimums (siehe Abbildung 11-7). Ein zusätzlicher Vorteil ist, dass wesentlich weniger Feinabstimmung der Lernrate als Hyperparameter nötig ist η .

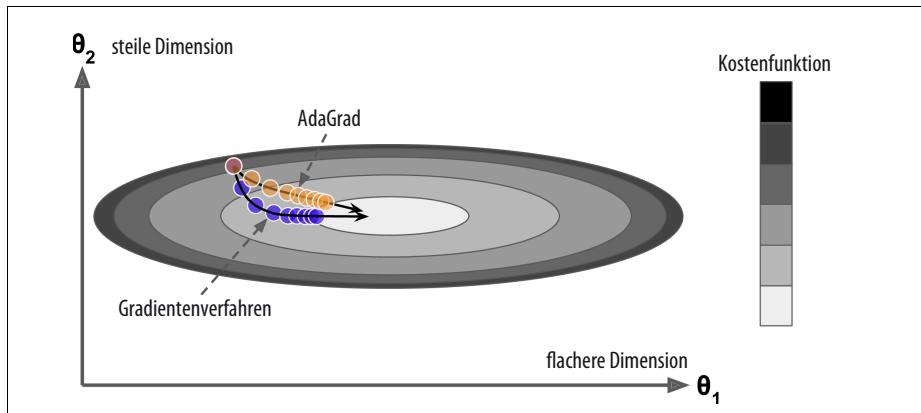


Abbildung 11-7: AdaGrad im Vergleich zum Gradientenverfahren

AdaGrad schneidet bei einfachen quadratischen Problemen oft gut ab, hält aber beim Trainieren neuronaler Netze häufig zu früh an. Die Lernrate wird dann so weit herunterskaliert, dass der Algorithmus komplett anhält, bevor er das globale Optimum erreicht. Obwohl es in TensorFlow einen AdagradOptimizer gibt, sollten Sie

diesen nicht zum Trainieren von Deep-Learning-Netzen verwenden (bei einfacheren Aufgaben wie der linearen Regression kann er sich aber als effizient erweisen).

RMSProp

Weil AdaGrad ein wenig zu schnell abbremst und daher in manchen Fällen das globale Optimum nie erreicht, kümmert sich der *RMSProp*-Algorithmus¹⁴ um dieses Problem, indem er nur die Gradienten aus den letzten Iterationen akkumuliert (anstatt sämtliche Gradienten seit Trainingsbeginn zu verwenden). Dazu wird im ersten Schritt ein exponentieller Zerfall verwendet (siehe Formel 11-7).

Formel 11-7: *RMSProp*-Algorithmus

1. $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

Die Zerfallsrate β wird normalerweise auf 0.9 gesetzt. Ja, es ist schon wieder ein neuer Hyperparameter, aber dieser Standardwert funktioniert meist so gut, dass Sie ihn nicht weiter verändern müssen.

Wie Sie sich vielleicht schon denken, gibt es in TensorFlow die Klasse `RMSPropOptimizer`:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Außer bei sehr einfachen Aufgaben ist dieser Optimierer AdaGrad so gut wie immer überlegen. Er konvergiert zumeist auch schneller als Momentum Optimization und der beschleunigte Gradient nach Nesterov. Deshalb war dies der von vielen Forschern bevorzugte Algorithmus, bis die Adam-Optimierung auf dem Plan erschien.

Adam-Optimierung

Adam (<https://goo.gl/Un8Axu>)¹⁵, eine Abkürzung für *Adaptive Moment Estimation*, kombiniert die Idee der Momentum Optimization und RMSProp: Wie die Momentum Optimization merkt sich das Verfahren den Durchschnitt der vorigen Gradienten, und wie RMSProp merkt es sich auch den Durchschnitt der quadrierten Gradienten, beide unter exponentiellem Zerfall (siehe Formel 11-8).¹⁶

¹⁴ Dieser Algorithmus wurde von Tijmen Tieleman und Geoffrey Hinton im Jahr 2012 entwickelt und von Geoffrey Hinton in seinem Coursera-Kurs zu neuronalen Netzen präsentiert (Folien: <http://goo.gl/RsQeis>; Video: <https://goo.gl/XUblyf>). Da die Autoren keinen Fachartikel über das Verfahren verfasst haben, zitieren Forscher in ihren Artikeln häufig »Folie 29 in Vorlesung 6«.

¹⁵ »Adam: A Method for Stochastic Optimization«, D. Kingma, J. Ba (2015).

¹⁶ Dies sind Schätzungen des Mittelwerts und der (nicht zentrierten) Varianz der Gradienten. Der Mittelwert wird oft als *erstes Moment*, die Varianz als *zweites Moment* bezeichnet. Daher röhrt der Name des Algorithmus.

Formel 11-8: Adam-Algorithmus

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$
5. $\theta \leftarrow \theta - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- T steht für den Index der Iteration (beginnend bei 1).

Wenn Sie sich nur die Schritte 1, 2 und 5 ansehen, bemerken Sie die Nähe der Adam-Optimierung zu Momentum Optimization und RMSProp. Der einzige Unterschied ist, dass bei Schritt 1 ein exponentiell abfallender Durchschnitt anstatt einer exponentiell abfallenden Summe berechnet wird. Diese sind aber bis auf einen konstanten Faktor gleich (der abfallende Durchschnitt beträgt $1 - \beta_1$ Mal die abfallende Summe). Die Schritte 3 und 4 enthalten ein technisches Detail: Weil \mathbf{m} und \mathbf{s} mit 0 initialisiert wurden, enthalten Sie zu Beginn des Trainings ein Bias in Richtung 0. Deshalb werten diese beiden Schritte \mathbf{m} und \mathbf{s} zu Beginn des Trainierens auf.

Der Hyperparameter für den Abfall des Moments β_1 wird normalerweise mit 0.9 initialisiert, der Hyperparameter für den Abfall der Skalierung β_2 wird häufig mit 0.999 initialisiert. Wie weiter oben wird der Glättungsterm ϵ normalerweise mit einer sehr kleinen Zahl wie 10^{-8} initialisiert. Dies sind die voreingestellten Werte der TensorFlow-Klasse AdamOptimizer, die Sie folgendermaßen verwenden:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Da Adam ein Algorithmus mit adaptiver Lernrate ist (wie auch AdaGrad und RMSProp), muss weniger Feineinstellung der Lernrate η erfolgen. Sie können meist den Standardwert $\eta = 0.001$ verwenden, wodurch Adam sogar einfacher als das Gradientenverfahren verwendbar ist.



Diese Buch hatte die Adam-Optimierung ursprünglich empfohlen, weil man sie im Vergleich mit anderen Methoden für grundsätzlich schneller und besser hielt. Allerdings kam ein Artikel aus dem Jahr 2017 (<https://goo.gl/NAkW1a>)¹⁷ von Ashia C. Wilson et al. zu dem Schluss, dass adaptive Optimierungsverfahren (wie AdaGrad, RMSProp und die Adam-Optimierung) bei manchen Datensätzen zu einer schwachen Verallgemeinerungsleistung führen. Sie können also im Moment die Momentum Optimization oder den beschleunigten Gradienten nach Nesterov vorziehen, bis die Wissenschaftler dieses Problem besser verstanden haben.

¹⁷ »The Marginal Value of Adaptive Gradient Methods in Machine Learning«, A. C. Wilson et al. (2017).

Alle bisher besprochenen Optimierungstechniken beruhen lediglich auf den *partiellen Ableitungen erster Ordnung (Jacobians)*. Die Literatur zu Optimierungsverfahren enthält faszinierende Algorithmen, die mit den *partiellen Ableitungen zweiter Ordnung (Hessians)* arbeiten. Leider lassen sich diese Algorithmen sehr schwer auf neuronale Netze anwenden, weil es pro Ausgabe n^2 Hessians gibt (wobei n die Anzahl der Parameter ist). Im Gegensatz dazu gibt es nur n Jacobians pro Ausgabe. Da DNNs normalerweise Zehntausende Parameter enthalten, passen die Algorithmen zweiten Grades oft nicht einmal in den Speicher. Selbst wenn sie passen, ist die Berechnung der Hessians einfach zu langsam.

Trainieren spärlicher Modelle

Alle eben vorgestellten Optimierungsalgorithmen erzeugen dichte Modelle, solche, bei denen die meisten Parameter ungleich null sind. Wenn Sie ein zur Laufzeit blitzschnelles Modell benötigen oder wenn es wenig Speicher verbrauchen soll, sollten Sie ein spärliches Modell in Betracht ziehen.

Eine triviale Möglichkeit hierzu ist, das Modell wie gewohnt zu trainieren und anschließend winzige Gewichte auf null zu setzen.

Eine andere Möglichkeit ist, während des Trainierens eine starke ℓ_1 -Regularisierung anzuwenden, da diese den Optimierer veranlasst, möglichst viele Gewichte auf null zu setzen (wie im Falle der in Kapitel 4 besprochenen Lasso-Regression).

In einigen Fällen erweisen sich diese Techniken jedoch als unzureichend. Eine letzte Option ist dann das *Dual Averaging*, auch genannt *Follow The Regularized Leader* (FTRL), eine von Yurii Nesterov vorgeschlagene Technik (<https://goo.gl/xSQD4C>).¹⁸ Zusammen mit ℓ_1 -Regularisierung führt diese Technik häufig zu sehr spärlichen Modellen. In TensorFlow ist mit der Klasse `FTRLOptimizer` eine Variante der FTRL namens *FTRL-Proximal* (<https://goo.gl/bxme2B>)¹⁹ implementiert.

Scheduling der Lernrate

Eine gute Lernrate zu finden, ist manchmal schwierig. Wenn Sie sie zu hoch einstellen, divergiert das Modell beim Trainieren (wie in Kapitel 4 besprochen). Wenn Sie sie zu niedrig einstellen, konvergiert das Modell irgendwann beim Optimum, das Trainieren wird aber sehr lange dauern. Wenn Sie sie ein klein wenig zu hoch einstellen, macht das Modell zunächst sehr gute Fortschritte, tanzt dann aber um das Optimum herum und kommt nie zur Ruhe (es sei denn, Sie verwenden einen Algorithmus mit adaptiver Lernrate wie AdaGrad, RMSProp oder Adam, aber selbst dann beruhigt es sich nicht immer sofort). Wenn Ihre Rechenzeit begrenzt ist, müs-

¹⁸ »Primal-Dual Subgradient Methods for Convex Problems«, Yurii Nesterov (2005).

¹⁹ »Ad Click Prediction: a View from the Trenches«, H. McMahan et al. (2013).

sen Sie das Trainieren unterbrechen, bevor es anständig konvergiert, und sich mit einer suboptimalen Lösung zufriedengeben (siehe Abbildung 11-8).

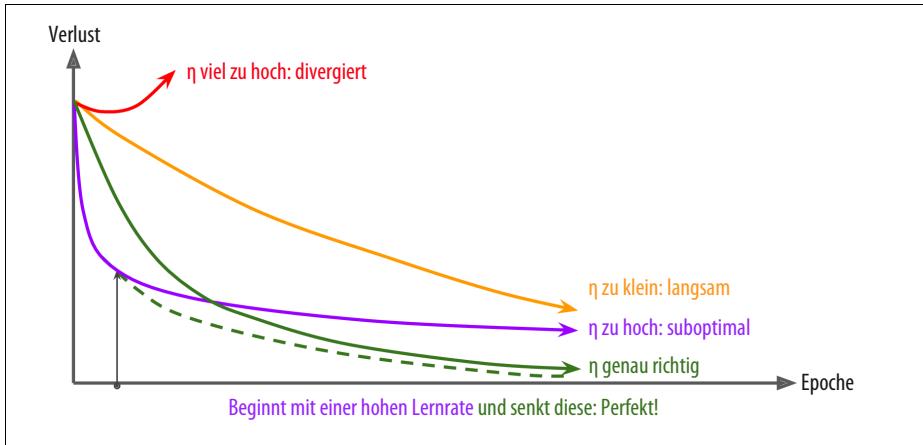


Abbildung 11-8: Lernkurven für unterschiedliche Lernraten η

Vielleicht finden Sie eine gute Lernrate, indem Sie Ihr Netz mehrmals für nur wenige Epochen mit verschiedenen Lernraten trainieren und die Lernkurven vergleichen. Die ideale Lernrate sollte schnell lernen und bei einer anständigen Lösung konvergieren.

Sie können aber auch etwas Besseres als eine konstante Lernrate erzielen: Wenn Sie mit einer hohen Lernrate beginnen und diese reduzieren, sobald sie keine großen Fortschritte mehr macht, lässt sich eine gute Lösung schneller als mit einer konstanten Lernrate finden. Es gibt viele unterschiedliche Strategien, mit denen sich die Lernrate beim Trainieren verringern lässt. Diese Strategien bezeichnet man als *Scheduling der Lernrate* (wir haben den Begriff kurz in Kapitel 4 erwähnt). Die häufigsten Strategien sind:

Vorgefertigte stückweise konstante Lernrate

Setzen Sie die Lernrate zu Beginn auf $\eta_0 = 0.1$, dann nach 50 Epochen auf $\eta_1 = 0.001$. Obwohl diese Lösung sehr gut funktioniert, muss man meist ein wenig experimentieren, bis man die richtigen Zeitpunkte und Lernraten gefunden hat.

Performance Scheduling

Messen Sie alle N Schritte den Validierungsfehler (wie beim Early Stopping) und verringern Sie die Lernrate um den Faktor λ , sobald der Fehler nicht mehr abfällt.

Exponentielles Scheduling

Legen Sie die Lernrate als Funktion der Iteration t fest: $\eta(t) = \eta_0 10^{-t/r}$. Dies funktioniert prima, es muss aber eine Feineinstellung von η_0 und r vorgenommen werden. Die Lernrate fällt alle r Schritte um den Faktor 10.

Power Scheduling

Setzen Sie die Lernrate auf $\eta(t) = \eta_0 (1 + t/r)^{-c}$. Der Hyperparameter c wird normalerweise auf 1 gesetzt. Dies ist dem exponentiellen Scheduling ähnlich, die Lernrate fällt aber viel langsamer ab.

Ein Artikel aus dem Jahr 2013 (<http://goo.gl/Hu6Zyq>)²⁰ von Andrew Senior et al. verglich die Leistung der beliebtesten Scheduling-Verfahren zum Trainieren von Deep-Learning-Netzen zur Sprachverarbeitung mit Momentum Optimization. Die Autoren kamen zu dem Schluss, dass in diesem Szenario sowohl Performance Scheduling als auch exponentielles Scheduling gut abschneiden. Sie bevorzugten jedoch das exponentielle Scheduling, weil es sich einfacher implementieren und abstimmen lässt und ein wenig schneller zur optimalen Lösung gelangt. Ein Scheduling der Lernrate lässt sich mit TensorFlow recht einfach implementieren:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False, name="global_step")
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                             decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

Nach dem Einstellen der Hyperparameter erstellen wir die nicht trainierbare Variable `global_step` (bei 0 initialisiert), um die Nummer des aktuellen Trainings schritts zu verfolgen. Dann definieren wir eine exponentiell abfallende Lernrate (mit $\eta_0 = 0.1$ und $r = 10000$), wozu wir die TensorFlow-Funktion `exponential_decay()` verwenden. Danach erstellen wir einen Optimierer (in diesem Fall ein `MomentumOptimizer`) mit dieser abfallenden Lernrate. Schließlich erstellen wir die Operation zum Trainieren, indem wir die Methode `minimize()` des Optimierers aufrufen; da wir ihr die Variable `global_step` übergeben, kümmert sie sich freund licherweise um deren Erhöhung. Das ist alles!

Da AdaGrad, RMSProp und die Adam-Optimierung die Lernrate beim Trainieren automatisch optimieren, ist es nicht nötig, ein zusätzliches Scheduling-Verfahren zu verwenden. Bei allen anderen Optimierungsalgorithmen können exponentieller Abfall oder Performance Scheduling das Konvergieren erheblich beschleunigen.

²⁰ »An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition«, A. Senior et al. (2013).

Vermeiden von Overfitting durch Regularisierung

Mit vier Parametern kann ich einen Elefanten fitten, und mit fünf wackelt er mit dem Rüssel.

John von Neumann, zitiert von Enrico Fermi in *Nature* 427

Deep-Learning-Netze haben meist Zehntausende Parameter, manchmal sogar Millionen. Mit derart vielen Parametern hat das Netz enorm viele Freiheitsgrade und kann eine riesige Bandbreite komplexer Datensätze erfassen. Aber diese Flexibilität bedeutet auch, dass es anfällig für das Overfitten der Trainingsdaten ist.

Mit Millionen Parametern können Sie einen ganzen Zoo fitten. In diesem Abschnitt werden wir einige der beliebtesten Regularisierungstechniken für neuronale Netze vorstellen und wie sich diese mit TensorFlow implementieren lassen: Early Stopping, ℓ_1 und ℓ_2 -Regularisierung, Drop-out, Max-Norm-Regularisierung und Data Augmentation.

Early Stopping

Um ein Overfitting der Trainingsdaten zu vermeiden, ist das in Kapitel 4 erwähnte Early Stopping eine gute Lösung: Sobald die Vorhersageleistung auf den Validierungsdaten abzufallen beginnt, wird das Trainieren abgebrochen.

Mit TensorFlow lässt sich dies als Evaluierung des Modells auf einem Validierungsdatensatz in regelmäßigen Abständen umsetzen (z.B. alle 50 Schritte) und ein »Gewinnermodell« abspeichern, falls dieses Modell den zuvor gespeicherten Gewinner schlägt. Verfolgen Sie die Anzahl Schritte seit dem Abspeichern des letzten »Gewinners« und brechen Sie das Trainieren ab, sobald diese Zahl einen Schwellenwert erreicht (z.B. 2000 Schritte). Anschließend stellen Sie das letzte »Gewinnermodell« wieder her.

Obwohl Early Stopping in der Praxis gut funktioniert, können Sie normalerweise eine höhere Leistung aus Ihrem Netz herausholen, wenn Sie es mit anderen Techniken zur Regularisierung kombinieren.

ℓ_1 - und ℓ_2 -Regularisierung

Wie bei den einfachen linearen Modellen in Kapitel 4 können Sie mit ℓ_1 - und ℓ_2 -Regularisierung den Gewichten der Verbindungen eines neuronalen Netzes (aber nicht den Bias-Termen) Beschränkungen auferlegen.

Mit TensorFlow können Sie dazu einfach die entsprechenden Regularisierungsterme zu Ihrer Kostenfunktion hinzufügen. Wenn Sie beispielsweise eine verborgene Schicht mit den Gewichten W_1 und eine Ausgabeschicht mit den Gewichten W_2 haben, können Sie die ℓ_1 -Regularisierung folgendermaßen anwenden:

```
[...] # Konstruktion des neuronalen Netzes  
W1 = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
```

```

W2 = tf.get_default_graph().get_tensor_by_name("outputs/kernel:0")

scale = 0.001 # l1-Hyperparameter zur Regularisierung

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                               logits=logits)
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
    reg_losses = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
    loss = tf.add(base_loss, scale * reg_losses, name="loss")

```

Wenn es jedoch viele Schichten gibt, ist dieser Ansatz nicht besonders bequem. Glücklicherweise stellt uns TensorFlow eine bessere Alternative zur Verfügung. Viele Funktionen, die Variablen erstellen (wie etwa `get_variable()` oder `tf.layers.dense()`), akzeptieren für jede erstellte Variable das Argument `*_regularizer` (z.B. `kernel_regularizer`). Sie können hier eine beliebige Funktion angeben, die als Argument Gewichte annimmt und den entsprechenden Regularisierungsverlust zurückgibt. Die Funktionen `l1_regularizer()`, `l2_regularizer()` und `l1_l2_regularizer()` liefern solche Funktionen. Im folgenden Codebeispiel wird alles zusammen gesetzt:

```

my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None, name="outputs")

```

Dieser Code erstellt ein neuronales Netz mit zwei verborgenen Schichten und einer Ausgabeschicht. Er erstellt außerdem Knoten im Graphen, die den Regularisierungsverlust ℓ_1 für sämtliche Gewichte der jeweiligen Schicht berechnen. TensorFlow fügt die entsprechenden Knoten automatisch einer Collection mit allen Regularisierungsverlusten hinzu. Sie müssen nur noch diese Regularisierungsverluste zu Ihrer Verlustfunktion addieren:

```

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")

```



Vergessen Sie nicht, Ihre Regularisierungsverluste zur Verlustfunktion zu addieren, andernfalls werden sie einfach ignoriert.

Drop-out

Die bei Deep-Learning-Netzen beliebteste Regularisierungstechnik ist ohne Zweifel *Drop-out*. Sie wurde von G. E. Hinton im Jahr 2012 vorgeschlagen (<https://goo.gl/PMjVnG>)²¹ und in einem Artikel (<http://goo.gl/DNKZo1>)²² von Nitish Srivastava et

al. genauer ausgeführt. Sie hat sich als höchst erfolgreich erwiesen: Selbst die am weitesten entwickelten neuronalen Netze erfuhren durch das Hinzufügen von Drop-out einen 1 – 2%igen Zugewinn an Genauigkeit. Dies klingt nicht nach besonders viel, aber wenn ein Modell bereits eine Genauigkeit von 95% erzielt, bedeuten 2% Genauigkeit, dass Sie die Fehlerquote um beinahe 40% senken müssen (von 5% Fehlern auf etwa 3%).

Der Algorithmus ist recht einfach: Bei jedem Trainingsschritt wird jedes Neuron (auch die Eingabeneuronen, nicht aber die Ausgabeneuronen) mit einer Wahrscheinlichkeit p zwischenzeitlich »weggelassen.« Es wird also während dieses Trainingsschritts vollständig ignoriert, kann aber im nächsten Schritt wieder aktiv sein (siehe Abbildung 11-9). Den Hyperparameter p nennt man die *Drop-out-Rate*, und er wird normalerweise auf 50% gesetzt. Nach dem Trainieren werden keine Neuronen mehr ausgelassen. Und das ist auch schon alles (bis auf einige technische Details, denen wir uns gleich widmen).

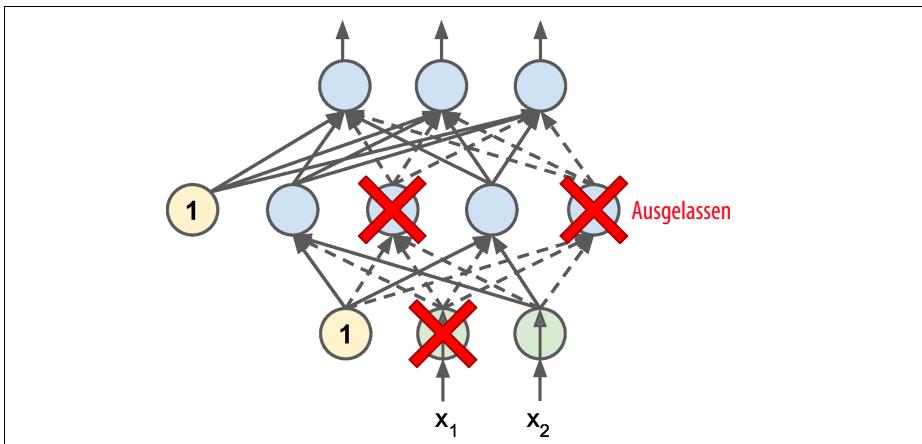


Abbildung 11-9: Drop-out-Regularisierung

Es ist auf den ersten Blick ein wenig überraschend, dass diese doch recht brutale Technik überhaupt funktioniert. Würde ein Unternehmen besser funktionieren, wenn dessen Mitarbeiter jeden Morgen eine Münze werfen, um zu entscheiden, ob sie zur Arbeit gehen? Tja, wer weiß; vielleicht würde es das sogar! Das Unternehmen wäre auf jeden Fall gezwungen, seine Organisation anzupassen; es könnte sich nicht auf eine Einzelperson verlassen, um die Kaffeemaschine zu befüllen oder ähnlich wichtige Aufgaben auszuführen. Die Mitarbeiter müssten lernen, mit vielen ihrer Kollegen zu kooperieren, nicht nur mit einer Handvoll. Das Unternehmen würde dadurch deutlich widerstandsfähiger werden. Falls eine Person kündigt, würde es keinen großen Unterschied machen. Es ist nicht klar, ob diese Idee bei

21 »Improving neural networks by preventing co-adaptation of feature detectors«, G. Hinton et al. (2012).

22 »Dropout: A Simple Way to Prevent Neural Networks from Overfitting«, N. Srivastava et al. (2014).

Unternehmen funktionieren würde, aber bei neuronalen Netzen funktioniert sie mit Sicherheit. Mit Drop-out trainierte Neuronen können sich nicht mit ihren Nachbarneuronen co-adaptieren; sie müssen für sich allein so nützlich wie möglich sein. Sie können sich auch nicht exzessiv auf einige Eingabeneuronen verlassen; sie müssen auf jedes ihrer Eingabeneuronen achten. Dadurch sind sie weniger anfällig für kleine Änderungen der Eingabe. Am Ende erhalten Sie ein robusteres Netz, das besser verallgemeinert.

Eine andere Betrachtungsweise, die die Mächtigkeit der Drop-out-Technik illustriert, ist, dass bei jedem Trainingsschritt ein einzigartiges neuronales Netz generiert wird. Da jedes Neuron entweder anwesend oder abwesend sein kann, gibt es insgesamt 2^N mögliche Netze (wobei N die Gesamtzahl der abschaltbaren Neuronen ist). Aufgrund einer derart großen Zahl ist es praktisch unmöglich, dass das gleiche neuronale Netz doppelt ausgewürfelt wird. Nach 10000 durchgeführten Trainingsschritten haben Sie 10000 unterschiedliche neuronale Netze trainiert (mit jeweils einem Trainingsdatenpunkt). Natürlich sind diese neuronalen Netze nicht voneinander unabhängig, da sie viele Gewichte untereinander teilen, aber sie sind nichtsdestoweniger alle unterschiedlich. Das am Ende erhaltene neuronale Netz lässt sich als Ensemble aus all diesen kleineren Netzen ansehen.

Es gibt ein kleines aber wichtiges technisches Detail. Mit $p = 50\%$ ist ein Neuron beim Testen durchschnittlich mit doppelt so vielen Eingabeneuronen wie beim Trainieren verbunden. Um diesen Umstand zu kompensieren, müssen wir die Gewichte der Eingaben aller Neuronen nach dem Trainieren mit 0.5 multiplizieren. Andernfalls erhält jedes Neuron ein etwa doppelt so großes Eingabesignal und wird vermutlich keine gute Leistung erbringen. Allgemein müssen wir das Gewicht jeder Eingabeverbindung nach dem Trainieren mit der *keep-Wahrscheinlichkeit* ($1 - p$) multiplizieren. Alternativ können wir auch die Ausgabe jedes Neurons beim Trainieren durch die *keep-Wahrscheinlichkeit* teilen (diese beiden Alternativen sind nicht exakt äquivalent, funktionieren aber gleich gut).

Um Drop-out mit TensorFlow zu implementieren, wenden Sie die Funktion `tf.layers.dropout()` auf die Eingabeschichten und/oder die Ausgabe der gewünschten verborgenen Schichten an. Beim Trainieren lässt diese Funktion zufällig einige Neuronen aus (setzt diese auf 0) und teilt die verbliebenen durch die *keep-Wahrscheinlichkeit*. Nach dem Trainieren tut diese Funktion überhaupt nichts. Der folgende Code wendet die Regularisierung mittels Drop-out auf unser dreischichtiges neuronales Netz an:

```
[...]
training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5 # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                            name="hidden1")
```

```

hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                         name="hidden2")
hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")

```



Sie sollten die Funktion `tf.layers.dropout()` und nicht `tf.nn.dropout()` verwenden. Erstere schaltet sich außerhalb des Trainings ab (no-op), was hier erwünscht ist, Letztere nicht.

Natürlich müssen Sie wie bei der Batch-Normalisierung beim Trainieren `training` auf `True` setzen und beim Testen den voreingestellten Wert `False` beibehalten.

Wenn Sie Overfitting beobachten, können Sie die Drop-out-Rate erhöhen. Wenn dagegen Underfitting der Trainingsdaten vorliegt, sollten Sie die Drop-out-Rate senken. Bei großen Schichten hilft das Erhöhen der Drop-out-Rate ebenfalls, und bei kleinen Schichten das Verringern.

Drop-out verlangsamt die Konvergenz erheblich, dafür ist das erhaltene Modell mit den richtigen Einstellungen in der Regel sehr viel besser. Der zusätzliche Zeit- und Vorbereitungsaufwand zahlt sich also grundsätzlich aus.



Dropconnect ist eine Variante der Drop-out-Technik, bei der einzelne Verbindungen anstelle von ganzen Neuronen zufällig ausgelassen werden. Im Allgemeinen erbringt Drop-out eine höhere Leistung.

Max-Norm-Regularisierung

Eine weitere bei neuronalen Netzen recht verbreitete Regularisierungstechnik ist die *Max-Norm Regularisierung*: Bei jedem Neuron werden die Gewichte der eingehenden Verbindungen \mathbf{w} so eingeschränkt, dass $\|\mathbf{w}\|_2 \leq r$, wobei r der Max-Norm-Hyperparameter und $\|\cdot\|_2$ die ℓ_2 -Norm ist.

Üblicherweise wird diese Bedingung durch Berechnen von $\|\mathbf{w}\|_2$ nach jedem Trainingsschritt und bei Bedarf Kappen von \mathbf{w} umgesetzt ($\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|\mathbf{w}\|_2}$).

Die Regularisierung wird mit einem kleineren Wert für r stärker und bekämpft das Overfitting. Die Max-Norm-Regularisierung wirkt außerdem dem Problem schwindender/explodierender Gradienten entgegen (wenn Sie keine Batch-Normalisierung verwenden).

TensorFlow enthält keine eingebaute Max-Norm-Regularisierung, aber sie ist nicht schwer zu implementieren. Der folgende Code beschafft sich eine Referenz auf die Gewichte der ersten verborgenen Schicht und verwendet dann die Funktion `clip_by_norm()`. Diese erstellt eine Operation, die die Gewichte entlang der zweiten Achse so abschneidet, dass jeder Zeilenvektor maximal eine Norm von 1.0 besitzt.

Die letzte Zeile erzeugt eine Zuweisungsoperation, mit der die abgeschnittenen Gewichte der Gewichtsvariablen zugewiesen werden:

```
threshold = 1.0
weights = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

Anschließend wenden Sie diese Operation nach jedem Trainingsschritt folgendermaßen an:

```
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
clip_weights.eval()
```

Im Allgemeinen sollten Sie dies für jede verborgene Schicht vornehmen. Obwohl diese Lösung gut funktioniert, ist sie ein wenig chaotisch. Eine sauberere Lösung ist, eine Funktion `max_norm_regularizer()` zu schreiben und diese wie die Funktion `l1_regularizer()` zu verwenden:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None # es gibt keinen Term für den Regularisierungsverlust
    return max_norm
```

Diese Funktion liefert eine parametrisierte `max_norm()`-Funktion zurück, die Sie wie jeden anderen Regularisierer verwenden können:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                            kernel_regularizer=max_norm_reg, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                            kernel_regularizer=max_norm_reg, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Beachten Sie, dass Sie bei der Max-Norm-Regularisierung keinen Term für den Regularisierungsverlust zu Ihrer Verlustfunktion hinzufügen müssen. Deshalb gibt die Funktion `max_norm()` den Wert `None` zurück. Sie müssen aber trotzdem noch nach jedem Trainingsschritt die Operationen `clip_weights` ausführen, sodass Sie eine Referenz auf diese erhalten. Deshalb fügt die Funktion `max_norm()` die Operation `clip_weights` zu einer Kollektion von Max-Norm-Clipping-Operationen hinzu. Diese Operationen müssen nach jedem Trainingsschritt ausgeführt werden:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
```

```
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})  
sess.run(clip_all_weights)
```

Der Code ist so *viel* ordentlicher, nicht wahr?

Data Augmentation

Eine letzte Regularisierungstechnik, Data Augmentation, besteht aus dem Generieren neuer Trainingsdaten aus den bestehenden, sodass der Trainingsdatensatz künstlich vergrößert wird. Dies wirkt Overfitting entgegen und zählt daher als Regularisierungstechnik. Der Trick besteht darin, realistische Trainingsdaten zu erzeugen; idealerweise sollte ein menschlicher Betrachter nicht erkennen können, welche Datenpunkte generiert wurden und welche nicht. Außerdem hilft es nicht, einfach nur Rauschen hinzuzufügen; die Modifikationen sollten auch noch erlernbar sein (Rauschen ist es nicht).

Wenn Ihr Modell beispielsweise Bilder von Pilzen klassifizieren soll, können Sie jedes Bild im Trainingsdatensatz ein wenig verschieben, drehen und die Größe verändern. Anschließend fügen Sie die so erhaltenen Bilder dem Trainingsdatensatz hinzu (siehe Abbildung 11-10). Dies zwingt das Modell, in Bezug auf Position, Ausrichtung und Größe der Pilze auf den Bildern toleranter zu sein. Wenn Ihr Modelle der Belichtung gegenüber toleranter sein soll, können Sie ebenso Bilder mit unterschiedlichem Kontrast generieren. Vorausgesetzt, dass die Pilze symmetrisch sind, können Sie die Bilder auch horizontal spiegeln. Indem Sie diese Transformationen miteinander kombinieren, können Sie Ihren Trainingsdatensatz erheblich vergrößern.

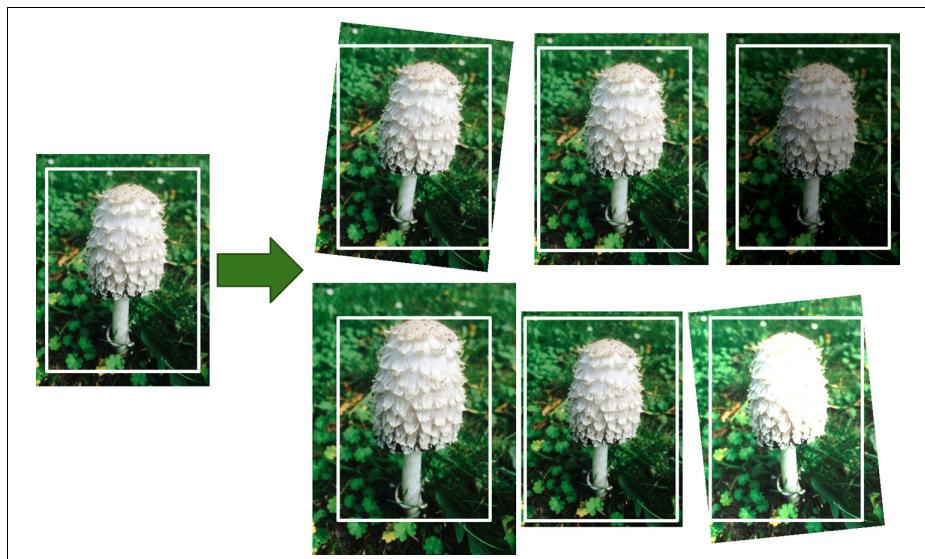


Abbildung 11-10: Generieren neuer Trainingsdatenpunkte aus den bestehenden

Es ist meist günstiger, Trainingsdaten erst beim Trainieren zu erzeugen, anstatt Speicherplatz und Netzwerkbandbreite zu vergeuden. TensorFlow enthält verschiedene Operationen, um Bilder zu transponieren (verschieben), zu rotieren, zu skalieren, zu spiegeln, zurechtzuschneiden und Werte wie Helligkeit, Kontrast, Sättigung und Farbton zu verändern (Details finden Sie in der Dokumentation der API). Diese Operationen erleichtern das Implementieren von Data Augmentation für Bilddatensätze.



Eine weitere mächtige Technik zum Trainieren sehr tiefer Deep-Learning-Netze sind *Skip-Verbindungen* (bei einer Skip-Verbindung wird die Eingabe einer Schicht zur Ausgabe einer höheren Schicht hinzugefügt). Wir werden diesen Gedanken in Kapitel 13 beim Besprechen von Deep Residual Networks fortführen.

Praktische Tipps

In diesem Kapitel haben wir eine Vielzahl Techniken besprochen. Sie fragen sich vielleicht, welche davon Sie verwenden sollen. Die in Tabelle 11-2 aufgeführte Konfiguration funktioniert in den meisten Fällen gut.

Tabelle 11-2: Standardkonfiguration eines DNN

Initialisierung	Initialisierung nach He
Aktivierungsfunktion	ELU
Normalisierung	Batch-Normalisierung
Regularisierung	Drop-out
Optimierer	Beschleunigter Gradient nach Nesterov
Scheduler für die Lernrate	keiner

Natürlich sollten Sie Teile eines vortrainierten neuronalen Netzes wiederverwenden, wenn Sie eines auftreiben können, das ein ähnliches Problem löst.

Diese Standardkonfiguration muss angepasst werden:

- Wenn Sie keine gute Lernrate finden (die Konvergenz war zu langsam, also haben Sie die Lernrate erhöht; nun konvergiert das Netz schnell, aber die Genauigkeit ist suboptimal), können Sie einen Scheduler für die Lernrate einsetzen, z.B. den exponentiellen Zerfall.
- Wenn Ihr Trainingsdatensatz ein wenig zu klein ist, können Sie Data Augmentation einsetzen.
- Wenn Sie ein spärliches Modell benötigen, können Sie ℓ_1 -Regularisierung hinzufügen (und nach dem Trainieren eventuell sehr kleine Gewichte auf null setzen). Wenn Sie ein noch spärlicheres Modell benötigen, können Sie anstelle der Adam-Optimierung auch FTRL zusammen mit der ℓ_1 -Regularisierung ausprobieren.

- Wenn Sie ein zur Laufzeit blitzschnelles Modell benötigen, können Sie die Batch-Normalisierung auslassen und die ELU-Aktivierungsfunktion durch Leaky ReLU ersetzen. Ein spärliches Modell hilft ebenfalls.

Mit diesen Richtlinien sind Sie bereit, sehr tiefe Netze zu trainieren – d.h., wenn Sie sehr viel Geduld haben! Auf einem einzelnen Computer müssen Sie unter Umständen Tage oder sogar Monate warten, bis das Trainieren beendet ist. Im nächsten Kapitel besprechen wir, wie Sie TensorFlow im verteilten Betrieb verwenden können, um Modelle auf mehrere Server und GPUs verteilt zu trainieren und zu verwenden.

Übungen

1. Ist es in Ordnung, die Bias-Terme mit 0 zu initialisieren?
2. Ist es in Ordnung, sämtliche Gewichte mit dem gleichen Wert zu initialisieren, solange dieser Wert mittels Initialisierung nach He zufällig ausgewählt wird?
3. Nennen Sie drei Vorteile der ELU-Aktivierungsfunktion gegenüber ReLU.
4. In welchen Fällen würden Sie die folgenden Aktivierungsfunktionen verwenden: ELU, Leaky ReLU (und Varianten), ReLU, tanh, logistische und Softmax?
5. Was passiert, wenn Sie beim Verwenden des MomentumOptimizer den Hyperparameter `momentum` zu nah an 1 setzen (z.B. 0.99999)?
6. Nennen Sie drei Möglichkeiten, ein dünn besetztes Modell zu erstellen.
7. Wird das Trainieren durch Drop-out langsamer? Wird die Inferenz langsamer (die Vorhersage auf neuen Datenpunkten)?
8. Deep Learning.
 - a. Erstellen Sie ein DNN mit fünf verborgenen Schichten aus jeweils 100 Neuronen. Verwenden Sie die Initialisierung nach He und ELU als Aktivierungsfunktion.
 - b. Trainieren Sie das Netz mit Adam-Optimierung und Early Stopping auf den MNIST-Daten, aber nur auf den Ziffern 0 bis 4. Wir werden in der nächsten Übung Transfer Learning für die Ziffern 5 bis 9 einsetzen. Sie benötigen eine Ausgabeschicht mit Softmax und fünf Neuronen. Sichern Sie den Zwischenstand in regelmäßigen Abständen und das endgültige Modell, sodass Sie es später wiederverwenden können.
 - c. Optimieren Sie die Hyperparameter mittels Kreuzvalidierung und schauen Sie, welche Relevanz sie erreichen.
 - d. Fügen Sie nun Batch-Normalisierung hinzu und vergleichen Sie die Lernkurven: Konvergiert das Netz schneller als zuvor? Ist das entstehende Modell besser?

e. Findet Overfitting der Trainingsdaten statt? Versuchen Sie, bei jeder Schicht, Drop-out hinzuzufügen. Hilft das?

9. Transfer Learning.

- a. Erstellen Sie ein neues DNN, das alle vortrainierten verborgenen Schichten des vorigen Modells verwendet, einfriert und die Softmax-Ausgabeschicht durch eine neue ersetzt.
- b. Trainieren Sie dieses neue DNN auf den Ziffern 5 bis 9 mit nur 100 Bildern pro Ziffer. Stoppen Sie die Zeit. Können Sie trotz der kleinen Anzahl an Beispielen eine hohe Präzision erzielen?
- c. Versuchen Sie die eingefrorenen Schichten zu cachen und trainieren Sie das Modell erneut: Wie schnell ist es diesmal?
- d. Versuchen Sie es noch einmal, indem Sie nur vier statt fünf verborgene Schichten verwenden. Können Sie eine höhere Präzision erzielen?
- e. Schalten Sie nun die obersten zwei eingefrorenen Schichten hinzu und setzen Sie das Trainieren fort: Wird das Modell hierdurch noch besser?

10. Vortrainieren auf einer Hilfsaufgabe.

- a. In dieser Übung erstellen Sie ein DNN, das zwei Bilder von MNIST-Ziffern miteinander vergleicht und vorhersagt, ob sie die gleiche Ziffer enthalten oder nicht. Anschließend werden Sie die unteren Schichten dieses Netzes wiederverwenden, um einen MNIST-Klassifikator mit sehr wenigen Trainingsdaten zu erstellen. Beginnen Sie, indem Sie zwei DNNs erstellen (nennen wir sie DNN A und B). Beide sollten zu den zuvor erstellten ähnlich sein, aber keine Ausgabeschicht enthalten: Jedes DNN soll fünf verborgene Schichten mit je 100 Neuronen enthalten, die Initialisierung nach He und ELU als Aktivierungsfunktion verwenden. Fügen Sie anschließend beiden DNNs eine einzelne Ausgabeschicht hinzu. Verwenden Sie die TensorFlow-Funktion `concat()` mit `axis=1`, um die Ausgaben beider DNNs entlang der horizontalen Achse miteinander zu verbinden. Geben Sie das Ergebnis in die Ausgabeschicht ein. Diese Ausgabeschicht sollte aus einem einzelnen Neuron mit logistischer Aktivierungsfunktion bestehen.
- b. Teilen Sie den MNIST-Datensatz in zwei Teile auf: Teil #1 besteht aus 55000 Bildern, Teil #2 aus 5000 Bildern. Erstellen Sie eine Funktion, die einen Trainings-Batch erstellt, wobei jeder Datenpunkt aus einem Paar MNIST-Bildern aus Teilmenge #1 besteht. Die Hälfte der Trainingsdaten sollte aus Bildpaaren mit der gleichen Ziffer bestehen, die andere Hälfte aus jeweils unterschiedlichen Ziffern. Setzen Sie als Labels dieser Paare 0 ein, falls die Bilder die gleiche Ziffer zeigen, und 1, falls sie unterschiedlichen Kategorien angehören.
- c. Trainieren Sie das DNN auf diesem Trainingsdatensatz. Bei jedem Bildpaar können Sie gleichzeitig das erste Bild in DNN A und das zweite Bild

in DNN B einspeisen. Das gesamte Netz lernt nach und nach, ob zwei Bilder der gleichen Kategorie angehören oder nicht.

- d. Erstellen Sie ein neues DNN, indem Sie die verborgenen Schichten von DNN A wiederverwenden und einfrieren. Fügen Sie eine Softmax-Ausgangsschicht mit 10 Neuronen hinzu. Trainieren Sie dieses Netz auf Teilmenge #2 und schauen Sie, ob Sie eine hohe Vorhersagequalität erzielen, auch wenn es pro Kategorie nur 500 Bilder gibt.

Lösungen zu diesen Übungen finden Sie in Anhang A.

TensorFlow über mehrere Geräte und Server verteilen

In Kapitel 11 haben wir mehrere Techniken erwähnt, die das Trainieren erheblich beschleunigen können: bessere Initialisierung der Gewichte, Batch-Normalisierung, ausgefeilte Optimierer und so weiter. Allerdings kann das Trainieren eines großen neuronalen Netzes auf einem einzelnen Computer mit einer einzelnen CPU trotz dieser Techniken Tage oder sogar Wochen dauern.

In diesem Kapitel werden wir uns ansehen, wie sich Berechnungen mit TensorFlow auf mehrere Geräte (CPUs und GPUs) verteilen und parallel ausführen lassen (siehe Abbildung 12-1). Zuerst werden wir Berechnungen auf mehrere Recheneinheiten im gleichen Computer verteilen, anschließend auf mehrere Geräte auf mehreren Computern.

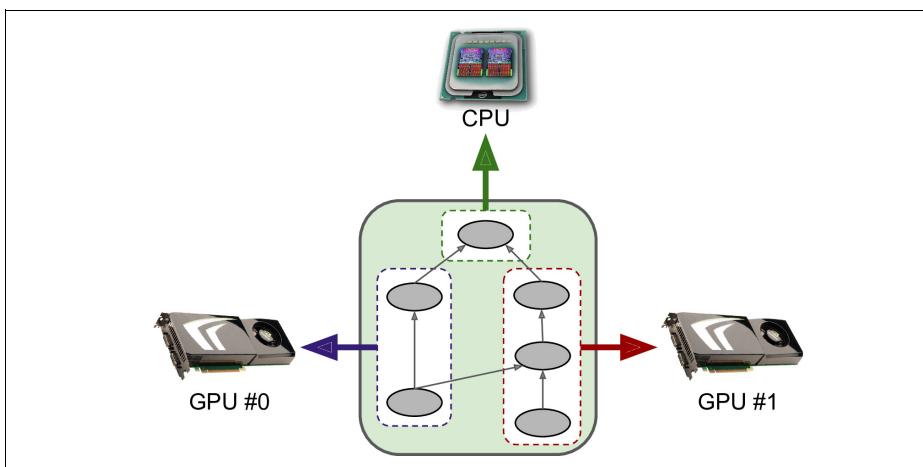


Abbildung 12-1: Paralleles Ausführen eines TensorFlow-Graphen auf mehreren Geräten

Verglichen mit anderen Frameworks für neuronale Netze ist die Unterstützung für verteiltes Rechnen eines der herausstechenden Merkmale von TensorFlow. Es gibt Ihnen vollständige Kontrolle darüber, wie Sie Ihren Berechnungsgraphen auf Geräte oder Server aufteilen (oder replizieren). Sie können Operationen flexibel

parallelisieren und synchronisieren, sodass Sie die Wahl zwischen allen möglichen Ansätzen zur Parallelisierung haben.

Wir werden uns einige der beliebtesten Ansätze ansehen, mit denen sich das Ausführen und Trainieren eines neuronalen Netzes parallelisieren lässt. Anstatt wochenlang auf einen Trainingsalgorithmus zu warten, müssen Sie sich vielleicht nur noch einige Stunden gedulden. Damit sparen Sie nicht nur enorm viel Zeit, sondern können auch leichter mit unterschiedlichen Modellen experimentieren und Ihre Modelle häufiger mit neuen Daten trainieren.

Weitere nützliche Anwendungen der Parallelisierung sind das Erkunden eines sehr viel größeren Hyperparameterraumes bei der Feinabstimmung Ihres Modells und das effiziente Ausführen großer Ensembles neuronaler Netze.

Bevor wir allerdings rennen, müssen wir lernen zu laufen. Parallelisieren wir zunächst einige einfache Graphen auf mehreren GPUs auf dem gleichen Computer.

Mehrere Recheneinheiten auf einem Computer

Ein großer Geschwindigkeitsgewinn lässt sich durch das bloße Hinzufügen von GPU-Karten zu einem Rechner erreichen. Oft ist das schon ausreichend, und Sie müssen sich dann erst gar nicht um mehrere Rechner kümmern. Beispielsweise können Sie ein neuronales Netzwerk mit 8 GPUs auf dem gleichen Computer etwa genauso schnell trainieren wie 16 GPUs auf mehreren Computern (aufgrund der zusätzlichen Verzögerung durch das Netzwerk bei mehreren Computern).

In diesem Abschnitt werden wir unsere Umgebung so einrichten, dass TensorFlow mehrere GPU-Karten auf einem Computer nutzen kann. Anschließend werden wir uns ansehen, wie Sie Operationen auf die verfügbaren Recheneinheiten verteilen und parallel ausführen können.

Installation

Um TensorFlow auf mehreren GPUs auszuführen, müssen Sie zunächst sicherstellen, dass Ihre GPU-Karten Nvidia Compute Capability unterstützen (größer oder gleich 3.0). Darunter fallen die Nvidia-Karten Titan, Titan X, K20 und K40 (wenn Sie eine andere Grafikkarte besitzen, können Sie die Kompatibilität unter <https://developer.nvidia.com/cuda-gpus> überprüfen).



Wenn Sie keine GPU-Karte besitzen, können Sie einen GPU-fähigen Hoster wie Amazon AWS verwenden. Detaillierte Anweisung zum Aufsetzen von TensorFlow 0.9 mit Python 3.5 auf einer Amazon-AWS-GPU-Instanz finden Sie in einem hilfreichen Blogbeitrag (<http://goo.gl/kbge5b>) von Źiga Avsec. Es sollte nicht allzu schwer sein, ein Update auf die neueste Version von TensorFlow durchzuführen. Google hat außerdem einen Cloud-Service

namens *Cloud Machine Learning* (<https://cloud.google.com/ml>) zum Ausführen von TensorFlow-Graphen entwickelt. Im Mai 2016 wurde bekannt gegeben, dass diese Plattform nun Server mit *Tensor Processing Units* (TPUs) enthält, auf Machine Learning spezialisierte Prozessoren, die bei vielen ML-Aufgaben deutlich schneller als GPUs sind. Natürlich können Sie sich auch einfach eine eigene GPU-Karte kaufen. Tim Dettmers hat einen sehr guten Blogbeitrag (<https://goo.gl/pCtSAn>) geschrieben, der Ihnen bei der Auswahl hilft. Er aktualisiert diesen regelmäßig.

Sie müssen anschließend die richtigen Versionen der Bibliotheken CUDA und cuDNN herunterladen und installieren (CUDA 8.0 und cuDNN 5.1, falls Sie die Binärinstallation von TensorFlow 1.0.0 verwenden) sowie einige Umgebungsvariablen setzen, damit TensorFlow sowohl CUDA als auch cuDNN findet. Die Installationsanleitung ändert sich vermutlich recht häufig, daher folgen Sie am besten den Anweisungen auf der TensorFlow-Webseite.

Die Nvidia-Bibliothek *Compute Unified Device Architecture* (CUDA) ermöglicht Entwicklern die Nutzung von CUDA-fähigen GPUs für alle möglichen Berechnungen (nicht nur zum Beschleunigen von Grafik). Die Nvidia-Bibliothek *CUDA Deep Neural Network* (cuDNN) ist eine GPU-beschleunigte Bibliothek mit Grundbausteinen für DNNs. Sie enthält optimierte Implementierungen bei DNNs verbreiteter Berechnungen wie Aktivierungsschichten, Normalisierung, Vorwärts- und Rückwärts-Convolutions sowie Pooling (siehe Kapitel 13). Sie ist ein Teil des Nvidia-Deep-Learning-SDK (zum Herunterladen müssen Sie sich ein Nvidia-Entwicklerkonto anlegen). TensorFlow verwendet CUDA und cuDNN, um die GPU-Karten zu steuern und Berechnungen zu beschleunigen (siehe Abbildung 12-2).

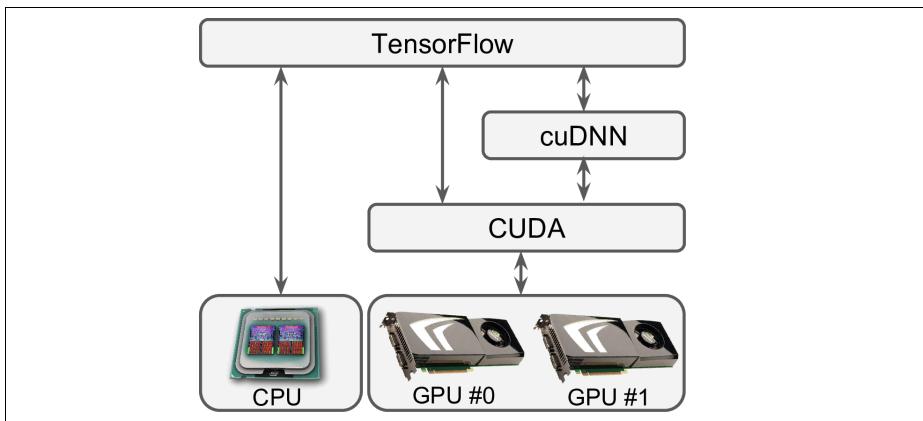


Abbildung 12-2: Tensorflows Verwendung von CUDA und cuDNN

Sie können mit dem Befehl `nvidia-smi` prüfen, ob CUDA korrekt installiert ist. Dieser gibt die verfügbaren GPU-Karten und die auf jeder Karte laufenden Prozesse aus:

```
$ nvidia-smi
Wed Sep 16 09:50:03 2016
+-----+
| NVIDIA-SMI 352.63     Driver Version: 352.63    |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====
|   0  GRID K520        Off | 0000:00:03.0    Off |                  N/A |
| N/A   27C   P8    17W / 125W |      11MiB / 4095MiB |      0%     Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU     PID  Type  Process name          Usage      |
| =====+=====+=====+=====
| No running processes found               |
+-----+
```

Schließlich müssen Sie TensorFlow mit GPU-Unterstützung installieren. Wenn Sie eine isolierte Umgebung mit virtualenv erstellt haben, müssen Sie diese zuerst aktivieren:

```
$ cd $ML_PATH           # Ihr ML-Arbeitsverzeichnis (z.B. $HOME/ml)
$ source env/bin/activate
```

Anschließend installieren Sie die entsprechende GPU-fähige Version von TensorFlow:

```
$ pip3 install --upgrade tensorflow-gpu
```

Nun können Sie eine Python-Shell öffnen und überprüfen, ob TensorFlow CUDA und cuDNN richtig erkennt, indem Sie TensorFlow importieren und eine Session erstellen:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0: Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Es sieht gut aus! TensorFlow hat die CUDA und cuDNN-Bibliotheken und mit der CUDA-Bibliothek die GPU-Karte erkannt (in diesem Fall eine Nvidia-Grid-K520-Karte).

Das RAM der GPU verwalten

TensorFlow beansprucht automatisch das gesamte verfügbare RAM aller GPUs, wenn Sie einen Graphen das erste Mal ausführen. Daher können Sie kein zweites TensorFlow-Programm starten, während das erste noch läuft. Wenn Sie es dennoch versuchen, erhalten Sie folgenden Fehler:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from device: CUDA_ERROR_OUT_OF_MEMORY
```

Wir können stattdessen jeden Prozess auf einer anderen GPU-Karte laufen lassen. Dies geht am einfachsten, indem Sie die Umgebungsvariable `CUDA_VISIBLE_DEVICES` so setzen, dass jeder Prozess nur die gewünschten GPU-Karten sieht. Beispielsweise könnten Sie zwei Programme folgendermaßen starten:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py  
# und in einem zweiten Terminal:  
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Programm #1 wird nur die GPU-Karten 0 und 1 wahrnehmen (mit den Nummern 0 und 1). Programm #2 wird nur die GPU-Karten 2 und 3 wahrnehmen (mit den Nummern 1 und 0). Alles sollte gut funktionieren (siehe Abbildung 12-3).

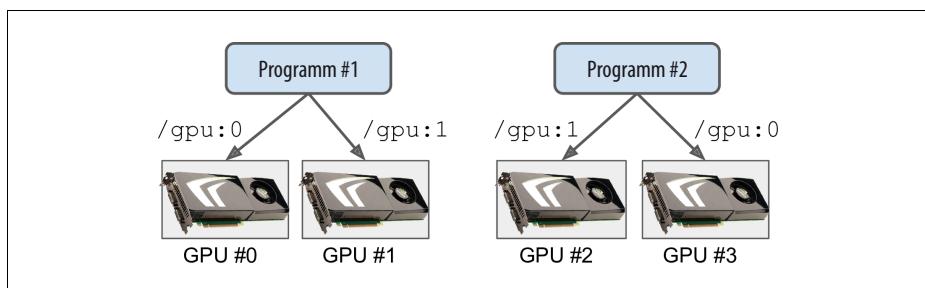


Abbildung 12-3: Jedes Programm erhält zwei eigene GPUs.

Als Alternative können Sie TensorFlow anweisen, nur einen Teil des Speichers zu verwenden. Beispielsweise können Sie nur 40% des Speichers jeder GPU verwenden, indem Sie ein `ConfigProto`-Objekt erzeugen, dessen Option `gpu_options.per_process_gpu_memory_fraction` auf 0.4 setzen und die Session mit dieser Konfiguration erstellen:

```
config = tf.ConfigProto()  
config.gpu_options.per_process_gpu_memory_fraction = 0.4  
session = tf.Session(config=config)
```

Nun lassen sich zwei solche Programme mit den gleichen GPU-Karten parallel ausführen (nicht aber drei, da $3 \times 0.4 > 1$ ist). Siehe Abbildung 12-4.

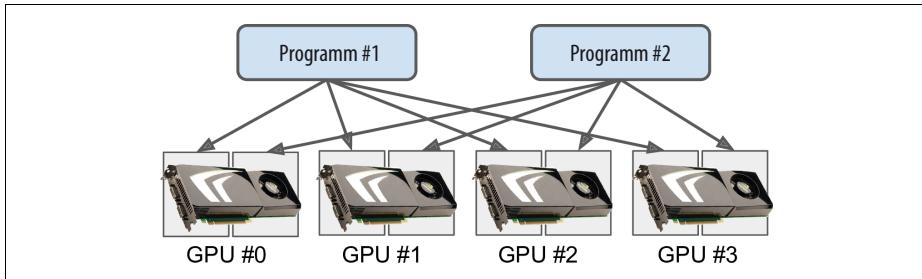


Abbildung 12-4: Jedes Programm verwendet alle vier GPUs, aber nur mit jeweils 40% des RAM.

Wenn Sie den Befehl `nvidia-smi` ausführen, während beide Programme laufen, sollten Sie sehen, dass jeder Prozess etwa 40% des insgesamt auf jeder Karte verfügbaren RAM verwendet:

```
$ nvidia-smi
[...]
+-----+
| Processes:
| GPU     PID  Type  Process name          GPU Memory |
|           Usage
| =====
|   0      5231  C    python                 1677MiB |
|   0      5262  C    python                 1677MiB |
|   1      5231  C    python                 1677MiB |
|   1      5262  C    python                 1677MiB |
[...]
```

Sie können auch einen anderen Weg wählen und TensorFlow anweisen, Speicher nur dann zu belegen, wenn er gebraucht wird. Dazu müssen Sie `config.gpu_options.allow_growth` auf `True` setzen. Allerdings gibt TensorFlow einmal belegten Speicher grundsätzlich nicht frei (um eine Fragmentierung des Speichers zu vermeiden), daher kann nach einer Weile der verfügbare freie Speicher dennoch knapp werden. Es ist schwieriger, bei dieser Option ein vorhersagbares Verhalten zu garantieren, daher sollten Sie im Allgemeinen bei einer der vorigen Möglichkeiten bleiben.

Okay, Sie haben nun eine lauffähige TensorFlow-Installation mit GPU-Unterstützung. Schauen wir uns an, wie Sie diese verwenden!

Operationen auf Recheneinheiten platzieren

Das TensorFlow Whitepaper (<http://goo.gl/vSjA14>)¹ stellt einen *dynamischen Platzierungsalgorithmus* vor, der Operationen automatisch auf sämtliche verfügbaren Recheneinheiten verteilt und dabei viele Faktoren berücksichtigt: die in vorigen Ausführungsrunden dieses Graphen gemessene Rechenzeit, Schätzungen der

¹ »TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems«, Google Research (2015).

Größe von Ein- und Ausgabetensoren jeder Operation, die auf jedem Gerät verfügbare Menge RAM, die Verzögerung bei der Übertragung von Daten von und zum Gerät sowie vom Nutzer angegebene Hinweise und Einschränkungen. Leider ist dieser ausgefeilte Algorithmus ein internes Werk von Google; es wurde nicht zusammen mit der Open-Source-Version von TensorFlow veröffentlicht. Der Grund dafür scheint zu sein, dass in der Praxis eine kleine Menge vom Nutzer angegebener Platzierungsregeln zu einer effizienteren Platzierung führt als die vom dynamischen Platzierungsverfahren berechnete. Allerdings arbeitet das TensorFlow-Team an einer Verbesserung der dynamischen Platzierung. Vielleicht wird sie eines Tages gut genug für die Öffentlichkeit sein.

Bis dahin verlässt sich TensorFlow auf die *einfache Platzierung*, die (ihrem Namen entsprechend) wirklich sehr einfach ist.

Einfache Platzierung

Wann immer Sie einen Graphen ausführen und TensorFlow einen Knoten auswerten muss, der noch auf keiner Recheneinheit platziert wurde, verwendet es die einfache Platzierung für die Zuordnung, genauso wie für alle anderen noch nicht platzierten Knoten. Das einfache Platzierungsverfahren berücksichtigt folgende Regeln:

- Wenn ein Knoten in einer vorigen Ausführungsrounde des Graphen bereits auf einer Recheneinheit platziert wurde, bleibt er dort.
- Wenn stattdessen ein Nutzer einen Knoten auf eine Recheneinheit *gepinnt* hat (wird gleich erklärt), wird er auf diesem Gerät platziert.
- Andernfalls landet er standardmäßig auf GPU #0 oder auf der CPU, falls es keine GPU gibt.

Wie Sie sehen, ist das Platzieren von Operationen auf den richtigen Recheneinheiten weitgehend Ihnen überlassen. Wenn Sie nichts weiter tun, wird der gesamte Graph auf dem Standardgerät platziert. Um Knoten auf eine Recheneinheit zu pinnen, müssen Sie einen Geräteblock mit der Funktion `device()` erstellen. Das folgende Codebeispiel pinnt die Variable `a` und die Konstante `b` auf die CPU, aber der Multiplikationsknoten `c` wird auf keine bestimmte Recheneinheit gepinnt. Er wird daher auf dem Standardgerät platziert:

```
with tf.device("/cpu:0"):  
    a = tf.Variable(3.0)  
    b = tf.constant(4.0)  
  
    c = a * b
```



Die Recheneinheit "/cpu:0" steht für sämtliche CPUs eines Systems mit mehreren CPUs. Es gibt im Moment keine Möglichkeit, Knoten auf bestimmte CPUs zu pinnen oder nur einen Teil aller vorhandenen CPUs zu verwenden.

Loggen von Platzierungen

Überprüfen wir, ob das einfache Platzierungsverfahren die soeben gesetzten Bedingungen berücksichtigt. Dazu setzen wir den Parameter `log_device_placement` auf `True`; dadurch wird jedes Mal beim Platzieren eines Knotens eine Log-Nachricht geschrieben. Zum Beispiel:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

Die Zeilen mit "I" (für Info) sind die Log-Nachrichten. Sobald wir eine Session erstellen, schreibt TensorFlow eine Nachricht, dass es eine GPU-Karte gefunden hat (in diesem Fall die Grid-K520-Karte). Wenn wir den Graphen zum ersten Mal ausführen (beim Initialisieren der Variable `a`), weist der Platzierungsalgorithmus jeden Knoten der zugeordneten Recheneinheit zu. Wie erwartet tauchen alle Knoten auf `/cpu:0` auf, nur der Knoten mit der Multiplikation landet auf der Standard-Recheneinheit `/gpu:0` (das Präfix `/job:localhost/replica:0/task:0` können Sie im Moment ignorieren; wir werden es gleich besprechen). Wenn wir den Graphen zum zweiten Mal ausführen (um `c` zu berechnen), findet gar keine Platzierung statt, weil alle von TensorFlow zur Berechnung von `c` nötigen Knoten bereits platziert wurden.

Dynamische Platzierung

Wenn Sie einen `device`-Block definieren, können Sie statt des Namens einer Recheneinheit eine Funktion angeben. TensorFlow ruft diese Funktion für jede in diesem Block zu platzierende Operation auf, und die Funktion muss den Namen einer Recheneinheit zurückgeben, auf die die Operation gepinnt werden soll. Das folgende Codebeispiel pinnt alle Knoten mit Variablen auf `/cpu:0` (in diesem Fall nur die Variable `a`) und alle anderen Knoten auf `/gpu:0`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"
```

```

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b

```

Sie können leicht komplexere Algorithmen implementieren, etwa Variablen der Reihe nach auf die GPUs verteilen.

Operationen und Kernels

Um eine TensorFlow-Operation auf einer Recheneinheit auszuführen, ist eine Implementierung für dieses Gerät nötig; dies nennt man einen *Kernel*. Für viele Operationen existieren sowohl CPU-Kernel als auch GPU-Kernel, aber nicht für alle. Beispielsweise enthält TensorFlow keinen GPU-Kernel für Integer-Variablen. Deshalb scheitert der folgende Code, sobald TensorFlow versucht, die Variable i auf GPU #0 zu platzieren:

```

>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification

```

Beachten Sie, dass TensorFlow den Typ int32 zuordnet, da sie mit einer Integerzahl initialisiert wurde. Wenn Sie eine Variable mit 3.0 anstatt 3 initialisieren, ihren Typ beim Erstellen der Variablen explizit auf `dtype=tf.float32` setzen, funktioniert alles prima.

Soft Placement

Wenn Sie eine Operation auf einer Recheneinheit platzieren, auf der es für diese Operation keinen Kernel gibt, erhalten Sie standardmäßig den oben gezeigten Ausnahmefehler. Wenn Sie bevorzugen, dass TensorFlow in diesem Fall auf die CPU ausweicht, können Sie den Konfigurationsparameter `allow_soft_placement` auf True setzen:

```

with tf.device("/gpu:0"):
    i = tf.Variable(3)

    config = tf.ConfigProto()
    config.allow_soft_placement = True
    sess = tf.Session(config=config)
    sess.run(i.initializer) # der Platzierer weicht auf /cpu:0 aus

```

Wir haben bisher besprochen, wie sich Knoten auf unterschiedlichen Recheneinheiten platzieren lassen. Schauen wir uns als Nächstes an, wie sich diese Knoten parallel ausführen lassen.

Paralleles Ausführen

Wenn TensorFlow einen Graphen ausführt, ermittelt es zuerst eine Liste der auszuwertenden Knoten und zählt, wie viele Abhängigkeiten jeder besitzt. Anschließend beginnt TensorFlow, Knoten ohne Abhängigkeiten auszuwerten (die Source-Knoten). Wenn diese Knoten auf unterschiedlichen Recheneinheiten platziert wurden, werden Sie natürlich parallel ausgeführt. Falls Sie auf der gleichen Recheneinheit platziert wurden, werden sie in unterschiedlichen Threads ausgewertet, sodass ein paralleles Ausführen ebenfalls möglich ist (in unterschiedlichen GPU-Threads oder CPU-Cores).

TensorFlow verwaltet auf jeder Recheneinheit einen Thread Pool, um Operationen zu parallelisieren (siehe Abbildung 12-5). Diese bezeichnet man als *inter-Op Thread Pools*. Einige Operationen haben Kernels mit mehreren Threads: Diese können andere Thread Pools verwenden (einen pro Recheneinheit), die *intra-Op Thread Pools* genannt werden.

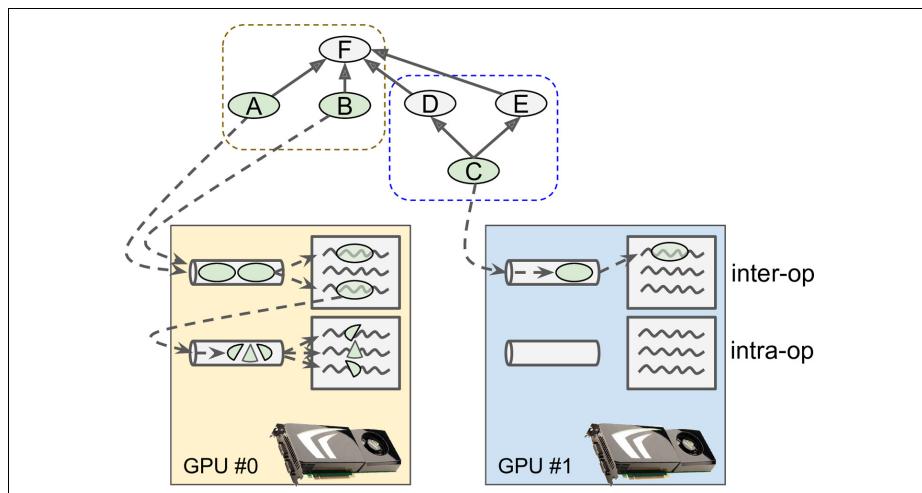


Abbildung 12-5: Parallele Ausführung eines TensorFlow-Graphen

Beispielsweise sind Operationen A, B und C in Abbildung 12-5 Source-Ops, sie können also unmittelbar ausgewertet werden. Operationen A und B werden auf GPU #0 platziert, beanspruchen also den inter-Op Thread Pool dieser Recheneinheit. Sie werden unmittelbar parallel ausgeführt. Operation A besitzt einen Kernel mit mehreren Threads; die Berechnung wird in drei Teile aufgeteilt, die vom intra-Op Thread Pool ausgeführt werden. Operation C belegt den inter-Op Thread Pool von GPU #1.

Sobald Operation C beendet ist, werden die Abhängigkeitszähler der Operationen D und E um eins verringert. Beide erreichen 0, daher werden beide Operationen zum inter-Op Thread Pool zum Ausführen geschickt.



Sie können die Anzahl der Threads pro inter-Op Pool steuern, indem Sie die Option `inter_op_parallelism_threads` setzen. Beachten Sie, dass die inter-Op Thread Pools beim Starten der ersten Session erstellt werden. Alle weiteren Sessions verwenden sie einfach weiter, es sei denn, Sie setzen den Parameter `use_per_session_threads` auf True. Die Anzahl der Threads pro intra-Op Pool lässt sich über die Option `intra_op_parallelism_threads` steuern.

Control Dependencies

Manchmal ist es schlauer, die Evaluation einer Operation aufzuschieben, obwohl alle Operationen, von denen sie abhängt, bereits ausgeführt wurden. Wenn sie beispielsweise eine Menge Speicher verbraucht, aber ihr Ergebnis erst sehr viel später im Graphen benötigt wird, wäre es am besten, sie so spät wie möglich zu evaluieren. Dies vermeidet unnötiges Belegen von RAM, das eventuell von anderen Operationen benötigt wird. Ein anderes Beispiel wäre eine Gruppe von Operationen, die von Daten außerhalb der Recheneinheit abhängen. Wenn diese Operationen alle gleichzeitig ausgeführt würden, belegten sie die gesamte Bandbreite des Geräts, sodass am Ende alle I/O-Kanäle warten müssten. Andere Operationen, die ebenfalls Daten kommunizieren, würden dann gleichermaßen blockiert. Es wäre also besser, solche Operationen mit hohem Kommunikationsaufwand nacheinander auszuführen, damit das Gerät auch andere Operationen parallel laufen lassen kann.

Um die Auswertung einiger Knoten aufzuschieben, können Sie *Control Dependencies* hinzufügen. Das folgende Codebeispiel weist TensorFlow an, x und y erst zu evaluieren, nachdem a und b bereits evaluiert wurden:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Da z von x und y abhängt, wartet z natürlich ebenfalls auf die Evaluation von a und b, obwohl nicht explizit im `control_dependencies()`-Block angegeben. Da außerdem b von a abhängt, könnten wir den obigen Code vereinfachen, indem wir einfach eine Control Dependency von [b] anstelle von [a, b] erstellen. In manchen Fällen gilt aber die Regel »explicit is better than implicit.«

Großartig! Nun wissen Sie, ...

- wie Sie Operationen beliebig auf mehreren Recheneinheiten platzieren,
- wie diese Operationen parallel ausgeführt werden und
- wie Sie Control Dependencies erstellen, um das parallele Ausführen zu optimieren.

Es ist an der Zeit, Berechnungen auf mehrere Server zu verteilen!

Mehrere Recheneinheiten auf mehreren Servern

Um einen Graphen auf mehreren Servern auszuführen, müssen Sie zunächst einen *Cluster* definieren. Ein Cluster besteht aus einem oder mehreren TensorFlow-Servern, den sogenannten *Tasks*. Diese sind normalerweise über mehrere Maschinen verteilt (siehe Abbildung 12-6). Jeder Task gehört zu einem *Job*. Ein Job ist eine Gruppe Tasks mit einem Namen, die meist eine gemeinsame Aufgabe haben, z.B. die Modellparameter zu verwalten (solch ein Job erhält oft die Bezeichnung "ps" für *Parameter Server*) oder Berechnungen durchzuführen (solch ein Job heißt normalerweise "Worker").

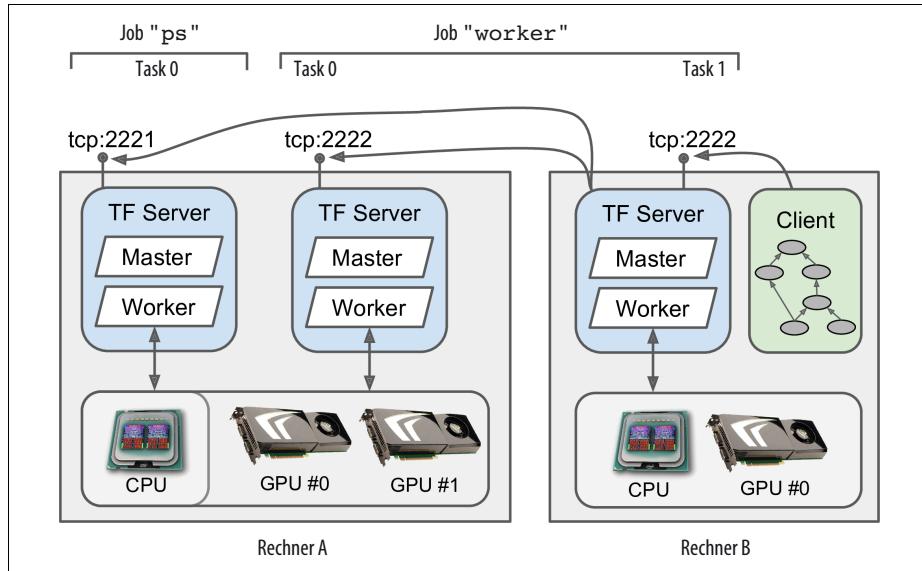


Abbildung 12-6: TensorFlow-Cluster

Die folgende *Cluster-Spezifikation* definiert zwei Jobs, "ps" und "worker" mit jeweils einem und zwei Tasks. In diesem Beispiel laufen auf Rechner A zwei TensorFlow-Servers (also Tasks) auf unterschiedlichen Ports: Einer ist Teil des Jobs "ps", der andere Teil des Jobs "worker". Auf Rechner B läuft nur ein TensorFlow-Server, der Teil des Jobs "worker" ist.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221",  # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222",  # /job:worker/task:1
    ]
})
```

Um einen TensorFlow-Server zu starten, müssen Sie ein Server-Objekt erstellen und diesem die Cluster-Spezifikation (damit dieser mit anderen Servern kommuniziert)

zieren kann), einen Namen für den Job und eine Task-Nummer übergeben. Um beispielsweise den ersten Task für den Worker zu starten, müssten Sie auf Rechner A folgenden Code ausführen:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

Es ist meist einfacher, nur einen Task pro Rechner auszuführen. Das obige Beispiel zeigt aber, dass Sie mit TensorFlow auch mehrere Tasks auf dem gleichen Rechner ausführen können, wenn Sie das möchten.² Wenn Sie auf einem Rechner mehrere Server haben, müssen Sie sicherstellen, dass nicht alle versuchen, wie oben erläutert, das gesamte RAM jeder GPU zu belegen. Beispielsweise sieht der Task "ps" in Abbildung 12-6 die GPU-Recheneinheiten nicht, da der Prozess mit CUDA_VISIBLE_DEVICES="" gestartet wurde. Beachten Sie, dass sich alle Tasks auf dem gleichen Rechner die CPU teilen.

Wenn Sie möchten, dass der Prozess nichts weiter tut, als den TensorFlow-Server auszuführen, können Sie den Main-Thread mit der Methode `join()` auf das Beenden des Servers warten lassen (andernfalls wird der Server abgeschlossen, sobald der Main-Thread verlassen wird). Da im Moment keine Möglichkeit zum Beenden des Servers vorgesehen ist, blockiert dies den Thread für immer:

```
server.join() # blockiert, bis der Server-Thread endet (nie)
```

Öffnen einer Session

Sobald sämtliche Tasks laufen (und noch nichts tun), können Sie von einem Client in einem beliebigen Prozess auf einem beliebigen Rechner eine Session auf einem beliebigen Server starten (dies ist sogar von einem Prozess möglich, auf dem einer der Tasks läuft). Sie können diese Session dann wie eine normale lokale Session nutzen. Beispielsweise:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval()) # 9.0
```

Dieser Code für den Client erstellt zunächst einen einfachen Graphen, öffnet dann eine Session auf dem TensorFlow-Server auf Rechner B (den wir als *Master* bezeichnen) und veranlasst die Evaluation von c. Der Master platziert zunächst Operationen auf den entsprechenden Recheneinheiten. In diesem Beispiel haben wir keine Operationen bestimmten Recheneinheiten zugewiesen. Daher wird alles auf dem Standardgerät platziert – in diesem Fall der GPU von Rechner B. Anschließend wird einfach c ausgewertet, wie vom Client angefordert, und das Ergebnis zurückgegeben.

² Sie können sogar mehrere Tasks im gleichen Prozess starten. Dies kann zum Testen nützlich sein, aber im Produktionsbetrieb ist es nicht empfehlenswert.

Master- und Worker-Dienste

Der Client verwendet zur Kommunikation mit dem Server das *gRPC*-Protokoll (*Google Remote Procedure Call*). Dies ist ein effizientes Open-Source-Framework zum Aufrufen von Funktionen über ein Netzwerk und zum Versenden ihrer Ausgaben über eine Vielzahl Plattformen und Programmiersprachen.³ Es basiert auf HTTP2, das eine Verbindung öffnet und sie über die gesamte Session geöffnet lässt, sodass eine effiziente Kommunikation in beiden Richtungen möglich ist, sobald die Verbindung erst einmal steht. Die Daten werden in Form von *Protokollpuffern* übertragen, einer weiteren Open-Source-Technologie von Google. Dies ist ein leichtgewichtiges Binärformat zum Datenaustausch.



Sämtliche Server in einem TensorFlow-Cluster können mit jedem anderen Server im Cluster kommunizieren. Sie sollten also sicherstellen, dass die entsprechenden Ports in Ihrer Firewall offen sind.

Jeder TensorFlow-Server stellt zwei Dienste zur Verfügung: den *Master-Dienst* und den *Worker-Dienst*. Der Master-Dienst ermöglicht es Clients, Sessions zu öffnen und sie zum Ausführen von Graphen zu verwenden. Er koordiniert die Berechnungen über mehrere Tasks und verlässt sich darauf, dass der Worker-Dienst Berechnungen auf anderen Tasks ausführt und deren Ergebnisse beschafft.

Durch diese Architektur erhalten Sie eine Menge Flexibilität: Ein Client kann sich mit mehreren Servern verbinden, indem er mehrere Sessions in verschiedenen Threads startet. Ein Server kann mehrere Sessions von einem oder mehreren Clients gleichzeitig bearbeiten. Sie können pro Task einen Client ausführen (normalerweise im gleichen Prozess) oder einen Client alle Tasks steuern lassen. Ihnen stehen alle Möglichkeiten offen.

Operationen auf Tasks pinnen

Mit device-Blöcken können Sie Operationen auf eine beliebige Recheneinheit und einen beliebigen Task pinnen, indem Sie den Namen des Jobs, den Index des Tasks, die Art und den Index der Recheneinheit angeben. Der folgende Code pinnt beispielsweise `a` auf die CPU des ersten Tasks im Job "ps" (dies ist die CPU von Rechner A). Er pinnt außerdem `b` auf die zweite GPU im ersten Task des Jobs "worker" (dies ist GPU #1 auf Rechner A). Die Operation `c` wird keiner Recheneinheit zugewiesen, daher platziert der Master sie auf seinem eigenen Standardgerät (GPU #0 auf Rechner B).

```
[source,python]  
  
with tf.device("/job:ps/task:0/cpu:0"):  
  
    
```

³ Es ist die nächste Version von Googles internem Dienst *Stubby*, den Google mehr als 10 Jahre lang erfolgreich eingesetzt hat. Details finden Sie unter <http://grpc.io/>.

```

a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1"):
    b = a + 2

c = a + b

```

Wenn Sie den Typ der Recheneinheit und den Index weglassen, verwendet TensorFlow wie weiter oben die Standard-Recheneinheit dieses Tasks; das Pinnen einer Operation auf "/job:ps/task:0" platziert es auf dem Standardgerät des Jobs "ps" (der CPU von Rechner A). Wenn Sie außerdem den Index des Tasks auslassen (z.B. "/job:ps"), verwendet TensorFlow standardmäßig "/task:0". Wenn Sie den Namen des Jobs und den Index des Tasks auslassen, verwendet TensorFlow standardmäßig den Master-Task der Session.

Sharding von Variablen über mehrere Parameterserver

Wie wir in Kürze sehen werden, ist ein verbreitetes Muster beim Trainieren eines verteilten neuronalen Netzes, die Modellparameter auf einigen Parameterservern zu speichern (also den Tasks im Job "ps"). Andere Tasks konzentrieren sich auf Berechnungen (die Tasks im Job "worker"). Bei großen Modellen mit Millionen Parametern zahlt es sich aus, die Parameter auf mehrere Parameterserver aufzuteilen, damit die Netzwerkverbindung des einzigen Parameterservers nicht zum Flaschenhals wird. Es wäre aber sehr mühselig, jede Variable von Hand einem bestimmten Parameterserver zuzuweisen. Glücklicherweise gibt es in TensorFlow die Funktion `replica_device_setter()`, die Variablen ringsherum auf alle "ps"-Tasks verteilt. Das folgende Codebeispiel pinnt fünf Variablen auf zwei Parameterserver:

```

with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # auf /job:ps/task:0 gepinnt
    v2 = tf.Variable(2.0) # auf /job:ps/task:1 gepinnt
    v3 = tf.Variable(3.0) # auf /job:ps/task:0 gepinnt
    v4 = tf.Variable(4.0) # auf /job:ps/task:1 gepinnt
    v5 = tf.Variable(5.0) # auf /job:ps/task:0 gepinnt

```

Anstatt die Zahl `ps_tasks` anzugeben, können Sie auch die Cluster-Spezifikation mit `cluster=cluster_spec` übergeben. TensorFlow ermittelt dann einfach die Anzahl der Tasks im Job "ps". Wenn Sie in diesem Block andere Operationen außer Variablen erstellen, pinnt TensorFlow diese automatisch auf "/job:worker", der sie standardmäßig der ersten Recheneinheit im ersten Task des Jobs "worker" zuweist. Sie können die Operationen über den Parameter `worker_device` auch einer anderen Recheneinheit zuweisen. Allerdings sind hierzu verschachtelte device-Blöcke die bessere Alternative. Ein verschachtelter device-Block kann im äußeren Codeblock definierte Jobs, Tasks oder Recheneinheiten außer Kraft setzen. Hier ein Beispiel:

```

with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # gepinnt auf /job:ps/task:0 (+ defaults to /cpu:0)
    v2 = tf.Variable(2.0) # gepinnt auf /job:ps/task:1 (+ defaults to /cpu:0)
    v3 = tf.Variable(3.0) # gepinnt auf /job:ps/task:0 (+ defaults to /cpu:0)

```

```
[...]
s = v1 + v2          # gepinnt auf /job:worker (+ defaults to task:0/gpu:0)
with tf.device("/gpu:1"):
    p1 = 2 * s      # gepinnt auf /job:worker/gpu:1 (+ defaults to /task:0)
    with tf.device("/task:1"):
        p2 = 3 * s    # gepinnt auf /job:worker/task:1/gpu:1
```



Dieses Beispiel geht davon aus, dass die Parameterserver nur die CPU verwenden. Dies ist normalerweise der Fall, da sie lediglich Parameter speichern und versenden müssen, aber keine aufwendigen Berechnungen durchführen.

Zustände mit Resource Containers zwischen Sessions teilen

Wenn Sie eine einfache *lokale Session* verwenden (nicht die verteilte Variante), verwaltet die Session den Zustand jeder Variable selbst; sobald die Session endet, gehen die Werte sämtlicher Variablen verloren. Außerdem können sich mehrere lokale Sessions keinen Zustand teilen, nicht einmal, wenn beide den gleichen Graphen ausführen; jede Session besitzt ihre eigene Kopie jeder einzelnen Variablen (wie in Kapitel 9 besprochen). Im Gegensatz dazu wird der Zustand von Variablen bei *verteilten Sessions* von *Resource Containers* auf dem Cluster verwaltet, und nicht von den Sessions. Wenn Sie also eine Variable *x* mit einer Client-Session erstellen, ist diese automatisch für jede andere Session auf dem gleichen Cluster verfügbar (selbst wenn beide Sessions mit einem anderen Server verbunden sind). Betrachten Sie als Beispiel den folgenden Client-Code:

```
# simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2]==["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Nehmen wir an, Sie haben einen laufenden TensorFlow-Cluster auf den Rechnern A und B auf Port 2222. Sie können mit folgendem Befehl den Client starten, eine Session mit dem Server auf Rechner A öffnen, die Variable initialisieren lassen, erhöhen und ihren Wert ausgeben:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222 init
1.0
```

Wenn Sie den Client stattdessen mit folgendem Befehl starten, verbindet er sich mit dem Server auf Rechner B und verwendet die gleiche Variable *x* (diesmal bitten wir den Server nicht, die Variable zu initialisieren):

```
$ python3 simple_client.py grpc://machine-b.example.com:2222
2.0
```

Dies hat Vor- und Nachteile: Es ist prima, wenn Sie Variablen über mehrere Sessions teilen möchten, aber wenn Sie vollständig unabhängige Berechnungen auf dem gleichen Cluster durchführen möchten, müssen Sie darauf achten, nicht aus Versehen die gleichen Variablennamen zu verwenden. Diese Art von Namenskollisionen können Sie vermeiden, indem Sie Ihre gesamte Konstruktionsphase in einen Scope mit einem eindeutigen Namen für alle Berechnungen verpacken:

```
with tf.variable_scope("my_problem_1"):
    [...] # Konstruktionsphase für Aufgabe 1
```

Eine bessere Variante ist das Verwenden eines Container-Blocks:

```
with tf.container("my_problem_1"):
    [...] # Konstruktionsphase für Aufgabe 1
```

Damit wird für Aufgabe #1 anstelle des voreingestellten (mit dem leeren String "") als Namen) ein eigener Container verwendet. Dies hat den Vorteil, dass die Namen von Variablen kurz gehalten werden. Außerdem können Sie einen Container mit Namen leicht zurücksetzen. Die folgende Anweisung beispielsweise verbindet sich mit dem Server auf Rechner A und setzt den Container namens "my_problem_1" zurück, wodurch sämtliche von diesem Container belegten Ressourcen wieder freigegeben werden (alle auf dem Container offenen Sessions werden geschlossen). Bevor Sie eine von diesem Container verwaltete Variable abermals verwenden können, müssen Sie diese initialisieren:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Mit Resource Containers können Sie Variablen zwischen Sessions flexibel teilen. Beispielsweise zeigt Abbildung 12-7 vier Clients mit unterschiedlichen Graphen auf dem gleichen Cluster, die aber einige Variablen miteinander teilen. Die Clients A und B teilen sich Variable x, die vom Standardcontainer verwaltet wird, Clients C und D teilen sich eine andere Variable namens x im Container namens "my_problem_1". Es ist zu betonen, dass Client C beide Variablen aus beiden Containern verwendet.

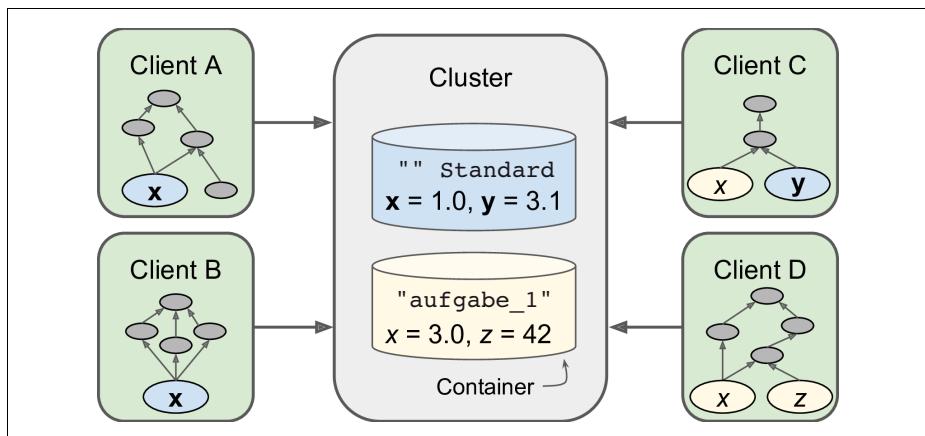


Abbildung 12-7: Resource Container

Resource Container kümmern sich auch um den Zustand von anderen zustandsbasierten Operationen wie Queues und Readern. Schauen wir uns zunächst Queues an.

Asynchrone Kommunikation mit TensorFlow-Queues

Queues sind ebenfalls ein großartiges Hilfsmittel zum Datenaustausch zwischen mehreren Sessions; ein verbreiteter Anwendungsfall ist beispielsweise, dass ein Client einen Graphen zum Laden der Trainingsdaten erstellt und in die Queue eingibt, ein anderer Client dagegen erstellt einen Graphen, der die Daten aus der Queue holt und ein Modell trainiert (siehe Abbildung 12-8). Dies kann das Trainieren erheblich beschleunigen, da die Operationen zum Trainieren nicht bei jedem Schritt das nächste Mini-Batch abwarten müssen.

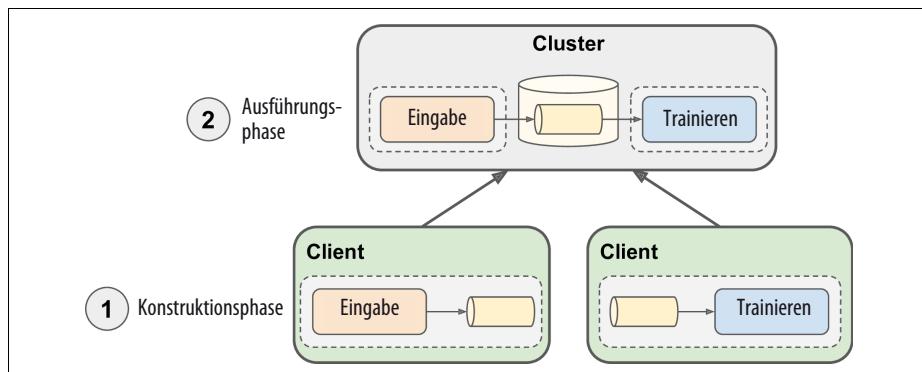


Abbildung 12-8: Trainingsdaten mit Queues asynchron laden

In TensorFlow gibt es unterschiedliche Arten von Queues. Die einfachste ist die *First-In-First-Out-Queue (FIFO)*. Das folgende Codebeispiel erstellt eine FIFO-Queue, die bis zu zehn Tensoren mit jeweils zwei Floats speichern kann:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]], name="q", shared_name="shared_q")
```



Um Variablen zwischen Sessions zu teilen, müssen Sie lediglich auf beiden Seiten den gleichen Namen und Container angeben. Bei Queues verwendet TensorFlow anstatt des Attributs `name` das Attribut `shared_name`, das deshalb angegeben werden muss (selbst wenn es zu `name` identisch ist). Natürlich müssen Sie auch den gleichen Container verwenden.

Daten in eine Queue stellen

Um Daten in eine Queue einzustellen, müssen Sie eine `enqueue`-Operation erstellen. Das folgende Codebeispiel speist drei Trainingsdatenpunkte in die Queue ein:

```

# training_data_loader.py
import tensorflow as tf

q = [...]
training_instance = tf.placeholder(tf.float32, shape=(2))
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
    sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
    sess.run(enqueue, feed_dict={training_instance: [5., 6.]})

```

Anstatt die Datenpunkte einen nach dem anderen in die Queue zu stellen, können Sie mit der Operation enqueue_many auch mehrere auf einmal abfertigen:

```

[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue_many([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
              feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})

```

Beide Codebeispiele stellen die gleichen drei Tensoren in die Queue.

Daten aus einer Queue holen

Um die Datenpunkte wieder aus der Queue herauszuziehen, müssen Sie die Operation dequeue erzeugen:

```

# trainer.py
import tensorflow as tf

q = [...]
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue)) # [1., 2.]
    print(sess.run(dequeue)) # [3., 4.]
    print(sess.run(dequeue)) # [5., 6.]

```

Im Normalfall sollten Sie ein ganzes Mini-Batch auf einmal herausziehen, anstatt die Datenpunkte einzeln abzufragen. Dazu verwenden Sie die Operation dequeue_many und geben die Größe des Mini-Batches an:

```

[...]
batch_size = 2
dequeue_mini_batch= q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch)) # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch)) # blockiert und wartet auf weitere Daten

```

Ist eine Queue voll, blockiert die Operation enqueue, bis Einträge über die Operation dequeue herausgezogen werden. Umgekehrt blockiert die Operation dequeue, wenn eine Queue leer ist (oder Sie dequeue_many() verwenden und weniger Ele-

mente vorhanden sind, als für ein Mini-Batch angefordert wurden), bis genug Daten über enqueue in die Queue eingespeist wurden.

Queues aus Tupeln

Jedes Element einer Queue kann ein Tupel aus Tensoren (mit unterschiedlichen Typen und Abmessungen) sein, nicht nur ein einzelner Tensor. Beispielsweise speichert die folgende Queue Paare von Tensoren, einen vom Typ int32 mit den Abmessungen () und einen vom Typ float32 mit den Abmessungen [3,2]:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[[],[3,2]],  
name="q", shared_name="shared_q")
```

Die Operation enqueue muss Paare von Tensoren erhalten (jedes Paar entspricht nur einem Element der Queue):

```
a = tf.placeholder(tf.int32, shape=())  
b = tf.placeholder(tf.float32, shape=(3, 2))  
enqueue = q.enqueue((a, b))
```

```
with tf.Session([...]) as sess:  
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.]]})  
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.]]})  
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.]]})
```

Am anderen Ende erstellt die Funktion dequeue() nun paarweise dequeue-Operationen:

```
dequeue_a, dequeue_b = q.dequeue()
```

Sie sollten diese Operationen gemeinsam ausführen:

```
with tf.Session([...]) as sess:  
    a_val, b_val = sess.run([dequeue_a, dequeue_b])  
    print(a_val) # 10  
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```



Wenn Sie dequeue_a alleine ausführen, wird es ein Paar aus der Queue holen und nur das erste Element zurückgeben; das zweite Element geht verloren (ebenso geht das erste Element verloren, wenn Sie nur dequeue_b ausführen).

Die Funktion dequeue_many() liefert ebenfalls zwei Operationen:

```
batch_size = 2  
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

Sie können diese wie erwartet verwenden:

```
with tf.Session([...]) as sess:  
    a, b = sess.run([dequeue_a, dequeue_b])  
    print(a) # [10, 11]  
    print(b) # [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]  
    a, b = sess.run([dequeue_a, dequeue_  
b]) # blockiert und wartet auf ein Datenpaar
```

Schließen einer Queue

Sie können eine Queue schließen, um andere Sessions darauf hinzuweisen, dass keine weiteren Daten folgen werden:

```
close_q = q.close()

with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Jedes folgende Ausführen der Operationen enqueue oder enqueue_many erzeugt einen Ausnahmefehler. Standardmäßig werden alle noch ausstehenden Anfragen für enqueue auch umgesetzt, es sei denn, Sie rufen q.close(cancel_pending_enqueues=True) auf.

Das Ausführen der Operationen dequeue oder dequeue_many funktioniert, solange noch Elemente in der Queue sind. Sobald es nicht mehr genug Elemente in der Queue gibt, schlagen auch diese fehl. Wenn Sie die Operation dequeue_many verwenden und es weniger Daten gibt, als für einen Mini-Batch nötig sind, gehen diese verloren. Sie können stattdessen auch die Operation dequeue_up_to verwenden; sie verhält sich genauso wie dequeue_many, außer wenn eine Queue geschlossen ist und weniger als batch_size Elemente in der Queue übrig sind. In diesem Falle werden einfach nur die vorhandenen zurückgegeben.

RandomShuffleQueue

TensorFlow enthält einige weitere Arten von Queues, darunter die RandomShuffleQueue. Diese lässt sich genauso wie eine FIFOQueue verwenden, außer dass die Elemente in zufälliger Reihenfolge ausgegeben werden. Dies ist nützlich, um Trainingsdatenpunkte beim Trainieren in jeder Epoche zu durchmischen. Erstellen wir zunächst die Queue:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                           dtypes=[tf.float32], shapes=[()],
                           name="q", shared_name="shared_q")
```

Der Wert min_after_dequeue gibt an, wie viele Einträge nach einer dequeue-Operation mindestens in der Queue verbleiben müssen. Dies stellt sicher, dass ausreichend Datenpunkte in der Queue sind, um genug Zufälligkeit zu garantieren (nach Schließen der Queue wird min_after_dequeue ignoriert). Nehmen wir an, Sie haben 22 Elemente in diese Queue eingespeist (die Floats 1. bis 22.), dann könnten Sie diese folgendermaßen wieder aus der Queue herausziehen:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
    print(sess.run(dequeue)) # [ 20. 15. 11. 12. 4.] (17 Elemente übrig)
    print(sess.run(dequeue)) # [ 5. 13. 6. 0. 17.] (12 Elemente übrig)
    print(sess.run(dequeue)) # 12 - 5 < 10:
    # blockiert, wartet auf 3 weitere Elemente
```

PaddingFIFOQueue

Eine PaddingFIFOQueue lässt sich ebenfalls wie eine FIFOQueue verwenden, nur dass sie Tensoren mit variabler Größe in jeder Dimension annimmt (aber mit festgelegter Anzahl Dimensionen). Wenn Sie die Operationen `dequeue_many` oder `dequeue_up_to` anwenden, wird jeder Tensor entlang jeder variablen Dimension mit Nullen aufgefüllt, bis er so groß wie der größte Tensor im Mini-Batch ist. Sie könnten beispielsweise 2-D-Tensoren (Matrizen) beliebiger Größe in die Queue eingeben:

```
q = tf.PaddingFIFOQueue(capacity=50, dtypes=[tf.float32], shapes=[(None, None)]
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})      # 3x2
    sess.run(enqueue, feed_dict={v: [[1.]}])                                # 1x1
    sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

Wenn wir die Elemente einzeln anfordern, erhalten wir wieder die in die Queue eingegebenen Tensoren in identischer Form. Wenn wir aber mehrere Elemente anfordern (mit `dequeue_many()` oder `dequeue_up_to()`), vergrößert die Queue die Tensoren automatisch. Wenn wir beispielsweise alle drei Elemente auf einmal anfordern, werden alle Tensoren so mit Nullen ausgepolstert, dass sie Tensoren der Größe 3×4 werden, da das Maximum der ersten Dimension 3 (erstes Element) und das Maximum der zweiten Dimension 4 ist (drittes Element):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
[[[ 1.  2.  0.  0.]
 [ 3.  4.  0.  0.]
 [ 5.  6.  0.  0.]]
 [[ 1.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
 [[ 7.  8.  9.  5.]
 [ 6.  7.  8.  9.]
 [ 0.  0.  0.  0.]]]
```

Diese Art Queue ist bei Eingaben variabler Länge nützlich, wie etwa bei Wortfolgen (siehe Kapitel 14).

Halten wir für einen Moment inne: Bisher haben Sie gelernt, Berechnungen auf mehrere Geräte und Server zu verteilen, Variablen zwischen Sessions zu teilen und über Queues asynchron zu kommunizieren. Bevor wir beginnen, neuronale Netze zu trainieren, müssen wir ein letztes Thema besprechen: Wie lassen sich Trainingsdaten effizient laden?

Daten direkt aus dem Graphen laden

Bisher sind wir davon ausgegangen, dass die Clients ihre Trainingsdaten laden und über Platzhalter an den Cluster übergeben. Dieses Vorgehen ist einfach und funktioniert in einfachen Szenarien recht gut, aber es ist nicht besonders effizient, da die Trainingsdaten mehrmals übertragen werden:

1. vom Dateisystem zum Client
2. vom Client zum Master-Task
3. eventuell vom Master-Task zu anderen Tasks, die die Daten benötigen

Bei mehreren Clients, die unterschiedliche neuronale Netze mit den gleichen Trainingsdaten trainieren (z.B. beim Optimieren von Hyperparametern) wird es schlimmer: Wenn jeder Client die Daten simultan lädt, verliert Ihr Fileserver oder die Bandbreite Ihres Netzwerks leicht an Performance.

Preloading von Daten in Variablen

Bei Datensätzen, die in den Speicher passen, ist es besser, die Trainingsdaten einmal zu laden, einer Variablen zuzuweisen und anschließend diese Variable im Graphen zu verwenden. Dies nennt man *Preloading* des Trainingsdatensatzes. Die Daten werden so nur einmal vom Client auf den Cluster übertragen (müssen aber eventuell trotzdem von Task zu Task kopiert werden, falls einzelne Operationen dies erfordern). Der folgende Code lädt den gesamten Trainingsdatensatz in eine Variable:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                         name="training_set")

with tf.Session([...]) as sess:
    data = [...] # lade die Trainingsdaten aus dem Datenspeicher
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

Sie müssen `trainable=False` setzen, damit die Optimierer nicht versuchen, diese Variable zu verändern. Sie sollten außerdem `collections=[]` setzen, damit diese Variable nicht der Kollektion `GraphKeys.GLOBAL_VARIABLES` hinzugefügt wird, die zum Speichern und Wiederherstellen von Zwischenständen verwendet wird.



In diesem Beispiel gehen wir davon aus, dass Ihr gesamter Trainingsdatensatz (einschließlich der Labels) ausschließlich aus Werten vom Typ `float32` besteht. Ist dies nicht der Fall, benötigen Sie eine Variable pro Typ.

Direktes Einlesen der Trainingsdaten im Graphen

Falls der Trainingsdatensatz nicht in den Speicher passt, können Sie *Leseoperationen* verwenden: Diese Operationen lesen Daten direkt aus dem Dateisystem ein.

Dadurch müssen die Trainingsdaten überhaupt nicht durch die Clients geschleust werden. TensorFlow enthält Leseoperationen für diverse Dateiformate:

- CSV
- binäre Records mit fester Länge
- das in TensorFlow eingebaute Format `TFRecords`, das auf Protocol Buffers basiert

Schauen wir uns ein einfaches Beispiel an, das eine CSV-Datei einliest (für andere Formate lesen Sie bitte in der API-Dokumentation nach). Nehmen wir an, Sie möchten Operationen zum Lesen der Trainingsdatenpunkte aus der Datei `my_test.csv` erstellen. Die Datei enthalte die Merkmale `x1` und `x2` als Floats und die binäre Kategorie `target` als Integerzahl:

```
x1, x2, target
1., 2., 0
4., 5., 1
7., , 0
```

Zunächst erstellen wir einen `TextLineReader`, um diese Datei einzulesen. Ein `TextLineReader` öffnet eine Datei (sobald wir ihm sagen, welche geöffnet werden soll) und liest den Inhalt zeilenweise ein. Dies ist wie bei Variablen und Queues eine Zustandsbasierte Operation: Ihr Zustand bleibt über mehrere Durchläufe des Graphen erhalten und merkt sich, welche Datei aktuell gelesen wird und welches die aktuelle Position innerhalb dieser Datei ist.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Anschließend legen wir eine Queue an, aus der die Leseoperation den Namen der zu lesenden Datei herauszieht. Wir erzeugen des Weiteren eine `enqueue`-Operation und einen Platzhalter, mit dem wir beliebige Dateinamen in die Queue eingeben können. Schließlich erstellen wir eine Operation zum Schließen der Queue, sobald es keine Dateien mehr zu lesen gibt:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Damit sind wir bereit, die Operation `read` zu erstellen, in der wir einen einzelnen Eintrag (d. h. eine Zeile) lesen und ein Schlüssel-Wert-Paar zurückgeben. Der Schlüssel ist der eindeutige Bezeichner des Eintrags – ein String aus dem Dateinamen, einem Doppelpunkt (`:`) und der Zeilenummer – und der Wert einfach ein String mit dem Inhalt der Zeile:

```
key, value = reader.read(filename_queue)
```

Wir haben damit alles, was wir zum zeilenweisen Lesen der Datei benötigen, sind aber noch nicht ganz fertig, denn wir müssen diesen String noch parsen, um die Merkmale und Zielwerte zu erhalten:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
features = tf.stack([x1, x2])
```

Die erste Zeile verwendete den CSV-Parser aus TensorFlow, um die Werte aus der aktuellen Zeile zu ermitteln. Wenn ein Eintrag fehlt, wird ein Standardwert verwendet (in diesem Fall fehlt der Wert für `x2` im dritten Trainingsdatenpunkt). Die Standardwerte dienen auch zum Festlegen des Typs jedes Felds (in diesem Falle zwei Floats und ein Integer). Schließlich übergeben wir Trainingsdatenpunkt und Zielwert einer `RandomShuffleQueue`, die wir mit dem Trainingsgraphen teilen (sodass wir Mini-Batches daraus ziehen können). Wir erstellen außerdem eine Operation zum Schließen der Queue, sobald wir mit dem Einspeisen der Instanzen fertig sind:

```
instance_queue = tf.RandomShuffleQueue(
    capacity=10, min_after_dequeue=2,
    dtypes=[tf.float32, tf.int32], shapes=[[2], []],
    name="instance_q", shared_name="shared_instance_q")
enqueue_instance = instance_queue.enqueue([features, target])
close_instance_queue = instance_queue.close()
```

Das war eine Menge Arbeit, nur um eine Datei einzulesen. Wir haben den Graphen zum jetzigen Zeitpunkt lediglich erstellt, müssen ihn also noch ausführen:

```
with tf.Session([...]) as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    try:
        while True:
            sess.run(enqueue_instance)
    except tf.errors.OutOfRangeError as ex:
        pass # keine weiteren Einträge in der Datei und keine weiteren Dateien zu
lesen
    sess.run(close_instance_queue)
```

Zunächst öffnen wir die Session, schicken den Dateinamen "my_test.csv" in die Queue und schließen diese unmittelbar danach, da keine weiteren Dateinamen folgen werden. Anschließend schicken wir Datenpunkte in einer Endlosschleife einzeln in die Queue. Dabei ist `enqueue_instance` vom `reader`-Objekt abhängig, das die Zeilen einliest. Bei jedem Durchlauf wird ein neuer Eintrag eingelesen, bis das Ende der Datei erreicht ist. An diesem Punkt wird versucht, den nächsten Dateinamen einzulesen, und da diese Queue bereits geschlossen ist, wird der Ausnahmefehler `OutOfRangeError` erzeugt (wenn wir die Queue nicht schließen würden, würde die Operation einfach nur blockieren, bis wir einen weiteren Dateinamen einspeisen oder die Queue schließen).

Am Ende schließen wir die Queue mit den Datenpunkten, sodass die Operationen zur Entnahme der Trainingsdaten nicht blockiert. In Abbildung 12-9 sind unsere bisherigen Erkenntnisse zusammengefasst; dort ist ein typischer Graph zum Lesen von Trainingsdaten aus CSV-Dateien dargestellt.

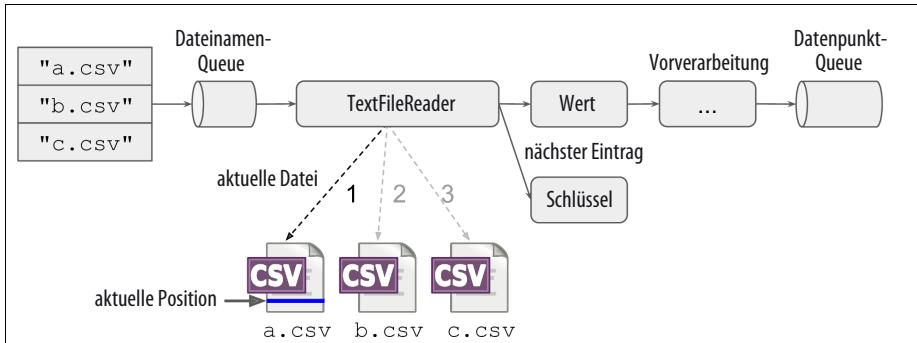


Abbildung 12-9: Ein Graph zum Lesen von Trainingsdatenpunkten aus CSV-Dateien

Im Trainingsgraphen müssen wir die geteilte Queue für die Datenpunkte erstellen und dieser Mini-Batches entnehmen:

```
instance_queue = tf.RandomShuffleQueue([...], shared_name="shared_instance_q")
mini_batch_instances, mini_batch_targets = instance_queue.dequeue_up_to(2)
[...] # verwende die Datenpunkte und Zielwerte aus mini_
batch und erstelle den Graphen zum Trainieren
training_op = [...]

with tf.Session([...]) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
        pass # keine weiteren Trainingsdaten
```

Das erste Mini-Batch in diesem Beispiel enthält die ersten zwei Einträge der CSV-Datei, und das zweite Mini-Batch den letzten Eintrag.



Queues in TensorFlow können nicht gut mit dünn besetzten Tensoren umgehen. Falls Ihre Trainingsdaten also dünn besetzt sind, sollten Sie die Einträge nach dem Entnehmen aus der Queue mit den Datenpunkten parsen.

Diese Architektur verwendet nur einen Thread, um Einträge zu lesen und diese in die Queue einzugeben. Sie können die Durchsatzrate stark erhöhen, indem mehrere Threads gleichzeitig aus mehreren Dateien lesen. Schauen wir uns das einmal an.

Lesen in mehreren Threads mit Coordinator und QueueRunner

Damit mehrere Threads gleichzeitig Datenpunkte einlesen, könnten Sie Python-Threads erstellen (mit dem Modul `threading`) und diese selbst verwalten. Allerdings enthält TensorFlow einige Hilfsmittel für eine solche Aufgabe: die Klassen `Coordinator` und `QueueRunner`.

Ein Coordinator ist ein sehr einfaches Objekt, dessen einziger Zweck das koordinierte Anhalten mehrerer Threads ist. Dazu erstellen Sie zunächst einen Coordinator:

```
coord = tf.train.Coordinator()
```

Als Nächstes übergeben Sie ihm alle Threads, die gemeinsam angehalten werden sollen. Deren Hauptschleife sieht folgendermaßen aus:

```
while not coord.should_stop():
    [...] # tue etwas
```

Jeder Thread kann das Anhalten aller Threads durch Aufrufen der Methode `request_stop()` des Objekts Coordinator einfordern:

```
coord.request_stop()
```

Jeder Thread hält an, sobald er seine aktuelle Iteration abgeschlossen hat. Durch Aufrufen der Methode `join()` des Objekts Coordinator können Sie auf das Beenden aller Threads warten, indem Sie ihr eine Liste von Threads übergeben:

```
coord.join(list_of_threads)
```

Ein QueueRunner führt mehrere Threads aus, von denen jeder immer wieder die Operation `enqueue` ausführt und so die Queue so schnell wie möglich auffüllt. Sobald die Queue geschlossen wird, wirft der nächste Thread beim Versuch, ein Element in die Queue zu geben, einen `OutOfRangeError` aus; dieser Thread fängt den Ausnahmefehler ab und veranlasst über den Coordinator die übrigen Threads, anzuhalten. Der folgende Code zeigt, wie fünf Threads über einen QueueRunner Daten lesen und in eine Queue einspeisen:

```
[...] # gleiche Konstruktionsphase wie zuvor
queue_runner = tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)

with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads = queue_runner.create_threads(sess, coord=coord, start=True)
```

Die erste Zeile erstellt den QueueRunner und weist ihn an, fünf Threads auszuführen, die alle die gleiche Operation `enqueue_instance` wiederholen. Anschließend starten wir eine Session und geben die Namen der zu lesenden Dateien in die Queue (hier nur "my_test.csv"). Anschließend erstellen wir einen Coordinator, den der QueueRunner zum kontrollierten Anhalten verwendet. Schließlich weisen wir den QueueRunner an, die Threads zu erstellen und zu starten. Die Threads lesen alle Trainingsdatenpunkte ein, geben sie in die entsprechende Queue und halten anschließend kontrolliert an.

Dies ist schon ein wenig effizienter als zuvor, es geht aber noch besser. Im Moment lesen alle Threads aus der gleichen Datei. Wir können sie gleichzeitig unterschiedliche Dateien auslesen lassen (wenn die Trainingsdaten auf mehrere CSV-Dateien

aufgeteilt sind), indem wir mehrere Reader-Objekte erstellen (siehe Abbildung 12-10).

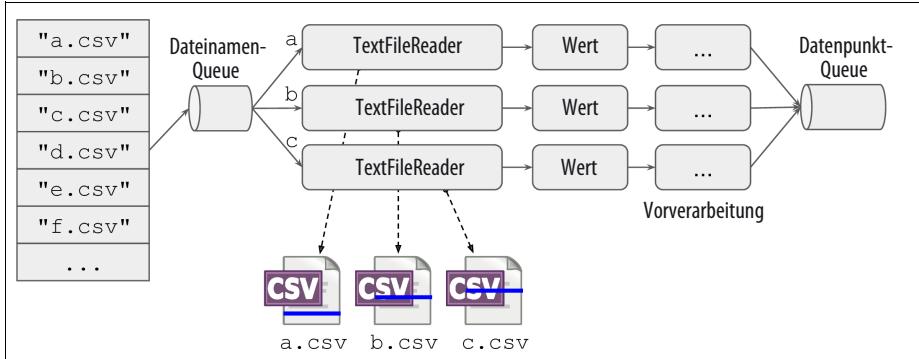


Abbildung 12-10: Gleichzeitiges Lesen aus mehreren Dateien

Dazu müssen wir eine kleine Funktion schreiben, um ein Reader-Objekt und die Knoten zum Auslesen eines Eintrags und Einspeisen in eine Queue zu erstellen:

```

def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
    features = tf.stack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance

```

Anschließend definieren wir die Queues:

```

filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()

instance_queue = tf.RandomShuffleQueue([...])

```

Zum Schluss erstellen wir den QueueRunner, dem wir diesmal aber eine Liste von enqueue-Operationen übergeben. Jede Operation verwendet einen anderen Reader, sodass die Threads gleichzeitig aus unterschiedlichen Dateien lesen:

```

read_and_enqueue_ops = [
    read_and_push_instance(filename_queue, instance_queue)
    for i in range(5)]
queue_runner = tf.train.QueueRunner(instance_queue, read_and_enqueue_ops)

```

Die Ausführungsphase ist die gleiche wie zuvor: Zuerst geben Sie die Namen der zu lesenden Dateien in die Queue, erstellen dann einen Coordinator und starten die Threads im QueueRunner. Diesmal liest jeder Thread eine andere Datei, bis alle Dateien vollständig ausgelesen wurden. Darauf schließt der QueueRunner die Queue mit den Datenpunkten, sodass die sie nutzenden Operationen nicht blockiert werden.

Weitere nützliche Funktionen

TensorFlow enthält einige nützliche Funktionen, die häufige Aufgaben beim Lesen von Trainingsdaten vereinfachen. Wir werden hier einige davon kurz besprechen (die komplette Liste finden Sie in der Dokumentation der API).

Die Funktion `string_input_producer()` nimmt einen 1-D-Tensor mit einer Liste von Dateinamen, erstellt einen Thread, der Dateinamen einzeln in diese Queue eingibt, und schließt die Queue anschließend. Wenn Sie eine Anzahl Epochen angeben, werden die Dateinamen einmal pro Epoche durchlaufen, bevor die Queue geschlossen wird. Die Dateinamen werden in jeder Epoche automatisch durchmischt. Die Funktion erstellt einen QueueRunner, um den eigenen Thread zu verwalten, und fügt diesen der Kollektion `GraphKeys.QUEUE_RUNNERS` hinzu. Um jeden QueueRunner in dieser Collection zu starten, rufen Sie die Funktion `tf.train.start_queue_runners()` auf. Achten Sie darauf, den QueueRunner zu starten. Wenn Sie diesen Schritt vergessen, ist die Queue mit den Dateinamen offen und leer, und Ihre Reader sind dauerhaft blockiert.

Es gibt einige weitere *Producer*-Funktionen, die auf ähnliche Weise eine Queue und einen QueueRunner zum Ausführen einer enqueue-Operation erstellen (z.B. `input_producer()`, `range_input_producer()` und `slice_input_producer()`).

Die Funktion `shuffle_batch()` nimmt eine Liste von Tensoren entgegen (z.B. `[merkmale, zielgroesse]`) und erstellt:

- eine RandomShuffleQueue
- einen QueueRunner, um die Tensoren in die Queue einzugeben (wird zur Kollektion `GraphKeys.QUEUE_RUNNERS` hinzugefügt)
- eine Operation `dequeue_many`, um der Queue einen Mini-Batch zu entnehmen

Dies erleichtert das Verwalten einer Eingabepipeline mit mehreren Threads, einer angeschlossenen Queue für die Mini-Batches und einer Trainingsspipeline in einem Prozess. Schlagen Sie auch die Funktionen `batch()`, `batch_join()` und `shuffle_batch_join()` nach, die ähnliche Vorgänge ermöglichen.

In Ordnung! Sie haben nun alle Werkzeuge beisammen, die Sie zum Trainieren und Ausführen neuronaler Netzwerke über mehrere Recheneinheiten und Server auf einem TensorFlow-Cluster benötigen. Fassen wir das Erlernte zusammen:

- Verwenden mehrerer GPU-Recheneinheiten
- Aufsetzen und Starten eines TensorFlow-Clusters
- Verteilen von Berechnungen über mehrere Geräte und Server
- Teilen von Variablen (und anderen zustandsbasierten Operationen wie Queues und Readern) zwischen Sessions mithilfe von Containern
- Koordinieren mehrerer asynchroner Graphen mithilfe von Queues
- Effizientes Einlesen von Eingabedaten mit Readern, Queues und Coordinators

Wenden wir all das an, um neuronale Netze zu parallelisieren!

Parallelisieren neuronaler Netze auf einem TensorFlow-Cluster

In diesem Abschnitt behandeln wir zuerst, wie sich mehrere neuronale Netze parallelisieren lassen, indem jedes auf einem eigenen Gerät platziert wird. Anschließend betrachten wir die viel schwierigere Aufgabe, ein einzelnes neuronales Netz auf mehrere Recheneinheiten und Server zu verteilen.

Ein neuronales Netz pro Recheneinheit

Der trivialste Ansatz, neuronale Netze auf einem TensorFlow-Cluster zu trainieren und auszuführen, ist, den exakt gleichen Code für ein einzelnes Gerät zu verwenden, die Adresse des Master-Servers anzugeben und eine Session zu erstellen. Das ist alles – Sie sind fertig! Ihr Code wird auf der Standard-Recheneinheit des Servers laufen. Sie können die Recheneinheit für Ihren Graphen ändern, indem Sie die Konstruktionsphase in einen device-Block einschließen.

Indem Sie mehrere Client-Sessions parallel laufen lassen (in unterschiedlichen Threads oder Prozessen) und diese sich mit unterschiedlichen Servern verbinden oder Sie die Verwendung unterschiedlicher Recheneinheiten konfigurieren, lassen sich leicht viele neuronale Netze parallel auf allen Recheneinheiten und Rechnern in Ihrem Cluster trainieren oder ausführen (siehe Abbildung 12-11). Der Zugewinn an Geschwindigkeit ist beinahe linear.⁴ Das Trainieren von 100 neuronalen Netzen auf 50 Servern mit je 2 GPUs wird so nicht viel länger dauern als das Trainieren von nur 1 neuronalen Netz mit 1 GPU.

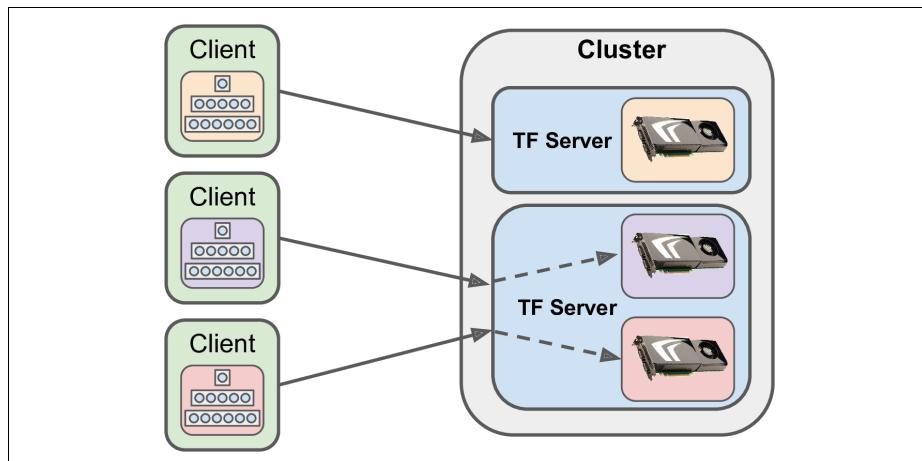


Abbildung 12-11: Trainieren eines neuronalen Netzes pro Gerät

⁴ Wenn Sie warten, bis alle Geräte fertig sind, ist dieser Zugewinn nicht zu 100 % linear, da das langsamste Gerät die Gesamtzeit bestimmt.

Dieser Ansatz ist perfekt geeignet für das Optimieren von Hyperparametern: Jedes Gerät im Cluster trainiert ein anderes Modell mit einem eigenen Satz Hyperparameter. Je mehr Rechenkapazität Sie haben, umso größer ist der abgedeckte Hyperparameterraum.

Er funktioniert ebenfalls ausgezeichnet, wenn Sie einen Webdienst anbieten, der eine große Anzahl *Queries pro Sekunde* (QPS) erhält und Ihr neuronales Netz für jede Query eine Vorhersage treffen muss. Sie können das neuronale Netz einfach über sämtliche Geräte im Cluster replizieren und die Queries auf diese verteilen. Durch das Hinzufügen weiterer Server können Sie eine quasi unbegrenzte Anzahl QPS bearbeiten (allerdings wird die Bearbeitungszeit einer Anfrage dadurch nicht verkürzt, da man nach wie vor auf die Vorhersage eines neuronalen Netzes warten muss).



Alternativ dazu können Sie Ihre neuronalen Netze auch mit *TensorFlow Serving* zur Verfügung stellen. Dies ist ein von Google im Februar 2016 veröffentlichtes Open-Source-System, das eine große Anzahl Anfragen an Machine-Learning-Modelle bearbeiten kann (typischerweise auf der Basis von TensorFlow). Es nimmt eine Versionierung der Modelle vor, Sie können daher leicht eine neue Version Ihres Netzes in die Produktionsumgebung bringen oder mit Algorithmen experimentieren, ohne Ihren Dienst zu unterbrechen. Durch das Hinzufügen von Servern kann es hohe Nutzerzahlen aushalten. Details dazu finden Sie unter <https://tensorflow.github.io/serving/>.

In-Graph- und Between-Graph-Replikation

Sie können auch das Trainieren eines großen Ensembles neuronaler Netze parallelisieren, indem Sie jedes neuronale Netz auf einer eigenen Recheneinheit platzieren (Ensembles wurden in Kapitel 7 vorgestellt). Wenn Sie aber das Ensemble *ausführen* möchten, müssen Sie die einzelnen Vorhersagen aller neuronalen Netze aggregieren, um die Vorhersage des Ensembles zu berechnen. Dies erfordert ein wenig Koordination.

Es gibt zwei wichtige Ansätze für Ensembles neuronaler Netze (oder anderer Graphen, die große Stücke unabhängiger Berechnungen enthalten):

- Sie erstellen einen großen Graphen, der jedes neuronale Netz enthält, jeweils auf ein anderes Gerät gepinnt, sowie die zur Aggregation der einzelnen Vorhersagen nötigen Berechnungen (siehe Abbildung 12-12). Dann erstellen Sie eine Session auf einem beliebigen Server im Cluster und lassen diesen sich um alles kümmern (einschließlich auf die einzelnen Vorhersagen zu warten, so dass diese aggregiert werden können). Diesen Ansatz bezeichnet man als *In-Graph-Replikation*.
- Alternativ können Sie für jedes neuronale Netz einen separaten Graphen erstellen und sich um die Synchronisation der Graphen selbst kümmern. Diesen Ansatz nennt man *Between-Graph-Replikation*. Eine übliche Implementierung

verwendet Queues, um das Ausführen der Graphen zu koordinieren (siehe Abbildung 12-13). Eine Anzahl Clients kümmert sich um jeweils ein neuronales Netz, liest eine eigene Eingabequeue aus und schreibt in eine eigene Ausgabequeue. Ein weiterer Client ist dafür verantwortlich, alle Eingabedaten zu lesen und in die Eingabequeues einzuspeisen (alle Eingaben werden in jede Queue kopiert). Schließlich kümmert sich ein letzter Client darum, eine Vorhersage aus jeder Vorhersagequeue auszulesen und diese zur Vorhersage des Ensembles zu aggregieren.

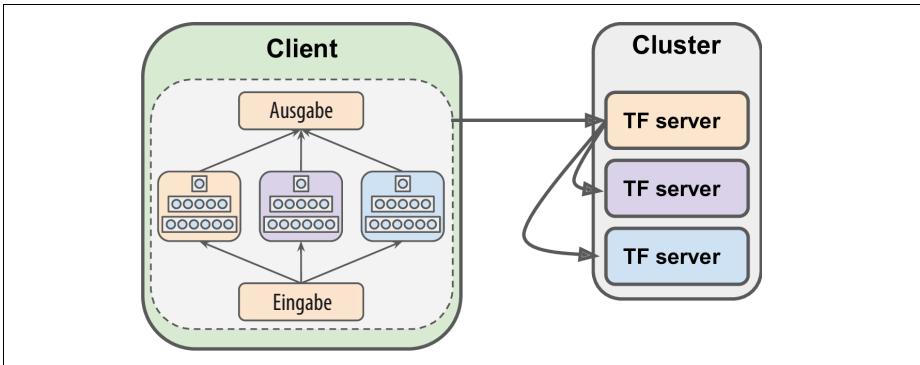


Abbildung 12-12: In-Graph-Replikation

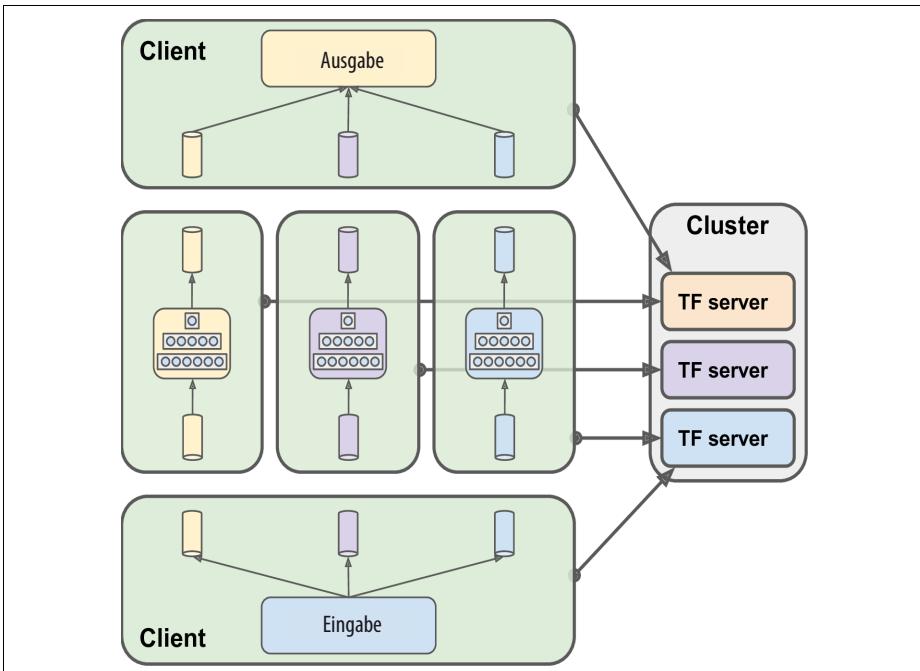


Abbildung 12-13: Between-Graph-Replikation

Beide Lösungen haben Vor- und Nachteile. Die In-Graph-Replikation lässt sich leichter implementieren, da Sie sich nicht um mehrere Clients und Queues kümmern müssen. Andererseits lässt sich die Between-Graph-Replikation etwas einfacher in klar abgegrenzte und testbare Module aufteilen. Obendrein ist sie flexibler. Sie könnten beispielsweise im Aggregator-Client einen Timeout festlegen, sodass das Ensemble nicht fehlschlägt, falls eines der neuronalen Netze abstürzt oder zu lange an einer Vorhersage herumrechnet. In TensorFlow können Sie einen Timeout beim Aufruf der Funktion `run()` festlegen, indem Sie ein `RunOptions`-Objekt mit `timeout_in_ms` übergeben:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000 # 1 Sek Zeitlimit
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # die Operation dequeue hat nach 1s das Zeitlimit überschritten
```

Sie können das Zeitlimit auch über den Konfigurationsparameter `operation_timeout_in_ms` einer Session einstellen, in diesem Fall wird die Funktion `run()` aber beendet, sobald *eine beliebige* Operation länger als die angegebene Zeit dauert:

```
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000 # 1s Zeitlimit für jede Operation

with tf.Session(..., config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...] # die Operation wurde nach 1s beendet
```

Parallelisierte Modelle

Bisher haben wir jedes neuronale Netz auf einer einzelnen Recheneinheit ausgeführt. Wie können wir ein einzelnes neuronales Netz auf mehrere Recheneinheiten verteilen? Dazu müssen Sie Ihr Modell in einzelne Stücke teilen und jedes Teilstück auf einer anderen Recheneinheit ausführen. Dies nennt man ein *parallelisiertes Modell*. Leider sind parallelisierte Modelle in der Praxis nicht einfach, und es kommt auf die Architektur Ihres neuronalen Netzes an. Bei vollständig verbundenen Netzen können Sie mit diesem Ansatz nicht wirklich viel herausholen (siehe Abbildung 12-14). Intuitiv mag es so erscheinen, dass man einfach jede Schicht auf einer anderen Recheneinheit platzieren kann. Dies funktioniert aber nicht, weil jede Schicht auf die Ausgabe der vorigen Schicht warten muss, bevor sie überhaupt etwas tun kann. Vielleicht lässt sich das Netz stattdessen vertikal zerschneiden – die linke Hälfte jeder Schicht auf einer Recheneinheit, die rechte Hälfte auf einer anderen? Das funktioniert ein wenig besser, weil beide Hälften jeder Schicht tatsächlich

parallel arbeiten können. Das Problem hierbei ist, dass jede Hälfte der jeweils nächsten Schicht die Ausgabe der vorigen benötigt. Es kommt also zu einer Menge Kommunikation zwischen den Recheneinheiten (durch die gestrichelten Pfeile dargestellt). Dies macht mit hoher Wahrscheinlichkeit die Vorteile des parallelen Rechnens wieder zunichte, da die Kommunikation zwischen Geräten langsam ist (besonders zwischen separaten Rechnern).

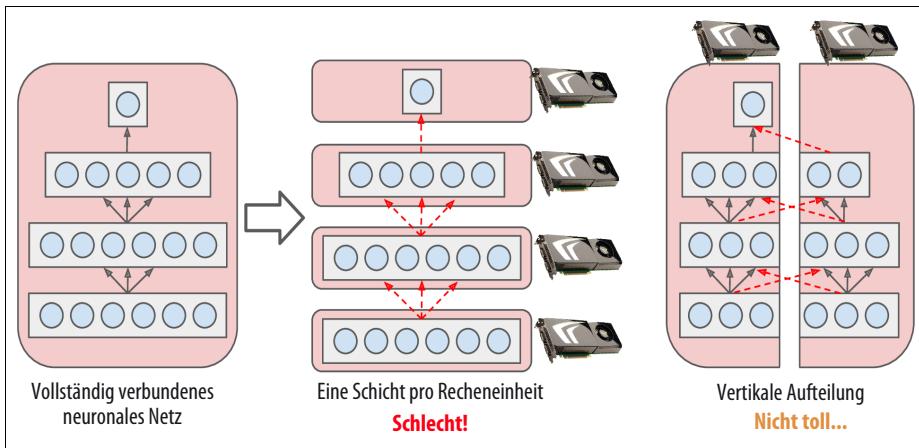


Abbildung 12-14: Aufteilen eines vollständig verbundenen neuronalen Netzes

Allerdings werden wir in Kapitel 13 erfahren, dass die Schichten bei einigen Architekturen neuronaler Netze nur teilweise mit den weiter vorn gelegenen Schichten verbunden sind. Dort ist das effiziente Aufteilen auf mehrere Geräte viel einfacher.

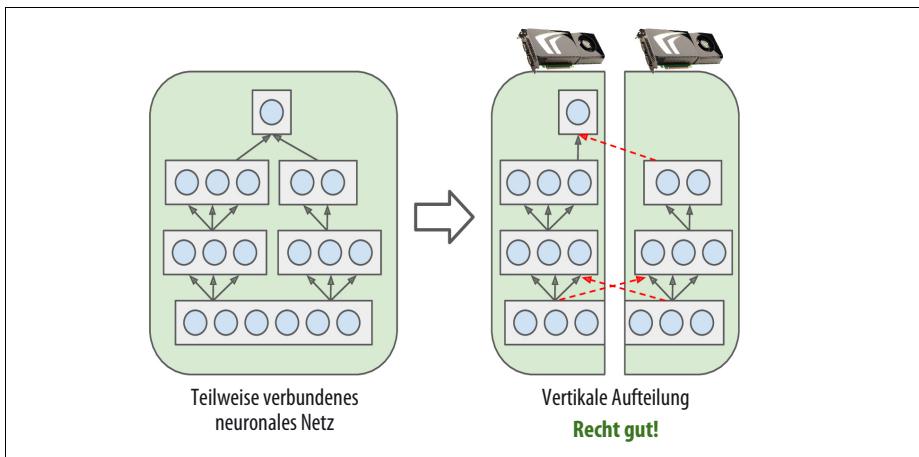


Abbildung 12-15: Aufteilen eines teilweise verbundenen neuronalen Netzes

Wie wir in Kapitel 14 sehen werden, bestehen einige rekurrente Deep-Learning-Netze aus mehreren Schichten *Gedächtniszellen* (auf der linken Seite von Abbil-

dung 12-16). Die Ausgabe einer Zelle zum Zeitpunkt t wird zum Zeitpunkt $t + 1$ wieder als Eingabe verwendet (wie Sie deutlicher auf der rechten Seite von Abbildung 12-16 sehen können). Wenn Sie ein derartiges Netz horizontal aufteilen und jede Schicht auf einem anderen Gerät landet, ist beim ersten Schritt nur ein Gerät aktiv, beim zweiten Schritt sind es zwei, und beim dritten Schritt hat sich das Signal bis zur Ausgabeschicht fortgepflanzt, sodass alle Geräte gleichzeitig aktiv sind. Auch hier gibt es noch eine Menge Kommunikation zwischen den Recheneinheiten, aber da jede Zelle sehr komplex sein kann, ist der Nutzen des parallelen Ausführens größer als der Kommunikationsaufwand.

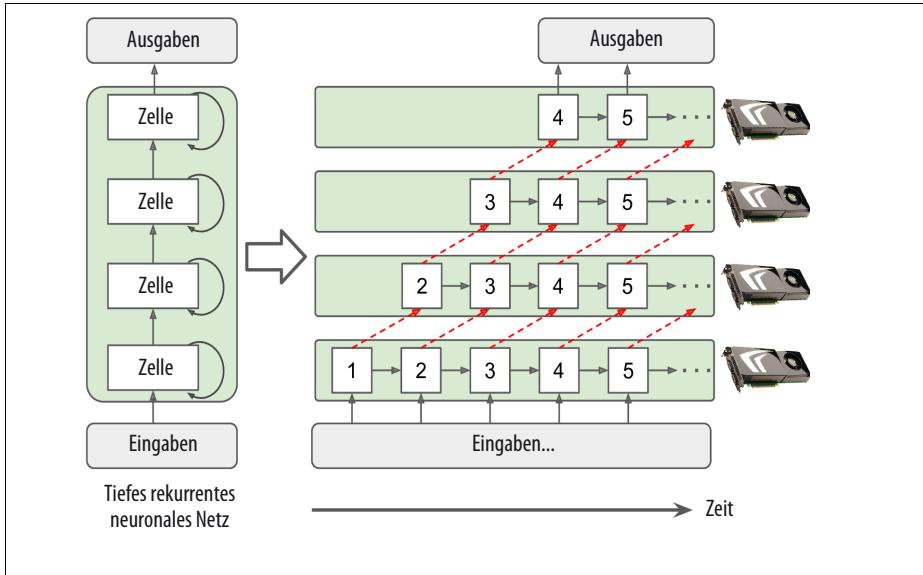


Abbildung 12-16: Aufteilen eines rekurrenten Deep-Learning-Netzes

Zusammengefasst können parallelisierte Modelle das Trainieren oder Ausführen einiger, aber nicht aller Arten neuronaler Netze beschleunigen. Besondere Aufmerksamkeit und Feinabstimmung ist vonnöten, um beispielsweise die Recheneinheiten mit dem höchsten Kommunikationsaufwand auf dem gleichen Rechner zu platzieren.

Parallelisierte Daten

Eine weitere Möglichkeit, das Trainieren eines neuronalen Netzes zu parallelisieren, ist, es auf jeder Recheneinheit zu replizieren, aus allen Repliken gleichzeitig je einen Trainingsschritt mit unterschiedlichen Mini-Batches auszuführen und anschließend die Gradienten zu aggregieren und die Modellparameter zu aktualisieren. Dies bezeichnet man als *parallelisierte Daten* (siehe Abbildung 12-17).

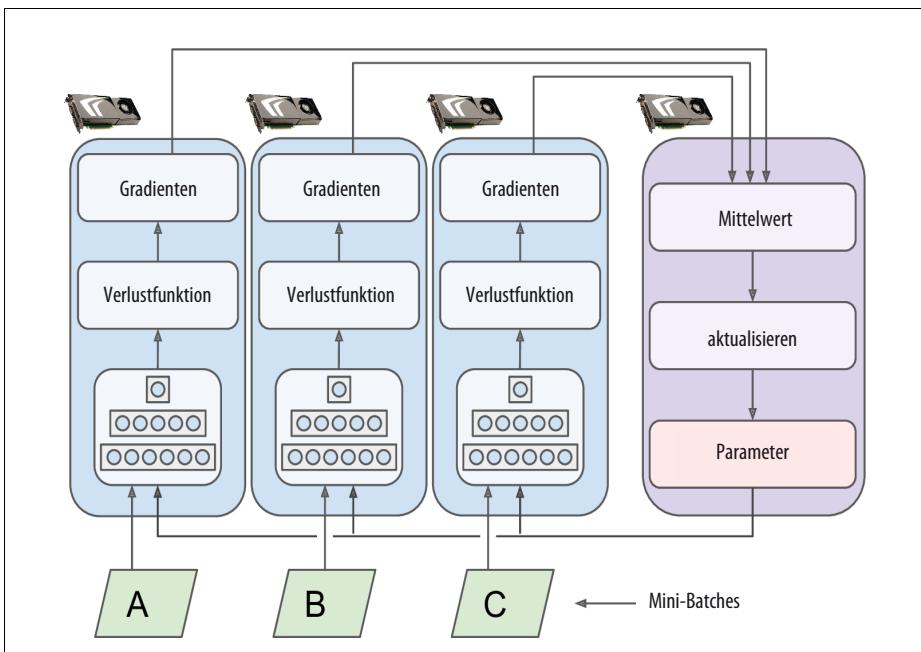


Abbildung 12-17: Parallelisierte Daten

Es gibt zwei Varianten dieses Ansatzes: *synchrone Updates* und *asynchrone Updates*.

Synchrone Updates

Bei *synchrone Updates* wartet der Aggregationsmechanismus, bis alle Gradienten berechnet wurden, bevor er den Durchschnitt berechnet und das Ergebnis verwendet (d.h., die Modellparameter mit den aggregierten Gradienten aktualisiert). Sobald eine Replik ihre Gradienten fertig berechnet hat, muss sie warten, bis die Parameter aktualisiert wurden, bevor sie mit dem nächsten Mini-Batch fortfahren kann. Der Nachteil dabei ist, dass manche Recheneinheiten langsamer als andere sein können. Daher müssen alle übrigen Recheneinheiten bei jedem Schritt auf die langsameren warten. Außerdem müssen die Parameter etwa zeitgleich auf jede Recheneinheit kopiert werden (unmittelbar nachdem die Gradienten angewendet wurden), wodurch sich die Netzwerk-Bandbreite des Parameterservers erschöpfen kann.



Um die Wartezeit bei jedem Schritt zu reduzieren, können Sie die Gradienten der langsamsten Repliken ignorieren (üblicherweise ~10%). Beispielsweise könnten Sie 20 Repliken laufen lassen, aber bei jedem Schritt nur die Gradienten der schnellsten 18 aggregieren und die Gradienten der letzten 2 ignorieren. Sobald die Parameter aktualisiert wurden, können die ersten 18 Repliken

sofort weiterarbeiten, ohne auf die 2 langsamten Repliken zu warten. Diese Konfiguration lässt sich als 18 Repliken plus 2 *Ersatzrepliken* umschreiben.⁵

Asynchrone Updates

Bei asynchronen Updates werden die Modellparameter unmittelbar aktualisiert, sobald eine Replik mit der Berechnung ihrer Gradienten fertig ist. Es findet keine Aggregation statt (den Schritt zur Mittelwertbildung in Abbildung 12-17 können Sie ignorieren), und es ist keine Synchronisierung nötig. Die Repliken arbeiten einfach unabhängig von den übrigen Repliken. Da auf die anderen Repliken nicht gewartet wird, lassen sich bei diesem Verfahren mehr Trainingsschritte pro Minute erzielen. Zwar müssen die Parameter auch hier bei jedem Schritt auf alle Recheneinheiten kopiert werden, dies findet aber zu unterschiedlichen Zeitpunkten statt. Dadurch sinkt das Risiko, dass die Bandbreite des Netzwerks zum Flaschenhals wird.

Parallelisierte Daten mit asynchronen Updates sind ihrer Einfachheit, dem Fehlen einer Verzögerung durch Synchronisierung und der besseren Nutzung des Netzwerks wegen attraktiv. Obwohl das Verfahren in der Praxis gut funktioniert, ist es verwunderlich, dass es überhaupt funktioniert! Wenn nämlich eine Replik die Gradienten anhand der Modellparameter fertig berechnet hat, sind diese Parameter in der Zwischenzeit von den anderen Repliken mehrmals geändert worden (im Durchschnitt $N - 1$ Mal, bei N Repliken). Es gibt keine Garantie dafür, dass die berechneten Gradienten noch immer in die richtige Richtung zeigen (siehe Abbildung 12-18). Gradienten, die hoffnungslos veraltet sind, nennt man auch *Stale Gradients*: Sie verlangsamen das Konvergieren, verursachen Rauschen und Oszillationen (die Lernkurve kann zeitweise oszillieren) oder können sogar zum Divergieren des Trainingsalgorithmus führen.

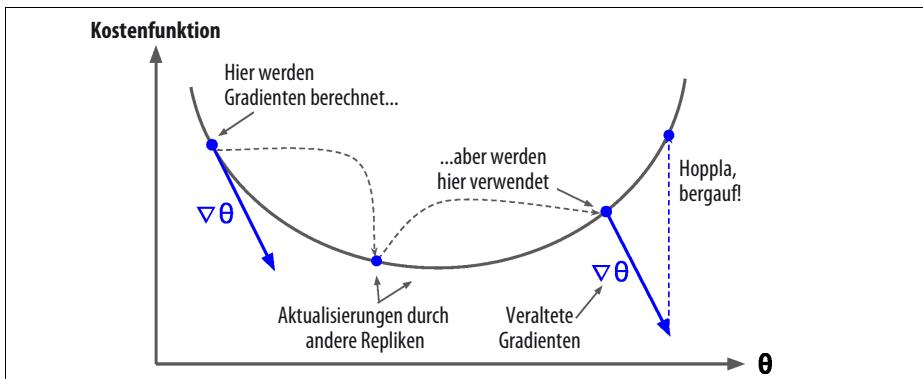


Abbildung 12-18: *Stale Gradients* beim Verwenden asynchroner Updates

5 Dieser Name ist etwas irreführend, da er vermuten lässt, dass einige Repliken irgendwie besonders sind und nichts tun. Tatsächlich sind alle Repliken gleich: Sie alle bemühen sich eifrig, bei jedem Trainingsschritt unter den schnellsten zu sein, und die Verlierer sind bei jedem Schritt andere (es sei denn, einige Recheneinheiten sind tatsächlich langsamer als andere).

Den Auswirkungen veralteter Gradienten lässt sich folgendermaßen entgegenwirken:

- Reduzieren der Lernrate
- Verwerfen oder Herunterskalieren veralteter Gradienten
- Anpassen der Größe der Mini-Batches
- Starten der ersten Epochen mit nur einer Replik (dies nennt man die *Warm-up-Phase*). Veraltete Gradienten richten zu Beginn des Trainings am meisten Schaden an, während die Gradienten tendenziell groß sind und die Parameter sich noch nicht auf ein Tal der Kostenfunktion festgelegt haben. Hier können unterschiedliche Repliken die Parameter in völlig unterschiedliche Richtungen bewegen.

Ein im April 2016 vom Google-Brain-Team veröffentlichter Artikel (<http://goo.gl/9GCiPb>) wertete unterschiedliche Ansätze aus und kam zu dem Schluss, dass parallelisierte Daten mit synchronen Updates und einigen Ersatz-Repliken das effizienteste Verfahren sei. Es konvergiert nicht nur schneller, sondern gelangt auch zu einem besseren Modell. Allerdings ist dies noch immer ein aktives Forschungsbereich, Sie sollen daher die asynchronen Updates noch nicht abtun.

Erschöpfen der Netzwerk-Bandbreite

Sowohl mit synchronen und asynchronen Updates müssen bei parallelisierten Daten die Modellparameter zu Beginn jedes Trainingsschritts von den Parameterservern auf jede Replik kommuniziert werden. Am Ende jedes Trainingsschritts müssen die Gradienten in umgekehrter Richtung versendet werden. Leider bedeutet das, dass es immer einen Punkt gibt, an dem zusätzliche GPUs überhaupt keinen Zugewinn an Geschwindigkeit mehr erbringen, da die Zeit zum Hinein- und Herauskopieren der Daten aus dem RAM der GPU (und eventuell über das Netzwerk) die Beschleunigung durch das Verteilen der Berechnungslast vollständig aufwiegt. An diesem Punkt erhöhen weitere GPUs nur noch die Netzlast und verlangsamen das Trainieren.



Bei einigen, meist kleinen Modellen mit sehr großen Trainingsdatensätzen sind Sie mit einem einzelnen Rechner mit einer GPU oft besser bedient.

Die Sättigung des Netzwerks ist bei großen, dichten Modellen schwerwiegender, da es dort viele Parameter und zu übertragende Gradienten gibt. Bei kleinen Modellen ist dies weniger schlimm (aber auch der Gewinn durch Parallelisierung ist geringer), ebenso bei großen, spärlichen Modellen, da die Gradienten dort zum größten Teil aus Nullen bestehen und sich so effizient übermitteln lassen. Jeff Dean, Initiator und Chefentwickler des Google-Brain-Projekts, berichtet (<http://goo.gl/E4ypxo>), dass die Beschleunigung beim Verteilen von Berechnungen auf 50 GPUs bei dichten Modellen im Bereich von 25 bis 40 Mal und bei spärlicheren Modellen mit 500

GPUs im Bereich von 300 Mal liegt. Wie Sie sehen, skalieren spärliche Modelle deutlich besser. Hier sind einige konkrete Beispiele:

- Übersetzung durch ein neuronales Netz: 6 Mal Beschleunigung auf 8 GPUs
- Inception/ImageNet: 32 Mal Beschleunigung auf 50 GPUs
- RankBrain: 300 Mal Beschleunigung auf 500 GPUs

Diese Zahlen geben den letzten Stand der Dinge im 1. Quartal 2016 wieder. Jenseits von einigen Dutzend GPUs bei einem dichten Modell oder einigen Hundert GPUs bei einem spärlichen Modell macht sich die Sättigung bemerkbar, und die Leistung sinkt. Es findet eine Menge Forschungsarbeit zu diesem Thema statt (Peer-to-Peer-Architekturen anstelle zentraler Parameterserver, verlustbehaftete Komprimierung von Modellen, Optimieren, wann und was Repliken kommunizieren und so weiter). Vermutlich werden wir in den nächsten Jahren beim Parallelisieren neuronaler Netze große Fortschritte sehen.

In der Zwischenzeit können Sie das Sättigungsproblem mit folgenden einfachen Gegenmaßnahmen bekämpfen:

- Gruppieren Sie Ihre GPUs auf wenigen Servern, anstatt sie über viele Server zu verstreuen. Damit vermeiden Sie unnötige Sprünge im Netzwerk.
- Teilen Sie die Parameter (wie oben besprochen) auf mehrere Parameterserver auf.
- Reduzieren Sie die Genauigkeit der Modellparameter von Floats mit 32 Bit (`tf.float32`) auf 16 Bit (`tf.bfloat16`). Damit halbieren Sie die zu übertragende Datenmenge, ohne die Konvergenzrate oder die Leistung des Modells deutlich zu schmälern.



Obwohl eine Präzision von 16 Bit das Minimum für das Trainieren neuronaler Netze ist, können Sie nach dem Trainieren auch auf 8 Bit heruntergehen, um das Modell zu verkleinern und die Berechnungen zu beschleunigen. Dies nennt man *Quantizing* des neuronalen Netzes. Es ist besonders für das Ausführen vortrainierter Modelle auf Mobiltelefonen nützlich. Mehr zu diesem Thema finden Sie in einem großartigen Blogbeitrag (<http://goo.gl/09Cb6v>) von Pete Warden.

Implementierung in TensorFlow

Um parallelisierte Daten mit TensorFlow zu implementieren, müssen Sie sich zunächst zwischen der In-Graph-Replikation und der Between-Graph-Replikation entscheiden sowie zwischen synchronen und asynchronen Updates wählen. Schauen wir uns an, wie sich jede dieser Kombinationen implementieren lässt (vollständige Codebeispiele finden Sie in den Übungen und den Jupyter Notebooks).

Bei der In-Graph-Replikation mit synchronen Updates erstellen Sie einen großen Graphen, der sämtliche Repliken des Modells (auf unterschiedlichen Geräten) und

einige Knoten zum Aggregieren der Gradienten und zur Eingabe in einen Optimierer enthält. Der Code öffnet eine Session auf dem Cluster und führt einfach die Operation zum Trainieren mehrfach aus.

Bei der In-Graph-Replikation mit asynchronen Updates erstellen Sie ebenfalls einen großen Graphen, aber diesmal mit einem Optimierer pro Replik. Pro Replik wird ein Thread gestartet, der den Optimierer der Replik immer wieder ausführt.

Bei der Between-Graph-Replikation mit asynchronen Updates starten Sie mehrere voneinander unabhängige Clients (normalerweise in eigenen Prozessen). Jeder Client trainiert die Replik des Modells, als wäre sie allein auf der Welt, aber die Parameter werden tatsächlich mit den übrigen Repliken geteilt (über einen Resource Container).

Bei der Between-Graph-Replikation mit synchronen Updates, starten Sie ebenfalls mehrere Clients, die jeweils eine Replik des Modells mit geteilten Parametern trainieren, diesmal aber wird der Optimierer (z. B. ein `MomentumOptimizer`) in einen `SyncReplicasOptimizer` verpackt. Jede Replik verwendet diesen Optimierer ganz normal, aber hinter den Kulissen verschickt dieser Optimierer die Gradienten in eine Anzahl Queues (eine pro Variable), die von einem `SyncReplicasOptimizer` der Replik, dem sogenannten *Chief*, ausgelesen werden. Der Chief aggregiert die Gradienten und wendet sie an. Anschließend schreibt er ein Token in eine *Token-Queue* pro Replik, das dieser signalisiert, dass sie mit der Berechnung der nächsten Runde Gradienten fortfahren kann. Dieser Ansatz unterstützt das Verwenden von *Sparse Replicas*.

Wenn Sie sich mit den Übungsaufgaben beschäftigen, werden Sie jede dieser vier Lösungen implementieren. Sie werden damit in der Lage sein, das Gelernte zum Trainieren großer Deep-Learning-Netze auf Dutzenden Servern und GPUs anzuwenden! In den folgenden Kapiteln werden wir uns einigen weiteren wichtigen Architekturen neuronaler Netze widmen, bevor wir uns dem Reinforcement Learning zuwenden.

Übungen

1. Wenn Sie beim Starten Ihres TensorFlow-Programms einen `CUDA_ERROR_OUT_OF_MEMORY` erhalten, was passiert vermutlich? Was können Sie dagegen tun?
2. Was ist der Unterschied zwischen pinnen und platzieren einer Operation auf einer Recheneinheit?
3. Werden bei einer TensorFlow-Installation mit GPU-Unterstützung bei Verwenden der Standardplatzierung sämtliche Operationen auf der ersten GPU platziert?
4. Wenn Sie eine Variable auf `"/gpu:0"` pinnen, lässt sich diese von Operationen auf `/gpu:1` nutzen? Wie sieht es mit Operationen auf `"/cpu:0"` aus? Oder mit Operationen, die auf Geräte auf anderen Servern gepinnt sind?

5. Können zwei auf der gleichen Recheneinheit platzierte Operationen parallel ausgeführt werden?
6. Was ist eine Control Dependency und wann sollten Sie sie einsetzen?
7. Angenommen Sie trainieren ein DNN mehrere Tage lang auf einem TensorFlow-Cluster. Unmittelbar nach Beenden des Trainingsprogramms bemerken Sie, dass Sie vergessen haben, das Modell über einen Saver zu speichern. Ist Ihr trainiertes Modell verloren?
8. Trainieren Sie mehrere DNNs parallel auf einem TensorFlow-Cluster mit unterschiedlich eingestellten Hyperparametern. Dies könnten DNNs zur MNIST-Klassifikation oder eine andere interessante Aufgabe sein. Die einfachste Möglichkeit ist, einfach ein einzelnes Client-Programm zu schreiben, das ein DNN trainiert, und dieses Programm anschließend in mehreren parallelen Prozessen mit unterschiedlichen Hyperparametern auszuführen. Das Programm sollte Kommandozeilenoptionen haben, um zu steuern, auf welchem Server und welcher Recheneinheit das DNN platziert werden soll, welcher Resource Container und welche Werte für die Hyperparameter verwendet werden sollen (stellen Sie sicher, dass jedes DNN einen anderen Resource Container verwendet). Verwenden Sie einen Validierungsdatensatz oder Kreuzvalidierung, um die drei besten Modelle zu ermitteln.
9. Erstellen Sie ein Ensemble aus den besten drei Modellen der vorigen Aufgabe. Definieren Sie es in einem einzelnen Graphen und stellen Sie sicher, dass jedes DNN auf einer anderen Recheneinheit läuft. Evaluieren Sie es auf dem Validierungsdatensatz: Erbringt das DNN eine höhere Leistung als die einzelnen DNNs?
10. Trainieren Sie ein DNN mit Replikation zwischen Graphen und parallelen Daten mit asynchronen Updates. Messen Sie, wie lange es dauert, eine zufriedenstellende Leistung zu erreichen. Probieren Sie als Nächstes synchrone Updates aus. Führen synchrone Updates zu einem besseren Modell? Verläuft das Trainieren schneller? Teilen Sie das DNN vertikal auf und platzieren Sie jede vertikale Scheibe auf einer anderen Recheneinheit. Trainieren Sie das Modell erneut. Ist das Trainieren diesmal schneller? Gibt es Unterschiede in der Leistung?

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

Convolutional Neural Networks

Obwohl der Supercomputer Deep Blue von IBM den Schachweltmeister Garry Kasparov bereits im Jahr 1996 schlug, waren Computer bis vor Kurzem scheinbar nicht fähig, triviale Aufgaben wie das Erkennen von Hündchen auf Bildern oder gesprochenen Wörtern zu lösen. Warum gehen Menschen diese Aufgaben so leicht von der Hand? Die Antwort liegt daran, dass die Wahrnehmung hauptsächlich außerhalb unseres Bewusstseins, also in den auf Sicht, Gehör oder andere Sinne spezialisierten Bereichen unseres Gehirns stattfindet. Bis sensorische Information unser Bewusstsein erreicht, ist sie bereits mit hoch abstrakten Merkmalen angereichert; wenn Sie beispielsweise ein Bild mit einem niedlichen Hündchen betrachten, können Sie sich nicht entscheiden, das Hündchen *nicht* zu sehen oder seine Niedlichkeit *nicht* zu bemerken. Sie können auch nicht erklären, *wie* Sie ein niedliches Hündchen erkennen; es ist Ihnen einfach klar. Wir können unserer subjektiven Erfahrung nicht ganz trauen: Wahrnehmung ist überhaupt keine triviale Aufgabe. Um sie zu verstehen, müssen wir uns anschauen, wie sensorische Module funktionieren.

Convolutional Neural Networks (CNNs) sind aus Untersuchungen des visuellen Cortex des Gehirns entstanden. Sie wurden seit den 1980ern zur Bilderkennung eingesetzt. In den letzten Jahren konnten CNNs dank der angewachsenen Rechenkapazitäten, der Menge an verfügbaren Trainingsdaten und der in Kapitel 11 vorgestellten Tricks zum Trainieren von Deep-Learning-Netzen eine übermenschliche Leistung erreichen. Sie finden sich in Diensten zur Bildersuche, selbstfahrenden Autos, Systemen zur automatischen Videoklassifikation und anderen. CNNs sind außerdem nicht auf die visuelle Wahrnehmung beschränkt: Sie sind auch bei anderen Aufgaben wie der *Stimmerkennung* oder *Sprachverarbeitung* (NLP) erfolgreich; wir werden uns hier aber auf die visuellen Anwendungen konzentrieren.

In diesem Kapitel stellen wir vor, woher CNNs stammen, aus welchen Elementen sie bestehen und wie sie sich mit TensorFlow implementieren lassen. Anschließend präsentieren wir einige der besten CNN-Architekturen.

Der Aufbau des visuellen Cortex

David H. Hubel und Torsten Wiesel führten in den Jahren 1958 (<http://goo.gl/VLxXf9>)¹ und 1959 (<http://goo.gl/OYuFUZ>)² eine Reihe Experimente an Katzen durch (und ein paar Jahre später an Affen (<http://goo.gl/95F7QH>)³), die wichtige Erkenntnisse zur Struktur des visuellen Cortex lieferten (die Autoren erhielten für Ihre Arbeit im Jahr 1981 den Nobelpreis in Physiologie und Medizin). Sie konnten vor allem zeigen, dass viele Neuronen im visuellen Cortex ein kleines *lokales Wahrnehmungsfeld* haben, also nur auf visuelle Stimuli in einem begrenzten Bereich des Gesichtsfelds reagieren (in Abbildung 13-1 sind die lokalen Wahrnehmungsfelder von fünf Neuronen durch gestrichelte Kreise dargestellt). Die Wahrnehmungsfelder unterschiedlicher Neuronen können einander überlappen und gemeinsam das gesamte Gesichtsfeld abdecken. Die Autoren konnten auch nachweisen, dass einige Neuronen nur auf Bilder mit horizontalen Linien reagieren, andere nur auf Linien mit anderer Ausrichtung (zwei Neuronen können das gleiche Wahrnehmungsfeld haben, aber auf eine unterschiedliche Orientierung der Linien reagieren). Sie stellten auch fest, dass einige Neuronen größere Wahrnehmungsfelder haben und auf komplexere Muster reagieren, die Kombinationen kleinteiliger Muster sind. Diese Beobachtungen führten zu der Vorstellung, dass Neuronen auf höherer Abstraktionsebene die Ausgabe der benachbarten Neuronen auf niedrigerer Abstraktionsebene verarbeiten (in Abbildung 13-1 beachten Sie, dass jedes Neuron nur mit wenigen Neuronen der vorigen Schicht verbunden ist). Diese mächtige Architektur ist in der Lage, alle möglichen komplexen Muster in einem beliebigen Teil des Gesichtsfelds zu erkennen.

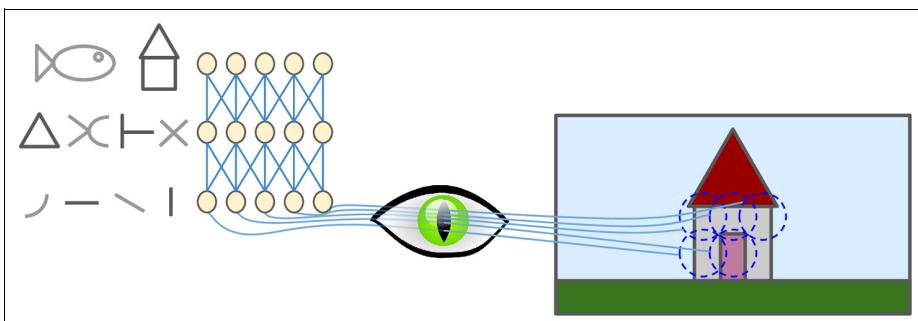


Abbildung 13-1: Lokales Wahrnehmungsfeld im visuellen Cortex

Diese Untersuchungen des visuellen Cortex inspirierten das im Jahr 1980 vorgestellte Neocognitron (<http://goo.gl/XwiXs9>)⁴, das sich nach und nach zu dem ent-

1 »Single Unit Activity in Striate Cortex of Unrestrained Cats«, D. Hubel and T. Wiesel (1958).

2 »Receptive Fields of Single Neurones in the Cat's Striate Cortex«, D. Hubel and T. Wiesel (1959).

3 »Receptive Fields and Functional Architecture of Monkey Striate Cortex«, D. Hubel and T. Wiesel (1968).

wickelte, was wir heute als *Convolutional Neural Networks* bezeichnen. Ein wichtiger Meilenstein war dabei ein Artikel aus dem Jahr 1998 (<http://goo.gl/A347S4>)⁵ von Yann LeCun, Léon Bottou, Yoshua Bengio und Patrick Haffner, in dem die berühmte *LeNet-5*-Architektur vorgestellt wurde. Sie wurde im großen Stil zum Erkennen handgeschriebener Ziffern auf Schecks eingesetzt. Diese Architektur enthält einige Bausteine, die Sie bereits kennen, etwa vollständig verbundene Schichten und die sigmoide Aktivierungsfunktion. Sie enthält aber auch zwei neue Bauelemente: *Convolutional Layers* und *Pooling Layers*. Schauen wir uns beide einmal an.



Warum verwenden wir nicht einfach ein gewöhnliches Deep-Learning-Netz mit vollständig verbundenen Schichten zur Bilderkennung? Leider funktioniert dies nur für kleine Bilder (z.B. MNIST) und scheitert bei größeren Bildern wegen der riesigen Zahl benötigter Parameter. Zum Beispiel enthält ein Bild der Größe 100×100 insgesamt 10000 Pixel, und wenn die erste Schicht nur 1000 Neuronen enthält (was die zur nächsten Schicht übertragene Informationsmenge bereits erheblich einschränkt), sind insgesamt bereits 10 Millionen Verbindungen nötig. Und das ist nur die erste Schicht. CNNs lösen dieses Problem durch teilweise verbundene Schichten.

Convolutional Layer

Der wichtigste Bestandteil eines CNN sind die *Convolutional Layer*:⁶ Neuronen im ersten Convolutional Layer sind nicht mit jedem Pixel im Eingabebild verbunden (wie in den vorigen Kapiteln), sondern nur mit Pixeln in ihrem Wahrnehmungsfeld (siehe Abbildung 13-2). Im Gegenzug ist jedes Neuron im zweiten Convolutional Layer ausschließlich mit Neuronen innerhalb eines kleinen Rechtecks der ersten Schicht verbunden. Durch diese Architektur kann das Netzwerk sich in der ersten verborgenen Schicht auf kleinteilige Merkmale konzentrieren, diese in der nächsten verborgenen Schicht zu übergeordneten Merkmalen zusammensetzen und so weiter. Diese hierarchische Struktur ist in echten Bildern verbreitet, weswegen CNNs in der Bilderkennung so gut funktionieren.

-
- 4 »Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position«, K. Fukushima (1980).
 - 5 »Gradient-Based Learning Applied to Document Recognition«, Y. LeCun et al. (1998).
 - 6 Eine Konvolution (Faltung) ist eine mathematische Operation, die eine Funktion mit einer zweiten überlagert und das Integral ihrer punktweisen Multiplikation ermittelt. Sie hängt eng mit der Fourier-Transformation und der Laplace-Transformation zusammen und wird in der Signalverarbeitung häufig eingesetzt. Convolutional Layers verwenden Kreuzkorrelationen, die Konvolutionen sehr ähnlich sind (auf <http://goo.gl/HAfXxD> finden Sie Details).

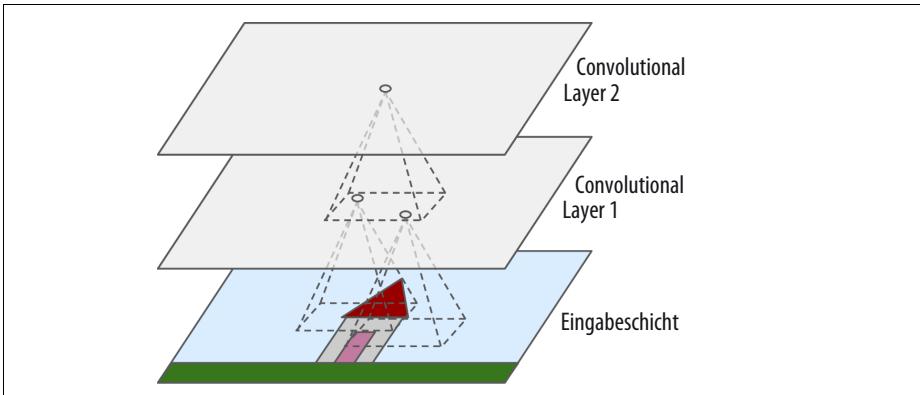


Abbildung 13-2: Schichten eines CNN mit rechteckigen lokalen Wahrnehmungsfeldern



Bisher bestanden alle betrachteten mehrschichtigen neuronalen Netze aus einer langen Reihe Neuronen. Wir mussten Eingabebilder zu 1-D-Daten verflachen, um sie in das neuronale Netz einzugeben. Nun ist jede Schicht in 2-D angeordnet, wodurch sich Neuronen leichter ihren jeweiligen Eingaben zuordnen lassen.

Ein Neuron in Zeile i und Spalte j einer bestimmten Schicht ist mit den Ausgaben der Neuronen in der vorigen Schicht in den Zeilen i bis $i + f_h - 1$ und den Spalten j bis $j + f_w - 1$ verbunden, wobei f_h und f_w die Höhe und Breite des Wahrnehmungsfelds sind (siehe Abbildung 13-3). Damit eine Schicht die gleiche Höhe und Breite wie die vorige hat, ist es üblich, Nullen um die Eingaben herum zu platzieren wie die Abbildung zeigt. Dies nennt man auch *Zero Padding*.

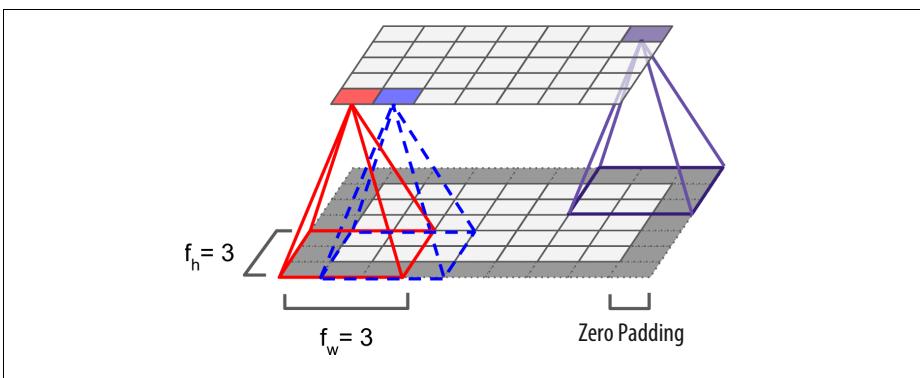


Abbildung 13-3: Verbindungen zwischen Schichten und Zero Padding

Es ist auch möglich, eine große Eingabeschicht mit einer viel kleineren Schicht zu verbinden, indem die Wahrnehmungsfelder wie in Abbildung 13-4 in größeren Abständen gestaffelt werden. Der Abstand zwischen zwei aufeinanderfolgenden Wahrnehmungsfeldern wird als *Schrittweite* (engl. stride) bezeichnet. Im Dia-

gramm ist eine Eingabeschicht der Größe 5×7 (zuzüglich Zero Padding) mit einer Schicht der Größe 3×4 über Wahrnehmungsfelder der Größe 3×3 mit einer Schrittweite von 2 verbunden (in diesem Beispiel ist die Schrittweite in beiden Richtungen gleich groß, dies muss aber nicht so sein). Ein Neuron in Zeile i und Spalte j in der höher gelegenen Schicht ist mit den Ausgaben der Neuronen aus der vorigen Schicht in den Zeilen $i \times s_h$ bis $i \times s_h + f_h - 1$ und den Spalten $j \times s_w$ bis $j \times s_w + f_w - 1$ verbunden, wobei s_h und s_w die vertikalen und horizontalen Schrittweiten sind.

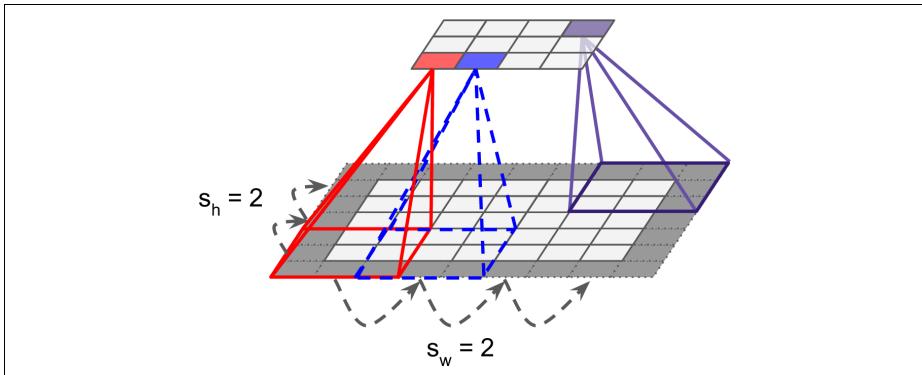


Abbildung 13-4: Dimensionsreduktion über eine Schrittweite von 2

Filter

Die Gewichte eines Neurons lassen sich als kleines Bild mit der Größe des Wahrnehmungsfelds darstellen. Das Beispiel in Abbildung 13-5 zeigt zwei mögliche Sätze Gewichte, genannt *Filter* (oder *Convolution-Kernel*). Der erste zeigt ein schwarzes Quadrat mit einer vertikalen weißen Linie in der Mitte (es ist eine 7×7 -Matrix voller Nullen sowie Einsen in der mittleren Spalte); Neuronen mit diesen Gewichten werden alles in ihrem Wahrnehmungsfeld außer der mittleren Spalte ignorieren (da alle Eingaben außer denen entlang der vertikalen Mittellinie mit 0 multipliziert werden). Der zweite Filter ist ein schwarzes Quadrat mit einer horizontalen weißen Linie in der Mitte. Wieder werden Neuronen mit diesen Gewichten alles in ihrem Wahrnehmungsfeld außer dieser horizontalen Mittellinie ignorieren.

Wenn nun sämtliche Neuronen einer Schicht den Filter mit der vertikalen Linie verwenden (und den gleichen Bias-Term) und Sie dem Netz das Bild in Abbildung 13-5 (unten) zeigen, erhalten Sie als Ausgabe das Bild links oben. Die vertikalen weißen Linien werden verstärkt, der Rest dagegen verschwimmt. In ähnlicher Weise erhalten Sie das Bild rechts oben, wenn alle Neuronen den Filter mit der horizontalen Linie verwenden; die horizontalen weißen Linien werden hervorgehoben, und der Rest verschwimmt. Daher bildet eine Schicht Neuronen mit dem gleichen Filter eine *Feature Map*, die die dem Filter am ähnlichsten Bildbereiche hervorhebt. Beim Trainieren findet ein CNN die für die Aufgabe nützlichsten Filter

und lernt, diese zu komplexeren Mustern zu kombinieren (z.B. ein Kreuz ist ein Bildbereich, in dem sowohl der vertikale als auch der horizontale Filter aktiv sind).

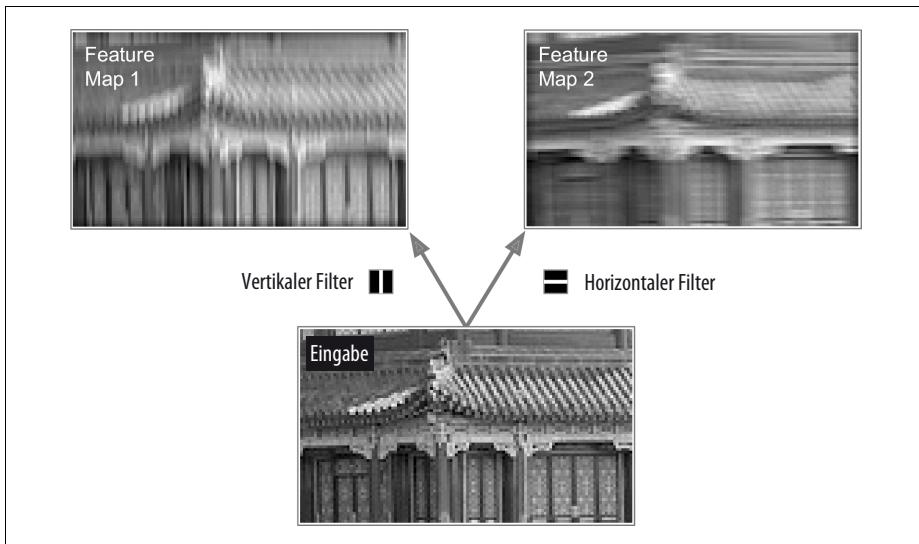


Abbildung 13-5: Anwenden von zwei unterschiedlichen Filtern, um zwei Feature Maps zu erhalten

Stapeln mehrerer Feature Maps

Bisher haben wir jeden Convolutional Layer als dünne 2-D-Schicht dargestellt, tatsächlich bestehen sie aber aus mehreren Feature Maps gleicher Größe, die sich besser in 3-D darstellen lassen (siehe Abbildung 13-6). Innerhalb einer Feature Map haben sämtliche Neuronen die gleichen Parameter (Gewichte und Bias-Term), aber unterschiedliche Feature Maps können sich in den Parametern unterscheiden. Das Wahrnehmungsfeld eines Neurons entspricht dem oben beschriebenen, es erstreckt sich aber über alle Feature Maps der vorigen Schicht. Zusammengefasst wendet ein Convolutional Layer gleichzeitig mehrere Filter auf seine Eingaben an, wodurch er mehrere Merkmale an beliebiger Stelle der Eingaben erkennen kann.



Da sämtliche Neuronen einer Feature Map die gleichen Parameter haben, reduziert sich die Anzahl Parameter im Modell erheblich. Noch wichtiger ist aber, dass ein CNN, das ein Muster an einer Stelle erkennen kann, dieses auch an einer beliebigen anderen Stelle erkennt. Im Gegensatz dazu kann ein gewöhnliches DNN, das ein Muster an einer Stelle erlernt hat, es auch nur an dieser Stelle wiedererkennen.

Außerdem bestehen auch die Eingabebilder aus mehreren Schichten, nämlich eine pro *Farbkanal*: Rot, Grün und Blau (RGB). Graustufenbilder bestehen nur aus

einem Kanal, aber manche Bilder besitzen mehr – z.B. Satellitenbilder, die zusätzliche Lichtfrequenzen erfassen (wie das Infrarotspektrum).

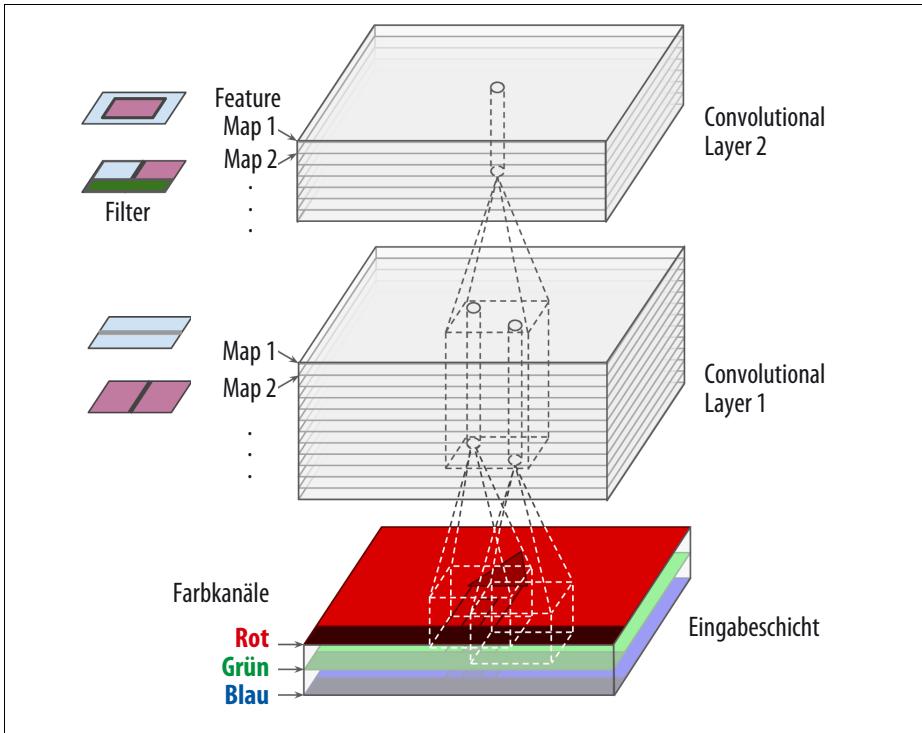


Abbildung 13-6: Convolutional Layers mit mehreren Feature Maps und Bilder mit drei Kanälen

Ein Neuron in Zeile i und Spalte j der Feature Map k im Convolutional Layer l ist mit den Ausgaben der Neuronen der vorigen Schicht $l - 1$ in den Zeilen $i \times s_h$ bis $i \times s_h + f_h - 1$ und den Spalten $j \times s_w$ bis $j \times s_w + f_w - 1$ über alle Feature Maps (in Schicht $l - 1$) verbunden. Beachten Sie, dass alle Neuronen, die in der gleichen Zeile i und Spalte j , aber in unterschiedlichen Feature Maps liegen, mit den Ausgaben der exakt gleichen Neuronen der vorigen Schicht verbunden sind.

Formel 13-1 fasst die bisherigen Erklärungen in einer großen mathematischen Formel zusammen: Sie zeigt, wie sich die Ausgabe eines bestimmten Neurons in einem Convolutional Layer berechnen lässt. Sie ist wegen all der Indizes ein wenig unansehnlich, aber sie tut nichts weiter, als eine gewichtete Summe aller Eingaben zu berechnen und den Bias-Term zu addieren.

Formel 13-1: Berechnung der Ausgabe eines Neurons in einem Convolutional Layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n'-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ ist die Ausgabe des Neurons in Zeile i und Spalte j in Feature Map k des Convolutional Layers (Schicht l).
- Wie oben erklärt, sind s_h und s_w die vertikalen und horizontalen Schrittweiten, f_h und f_w sind Höhe und Breite des Wahrnehmungsfelds, und f_n' ist die Anzahl der Feature Maps in der vorigen Schicht (Schicht $l - 1$).
- $x_{i',j',k'}$ ist die Ausgabe des Neurons in Schicht $l - 1$, Zeile i' , Spalte j' , Feature Map k' (oder Kanal k' , falls die vorige Schicht die Eingabeschicht ist).
- b_k ist der Bias-Term von Feature Map k (in Schicht l). Sie können ihn sich als Regler für die Helligkeit der Feature Map k vorstellen.
- $w_{u,v,k',k}$ ist das Gewicht der Verbindung zwischen einem beliebigen Neuron in Feature Map k der Schicht l und seiner Eingabe in Zeile u , Spalte v (relativ zum Wahrnehmungsfeld des Neurons) und Feature Map k' .

Implementierung in TensorFlow

In TensorFlow wird jedes Eingabebild normalerweise als 3-D-Tensor mit den Abmessungen [Höhe, Breite, Kanäle] repräsentiert. Ein Mini-Batch ist demnach ein 4-D-Tensor mit den Abmessungen [Größe des Mini-Batches, Höhe, Breite, Kanäle]. Die Gewichte eines Convolutional Layer sind ein 4-D-Tensor mit den Abmessungen [f_h, f_w, f_n, f_n]. Die Bias-Terme eines Convolutional Layer sind einfach ein 1-D-Tensor der Größe [f_n].

Betrachten wir ein einfaches Beispiel. Der folgende Code lädt zwei Beispielbilder mit der Scikit-Learn-Funktion `load_sample_images()` (diese lädt zwei Farbbilder, eines mit einem chinesischen Tempel, das andere mit einer Blume). Dann erstellt er zwei Filter der Größe 7×7 (einen mit einer vertikalen weißen Mittellinie, den anderen mit einer horizontalen weißen Mittellinie) und wendet diese über einen Convolutional Layer auf beide Bilder an. Der Convolutional Layer wird mit der TensorFlow-Funktion `tf.nn.conv2d()` erstellt (mit Zero Padding und einer Schrittweite von 2). Eine der sich ergebenden Feature Maps wird als Bild dargestellt (ähnlich dem Bild oben rechts in Abbildung 13-5).

```
import numpy as np
from sklearn.datasets import load_sample_images

# Laden der Beispielbilder
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")
dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Erstelle 2 Filter
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertikale Linie
filters[3, :, :, 1] = 1 # horizontale Linie

# Erstelle einen Graphen mit der Eingabe X
```

```

# und einem Convolutional Layer mit den zwei Filtern
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

    plt.imshow(output[0, :, :, 1], cmap="gray") # zeichne die 2. Feature Map
                                                # des 1. Bilds
    plt.show()

```

Ein Großteil dieses Codes ist selbsterklärend, nur die Zeile mit `tf.nn.conv2d()` verdient ein paar Erläuterungen:

- `X` ist der Mini-Batch mit den Eingabedaten (wie oben erwähnt, ein 4-D-Tensor).
- `filters` sind die anzuwendenden Filter (ebenfalls wie oben erläutert, ein 4-D-Tensor).
- `strides` ist ein 1-D-Array mit vier Elementen, wobei die beiden mittleren Elemente die vertikalen und horizontalen Schrittweiten sind (s_h und s_w). Das erste und das letzte Element müssen im Moment auf 1 gesetzt werden. Eines Tages könnten sie dafür eingesetzt werden, um eine Schrittweite anzugeben für Batches (um einige Datenpunkte zu überspringen) oder Kanäle (um einige der Feature Maps oder Kanäle der vorigen Schicht zu überspringen).
- `padding` muss auf "VALID" oder "SAME" gesetzt werden:
 - Mit dem Wert "VALID" verwendet der Convolutional Layer *kein* Zero Padding und ignoriert je nach Schrittweite einige Zeilen und Spalten am unteren und rechten Rand des Bilds, wie Abbildung 13-7 zeigt (der Einfachheit halber ist hier nur die horizontale Dimension dargestellt, das gleiche Prinzip findet natürlich auch bei der vertikalen Dimension Anwendung).

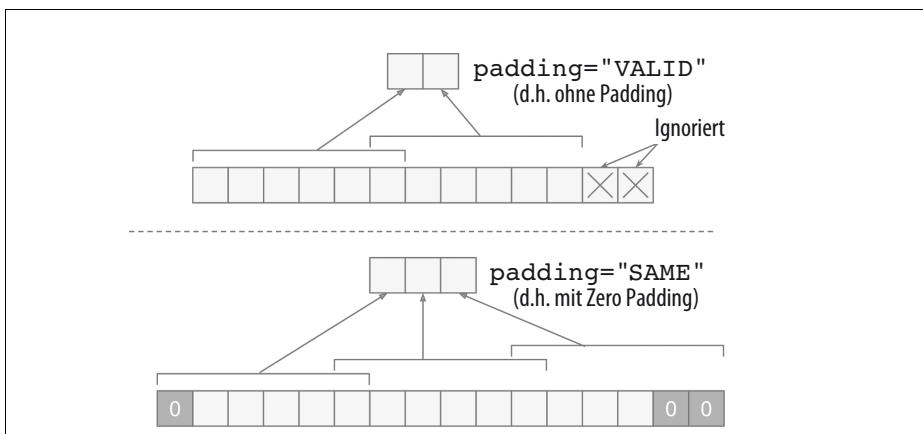


Abbildung 13-7: Optionen beim Padding – Breite der Eingabe: 13, Filter-Breite: 6, Schrittweite: 5

- Mit dem Wert "SAME" verwendet der Convolutional Layer bei Bedarf Zero Padding. In diesem Fall ist die Anzahl der Ausgabeneuronen gleich der Anzahl der Eingabeneuronen geteilt durch die Schrittweite (aufgerundet, in diesem Beispiel $(13 / 5) = 3$). Die Nullen werden so gleichmäßig wie möglich um die Eingabedaten platziert.

In diesem einfachen Beispiel haben wir die Filter von Hand erstellt, aber in einem echten CNN würde der Trainingsalgorithmus die besten Filter automatisch ermitteln. TensorFlow enthält die Funktion `tf.layers.conv2d()`, die die Filtervariable (den Kernel) erstellt und zufällig initialisiert. Beispielsweise erstellt der folgende Code einen Platzhalter für die Eingabedaten, gefolgt von einem Convolutional Layer mit zwei Feature Maps der Größe 7×7 , mit den Schrittweiten 2×2 (diese Funktion benötigt lediglich Werte für die vertikale und horizontale Schrittweite) und dem Wert "SAME" für das Padding:

```
X = tf.placeholder(shape=(None, height, width, channels), dtype=tf.float32)
conv = tf.layers.conv2d(X, filters=2, kernel_size=7, strides=[2,2],
                      padding="SAME")
```

Leider enthalten Convolutional Layers recht viele Hyperparameter: Sie müssen die Anzahl Filter, deren Höhe und Breite, die Schrittweiten und die Art von Padding auswählen. Wie immer können Sie die richtigen Werte für die Hyperparameter durch Kreuzvalidierung herausfinden, dies ist aber sehr zeitaufwendig. Wir werden später einige verbreitete Architekturen von CNNs besprechen, damit Sie eine Vorstellung davon haben, welche Hyperparametereinstellungen in der Praxis am besten funktionieren.

Speicherbedarf

Ein weiteres Problem bei CNNs ist der riesige Speicherbedarf von Convolutional Layers, besonders beim Trainieren, weil der Rückwärtsdurchlauf beim Backpropagation-Verfahren sämtliche beim Vorwärtsdurchlauf berechneten Zwischenergebnisse benötigt.

Betrachten wir als Beispiel einen Convolutional Layer mit 5×5 Filtern, der 200 Feature Maps der Größe 150×100 mit stride 1 und SAME-Padding ausgibt. Wenn die Eingabe ein RGB-Bild der Größe 150×100 ist (drei Kanäle), so beträgt die Anzahl der Parameter $(5 \times 5 \times 3 + 1) \times 200 = 15200$ (das +1 ist für die Bias-Terme), was im Vergleich zu einer vollständig verbundenen Schicht recht wenig ist.⁷ Allerdings enthält jede der 200 Feature Maps 150×100 Neuronen, und jedes dieser Neuronen muss eine gewichtete Summe seiner $5 \times 5 \times 3 = 75$ Eingaben berechnen: Das sind insgesamt 225 Millionen Gleitkommamultiplikationen. Nicht so schlimm wie bei einer vollständig verbundenen Schicht, aber noch immer recht rechenintensiv. Wenn außerdem die Feature Maps als 32-Bit-Floats abgelegt sind, beansprucht die

⁷ Eine vollständig verbundene Schicht mit 150×100 Neuronen, von denen jedes mit allen $150 \times 100 \times 3$ Eingaben verbunden ist, hätte $150^2 \times 100^2 \times 3 = 675$ Millionen Parameter!

Ausgabe des Convolutional Layer $200 \times 150 \times 100 \times 32 = 96$ Millionen Bits (etwa 11.4 MB) im RAM.⁸ Und dies nur für einen Datenpunkt! Wenn ein Batch beim Trainieren 100 Datenpunkte enthält, verbraucht diese Schicht mehr als 1 GB an RAM!

Während der Inferenz (d.h. beim Treffen von Vorhersagen für einen neuen Datenpunkt) dann das für eine Schicht belegte RAM freigegeben werden, sobald die nächste Schicht berechnet wurde. Sie benötigen also nur genug RAM für zwei aufeinanderfolgende Schichten. Beim Training müssen aber sämtliche Rechenergebnisse aus dem Vorwärtsdurchlauf für den Rückwärtsdurchlauf aufgehoben werden, daher ist der gesamte Speicherbedarf (mindestens) so groß wie das von allen Schichten insgesamt benötigte RAM.



Wenn das Trainieren aufgrund fehlenden Speicherplatzes fehlschlägt, können Sie versuchen, die Größe der Mini-Batches zu reduzieren. Sie können alternativ die Dimensionalität über die Schrittweite reduzieren oder einige Schichten auslassen. Sie können auch Floats mit 16 Bit statt mit 32 Bit verwenden oder das CNN über mehrere Geräte verteilen.

Nun wenden wir uns dem zweiten verbreiteten Bauelement von CNNs zu: dem *Pooling Layer*.

Pooling Layer

Sobald Sie die Funktionsweise von Convolutional Layers verstanden haben, sind die Pooling Layers recht einfach zu erfassen. Ihr Zweck ist, *Unterstichproben* aus dem Eingabebild zu ziehen (d.h., es zu verkleinern), um die Rechenlast und die Anzahl der Parameter zu verringern (und nebenbei das Risiko für Overfitting zu senken). Eine Verkleinerung des Eingabebilds macht das neuronale Netz auch weniger anfällig für Verschiebungen (engl. *location invariance*).

Wie bei Convolutional Layers ist jedes Neuron in einem Pooling Layer mit einer begrenzten Anzahl Neuronen der vorigen Schicht verbunden, die in einem rechteckigen Wahrnehmungsfeld liegen. Sie müssen wie zuvor dessen Größe, Schrittweite und Art des Padding angeben. Ein Pooling-Neuron besitzt aber keine Gewichte; es aggregiert lediglich die Eingabedaten über eine Aggregatfunktion wie max oder mean. Abbildung 13-8 zeigt einen *Max-Pooling Layer*, die verbreitetste Art eines Pooling Layer. In diesem Beispiel verwenden wir einen *Pooling-Kernel* der Größe 2×2 , einen stride von 2 und kein Padding. Nur der größte Eingabewert in jedem Kernel wird an die nächste Schicht übergeben, alle übrigen Eingaben werden verworfen.

⁸ 1 MB = 1024 KB = 1024×1024 Bytes = $1024 \times 1024 \times 8$ Bits.

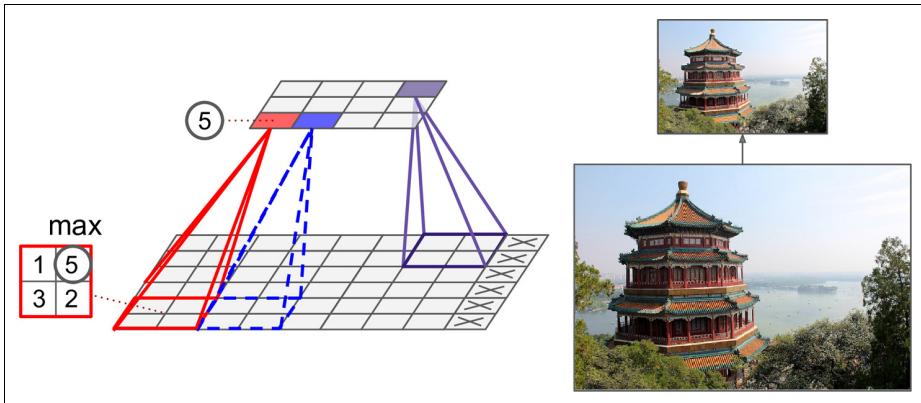


Abbildung 13-8: Max-Pooling Layer (Pooling-Kernell der Größe 2 × 2, Schrittweite 2, kein Padding)

Diese Art Schicht ist offensichtlich eine sehr destruktive: Selbst mit einem winzigen Kernel der Größe 2×2 und einer Schrittweite von 2 ist die Ausgabe in beiden Richtungen nur noch halb so groß (damit beträgt die Fläche ein Viertel), und es werden 75% der Eingabedaten verworfen.

Ein Pooling Layer prozessiert normalerweise jeden Eingabekanal einzeln, die Tiefe der Ausgabe bleibt also die gleiche. Sie können alternativ auch über die Tiefe als Dimension poolen, wie wir gleich sehen werden. In diesem Fall bleiben die räumlichen Dimensionen (Höhe und Breite) unverändert, aber die Anzahl der Kanäle verringert sich.

Ein Max-Pooling Layer lässt sich in TensorFlow recht leicht implementieren. Der folgende Code erstellt einen Max-Pooling Layer mit einem Kernel der Größe 2×2 , Schrittweite 2 und ohne Padding und wendet diesen anschließend auf alle Bilder im Datensatz an:

```
[...] # Lade den Bilddatensatz wie oben

# Erstelle einen Graphen mit der Eingabe X und einem Max-Pooling Layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1], padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8)) # zeichne die Ausgabe für das 1. Bild
plt.show()
```

Das Argument `ksize` enthält die Abmessungen des Kernels entlang aller vier Dimensionen des Eingabe-Tensors: [Größe des Batches, Höhe, Breite, Kanäle]. TensorFlow unterstützt im Moment kein Pooling über mehrere Datenpunkte, daher muss das erste Element von `ksize` 1 betragen. Außerdem ist kein gleichzeitiges Pooling über die räumlichen Dimensionen (Höhe und Breite) und die Tiefe

möglich, daher müssen entweder sowohl `ksize[1]` als auch `ksize[2]` 1 betragen, oder `ksize[3]` muss auf 1 gesetzt werden.

Einen *Average-Pooling Layer* können Sie mit der Funktion `avg_pool()` anstelle von `max_pool()` erstellen.

Nun kennen Sie alle Bauelemente zum Erstellen eines Convolutional Neural Network. Sehen wir uns nun an, wie sich diese zusammensetzen lassen.

Architekturen von CNNs

In typischen CNN-Architekturen sind einige Convolutional Layers aufeinanderge-stapelt (normalerweise folgt auf jeden eine ReLU-Schicht), anschließend ein Pooling Layer, dann einige weitere Convolutional Layers (+ReLU), dann noch ein Pooling Layer und so weiter. Das Bild wird beim Durchlaufen des Netzes immer kleiner, aber dank der Convolutional Layers normalerweise auch immer tiefer (d.h. mit mehr Feature Maps, siehe Abbildung 13-9). Am oberen Ende des Stapels befindet sich ein normales Feed-Forward-Netz aus einigen vollständig verbundenen Schichten (+ReLUs), und eine abschließende Schicht gibt die Vorhersage aus (z.B. eine Softmax-Schicht, die geschätzte Wahrscheinlichkeiten für Kategorien ausgibt).

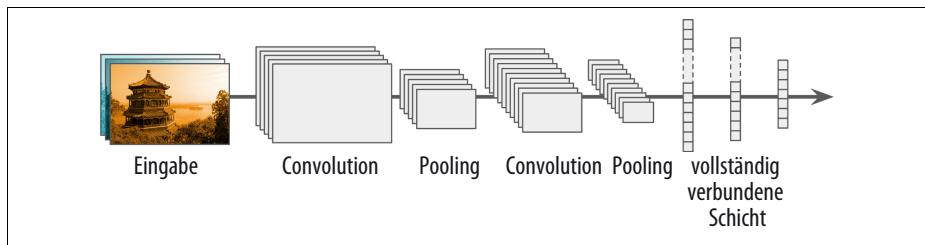


Abbildung 13-9: Typische Architektur eines CNN



Ein verbreiteter Fehler ist, zu große Convolution Kernels zu verwenden. Sie können oft mit zwei aufeinanderfolgenden Kernels der Größe 3×3 bei geringerem Rechenaufwand den gleichen Effekt wie mit einem Kernel der Größe 9×9 erzielen.

Über die Jahre wurden Varianten dieser Architektur entwickelt, die zu faszinierenden Fortschritten führten. Ein gutes Maß für den Fortschritt ist die Fehlerquote in Wettbewerben wie der ILSVRC ImageNet Challenge (<http://image-net.org/>). In diesem Wettbewerb ist die Top-5-Fehlerquote bei der Bildklassifikation in nur fünf Jahren von über 26% auf etwas über 3% gefallen. Die Top-5-Fehlerquote ist die Anzahl der Testbilder, bei denen die besten 5 Vorhersagen des Systems nicht die korrekte Lösung enthielten. Die Bilder sind groß (256 Pixel hoch), und es gibt 1000 Kategorien, von denen einige wirklich subtil sind (versuchen Sie einmal, 120 Hunderassen auseinanderzuhalten). Die Funktionsweise von CNNs lässt sich gut verstehen, indem man sich die Entwicklung der Gewinner ansieht.

Wir werden zuerst die klassische LeNet-5-Architektur (1998) untersuchen und anschließend drei Gewinner der ILSVRC Challenge: AlexNet (2012), GoogLeNet (2014) und ResNet (2015).

Andere visuelle Aufgaben

Es gab auch bei anderen visuellen Aufgaben wie der Objekterkennung, Lokalisierung und Bildsegmentierung bemerkenswerte Fortschritte. Bei der Objekterkennung und Lokalisierung gibt das neuronale Netz eine Folge von Rahmen um unterschiedliche Objekte im Bild aus. Gute Beispiele sind ein Artikel (<https://goo.gl/ZKuDt>) von Maxine Oquab et al. aus dem Jahr 2015, der für die Kategorie jedes Objekts eine Heatmap ausgibt, und ein Artikel (<https://goo.gl/upuHl2>) von Russell Stewart et al. aus dem Jahr 2015, der CNNs mit rekurrenten neuronalen Netzen kombiniert, um Gesichter zu erkennen und eine Folge von Rahmen um sie herum auszugeben. Bei der Bildsegmentierung gibt das Netz ein Bild aus (meist mit der gleichen Größe wie die Eingabe), wobei jedes Pixel die Kategorie des Objekts des entsprechenden Eingabepixels codiert. Ein Beispiel hierfür finden Sie im Artikel (<https://goo.gl/7ReZql>) von Evan Shelhamer et al. aus dem Jahr 2016.

LeNet-5

Die LeNet-5-Architektur ist vermutlich die am besten bekannte CNN-Architektur. Wie oben erwähnt, wurde sie im Jahr 1998 von Yann LeCun im Jahr 1998 erstellt und im großen Stil zur Erkennung handgeschriebener Ziffern (MNIST) eingesetzt. Sie besteht aus den in Tabelle aufgeführten Schichten.

Tabelle 13-1: LeNet-5-Architektur

Schicht	Typ	Maps	Größe	Kernel-Größe	Schrittweite	Aktivierung
Out	vollst. verbunden	–	10	–	–	RBF
F6	vollst. verbunden	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg-Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg-Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Eingabe	1	32 × 32	–	–	–

Einige Details sind hierbei anzumerken:

- Die MNIST-Bilder sind 28×28 Pixel groß, werden aber durch Zero Padding auf 32×32 Pixel vergrößert und vor dem Eingeben ins Netz normalisiert. Im übrigen Netz wird kein Padding verwendet, weswegen die Größe des Bilds beim Durchlaufen des Netzes immer weiter schrumpft.

- Die Average-Pooling Layers sind etwas komplexer als gewöhnlich: Jedes Neuron berechnet den Mittelwert seiner Eingaben und multipliziert das Ergebnis mit einem erlernbaren Koeffizienten (einem pro Map) und fügt einen erlernbaren Bias-Term hinzu (wieder einer pro Map). Schließlich wird die Aktivierungsfunktion angewendet.
- Die meisten Neuronen in den Maps von C3 sind mit den Neuronen in nur drei oder vier Maps in S2 verbunden (anstatt mit allen sechs Maps in S2). Eine Erläuterung finden Sie in Tabelle 1 im Originalartikel.
- Die Ausgabeschicht ist ein wenig besonders: Anstatt das Skalarprodukt der Eingaben und des Gewichtsvektors zu berechnen, gibt jedes Neuron den quadrierten euklidischen Abstand zwischen seinem Eingabevektor und seinem Gewichtsvektor aus. Jede Ausgabe misst, wie stark ein Bild einer bestimmten Ziffernkategorie angehört. Inzwischen bevorzugt man die Kreuzentropie als Kostenfunktion, da sie schlechte Vorhersagen weitaus stärker abstrahrt, große Gradienten hervorbringt und dadurch schneller konvergiert.

Die Webseite (<http://yann.lecun.com/>) von Yann LeCun (im Abschnitt »LENET«) enthält großartige Demos, in denen LeNet-5 Ziffern klassifiziert.

AlexNet

Die CNN-Architektur *AlexNet* (<http://goo.gl/mWRBRp>)⁹ gewann im Jahr 2012 die ImageNet ILSVRC Challenge mit großem Abstand: Sie erreichte eine Top-5-Fehlerquote von 17%, der Zweitplatzierte nur 26%! Sie wurde von Alex Krizhevsky (daher der Name), Ilya Sutskever und Geoffrey Hinton entwickelt. Sie ist zu LeNet-5 recht ähnlich, nur viel größer und tiefer, und war das erste Netz, bei dem Convolutional Layers direkt aufeinandergestapelt wurden, anstatt auf jeden Convolutional Layer einen Pooling Layer zu platzieren. Tabelle stellt diese Architektur vor.

Tabelle 13-2: AlexNet-Architektur

Schicht	Typ	Maps	Größe	Kernel-Größe	Schrittweite	Padding	Aktivierung
Out	vollst. verbunden	–	1000	–	–	–	Softmax
F9	vollst. verbunden	–	4096	–	–	–	ReLU
F8	vollst. verbunden	–	4096	–	–	–	ReLU
C7	Convolution	256	13×13	3×3	1	SAME	ReLU
C6	Convolution	384	13×13	3×3	1	SAME	ReLU
C5	Convolution	384	13×13	3×3	1	SAME	ReLU
S4	Max-Pooling	256	13×13	3×3	2	VALID	–
C3	Convolution	256	27×27	5×5	1	SAME	ReLU
S2	Max-Pooling	96	27×27	3×3	2	VALID	–

⁹ »ImageNet Classification with Deep Convolutional Neural Networks«, A. Krizhevsky et al. (2012).

Tabelle 13-2: AlexNet-Architektur (Fortsetzung)

Schicht	Typ	Maps	Größe	Kernel-Größe	Schrittweite	Padding	Aktivierung
C1	Convolution	96	55×55	11×11	4	SAME	ReLU
In	Input	3 (RGB)	224×224	–	–	–	–

Um Overfitting zu vermeiden, verwendeten die Autoren zwei in den vorigen Kapiteln besprochene Regularisierungstechniken: Erstens wurde beim Trainieren auf Ausgaben der Schichten F8 und F9 Drop-out angewandt (mit einer Drop-out-Rate von 50%). Zweitens wurde Data Augmentation verwendet, indem die Trainingsbilder zufällig um unterschiedliche Beträge verschoben, horizontal gespiegelt und die Lichtverhältnisse verändert wurden.

AlexNet führt unmittelbar nach dem ReLU-Schritt der Schichten C1 und C3 auch einen kompetitiven Normalisierungsschritt durch, die *Local Response Normalization*. Bei dieser Art Normalisierung inhibieren die aktivsten Neuronen die an der gleichen Stelle in benachbarten Feature Maps liegenden Neuronen (solche konkurrierenden Aktivierungen sind in biologischen Neuronen beobachtet worden). Dies befördert die Spezialisierung von Feature Maps, sodass diese sich stärker unterscheiden und eine größere Bandbreite von Merkmalen abdecken. Dies verbessert im Endeffekt die Verallgemeinerungsfähigkeit. Formel 13-2 zeigt, wie LRN angewendet wird.

Formel 13-2: Local Response Normalization

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{tief}}}^{j_{\text{hoch}}} a_j^2 \right)^{-\beta} \quad \text{mit} \quad \begin{cases} j_{\text{hoch}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{tief}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

- b_i ist die normalisierte Ausgabe des Neurons in Feature Map i in Reihe u und Spalte v (in dieser Gleichung berücksichtigen wir nur Neuronen in dieser Reihe und Spalte, daher wurden u und v nicht ausgeschrieben).
- a_i ist die Aktivierung dieses Neurons nach dem ReLU-Schritt, aber vor der Normalisierung.
- k, α, β und r sind Hyperparameter. k bezeichnet man als *Bias*, r als den *Tiefenradius*.
- f_n ist die Anzahl der Feature Maps.

Beispielsweise inhibiert mit $r = 2$ ein stark aktiviertes Neuron die Aktivierung der Neuronen in den Feature Maps unmittelbar über und unter der eigenen.

In AlexNet sind die Hyperparameter wie folgt eingestellt: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$ und $k = 1$. Dieser Schritt lässt sich mit der TensorFlow-Operation `tf.nn.local_response_normalization()` implementieren.

Eine Variante von AlexNet namens *ZF Net* wurde von Matthew Zeiler und Rob Fergus entwickelt. Sie gewann die ILSVRC Challenge im Jahr 2013. Sie ist im Wesentlichen ein AlexNet mit einigen geänderten Hyperparametern (Anzahl der Feature Maps, Größe der Kerne, Schrittweite und so weiter).

GoogLeNet

Die *GoogLeNet-Architektur* (<http://goo.gl/tCFzVs>) wurde von Christian Szegedy et al. aus dem Google-Research-Team entwickelt¹⁰ und gewann die ILSVRC Challenge im Jahr 2014, wobei die Top-5-Fehlerquote unter 7% fiel. Diese großartige Leistung kam vor allem dadurch zustande, dass das Netz viel tiefer als vorige CNNs war (siehe Abbildung 13-11). Dies wurde durch Sub-Netze ermöglicht, die *Inception-Module*,¹¹ mit denen GoogLeNet die Parameter deutlich effizienter als frühere Architekturen nutzen kann: GoogLeNet enthält 10 Mal weniger Parameter als AlexNet (etwa 6 Millionen anstatt 60 Millionen).

Abbildung 13-10 zeigt die Architektur eines Inception-Moduls. Die Notation » $3 \times 3 + 2(S)$ « bedeutet, dass die Schicht einen Kernel der Größe 3×3 , Schrittweite 2 und SAME-Padding verwendet. Das Eingabesignal wird zuerst kopiert und dann in vier unterschiedliche Schichten eingegeben. Alle Convolutional Layers verwenden die ReLU-Aktivierungsfunktion. Die zweite Gruppe Convolutional Layers verwendet andere Kernel-Größen (1×1 , 3×3 und 5×5), wodurch Muster in unterschiedlichen Größenordnungen erfasst werden.

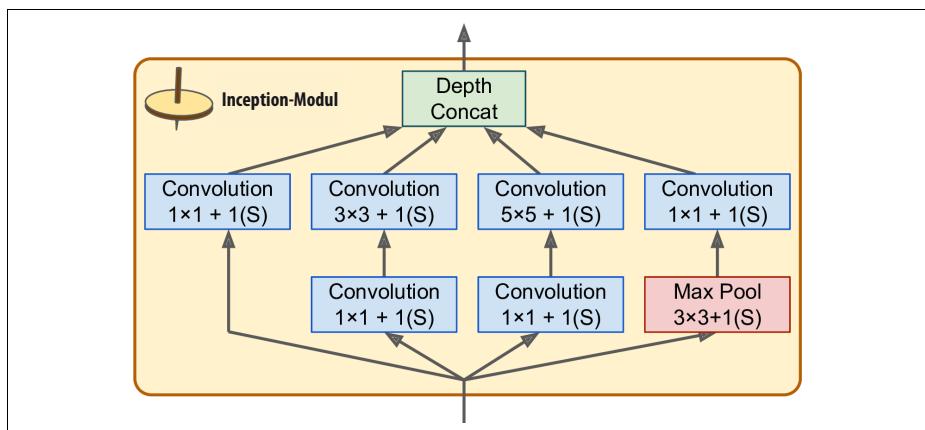


Abbildung 13-10: Inception-Modul

Jede einzelne Schicht verwendet Schrittweite 1 und SAME-Padding (sogar der Max-Pooling Layer), sodass alle Ausgaben die gleiche Höhe und Breite wie ihre

10 »Going Deeper with Convolutions«, C. Szegedy et al. (2015).

11 Im Film *Inception* aus dem Jahr 2010 reisen die Akteure durch mehrere Traumebenen immer tiefer. Daher stammt der Name dieser Module.

Eingaben haben. Dadurch lassen sich alle Ausgaben im letzten *Depth Concat Layer* entlang der dritten Dimension verketten (d.h., die Feature Maps aller vier Convolutional Layers werden aufeinandergestapelt). Diese Schicht lässt sich in TensorFlow mit der Operation zum Verketten `tf.concat()` mit `axis=3` implementieren (Achse 3 ist die Tiefe).

Sie fragen sich vielleicht, warum die Inception-Module Convolutional Layers mit Kernels der Größe 1×1 enthalten. Können diese Schichten überhaupt irgendwelche Merkmale erfassen, wenn sie nur genau ein Pixel betrachten? Diese Schichten sind für zwei Dinge gut:

- Erstens sind sie so konfiguriert, dass sie viel weniger Feature Maps ausgeben als ihre Eingaben enthalten. Diese Schichten dienen als *Flaschenhals*, sie dienen der Dimensionsreduktion. Dies ist besonders vor den Convolutions der Größen 3×3 und 5×5 nützlich, da diese besonders rechenintensiv sind.
- Zweitens verhält sich jedes Paar von Convolutional Layers ($[1 \times 1, 3 \times 3]$ und $[1 \times 1, 5 \times 5]$) wie ein einzelner, mächtiger Convolutional Layer, der zum Erfassen komplexerer Muster in der Lage ist. Anstatt mit einem einzelnen linearen Klassifikator über das Bild zu fahren (wie bei einem einzelnen Convolutional Layer), fährt ein solches Paar von Convolutional Layers mit einem zweischichtigen neuronalen Netz über das Bild.

Kurz, Sie können sich das gesamte Inception-Modul als einen Convolutional Layer auf Steroiden vorstellen, der Feature Maps mit komplexen Mustern in verschiedenen Größenordnungen ausgibt.



Die Anzahl der Convolution-Kernels ist in jedem Convolutional Layer ein Hyperparameter. Leider bedeutet das, dass jeder hinzugefügte Inception-Layer sechs zusätzliche Hyperparameter mit sich bringt.

Betrachten wir nun die Architektur des GoogLeNet-CNN (siehe Abbildung 13-11). Sie ist so tief, dass wir es in drei Spalten abbilden mussten, denn GoogLeNet ist einfach ein hoher Stapel mit neun Inception-Modulen (die Kisten mit Kreiseln), jedes mit je drei Schichten. Die Anzahl der von jedem Convolutional Layer und jedem Pooling Layer ausgegebenen Feature Maps ist vor der Größe des Kernels angegeben.

Die sechs Zahlen in den Inception-Modulen stehen für die Anzahl der von jedem Convolutional Layer im Modul ausgegebenen Feature Maps (in der gleichen Reihenfolge wie in Abbildung 13-10). Sämtliche Convolutional Layers verwenden die ReLU-Aktivierungsfunktion.

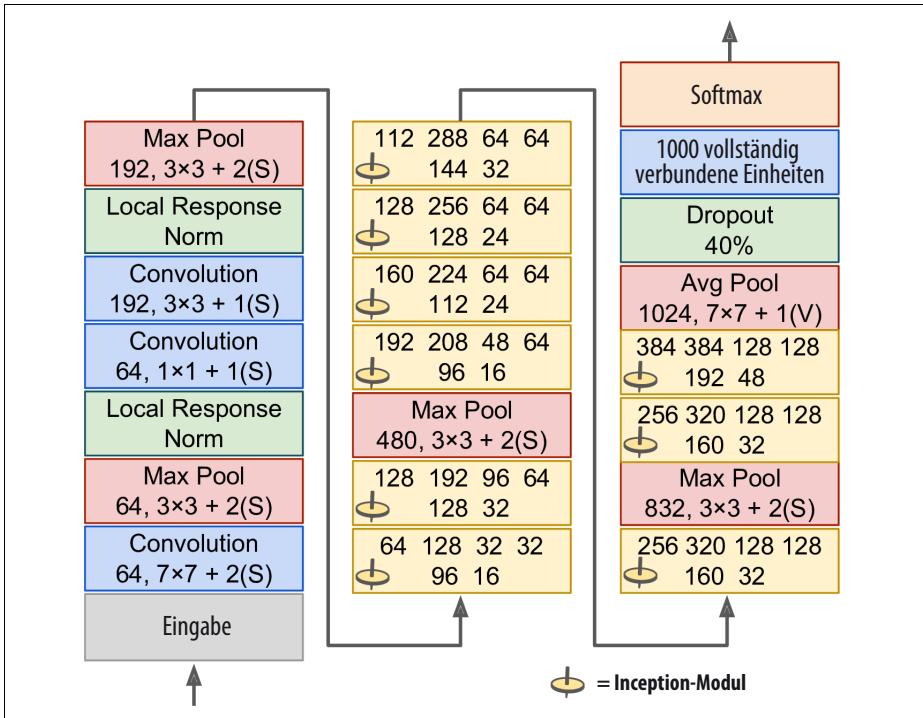


Abbildung 13-11: GoogLeNet-Architektur

Gehen wir dieses Netz einmal durch:

- Die ersten zwei Schichten vierteln die Höhe und Breite des Bilds (damit wird die Fläche durch 16 geteilt), um den Rechenaufwand zu reduzieren.
- Anschließend stellt der Local Response Normalization Layer sicher, dass die vorigen Schichten eine große Bandbreite an Merkmalen erlernen (wie oben besprochen).
- Es folgen zwei Convolutional Layers, wobei die erste Schicht als *Flaschenhals* dient. Wie bereits erläutert, können Sie sich dieses Paar als einzelnen, schlauen Convolutional Layer veranschaulichen.
- Ein weiterer Local Response Normalization Layer stellt sicher, dass die vorigen Schichten eine große Bandbreite an Mustern erfassen.
- Anschließend halbiert ein Max-Pooling Layer die Höhe und Breite des Bilds, wieder, um Rechenzeit zu sparen.
- Es folgt der große Stapel aus neun Inception-Modulen, unterbrochen von einigen Max-Pooling Layers zur Dimensionsreduktion und Steigerung der Geschwindigkeit.
- Der danach folgende Average-Pooling Layer verwendet einen Kernel der Größe der Feature Maps mit VALID-Padding und gibt Feature Maps der

Größe 1×1 aus: Diese überraschende Strategie nennt man *Global Average Pooling*. Sie zwingt im Endeffekt die vorigen Schichten dazu, Feature Maps zu produzieren, die Konfidenzwerte für jede Zielkategorie enthalten (andere Arten von Merkmalen würden durch die Mittelwertbildung zerstört). Dadurch ist es unnötig, mehrere vollständig verbundene Schichten am oberen Ende des CNN zu haben (wie bei AlexNet), was die Anzahl der Parameter im Netz und die Gefahr von Overfitting erheblich senkt.

- Die letzten Schichten sind selbsterklärend: Drop-out dient der Regularisierung, die vollständig verbundene Schicht mit Softmax-Aktivierungsfunktion gibt die geschätzten Wahrscheinlichkeiten für die einzelnen Kategorien aus.

Diese Darstellung ist leicht vereinfacht: Die ursprüngliche GoogLeNet-Architektur enthielt noch zwei Hilfsklassifikatoren, die auf dem dritten und sechsten Inception-Modul aufsetzten. Beide bestanden aus einem Average-Pooling Layer, einem Convolutional Layer, zwei vollständig verbundenen Schichten und einer Softmax-aktivierten Schicht. Beim Trainieren wurde deren Verlustfunktion (um 70 % herunterskaliert) zur gesamten Verlustfunktion addiert. Die Absicht war, das Problem der schwindenden Gradienten zu bekämpfen und das Netz zu regularisieren. Es konnte aber gezeigt werden, dass sie nur geringe Auswirkungen hatten.

ResNet

Zu guter Letzt stellen wir den Gewinner der ILSVRC Challenge von 2015 vor, das von Kaiming He et al.,¹² entwickelte *Residual Network* (<http://goo.gl/4puHU5>) (oder *ResNet*). Es erzielte mithilfe eines extrem tiefen CNN mit 152 Schichten eine erstaunliche Top-5-Fehlerquote unter 3.6 %. Der Schlüssel zum Trainieren eines derart tiefen Netzes sind *Skip-Verbindungen* (auch *Shortcut-Verbindungen* genannt): Das in eine Schicht eingegebene Signal wird auch an eine im Stapel etwas höher gelegene Schicht gereicht. Schauen wir uns an, warum dies zielführend ist.

Beim Trainieren eines neuronalen Netzes geht es darum, eine Zielfunktion $h(\mathbf{x})$ zu modellieren. Wenn Sie die Eingabe \mathbf{x} zur Ausgabe des Netzes hinzufügen (d.h. eine Skip-Verbindung erzeugen), wird das Netz gezwungen, $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ anstelle von $h(\mathbf{x})$ zu modellieren. Dies nennt man *Residual Learning* (siehe Abbildung 13-12).

Wenn Sie ein gewöhnliches neuronales Netz initialisieren, liegen dessen Gewichte um null, sodass das Netz Werte um null ausgibt. Wenn Sie eine Skip-Verbindung hinzufügen, gibt das entstehende Netz einfach eine Kopie der Eingabe aus; anders ausgedrückt, modelliert es zunächst die Identitätsfunktion. Wenn die Zielfunktion in der Nähe der Identitätsfunktion liegt (was oft der Fall ist), beschleunigt sich das Trainieren erheblich.

12 »Deep Residual Learning for Image Recognition«, K. He (2015).

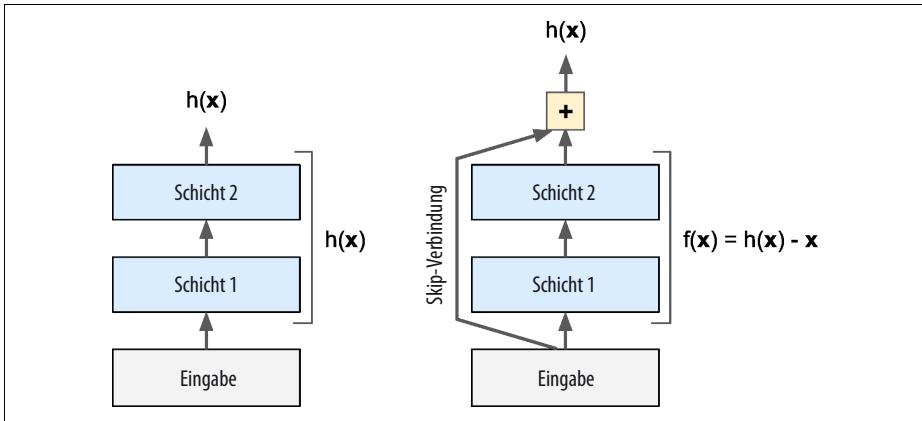


Abbildung 13-12: Residual Learning

Wenn Sie außerdem viele Skip-Verbindungen hinzufügen, kann das Netz auch dann Fortschritte erzielen, wenn mehrere Schritte noch nicht mit dem Lernen begonnen haben (siehe Abbildung 13-13). Dank der Skip-Verbindungen kann das Signal das gesamte Netz leicht durchlaufen. Das Deep-Residual-Netz lässt sich als Stapel von *Residual Units* ansehen, wobei jede Residual Unit ein kleines neuronales Netz mit einer Skip-Verbindung ist.

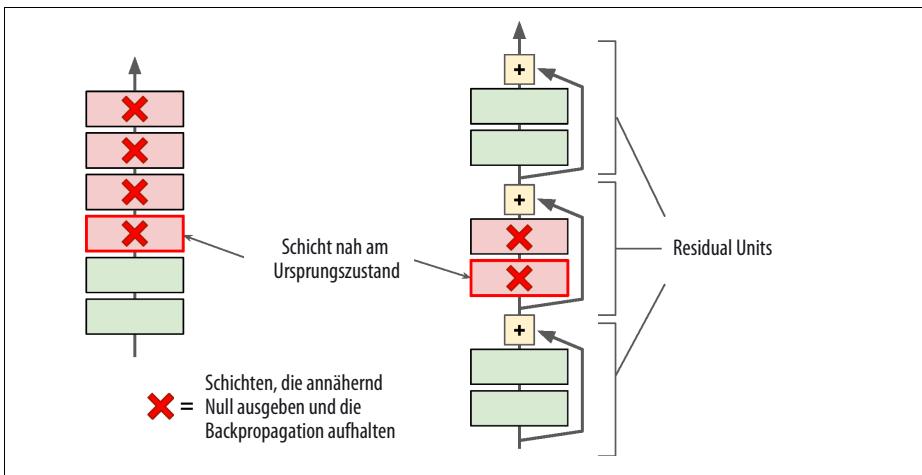


Abbildung 13-13: Gewöhnliches Deep-Learning-Netz (links) und Deep-Residual-Netz (rechts)

Betrachten wir nun die Architektur von ResNet (siehe Abbildung 13-14). Sie ist eigentlich überraschend einfach. Sie beginnt und endet genauso wie GoogLeNet (nur die Drop-out-Schicht fehlt), und in der Mitte befindet sich ein sehr tiefer Stapel einfacher Residual Units. Jede Residual Unit besteht aus zwei Convolutional Layers mit Batch-Normalisierung (BN), der ReLU-Aktivierungsfunktion, Kernels

der Größe 3×3 und Erhalt der räumlichen Dimensionen (Schrittweite 1, SAME-Padding).

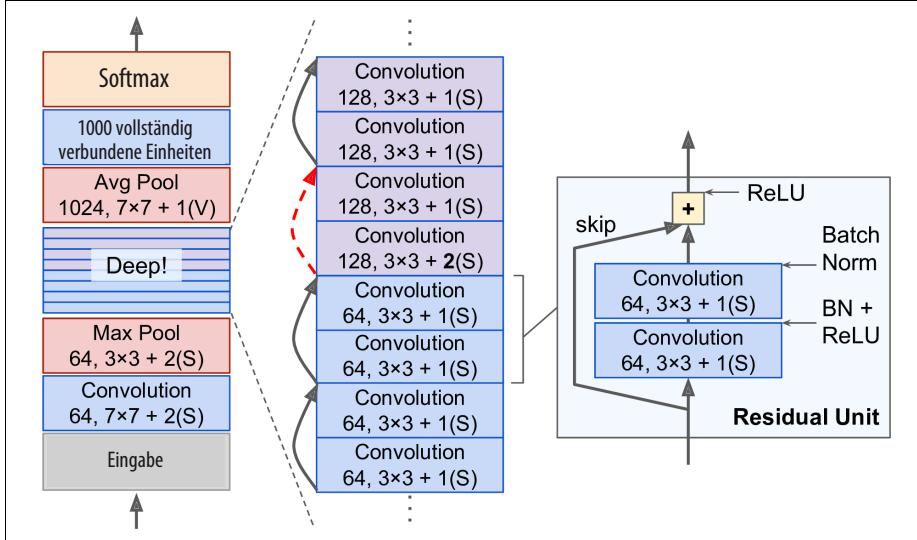


Abbildung 13-14: ResNet-Architektur

Beachten Sie, dass die Anzahl der Feature Maps sich jeweils nach einigen Residual Units verdoppelt, während sich gleichzeitig deren Höhe und Breite halbiert (durch einen Convolutional Layer mit Schrittweite 2). An dieser Stelle lässt sich die Eingabe nicht direkt zu den Ausgaben der Residual Unit addieren, da beide nicht die gleichen Abmessungen haben (dieses Problem betrifft beispielsweise die durch einen gestrichelten Pfeil gekennzeichnete Skip-Verbindung Abbildung 13-14). Um dieses Problem zu lösen, passieren die Eingaben einen Convolutional Layer der Größe 1×1 mit Schrittweite 2 und der richtigen Anzahl ausgegebener Feature Maps (siehe Abbildung 13-15).

ResNet-34 ist ein ResNet mit 34 Schichten (nur die Convolutional Layers und die vollständig verbundene Schicht werden gezählt). Es enthält drei Residual Units, die 64 Feature Maps ausgeben, 4 RUs, die 128 Maps ausgeben, 6 RUs mit 256 Maps sowie 3 RUs mit 512 Maps.

Tiefere ResNets wie ResNet-152 verwenden etwas unterschiedliche Residual Units. Anstelle von zwei Convolutional Layers der Größe 3×3 mit (sagen wir) 256 Feature Maps verwenden sie drei Convolutional Layers: Zuerst einen Convolutional Layer der Größe 1×1 mit nur 64 Feature Maps (4 Mal weniger), der als Flaschenhals fungiert (wie bereits besprochen), anschließende eine Schicht der Größe 3×3 mit 64 Feature Maps und schließlich einen weiteren Convolutional Layer der Größe 1×1 mit 256 Feature Maps (4 Mal 64), der die ursprüngliche Tiefe wiederherstellt. Res-

Net-152 enthält 3 solche RUs, die 256 maps ausgeben, anschließend 8 RUs mit 512 Maps, stolze 36 RUs mit 1024 Maps und schließlich 3 RUs mit 2048 Maps.

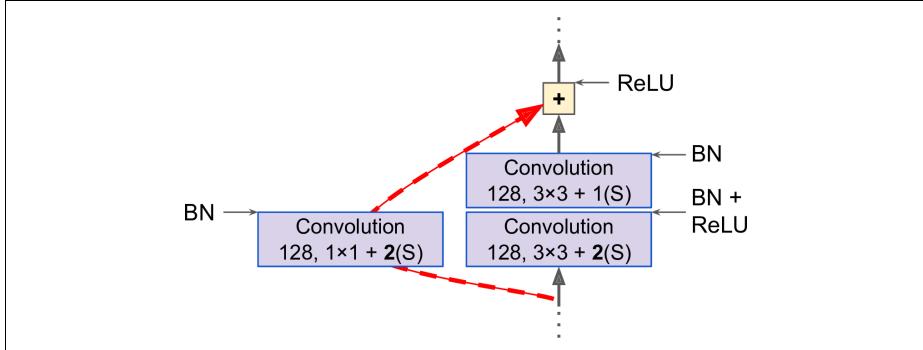


Abbildung 13-15: Skip-Verbindung beim Ändern der Größe und Tiefe von Feature Maps

Wie Sie sehen, entwickelt sich dieses Forschungsgebiet sehr schnell. Jedes Jahr treten alle möglichen Architekturen in Erscheinung. Ein deutlicher Trend ist, dass CNNs immer tiefer werden. Sie werden auch leichtgewichtiger und erfordern immer weniger Parameter. Im Moment ist die ResNet-Architektur sowohl die mächtigste als auch die vermutlich einfachste. Sie sollten daher vorerst diese verwenden. Behalten Sie aber die Ergebnisse der jährlichen ILSVRC Challenge im Auge. Der Gewinner im Jahr 2016 war das Trimpf-Soushen-Team aus China mit einer erstaunlichen Fehlerquote von 2.99%. Dazu hat das Team Kombinationen der vorigen Modelle trainiert und zu einem Ensemble vereinigt. Je nach Aufgabe kann die reduzierte Fehlerquote die zusätzliche Komplexität wert sein oder auch nicht.

Es gibt noch andere beachtenswerte Architekturen, darunter *VGGNet* (<http://goo.gl/QcMjXQ>)¹³, Zweitplatzierter in der LSVRC Challenge im Jahr 2014, und *Inception-v4* (<http://goo.gl/Ak2vBp>)¹⁴, das die Konzeptionen von GoogLeNet und ResNet miteinander verbindet und eine Top-5-Fehlerquote von etwa 3% bei der Image-Net-Klassifikation erzielt.



An der Implementierung der soeben besprochenen CNN-Architekturen ist nichts Besonderes dran. Wir haben oben gesehen, wie sich die einzelnen Bausteine erstellen lassen. Sie müssen diese nur noch zur gewünschten Architektur zusammensetzen. In den folgenden Übungen werden wir ein vollständiges CNN erstellen, und den funktionierenden Code dazu finden Sie in den Jupyter Notebooks.

¹³ »Very Deep Convolutional Networks for Large-Scale Image Recognition«, K. Simonyan and A. Zisserman (2015).

¹⁴ »Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning«, C. Szegedy et al. (2016).

TensorFlow-Convolution-Operationen

TensorFlow enthält noch einige andere Arten von Convolutional Layers:

- `tf.layers.conv1d()` erstellt einen Convolutional Layer für 1-D-Eingaben. Dies ist beispielsweise bei der Sprachverarbeitung nützlich, bei der ein Satz als eindimensionale Folge von Wörtern abgelegt ist und das Wahrnehmungsfeld einige benachbarte Wörter berücksichtigt.
- `tf.layers.conv3d()` erstellt einen Convolutional Layer für 3-D-Eingaben, wie etwa einen 3-D PET-Scan.
- `tf.nn.atrous_conv2d()` erstellt einen *atrous Convolutional Layer* (»à trous« ist das französische Wort für »löchrig«). Dies entspricht einem gewöhnlichen Convolutional Layer mit einem Filter, der Zeilen und Spalten durch Nullen erweitert (die Löcher). Beispielsweise könnte ein Filter der Größe 1×3 mit dem Inhalt $[[1, 2, 3]]$ über eine *Dilation Rate* von 4 erweitert werden, sodass der Filter $[[1, 0, 0, 0, 2, 0, 0, 0, 3]]$ entsteht. Dadurch erhält der Convolutional Layer ein größeres Wahrnehmungsfeld, ohne dass zusätzlicher Berechnungsaufwand oder zusätzliche Parameter entstehen.
- `tf.layers.conv2d_transpose()` erstellt einen *transpose Convolutional Layer*, der auch *deConvolutional Layer* genannt wird.¹⁵ Dieser führt ein *Upsampling* eines Bilds durch, indem Nullen zwischen den Eingaben eingefügt werden, sodass Sie sich dies als gewöhnlichen Convolutional Layer mit einem Bruch als Schrittweite vorstellen können. Upsampling ist beispielsweise bei der Bildsegmentierung nützlich: In einem typischen CNN werden die Feature Maps beim Durchlaufen des Netzes immer kleiner. Wenn Sie also ein Bild mit der gleichen Größe wie die Eingabe ausgeben möchten, müssen Sie eine Upsampling-Schicht verwenden.
- `tf.nn.depthwise_conv2d()` erstellte einen *depthwise Convolutional Layer*, der jeden Filter auf jeden Eingabekanal einzeln anwendet. Wenn es also f_n Filter und $f_{n'}$ Eingabekanäle gibt, werden $f_n \times f_{n'}$ Feature Maps ausgegeben.
- `tf.layers.separable_conv2d()` erstellt einen *separable Convolutional Layer*, der sich zunächst wie ein depthwise Convolutional Layer verhält, dann einen Convolutional Layer der Größe 1×1 auf die entstehenden Feature Maps anwendet. Dadurch können Sie Filter auf beliebige Kombinationen von Eingabekanälen anwenden.

Übungen

1. Was sind die Vorteile eines CNN gegenüber einem vollständig verbundenen DNN zur Bildklassifizierung?

¹⁵ Dieser Name ist irreführend, da die Schicht *keine* Dekonvolution, eine genau definierte mathematische Operation durchführt (das Gegenteil einer Konvolution).

2. Betrachten Sie ein aus drei Convolutional Layers zusammengesetztes CNN, wobei jeder Kernels der Größe 3×3 , eine Schrittweite von 2 und SAME-Padding besitzt. Die niedrigste Schicht gibt 100 Feature Maps aus, die mittlere 200 und die oberste 400. Die Eingaben sind RGB-Bilder mit 200×300 Pixeln. Wie viele Parameter hat das CNN insgesamt? Wenn wir Floats mit 32 Bit verwenden, wie viel RAM wird dieses Netz mindestens bei der Vorhersage aus einem einzelnen Datenpunkt beanspruchen? Wie sieht es beim Trainieren mit einem Mini-Batch aus 50 Bildern aus?
3. Zählen Sie fünf Dinge auf, die Sie ausprobieren können, wenn Ihre GPU beim Trainieren eines CNN zu wenig Speicher hat.
4. Warum sollten Sie eher einen Max-Pooling Layer anstelle eines Convolutional Layer mit der gleichen Schrittweite hinzufügen?
5. In welcher Situation sollten Sie einen *Local Response Normalization* Layer hinzufügen?
6. Welches sind die wichtigsten Innovationen bei AlexNet im Vergleich zu LeNet-5? Welche Innovationen liegen GoogLeNet und ResNet zugrunde?
7. Erstellen Sie Ihr eigenes CNN und versuchen Sie, die höchstmögliche Genauigkeit auf dem MNIST-Datensatz zu erzielen.
8. Klassifizieren großer Bilder mit Inception v3.
 - a. Laden Sie einige Bilder von verschiedenen Tieren herunter. Laden Sie diese in Python, beispielsweise mit einer der Funktionen `matplotlib.image.imread()` oder `scipy.misc.imread()`. Skalieren oder schneiden Sie diese auf 299×299 Pixel zurecht und stellen Sie sicher, dass es nur drei Kanäle gibt (RGB), keinen Kanal für die Transparenz. Die Bilder wurden für das Inception-Modell so verarbeitet, dass ihre Werte zwischen -1.0 und 1.0 lagen, Sie müssen also das Gleiche tun.
 - b. Laden Sie das neueste vortrainierte Inception-v3-Modell herunter: Der Checkpoint ist auf <https://goo.gl/nxSQvl> verfügbar. Die Liste der Kategorienamen finden Sie unter <https://goo.gl/brXRtZ>, aber Sie müssen zu Beginn eine »Hintergrundkategorie« einfügen.
 - c. Erstellen Sie das Inception-v3-Modell, indem Sie die Funktion `inception_v3()` wie unten gezeigt aufrufen. Dies muss innerhalb eines mit der Funktion `inception_v3_arg_scope()` erzeugten Scope geschehen. Sie müssen auch die Parameter `is_training=False` und `num_classes=1001` wie folgt setzen:

```

from tensorflow.contrib.slim.nets import inception
import tensorflow.contrib.slim as slim

X = tf.placeholder(tf.float32, shape=[None, 299, 299, 3], name="X")
with slim.arg_scope(inception.inception_v3_arg_scope()):
    logits, end_points = inception.inception_v3(
        X, num_classes=1001, is_training=False)
predictions = end_points["Predictions"]
saver = tf.train.Saver()

```

- d. Öffnen Sie eine Session und verwenden Sie den Saver, um den heruntergeladenen Checkpoint des vortrainierten Modells wiederherzustellen.
 - e. Führen Sie das Modell aus, um die vorbereiteten Bilder zu klassifizieren. Stellen Sie die ersten fünf Vorhersagen für jedes Bild mit der geschätzten Wahrscheinlichkeit für jedes Bild dar. Wie genau ist das Modell?
9. Transfer Learning zur Klassifikation großer Bilder.
- a. Erstellen Sie einen Trainingsdatensatz mit mindestens 100 Bildern pro Kategorie. Beispielsweise könnten Sie Ihre eigenen Bilder anhand des Orts (Strand, Berge, Stadt und so weiter) klassifizieren oder einen bestehenden Datensatz wie einen Blumen-Datensatz (<https://goo.gl/EgJVXZ>) oder den places-Datensatz (<http://places.csail.mit.edu/>) des MIT verwenden (erfordert eine Registrierung, und er ist gigantisch).
 - b. Schreiben Sie einen Vorverarbeitungsschritt, der das Bild auf eine Größe von 299×299 bringt und etwas zufällige Data Augmentation vornimmt.
 - c. Verwenden Sie das vortrainierte Inception-v3-Modell aus der vorigen Übung, frieren Sie sämtliche Schichten bis zum Flaschenhals ein (d.h. der letzten vor der Ausgabeschicht) und ersetzen Sie die Ausgabeschicht mit der entsprechenden Anzahl Ausgaben für Ihre Klassifikationsaufgabe (z.B. enthält der Blumen-Datensatz fünf einander ausschließende Kategorien, daher muss die Ausgabeschicht fünf Neuronen haben und die Softmax-Aktivierungsfunktion verwenden).
 - d. Unterteilen Sie Ihren Datensatz in Trainings- und Testdaten. Trainieren Sie das Modell auf dem Trainingsdatensatz und werten Sie es mit dem Testdatensatz aus.
10. Arbeiten Sie das DeepDream-Tutorial (<https://goo.gl/4b2s6g>) von TensorFlow durch. Es ist eine unterhaltsame Möglichkeit, sich mit unterschiedlichen Möglichkeiten zum Visualisieren der von einem CNN erlernten Muster vertraut zu machen. Außerdem können Sie dort mit Deep Learning Kunst erzeugen.

Lösungen zu diesen Übungen finden Sie in Anhang A.

Rekurrente neuronale Netze

Der Schlagmann trifft den Ball. Sie fangen sofort an zu rennen und antizipieren die Trajektorie des Balls. Sie verfolgen ihn und passen Ihre Bewegungen an und fangen ihn schließlich (unter tosendem Applaus). Wir sagen ständig die Zukunft vorher, sei es, dass Sie für einen Freund den Satz beenden oder den Kaffeeduft beim Frühstück im Voraus erahnen. In diesem Kapitel werden wir *rekurrente neuronale Netze (RNN)* besprechen, eine Klasse von Netzen, die die Zukunft vorhersagen kann (natürlich nur bis zu einem gewissen Punkt). Diese Netze können *Zeitreihen* wie Aktienkurse analysieren und Ihnen sagen, ob Sie kaufen oder verkaufen sollten. In Systemen zum autonomen Fahren antizipieren sie Trajektorien von Fahrzeugen und helfen dabei, Unfälle zu vermeiden. Allgemein arbeiten sie mit *Sequenzen* beliebiger Länge anstatt, wie alle bisher besprochenen Netze, mit Eingaben vorgegebener Größe. Beispielsweise verwenden sie Sätze, Dokumente oder Audiodaten als Eingabe, wodurch sie extrem nützlich für Systeme zur Sprachverarbeitung (NLP) wie automatische Übersetzung, Sprache-zu-Text oder *Meinungsanalyse* sind (z.B. Filmbewertungen zu lesen und das Urteil des Bewertenden über den Film zu extrahieren).

Da RNNs außerdem zur Antizipation fähig sind, sind sie überraschend kreativ. Sie können die wahrscheinlichsten nächsten Noten einer Melodie vorhersagen, dann zufällig eine dieser Noten auswählen und abspielen. Dann fragen Sie das Netz nach der nächsten voraussichtlichen Note, spielen sie ab und wiederholen diesen Vorgang immer wieder. Bevor Sie sich versehen, komponiert Ihr neuronales Netz eine Melodie wie die (<http://goo.gl/IxIL1V>) vom *Magenta-Project* (<https://magenta.tensorflow.org/>) von Google hervorgebrachten. In ähnlicher Weise können RNNs auch Sätze (<http://goo.gl/onkPNd>), Bildbeschriftungen (<http://goo.gl/Nwx7Kh>) und vieles mehr generieren. Die Ergebnisse haben noch nicht viel mit Shakespeare oder Mozart zu tun, aber wer weiß, was in einigen Jahren möglich sein wird?

In diesem Kapitel werden wir uns die Grundbegriffe von RNNs ansehen, die Hauptschwierigkeiten (vor allem das in Kapitel 11 besprochene Problem schwindender/explodierender Gradienten) und die verbreiteten Lösungen: LSTM- und GRU-Zellen. Dabei werden wir wie immer zeigen, wie sich RNNs mit TensorFlow implementieren lassen. Schließlich werden wir uns die Architektur eines Systems zum maschinellen Übersetzen ansehen.

Rekurrente Neuronen

Bisher haben wir uns hauptsächlich mit Feed-Forward-Netzen beschäftigt, bei denen die Aktivierung nur in einer Richtung erfolgt, nämlich von der Eingabeschicht zur Ausgabeschicht (mit Ausnahme einiger Netze in Anhang E). Ein rekurrentes neuronales Netz sieht einem Feed-Forward-Netz sehr ähnlich, außer dass es auch rückwärts gerichtete Verbindungen enthält. Das einfachste mögliche RNN besteht aus nur einem Neuron, das Eingaben erhält, eine Ausgabe produziert und diese Ausgabe wieder an sich selbst schickt, wie in Abbildung 14-1 (links) gezeigt. Zu jedem *Ausführungszeitpunkt* t (auch ein *Frame*) erhält dieses *rekurrente Neuron* die Eingaben $\mathbf{x}_{(t)}$ sowie seine eigene Ausgabe aus dem vorigen Schritt $y_{(t-1)}$. In Abbildung 14-1 (rechts) stellen wir dieses winzige Netz entlang einer Zeitachse dar. Dies nennt man *das Netz entlang der Zeitachse aufrollen*.

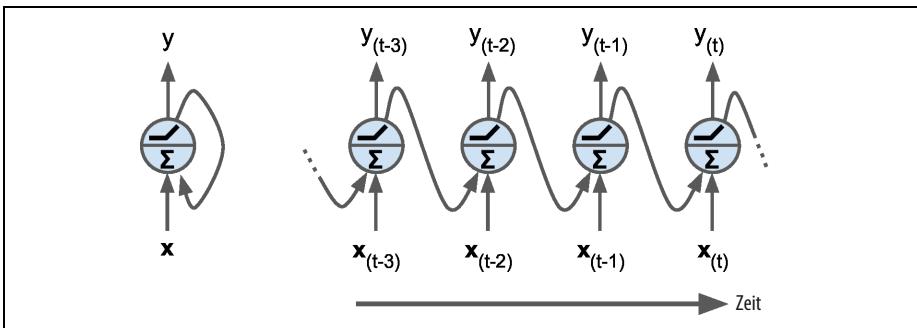


Abbildung 14-1: Ein rekurrentes Neuron (links) entlang der Zeitachse aufgerollt (rechts)

Sie können leicht eine ganze Schicht rekurrenter Neuronen erstellen. Bei jedem Schritt t erhält jedes Neuron sowohl den Eingabevektor $\mathbf{x}_{(t)}$ sowie den Ausgabevektor aus dem vorigen Schritt $\mathbf{y}_{(t-1)}$, wie in Abbildung 14-2 dargestellt. Sowohl die Eingaben als auch die Ausgaben sind nun Vektoren (bei einem einzelnen Neuron war die Ausgabe noch ein Skalar).

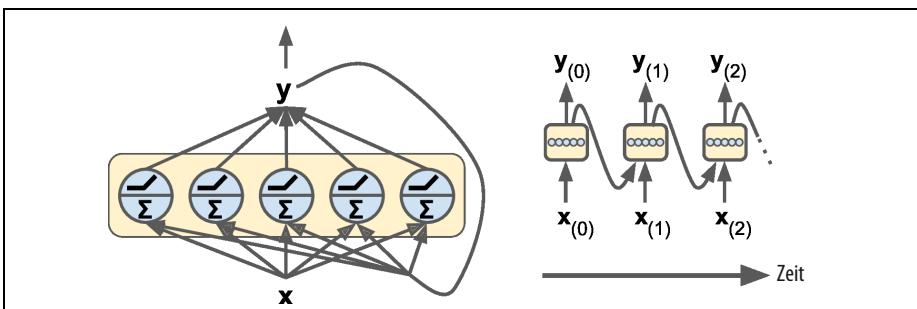


Abbildung 14-2: Eine Schicht rekurrenter Neuronen (links) entlang der Zeitachse aufgerollt (rechts)

Jedes rekurrente Neuron enthält zwei Sätze Gewichte: einen für die Eingaben $\mathbf{x}_{(t)}$, den anderen für die Ausgaben des vorangegangenen Schritts $\mathbf{y}_{(t-1)}$. Wir bezeichnen diese Gewichtsvektoren mit \mathbf{w}_x und \mathbf{w}_y . Die Ausgabe einer rekurrenten Schicht lässt sich wie in Formel 14-1 gezeigt berechnen (b ist der Bias-Term und $\phi(\cdot)$ ist die Aktivierungsfunktion, z.B. ReLU¹).

Formel 14-1: Ausgabe eines einzelnen rekurrenten Neurons für einen einzelnen Datenpunkt

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^T \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^T \cdot \mathbf{w}_y + b\right)$$

Wie bei Feed-Forward-Netzen können wir die Ausgabe einer ganzen Schicht für ein ganzes Mini-Batch auf einen Schlag berechnen, indem wir eine vektorisierte Variante der obigen Formel verwenden (siehe Formel 14-2).

Formel 14-2: Ausgabe einer Schicht rekurrenter Neuronen für alle Datenpunkte in einem Mini-Batch

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right) \\ &= \phi\left(\begin{bmatrix} \mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)} \end{bmatrix} \cdot \mathbf{W} + \mathbf{b}\right) \text{ mit } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}\end{aligned}$$

- $\mathbf{Y}_{(t)}$ ist eine Matrix der Größe $m \times n_{\text{Neuronen}}$, die die Ausgaben der Schicht im Schritt t für jeden Datenpunkt im Mini-Batch enthält (m ist die Anzahl Datenpunkte im Mini-Batch und n_{Neuronen} die Anzahl der Neuronen).
- $\mathbf{X}_{(t)}$ ist eine Matrix der Größe $m \times n_{\text{Eingaben}}$ mit den Eingabedaten aller Datenpunkte (n_{Eingaben} ist die Anzahl der eingespeisten Merkmale).
- \mathbf{W}_x ist eine Matrix der Größe $n_{\text{Eingaben}} \times n_{\text{Neuronen}}$ mit den Gewichten der Verbindungen, durch die die Eingaben im aktuellen Schritt erfolgen.
- \mathbf{W}_y ist eine Matrix der Größe $n_{\text{Neuronen}} \times n_{\text{Neuronen}}$ mit den Gewichten der Verbindungen, durch die die Ausgaben des vorigen Schritts ankommen.
- Die Gewichtsmatrizen \mathbf{W}_x und \mathbf{W}_y sind häufig zu einer einzigen Gewichtsmatrix \mathbf{W} mit den Abmessungen $(n_{\text{Eingaben}} + n_{\text{Neuronen}}) \times n_{\text{Neuronen}}$ verbunden (siehe zweite Zeile in Formel 14-2).
- \mathbf{b} ist ein Vektor der Größe n_{Neuronen} mit den Bias-Termen jedes Neurons.

Beachten Sie, dass $\mathbf{Y}_{(t)}$ eine Funktion von $\mathbf{X}_{(t)}$ und $\mathbf{Y}_{(t-1)}$ ist, das eine Funktion von $\mathbf{X}_{(t-1)}$ und $\mathbf{Y}_{(t-2)}$ ist, das eine Funktion von $\mathbf{X}_{(t-2)}$ und $\mathbf{Y}_{(t-3)}$ ist und so weiter. Damit wird $\mathbf{Y}_{(t)}$ eine Funktion aller seit dem Zeitpunkt $t = 0$ erfolgten Eingaben (also $\mathbf{X}_{(0)}$,

1 Viele Forscher bevorzugen bei RNNs den Tangens hyperbolicus (tanh) als Aktivierungsfunktion anstelle von ReLU. Lesen Sie dazu beispielsweise den Artikel »Dropout Improves Recurrent Neural Networks for Handwriting Recognition« (<https://goo.gl/2WSnaj>) von Vu Pham et al. Auf ReLU aufbauende RNNs sind aber ebenfalls möglich, wie im Artikel »A Simple Way to Initialize Recurrent Networks of Rectified Linear Units« (<https://goo.gl/NrKAP0>) von Quoc V. Le et al. gezeigt.

$\mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$). Beim ersten Schritt $t = 0$ gibt es keine vorigen Ausgaben, daher werden diese üblicherweise auf null gesetzt.

Gedächtniszellen

Da die Ausgabe eines rekurrenten Neurons zum Zeitpunkt t eine Funktion aller Eingaben der vorigen Schritte ist, können Sie auch sagen, dass es eine Art *Gedächtnis* hat. Den Teil eines neuronalen Netzes, dessen Zustand über mehrere zeitliche Schritte erhalten bleibt, bezeichnet man als *Gedächtniszelle* (oder einfach als *Zelle*). Ein einfaches rekurrentes Neuron oder eine Schicht rekurrenter Neuronen sind sehr *einfache Zellen*. Wir werden uns im Verlauf dieses Kapitels aber noch komplexere und mächtigere Zellen ansehen.

Allgemeinen ausgedrückt ist der Zustand einer Zelle zum Zeitpunkt t , geschrieben $\mathbf{h}_{(t)}$ (das » \mathbf{h} « steht für »hidden«), eine Funktion der Eingaben zu diesem Zeitpunkt und ihres Zustands zum vorigen Zeitpunkt: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Die Ausgabe zum Zeitpunkt t , geschrieben $\mathbf{y}_{(t)}$, ist ebenfalls eine Funktion des vorigen Zustands und der aktuellen Eingaben. Bei den bisher besprochenen einfachen Zellen ist die Ausgabe mit dem Zustand identisch, aber bei komplexeren Zellen ist dies nicht immer der Fall, wie Abbildung 14-3 zeigt.

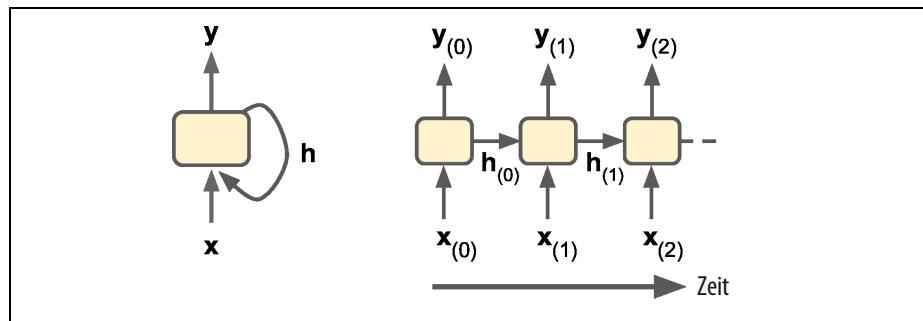


Abbildung 14-3: Der verborgene Zustand einer Zelle und ihre Ausgabe können unterschiedlich sein

Ein- und Ausgabesequenzen

Ein RNN kann gleichzeitig eine Sequenz von Eingabedaten aufnehmen und eine Sequenz von Ausgaben produzieren (siehe Abbildung 14-4, Netz oben links). Diese Art Netz ist beispielsweise zur Vorhersage von Zeitreihen wie etwa Aktienkursen geeignet: Sie geben die Preise der letzten N Tage ein, und das Netz muss die um einen Tag in die Zukunft verschobenen Preise ausgeben (d.h. anhand der $N - 1$ Tage bis morgen).

Alternativ können Sie eine Sequenz aus Eingaben in das Netz einspeisen und sämtliche Ausgaben außer der letzten ignorieren (im Netz oben rechts). Anders ausge-

drückt, bildet dieses Netz eine Sequenz auf einen Vektor ab. Sie könnten in dieses Netz nacheinander die Wörter einer Filmbewertung einspeisen, und das Netz würde als Ausgabe eine Bewertung ausgeben (z.B. von -1 [miserabel] bis $+1$ [fantastisch]).

Umgekehrt könnten Sie dem Netz im ersten Schritt eine einzelne Eingabe zur Verfügung stellen (und in allen folgenden Schritten Nullen) und das Netz eine Folge von Ausgaben generieren lassen (siehe Netz unten links). Dieses Netz wandelt einen Vektor in eine Folge um. Beispielsweise könnte die Eingabe ein Bild sein und die Ausgabe eine Beschreibung dieses Bilds.

Schließlich könnte ein Netz eine Sequenz zu einem Vektor umwandeln, genannt *Encoder*, und anschließend diesen Vektor in eine Sequenz umwandeln, genannt *Decoder* (siehe Netz unten rechts). Dies lässt sich beispielsweise zum Übersetzen eines Satzes aus einer Sprache in eine andere verwenden. Sie könnten in das Netz einen Satz in einer Sprache eingeben, der Encoder würde diesen Satz in eine vektorielle Repräsentation umwandeln. Anschließend würde der Decoder diesen Vektor in einen Satz in einer anderen Sprache überführen. Dieses zweistufige Modell namens Encoder-Decoder funktioniert viel besser als eine laufende Übersetzung mit einem Sequenz-zu-Sequenz-RNN (wie dem oben links), da das letzte Wort eines Satzes das erste Wort der Übersetzung beeinflussen kann. Sie müssen also warten, bis Sie den ganzen Satz gehört haben, bevor er übersetzt wird.

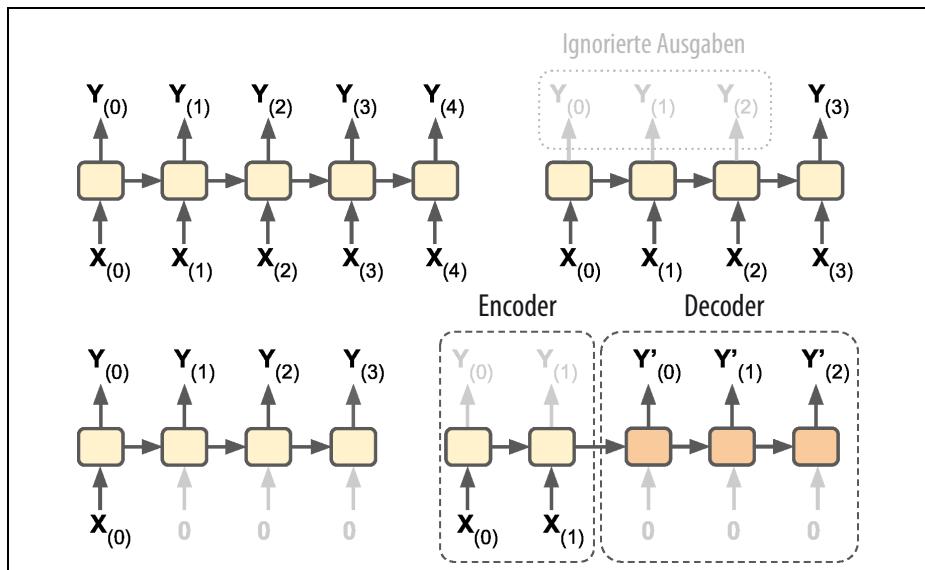


Abbildung 14-4: Sequenz zu Sequenz (oben links), Sequenz zu Vektor (oben rechts), Vektor zu Sequenz (unten links), verzögerte Sequenz zu Sequenz (unten rechts)

Dies klingt vielversprechend. Fangen wir also an zu programmieren!

Einfache RNNs in TensorFlow

Als Erstes implementieren wir ein sehr einfaches RNN-Modell, ohne die in TensorFlow eingebauten RNN-Operationen zu verwenden, um den Vorgang besser nachzuvollziehen. Wir werden ein RNN mit einer Schicht aus fünf rekurrenten Neuronen erstellen (wie das in Abbildung 14-2 dargestellte RNN) und tanh als Aktivierungsfunktion verwenden. Wir nehmen an, dass das RNN nur zwei Schritte lang läuft und in jedem Schritt Eingabevektoren der Größe 3 annimmt. Der folgende Code erstellt das über zwei zeitliche Schritte aufgerollte RNN:

```
n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

Dieses Netz erinnert stark an ein zweischichtiges Feed-Forward-Netz, enthält aber einige Besonderheiten: Erstens verwenden beide Schichten die gleichen Gewichte und Bias-Terme. Zweitens kommen in jeder Schicht neue Eingaben hinzu, und wir erhalten in jeder Schicht eine Ausgabe. Um das Modell auszuführen, müssen wir die Eingaben in beiden Schritten einspeisen:

```
import numpy as np

# Mini-Batch:      Punkt 0,   Punkt 1,   Punkt 2,   Punkt 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

Dieser Mini-Batch enthält vier Datenpunkte, jeder mit einer Sequenz aus zwei Eingaben. Am Ende enthalten `Y0_val` und `Y1_val` die Ausgaben des Netzes aus beiden Schritten für alle Neuronen und alle Datenpunkte im Mini-Batch:

```
>>> print(Y0_val) # Ausgabe bei t = 0
[[ -0.0664006  0.96257669  0.68105787  0.70918542 -0.89821595] # Punkt 0
 [ 0.9977755 -0.71978885 -0.99657625  0.9673925 -0.99989718] # Punkt 1
 [ 0.99999774 -0.99898815 -0.99999893  0.99677622 -0.99999988] # Punkt 2
 [ 1.          -1.          -1.          -0.99818915  0.99950868]] # Punkt 3
>>> print(Y1_val) # Ausgabe bei t = 1
[[ 1.          -1.          -1.          0.40200216 -1.          ] # Punkt 0
 [-0.12210433  0.62805319  0.96718419 -0.99371207 -0.25839335] # Punkt 1]
```

```
[ 0.99999827 -0.9999994 -0.9999975 -0.85943311 -0.9999879 ] # Punkt 2
[ 0.99928284 -0.99999815 -0.99990582  0.98579615 -0.92205751]] # Punkt 3
```

Das war nicht allzu schwer. Natürlich wird der Graph sehr groß, wenn Sie auf diese Weise ein RNN über 100 Schritte laufen lassen möchten. Sehen wir uns deshalb an, wie Sie das gleiche Modell mit den in TensorFlow eingebauten RNN-Operationen implementieren können.

Statisches Aufrollen entlang der Zeitachse

Die Funktion `static_rnn()` erstellt ein aufgerolltes RNN, indem es Zellen miteinander verkettet. Der folgende Code erstellt ein zum vorigen identisches Modell:

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
                                                dtype=tf.float32)

Y0, Y1 = output_seqs
```

Zunächst erstellen wir wie gewohnt Platzhalter für die Eingaben. Anschließend erzeugen wir eine `BasicRNNCell`, die Sie sich als eine Fabrik vorstellen können, die Kopien der Zelle zum Aufbau des aufgerollten RNN produziert (eine für jeden zeitlichen Schritt). Als Nächstes rufen wir `static_rnn()` auf und übergeben die Zellfabrik und die Eingabe-Tensoren. Wir geben außerdem den Datentyp der Eingaben an (damit wird die anfängliche Zustandsmatrix erstellt, die zu Beginn voller Nullen ist). Die Funktion `static_rnn()` ruft die Funktion `_call()` der Fabrik einmal pro Eingabe auf, erstellt zwei Kopien der Zelle (wobei jede eine Schicht aus fünf rekurrenten Neuronen enthält), die sich Gewichte und Bias-Terme teilen, und verkettet diese miteinander. Die Funktion `static_rnn()` gibt zwei Objekte zurück. Das erste ist eine Python-Liste mit den Ausgabe-Tensoren für jeden Schritt. Das zweite ist ein Tensor mit dem Endzustand des Netzes. Bei einfachen Zellen ist der Endzustand zur letzten Ausgabe identisch.

Bei 50 Zeitschritten wäre es nicht besonders bequem, 50 Platzhalter für die Eingaben und 50 Tensoren für die Ausgaben zu definieren. Beim Ausführen müssten Sie außerdem jeden der 50 Platzhalter bespielen und die 50 Ausgaben verändern. Dies lässt sich vereinfachen. Der folgende Code baut das gleiche RNN noch einmal auf, aber diesmal wird ein einzelner Platzhalter mit den Abmessungen `[None, n_steps, n_inputs]` verwendet, wobei die erste Dimension die Größe des Mini-Batches ist. Anschließend wird eine Liste der Eingabesequenzen für jeden Schritt extrahiert. `X_seqs` ist eine Python-Liste mit `n_steps` Tensoren mit den Abmessungen `[None, n_inputs]`, wobei auch hier die erste Dimension die Größe der Mini-Batches ist. Dazu vertauschen wir die beiden Dimensionen mit der Funktion `transpose()`, sodass die Schritte nun in der ersten Dimension liegen. Wir extrahieren daraufhin mit der Funktion `unstack()` eine Liste von Python-Tensoren entlang der ersten Dimension (d.h. einen Tensor pro Zeitschritt). Die nächsten zwei Zeilen sind die gleichen wie

zuvor. Schließlich sammeln wir alle Ausgabe-Tensoren mit der Funktion `stack()` in einem Tensor und vertauschen die ersten zwei Dimensionen. Damit erhalten wir den endgültigen Tensor `outputs` mit den Abmessungen `[None, n_steps, n_neurons]` (die erste Dimension ist auch hier die Größe der Mini-Batches).

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

Nun können wir das Netz ausführen, indem wir einen einzelnen Tensor mit den Sequenzen aller Mini-Batches einspeisen:

```
X_batch = np.array([
    # t = 0      t = 1
    [[0, 1, 2], [9, 8, 7]], # Punkt 0
    [[3, 4, 5], [0, 0, 0]], # Punkt 1
    [[6, 7, 8], [6, 5, 4]], # Punkt 2
    [[9, 0, 1], [3, 2, 1]], # Punkt 3
])
with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

Wir erhalten den einzelnen Tensor `outputs_val` mit allen Datenpunkten, Instanzen, Zeitschritten und Neuronen:

```
>>> print(outputs_val)
[[[-0.91279727  0.83698678 -0.89277941  0.80308062 -0.5283336 ]
 [-1.          1.          -0.99794829  0.99985468 -0.99273592]]

 [[-0.99994391  0.99951613 -0.9946925   0.99030769 -0.94413054]
 [ 0.48733309  0.93389565 -0.31362072  0.88573611  0.2424476 ]]

 [[-1.          0.99999875 -0.99975014  0.99956584 -0.99466234]
 [-0.99994856  0.99999434 -0.96058172  0.99784708 -0.9099462 ]]

 [[-0.95972425  0.99951482  0.96938795 -0.969908   -0.67668229]
 [-0.84596014  0.96288228  0.96856463 -0.14777924 -0.9119423 ]]]
```

Allerdings wird auch bei diesem Ansatz ein Graph mit einer Zelle pro Schritt aufgebaut. Bei 50 Schritten sähe der Graph recht hässlich aus. Dies ist mit dem Schreiben eines Programms ohne Schleifen vergleichbar (z.B. $Y_0=f(0, X_0)$; $Y_1=f(Y_0, X_1)$; $Y_2=f(Y_1, X_2)$; ...; $Y_{50}=f(Y_{49}, X_{50})$). Mit einem derart großen Graphen können Sie während der Backpropagation sogar Out-of-Memory-(OOM-)Fehler erhalten (vor allem, weil der Speicher von GPU-Karten begrenzt ist), weil die Werte sämtlicher Tensoren im Vorwärtsdurchlauf gespeichert werden müssen, sodass sich diese zur Berechnung der Gradienten im Rückwärtssdurchlauf verwenden lassen.

Glücklicherweise gibt es eine bessere Möglichkeit: die Funktion `dynamic_rnn()`.

Dynamisches Aufrollen entlang der Zeitachse

Die Funktion `dynamic_rnn()` verwendet die Operation `while_loop()`, um entsprechend oft über die Zelle zu iterieren. Sie können außerdem mit `swap_memory=True` den Speicher der GPU in den CPU-Speicher auslagern, um OOM-Fehler bei der Backpropagation zu umgehen. Bequemerweise lassen sich die Eingaben aller Schritte als ein Tensor übergeben (Abmessungen `[None, n_steps, n_inputs]`). Die Funktion gibt einen Tensor mit den Ausgaben jedes Schritts zurück (Abmessungen `[None, n_steps, n_neurons]`); es ist nicht nötig, Daten mit `stack`, `unstack` oder `transpose` umzuformen. Der folgende Code erstellt das oben verwendete RNN mithilfe der Funktion `dynamic_rnn()`. Dies ist viel angenehmer!

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])  
  
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)  
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```



Bei der Backpropagation ist die Operation `while_loop()` für den Zaubertrick zuständig: Sie speichert im Vorwärtsdurchlauf die Werte des Tensors für jede Iteration, sodass sich diese im Rückwärtsdurchlauf zum Berechnen der Gradienten einsetzen lassen.

Eingabesequenzen unterschiedlicher Länge

Bisher haben wir nur Eingabesequenzen mit fester Länge verwendet (genau zwei Schritte lange). Was, wenn die Eingabesequenzen sich in der Länge unterscheiden (z.B. Sätze)? In diesem Fall sollten Sie das Argument `sequence_length` beim Aufrufen der Funktion `dynamic_rnn()` (oder `static_rnn()`) angeben; dieses ist ein 1-D-Tensor, der die Länge der Eingabesequenz für jeden Datenpunkt angibt. Beispielsweise so:

```
seq_length = tf.placeholder(tf.int32, [None])  
  
[...]  
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,  
                                    sequence_length=seq_length)
```

Nehmen wir an, die zweite Eingabesequenz enthält nur eine anstatt zweier Eingaben. Diese muss mit einem Nullvektor aufgefüllt werden, um in den Eingabe-Tensor `X` zu passen (weil dessen zweite Dimension die Größe der längsten Sequenz ist – in diesem Fall 2).

```
X_batch = np.array([  
    # Schritt 0 Schritt 1  
    [[0, 1, 2], [9, 8, 7]], # Punkt 0  
    [[3, 4, 5], [0, 0, 0]], # Punkt 1 (mit Nullvektor aufgefüllt)  
    [[6, 7, 8], [6, 5, 4]], # Punkt 2  
    [[9, 0, 1], [3, 2, 1]], # Punkt 3  
])  
seq_length_batch = np.array([2, 1, 2, 2])
```

Natürlich müssen Sie nun den beiden Platzhaltern `X` und `seq_length` Werte übergeben:

```
with tf.Session() as sess:  
    init.run()  
    outputs_val, states_val = sess.run(  
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Das RNN gibt nur bei jedem Schritt jenseits der Länge der Eingabesequenz Nullvektoren aus (beachten Sie die Ausgabe des zweiten Datenpunkts im zweiten Schritt):

```
>>> print(outputs_val)  
[[[-0.68579948 -0.25901747 -0.80249101 -0.18141513 -0.37491536]  
 [-0.99996698 -0.94501185 0.98072106 -0.9689762 0.99966913]] # Endzustand  
  
 [[-0.99099374 -0.64768541 -0.67801034 -0.7415446 0.7719509] # Endzustand  
 [ 0. 0. 0. 0. 0.] ] # Nullvektor  
  
 [[-0.99978048 -0.85583007 -0.49696958 -0.93838578 0.98505187]  
 [-0.99951065 -0.89148796 0.94170523 -0.38407657 0.97499216]] # Endzustand  
  
 [[-0.02052618 -0.94588047 0.99935204 0.37283331 0.9998163]  
 [-0.91052347 0.05769409 0.47446665 -0.44611037 0.89394671]]] # Endzustand
```

Der Tensor `states` enthält außerdem den Endzustand aller Zellen (ohne die Nullvektoren):

```
>>> print(states_val)  
[[[-0.99996698 -0.94501185 0.98072106 -0.9689762 0.99966913] # t = 1  
 [-0.99099374 -0.64768541 -0.67801034 -0.7415446 0.7719509] # t = 0  
 [-0.99951065 -0.89148796 0.94170523 -0.38407657 0.97499216] # t = 1  
 [-0.91052347 0.05769409 0.47446665 -0.44611037 0.89394671]] # t = 1
```

Ausgabesequenzen unterschiedlicher Länge

Wie sieht es aus, wenn auch die Ausgabesequenzen unterschiedliche Längen haben? Wenn Sie im Voraus wissen, welche Länge jede Sequenz hat (beispielsweise die gleiche Länge wie die Eingabesequenz), können Sie wie oben beschrieben den Parameter `sequence_length` verwenden. Leider ist dies nicht immer möglich: Beispielsweise hat die Übersetzung eines Satzes grundsätzlich nicht die gleiche Länge wie der übersetzte Satz. In diesem Fall ist es üblich, eine spezielle Ausgabe, das *End-of-Sequence Token* (EOS Token), zu erzeugen. Jede Ausgabe nach dem EOS wird ignoriert (dies werden wir weiter unten besprechen).

Nun wissen Sie, wie man ein RNN erstellt (oder genauer entlang der Zeitachse aufgerolltes RNN). Aber wie wird es trainiert?

Trainieren von RNNs

Der Trick zum Trainieren eines RNN ist, es entlang der Zeitachse aufzurollen (wie oben) und dann das gewöhnliche Backpropagation-Verfahren zu verwenden (siehe Abbildung 14-5). Diese Strategie bezeichnet man auch als *Backpropagation through Time* (BPTT).

Wie beim gewöhnlichen Backpropagation-Verfahren gibt es zuerst einen Vorwärtsdurchlauf durch das aufgerollte Netz (als gestrichelte Linien dargestellt); die Ausgabesequenz wird dann mithilfe der Kostenfunktion $C(Y_{(t_{\min})}, Y_{(t_{\min}+1)}, \dots, Y_{(t_{\max})})$ evaluiert (wobei t_{\min} und t_{\max} der erste und letzte Zeitschritt bei der Ausgabe sind; ignorierte Ausgaben zählen nicht), und die Gradienten dieser Kostenfunktion werden rückwärts durch das aufgerollte Netz verfolgt (als durchgezogene Pfeile dargestellt). Schließlich werden die Modellparameter mit den während der BPTT berechneten Gradienten aktualisiert. Es werden dabei die Gradienten sämtlicher von der Kostenfunktion verwendeten Ausgaben verfolgt, nicht nur die der letzten Ausgabe (beispielsweise wird die Kostenfunktion in Abbildung 14-5 aus den letzten drei Ausgaben des Netzes $Y_{(2)}$, $Y_{(3)}$ und $Y_{(4)}$ berechnet, daher fließen die Gradienten durch diese drei Ausgaben, nicht aber durch $Y_{(0)}$ und $Y_{(1)}$). Da außerdem bei jedem Zeitschritt die gleichen Parameter \mathbf{W} und \mathbf{b} verwendet werden, ist es richtig, dass das Backpropagation-Verfahren die Summe über sämtliche Schritte bildet.

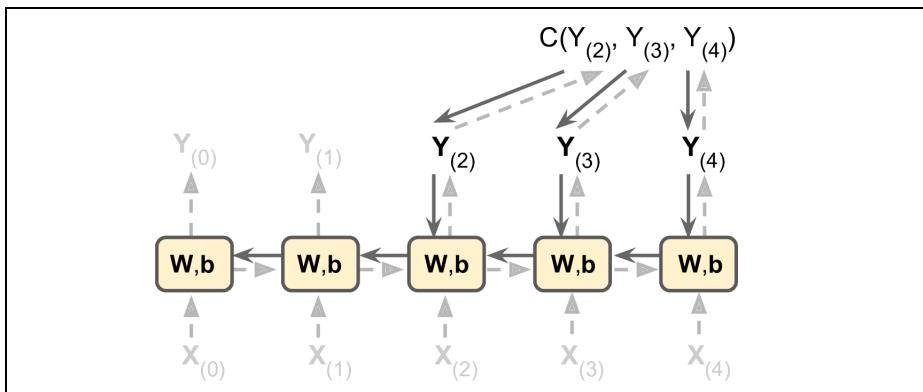


Abbildung 14-5: Backpropagation through Time

Trainieren eines Sequenz-Klassifikators

Trainieren wir nun ein RNN, um MNIST-Bilder zu klassifizieren. Ein Convolutional Neural Network wäre zur Bildklassifikation besser geeignet (siehe Kapitel 13), aber wir erhalten so ein einfaches Beispiel, das Sie bereits kennen. Wir sehen jedes Bild als Sequenz von 28 Zeilen mit je 28 Pixeln an (jedes MNIST-Bild ist 28×28 Pixel groß). Wir verwenden Zellen mit 150 rekurrenten Neuronen, eine vollständig

verbundene Schicht mit 10 Neuronen (eine pro Kategorie), die mit der Ausgabe des letzten Schritts verbunden ist, sowie eine Softmax-Schicht (siehe Abbildung 14-6).

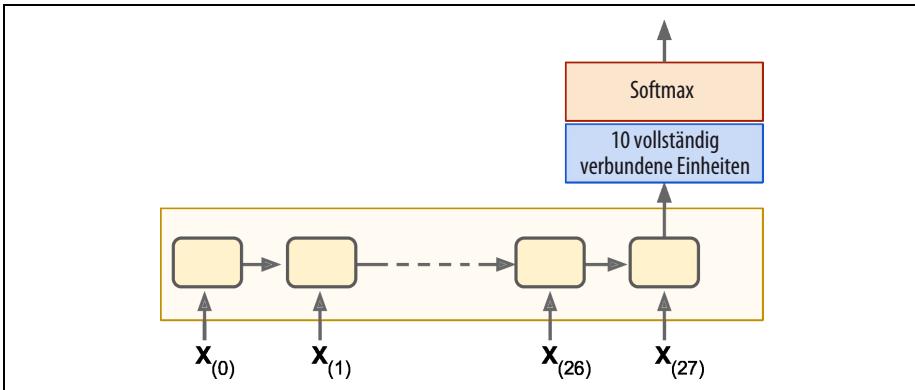


Abbildung 14-6: Sequenz-Klassifikator

Die Konstruktionsphase ist recht einfach; sie entspricht im Wesentlichen dem in Kapitel 10 erstellten MNIST-Klassifikator, außer dass wir die verborgenen Schichten durch ein aufgerolltes RNN ersetzen. Die vollständig verbundene Schicht ist mit dem Tensor `states` verbunden, der nur den Endzustand des RNN enthält (d.h. die 28. Ausgabe). Außerdem ist `y` ein Platzhalter für die Zielkategorien.

```

n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                          logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()

```

Nun laden wir die MNIST-Daten und formen die Testdaten auf die vom Netz erwarteten Abmessungen [`batch_size`, `n_steps`, `n_inputs`] um. Um das Umformen der Trainingsdaten kümmern wir uns auch gleich.

```

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels

```

Nun sind wir soweit, das RNN zu trainieren. Die Ausführungsphase ist exakt die gleiche wie beim MNIST-Klassifikator in Kapitel 10, außer dass wir jeden Batch vor der Eingabe ins Netz umformen.

```

n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
            acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
            print(epoch, "Genauigkeit Training:", acc_train, "Genauigkeit Test:", acc_
test)

```

Die Ausgabe sollte folgendermaßen aussehen:

```

0 Genauigkeit Training: 0.94 Genauigkeit Test: 0.9308
1 Genauigkeit Training: 0.933333 Genauigkeit Test: 0.9431
[...]
98 Genauigkeit Training: 0.98 Genauigkeit Test: 0.9794
99 Genauigkeit Training: 1.0 Genauigkeit Test: 0.9804

```

Wir erhalten eine Genauigkeit von mehr als 98% – nicht schlecht! Sie könnten dieses Ergebnis durch Optimieren der Hyperparameter, durch Initialisierung nach He, längeres Trainieren oder etwas Regularisierung (z.B. Drop-out) sicher noch verbessern.



Sie können für das RNN einen Initializer angeben, indem Sie den Code zur Konstruktion in einen Scope einbetten (z.B. `variable_scope("rnn", initializer=variance_scaling_initializer())`) zum Verwenden der Initialisierung nach He).

Trainieren der Vorhersage von Zeitreihen

Wenden wir uns nun Zeitreihen wie Aktienkursen, Lufttemperaturen, Gehirnwellen und so weiter zu. In diesem Abschnitt werden wir ein RNN so trainieren, dass es den nächsten Wert einer künstlich generierten Zeitreihe vorhersagen kann. Jeder Trainingsdatenpunkt ist eine zufällig ausgewählte Sequenz aus 20 aufeinanderfolgenden Werten der gleichen Zeitreihe, und die Zielsequenz entspricht der um einen Schritt in die Zukunft verschobenen Eingabesequenz (siehe Abbildung 14-7).

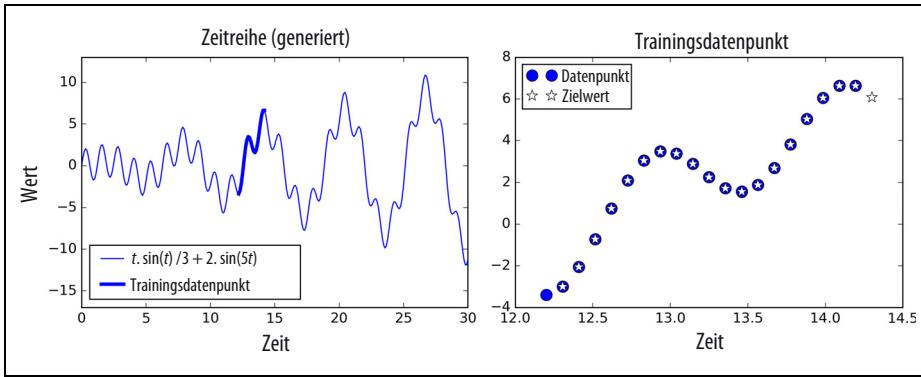


Abbildung 14-7: Zeitreihe (links) und ein Trainingsdatenpunkt aus dieser Serie (rechts)

Erstellen wir zunächst das RNN. Es enthält 100 rekurrente Neuronen, die wir entlang 20 Zeitschritten aufrollen, da jeder Trainingsdatenpunkt 20 Eingabewerte lang ist. Jede Eingabe enthält nur ein Merkmal (den Wert zu diesem Zeitpunkt). Die Zielwerte sind ebenfalls Sequenzen mit 20 Eingaben, jede mit einem einzelnen Wert. Der Code ist fast der gleiche wie zuvor:

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```



Meistens gibt es mehr als nur ein Eingabemerkmals. Wenn Sie beispielsweise Aktienkurse vorhersagen möchten, hätten Sie vermutlich in jedem Schritt viele Eingabemerkmale wie die Preise konkurrierender Aktien, Ratings von Analysten oder andere Merkmale, die dem System beim Treffen von Vorhersagen helfen können.

Wir haben damit bei jedem Schritt einen Ausgabevektor der Größe 100. Aber wir benötigen eigentlich bei jedem Zeitschritt einen einzelnen Ausgabewert. Die einfachste Lösung hierfür ist, die Zelle in einen `OutputProjectionWrapper` einzupacken. Ein Wrapper einer Zelle verhält sich wie eine gewöhnliche Zelle und leitet jeden Methodenaufruf an die eingepackte Zelle weiter. Sie fügt aber auch neue Funktionalität hinzu. Der `OutputProjectionWrapper` fügt eine vollständig verbundene Schicht linearer Neuronen (d.h. ohne Aktivierungsfunktion) nach jeder Ausgabe hinzu (dies beeinflusst den Zustand der Zelle nicht). Diese vollständig verbundenen Schichten besitzen die gleichen (trainierbaren) Gewichte und Bias-Terme. Das sich daraus ergebende RNN ist in Abbildung 14-8 dargestellt.

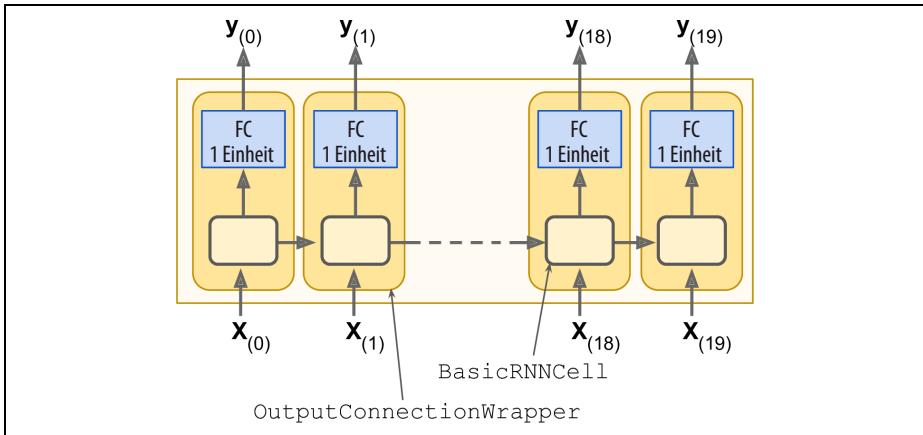


Abbildung 14-8: RNN-Zellen mit Projektion der Ausgabe

Einen Wrapper auf eine Zelle anzuwenden, ist recht leicht. Verändern wir den obigen Code, um die `BasicRNNCell` in einen `OutputProjectionWrapper` zu packen:

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```

So weit, so gut. Nun müssen wir die Kostenfunktion definieren. Wir verwenden dazu wie bei den bisherigen Regressionsaufgaben den mittleren quadrierten Fehler (MSE). Anschließend erstellen wir wie gewohnt einen Adam-Optimierer, die Operation zum Trainieren und die Operation zum Initialisieren der Variablen:

```
learning_rate = 0.001

loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Auf geht es zur Ausführungsphase:

```
n_iterations = 1500
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = [...] # hole nächsten Batch
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)
```

Die Ausgabe des Programms sollte folgendermaßen aussehen:

```
0      MSE: 13.6543  
100    MSE: 0.538476  
200    MSE: 0.168532  
300    MSE: 0.0879579  
400    MSE: 0.0633425  
[...]
```

Ist das Modell erst einmal trainiert, können Sie Vorhersagen treffen:

```
X_new = [...] # Neue Sequenzen  
y_pred = sess.run(outputs, feed_dict={X: X_new})
```

Abbildung 14-9 zeigt die vorhergesagte Sequenz für den zuvor betrachteten Datenpunkt (in Abbildung 14-7) nach einem 1000 Iterationen langen Training.

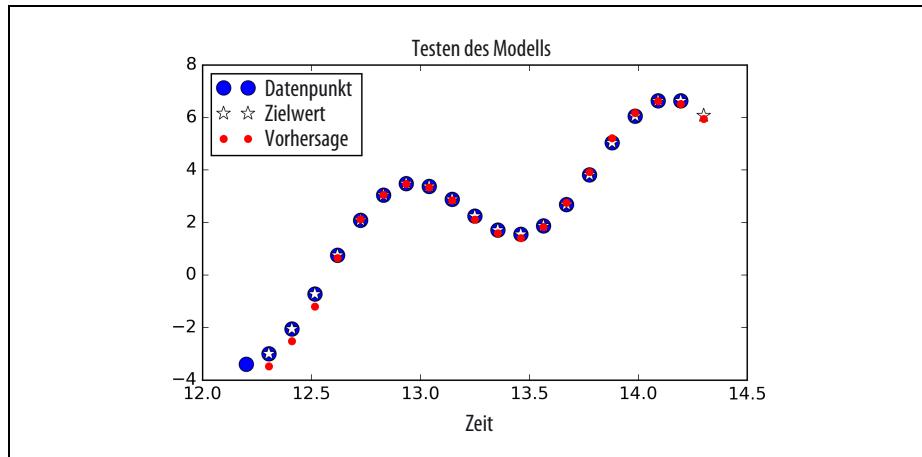


Abbildung 14-9: Vorhersage einer Zeitreihe

Auch wenn der `OutputProjectionWrapper` die einfachste Lösung ist, um die Dimensionalität der Ausgabesequenz eines RNN auf einen Wert pro Schritt zu reduzieren (pro Datenpunkt), ist es nicht die effizienteste: Sie können auch die Ausgaben des RNNs von den Abmessungen `[batch_size, n_steps, n_neurons]` auf `[batch_size * n_steps, n_neurons]` ändern und dann eine einzelne, vollständig verbundene Schicht mit der gewünschten Anzahl Ausgaben hinzufügen (in unserem Fall nur 1). Dies führt zu einem Ausgabe-Tensor mit den Abmessungen `[batch_size * n_steps, n_outputs]`, den Sie auf `[batch_size, n_steps, n_outputs]` umformen können. Diese Operationen sind in Abbildung 14-10 dargestellt.

Um diese Lösung zu implementieren, kehren wir zunächst zu einer einfachen Zelle ohne den `OutputProjectionWrapper` zurück:

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)  
rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

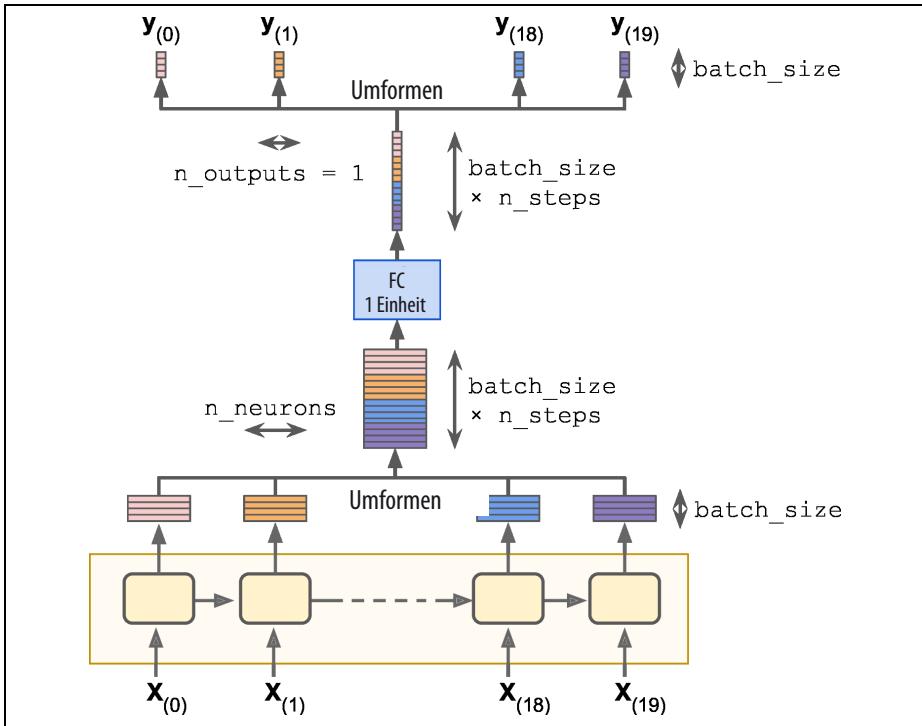


Abbildung 14-10: Anwenden von stack auf alle Ausgaben, Anwenden der Projektion, anschließend Anwenden von unstack auf das Ergebnis

Anschließend stapeln wir die Ausgaben mit der Operation `reshape()` auf, fügen die vollständig verbundene lineare Schicht hinzu (ohne Aktivierungsfunktion, dies ist nur eine Projektion) und teilen die Ausgabe schließlich wieder mit `reshape()` auf:

```
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
```

Der restliche Code ist der gleiche wie vorher. Dies bedeutet einen erheblichen Zuge-
winn an Geschwindigkeit, da es nun anstelle einer vollständig verbundenen Schicht
pro Zeitschritt nur eine Schicht insgesamt gibt.

Kreative RNNs

Mit einem Modell, das die Zukunft vorhersagen kann, können wir wie eingangs erwähnt neue Sequenzen generieren. Dazu benötigen wir lediglich eine Startsequenz mit n_steps -Werten (z.B. Nullen) und verwenden unser Modell, um den nächsten Wert vorherzusagen. Diese Vorhersage fügen wir zur Sequenz hinzu und geben die letzten n_steps -Werte wieder in das Modell ein, um den nächsten Wert vorherzusagen und so weiter. Dieser Vorgang erzeugt eine neue Sequenz, die einige Ähnlichkeit mit der ursprünglichen Zeitreihe aufweist (siehe Abbildung 14-11).

```

sequence = [0.] * n_steps
for iteration in range(300):
    X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    sequence.append(y_pred[0, -1, 0])

```

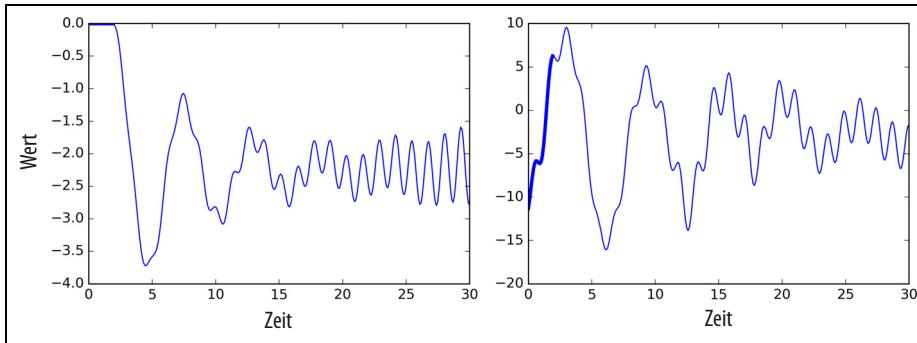


Abbildung 14-11: Kreative Sequenzen, mit Nullen gestartet (links) und mit einem Datenpunkt (rechts)

Nun können Sie versuchen, alle Ihre Platten von John Lennon in ein RNN einzugeben und zu schauen, ob es das nächste »Imagine« generieren kann. Sie werden dazu aber vermutlich ein weitaus mächtigeres, tieferes RNN mit mehr Neuronen benötigen. Schauen wir uns diese Deep-RNNs als Nächstes an.

Deep-RNNs

Es ist recht verbreitet, Zellen wie in Abbildung 14-12 in mehreren Schichten aufeinanderzustapeln. Damit erhalten Sie ein *Deep-RNN*.

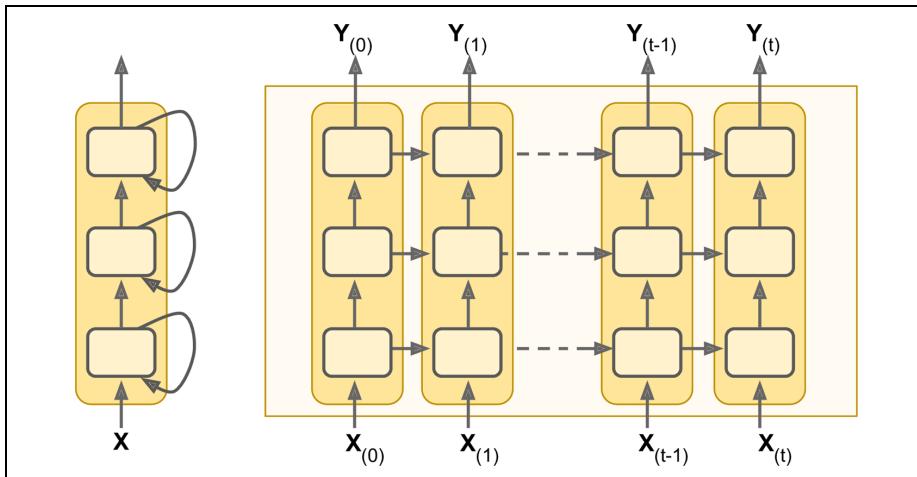


Abbildung 14-12: Deep-RNN (links), entlang der Zeitachse aufgerollt (rechts)

Um ein Deep-RNN in TensorFlow zu implementieren, erstellen Sie mehrere Zellen und stapeln diese zu einer MultiRNNCell auf. Im folgenden Codebeispiel stapeln wir drei identische Zellen auf (Sie könnten aber auch genauso gut unterschiedliche Zellen mit unterschiedlich vielen Neuronen verwenden):

```
n_neurons = 100
n_layers = 3

layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
          for layer in range(n_layers)]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

Das ist alles! Die Variable `states` ist ein Tupel mit einem Tensor pro Schicht, der den Endzustand der Zelle dieser Schicht repräsentiert (mit den Abmessungen `[batch_size, n_neurons]`). Wenn Sie beim Erstellen der MultiRNNCell den Parameter `state_is_tuple=False` setzen, wird `states` zu einem einzelnen Tensor mit den Zuständen aller Schichten und den Abmessungen `[batch_size, n_layers * n_neurons]`. Bis zur TensorFlow-Version 0.11.0 war dieses Verhalten die Standardeinstellung.

Ein Deep-RNN über mehrere GPUs verteilen

In Kapitel 12 betonten wir, dass sich RNNs effizient über mehrere GPUs verteilen lassen, indem jede Schicht auf einer anderen GPU platziert wird (siehe Abbildung 12-6). Wenn Sie allerdings versuchen, jede Zelle in einem anderen `device()`-Block zu erstellen, funktioniert es nicht:

```
with tf.device("/gpu:0"): # FALSCH! Dies wird ignoriert.
    layer1 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)

with tf.device("/gpu:1"): # FALSCH! Dies auch.
    layer2 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

Dieser Versuch scheitert, weil `BasicRNNCell` eine Fabrik für Zellen ist, nicht die Zelle *selbst* (siehe oben); beim Erstellen der Fabrik werden keine Zellen erstellt und damit auch keine Variablen. Der `device`-Block wird einfach ignoriert. Die Zellen werden erst später erstellt. Wenn Sie `dynamic_rnn()` aufrufen, wird `MultiRNNCell` aufgerufen, dieses ruft jedes einzelne `BasicRNNCell`-Objekt auf, die wiederum die einzelnen Zellen erzeugen (einschließlich ihrer Variablen). Leider erlaubt keine dieser Klassen eine Kontrolle darüber, auf welchen Recheneinheiten die Variablen erstellt werden. Wenn Sie versuchen, den Aufruf von `dynamic_rnn()` in einem `device`-Block zu platzieren, wird das gesamte RNN einer Recheneinheit zugeordnet. Hängen wir also fest? Glücklicherweise nicht! Der Trick besteht darin, einen eigenen Wrapper für die Zellen zu erstellen:

```
import tensorflow as tf

class DeviceCellWrapper(tf.contrib.rnn.RNNCell):
```

```

def __init__(self, device, cell):
    self._cell = cell
    self._device = device

@property
def state_size(self):
    return self._cell.state_size

@property
def output_size(self):
    return self._cell.output_size

def __call__(self, inputs, state, scope=None):
    with tf.device(self._device):
        return self._cell(inputs, state, scope)

```

Dieser Wrapper leitet einfach jeden Methodenaufruf an eine andere Zelle weiter, nur dass die Funktion `__call__()` einen device-Block enthält.² Nun können Sie jede Schicht auf einer anderen GPU platzieren:

```

devices = ["/gpu:0", "/gpu:1", "/gpu:2"]
cells = [DeviceCellWrapper(dev, tf.contrib.rnn.BasicRNNCell(num_units=n_neurons))
         for dev in devices]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

```



Setzen Sie nicht `state_is_tuple=False`, andernfalls verkettet das MultiRNNCell-Objekt die Zustände aller Zellen zu einem einzelnen Tensor auf einer einzelnen GPU.

Drop-out verwenden

Wenn Sie ein sehr tiefes RNN konstruieren, kann es zum Overfitting der Trainingsdaten kommen. Drop-out ist eine verbreitete Technik, um dies zu verhindern (siehe Kapitel 11). Sie können einfach wie gewohnt vor oder nach dem RNN eine Drop-out-Schicht hinzufügen, aber wenn Sie auch innerhalb der RNN-Schichten Drop-out verwenden möchten, benötigen Sie den DropoutWrapper. Der folgende Code verwendet in jeder Schicht des RNN Drop-out und eliminiert jede Eingabe mit 50 %iger Wahrscheinlichkeit:

```

keep_prob = 0.5

cells = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
         for layer in range(n_layers)]
cells_drop = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
              for cell in cells]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

```

² Hier wird das Entwurfsmuster *Decorator* verwendet.

Es ist ebenso möglich, durch Setzen von `output_keep_prob` Drop-out auch auf die Ausgaben anzuwenden.

Das Hauptproblem mit diesem Code ist, dass er Drop-out nicht nur beim Trainieren, sondern auch beim Testen verwendet (Sie erinnern sich, dass Drop-out nur beim Trainieren verwendet werden sollte). Leider enthält der `DropoutWrapper` keinen Platzhalter `training` (noch nicht?), daher müssen Sie sich entweder einen eigenen Drop-out-Wrapper schreiben oder zwei unterschiedliche Graphen verwenden: einen zum Trainieren, den anderen zum Testen. Die zweite Option sieht folgendermaßen aus:

```
import sys
training = (sys.argv[-1] == "train")

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cells = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
         for layer in range(n_layers)]
if training:
    cells = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
             for cell in cells]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
[...] # erstelle den restlichen Graphen

with tf.Session() as sess:
    if training:
        init.run()
        for iteration in range(n_iterations):
            [...] # trainiere das Modell
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
    else:
        saver.restore(sess, "/tmp/my_model.ckpt")
        [...] # verwende das Modell
```

Damit sollten Sie in der Lage sein, alle möglichen RNNs zu trainieren! Leider wird das Trainieren von RNNs mit längeren Sequenzen ein wenig schwieriger. Schauen wir uns einmal an, warum und was man dagegen tun kann.

Die Schwierigkeit, über viele Schritte zu trainieren

Um ein RNN für lange Sequenzen zu trainieren, müssen Sie es über viele Zeitschritte laufen lassen. Das aufgerollte RNN wird damit zu einem sehr tiefen neuronalen Netz. Wie bei jedem Deep-Learning-Netz spielt das Problem der schwindenden/explodierenden Gradienten eine Rolle (besprochen in Kapitel 11), sodass das Trainieren ewig dauert. Viele Tricks zum Bekämpfen dieses Problems lassen sich auch bei tiefen aufgerollten RNNs einsetzen: gute Initialisierung der Parameter, eine Aktivierungsfunktion ohne Sättigung (z.B. ReLU), Batch-Normalisierung, Gradient Clipping und schnellere Optimierer. Allerdings ist das Trainieren mit Sequenzen mittlerer Länge (z.B. 100 Eingaben) auch dann noch sehr langsam.

Der einfachste und verbreitetste Lösungsansatz ist, das RNN beim Trainieren nur über eine begrenzte Anzahl Zeitschritte aufzurollen. Dies nennt man *Truncated Backpropagation through Time*. In TensorFlow lässt sich dieses Verfahren implementieren, indem Sie einfach die Eingabesequenzen abschneiden. Beispielsweise würden Sie bei der obigen Vorhersage der Zeitreihe beim Trainieren einfach `n_steps` verringern. Das Problem dabei ist natürlich, dass das Modell dann keine längerfristigen Muster erlernen kann. Eine Alternative wäre, in diese verkürzten Sequenzen sowohl ältere als auch neuere Daten aufzunehmen, sodass das Modell lernt, beide zu verwenden (z.B. könnte die Sequenz monatliche Einträge für die letzten fünf Monate enthalten, dann wöchentliche Einträge für die letzten fünf Wochen, dann tägliche für die letzten fünf Tage). Auch dieser Ansatz hat jedoch seine Grenzen: Was ist, wenn die feinkörnigen Daten aus dem letzten Jahr nützlich sind? Was ist, wenn es ein kurzes aber wichtiges Ereignis gab, das auch Jahre später noch unbedingt berücksichtigt werden muss (z.B. ein Wahlergebnis)?

Außer der langen Trainingszeit besteht ein zweites Problem bei lang laufenden RNNs darin, dass die Erinnerung an die ersten Eingaben allmählich verblasst. Aufgrund der Transformationen der Daten beim Durchlaufen eines RNN geht bei jedem Schritt ein wenig Information verloren. Nach einer Weile enthält der Zustand des RNN praktisch keine Spur der ersten Eingaben mehr. Dies kann sich als echte Spaßbremse herausstellen. Nehmen wir beispielsweise an, Sie möchten eine Meinungsanalyse einer langen Filmbewertung durchführen, die mit den Wörtern »Ich fand diesen Film toll« beginnt, aber anschließend viele Dinge aufzählt, die den Film noch besser gemacht hätten. Wenn das RNN allmählich die ersten vier Wörter vergisst, wird es die Bewertung komplett falsch beurteilen. Um dieses Problem zu lösen, gibt es verschiedenartige Zellen mit Langzeitgedächtnis. Sie haben sich als derart erfolgreich entpuppt, dass die einfachen Zellen kaum noch verwendet werden. Schauen wir uns zunächst die beliebteste dieser Gedächtniszellen an: die LSTM-Zelle.

LSTM-Zellen

Die *Long Short-Term Memory* (LSTM)-Zelle wurde im Jahr 1997 (<https://goo.gl/j39AGv>)³ von Sepp Hochreiter und Jürgen Schmidhuber vorgeschlagen. Im Lauf der Jahre wurde sie von mehreren Wissenschaftlern wie Alex Graves, Halim Sak (<https://goo.gl/6BHh81>)⁴, Wojciech Zaremba (<https://goo.gl/SZ9kzB>)⁵ und vielen

3 »Long Short-Term Memory«, S. Hochreiter and J. Schmidhuber (1997).

4 »Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling«, H. Sak et al. (2014).

5 »Recurrent Neural Network Regularization«, W. Zaremba et al. (2015).

weiteren allmählich verbessert. Wenn Sie sich eine LSTM-Zelle als Blackbox vorstellen, lässt sie sich im Wesentlichen wie eine einfache Zelle verwenden, nur dass sie viel besser funktioniert; das Trainieren konvergiert schneller und erkennt weitreichende Abhängigkeiten in den Daten. In TensorFlow können Sie einfach eine `BasicLSTMCell` anstelle der `BasicRNNCell` verwenden:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

LSTM-Zellen verwalten zwei Zustandsvektoren. Aus Performance-Gründen werden diese standardmäßig getrennt abgelegt. Sie können dies ändern, indem Sie beim Erstellen einer `BasicLSTMCell` den Parameter `state_is_tuple=False` setzen.

Wie arbeitet eine LSTM-Zelle? Die Architektur einer einfachen LSTM-Zelle ist in Abbildung 14-13 dargestellt.

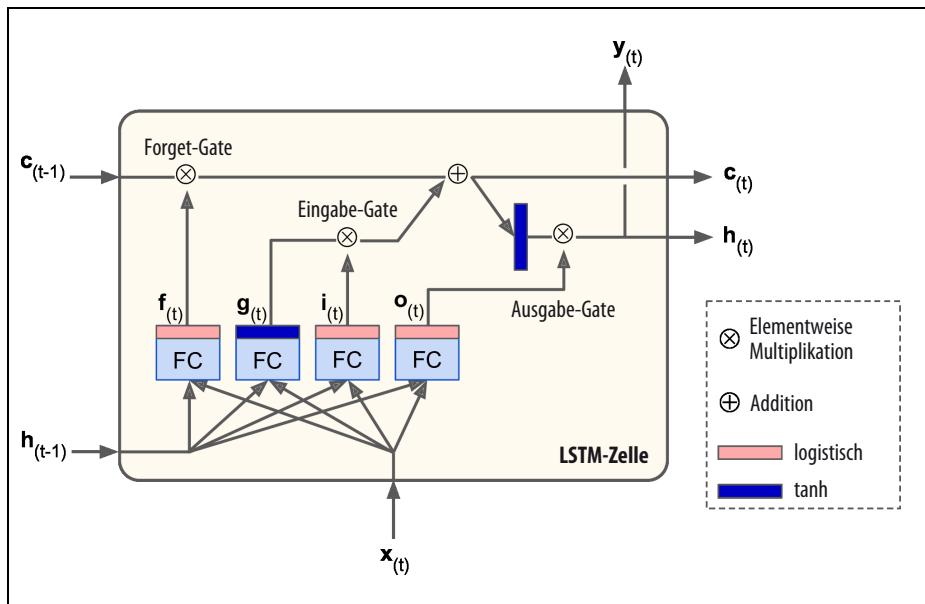


Abbildung 14-13: LSTM-Zelle

Solange Sie nicht in die Box schauen, sieht eine LSTM-Zelle genau wie eine gewöhnliche Zelle aus, außer dass ihr Zustand in zwei Vektoren aufgeteilt ist: $h_{(t)}$ und $c_{(t)}$ (»c« steht für »cell«). Sie können sich $h_{(t)}$ als das Kurzzeitgedächtnis und $c_{(t)}$ als das Langzeitgedächtnis vorstellen.

Öffnen wir nun die Blackbox! Der Hauptgedanke ist, dass das Netz lernen kann, was es im Langzeitgedächtnis speichern soll, was es vergessen und wie es das Gedächtnis interpretieren kann. Während der Langzeitzustand $c_{(t-1)}$ das Netz von links nach rechts durchschreitet, passiert er zuerst ein *Forget Gate*, das einige Erinnerungen vergisst. Anschließend werden über eine Operation zur Addition neue

Erinnerungen hinzugefügt (dieses addiert die von einem *Input Gate* ausgewählten Erinnerungen). Das Ergebnis $c_{(t)}$ wird unmittelbar und ohne weitere Transformation ausgegeben. Bei jedem Zeitschritt werden also einige Erinnerungen verworfen und einige hinzugefügt. Außerdem wird der Langzeitzustand nach der Addition kopiert und passiert die Funktion tanh. Das Ergebnis wird vom *Output Gate* gefiltert. Dabei ergibt sich der Zustand des Kurzzeitgedächtnisses $h_{(t)}$ (der der Ausgabe der Zelle bei diesem Schritt $y_{(t)}$ entspricht). Schauen wir uns nun an, woher neue Erinnerungen stammen und wie die Gates funktionieren.

Zuerst werden der aktuelle Eingabevektor $x_{(t)}$ und der Zustand des Kurzzeitgedächtnisses $h_{(t-1)}$ vier einzelnen vollständig verbundenen Schichten übergeben. Diese dienen unterschiedlichen Zwecken:

- Die wichtigste Schicht gibt $g_{(t)}$ aus. Sie hat die gewöhnliche Aufgabe, die aktuellen Eingaben $x_{(t)}$ und den vorigen Zustand (des Kurzzeitgedächtnisses) $h_{(t-1)}$ zu analysieren. In einer einfachen Zelle gibt es außer dieser Schicht nichts, ihre Ausgabe wird direkt an $y_{(t)}$ und $h_{(t)}$ übergeben. Im Gegensatz dazu wird die Ausgabe dieser Schicht in einer LSTM-Zelle nicht direkt nach außen geleitet. Stattdessen wird sie teilweise im Langzeitgedächtnis gespeichert.
- Die drei übrigen Schichten sind *Gate Controller*. Da sie die logistische Aktivierungsfunktion verwenden, reichen ihre Ausgaben von 0 bis 1. Wie Sie sehen, werden ihre Ausgaben in elementweisen Multiplikationen verwendet. Wenn sie also Nullen ausgeben, schließen sie das Gate, wenn sie Einsen ausgeben, wird es geöffnet. Im Einzelnen:
 - Das *Forget Gate* steuert, welche Teile des Langzeitgedächtnisses gelöscht werden sollen (von $f_{(t)}$ kontrolliert).
 - Das *Input Gate* steuert, welche Teile von $g_{(t)}$ ins Langzeitgedächtnis übergehen sollen (deshalb sagen wir, dass es nur »teilweise gespeichert« wird; wird von $i_{(t)}$ kontrolliert).
 - Schließlich steuert das *Output Gate*, welche Teile des Langzeitgedächtnisses ausgelesen und bei diesem Schritt ausgegeben werden sollen (sowohl an $h_{(t)}$ als auch $y_{(t)}$; von $o_{(t)}$ kontrolliert).

Zusammengefasst lernt eine LSTM-Zelle, wichtige Eingaben zu erkennen (dies ist Aufgabe des Input Gate), diese im Langzeitgedächtnis zu speichern, solange wie nötig aufzuheben (dies ist Aufgabe des Forget Gate) und das Notwendige daraus zu entnehmen. Dies erklärt, warum LSTM-Zellen ausgesprochen erfolgreich beim Erfassen weitreichender Muster in Zeitreihen, langen Texten, Tonsignalen und anderen Daten sind.

Formel 14-3 fasst zusammen, wie sich der Zustand des Langzeitgedächtnisses, des Kurzzeitgedächtnisses und die Ausgabe eines Zeitschrittes für einen einzelnen Datenpunkt berechnen lassen (die Gleichungen für einen ganzen Mini-Batch sehen sehr ähnlich aus).

Formel 14-3: LSTM-Berechnungen

$$\begin{aligned}
 i_{(t)} &= \sigma(\mathbf{W}_{xi}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\
 f_{(t)} &= \sigma(\mathbf{W}_{xf}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f) \\
 o_{(t)} &= \sigma(\mathbf{W}_{xo}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o) \\
 g_{(t)} &= \tanh(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g) \\
 c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
 y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})
 \end{aligned}$$

- \mathbf{W}_{xi} , \mathbf{W}_{xf} , \mathbf{W}_{xo} und \mathbf{W}_{xg} sind die Gewichtsmatrizen der Verbindungen des Eingabevektors $\mathbf{x}_{(t)}$ zu den vier Schichten.
- \mathbf{W}_{hi} , \mathbf{W}_{hf} , \mathbf{W}_{ho} und \mathbf{W}_{hg} sind die Gewichtsmatrizen der Verbindungen des Kurzzeitgedächtnisses $\mathbf{h}_{(t-1)}$ zu diesen vier Schichten.
- \mathbf{b}_i , \mathbf{b}_f , \mathbf{b}_o und \mathbf{b}_g sind die Bias-Terme dieser vier Schichten. TensorFlow initialisiert \mathbf{b}_f mit einem Vektor voller Einsen anstatt Nullen. Dies verhindert, dass zu Beginn des Trainings alles vergessen wird.

Peephole-Verbindungen

In einer einfachen LSTM-Zelle betrachten die Gate Controller nur die Eingabe $\mathbf{x}_{(t)}$ und den Zustand des Kurzzeitgedächtnisses aus dem vorigen Schritt $\mathbf{h}_{(t-1)}$. Manchmal lohnt es sich, den Gate Controllern etwas mehr Kontext zu geben, indem sie auch in das Langzeitgedächtnis schauen können. Diese Idee wurde von Felix Gers und Jürgen Schmidhuber im Jahr 2000 vorgeschlagen (<https://goo.gl/ch8xz3>).⁶ Sie schlugen eine LSTM-Variante mit zusätzlichen Verbindungen vor, den *Peephole-Verbindungen*: Der Zustand des Langzeitgedächtnisses aus dem vorigen Schritt $c_{(t-1)}$ wird dabei zu den Eingaben des Forget Gates und des Input Gates hinzugefügt, und der aktuelle Zustand des Langzeitgedächtnisses $c_{(t)}$ zur Eingabe des Output Gates.

Um Peephole-Verbindungen in TensorFlow zu implementieren, müssen Sie anstelle einer `BasicLSTMCell` die `LSTMCell` verwenden und den Parameter `use_peepholes=True` setzen:

```
lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons, use_peepholes=True)
```

Es gibt viele weitere Varianten der LSTM-Zelle. Eine besonders beliebte ist die GRU-Zelle, die wir uns als Nächstes anschauen werden.

⁶ »Recurrent Nets that Time and Count«, F. Gers and J. Schmidhuber (2000).

GRU-Zellen

Die *Gated Recurrent Unit*-*(GRU)*-Zelle (siehe Abbildung 14-14) wurde von Kyunghyun Cho et al. in einem Artikel aus dem Jahr 2014 (<https://goo.gl/ZnAEOZ>)⁷ vorgeschlagen, in dem auch das oben erwähnte Encoder-Decoder-Netz erstmalig erwähnt wurde.

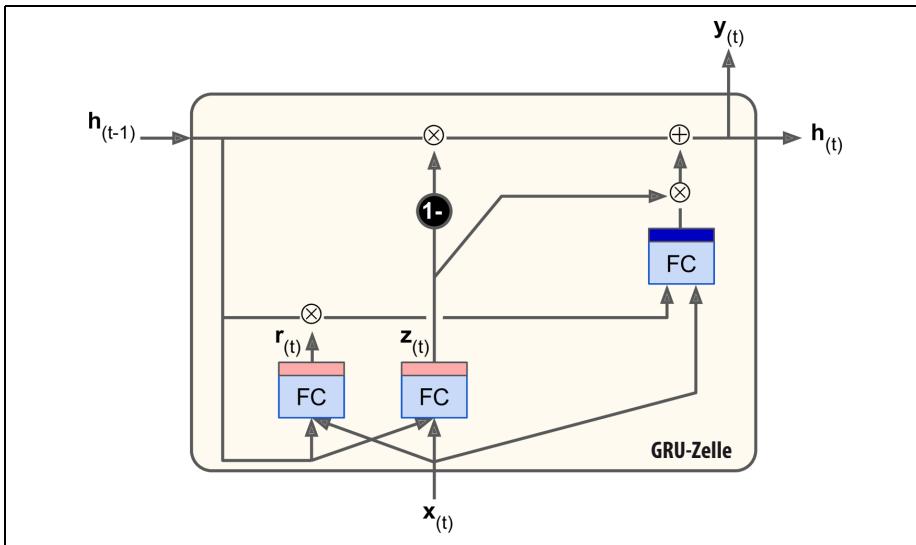


Abbildung 14-14: GRU-Zelle

Eine GRU-Zelle ist eine vereinfachte Variante der LSTM-Zelle, die aber genauso gut zu funktionieren scheint⁸ (was ihre wachsende Beliebtheit erklärt). Die wichtigsten Vereinfachungen sind:

- Beide Zustandsvektoren sind zu einem einzigen Vektor $h_{(t)}$ vereinigt.
- Ein einzelner Gate Controller steuert sowohl das Forget Gate als auch das Input Gate. Falls der Gate Controller eine 1 ausgibt, ist das Input Gate offen und das Forget Gate geschlossen. Gibt er eine 0 aus, passiert das Gegenteil. Anders ausgedrückt, wird beim Speichern von Erinnerungen der Speicherplatz dafür zuerst gelöscht. Dies für sich allein ist übrigens eine häufige Variante der LSTM-Zelle.
- Es gibt kein Output-Gate; bei jedem Schritt wird der vollständige Zustandsvektor ausgegeben. Es gibt dafür aber einen neuen Gate Controller, der steu-

7 »Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation«, K. Cho et al. (2014).

8 Ein Artikel von Klaus Greff et al. aus dem Jahr 2015, »LSTM: A Search Space Odyssey«, (<https://goo.gl/hZB4KW>) deutet darauf hin, dass sämtliche LSTM-Varianten etwa gleich gut abschneiden.

ert, welcher Teil des vorigen Zustands der Hauptschicht zur Verfügung gestellt wird.

Formel 14-4 fasst zusammen, wie sich der Zustand der Zelle bei einem Schritt für einen einzelnen Datenpunkt berechnen lässt.

Formel 14-4: GRU Berechnungen

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{(t-1)}) \\ \mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{(t-1)}) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)})\right) \\ \mathbf{h}_{(t)} &= (1 - \mathbf{z}_{(t)}) \otimes \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{(t)}\end{aligned}$$

Eine GRU-Zelle lässt sich in TensorFlow sehr einfach erstellen:

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

LSTM- und GRU-Zellen sind einer der Hauptgründe für den Erfolg von RNNs in den vergangenen Jahren, insbesondere bei der *natürlichen Sprachverarbeitung* (NLP).

Natürliche Sprachverarbeitung

Die meisten erstklassigen Anwendungen von NLP wie maschinelle Übersetzung, automatisches Zusammenfassen, Parsen, Meinungsanalyse und so weiter beruhen heute (zumindest teilweise) auf RNNs. In diesem letzten Abschnitt werfen wir einen kurzen Blick darauf, wie ein Modell zur maschinellen Übersetzung aussieht. Dieses Thema wird von den ausgezeichneten TensorFlow-Tutorials zu *Word2Vec* (<https://goo.gl/edArdi>) und *Seq2Seq* (<https://goo.gl/L82gvS>) sehr ausführlich behandelt. Sie sollten sich diese daher unbedingt ansehen.

Word Embeddings

Bevor wir beginnen, müssen wir uns für eine Repräsentation von Wörtern entscheiden. Eine Möglichkeit wäre, jedes Wort in einem One-Hot-Vektor abzulegen. Angenommen, unser Vokabular bestünde aus 50000 Wörtern, so würde das n^{te} Wort als Vektor mit 50000 Dimensionen repräsentiert sein, der bis auf eine Eins an der n^{ten} Position nur aus Nullen besteht. Mit einem derart großen Vokabular wäre diese dünn besetzte Repräsentation überhaupt nicht effizient. Idealerweise sollten ähnliche Wörter ähnlich repräsentiert werden, sodass das Modell leicht von einem Wort auf ähnliche Wörter verallgemeinern kann. Wenn das Modell beispielsweise lernt, dass »Ich trinke Milch« ein gültiger Satz ist, und weiß, dass »Milch« ähnlich zu »Wasser« ist, aber weit von »Schuhe« entfernt, so wird es zu dem Schluss kommen, dass »Ich trinke Wasser« wahrscheinlich ebenfalls ein gülti-

ger Satz ist, »Ich trinke Schuhe« hingegen nicht. Aber wie sollen Sie eine solche bedeutungstragende Repräsentation erhalten?

Die häufigste Lösung ist, jedes Wort im Vokabular durch ein sogenanntes *Embedding*, einen vergleichsweise kleinen und dichten Vektor (z.B. mit 150 Dimensionen), zu repräsentieren und das neuronale Netz beim Trainieren ein gutes Embedding für jedes Wort erlernen zu lassen. Zu Beginn des Trainings werden die Embeddings zufällig gewählt, im weiteren Verlauf werden sie aber durch das Backpropagation-Verfahren automatisch so verändert, dass das Netz seine Aufgabe erfüllen kann. Dies bedeutet üblicherweise, dass ähnliche Wörter nah beieinander geclustert und sogar in einer halbwegs sinnvollen Art und Weise organisiert werden. Beispielsweise können Embeddings entlang verschiedener Achsen für Geschlecht, Singular/Plural, Adjektiv/Nomen und so weiter platziert werden. Das Ergebnis ist recht faszinierend.⁹

In TensorFlow müssen Sie zunächst eine Variable erstellen, die die Embeddings für jedes Wort in Ihrem Vokabular enthält (zufällig initialisiert):

```
vocabulary_size = 50000
embedding_size = 150

init_embeds = tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0)
embeddings = tf.Variable(init_embeds)
```

Nehmen wir an, Sie möchten den Satz »Ich trinke Milch« in Ihr neuronales Netz einspeisen. Sie sollten den Satz zunächst vorverarbeiten und in eine Liste bekannter Wörter aufteilen. Beispielsweise könnten Sie unnötige Zeichen entfernen, unbekannte Wörter durch ein vordefiniertes Token wie »[UNK]« ersetzen, Zahlen durch »[NUM]« und URLs durch »[URL]« ersetzen und so weiter. Sobald Sie eine Liste bekannter Wörter haben, können Sie die als Identifikator des Worts dienende Integerzahl in einem Dictionary (von 0 bis 49999) nachschlagen, beispielsweise [72, 3335, 288]. An diesem Punkt sind Sie so weit, diese Identifikatoren über einen Platzhalter an TensorFlow zu schicken und mit der Funktion `embedding_lookup()` die entsprechenden Embeddings zu erhalten:

```
train_inputs = tf.placeholder(tf.int32, shape=[None]) # von IDs...
embed = tf.nn.embedding_lookup(embeddings, train_inputs) # ...zu Embeddings
```

Sobald Ihr Modell gute Embeddings der Wörter erlernt hat, lassen sich diese recht effizient in beliebigen NLP-Anwendungen nutzen: Schließlich sind »Milch« und »Wasser« unabhängig von Ihrer Anwendung einander ähnlich und zu »Schuhe« unähnlich. Anstatt aber Ihre eigenen Embeddings zu trainieren, sollten Sie sich vortrainierte Embeddings herunterladen. Wie beim Wiederverwenden vortrainierter Schichten (siehe Kapitel 11) können Sie sich entscheiden, die vortrainierten Embeddings einzufrieren (z.B. die Variable `embeddings` mit `trainable=False` anzulegen) oder das Backpropagation-Verfahren diese für ihre Anwendung optimieren

⁹ Details finden Sie im großartigen Blogeintrag (<https://goo.gl/5rLNTj>) von Christopher Olah oder der Serie von Blogeinträgen (<https://goo.gl/ojjjiE>) von Sebastian Ruder.

lassen. Die erste Möglichkeit beschleunigt das Trainieren, die zweite führt zu etwas höherer Leistung.



Embeddings sind auch zum Repräsentieren kategorischer Attribute nützlich, die eine große Anzahl unterschiedlicher Werte annehmen können, insbesondere wenn es komplexe Ähnlichkeiten zwischen den Werten gibt. Als Beispiele dienen Berufe, Hobbies, Gerichte, Tierarten, Marken und so weiter.

Sie haben nun beinahe alle Werkzeuge zum Implementieren eines Systems zur maschinellen Übersetzung beisammen. Sehen wir uns dieses näher an.

Ein Encoder-Decoder-Netz zur maschinellen Übersetzung

Schauen wir uns ein einfaches Modell zur maschinellen Übersetzung (<https://goo.gl/0g9zWP>)¹⁰ an, das englische Sätze ins Französische übersetzt (siehe Abbildung 14-15).

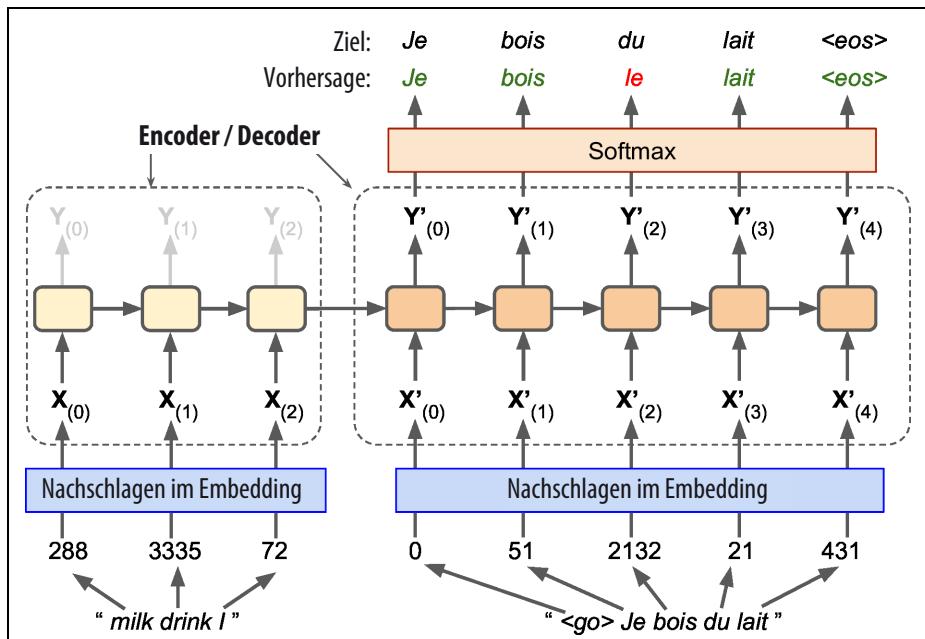


Abbildung 14-15: Ein einfaches Modell zur maschinellen Übersetzung

Die englischen Sätze werden in den Encoder eingegeben, und der Decoder gibt die französischen Übersetzungen aus. Beachten Sie, dass die französischen Übersetzungen auch als Eingabe für den Decoder verwendet werden, aber um einen Schritt nach hinten verschoben. Anders ausgedrückt, erhält der Decoder das Wort als Ein-

10 »Sequence to Sequence learning with Neural Networks«, I. Sutskever et al. (2014).

gabe, dass er im vorigen Schritt ausgeben *sollte* (unabhängig davon, was er tatsächlich ausgibt). Beim ersten Wort erhält er ein Token, das den Beginn des Satzes markiert (z.B. »<go>«). Der Decoder soll außerdem das Ende des Satzes mit einem End-of-Sequence-(EOS-)Token markieren (z.B. »<eos>«).

Die englischen Sätze werden umgedreht, bevor sie in den Encoder eingespeist werden. So wird »I drink milk« zu »milk drink I.« Dadurch wird der Beginn des Satzes als Letztes in den Encoder eingegeben. Dies ist nützlich, weil dieser meist das Erste ist, was der Decoder übersetzen muss.

Jedes Wort wird zu Beginn durch eine Integerzahl ausgedrückt (z.B. 288 für das Wort »milk«). Anschließend wird das Embedding für dieses Wort nachgeschlagen (wie oben erläutert, ist dies ein dichter, eher niedrig dimensionierter Vektor). Diese Word Embeddings werden anschließend in den Encoder und den Decoder eingespeist.

Bei jedem Schritt gibt der Decoder für jedes Wort einen Score im Ausgabe-Vokabular (Französisch) aus. Die Softmax-Schicht wandelt diese Scores in Wahrscheinlichkeiten um. Beispielsweise könnte das Wort »Je« beim ersten Schritt eine Wahrscheinlichkeit von 20% haben, »Tu« von 1% und so weiter. Das Wort mit der höchsten Wahrscheinlichkeit wird ausgegeben. Dies funktioniert so ähnlich wie eine gewöhnliche Klassifikationsaufgabe, Sie können das Modell also mit der Funktion `softmax_cross_entropy_with_logits()` trainieren.

Zu beachten ist, dass Sie zum Zeitpunkt der Inferenz (nach dem Trainieren) keine fertige Zielsequenz zum Eingeben in den Decoder haben. Sie können aber einfach das im vorigen Schritt ausgegebene Wort in den Decoder eingeben, wie in Abbildung 14-16 gezeigt (dazu ist ein Nachschlagen des Embedding nötig, das im Diagramm nicht gezeigt ist).

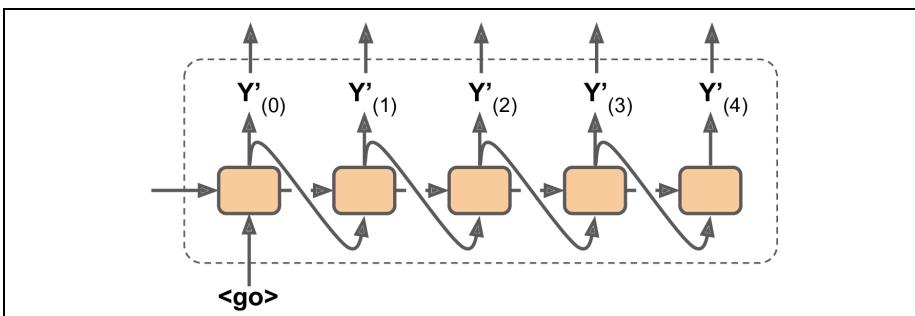


Abbildung 14-16: Eingabe des zuvor ausgegebenen Worts zum Zeitpunkt der Inferenz

Damit haben Sie einen Überblick. Wenn Sie sich durch das TensorFlow-Tutorial Sequence-to-Sequence arbeiten und sich den Code in `rnn/translate/seq2seq_model.py` anschauen (bei den TensorFlow-Modellen (<https://github.com/tensorflow/models>)), sollten Ihnen einige wichtige Unterschiede auffallen:

- Erstens sind wir bisher davon ausgegangen, dass sämtliche Eingabesequenzen (im Encoder und im Decoder) eine konstante Länge haben. Natürlich kann sich die Länge von Sätzen unterscheiden. Es gibt einige Möglichkeiten, damit umzugehen – beispielsweise das Argument `sequence_length` der Funktionen `static_rnn()` und `dynamic_rnn()`, mit denen sich die Länge jedes Satzes angeben lässt (siehe oben). Im Tutorial wird allerdings ein anderer Ansatz verfolgt (vermutlich aus Performancegründen): Sätze werden in Gruppen ähnlicher Länge eingeteilt (z.B. eine Gruppe für Sätze mit 1 bis 6 Wörtern, eine für Sätze mit 7 bis 12 Wörtern und so weiter¹¹), und die kürzeren Sätze werden mit einem Padding Token (z.B. »<pad>«) aufgefüllt. Damit wird beispielsweise »I drink milk« zu »<pad> <pad> <pad> milk drink I«, und die Übersetzung lautet »Je bois du lait <eos> <pad>«. Natürlich möchten wir die Ausgabe hinter dem EOS-Token ignorieren. Dazu wird im Tutorial der Vektor `target_weights` verwendet. Die Gewichte für den Satz »Je bois du lait <eos> <pad>« würden beispielsweise auf [1.0, 1.0, 1.0, 1.0, 1.0, 0.0] gesetzt (das Gewicht 0.0 entspricht dem Padding Token im Zielsatz). Einfaches Multiplizieren der Verlustbeträge der Gewichte mit diesen Gewichten setzt die Verlustbeträge für Wörter nach dem EOS-Token auf null.
- Zweitens wäre das Ausgeben einer Wahrscheinlichkeit für jedes mögliche Wort bei einem großen Ausgabevokabular (was hier der Fall ist) enorm langsam. Wenn das Vokabular sagen wir 50000 französische Wörter enthält, würde der Decoder Vektoren mit 50000 Dimensionen ausgeben. Die Softmax-Funktion über einen derart großen Vektor zu berechnen, würde einen erheblichen Rechenaufwand bedeuten. Um dies zu vermeiden, lässt man den Decoder wesentlich kleinere Vektoren ausgeben, etwa Vektoren mit 1000 Dimensionen, und verwendet dann ein Sampling-Verfahren, um den Verlustbetrag zu schätzen. Damit ist es nicht nötig, diesen für jedes einzelne Wort im Vokabular zu berechnen. Diese Technik namens *Sampled Softmax* wurde im Jahr 2015 von Sébastien Jean et al. vorgestellt (<https://goo.gl/u0GR8k>).¹² In TensorFlow können Sie die Funktion `sampled_softmax_loss()` verwenden.
- Drittens enthält die Implementierung im Tutorial einen *Aufmerksamkeitsmechanismus*, der den Decoder in die Eingabesequenz schauen lässt. Durch Aufmerksamkeit erweiterte RNNs sprengen an dieser Stelle den Rahmen, aber wenn Sie sich dafür interessieren, gibt es einige hilfreiche Artikel zu maschinelner Übersetzung (<https://goo.gl/8RCous>)¹³, maschinellem Lesen (<https://goo.gl/X0Nau8>)¹⁴ und Bildbeschriftungen (<https://goo.gl/xmhvfK>)¹⁵.

¹¹ Die Größen im Tutorial sind andere.

¹² »On Using Very Large Target Vocabulary for Neural Machine Translation«, S. Jean et al. (2015).

¹³ »Neural Machine Translation by Jointly Learning to Align and Translate«, D. Bahdanau et al. (2014).

¹⁴ »Long Short-Term Memory-Networks for Machine Reading«, J. Cheng (2016).

¹⁵ »Show, Attend and Tell: Neural Image Caption Generation with Visual Attention«, K. Xu et al. (2015).

- Schließlich verwendet das Tutorial das Modul `tf.nn.legacy_seq2seq`, das Werkzeuge zum einfachen Erstellen unterschiedlicher Encoder-Decoder-Modelle enthält. Beispielsweise erstellt die Funktion `embedding_rnn_seq2seq()` ein einfaches Encoder-Decoder-Modell, das sich automatisch um Word Embeddings kümmert, wie das in Abbildung 14-15 gezeigte. Dieser Code wird vermutlich bald geändert, um das neuere Modul `tf.nn.seq2seq` zu verwenden.

Sie haben nun alle Werkzeuge beisammen, um die Implementierung des Sequence-to-Sequence-Tutorials nachzuvollziehen. Probieren Sie es aus und trainieren Sie Ihren eigenen Englisch-Französisch-Übersetzer!

Übungen

1. Welche Anwendungen für ein Sequenz-zu-Sequenz-RNN fallen Ihnen ein? Wie sieht es mit einem Sequenz-zu-Vektor-RNN aus? Und einem Vektor-zu-Sequenz-RNN?
2. Warum verwendet man zur automatischen Übersetzung Encoder-Decoder-RNNs anstatt einfacher Sequenz-zu-Sequenz-RNNs?
3. Wie könnten Sie ein Convolutional Neural Network mit einem RNN kombinieren, um Filme zu klassifizieren?
4. Welche Vorteile bietet das Erstellen eines RNN mit `dynamic_rnn()` anstelle von `static_rnn()`?
5. Wie können Sie mit Eingabesequenzen unterschiedlicher Länge umgehen? Wie sieht es bei Ausgabesequenzen unterschiedlicher Länge aus?
6. Was ist ein üblicher Ansatz zum Verteilen von Training und Ausführen eines Deep-RNN auf mehrere GPUs?
7. *Embedded Reber Grammars* wurden von Hochreiter und Schmidhuber in ihrem Artikel über LSTMs verwendet. Dies sind künstliche Grammatiken, die Strings wie »BPBT\$XXVPSEPE« produzieren. Lesen Sie zu diesem Thema die tolle Einführung (<https://goo.gl/7CkNRn>) von Jenny Orr. Wählen Sie eine besonders verschachtelte Reber-Grammatik (wie die auf Jenny Orrs Webseite) und trainieren Sie ein RNN so, dass es erkennt, ob ein String dieser Grammatik entspricht oder nicht. Dazu müssen Sie sich eine Funktion schreiben, die einen Trainingsbatch aus 50% dieser Grammatik entsprechenden Strings generiert, sowie 50% Strings, die ihr nicht entsprechen.
8. Bearbeiten Sie den Kaggle-Wettbewerb (<https://goo.gl/0DS5Xe>) »How much did it rain? II«. Dies ist eine Aufgabe zur Vorhersage einer Zeitreihe: Sie erhalten Schnappschüsse eines polarimetrischen Radars und sollen daraus die stündlichen Regenfälle vorhersagen. Das Interview (<https://goo.gl/fTA90W>) mit Luis Andre Dutra e Silva enthält einige interessante Hinweise zu Techni-

ken, mit denen er den zweiten Platz im Wettbewerb erreichen konnte. Insbesondere verwendete er ein RNN aus zwei LSTM-Schichten.

9. Bearbeiten Sie das TensorFlow-Tutorial zu *Word2Vec* (<https://goo.gl/edArdi>), um Word Embeddings zu erzeugen. Arbeiten Sie anschließend das *Seq2Seq* (<https://goo.gl/L82gvS>)-Tutorial durch, um ein System zur Übersetzung von Englisch ins Französische zu trainieren.

Lösungen zu diesen Übungen finden Sie in Anhang A.

KAPITEL 15

Autoencoder

Autoencoder sind künstliche neuronale Netze, die eine effiziente Repräsentation der Eingabedaten, die *Codings*, ohne jegliche Überwachung erlernen können (d.h., der Trainingsdatensatz enthält keine Labels). Diese Codings haben üblicherweise eine viel niedrigere Dimensionalität als die Eingabedaten, was Autoencoder für die Dimensionsreduktion einsetzbar macht (siehe Kapitel 8). Vor allem aber sind Autoencoder bei der Merkmalserkennung mächtig und lassen sich zum unüberwachten Vortrainieren von Deep-Learning-Netzen einsetzen (wie in Kapitel 11 besprochen). Schließlich können Sie zufällige neue Daten generieren, die den Trainingsdaten sehr ähnlich sind; dies nennt man ein *generatives Modell*. Beispielsweise könnten Sie einen Autoencoder mit Bildern von Gesichtern trainieren, und er würde daraufhin neue Gesichter generieren können.

Überraschenderweise tun Autoencoder nichts weiter, als zu lernen, ihre Eingaben zu den Ausgaben zu kopieren. Dies klingt wie eine triviale Aufgabe, sie wird aber durch zusätzliche Einschränkungen des Netzes absichtlich erschwert. Beispielsweise können Sie die Größe der internen Repräsentation begrenzen oder den Eingabedaten Rauschen beimischen und dann das Netz so trainieren, dass es die ursprünglichen Eingaben wiederherstellt. Solche Bedingungen verhindern, dass der Autoencoder die Eingaben einfach direkt in die Ausgabe kopiert, und zwingen ihn dazu, effiziente Wege zur Repräsentation der Daten zu finden. Kurz, die Codings sind ein Nebenprodukt beim Versuch, die Identitätsfunktion unter erschwerten Bedingungen zu erlernen.

In diesem Kapitel werden wir genauer erklären, wie Autoencoder funktionieren, was für Arten von Bedingungen anwendbar sind und wie sie sich in TensorFlow implementieren lassen, sei es zur Dimensionsreduktion, zur Merkmalserkennung, zum unüberwachten Vortrainieren oder als generatives Modell.

Effiziente Repräsentation von Daten

Welche der Zahlenfolgen ist für Sie am einfachsten zu merken?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

Auf den ersten Blick könnte man denken, dass die erste Folge einfacher sein müsste, da sie viel kürzer ist. Wenn Sie sich jedoch die zweite Zahlenfolge genau ansehen, fällt auf, dass sie zwei einfachen Regeln folgt: Gerade Zahlen werden vom halbierten Wert gefolgt und ungerade vom Dreifachen plus eins (dies ist eine berühmte Folge, die *Hailstone-Folge*). Haben Sie dieses Muster erst einmal erkannt, ist die zweite Folge viel einfacher zu merken, da Sie sich nur noch die zwei Regeln, die erste Zahl und die Länge der Folge merken müssen. Wenn Sie sich lange Zahlenfolgen leicht und schnell merken können, würden Sie sich nicht groß um die Existenz dieses Musters scheren. Sie würden sie einfach auswendig lernen. Der Umstand, dass lange Zahlenfolgen schwierig zu merken sind, macht das Erkennen von Mustern überhaupt erst nützlich. Wir hoffen, dies veranschaulicht, warum zusätzliche Restriktionen beim Trainieren einen Autoencoder veranlassen, Muster in den Daten zu finden und zu nutzen.

Der Zusammenhang zwischen Gedächtnis, Wahrnehmung und Mustererkennung wurde spektakulär von William Chase und Herbert Simon in den frühen 1970ern untersucht (<https://goo.gl/kSNcX0>).¹ Sie hatten beobachtet, dass professionelle Schachspieler sich die Position sämtlicher Figuren innerhalb von fünf Sekunden merken konnten, was die meisten Leute für unmöglich halten würden. Allerdings war dies nur der Fall, wenn die Figuren in realistischen Positionen (aus echten Partien) standen, nicht wenn sie zufällig platziert wurden. Professionelle Schachspieler haben kein besseres Gedächtnis als Sie oder ich, sie können nur dank ihrer Erfahrung die Muster im Schachspiel leichter erkennen. Das Erkennen von Mustern hilft ihnen, diese Information effizient abzulegen.

Wie die Schachspieler in diesem Gedächtnisexperiment sieht sich ein Autoencoder die Eingaben an, konvertiert sie in eine effiziente interne Repräsentation und spuckt dann etwas aus, das (hoffentlich) den ursprünglichen Eingaben ähnlich sieht. Ein Autoencoder besteht immer aus zwei Teilen: einem *Encoder* (oder *Recognition Network*), der die Eingaben in eine interne Repräsentation überführt und einem *Decoder* (oder *generativem Netz*), der die interne Repräsentation in die Ausgabe umwandelt (siehe Abbildung 15-1).

Ein Autoencoder hat also normalerweise die gleiche Architektur wie ein mehrschichtiges Perzeptron (MLP; siehe Kapitel 10), nur dass die Anzahl Neuronen in der Ausgabeschicht mit der Anzahl Eingaben übereinstimmen muss. In diesem Beispiel gibt es nur eine aus zwei Neuronen bestehende verborgene Schicht (den Encoder) und eine aus drei Neuronen bestehende Ausgabeschicht (den Decoder). Die Ausgaben werden häufig als *Rekonstruktionen* bezeichnet, da der Autoencoder versucht, die Eingaben zu rekonstruieren. Dementsprechend berechnet die Kostenfunktion den *Rekonstruktionsverlust*, der das Modell für Abweichungen der Rekonstruktion von den Eingaben bestraft.

¹ »Perception in chess«, W. Chase and H. Simon (1973).

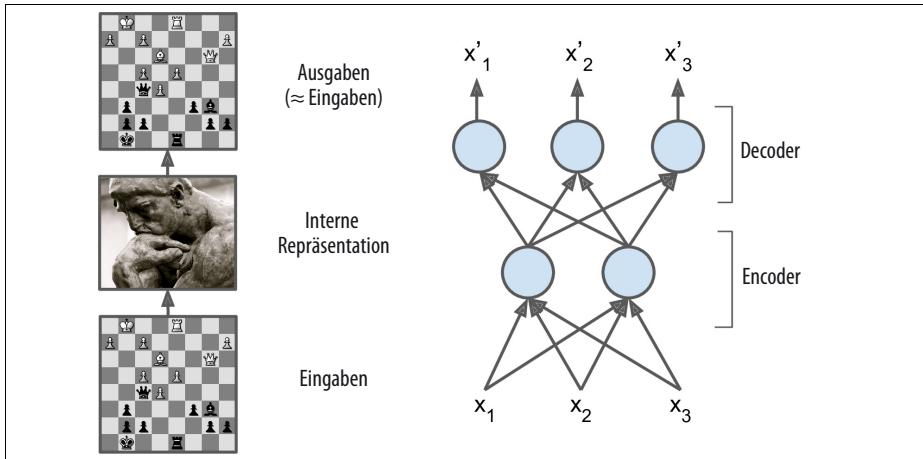


Abbildung 15-1: Das Gedächtnisexperiment mit Schachpartien (links) und ein einfacher Autoencoder (rechts)

Weil die interne Repräsentation eine niedrigere Dimensionalität als die Eingabedaten aufweist (2-D statt 3-D), nennt man diesen Autoencoder *unvollständig*. Ein unvollständiger Autoencoder kann seine Eingaben nicht einfach in die Codierung kopieren, sondern muss eine andere Repräsentation der Eingabedaten erzeugen. Er ist gezwungen, die wichtigsten Merkmale in den Eingabedaten zu erlernen (und die unwichtigen zu verwerfen).

Sehen wir uns an, wie sich ein sehr einfacher unvollständiger Autoencoder zur Dimensionsreduktion implementieren lässt.

Hauptkomponentenzerlegung mit einem unvollständigen linearen Autoencoder

Wenn der Autoencoder nur lineare Aktivierungen verwendet und die Kostenfunktion die mittlere quadratische Abweichung (MSE) ist, lässt sich nachweisen, dass der Autoencoder eine Hauptkomponentenzerlegung durchführt (siehe Kapitel 8).

Der folgende Code erstellt einen einfachen linearen Autoencoder, um eine Hauptkomponentenzerlegung auf einem 3-D-Datensatz und eine Projektion auf 2-D durchzuführen:

```
import tensorflow as tf

n_inputs = 3 # 3-D Eingaben
n_hidden = 2 # 2-D Codierung
n_outputs = n_inputs
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
```

```

hidden = tf.layers.dense(X, n_hidden)
outputs = tf.layers.dense(hidden, n_outputs)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(reconstruction_loss)

init = tf.global_variables_initializer()

```

Dieser Code unterscheidet sich wirklich nicht besonders von den MLPs aus den vergangenen Kapiteln. Zwei Dinge sind dabei zu betonen:

- Die Anzahl der Ausgaben und Eingaben ist identisch.
- Um eine einfache PCA durchzuführen, verwenden wir keine Aktivierungsfunktion (alle Neuronen sind linear), und die MSE wird als Kostenfunktion verwendet. Wir werden in Kürze komplexere Autoencoder sehen.

Laden wir nun den Datensatz, um das Modell auf dem Trainingsdatensatz zu trainieren und es zum Codieren des Testdatensatzes zu verwenden (d.h., auf zwei Dimensionen zu projizieren):

```

X_train, X_test = [...] # lade den Datensatz

n_iterations = 1000
codings = hidden # die Ausgabe der verborgenen Schicht enthält das Coding

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train}) # keine Labels (unüberwacht)
    codings_val = codings.eval(feed_dict={X: X_test})

```

Abbildung 15-2 zeigt den ursprünglichen 3-D-Datensatz (links) und die Ausgabe der verborgenen Schicht des Autoencoders (d.h. die codierende Schicht, rechts). Wie Sie sehen, hat der Autoencoder die bestmögliche 2-D-Ebene zur Projektion gefunden, sodass möglichst viel Varianz in den Daten erhalten bleibt (wie bei einer Hauptkomponentenzerlegung).

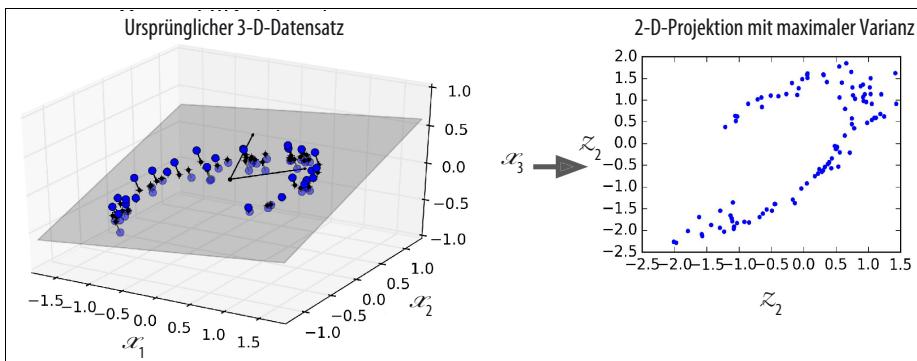


Abbildung 15-2: Von einem unvollständigen linearen Autoencoder durchgeführte PCA

Stacked Autoencoder

Wie andere neuronale Netze können auch Autoencoder mehrere verborgene Schichten besitzen. Diese bezeichnet man als *Stacked Autoencoder* (oder *Deep Autoencoder*). Mit zusätzlichen Schichten kann ein Autoencoder komplexere Codings erlernen. Allerdings darf der Autoencoder auch nicht zu mächtig werden. Stellen Sie sich einen ausreichend mächtigen Autoencoder vor, der jeden Eingabewert einer beliebigen Zahl zuordnen kann (und im Decoder die umgekehrte Zuordnung erlernt). Natürlich wird ein solcher Autoencoder die Trainingsdaten perfekt rekonstruieren, dabei aber keine nützliche Repräsentation der Daten erlernen (und bei neuen Daten kaum verallgemeinern).

Die Architektur eines Stacked Autoencoders ist normalerweise symmetrisch um die zentrale verborgene Schicht (die codierende Schicht) herum angeordnet. Einfach formuliert ähnelt die Architektur einem Sandwich. Ein Autoencoder für den MNIST-Datensatz (aus Kapitel 3) könnte 784 Eingaben haben, gefolgt von einer verborgenen Schicht mit 300 Neuronen, einer zentralen verborgenen Schicht mit 150 Neuronen, einer weiteren Schicht mit 300 Neuronen und schließlich einer Ausgabeschicht mit 784 Neuronen. Ein solcher Stacked Autoencoder ist in Abbildung 15-3 dargestellt.

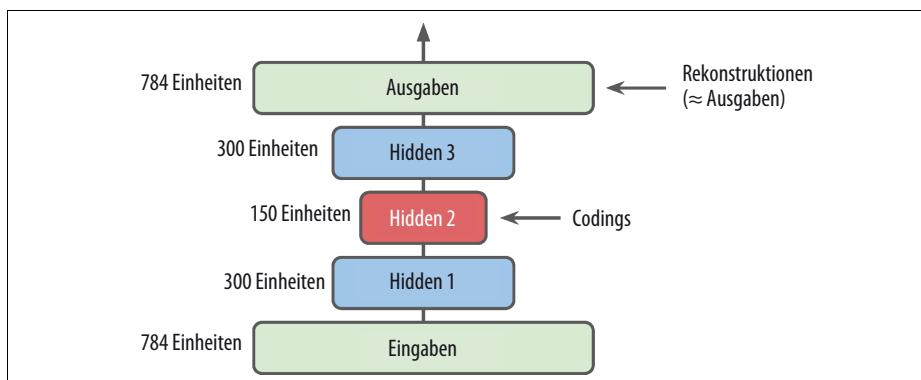


Abbildung 15-3: Stacked Autoencoder

Implementierung mit TensorFlow

Sie können einen Stacked Autoencoder wie ein gewöhnliches tiefes MLP implementieren. Insbesondere können Sie die in Kapitel 11 vorgestellten Techniken zum Trainieren von Deep-Learning-Netzen anwenden. Das folgende Codebeispiel erstellt einen Stacked Autoencoder für die MNIST-Daten. Er verwendet die Initialisierung nach He, die ELU-Aktivierungsfunktion und ℓ_2 -Regularisierung. Der Code sollte Ihnen bekannt vorkommen, außer dass es kein Label y gibt:

```
from functools import partial

n_inputs = 28 * 28 # für MNIST
```

```

n_hidden1 = 300
n_hidden2 = 150 # Codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.0001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

he_init = tf.contrib.layers.variance_scaling_initializer()
l2_regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
my_dense_layer = partial(tf.layers.dense,
                        activation=tf.nn.elu,
                        kernel_initializer=he_init,
                        kernel_regularizer=l2_regularizer)

hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2) # Codings
hidden3 = my_dense_layer(hidden2, n_hidden3)
outputs = my_dense_layer(hidden3, n_outputs, activation=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

```

Dieses Modell lässt sich wie gewohnt trainieren. Es ist zu betonen, dass wir die Labels der Ziffern (y_{batch}) nicht verwenden:

```

n_epochs = 5
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

```

Kopplung von Gewichten

Ist ein Autoencoder wie der soeben erstellte symmetrisch, lassen sich die Gewichte der Decoder- und Encoderschichten miteinander *koppeln*. Damit wird die Anzahl der Gewichte im Modell halbiert, was das Trainieren beschleunigt und das Risiko für Overfitting verringert. Wenn es im Autoencoder insgesamt N Schichten gibt (ohne die Eingabeschicht) und die \mathbf{W}_L für die Gewichte der Verbindungen in der

L -ten Schicht steht (z.B. Schicht 1 die erste verborgene Schicht ist, Schicht $\frac{N}{2}$ die codierende Schicht und Schicht N die Ausgabeschicht), dann lassen sich die Gewichte der decodierenden Schicht einfach durch $\mathbf{W}_{N-L+1} = \mathbf{W}_L^T$ (mit $L = 1, 2, \dots, \frac{N}{2}$) definieren.

Leider sind gekoppelte Gewichte in TensorFlow mit der Funktion `dense()` ein wenig umständlich zu implementieren; es ist letztendlich einfacher, die Schichten manuell zu definieren. Der Code wird dadurch etwas länger:

```

activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3") # tied weights
weights4 = tf.transpose(weights1, name="weights4") # tied weights

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

```

Dieser Code ist recht gradlinig. Es gibt trotzdem einige wichtige Dinge zu anzumerken:

- Erstens sind `weights3` und `weights4` keine Variablen, sondern Transponierungen von `weights2` und `weights1` (sie sind mit diesen »gekoppelt«).
- Zweitens macht es keinen Sinn, sie zu regularisieren, da es keine Variablen sind: Wir regularisieren lediglich `weights1` und `weights2`.
- Drittens sind die Bias-Terme niemals gekoppelt oder regularisiert.

Trainieren mehrerer Autoencoder nacheinander

Anstatt den gesamten Stacked Autoencoder wie eben gezeigt in einem Durchgang zu trainieren, ist es meist schneller, einen einschichtigen Autoencoder nach dem anderen zu trainieren. Anschließend lassen sich diese zu einem Stacked Autoencoder aufstapeln (daher der Name), wie Abbildung 15-4 zeigt. Dies ist besonders für sehr tiefe Autoencoder geeignet.

In der ersten Trainingsphase lernt der erste Autoencoder, die Eingaben zu rekonstruieren. In der zweiten Phase lernt der zweite Autoencoder, die Ausgabe der ersten verborgenen Schicht zu rekonstruieren. Am Ende erstellen Sie einfach das in Abbildung 15-4 dargestellte große Sandwich aus allen Autoencodern (Sie stapeln also zuerst die verborgenen Schichten der Autoencoder nacheinander auf, gefolgt von den Ausgabeschichten in umgekehrter Reihenfolge). Damit erhalten Sie den endgültigen Stacked Autoencoder. Sie können mit dieser Methode weitere Autoencoder trainieren, sodass ein sehr tiefer Stacked Autoencoder entsteht.

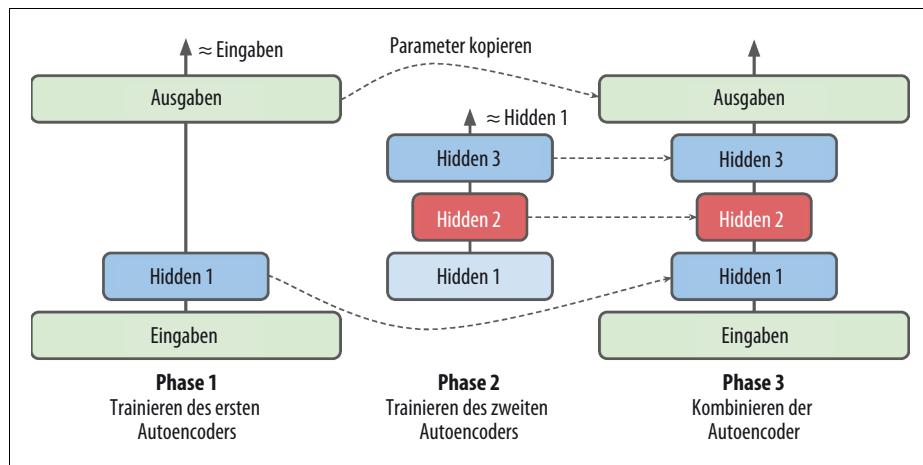


Abbildung 15-4: Trainieren mehrerer Autoencoder nacheinander

Der einfachste Ansatz zum Implementieren dieses mehrphasigen Trainingsalgorithmus ist, in jeder Phase einen anderen TensorFlow-Graphen zu verwenden. Nach dem Trainieren eines Autoencoders können Sie einfach den Trainingsdatensatz durchlaufen lassen und die Ausgaben der verborgenen Schicht aufzeichnen. Diese Ausgaben dienen dem nächsten Autoencoder als Trainingsdaten. Sobald sämtliche Autoencoder auf diese Weise trainiert wurden, können Sie die Gewichte und Biases aus jedem Autoencoder kopieren und den Stacked Autoencoder damit erstellen. Diese Implementierung ist recht gradlinig, weswegen wir sie hier nicht aufführen möchten, Sie finden sie aber als Codebeispiel in den *Jupyter Notebooks* (<https://github.com/ageron/handson-ml>).

Ein weiterer Ansatz ist, einen einzigen Graphen mit dem gesamten Stacked Autoencoder und einige zusätzliche Operationen für die jeweiligen Trainingsphasen zu definieren, wie in Abbildung 15-5 gezeigt.

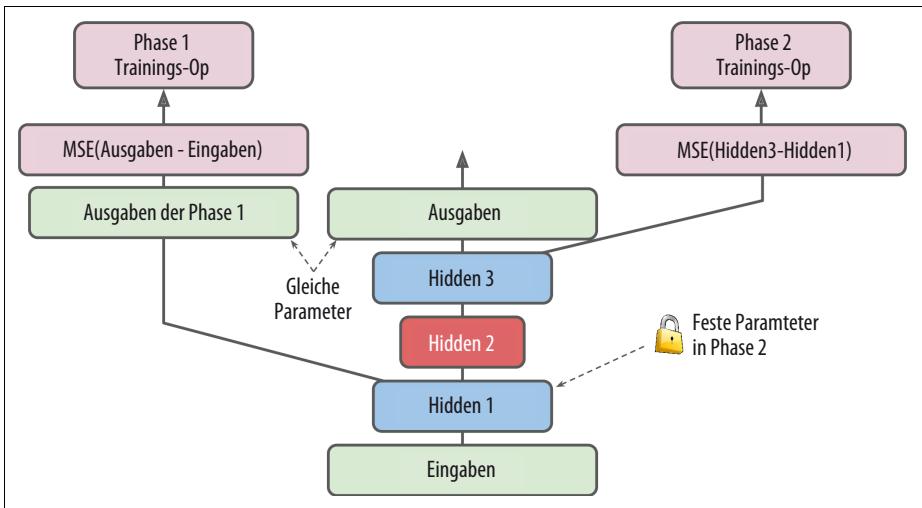


Abbildung 15-5: Ein einzelner Graph zum Trainieren eines Stacked Autoencoders

Dies bedarf einiger Erklärungen:

- Die mittlere Spalte des Graphen ist der vollständige Stacked Autoencoder. Dieser Teil lässt sich nach dem Trainieren verwenden.
- Die linke Spalte enthält die für die erste Trainingsphase nötigen Operationen. Sie erstellt eine Ausgabeschicht, die die verborgenen Schichten 2 und 3 umgeht. Diese Ausgabeschicht enthält die gleichen Gewichte und Biases wie die Ausgabeschicht des Stacked Autoencoders. Darüber befinden sich die Trainingsoperationen, mit denen sich die Ausgaben den Eingaben so weit wie möglich annähern lassen. Damit werden in dieser Phase die Gewichte und Biases der verborgenen Schicht 1 und der Ausgabeschicht trainiert (und damit der erste Autoencoder).
- Die rechte Spalte des Graphen enthält die Operationen für die zweite Trainingsphase. Sie fügt die Trainingsoperationen hinzu, die darauf abzielen, die Ausgabe der versteckten Schicht 3 so nahe wie möglich an die Ausgabe der versteckten Schicht 1 zu bringen. Beachten Sie, dass wir die verborgene Schicht 1 in Phase 2 einfrieren müssen. Diese Phase trainiert die Gewichte und Bias-Terme der verborgenen Schichten 2 und 3 (also den zweiten Autoencoder).

Der TensorFlow-Code dazu sieht folgendermaßen aus:

```
[...] # Erstelle den gesamten Stacked Autoencoder wie gewohnt.  
# In diesem Beispiel sind die Gewichte nicht gekoppelt.
```

```
optimizer = tf.train.AdamOptimizer(learning_rate)
```

```

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss = tf.reduce_mean(tf.square(phase1_outputs - X))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss = tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op = optimizer.minimize(phase2_loss, var_list=train_vars)

```

Die erste Phase ist recht einfach: Wir erstellen eine Ausgabeschicht, die die verborgenen Schichten 2 und 3 auslässt. Anschließend definieren wir den Trainingsvorgang zum Minimieren des Unterschieds zwischen Ein- und Ausgabe (sowie ein wenig Regularisierung).

In der zweiten Phase fügen wir einfach die Operationen zum Minimieren des Abstands zwischen den Ausgaben der verborgenen Schichten 3 und 1 hinzu (ebenfalls mit Regularisierung). Vor allem aber übergeben wir der Methode `minimize()` eine Liste der trainierbaren Variablen, wobei wir `weights1` und `biases1` absichtlich auslassen; dadurch frieren wir die verborgene Schicht 1 in Phase 2 ein.

Während der Ausführungsphase müssen Sie lediglich die Trainingsoperation für Phase 1 einige Epochen lang ausführen, anschließend die Trainingsoperation für Phase 2 für einige weitere Epochen.



Da die verborgene Schicht 1 während Phase 2 eingefroren ist, ist ihre Ausgabe für einen bestimmten Trainingsdatenpunkt stets die gleiche. Um eine mehrmalige Berechnung der Ausgabe dieser Schicht während jeder Epoche zu vermeiden, können Sie diese am Ende von Phase 1 für den gesamten Trainingsdatensatz vorab berechnen, speichern und in Phase 2 als Eingabe verwenden. So erhalten Sie einen ordentlichen Geschwindigkeitsbonus.

Visualisieren der Rekonstruktionen

Um sicherzustellen, dass ein Autoencoder gut trainiert ist, können Sie die Ein- und Ausgaben miteinander vergleichen. Diese sollten recht ähnlich zueinander sein, und die Unterschiede sollten aus unwichtigen Details bestehen. Zeichnen wir deshalb zwei zufällig ausgewählte Ziffern und deren Rekonstruktionen:

```

n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
    [...] # Trainieren des Autoencoders
    outputs_val = outputs.eval(feed_dict={X: X_test})

def plot_image(image, shape=[28, 28]):

```

```

plt.imshow(image.reshape(shape), cmap="Greys", interpolation="nearest")
plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])

```

Abbildung 15-6 zeigt die sich ergebenden Bilder.

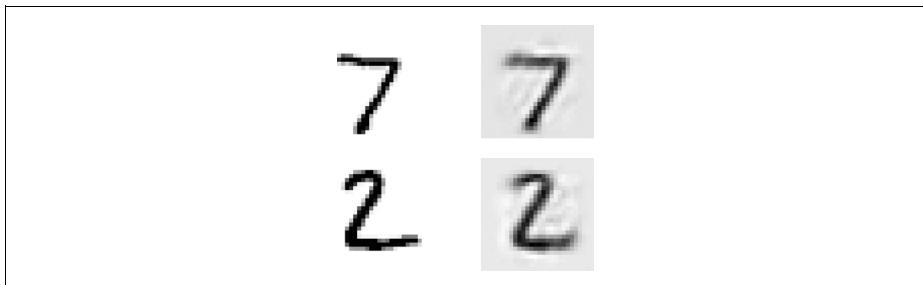


Abbildung 15-6: Ursprüngliche Ziffern (links) und ihre Rekonstruktionen (rechts)

Die Bilder sehen sich recht ähnlich. Der Autoencoder hat also gelernt, die Eingaben zu reproduzieren, aber hat er auch nützliche Merkmale erlernt? Diese untersuchen wir als Nächstes.

Visualisieren von Merkmalen

Sobald Ihr Autoencoder einige Merkmale erlernt hat, sollten Sie sich diese ansehen. Dazu gibt es verschiedene Techniken. Die einfachste ist, für jedes Neuron in jeder verborgenen Schicht die Trainingsdatenpunkte zu finden, die es am stärksten aktivieren. Dies ist besonders bei den mittleren verborgenen Schichten nützlich, da diese meist komplexe Merkmale erfassen, die Sie leicht in mehreren Trainingsdatenpunkten wiedererkennen können. Wenn beispielsweise ein Neuron stark aktiviert wird, sobald sich eine Katze im Bild befindet, können Sie dies beim Überprüfen mehrerer Katzenbilder leicht erkennen. Bei den ersten Schichten funktioniert diese Technik jedoch nicht so gut, da die Merkmale kleinteiliger und abstrakter sind. Es ist daher schwer zu verstehen, weshalb genau ein Neuron so aufgeregt ist.

Schauen wir uns eine weitere Technik an. Für jedes Neuron in der ersten verborgenen Schicht können Sie ein Bild erstellen, in dem die Intensität eines Pixels dem Gewicht der Verbindung zu diesem Neuron entspricht. Beispielsweise plottet der folgende Code die von fünf Neuronen in der ersten verborgenen Schicht erlernten Merkmale:

```

with tf.Session() as sess:
    [...] # trainieren des Autoencoders
    weights1_val = weights1.eval()

```

```

for i in range(5):
    plt.subplot(1, 5, i + 1)
    plot_image(weights1_val.T[i])

```

Sie erhalten kleinteilige Merkmale wie die in Abbildung 15-7 gezeigten.

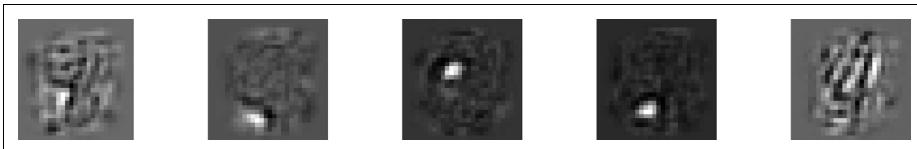


Abbildung 15-7: Von fünf Neuronen in der ersten verborgenen Schicht erlernte Merkmale

Die ersten vier Merkmale scheinen kleinen Bildausschnitten zu entsprechen, der fünfte scheint nach vertikalen Strichen zu suchen (diese Merkmale sind dem Stacked Denoising Autoencoder entnommen, den wir in Kürze besprechen).

Eine dritte Technik ist, ein zufälliges Bild in den Autoencoder einzuspeisen, die Aktivierung eines für uns interessanten Neurons zu messen und dann mit dem Backpropagation-Verfahren das Bild in Richtung einer noch stärkeren Aktivierung dieses Neurons zu verändern. Wenn Sie dies mehrfach wiederholen (mit dem Gradientenverfahren), wird das Bild sich nach und nach in das (für dieses Neuron) aufregendste Bild verwandeln. Diese Technik eignet sich, um die Eingaben zu visualisieren, nach denen ein Neuron sucht.

Schließlich können Sie beim Anwenden eines Autoencoders zum unüberwachten Vortrainieren – etwa bei einer Klassifikationsaufgabe – verifizieren, dass die vom Autoencoder erlernten Merkmale nützlich sind, indem Sie die Genauigkeit des Klassifikators bestimmen.

Unüberwachtes Vortrainieren mit Stacked Autoencoder

Wie in Kapitel 11 besprochen, lassen sich komplexe überwachte Lernaufgaben auch mit wenig gelabelten Trainingsdaten bewältigen, indem Sie die ersten Schichten eines neuronalen Netzes für eine ähnliche Aufgabe übernehmen. Damit können Sie leistungsfähige Modelle mit nur wenigen Trainingsdaten trainieren, weil das Netz nicht alle kleinteiligen Merkmale erlernen muss; es verwendet die Merkmalserkennung aus dem bestehenden Netz.

In ähnlicher Weise können Sie auch einen großen Datensatz verwenden, der nur wenige Labels enthält. Dazu trainieren Sie einen Stacked Autoencoder mit sämtlichen Daten, bauen die ersten Schichten in ein neuronales Netz für die eigentliche Aufgabe ein und trainieren es mit den gelabelten Daten. Das Beispiel in Abbildung 15-8 zeigt, wie Sie einen Stacked Autoencoder zum unüberwachten Vortrainieren eines neuronalen Netzes zur Klassifikation einsetzen können. Der Stacked Autoen-

coder wird wie besprochen Schicht für Schicht trainiert. Beim Trainieren des Klassifikators sollten Sie die vortrainierten Schichten einfrieren (zumindest die ersten).

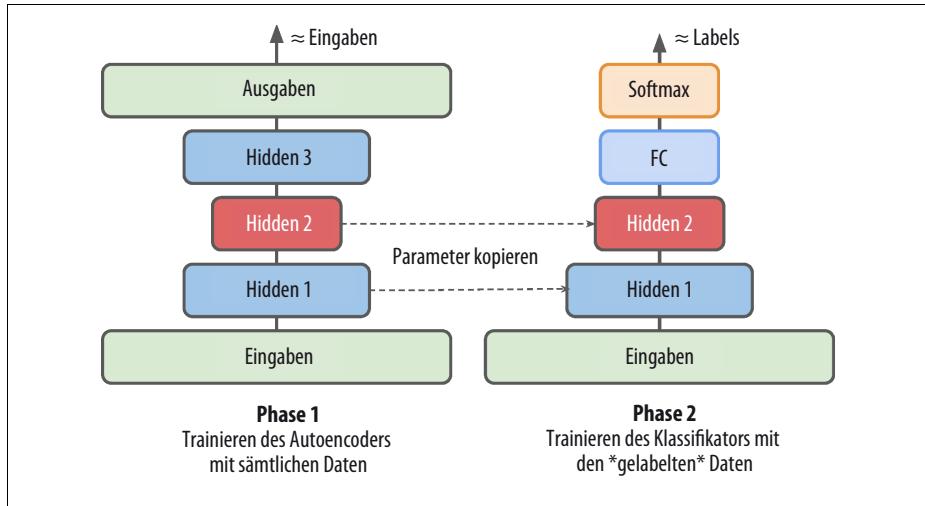


Abbildung 15-8: Unüberwachtes Vtrainieren mit Autoencodern



Diese Situation tritt häufig ein, da sich ein großer Datensatz ohne Labels meist kostengünstig erstellen lässt (z.B. kann ein einfaches Skript Millionen Bilder aus dem Internet herunterladen). Allerdings können nur Menschen die Labels zuverlässig anbringen (z.B. Bilder als niedlich oder nicht niedlich einstufen). Deshalb ist das Labeln von Datenpunkten zeitaufwendig und teuer, und meist hat man nur einige Tausend Datenpunkte mit Labels zur Verfügung.

Wie bereits erwähnt, war einer der Auslöser für die gegenwärtige Riesenwelle im Deep-Learning-Bereich die Entdeckung von Geoffrey Hinton et al. im Jahr 2006, dass sich Deep-Learning-Netze unüberwacht vortrainieren lassen. Dazu wurden zunächst Restricted Boltzmann Machines eingesetzt (siehe Anhang E). Im Jahr 2007 zeigten Yoshua Bengio et al. (<https://goo.gl/R5L7HJ>)² aber, dass Autoencoder ebenso gut funktionieren.

Bei der Implementierung mit TensorFlow gibt es keine Besonderheiten: Sie trainieren einfach einen Autoencoder mit sämtlichen Trainingsdaten. Anschließend erstellen Sie die Schichten des Encoders, um ein neues neuronales Netz zu erstellen (Details zum Wiederverwenden vortrainierter Schichten finden Sie in Kapitel 11 und in den als Jupyter Notebooks verfügbaren Codebeispielen).

2 »Greedy Layer-Wise Training of Deep Networks«, Y. Bengio et al. (2007).

Bisher haben wir die Größe der codierenden Schicht begrenzt, um den Autoencoder zum Erlernen interessanter Eigenschaften zu zwingen und ihn unvollständig zu machen. Es sind viele weitere Beschränkungen möglich. Bei einigen davon kann die codierende Schicht genauso groß wie die Eingabe sein oder noch größer. Ein solcher Autoencoder ist *übergelöst*. Betrachten wir einige dieser Ansätze genauer.

Denoising Autoencoder

Ein Autoencoder lässt sich auch durch das Hinzufügen von Rauschen zur Eingabe dazu bewegen, nützliche Merkmale zu erlernen. Dabei trainiert man ihn darauf, die ursprünglichen, rauschfreien Eingaben zu erkennen. Dies verhindert, dass der Autoencoder einfach die Eingabe in die Ausgabe kopiert. Stattdessen muss er Muster in den Daten finden.

Schon seit den 1980ern gibt es die Idee, Autoencoder als Rauschfilter einzusetzen (z.B. wird sie in der Masterarbeit von Yann LeCun aus dem Jahr 1987 erwähnt). In einem Artikel aus dem Jahr 2008 (<https://goo.gl/K9pqcx>)³ zeigen Pascal Vincent et al., dass Autoencoder sich auch zum Extrahieren von Merkmalen einsetzen lassen. In einem Artikel aus dem Jahr 2010 (<https://goo.gl/HgCDIA>)⁴ erwähnen Vincent et al. erstmalig *Stacked Denoising Autoencoder*.

Das Rauschen kann dabei reines zur Eingabe addiertes normalverteiltes Rauschen sein oder wie bei der Drop-out-Methode (aus Kapitel 11) zufällig deaktivierte Eingaben. Abbildung 15-9 zeigt beide Möglichkeiten.

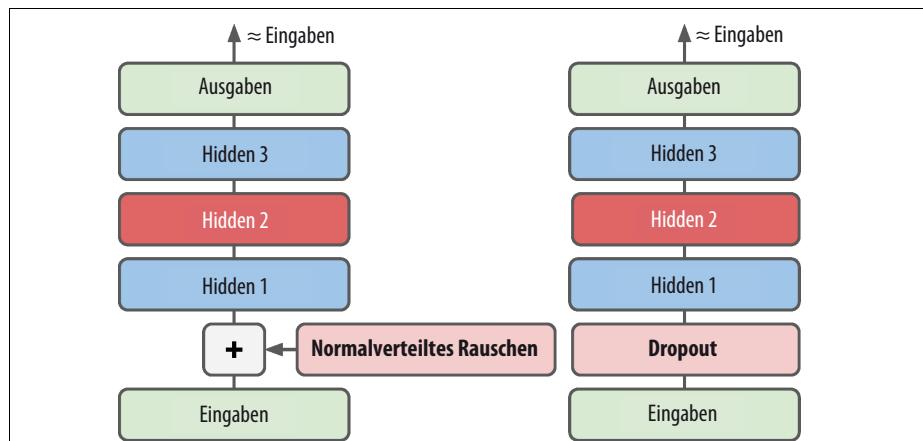


Abbildung 15-9: Denoising Autoencoder mit normalverteiltem Rauschen (links) oder Drop-out (rechts)

3 »Extracting and Composing Robust Features with Denoising Autoencoders«, P. Vincent et al. (2008).

4 »Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion«, P. Vincent et al. (2010).

Implementierung mit TensorFlow

Einen Denoising Autoencoder in TensorFlow zu implementieren, ist nicht besonders schwierig. Beginnen wir mit dem normalverteilten Rauschen. Es ähnelt wirklich sehr dem Trainieren eines gewöhnlichen Autoencoders, außer dass Sie den Eingaben etwas Rauschen hinzufügen und den Rekonstruktionsverlust anhand der ursprünglichen Eingabedaten berechnen:

```
noise_level = 1.0
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + noise_level * tf.random_normal(tf.shape(X))

hidden1 = tf.layers.dense(X_noisy, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```



Da die Abmessungen von X in der Konstruktionsphase nur teilweise definiert sind, können wir die Abmessungen des zu X zu addierenden Rauschens nicht im Voraus kennen. Wir können $X.get_shape()$ nicht aufrufen, weil wir damit nur die unvollständigen Abmessungen von X ([None, n_inputs]) erhalten würden. Die Funktion `random_normal()` erwartet aber vollständige Dimensionen und würde daher einen Ausnahmefehler erzeugen. Stattdessen rufen wir `tf.shape(X)` auf, wodurch wir eine Operation zum Berechnen der Abmessungen von X in der Ausführungsphase definieren. Zu diesem Zeitpunkt sind die Abmessungen vollständig bekannt.

Die etwas verbreiterte Version mit Drop-out ist nicht viel komplizierter:

```
dropout_rate = 0.3

training = tf.placeholder_with_default(False, shape=(), name='training')

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

Beim Trainieren müssen wir mithilfe von `feed_dict` die Variable `training` auf `True` setzen (wie in Kapitel 11 erläutert):

```
sess.run(training_op, feed_dict={X: X_batch, training: True})
```

Beim Testen muss `training` nicht unbedingt auf `False` gesetzt werden, weil dies automatisch beim Aufruf der Funktion `placeholder_with_default()` geschieht.

Sparse Autoencoder

Eine andere beim Ermitteln von Merkmalen hilfreiche Beschränkung ist *Spärlichkeit*: Durch Hinzufügen eines Terms zur Kostenfunktion wird der Autoencoder gezwungen, die Anzahl aktiver Neuronen in der codierenden Schicht zu reduzieren. Beispielsweise ließe sich erzwingen, dass im Durchschnitt nur 5% der Neuronen in der codierenden Schicht aktiv sind. Dadurch ist der Autoencoder gezwungen, jede Eingabe als Kombination einer geringen Anzahl Aktivierungen zu repräsentieren. Dadurch repräsentiert am Ende jedes Neuron ein nützliches Merkmal (wenn Sie nur ein paar Wörter pro Monat sprechen dürften, würden Sie diese auch gut wählen, sodass sich das Zuhören lohnt).

Um spärliche Modelle zu erhalten, müssen wir zunächst die Spärlichkeit der codierenden Schicht bei jedem Trainingsschritt bestimmen. Dazu berechnen wir die durchschnittliche Aktivierung jedes Neurons der codierenden Schicht über den gesamten Trainings-Batch. Die Größe des Batches darf nicht zu klein sein, sonst werden die Mittelwerte ungenau.

Sobald wir die mittlere Aktivierung pro Neuron ermittelt haben, sollten wir allzu aktive Neuronen abstrafen, indem wir den *Spärlichkeitsverlust* zur Kostenfunktion addieren. Wenn wir beispielsweise für ein Neuron eine durchschnittliche Aktivierung von 0.3 messen, die gewünschte Spärlichkeit jedoch 0.1 beträgt, muss es einen Strafterm erhalten, um weniger stark aktiviert zu werden. Dazu könnten wir einfach die quadratische Abweichung zur Kostenfunktion addieren $(0.3 - 0.1)^2$. In der Praxis hat sich aber die Kullback-Leibler-Divergenz (kurz in Kapitel 4 erwähnt) als praktischer erwiesen, da sie steilere Gradienten als die mittlere quadratische Abweichung besitzt, wie Sie Abbildung 15-10 entnehmen können.

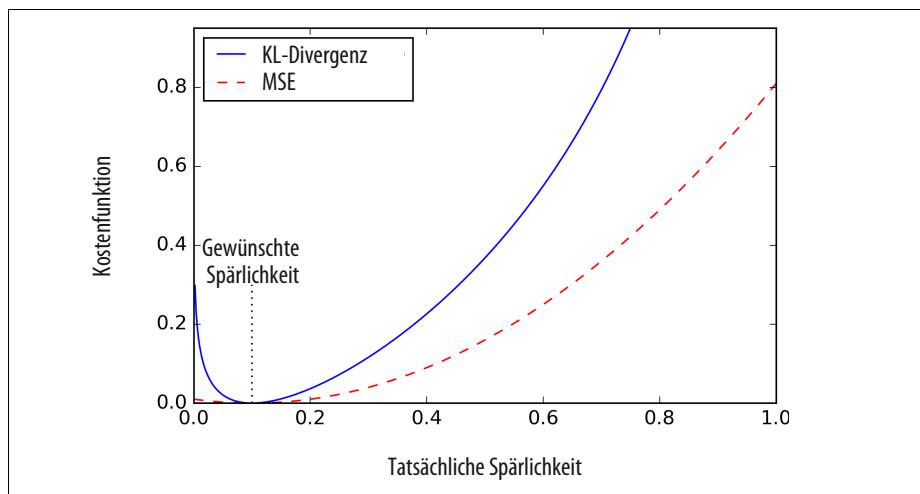


Abbildung 15-10: Spärlichkeitsverlust

Mit zwei diskreten Wahrscheinlichkeitsverteilungen P und Q lässt sich die KL-Divergenz zwischen diesen Verteilungen $D_{\text{KL}}(P \parallel Q)$ nach Formel 15-1 bestimmen.

Formel 15-1: Kullback-Leibler-Divergenz

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In unserem Fall möchten wir die Divergenz zwischen der gewünschten Aktivierungswahrscheinlichkeit p eines Neurons der codierenden Schicht und der tatsächlichen Aktivierungswahrscheinlichkeit q (der mittleren Aktivierung im Trainings-Batch) berechnen. Damit vereinfacht sich die KL-Divergenz zu Formel 15-2.

Formel 15-2: KL-Divergenz zwischen der gewünschten Spärlichkeit p und der gemessenen Spärlichkeit q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Haben wir erst einmal den Spärlichkeitsverlust jedes Neurons der codierenden Schicht berechnet, müssen wir lediglich diese Werte aufsummieren und zur Kostenfunktion addieren. Um den Spärlichkeitsverlust und den Rekonstruktionsverlust gegeneinander zu gewichten, multiplizieren wir den Spärlichkeitsverlust mit einem Hyperparameter. Ist dieser Wichtungsparameter zu groß, hält sich das Modell strikt an die vorgegebene Spärlichkeit, kann aber die Eingaben nicht mehr reproduzieren. Ist der Hyperparameter zu klein, ignoriert das Modell die Spärlichkeit weitgehend und erlernt keine nützlichen Merkmale. In beiden Fällen wäre das Modell nutzlos.

Implementierung in TensorFlow

Wir haben nun alles beisammen, was wir zum Implementieren eines Sparse Autoencoders mit TensorFlow benötigen:

```
def kl_divergence(p, q):
    return p * tf.log(p / q) + (1 - p) * tf.log((1 - p) / (1 - q))

learning_rate = 0.01
sparsity_target = 0.1
sparsity_weight = 0.2
[...] # Erstelle einen gewöhnlichen Autoencoder
      # (in diesem Beispiel ist hidden1 die codierende Schicht)

hidden1_mean = tf.reduce_mean(hidden1, axis=0) # Mittelwert des Batches
sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target, hidden1_mean))
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
loss = reconstruction_loss + sparsity_weight * sparsity_loss
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)
```

Es ist anzumerken, dass die Aktivierung der codierenden Schicht zwischen 0 und 1 liegen muss (aber nicht gleich 0 oder 1). Andernfalls würde die KL-Divergenz NaN (Not a Number) betragen. Eine einfache Lösung bietet das Verwenden der logistischen Aktivierungsfunktion in der codierenden Schicht:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.sigmoid)
```

Ein einfacher Trick beschleunigt das Konvergieren: Anstatt der mittleren quadratischen Abweichung verwenden wir einen Rekonstruktionsverlust, der steilere Gradienten aufweist. Die Kreuzentropie ist hierfür oft eine gute Wahl. Um sie zu verwenden, müssen wir die Eingaben auf Werte zwischen 0 und 1 normalisieren und in der Ausgabeschicht die logistische Aktivierungsfunktion verwenden, sodass auch die Ausgaben Werte zwischen 0 und 1 annehmen. Die TensorFlow-Funktion `sigmoid_cross_entropy_with_logits()` kümmert sich um die effiziente Anwendung der logistischen (sigmoide) Aktivierungsfunktion auf die Ausgaben und das Berechnen der Kreuzentropie:

```
[...]  
logits = tf.layers.dense(hidden1, n_outputs)  
outputs = tf.nn.sigmoid(logits)  
  
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits)  
reconstruction_loss = tf.reduce_sum(xentropy)
```

Die Operation `outputs` wird beim Trainieren nicht benötigt (wir verwenden sie erst, wenn wir die Rekonstruktionen betrachten).

Variational Autoencoder

Eine weitere wichtige Art Autoencoder wurde im Jahr 2014 von Diederik Kingma und Max Welling erstmalig erwähnt (<https://goo.gl/NZq7r2>)⁵ und erfreute sich schnell großer Beliebtheit: die *Variational Autoencoder*.

Diese unterscheiden sich stark von den bisher besprochenen Autoencodern:

- Es sind *probabilistische Autoencoder*, ihre Ausgabe wird auch nach dem Trainieren zum Teil zufällig festgelegt (im Gegensatz zu Denoising Autoencodern, die nur beim Trainieren ein Zufallselement enthalten).
- Vor allem sind es *generative Autoencoder*, die neue Daten generieren, die so aussehen, als kämen sie aus dem Trainingsdatensatz.

Diese beiden Eigenschaften rücken sie in die Nähe von RBMs (siehe Anhang E), aber sie lassen sich einfacher trainieren, und der Generierungsprozess ist viel schneller (bei RBMs müssen Sie warten, bis sich das Netzwerk in einem »thermischen Gleichgewicht« eingependelt, bevor Sie neue Daten generieren können).

⁵ »Auto-Encoding Variational Bayes«, D. Kingma and M. Welling (2014).

Sehen wir uns ihre Funktionsweise an. Abbildung 15-11 (links) stellt einen Variational Autoencoder dar. Sie können dort natürlich die Grundstruktur aller Autoencoder erkennen. Auf den Encoder folgt ein Decoder (in diesem Beispiel haben beide zwei verborgene Schichten), es gibt aber eine Besonderheit: Anstatt aus einer Eingabe direkt ein Coding herzustellen, berechnet der Encoder ein *mittleres Coding* μ und dessen Standardabweichung σ . Das tatsächliche Coding wird dann zufällig als Stichprobe einer Normalverteilung mit dem Mittelwert μ und der Standardabweichung σ entnommen. Nach diesem Schritt decodiert der Decoder das so erstellte Coding wie gewohnt. Die rechte Seite der Abbildung zeigt, wie ein Trainingsdatenpunkt diesen Autoencoder durchläuft. Zuerst erstellt der Encoder μ und σ , anschließend wird zufällig ein Coding generiert (beachten Sie, dass es nicht exakt bei μ liegt). Schließlich wird dieses Coding decodiert, und die endgültige Ausgabe entspricht dem Trainingsdatenpunkt.

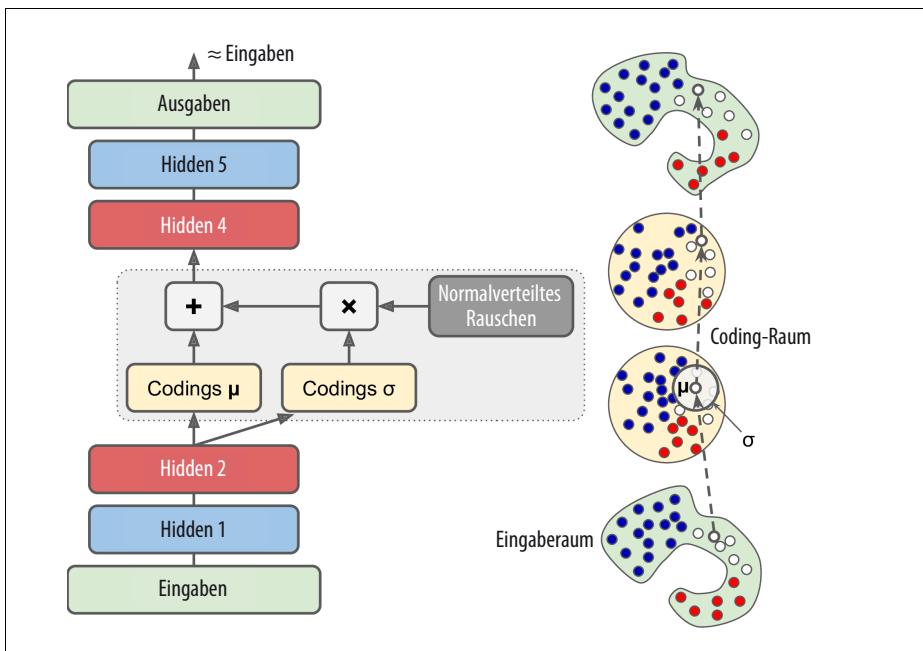


Abbildung 15-11: Variational Autoencoder (links) und ein von diesem verarbeiteter Datenpunkt (rechts)

Wie Sie im Diagramm erkennen können, entsprechen die von einem Variational Autoencoder produzierten Codings der Stichprobe einer gewöhnlichen Normalverteilung, auch wenn die Eingaben einer sehr verworrenen Verteilung folgen:⁶ Während des Trainierens drückt die Kostenfunktion (folgt in Kürze) die Codings allmählich in Richtung des Coderraums (auch als *latenter Raum* bezeichnet), eines

⁶ Variational Autoencoder sind noch allgemeiner; die Codings sind nicht auf Normalverteilungen beschränkt.

grob (hyper-)sphärischen Bereichs, der wie eine Wolke normalverteilter Punkte aussieht. Ein großer Vorteil hiervon ist, dass Sie nach dem Trainieren eines Variational Autoencoders sehr leicht neue Datenpunkte generieren können: Sie ziehen ein zufälliges Coding aus der Normalverteilung und decodieren es, voilà!

Betrachten wir als Nächstes die Kostenfunktion. Diese besteht aus zwei Teilen. Der erste Teil ist der gewöhnliche Rekonstruktionsverlust, der den Autoencoder zum Reproduzieren der Eingabe antreibt (wir verwenden hier wie oben besprochen die Kreuzentropie). Der zweite Teil ist der *latente Verlust*, der den Autoencoder in Richtung von Codings in Form einer Normalverteilung bewegt. Hierfür verwenden wir die KL-Divergenz zwischen der gewünschten Verteilung (der Normalverteilung) und der tatsächlichen Verteilung der Codings. Die Mathematik dahinter ist ein wenig komplizierter als bisher, insbesondere wegen des normalverteilten Rausschens, das die zur codierenden Schicht transportierte Informationsmenge begrenzt (und den Autoencoder zum Erlernen nützlicher Merkmale bewegt). Glücklicherweise vereinfachen sich die Gleichungen zum folgenden Code für den latenten Verlust:⁷

```
eps = 1e-10 # Glättungsterm zum Vermeiden von log(0), was NaN ergäbe
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

Eine verbreitete Variante ist, den Encoder zum Ausgeben von $\gamma = \log(\sigma^2)$ anstatt σ zu bewegen. Immer wenn wir σ benötigen, können wir $\sigma = \exp\left(\frac{\gamma}{2}\right)$ berechnen. Dies erleichtert es dem Encoder, sigmas auf unterschiedlichen Skalen zu erfassen, was die Konvergenz beschleunigt. Der latente Verlust vereinfacht sich damit:

```
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
```

Der folgende Code erstellt den in Abbildung 15-11 (links) gezeigten Variational Autoencoder in der Variante mit $\log(\sigma^2)$:

```
from functools import partial

n_inputs = 28 * 28
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20 # Codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs
learning_rate = 0.001

initializer = tf.contrib.layers.variance_scaling_initializer()
my_dense_layer = partial(
```

⁷ Die mathematischen Details finden Sie im ursprünglichen Artikel zu Variational Autoencodern und im großartigen Tutorial (<https://goo.gl/ViiAzQ>) von Carl Doersch (2016).

```

tf.layers.dense,
activation=tf.nn.elu,
kernel_initializer=initializer)

X = tf.placeholder(tf.float32, [None, n_inputs])
hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2)
hidden3_mean = my_dense_layer(hidden2, n_hidden3, activation=None)
hidden3_gamma = my_dense_layer(hidden2, n_hidden3, activation=None)
noise = tf.random_normal(tf.shape(hidden3_gamma), dtype=tf.float32)
hidden3 = hidden3_mean + tf.exp(0.5 * hidden3_gamma) * noise
hidden4 = my_dense_layer(hidden3, n_hidden4)
hidden5 = my_dense_layer(hidden4, n_hidden5)
logits = my_dense_layer(hidden5, n_outputs, activation=None)
outputs = tf.sigmoid(logits)
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits)
reconstruction_loss = tf.reduce_sum(xentropy)
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
loss = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

Generieren von Ziffern

Nun verwenden wir diesen Variational Autoencoder, um Bilder zu generieren, die wie handgeschriebene Ziffern aussehen. Dazu müssen wir lediglich das Modell trainieren, anschließend zufällige Codings aus einer Normalverteilung ermitteln und diese decodieren.

```

import numpy as np

n_digits = 60
n_epochs = 50
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

    codings_rnd = np.random.normal(size=[n_digits, n_hidden3])
    outputs_val = outputs.eval(feed_dict={hidden3: codings_rnd})

```

Das war alles. Nun sehen Sie die vom Autoencoder »handgeschriebenen« Ziffern (siehe Abbildung 15-12):

```

for iteration in range(n_digits):
    plt.subplot(n_digits, 10, iteration + 1)
    plot_image(outputs_val[iteration])

```



Abbildung 15-12: Von einem Variational Autoencoder generierte Bilder handgeschriebener Ziffern

Ein Großteil dieser Ziffern sieht sehr überzeugend aus, einige andere wirken etwas »kreativ.« Seien Sie aber mit dem Autoencoder nicht zu streng – er hat vor weniger als einer Stunde mit dem Lernen begonnen. Mit etwas mehr Trainingszeit verbessern sich die Ziffern immer mehr.

Weitere Autoencoder

Die faszinierenden Erfolge überwachten Lernens bei Bilderkennung, Spracherkennung, Textübersetzung und weiteren Gebieten haben das unüberwachte Lernen ein wenig überschattet. Tatsächlich findet aber ein Boom statt. Es werden regelmäßig neue Architekturen für Autoencoder und andere unüberwachte Lernalgorithmen entwickelt, zu viele, um sie in diesem Buch zu besprechen. Hier ist ein kurzer (keinesfalls vollständiger) Überblick einiger weiterer Arten von Autoencodern:

Contractive Autoencoder (CAE) (<https://goo.gl/U5t9Ux>)⁸

Der Autoencoder unterliegt beim Trainieren Beschränkungen, sodass die Ableitungen der Codings nach den Eingaben klein sind. Anders gesagt, müssen zwei ähnliche Eingaben auch ähnliche Codings besitzen.

Stacked Convolutional Autoencoder (<https://goo.gl/PTwsol>)⁹

Autoencoder, die durch das Rekonstruieren von durch Convolutional Layers verarbeiteten Bildern lernen, visuelle Eigenschaften zu erfassen.

⁸ »Contractive Auto-Encoders: Explicit Invariance During Feature Extraction«, S. Rifai et al. (2011).

⁹ »Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction«, J. Masci et al. (2011).

Generative stochastische Netze (GSN) (<https://goo.gl/HjON1m>)¹⁰

Eine Verallgemeinerung von Denoising Autoencodern, die außerdem Daten generieren können.

Winner-Take-All-(WTA-)Autoencoder (<https://goo.gl/I1LvzL>)¹¹

Beim Trainieren werden nach der Berechnung der Aktivierung aller Neuronen in der codierenden Schicht nur die ersten $k\%$ Aktivierungen jedes Neurons im Trainings-Batch gespeichert und der Rest auf null gesetzt. Dies führt natürlich zu spärlichen Codings. Der WTA-Ansatz lässt sich auch zum Erstellen von Sparse Convolutional Autoencodern verwenden.

Generative Adversarial Network (GAN) (<https://goo.gl/enC5fB>)¹²

Ein Netz, der »Diskriminator«, wird zum Unterscheiden von echten und falschen Daten trainiert, die ein zweites Netz, der »Generator«, erzeugt. Der Generator lernt, den Diskriminator auszutricksen, während der Diskriminator lernt, die Tricks des Generators zu umgehen. Dieser Wettbewerb führt zu ausgesprochen realistischen Fake-Daten und sehr robusten Codings. Adversarial Training ist eine sehr mächtige Idee, die gerade Fahrt aufnimmt. Yann Lecun hat sie sogar als »das Coolste seit geschnittenem Brot« bezeichnet.

Übungen

1. Für welche Aufgaben lassen sich Autoencoder vornehmlich einsetzen?
2. Ihnen stehen für eine Klassifikationsaufgabe reichlich Trainingsdaten ohne Labels, aber nur wenige Tausend gelabelte Datenpunkte zur Verfügung. Wie können Autoencoder dabei helfen? Wie würden Sie vorgehen?
3. Ist ein Autoencoder, der die Eingaben perfekt wiedergibt, automatisch gut? Wie lässt sich die Leistungsfähigkeit eines Autoencoders evaluieren?
4. Was sind unternvollständige und übergervollständige Autoencoder? Welches Hauptrisiko besteht bei einem extrem unternvollständigen Autoencoder? Welches bei einem übergervollständigen Autoencoder?
5. Wie lassen sich die Gewichte eines Stacked Autoencoders miteinander koppeln? Warum tut man dies?
6. Mit welcher Technik können Sie die von unteren Schichten eines Stacked Autoencoders erlernten Merkmale visualisieren? Wie die oberen?
7. Was ist ein generatives Modell? Können Sie eine Untergruppe der generativen Autoencoder benennen?
8. Konstruieren wir einen Denoising Autoencoder zum Vortrainieren eines Bildklassifikators:

10 »GSNs: Generative Stochastic Networks«, G. Alain et al. (2015).

11 »Winner-Take-All Autoencoders«, A. Makhzani and B. Frey (2015).

12 »Adversarial Autoencoders«, A. Makhzani et al. (2016).

- Verwenden Sie den MNIST-Datensatz (einfach) oder einen anderen großen Bilddatensatz wie CIFAR10 (<https://goo.gl/VbsmxG>), wenn Sie nach einer Herausforderung suchen. Für das Trainieren von CIFAR10 müssen Sie Code schreiben, der Batches von Bildern lädt. Wenn Sie diesen Teil überspringen möchten, enthält der Model Zoo in TensorFlow Hilfsmittel für diese Aufgabe (<https://goo.gl/3iENgb>).
- Unterteilen Sie den Datensatz in einen Trainingsdatensatz und einen Testdatensatz. Trainieren Sie einen Deep Denoising Autoencoder auf dem vollständigen Trainingsdatensatz.
- Stellen Sie sicher, dass sich die Bilder gut rekonstruieren lassen. Visualisieren Sie die kleinteiligen Merkmale. Visualisieren Sie Bilder, die jedes Neuron in der codierenden Schicht am stärksten aktivieren.
- Konstruieren Sie ein Deep-Learning-Netz zur Klassifikation unter Verwendung der ersten Schichten des Autoencoders. Trainieren Sie es mit nur 10% des Trainingsdatensatzes. Lässt sich die gleiche Vorhersagequalität erbringen, wie beim mit dem vollständigen Trainingsdatensatz trainierten Klassifikator?

9. *Semantic Hashing*, eine von Ruslan Salakhutdinov und Geoffrey Hinton im Jahr 2008 vorgestellte Technik (<https://goo.gl/LXzFX6>)¹³ wird als effiziente Suchtechnik eingesetzt: Ein Dokument (z.B. ein Bild) wird durch ein System, normalerweise ein neuronales Netz, verarbeitet. Dieses erstellt einen Binärvektor mit wenigen Dimensionen (z.B. 30 Bits). Zwei ähnliche Dokumente haben höchstwahrscheinlich identische oder sehr ähnliche Hashes. Indem jedes Dokument über seinen Hash indiziert wird, lassen sich viele ähnliche Dokumente im Handumdrehen finden, selbst bei Milliarden Dokumenten: Sie müssen nur den Hash des Dokuments berechnen und sämtliche Dokumente mit diesem Hash anfordern (oder in einen oder zwei Bits abweichende Hashes). Implementieren wir Semantic Hashing mit einem angepassten Stacked Autoencoder:

- Erstellen Sie einen Stacked Autoencoder mit zwei verborgenen Schichten unterhalb der codierenden Schicht. Trainieren Sie ihn auf dem Bilddatensatz aus der vorigen Übung. Die codierende Schicht sollte 30 Neuronen enthalten und über die logistische Aktivierungsfunktion Werte zwischen 0 und 1 ausgeben. Nach dem Trainieren können Sie die Ausgabe der codierenden Schicht auf ganze Zahlen (0 oder 1) runden, um den Hash eines Bilds zu erhalten.
- Ein von Salakhutdinov und Hinton vorgeschlagener Trick ist, beim Trainieren normalverteiltes Rauschen (mit dem Mittelwert Null) zur Eingabe der codierenden Schicht zu addieren. Um das Rauschen zu übertönen, wird der Autoencoder lernen, große Werte an die codierende Schicht zu übergeben. Das führt auch dazu, dass die logistische Funktion in der

¹³ »Semantic Hashing«, R. Salakhutdinov and G. Hinton (2008).

codierenden Schicht höchstwahrscheinlich die Sättigung bei 0 oder 1 erreicht. Dadurch verzerrt das Runden der Codierung auf 0 oder 1 diese nicht sonderlich, und die Hashes werden dadurch zuverlässiger.

- Berechnen Sie den Hash jedes Bilds und überprüfen Sie, ob sich Bilder mit ähnlichen Hashes ähnlich sehen. Da MNIST und CIFAR10 Datensätze mit Labels sind, können Sie als objektives Kriterium für die Leistung des Autoencoders mit Semantic Hashing auch prüfen, ob Bilder mit dem gleichen Hash der gleichen Kategorie angehören. Als Maß können Sie die durchschnittliche Gini-Reinheit (aus Kapitel 6) für Gruppen von Bildern mit identischen (oder sehr ähnlichen) Hashes bestimmen.
 - Nehmen Sie eine Feinabstimmung der Hyperparameter mittels Kreuzvalidierung vor.
 - Mit einem gelabelten Datensatz könnten Sie auch ein Convolutional Neural Network (siehe Kapitel 13) zur Klassifikation verwenden und anschließend die Hashes mit der Schicht unterhalb der Ausgabeschicht erstellen. Lesen Sie dazu den Artikel aus dem Jahr 2015 (<https://goo.gl/i9FTln>) von Jinma Gua und Jianmin Li.¹⁴ Prüfen Sie, ob dieses Verfahren besser abschneidet.
10. Trainieren Sie einen Variational Autoencoder auf dem Bilddatensatz aus den vorigen Übungen (MNIST oder CIFAR10) und lassen Sie diesen Bilder generieren. Als Alternative können Sie nach einem interessanten Datensatz ohne Labels suchen und schauen, ob Sie neue Daten generieren können.

Lösungen zu diesen Aufgaben finden Sie in Anhang A.

¹⁴ »CNN Based Hashing for Image Retrieval«, J. Gua and J. Li (2015).

KAPITEL 16

Reinforcement Learning

Reinforcement Learning (RL) ist heute eines der aufregendsten Teilgebiete im Machine Learning und gleichzeitig eines der ältesten. Es existiert seit den 1950ern und hat im Lauf der Jahre viele interessante Anwendungen hervorgebracht,¹ insbesondere bei Spielen (z.B. *TD-Gammon*, ein *Backgammon*-Programm) und der Maschinensteuerung. Es gelangt jedoch nur selten in die Schlagzeilen. Aber im Jahr 2013 fand eine kleine Revolution statt, als Forscher aus einem englischen Start-up namens DeepMind ein System vorführten, das jedes Atari-Spiel von null auf erlernen konnte (<https://goo.gl/hceDs5>)² und irgendwann Menschen überlegen war (<https://goo.gl/hgpvz7>).³ In den meisten Spielen bestand die Eingabe nur aus Pixeldaten, und das System besaß zu Beginn keinerlei Wissen über die Spielregeln.⁴ Dies war der erste einer ganzen Serie spektakulärer Durchbrüche, der im Mai 2017 im Sieg des Systems AlphaGo über Ke Jie gipfelte, den Weltmeister im Brettspiel Go. Kein Programm war jemals dem Sieg über einen professionellen Spieler in diesem Spiel nahe gekommen, geschweige denn dem über einen Weltmeister. Heute brodelt das gesamte RL-Feld mit neuen Ideen mit einer großen Anwendungsbreite. DeepMind wurde 2014 von Google für über 500 Millionen Dollar aufgekauft.

Wie war dieser Erfolg möglich? Im Nachhinein sieht es ganz einfach aus: Sie wandten die Lernkapazität von Deep Learning auf das Gebiet Reinforcement Learning an, und es übertraf die Erwartungen bei Weitem. In diesem Kapitel werden wir zuerst erklären, was Reinforcement Learning ist und wofür es sich gut eignet. Anschließend werden wir zwei der wichtigsten Techniken im Deep Reinforcement Learning besprechen: *Policy Gradienten* und *Deep Q-Netze* (DQN), darunter – Markov-Entscheidungsprozesse (MDP). Wir werden diese Techniken verwenden,

-
- 1 Mehr Details finden Sie im RL-Buch (<https://goo.gl/7utZaz>) von Richard Sutton und Andrew Barto, *Reinforcement Learning: An Introduction* (MIT Press) oder im kostenlosen Online-RL-Kurs (<https://goo.gl/AWcMFW>) von David Silver am University College London.
 - 2 »Playing Atari with Deep Reinforcement Learning«, V. Mnih et al. (2013)
 - 3 »Human-level control through deep reinforcement learning«, V. Mnih et al. (2015).
 - 4 Schauen Sie sich die Videos auf <https://goo.gl/yTsH6X> an, in denen das DeepMind-System lernt, *Space Invaders*, *Breakout* und weitere Spiele zu spielen.

um ein Modell zum Balancieren einer Stange auf einem beweglichen Wagen zu entwickeln, sowie ein weiteres zum Spielen von Atari-Konsolenspielen. Die gleichen Techniken lassen sich für eine Reihe anderer Aufgaben von laufenden Robotern bis hin zu selbstfahrenden Autos einsetzen.

Lernen zum Optimieren von Belohnungen

Beim Reinforcement Learning sammelt ein Software-Agent *Beobachtungen*, führt *Aktionen* innerhalb einer *Umwelt* durch und erhält dafür *Belohnungen*. Das Ziel des Agenten ist, durch sein Verhalten die langfristige Belohnung zu maximieren. Wenn Ihnen ein wenig Anthropomorphismus nichts ausmacht, können Sie sich die positiven Belohnungen als Vergnügen vorstellen und negative Belohnungen als Schmerzen (der Begriff »Belohnung« ist in diesem Zusammenhang etwas irreführend). Kurz, der Agent handelt in seiner Umwelt und lernt durch Versuch und Irrtum, um sein Vergnügen zu maximieren und die Schmerzen zu minimieren.

Dies ist ein recht weiträumiges Szenario, das bei vielen verschiedenen Aufgaben Anwendung findet. Hier sind einige Beispiele (siehe auch Abbildung 16-1):

1. Der Agent könnte ein Programm sein, das einen laufenden Roboter kontrolliert. In diesem Fall ist die Umwelt die reale Welt. Der Agent beobachtet die Umwelt über *Sensoren* wie Kameras und Berührungssensoren, und seine Aktionen bestehen aus Signalen zur Aktivierung von Motoren. Er kann darauf programmiert sein, positive Belohnungen zu erhalten, wenn er das Ziel erreicht, und negative Belohnungen, wenn er Zeit verplempert, in die falsche Richtung läuft oder umfällt.
2. Der Agent könnte ein Programm sein, das Ms. Pac-Man steuert. In diesem Fall ist die Umwelt die Emulation eines Atari-Konsolenspiel. Die Aktionen sind die neun möglichen Positionen des Joysticks (links oben, unten, mittig und so weiter), die Beobachtungen sind Screenshots, und die Belohnungen sind die Punkte im Spiel.
3. In ähnlicher Weise könnte der Agent ein Programm sein, das ein Brettspiel wie Go spielt.
4. Der Agent muss nicht unbedingt einen beweglichen physischen (oder virtuellen) Gegenstand steuern. Es könnte auch ein intelligenter Thermostat sein, der immer dann belohnt wird, wenn er sich nah an der Zieltemperatur befindet und Energie spart, und bestraft wird, wenn Menschen die Temperatur verstehen müssen. Der Agent muss also lernen, menschliche Bedürfnisse vorherzusehen.
5. Ein Agent könnte auch Aktienkurse beobachten und entscheiden, wie viel er jede Sekunde kaufen oder verkaufen soll. Die Belohnungen sind dann natürlich finanzielle Gewinne und Verluste.

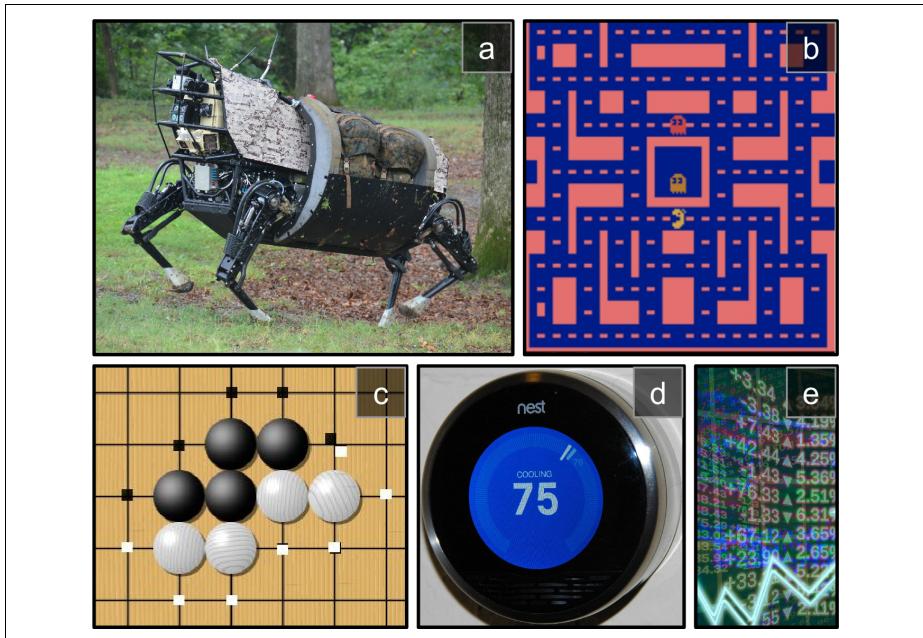


Abbildung 16-1: Beispiele für Reinforcement Learning: (a) laufender Roboter, (b) Ms. Pac-Man, (c) Go-Spieler, (d) Thermostat, (e) automatischer Börsenspekulant⁵

Es sollte betont werden, dass es gar keine positiven Belohnungen geben muss; der Agent könnte sich beispielsweise durch ein Labyrinth bewegen und bei jedem Schritt eine negative Belohnung erhalten. Somit ist es besser, den Ausgang so schnell wie möglich zu finden! Es gibt viele weitere Anwendungsbeispiele, für die Reinforcement Learning geeignet ist, wie selbstfahrende Autos, das Platzieren von Werbung auf Webseiten und zum Steuern, worauf ein System zur Bildklassifizierung seine Aufmerksamkeit richten sollte.

Suche nach Policies

Den Algorithmus, den die Agentensoftware zum Entscheidungsfindung verwendet, nennt man *Policy*. Beispielsweise könnte die Policy ein neuronales Netz sein, das die Beobachtungen als Eingaben und die auszuführende Aktion als Ausgabe hat (siehe Abbildung 16-2).

⁵ Die Bilder (a), (c) und (d) wurden aus Wikipedia reproduziert. (a) und (d) sind Allgemeingut. (c) wurde vom Nutzer Stevertigo unter der *Creative Commons BY-SA 2.0* (<https://creativecommons.org/licenses/by-sa/2.0/>) veröffentlicht. (b) ist ein Screenshot aus dem Spiel Ms. Pac-Man, Copyright von Atari (der Autor nimmt an, dass die Nutzung in diesem Kapitel gerechtfertigt ist). (e) wurde von Pixabay reproduziert und unter der *Creative Commons CC0* (<https://creativecommons.org/publicdomain/zero/1.0/>) veröffentlicht.

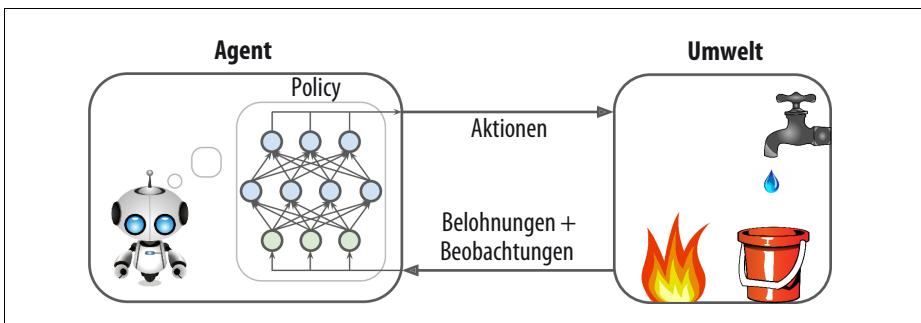


Abbildung 16-2: Reinforcement Learning mit einem neuronalen Netz als Policy

Die Policy kann ein beliebiger Algorithmus sein, der nicht einmal deterministisch arbeiten muss. Betrachten wir als Beispiel einen Saugroboter, dessen Belohnung die in 30 Minuten aufgesaugte Staubmenge ist. Dessen Policy könnte darin bestehen, sich jede Sekunde mit einer bestimmten Wahrscheinlichkeit p vorwärts zu bewegen oder sich mit der Wahrscheinlichkeit $1 - p$ zufällig nach links zu drehen. Der Drehwinkel wäre ein zufälliger Winkel zwischen $-r$ und $+r$. Da diese Policy ein Zufallselement enthält, bezeichnet man sie als *stochastische Policy*. Der Roboter wird einer sehr unsteten Trajektorie folgen, wodurch sichergestellt wird, dass er früher oder später jeden Ort erreicht und den Staub dort aufsaugt. Die Frage dabei ist: Wie viel Staub wird innerhalb von 30 Minuten aufgesaugt?

Wie würden Sie solch einen Roboter trainieren? Die Policy enthält nur zwei Parameter, die Sie ändern können: die Wahrscheinlichkeit p und den Winkelbereich r . Ein möglicher Lernalgorithmus bestünde darin, viele unterschiedliche Werte dieser Parameter auszuprobieren und die Kombination mit der besten Leistung auszuwählen (siehe Abbildung 16-3). Dies ist ein Beispiel für *Policy-Suche*, in diesem Fall nach einem brute-force-Ansatz. Falls jedoch der *Policy-Raum* zu groß ist (was meistens der Fall ist), dann ist das Finden eines guten Parametersatzes ähnlich wie das Suchen einer Nadel in einem gigantischen Heuhaufen.

Eine andere Möglichkeit zum Durchsuchen des Policy-Raums sind *genetische Algorithmen*. Sie könnten beispielsweise zufällig eine erste Generation von 100 Policies erzeugen, sie ausprobieren und dann die schlechten 80 Policies eliminieren.⁶ Die 20 Überlebenden produzieren dann jeweils 4 Nachkommen. Ein Nachkomme ist nichts weiter als eine Kopie seines Elternteils mit etwas Variation.⁷ Die überlebenden Policies und ihre Nachkommen bilden die zweite Generation. Sie können auf diese Weise weitere Generationen iterativ erzeugen, bis Sie eine gute Policy finden.

⁶ Es zahlt sich häufig aus, den Leistungsschwächeren eine geringe Überlebenschance zu geben, um eine Diversität im »Genpool« zu erhalten

⁷ Wenn es einen Elternteil gibt, nennt man dies *asexuelle Reproduktion*. Mit zwei (oder mehr) Elternteilen nennt man es *sexuelle Reproduktion*. Das Genom der Nachkommen (in diesem Fall die Parameter der Policy) wird zufällig aus Teilen der Elterngenoome zusammengesetzt.

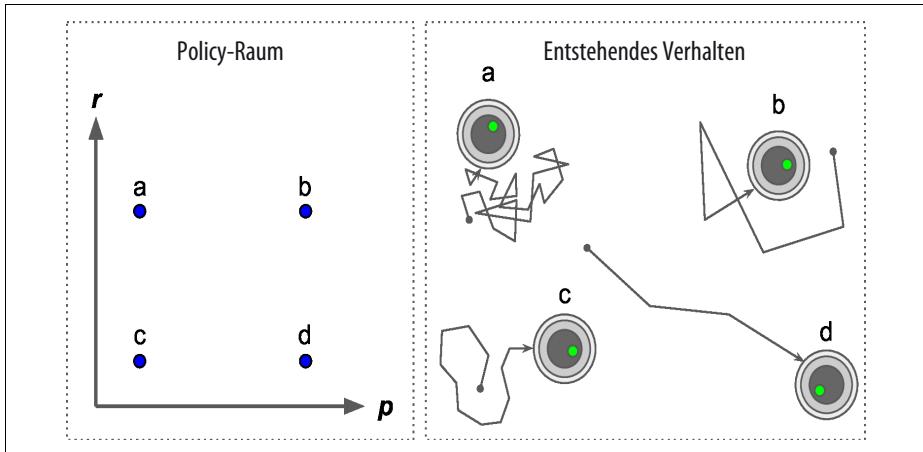


Abbildung 16-3: Vier Punkte im Policy-Raum und das entsprechende Verhalten des Agenten

Einen weiteren Ansatz bieten Optimierungstechniken, die die Gradienten der Belohnungen nach den Parametern der Policy auswerten. Anschließend verändern Sie diese Parameter, indem Sie dem Gradienten in Richtung höherer Belohnung folgen (*Gradientenaufstieg*). Diesen Ansatz nennt man *Policy-Gradienten* (PG). Wir werden ihn später in diesem Kapitel im Detail besprechen. Im Beispiel des Staubsauger-Roboters könnten Sie p ein wenig erhöhen und auswerten, ob sich die in 30 Minuten vom Roboter aufgenommene Staubmenge erhöht; falls ja, erhöhen Sie p ein wenig, ansonsten verringern Sie p . Wir werden einen beliebten PG-Algorithmus mit TensorFlow implementieren. Zuvor aber müssen wir eine Umwelt erstellen, in der unser Agent sich bewegen kann. Es ist daher an der Zeit, Ihnen OpenAI Gym vorzustellen.

Einführung in OpenAI Gym

Eine der Herausforderungen beim Reinforcement Learning ist, dass Sie zum Trainieren eines Agenten zunächst eine funktionierende Umwelt benötigen. Wenn Sie einen Agenten zum Spielen eines Atari-Spiels trainieren möchten, benötigen Sie einen Emulator für Atari-Spiele. Wenn Sie einen laufenden Roboter programmieren möchten, ist die Umwelt die reale Welt. Sie können Ihren Roboter direkt in dieser Umwelt trainieren. Dies hat aber seine Grenzen: Wenn der Roboter von einer Klippe stürzt, können Sie nicht einfach auf »Undo« klicken. Sie können den Prozess nur schwer beschleunigen; durch mehr Rechenleistung wird der Roboter nicht schneller laufen. Es ist auch meistens zu teuer, 1000 Roboter parallel zu trainieren. Kurz, das Trainieren in der realen Welt ist schwierig und langsam. Sie benötigen daher eine *simulierte Umwelt*, zumindest um den Trainingsprozess aufzubauen.

OpenAI Gym (<https://gym.openai.com/>)⁸ ist ein Werkzeugkasten mit einer großen Bandbreite simulierter Umgebungen (Atari-Spiele, Brettspiele, physikalische Simulationen in 2-D und 3-D und so weiter). Sie können damit Agenten trainieren, vergleichen und neue RL-Algorithmen entwickeln.

Installieren wir OpenAI Gym. Für eine minimale OpenAI-Gym-Installation können Sie pip verwenden:

```
$ pip3 install --upgrade gym
```

Öffnen Sie als Nächstes eine Python-Shell oder ein Jupyter Notebook und erstellen Sie Ihre erste Umwelt:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2017-09-13 10:48:27,402] Making new env: CartPole-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115,  0.02337094,  0.00720711])
>>> env.render()
```

Die Funktion `make()` erstellt eine Umwelt, in diesem Fall die Umwelt CartPole. Dies ist eine 2-D-Simulation, in der sich ein Wagen nach links oder rechts beschleunigen lässt, um eine darauf platzierte Stange zu balancieren (siehe Abbildung 16-4). Nachdem die Umwelt erstellt wurde, müssen wir sie mit der Methode `reset()` initialisieren. Damit erhalten wir die erste Beobachtung. Beobachtungen hängen von der Art der Umwelt ab. In der Umwelt CartPole ist jede Beobachtung ein 1-D-NumPy-Array mit vier floats: Diese floats stehen für die horizontale Position des Wagens (0.0 = mittig), seine Geschwindigkeit, den Winkel der Stange (0.0 = vertikal) und ihre Winkelgeschwindigkeit. Schließlich stellt die Methode `render()` die Umwelt wie in Abbildung 16-4 dar.

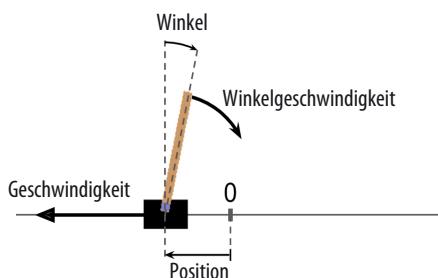


Abbildung 16-4: Die CartPole-Umwelt

⁸ OpenAI ist ein Non-Profit-Forschungsunternehmen im Bereich künstlicher Intelligenz, das teilweise von Elon Musk gegründet wurde. Sein erklärtes Ziel ist, freundlich gesinnte AIs zu ermöglichen und zu entwickeln, die der Menschheit nützen (anstatt sie auszulöschen).

Wenn Sie `render()` verwenden möchten, um ein gerendertes Bild als NumPy-Array abzulegen, können Sie den Parameter `mode` auf `rgb_array` setzen (andere Umwelten unterstützen andere Modi):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # Höhe, Breite, Farbkanäle (3=RGB)
(400, 600, 3)
```



Leider zeichnet CartPole (wie einige andere Umwelten) das Bild selbst im "rgb_array"-Modus grundsätzlich auf den Bildschirm. Die einzige Möglichkeit, dies zu umgehen, ist, ihm mit Xvfb oder Xdummy einen X-Server vorzutäuschen. Sie können Xvfb installieren und mit dem folgenden Befehl Python darauf starten: `xvfb-run -s "-screen 0 1400x900x24" python`. Alternativ können Sie das Paket xvfbwrapper (<https://goo.gl/wR1ojl>) verwenden.

Befragen wir unsere Umwelt, welche Aktionen möglich sind:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` bedeutet, dass die möglichen Aktionen die Integerzahlen 0 und 1 sind. Diese stehen für die Beschleunigung nach links (0) oder rechts (1). Andere Umwelten haben weitere diskrete Aktionen oder andere Arten von Aktionen (z.B. kontinuierliche). Da die Stange nach rechts geneigt ist, beschleunigen wir den Wagen nach rechts:

```
>>> action = 1 # Beschleunigung nach rechts
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

Die Methode `step()` führt diese Aktion aus und gibt vier Werte zurück:

`obs`

Dies ist die neue Beobachtung. Der Wagen bewegt sich nun nach rechts (`obs[1]>0`). Die Stange neigt sich noch immer nach rechts (`obs[2]>0`), aber ihre Winkelgeschwindigkeit ist nun negativ (`obs[3]<0`). Vermutlich wird sie sich nach dem nächsten Schritt nach links neigen.

`reward`

In dieser Umgebung erhalten Sie bei jedem Schritt unabhängig von Ihrer Aktion eine Belohnung von 1.0. Das Ziel ist also, so lange wie möglich im Spiel zu bleiben.

done

Dieser Wert wird True, wenn die *Episode* vorbei ist. Das passiert, sobald die Stange kippt. Danach muss die Umgebung zurückgesetzt werden, um neu verwendet werden zu können.

info

Dieses Dictionary kann in anderen Umgebungen zusätzliche Informationen zum Debuggen enthalten. Diese Daten sollten nicht zum Trainieren verwendet werden (das wäre Schummelei).

Programmieren wir eine einfache Policy, die nach links beschleunigt, wenn die Stange sich nach links neigt, und nach rechts beschleunigt, wenn sie sich nach rechts neigt. Wir führen diese Policy aus und lassen uns die durchschnittliche Belohnung nach 500 Episoden ausgeben:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000):
        # maximal 1000 Schritte, wir möchten es nicht ewig laufen lassen
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

Dieser Code ist hoffentlich selbsterklärend. Sehen wir uns das Ergebnis an:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(42.12599999999998, 9.1237121830974033, 24.0, 68.0)
```

Diese Policy schafft es nicht einmal mit 500 Versuchen, die Stange für mehr als 68 aufeinanderfolgende Schritte aufrecht zu halten. Kein großartiges Ergebnis. Wenn Sie sich die Simulation in den Jupyter Notebooks (<https://github.com/ageron/handson-ml>) ansehen, werden Sie bemerken, dass der Wagen immer stärker nach links und rechts oszilliert, bis die Stange ins Kippen gerät. Schauen wir einmal, ob ein neuronales Netz eine bessere Policy entwickeln kann.

Neuronale Netze als Policies

Erstellen wir nun ein neuronales Netz als Policy. Wie die obige hartcodierte Policy wird dieses neuronale Netz eine Beobachtung als Eingabe annehmen und die auszuführende Aktion ausgeben. Genauer wird sie die Wahrscheinlichkeit für jede

Aktion abschätzen und anhand dieser Wahrscheinlichkeiten zufällig eine Aktion auswählen (siehe Abbildung 16-5). Bei der CartPole-Umgebung gibt es nur zwei mögliche Aktionen (links und rechts). Daher benötigen wir nur ein Ausgabe-Neuron, das die Wahrscheinlichkeit p der Aktion 0 (links) ausgibt. Selbstverständlich ist die Wahrscheinlichkeit der Aktion 1 (rechts) dann $1 - p$. Wenn die Ausgabe z. B. 0.7 ist, werden wir Aktion 0 mit 70%iger Wahrscheinlichkeit und Aktion 1 mit 30%iger Wahrscheinlichkeit auswählen.

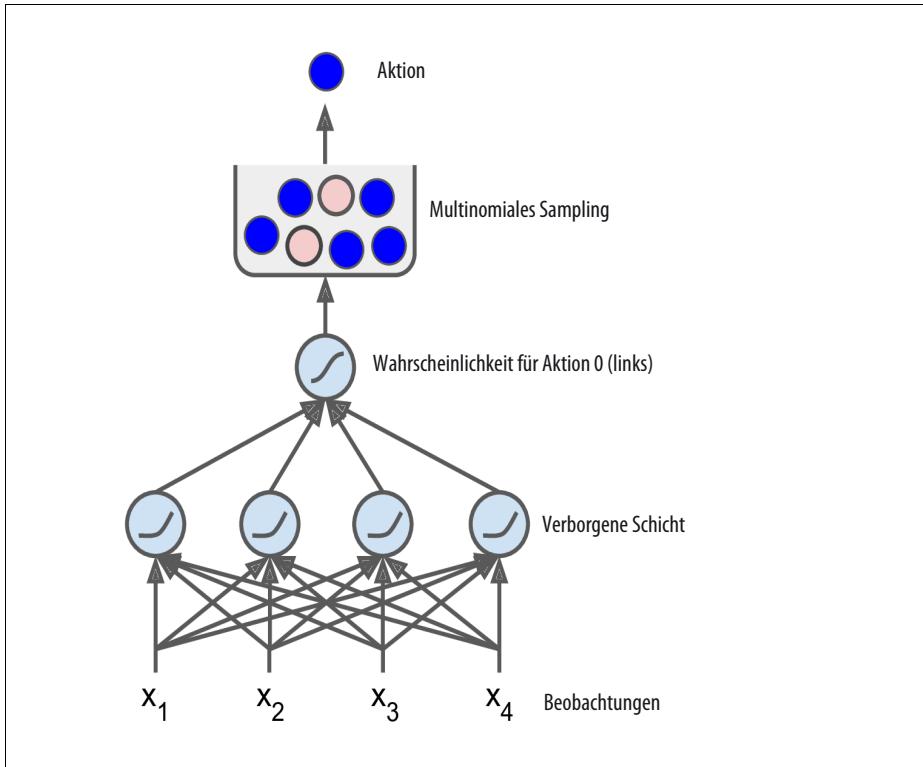


Abbildung 16-5: Ein neuronales Netz als Policy

Sie fragen sich womöglich, warum wir mit den Wahrscheinlichkeiten des neuronalen Netzes eine Aktion zufällig auswählen, anstatt einfach die Aktion mit dem höchsten Score zu nehmen. Dieser Ansatz erlaubt es dem Agenten, die richtige Balance zwischen dem *Ausprobieren* neuer Aktionen und dem *Anwenden* der bekannten Lösungen zu finden. Hier ist eine Analogie: Wenn Sie ein Restaurant zum ersten Mal besuchen, sehen alle Gerichte gleich appetitlich aus. Sie wählen daher eines zufällig aus. Wenn es Ihnen schmeckt, wählen Sie dieses beim nächsten Mal mit einer höheren Wahrscheinlichkeit wieder aus. Sie sollten diese Wahrscheinlichkeit jedoch nie auf 100% steigern, sonst werden Sie keines der anderen Gerichte ausprobieren, von denen manche womöglich noch besser schmecken.

Es ist zu betonen, dass in dieser Umgebung die vergangenen Aktionen und Beobachtungen keine Rolle spielen, da jede Beobachtung den vollständigen Zustand der Umgebung enthält. Falls es verborgene Zustände gibt, müssen Sie eventuell auch vergangene Aktionen und Beobachtungen berücksichtigen. Wenn die Umgebung beispielsweise nur die Position des Wagens, aber nicht dessen Geschwindigkeit enthält, müssten Sie auch die vorige Beobachtung berücksichtigen, um die Geschwindigkeit zu bestimmen. Ein anderes Beispiel ist, wenn die Beobachtungen verrauscht sind; in diesem Fall sollten Sie einige vergangene Beobachtungen verwenden, um den wahrscheinlichsten aktuellen Zustand zu schätzen. Die CartPole-Aufgabe ist daher so einfach wie möglich; die Beobachtungen sind frei von Rauschen und enthalten den vollständigen Zustand der Umgebung.

Der folgende Code erstellt das neuronale Netz für diese Policy mit TensorFlow:

```
import tensorflow as tf

# 1. Die Architektur des neuronalen Netzes definieren
n_inputs = 4 # == env.observation_space.shape[0]
n_hidden = 4 # die Aufgabe ist einfach, wir benötigen nicht mehr als
             # vier verborgene Neuronen
n_outputs = 1 # nur die Wahrscheinlichkeit einer Beschleunigung nach links ausgeben
initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Erstelle das neuronale Netz
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,
                        kernel_initializer=initializer)
logits = tf.layers.dense(hidden, n_outputs,
                        kernel_initializer=initializer)
outputs = tf.nn.sigmoid(logits)

# 3. Wähle anhand der geschätzten Wahrscheinlichkeiten zufällig eine Aktion aus
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

init = tf.global_variables_initializer()
```

Gehen wir diesen Code durch:

1. Nach den import-Anweisungen definieren wir die Architektur des neuronalen Netzes. Die Anzahl der Eingaben ist die Größe des Beobachtungsraums (im Falle von CartPole vier), wir benötigen nicht mehr als vier verborgene Neuronen, und es gibt nur eine Ausgabewahrscheinlichkeit (die Wahrscheinlichkeit einer Bewegung nach links).
2. Danach erstellen wir das neuronale Netz. In diesem Beispiel ist es ein reines mehrschichtiges Perzeptron mit einer einzelnen Ausgabe. Beachten Sie, dass die Ausgabeschicht die logistische (sigmoide) Aktivierungsfunktion verwendet, um eine Wahrscheinlichkeit zwischen 0.0 und 1.0 auszugeben. Wenn es mehr als zwei mögliche Aktionen gäbe, bräuchten wir ein Neuron pro Aktion und würden stattdessen die Softmax-Aktivierungsfunktion verwenden.

- Als Letztes rufen wir die Funktion `multinomial()` auf, um zufällig eine Aktion auszuwählen. Diese Funktion zieht eine Integerzahl (oder mehrere) anhand der logarithmierten Wahrscheinlichkeiten jedes Integers. Wenn Sie sie beispielsweise mit dem Array `[np.log(0.5), np.log(0.2), np.log(0.3)]` und `num_samples=5` aufrufen, erhalten Sie als Ausgabe fünf Integerzahlen, wobei jede mit 50%iger Wahrscheinlichkeit 0 beträgt, mit 20%iger Wahrscheinlichkeit 1 und mit 30%iger Wahrscheinlichkeit 2. In unserem Fall benötigen wir nur einen Integer für die zu wählende Aktion. Da unser Tensor `outputs` nur die Wahrscheinlichkeit für die Bewegung nach links enthält, fügen wir zunächst `1-outputs` hinzu, sodass der Tensor die Wahrscheinlichkeiten für die Aktionen links und rechts enthält. Wenn es mehr als zwei mögliche Aktionen gäbe, müsste das neuronale Netz eine Wahrscheinlichkeit pro Aktion ausgeben, sodass dieser Schritt entfiel.

Nun haben wir ein neuronales Netz als Policy, das Beobachtungen annimmt und Aktionen ausgibt. Aber wie trainieren wir es?

Auswerten von Aktionen: Das Credit-Assignment-Problem

Wenn wir die bestmögliche Aktion bei jedem Schritt bereits kennen würden, könnten wir das neuronale Netz wie gewohnt trainieren, indem wir die Kreuzentropie zwischen der geschätzten Wahrscheinlichkeit und der Zielwahrscheinlichkeit minimieren. Es wäre gewöhnliches überwachtes Lernen. Allerdings sind beim Reinforcement Learning die Belohnungen der einzige Anhaltspunkt für den Agenten, und normalerweise sind die Belohnungen dünn gesät und treten zeitlich versetzt ein. Nehmen wir an, der Agent schafft es, die Stange 100 Schritte lang zu balancieren. Woher sollen wir wissen, welche der 100 Aktionen gut und welche schlecht waren? Wir wissen nur, dass die Stange nach der letzten Aktion umgefallen ist, aber die letzte Aktion ist mit Sicherheit nicht allein verantwortlich. Dies nennt man das *Credit-Assignment-Problem*: Wenn der Agent eine Belohnung erhält, ist es schwierig, einzelne Aktionen hierfür zu loben (oder zu tadeln). Denken Sie an einen Hund, den Sie erst Stunden nach gutem Benehmen belohnen. Er wird den Zusammenhang zwischen Verhalten und Belohnung nicht erkennen.

Dieses Problem lässt sich angehen, indem wir eine Aktion anhand der Summe aller danach erfolgten Belohnungen evaluieren. Normalerweise wenden wir bei jedem Schritt eine *Discount-Rate* an. Wenn beispielsweise ein Agent beschließt, dreimal hintereinander nach rechts zu gehen, und nach dem ersten Schritt eine Belohnung von +10, nach dem zweiten Schritt 0 und nach dem dritten Schritt -50 erhält (siehe Abbildung 16-6), erhalten wir mit einer Discount-Rate $r = 0.8$ für die erste Aktion einen Score von $10 + r \times 0 + r^2 \times (-50) = -22$. Wenn die Discount-Rate nahe 0 ist, spielen langfristige Belohnungen im Vergleich zu unmittelbaren eine geringe Rolle. Wenn umgekehrt die Discount-Rate nahe 1 liegt, zählen Belohnungen in ferner

Zukunft fast so viel wie unmittelbare. Typische Discount-Raten sind 0.95 oder 0.99. Mit einer Discount-Rate von 0.95 sind Belohnungen, die 13 Schritte in der Zukunft liegen, etwa halb so viel wert wie unmittelbare (weil $0.95^{13} \approx 0.5$). Bei einer Discount-Rate von 0.99 dagegen sind Belohnungen nach 69 Schritten noch halb so viel wert wie unmittelbare. In der CartPole-Umgebung haben Aktionen recht kurzfristige Auswirkungen. Daher erscheint eine Discount-Rate von 0.95 angemessen.

Natürlich können auf eine gute Aktion mehrere schlechte folgen, wodurch die Stange am Ende schnell umfällt und die gute Aktion einen schlechten Score erhält (so wie ein guter Schauspieler manchmal in einem miesen Film mitspielt). Wenn wir das Spiel jedoch oft genug spielen, erhalten die guten Aktionen im Schnitt einen besseren Score als schlechte. Um also zuverlässige Scores für die Aktionen zu erhalten, müssen wir viele Episoden durchführen und sämtliche Scores der Aktionen normalisieren (den Mittelwert abziehen und durch die Standardabweichung teilen). Danach können wir davon ausgehen, dass Aktionen mit einem negativen Score schlecht und Aktionen mit positivem Score gut waren. Perfekt – da wir die Aktionen nun evaluieren können, sollten wir unseren ersten Agenten mit Policy-Gradienten trainieren. Sehen wir, wie das geht.

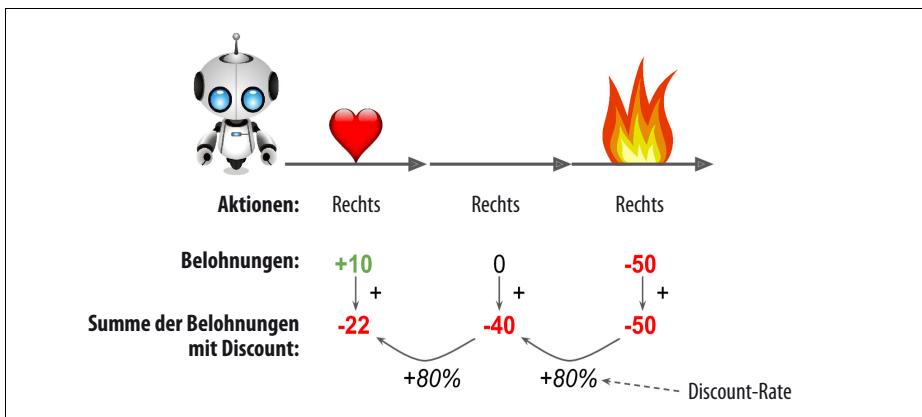


Abbildung 16-6: Belohnungen mit Discount-Rate

Policy-Gradienten

Wie bereits erwähnt, optimieren PG-Algorithmen die Parameter einer Policy, indem sie dem Gradienten in Richtung höherer Belohnungen folgen. Eine beliebte Klasse von PG-Algorithmen, die *REINFORCE-Algorithmen*, wurde von Ronald Williams bereits im Jahr 1992 vorgestellt (<https://goo.gl/tUe4Sh>).⁹ Hier folgt eine verbreitete Variante:

⁹ »Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning«, R. Williams (1992).

1. Zuerst spielt die Policy mit dem neuronalen Netz das Spiel einige Male und berechnet bei jedem Schritt Gradienten, mit denen die gewählte Aktion mit höherer Wahrscheinlichkeit ausgewählt würde, wendet diese aber noch nicht an.
2. Sobald mehrere Episoden ausgeführt wurden, berechnen Sie den Score jeder Aktion (nach dem im vorigen Abschnitt beschriebenen Verfahren).
3. Wenn der Score einer Aktion positiv ist, war die Aktion gut, und der zuvor berechnete Gradient sollte angewendet werden, um diese Aktion in der Zukunft wahrscheinlicher zu machen. Wenn der Score jedoch negativ ist, war die Aktion ungünstig, und der entgegengesetzte Gradient sollte angewendet werden, damit die Aktion in der Zukunft *weniger* wahrscheinlich wird. Dazu wird der Gradientenvektor einfach mit dem entsprechenden Score der Aktion multipliziert.
4. Schließlich werden alle erhaltenen Gradientenvektoren gemittelt und ein Schritt im Gradientenverfahren damit durchgeführt.

Implementieren wir diesen Algorithmus mit TensorFlow. Wir trainieren die zuvor erstellte Policy mit dem neuronalen Netz, sodass es lernt, die Stange auf dem Wagen zu balancieren. Vervollständigen wir zunächst die Konstruktionsphase, indem wir die Zielwahrscheinlichkeit, die Kostenfunktion und die Operation zum Trainieren hinzufügen. Da wir so tun, als wäre die ausgewählte Aktion die bestmögliche, beträgt die Zielwahrscheinlichkeit 1.0 bei Aktion 0 (links) und 0.0 bei Aktion 1 (rechts):

```
y = 1. - tf.to_float(action)
```

Mit der Zielwahrscheinlichkeit können wir die Kostenfunktion (Kreuzentropie) definieren und die Gradienten berechnen:

```
learning_rate = 0.01

cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

Wir rufen hier die Methode `compute_gradients()` anstatt der Methode `minimize()` auf. Dies liegt daran, dass wir die Gradienten verändern möchten, bevor wir sie anwenden.¹⁰ Die Methode `compute_gradients()` liefert eine Liste von Paaren aus dem Gradientenvektor und der Variablen (ein Paar pro trainierbarer Variable). Packen wir alle Gradienten in eine Liste, um das Ermitteln ihrer Werte zu vereinfachen:

```
gradients = [grad for grad, variable in grads_and_vars]
```

Okay, nun kommt der schwierige Teil. Während der Ausführungsphase führt der Algorithmus unsere Policy aus, evaluiert die Tensoren des Gradienten bei jedem

¹⁰ Wir haben in Kapitel 11 bereits etwas Ähnliches im Zusammenhang mit Gradient Clipping getan: Zuerst haben wir die Gradienten berechnet, anschließend zurechtgeschnitten und erst am Ende die zurechtgeschnittenen Gradienten angewendet.

Schritt und speichert ihre Werte. Nach einer Anzahl Episoden werden die Gradienten wie oben erklärt angepasst (d.h. mit den Scores der Aktionen multipliziert und dann normalisiert) und der Mittelwert dieser angepassten Gradienten berechnet. Anschließend werden die berechneten Gradienten wieder dem Optimierer übergeben, sodass dieser einen Optimierungsschritt durchführen kann. Wir benötigen also pro Gradientenvektor einen Platzhalter. Außerdem müssen wir eine Operation zum Anwenden der veränderten Gradienten erstellen. Dazu rufen wir im Optimierer die Funktion `apply_gradients()` auf, die eine Liste von Gradientenvektor-Variablen-Paaren annimmt. Anstatt ihr die ursprünglichen Gradientenvektoren zu übergeben, übergeben wir eine Liste der aktualisierten Gradienten (d.h. den als Platzhalter erstellten):

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))

training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

Betrachten wir noch einmal die gesamte Konstruktionsphase:

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,
                       kernel_initializer=initializer)
logits = tf.layers.dense(hidden, n_outputs, kernel_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Auf zur Ausführungsphase! Wir benötigen einige Funktionen, um aus den Rohdaten für die Belohnungen die gesamten Belohnungen zu berechnen und um die Ergebnisse über mehrere Episoden zu normalisieren:

```
def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards = rewards[step] + cumulative_rewards * discount_rate
        discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards, discount_rate)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std
            for discounted_rewards in all_discounted_rewards]
```

Überprüfen wir, ob das funktioniert:

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([ 1.26665318,  1.0727777 ])]
```

Der Aufruf von `discount_rewards()` liefert genau das erwartete Ergebnis (siehe Abbildung 16-6). Sie können nachweisen, dass die Funktion `discount_and_normalize_rewards()` tatsächlich die normierten Scores für jede Aktion in beiden Episoden zurückgibt. Beachten Sie, dass die erste Episode viel schlechter als die zweite war. Daher sind alle ihre normierten Scores negativ; alle Aktionen aus der ersten Episode würden als schlecht betrachtet, umgekehrt wären alle Aktionen aus der zweiten Episode gut.

Wir haben nun alles beisammen, was wir zum Trainieren der Policy benötigen:

```
n_iterations = 250      # Anzahl der Trainingsepisoden
n_max_steps = 1000       # maximale Anzahl Schritte pro Episode
n_games_per_update = 10 # trainiere die Policy alle 10 Episoden
save_
iterations = 10          # speichere das Modell beim Trainieren alle 10 Iterationen ab
discount_rate = 0.95

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []      # alle Folgen von Belohnungen aus jeder Episode
        all_gradients = []
                        # bei jedem Schritt jeder Episode gespeicherte Gradienten
        for game in range(n_games_per_update):
            current_rewards = [] # alle Belohnungen aus der aktuellen Episode
            current_gradients = [] # alle Gradienten aus der aktuellen Episode
```

```

obs = env.reset()
for step in range(n_max_steps):
    action_val, gradients_val = sess.run(
        [action, gradients],
        feed_dict={X: obs.reshape(1, n_inputs)}) # eine Beobachtung
    obs, reward, done, info = env.step(action_val[0][0])
    current_rewards.append(reward)
    current_gradients.append(gradients_val)
    if done:
        break
all_rewards.append(current_rewards)
all_gradients.append(current_gradients)

# An diesem Punkt haben wir die Policy 10 Episoden lang ausgeführt
# und sind bereit, sie nach dem oben beschriebenen Algorithmus
# zu aktualisieren.
all_rewards = discount_and_normalize_rewards(all_rewards, discount_rate)
feed_dict = {}
for var_index, grad_placeholder in enumerate(gradient_placeholders):
    # multipliziere die Gradienten mit den Scores der Aktionen
    # und berechne den Mittelwert
    mean_gradients = np.mean(
        [reward * all_gradients[game_index][step][var_index]
         for game_index, rewards in enumerate(all_rewards)
         for step, reward in enumerate(rewards)],
        axis=0)
    feed_dict[grad_placeholder] = mean_gradients
sess.run(training_op, feed_dict=feed_dict)
if iteration % save_iterations == 0:
    saver.save(sess, "./my_policy_net_pg.ckpt")

```

Jede Iteration des Trainings führt zuerst die Policy 10 Episoden lang aus (mit maximal 1000 Schritten pro Episode, damit sie nicht ewig läuft). Bei jedem Schritt berechnen wir die Gradienten und tun so, als wäre die ausgewählte Aktion die bestmögliche. Nach diesen 10 Episoden berechnen wir die Scores der Aktionen mit der Funktion `discount_and_normalize_rewards()`; wir betrachten jede trainierbare Variable über sämtliche Episoden und Schritte, um jeden Gradientenvektor mit dem Score der entsprechenden Aktion zu multiplizieren; wir berechnen dann den Mittelwerte der resultierenden Gradienten. Am Ende führen wir die Trainingsoperation aus und übergeben ihr die so gemittelten Gradienten (einer pro trainierbarer Variable). Wir speichern das Modell außerdem alle 10 Trainingsoperationen ab.

Damit sind wir fertig! Dieser Code trainiert das neuronale Netz in der Policy und lernt erfolgreich, die Stange auf dem Wagen zu balancieren (Sie können dies in den Jupyter Notebooks überprüfen). Beachten Sie, dass der Agent das Spiel auf zweierlei Weise verlieren kann: Entweder kippt die Stange zu stark, oder der Wagen verlässt den Bildschirm vollständig. Nach einem Training mit 250 Iterationen hat die Policy recht gut gelernt, die Stange zu balancieren, hat aber das Verlassen des Bildschirms noch nicht unter Kontrolle. Nach einigen Hundert weiteren Iterationen hat sie auch dieses Problem im Griff.



Forscher sind bemüht, Algorithmen zu finden, die auch ohne vorheriges Wissen über die Umwelt gut funktionieren. Wenn Sie aber nicht gerade einen Fachartikel schreiben, sollten Sie dem Agenten so viel Wissen wie möglich zur Verfügung stellen, da dies das Training erheblich beschleunigt. Sie könnten beispielsweise eine negative Belohnung hinzufügen, die proportional zur Entfernung von der Bildschirmmitte und dem Winkel der Stange ist. Auch wenn Sie bereits eine halbwegs gute Policy haben (z.B. eine hartcodierte), sollten Sie zunächst diese durch das neuronale Netz abbilden lassen, bevor Sie es mit Policy-Gradienten verbessern.

Trotz seiner Einfachheit ist dieser Algorithmus recht mächtig. Sie können damit viel schwierigere Aufgaben als das Balancieren einer Stange auf einem Wägelchen angehen. Tatsächlich basiert AlphaGo auf einem ähnlichen PG-Algorithmus (und auf dem Algorithmus *Monte-Carlo-Baumsuche*, den wir nicht in diesem Buch behandeln).

Wir werden uns nun eine weitere beliebte Familie von Algorithmen ansehen. Während die PG-Algorithmen die Policy direkt in Richtung höherer Belohnungen optimieren, betrachten wir nun einige weniger direkte Algorithmen: Der Agent lernt, die zu erwartende Summe zukünftiger Belohnungen entweder für jeden Zustand oder für jede Aktion in jedem Zustand abzuschätzen. Dieses Wissen wird dann zur Entscheidungsfindung eingesetzt. Um diese Art Algorithmen zu verstehen, müssen wir uns mit *Markov-Entscheidungsprozessen* (MDP) auseinandersetzen.

Markov-Entscheidungsprozesse

Im frühen 20. Jahrhundert untersuchte der Mathematiker Andrey Markov stochastische Prozesse ohne Gedächtnis, sogenannte *Markov-Ketten*. Solche Prozesse haben eine vorgegebene Anzahl Zustände und wandern bei jedem Schritt zufällig von einem Zustand zu einem anderen. Die Wahrscheinlichkeit für den Übertritt vom Zustand s zum Zustand s' ist vorgegeben und hängt nur vom Paar (s,s') ab, nicht von vergangenen Zuständen (das System besitzt kein Gedächtnis).

Abbildung 16-7 zeigt ein Beispiel einer Markov-Kette mit vier Zuständen. Sagen wir, der Prozess beginnt im Zustand s_0 und es gibt eine 70%ige Chance, dass er im nächsten Schritt in diesem Zustand verbleibt. Irgendwann wird dieser Zustand für immer verlassen, da kein anderer Zustand wieder auf s_0 verweist. Falls der Prozess in den Zustand s_1 übergeht, fährt er höchstwahrscheinlich mit Zustand s_2 fort (90% Wahrscheinlichkeit), anschließend gleich wieder mit Zustand s_1 (100% Wahrscheinlichkeit). Er mag einige Male zwischen diesen Zuständen umherspringen, aber irgendwann in den Zustand s_3 fallen und für immer dort bleiben (dies ist ein *terminaler Zustand*). Markov-Ketten können eine sehr unterschiedliche Dynamik aufweisen und werden regelmäßig in der Thermodynamik, der Chemie, der Statistik und in vielen anderen Gebieten eingesetzt.

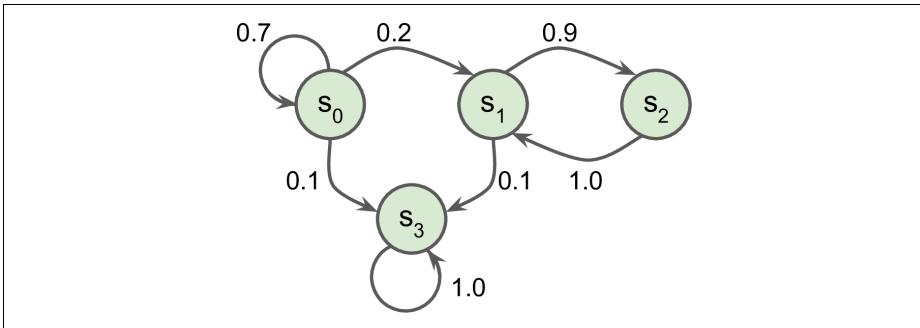


Abbildung 16-7: Beispiel einer Markov-Kette

Markov-Entscheidungsprozesse wurden erstmalig in den 1950ern von Richard Bellman beschrieben (<https://goo.gl/wZTVIN>).¹¹ Sie ähneln Markov-Ketten, aber mit einer Neuerung: Bei jedem Schritt kann sich ein Agent für eine von mehreren möglichen Aktionen entscheiden, und die Übergangswahrscheinlichkeiten hängen von der gewählten Aktion ab. Außerdem geben einige Zustandsübergänge eine (positive oder negative) Belohnung, und das Ziel des Agenten ist das Finden einer Policy, die die Belohnung über die Zeit maximiert.

Beispielsweise enthält das in Abbildung 16-8 dargestellte MDP drei Zustände und bei jedem Schritt bis zu drei mögliche diskrete Aktionen. Wenn es im Zustand s_0 beginnt, kann der Agent zwischen den Aktionen a_0 , a_1 oder a_2 wählen. Wenn er Aktion a_1 auswählt, bleibt er mit Sicherheit im Zustand s_0 und erhält keine Belohnung. Er kann sich daher entscheiden, für immer dort zu bleiben.

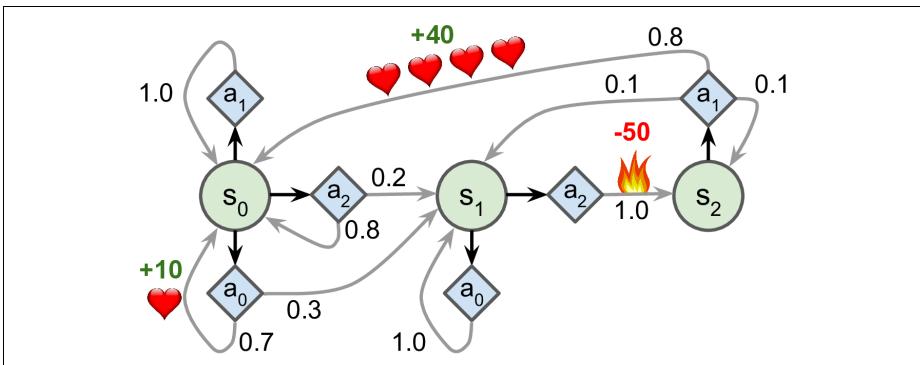


Abbildung 16-8: Beispiel für einen Markov-Entscheidungsprozess

Wenn er aber Aktion a_0 wählt, gibt es eine 70%ige Chance auf eine Belohnung von +10 und den Verbleib im Zustand s_0 . Er kann dies wieder und wieder versuchen, um so viel Belohnung wie möglich anzusammeln. Aber irgendwann landet er stattdessen im Zustand s_1 . Im Zustand s_1 gibt es nur zwei mögliche Aktionen: a_0 oder a_2 .

11 »A Markovian Decision Process«, R. Bellman (1957).

Er kann sich durch wiederholtes Wählen von Aktion a_0 ruhig verhalten oder sich für den Übergang zu Zustand s_2 entscheiden und eine negative Belohnung von -50 in Kauf zu nehmen (autsch!). Im Zustand s_2 gibt es keine andere Möglichkeit als Aktion a_1 , die höchstwahrscheinlich zum Zustand s_0 zurückführt und unterwegs eine Belohnung von $+40$ generiert. Sie verstehen, worauf es hinausläuft. Können Sie anhand dieses MDP erraten, welche Strategie langfristig den höchsten Gewinn erzielt? Im Zustand s_0 ist klar, dass Aktion a_0 die beste Wahl ist, und im Zustand s_2 hat der Agent keine Wahl außer Aktion a_1 , aber im Zustand s_1 ist nicht klar, ob der Agent sich ruhig verhalten (a_0) oder durchs Dickicht gehen sollte (a_2).

Bellman fand eine Möglichkeit zum Schätzen des *optimalen Zustandswerts* eines Zustands s , geschrieben $V^*(s)$, der der Summe aller zukünftigen Belohnungen entspricht, die der Agent nach Erreichen des Zustands s bei optimalem Verhalten im Durchschnitt erwarten kann. Er zeigte, dass bei optimalem Verhalten des Agenten das *Optimalitätsprinzip von Bellman* gilt (siehe Formel 16-1). Diese rekursive Formel besagt, dass bei optimalem Verhalten des Agenten der optimale Wert des gegenwärtigen Zustands gleich der durchschnittlichen Belohnung einer optimalen Aktion plus dem zu erwartenden optimalen Wert aller möglichen Folgezustände dieser Aktion ist.

Formel 16-1: Optimalitätsprinzip von Bellman

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{für alle } s$$

- $T(s, a, s')$ ist die Übergangswahrscheinlichkeit von Zustand s zu Zustand s' , vorausgesetzt, der Agent wählt Aktion a .
- $R(s, a, s')$ ist die Belohnung, die der Agent beim Übergang von Zustand s in Zustand s' erhält, vorausgesetzt, der Agent wählt Aktion a .
- γ ist die Discount-Rate.

Diese Gleichung führt direkt zu einem Algorithmus, mit dem sich der optimale Zustandswert jedes möglichen Zustands abschätzen lässt: Sie initialisieren zunächst die Schätzung sämtlicher Zustände mit null und aktualisieren diese iterativ mit dem *Value Iteration*-Algorithmus (siehe Formel 16-2). Ein bemerkenswertes Ergebnis ist, dass diese Schätzungen mit genug Zeit garantiert zu den optimalen Zustandswerten konvergieren, die der optimalen Policy entsprechen.

Formel 16-2: Value-Iteration-Algorithmus

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{für alle } s$$

- $V_k(s)$ ist der geschätzte Wert des Zustands s in der k^{ten} Iteration des Algorithmus.



Dieser Algorithmus ist ein Beispiel für *dynamische Programmierung*, die ein komplexes Problem (in diesem Fall das Schätzen einer potenziell unendlichen Summe zukünftiger Belohnungen) in bearbeitbare Teilprobleme aufteilt, die sich iterativ abarbeiten lassen (in diesem Fall das Finden einer Aktion, die die Summe aus durchschnittlicher Belohnung und dem verrechneten Wert des nächsten Zustands maximiert).

Die optimalen Zustandswerte zu kennen, ist vor allem zum Evaluieren einer Policy nützlich. Es verrät dem Agenten aber nicht, was zu tun ist. Glücklicherweise fand Bellman einen sehr ähnlichen Algorithmus zum Abschätzen der optimalen *Zustands-Aktions-Werte*, genannt *Q-Werte*. Der optimale Q-Werte eines Zustand-Aktion-Paares (s, a) , geschrieben $Q^*(s, a)$, ist die Summe der berechneten zukünftigen Belohnungen, die der Agent im Mittel nach Erreichen des Zustands s durch Auswahl der Aktion a erwarten kann. Dabei ist das Ergebnis der Aktion a noch nicht bekannt, und wir nehmen nach dieser Aktion optimales Verhalten an.

Und so funktioniert es: Wir initialisieren wieder sämtliche Schätzungen der Q-Werte mit null und aktualisieren diese mit dem *Q-Wert-Iterationsalgorithmus* (siehe Formel 16-3).

Formel 16-3: *Q-Wert-Iterationsalgorithmus*

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a')] \quad \text{für alle } (s, a)$$

Sobald Sie die optimalen Q-Werte gefunden haben, ist die optimale Policy $\pi^*(s)$ trivial: Wenn sich der Agent im Zustand s befindet, sollte er die Aktion mit dem höchsten Q-Wert für diesen Zustand auswählen: $\pi^*(s) = \arg\max_a Q^*(s, a)$.

Wenden wir diesen Algorithmus auf das in Abbildung 16-8 gezeigte MDP an. Zunächst definieren wir das MDP:

```
nan=np.nan # steht für unmögliche Aktionen
T = np.array([
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
    [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
])
R = np.array([
    [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
    [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
    [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Führen wir nun den Q-Wert-Iterationsalgorithmus aus:

```
Q = np.full((3, 3), -np.inf) # -inf bei unmöglichen Aktionen
for state, actions in enumerate(possible_actions):
```

```

Q[state, actions] = 0.0 # Startwert = 0.0 für alle möglichen Aktionen

discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp]))
                for sp in range(3)
            ])

```

Es ergeben sich folgende Q-Werte:

```

>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
       [ 1.11669335,          -inf,   1.17573546],
       [-inf,   53.86946068,          -inf]])
>>> np.argmax(Q, axis=1) # optimale Aktion für jeden Zustand
array([0, 2, 1])

```

Damit erhalten wir eine optimale Policy für dieses MDP mit einer Discount-Rate von 0.95: Wähle im Zustand s_0 Aktion a_0 , im Zustand s_1 Aktion a_2 (gehe durchs Feuer!) und im Zustand s_2 Aktion a_1 (die einzige Möglichkeit). Interessanterweise ändert sich die Policy, wenn Sie die Discount-Rate auf 0.9 senken: Im Zustand s_1 wird a_0 die beste Aktion (verhalte dich ruhig; gehe nicht durchs Feuer). Dies macht Sinn, weil mit einem höheren Gewicht auf der Gegenwart die Aussicht auf zukünftige Belohnungen den unmittelbaren Schmerz nicht aufwiegt.

Temporal Difference Learning und Q-Learning

Reinforcement-Learning-Aufgaben mit diskreten Aktionen lassen sich oft als Markov-Entscheidungsprozesse modellieren. Allerdings kennt der Agent zu Beginn die Übergangswahrscheinlichkeiten nicht (er kennt $T(s, a, s')$ nicht). Auch sind die Belohnungen zu Beginn unbekannt (der Agent kennt $R(s, a, s')$ nicht). Er muss jeden Zustand und jeden Übergang mindestens einmal erfahren, wenn er einen sinnvollen Schätzwert für die Übergangswahrscheinlichkeit haben soll.

Der *Temporal Difference Learning*-Algorithmus (TD Learning) ist zum Value-Iteration-Algorithmus sehr ähnlich, trägt aber dem Umstand Rechnung, dass der Agent den MDP nur teilweise kennt. Im Allgemeinen nehmen wir an, dass dem Agenten zu Beginn außer den möglichen Zuständen und Aktionen nichts weiter bekannt ist. Der Agent verwendet eine *Erkundungspolicy* – beispielsweise eine rein zufällige Policy –, um den MDP zu erkunden, und aktualisiert die Schätzungen der Zustandswerte anhand der beobachteten Übergänge und Belohnungen im Verlauf des TD-Learning-Algorithmus (siehe Formel 16-4).

Formel 16-4: TD Learning-Algorithmus

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

- α ist die Lernrate (z.B. 0.01).



Es gibt viele Ähnlichkeiten zwischen TD Learning und dem stochastischen Gradientenverfahren, besonders das Abarbeiten einer einzelnen Stichprobe. Wie das stochastische Gradientenverfahren, kann es nur wirklich konvergieren, wenn Sie die Lernrate allmählich senken (andernfalls hüpfst es um das Optimum herum).

Der Algorithmus merkt sich für jeden Zustand s den gleitenden Durchschnitt der unmittelbaren Belohnung für das Verlassen dieses Zustands zuzüglich der später zu erwartenden Belohnungen (optimales Verhalten angenommen).

Ähnlich dazu ist der Q-Learning-Algorithmus eine Anpassung des Q-Wert-Iterationsalgorithmus auf Situationen mit unbekannten Übergangswahrscheinlichkeiten und Belohnungen (siehe Formel 16-5).

Formel 16-5: Q-Learning-Algorithmus

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha \left(r + \gamma \cdot \max_{a'} Q_k(s', a') \right)$$

Dieser Algorithmus merkt sich den laufenden Durchschnitt der Belohnungen r für das Verlassen des Zustands s mit Aktion a zuzüglich der später zu erwartenden Belohnung für jedes Zustand-Aktion-Paar (s, a) . Da die Zielpolicy optimal handelt, wählen wir für den nächsten Zustand das Maximum der Q-Werte aus.

Q-Learning lässt sich folgendermaßen implementieren:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # beginne im Zustand 0

Q = np.full((3, 3), -np.inf) # -inf für unmögliche Aktionen
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Startwert = 0.0 für alle möglichen Aktionen

for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s]) # wähle eine Aktion (zufällig)
    sp = rnd.choice(range(3), p=T[s, a]) # wähle nächsten Zustand mit T[s, a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = (1 - learning_rate) * Q[s, a] + learning_rate * (
        reward + discount_rate * np.max(Q[sp]))
```

```

    )
s = sp # gehe zum nächsten Zustand

```

Mit genug Iterationen wird dieser Algorithmen bei optimalen Q-Werten konvergieren. Dies nennt man einen *off-Policy*-Algorithmus, weil die trainierte Policy nicht die ausgeführte ist. Es ist etwas überraschend, dass dieser Algorithmus die optimale Policy nicht einfach durch Beobachten eines zufälliges Agenten erlernen kann (stellen Sie sich vor, Golf zu lernen, wenn Ihr Golflehrer ein betrunkener Affe ist). Können wir dies noch verbessern?

Erkundungspolicies

Natürlich funktioniert Q-Learning nur, wenn die Erkundungspolicy den MDP gründlich genug durchsucht. Auch wenn bei einer zufallsbasierten Policy irgendwann jeder Zustand und jede Policy viele Male besucht werden, kann dies extrem lange dauern. Die ϵ -greedy Policy ist deshalb eine bessere Alternative: Bei jedem Schritt agiert sie mit der Wahrscheinlichkeit ϵ zufällig und mit der Wahrscheinlichkeit $1-\epsilon$ gierig (und wählt die Aktion mit dem höchsten Q-Wert aus). Der Vorteil der ϵ -greedy Policy (im Vergleich zur völlig zufälligen Policy) ist, dass sie mehr Zeit mit dem Erkunden der interessanten Teile der Umwelt verbringt, während die Schätzungen der Q-Werte immer genauer werden, aber trotzdem noch etwas Zeit in das Besuchen unbekannter Regionen des MDP investiert. Es ist üblich, zu Beginn einen hohen Wert für ϵ (z. B. 1.0) zu verwenden und diesen schrittweise zu reduzieren (z. B. bis auf 0.05).

Anstatt sich beim Erkunden auf den Zufall zu verlassen, könnte die Erkundungspolicy auch gezielt noch nicht ausprobierte Aktionen wählen. Dies lässt sich als Bonus auf den geschätzten Q-Werten implementieren, wie Formel 16-6 zeigt.

Formel 16-6: Q-Learning mit einer Erkundungsfunktion

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a')) \right)$$

- $N(s', a')$ zählt, wie oft Action a' im Zustand s' ausgewählt wurde.
- $f(q, n)$ ist eine *Erkundungsfunktion* wie $f(q, n) = q + K/(1 + n)$, wobei der Hyperparameter K die Neugierde festlegt, also wie stark der Agent vom Unbekannten angezogen wird.

Approximativer Q-Learning

Das Hauptproblem beim Q-Learning ist, dass es nicht besonders gut auf große (und nicht einmal mittelgroße) MDPs mit vielen Zuständen und Aktionen skaliert. Nehmen wir an, Sie versuchen, einem Agenten das Spielen von Ms. Pac-Man mit Q-Learning beizubringen. Es gibt über 250 Punkte, die Ms. Pac-Man essen kann. Jeder davon kann vorhanden oder bereits aufgegessen sein. Die Anzahl möglicher

Zustände übersteigt $2^{250} \approx 10^{75}$ (und dies berücksichtigt nur die möglichen Zustände der Punkte). Diese Zahl ist größer als die Anzahl der Atome im beobachtbaren Universum, es gibt also keine Möglichkeit, eine Schätzung für jeden einzelnen Q-Wert zu speichern.

Die Lösung ist daher, die Q-Werte eines Paares aus Zustand und Aktion über eine Funktion $Q_\theta(s, a)$ mit einer handhabbaren Anzahl Parameter zu approximieren (mit dem Parametervektor θ). Dies bezeichnet man als *approximatives Q-Learning*. Viele Jahre lang wurde empfohlen, Linearkombinationen von Hand aus dem Zustand extrahierter Merkmale zu verwenden (z.B. die Entfernung zum nächsten Geist, ihre Richtungen und so weiter), um Q-Werte abzuschätzen. DeepMind konnte jedoch demonstrieren, dass neuronale Netze besonders bei komplexen Aufgaben viel besser funktionieren und keinerlei Extraktion von Merkmalen nötig ist. Ein DNN zum Schätzen von Q-Werten nennt man ein *Deep-Q-Netz* (DQN), und das Verwenden eines DQN für approximatives Q-Learning heißt *Deep-Q-Learning*.

Wie lässt sich also ein DQN trainieren? Betrachten wir den vom DQN für ein gegebenes Paar aus Zustand und Aktion (s, a) berechneten approximierten Q-Wert. Dank Bellman wissen wir, dass dieser approximierte Q-Wert so nah wie möglich an der Belohnung r liegen soll, die wir nach dem Spielen der Aktion a in Zustand s erhalten, zuzüglich der mit Discount verrechneten zukünftigen Belohnungen für optimales Spiel. Um diese zukünftigen Belohnungen abzuschätzen, können wir das DQN einfach auf dem nächsten Zustand s' für alle möglichen Aktionen a' ausführen. Wir erhalten einen approximierten zukünftigen Q-Wert für jede mögliche Aktion. Dann wählen wir den höchsten Wert aus (unter der Annahme, dass wir optimal spielen), verrechnen den Discount und erhalten so eine Schätzung der zukünftigen Werte mit Discount. Indem wir die Belohnung r und die Schätzung des zukünftigen Werts mit Discount aufsummieren, erhalten wir den Ziel-Q-Wert $y(s, a)$ für das Paar aus Zustand und Aktion (s, a) , wie in Formel 16-7.

Formel 16-7: Ziel-Q-Wert

$$y(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

Mit diesem Ziel-Q-Wert können wir eine Trainingsiteration nach dem Gradientenverfahren durchführen. Insbesondere versuchen wir, den quadratischen Fehler zwischen dem geschätzten Q-Wert und dem Ziel-Q-Wert zu minimieren. Und das ist für den grundlegenden Deep-Q-Learning-Algorithmus auch schon alles!

Allerdings kamen im DQN-Algorithmus von DeepMind zwei entscheidende Modifikationen hinzu:

- Anstatt das DQN auf den jüngsten Erfahrungen zu trainieren, speichert der DQN-Algorithmus von DeepMind die Erfahrungen in einem grossen Replay-Speicher und zieht in jeder Trainingsiteration daraus einen zufälligen Trainings-Batch. Dies hilft dabei, die Korrelation zwischen den Erfahrungen in einem Trainings-Batch zu reduzieren, was beim Trainieren ungemein hilft.

- Der Algorithmus verwendet zwei DQNs anstatt eines: Das erste wird Online-DQN genannt und ist dasjenige, das spielt und bei jedem Trainingsschritt dazulernnt. Das zweite wird Ziel-DQN genannt und wird nur zum Berechnen der Ziel-Q-Werte verwendet (Formel 16-7). In regelmäßigen Abständen werden die Gewichte des Online-Netzes auf das Ziel-Netz kopiert. DeepMind konnte zeigen, dass diese Änderung die Leistung des Algorithmus erheblich verbesserte. Tatsächlich entsteht ohne diese Änderung ein Netz, das seine eigenen Ziele setzt und dann verfolgt, etwa so, als würde ein Hund seinen eigenen Schwanz verfolgen. Das kann zu Feedback-Schleifen führen, die das Netz destabilisieren (es kann divergieren, oszillieren, einfrieren und so weiter). Zwei Netzen wirken diesen Feedback-Schleifen entgegen und stabilisieren den Trainingsprozess.

Im restlichen Kapitel werden wir Deep-Q-Learning verwenden, um einem Agenten das Spielen von Ms. Pac-Man beizubringen, wie es DeepMind im Jahr 2013 tat. Der Code lässt sich leicht für fast alle Atari-Spiele anpassen und lernt diese recht gut zu meistern, ein ausreichend langes Training vorausgesetzt (es kann je nach Hardware Tage oder Wochen dauern). Bei den meisten Actionspielen erreicht er übermenschliche Fähigkeiten, ist jedoch bei Spielen mit länger andauernden Handlungen weniger gut.

Ms. Pac-Man mit dem DQN-Algorithmus spielen lernen

Weil wir eine Atari-Umgebung verwenden, müssen wir zunächst die Atari-Abhängigkeiten von OpenAI Gym installieren. Und wenn wir schon dabei sind, installieren wir auch gleich die Pakete für andere OpenAI-Gym-Umgebungen, die Sie ausprobieren können. Auf macOS müssen Sie beim Verwenden von *Homebrew* (<http://brew.sh/>) dazu Folgendes eingeben:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

Auf Ubuntu geben Sie den folgenden Befehl ein (ersetzen Sie `python3` durch `python`, falls Sie Python 2 verwenden):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-dev\x
xvfb libav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev swig
```

Anschließend installieren wir die zusätzlichen Python-Module:

```
$ pip3 install --upgrade 'gym[all]'
```

Wenn alles gut gegangen ist, sollten Sie eine Ms.-Pac-Man-Umgebung erzeugen können:

```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape # [Höhe, Breite, Farbkanäle]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

Wie Sie sehen, gibt es neun mögliche diskrete Aktionen, die den neun Positionen des Joysticks entsprechen (mittig, oben, rechts, links, unten, links oben und so weiter), und die Beobachtungen sind lediglich als 3-D-NumPy-Arrays abgelegte Screenshots des Atari-Bildschirms (siehe Abbildung 16-9, links). Diese Bilder sind ein wenig zu groß. Wir schreiben daher eine kleine Funktion zur Vorverarbeitung, die das Bild zurechtschneidet, auf 88×80 Pixel verkleinert, in Graustufen umwandelt und den Kontrast von Ms. Pac-Man verbessert. Damit sinkt der im DQN nötige Rechenaufwand, und die Trainingsgeschwindigkeit steigt.

```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # zurechtschneiden und skalieren
    img = img.mean(axis=2) # als Graustufen
    img[img==mspacman_color] = 0 # Kontrast verbessern
    img = (img - 128) / 128 - 1 # normalisieren von -1. bis 1.
    return img.reshape(88, 80, 1)
```

Das Ergebnis der Vorverarbeitung sehen Sie in Abbildung 16-9 (rechts).

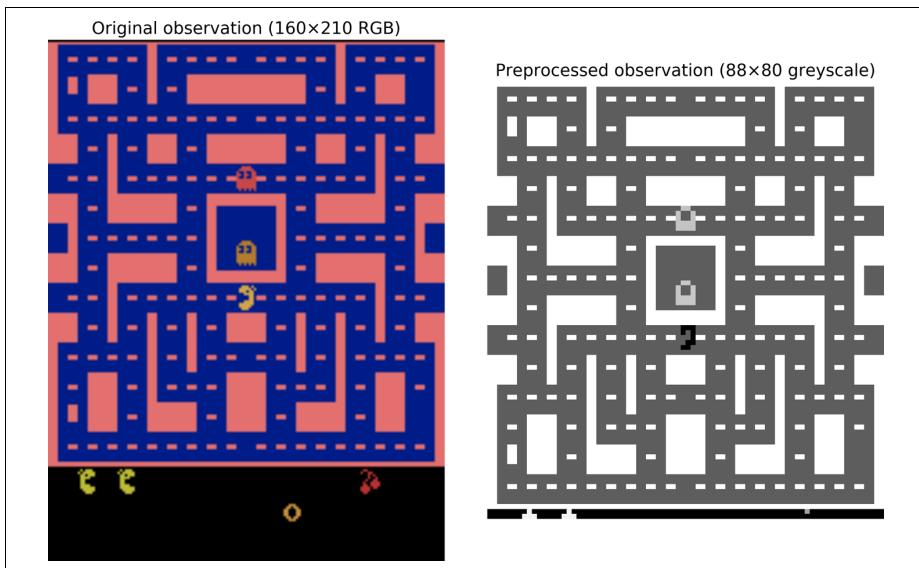


Abbildung 16-9: Beobachtung für Ms. Pac-Man, Original (links) und nach Vorverarbeitung (rechts)

Als Nächstes erstellen wir das DQN. Es könnte einfach ein Zustand-Aktion-Paar (s, a) als Eingabe verwenden und eine Schätzung des entsprechenden Q-Werts $Q(s, a)$ ausgeben, aber da die Aktionen diskret sind, ist ein neuronales Netz mit nur einem Zustand s als Eingabe und einem geschätzten Q-Wert pro Aktion als Ausgabe praktischer und effizienter. Das DQN setzt sich aus drei Convolutional Layers gefolgt von zwei vollständig verbundenen Schichten inklusive der Ausgabeschicht zusammen (siehe Abbildung 16-10).

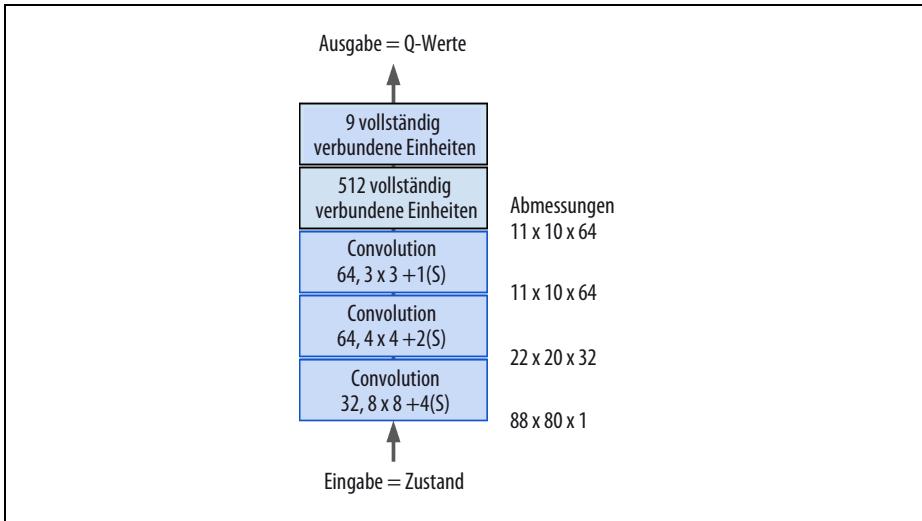


Abbildung 16-10: Deep-Q-Netz zum Spielen von Ms. Pac-Man

Wie bereits besprochen, erfordert der von DeepMind erstellte Trainingsalgorithmus zwei DQNs mit der gleichen Architektur (aber unterschiedlichen Parametern): Das Online-DQN wird lernen, Ms. Pac-Man zu steuern, und das Ziel-DQN wird zum Aufbau der Ziel-Q-Werte zum Trainieren des Online-DQN verwendet. In regelmäßigen Abständen kopieren wir das Online-DQN an das Ziel-DQN und ersetzen dessen Parameter. Da wir zwei identische DQNs mit der gleichen Architektur benötigen, schreiben wir uns die Funktion `q_network()`, um diese zu erstellen:

```

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"] * 3
conv_activation = [tf.nn.relu] * 3
n_hidden_in = 64 * 11 * 10 # conv3 enthält 64 Maps der Größe 11x10
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n # es gibt 9 diskrete Aktionen
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, name):
    prev_layer = X_state
    with tf.variable_scope(name) as scope:
        for n_maps, kernel_size, strides, padding, activation in zip(
            conv_n_maps, conv_kernel_sizes, conv_strides,
            conv_paddings, conv_activation):
            prev_layer = tf.layers.conv2d(
                prev_layer, filters=n_maps, kernel_size=kernel_size,

```

```

        stride=strides, padding=padding, activation=activation,
        kernel_initializer=initializer)
last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
hidden = tf.layers.dense(last_conv_layer_flat, n_hidden,
                        activation=hidden_activation,
                        kernel_initializer=initializer)
outputs = tf.layers.dense(hidden, n_outputs,
                        kernel_initializer=initializer)
trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                   scope=scope.name)
trainable_vars_by_name = {var.name[len(scope.name)]: var
                         for var in trainable_vars}
return outputs, trainable_vars_by_name

```

Der erste Teil dieses Codes definiert die Hyperparameter der DQN-Architektur. Anschließend erstellt die Funktion `q_network()` das DQN anhand des Zustands der Umwelt `X_state` als Eingabe und eines Namens für den Scope. Wir werden nur eine Beobachtung zur Repräsentation des Zustands der Umwelt verwenden, da es bei nahe keine verborgenen Zustände gibt (außer den blinkenden Gegenständen und den Richtungen der Geister).



Spiele wie Pong oder Breakout enthalten einen sich bewegenden Ball, dessen Richtung und Geschwindigkeit sich nicht durch eine einzige Beobachtung ermitteln lassen. Sie erfordern also eine Kombination der letzten Beobachtungen. Eine Möglichkeit dazu wäre ein Bild, in dem jede der letzten Beobachtungen einen Kanal darstellt. Alternativ könnten wir die letzten Beobachtungen zu einem einzigen Kanal vereinigen, indem wir das Maximum dieser Beobachtungen berechnen (nach Abschwächen der älteren Beobachtungen, sodass die Zeitachse im endgültigen Bild deutlich wird).

Das Dictionary `trainable_vars_by_name` enthält sämtliche trainierbaren Variablen dieses DQN. Es wird sich in Kürze als nützlich erweisen, wenn wir Operationen zum Kopieren des Online-DQN in das Ziel-DQN definieren. Die Schlüssel des Dictionary sind einfach die Namen der Variablen ohne das Präfix mit dem Namen des Scope. Es sieht folgendermaßen aus:

```

>>> trainable_vars_by_name
{'conv2d/bias:0': <tf.Variable... (shape=(32,), dtype=float32_ref,
'conv2d/kernel:0': <tf.Variable... (shape=(8, 8, 1, 32) dtype=float32_ref,
'conv2d_1/bias:0': <tf.Variable... (shape=(64,), dtype=float32_ref,
'conv2d_1/kernel:0': <tf.Variable... (shape=(4, 4, 32, 64) dtype=float32_ref,
'conv2d_2/bias:0': <tf.Variable... (shape=(64,), dtype=float32_ref,
'conv2d_2/kernel:0': <tf.Variable... (shape=(3, 3, 64, 64) dtype=float32_ref,
'dense/bias:0': <tf.Variable... (shape=(512,), dtype=float32_ref,
'dense/kernel:0': <tf.Variable... (shape=(7040, 512) dtype=float32_ref,
'dense_1/bias:0': <tf.Variable... (shape=(9,), dtype=float32_ref,
'dense_1/kernel:0': <tf.Variable... (shape=(512, 9) dtype=float32_ref}

```

Erstellen wir den Platzhalter für die Eingabe, die zwei DQNs und die Operation zum Kopieren des Online-DQN auf das Ziel-DQN:

```

X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                             input_channels])
online_q_values, online_vars = q_network(X_state, name="q_networks/online")
target_q_values, target_vars = q_network(X_state, name="q_networks/target")

copy_ops = [target_var.assign(online_vars[var_name])
            for var_name, target_var in target_vars.items()]
copy_online_to_target = tf.group(*copy_ops)

```

Fassen wir zusammen: Wir haben nun zwei DQNs, die beide einen Zustand der Umgebung (eine vorverarbeitete Beobachtung) als Eingabe verwenden und für jede mögliche Aktion in diesem Zustand einen Q-Wert ausgeben. Außerdem haben wir eine Operation namens `online_to_target`, mit der wir sämtliche trainierbaren Variablen des Online-DQN auf das Ziel-DQN kopieren können. Wir rufen die TensorFlow-Funktion `tf.group()` auf, um sämtliche Zuweisungen in einer bequemen Operation zu versammeln.

Fügen wir nun die Operationen zum Trainieren des Online-DQN hinzu. Zuerst müssen wir für jedes Zustand-Aktion-Paar in der Stichprobe von Erinnerungen die vorhergesagten Q-Werte berechnen. Da das DQN einen Q-Wert pro mögliche Aktion ausgibt, müssen wir nur den Q-Wert der tatsächlich gespielten Aktion verwenden. Dazu wandeln wir die Aktion in einen One-Hot-codierten Vektor um (also einen Vektor aus Nullen außer einer 1 beim i^{ten} Index) und multiplizieren diesen mit den Q-Werten: Damit werden alle Q-Werte bis auf den der gespeicherten Aktion herausgekürzt. Anschließend bilden wir eine Summe über die erste Achse, um nur die Vorhersage des Q-Werts für jede Erinnerung zu erhalten.

```

X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(target_q_values * tf.one_hot(X_action, n_outputs),
                        axis=1, keep_dims=True)

```

Nun erstellen wir einen Platzhalter `y` für die Q-Zielwerte und berechnen die Verlustfunktion: Wir verwenden die quadratische Abweichung, wenn diese kleiner als 1.0 ist, und den doppelten absoluten Fehler, wenn die quadratische Abweichung größer als 1.0 ist. Anders ausgedrückt, ist die Verlustfunktion für kleine Fehler quadratisch und für große Fehler linear. Dadurch verringern sich die Auswirkungen großer Fehler, und das Training stabilisiert sich.

```

y = tf.placeholder(tf.float32, shape=[None, 1])
error = tf.abs(y - q_value)
clipped_error = tf.clip_by_value(error, 0.0, 1.0)
linear_error = 2 * (error - clipped_error)
loss = tf.reduce_mean(tf.square(clipped_error) + linear_error)

```

Schließlich erstellen wir einen Optimizer mit dem Gradientenverfahren nach Nesterov, um die Verlustfunktion zu minimieren. Wir erstellen außerdem eine nicht trainierbare Variable `global_step`, um die Trainingsschritte zu zählen. Die Trainingsoperation kümmert sich um das Hochzählen. Wir erstellen natürlich auch die übliche Operation zum Initialisieren und einen Saver.

```

global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum, use_nesterov=True)
training_op = optimizer.minimize(loss, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

Damit ist die Konstruktionsphase abgeschlossen. Bevor wir uns die Ausführungsphase ansehen, benötigen wir einige Hilfsmittel. Implementieren wir zuerst den Replay-Speicher. Wir verwenden dazu eine deque-Liste, da sich damit sehr effizient Einträge hinzufügen und wieder entfernen lassen, falls die Liste ihre maximale Größe im Speicher erreicht. Wir schreiben außerdem eine kleine Funktion, die eine zufällige Stichprobe aus dem Replay-Speicher auswählt. Jede Erfahrung ist ein 5-Tupel (Zustand, Aktion, Belohnung, folgender Zustand, Continue), wobei das Element »Continue« gleich 0.0 ist, wenn das Spiel zu Ende ist, ansonsten 1.0.

```

from collections import deque

replay_memory_size = 500000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = np.random.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # Zustand, Aktion, Belohnung, Folgezustand, Continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3], cols[4].reshape(-1, 1))

```

Nun muss der Aktor das Spiel erkunden. Wir verwenden die ε -greedy Policy und senken ε schrittweise innerhalb von 2000000 Trainingsschritten von 1.0 auf 0.05 ab:

```

eps_min = 0.1
eps_max = 1.0
eps_decay_steps = 2000000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # zufällige Aktion
    else:
        return np.argmax(q_values) # optimale Aktion

```

Das ist es! Wir haben alles, was zum Trainieren nötig ist. Die Ausführungsphase ist nicht besonders komplex, der Code jedoch etwas lang. Also holen Sie noch einmal Luft. Bereit zum Endspurt? Los geht's! Initialisieren wir zunächst einige Parameter:

```

n_steps = 4000000 # Gesamtzahl Trainingsschritte
training_start = 10000 # starte Training nach 10000 Iterationen

```

```

training_interval = 4 # ein Trainingsschritt alle 4 Iterationen
save_steps = 1000 # speichere Modell alle 1000 Trainingsschritte
copy_steps = 10000 # kopiere Online-DQN auf Ziel-DQN alle 10000 Trainingsschritte
discount_rate = 0.99
skip_start = 90 # Überspringe den Beginn jeder Partie (reine Wartezeit)
batch_size = 50
iteration = 0 # Iterationen im Spiel
checkpoint_path = "./my_dqn.ckpt"
done = True # env muss zurückgesetzt werden

```

Öffnen wir die Session und führen wir die Trainingsschleife aus:

```

with tf.Session() as sess:
    if os.path.isfile(checkpoint_path + ".index"):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
        copy_online_to_target.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        if done: # game over, start again
            obs = env.reset()
            for skip in range(skip_start): # skip the start of each game
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

        # Online-DQN evaluiert, was zu tun ist
        q_values = online_q_values.eval(feed_dict={X_state: [state]})
        action = epsilon_greedy(q_values, step)

        # Online-DQN spielt
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # merkt sich, was passiert ist
        replay_memory.append((state, action, reward, next_state, 1.0 - done))
        state = next_state

        if iteration < training_start or iteration % training_interval != 0:
            continue # only train after warmup period and at regular intervals

        # Stichprobe aus Erinnerungen,
        # verwende Ziel-DQN, um den Ziel-Q-Wert zu ermitteln
        X_state_val, X_action_val, rewards, X_next_state_val, continues = (
            sample_memories(batch_size))
        next_q_values = target_q_values.eval(
            feed_dict={X_state: X_next_state_val})
        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
        y_val = rewards + continues * discount_rate * max_next_q_values

```

```

# trainiere das Online-DQN
training_op.run(feed_dict={X_state: X_state_val,
                           X_action: X_action_val, y: y_val})

# kopiere das Online-DQN in das Ziel-DQN
if step % copy_steps == 0:
    copy_online_to_target.run()

# speichere regelmäßig ab
if step % save_steps == 0:
    saver.save(sess, checkpoint_path)

```

Zuerst stellen wir die Modelle her, falls eine Datei mit einem Zwischenstand existiert. Ansonsten initialisieren wir die Variablen wie gewohnt und kopieren das Online-DQN in das Ziel-DQN. Dann beginnt die Hauptschleife, in der iteration die durchgeführten Schritte im Spiel seit Programmbeginn und step die Anzahl der Trainingsschritte nach Beginn des Trainierens zählt (dieser wird ebenfalls aus dem Zwischenstand wiederhergestellt). Anschließend wird das Spiel zurückgesetzt (und die ersten Schritte des Spiels werden übersprungen, weil dort nichts passiert). Anschließend evaluiert das Online-DQN, was zu tun ist, spielt und speichert seine Erfahrungen im Replay-Speicher. In regelmäßigen Abständen (nach einer Warmlaufphase) durchläuft das Online-DQN einen Trainingsschritt. Es bildet zuerst eine Stichprobe aus Erinnerungen und bittet das Ziel-DQN, die Q-Werte sämtlicher Aktionen für den Folgezustand abzuschätzen. Es verwendet Formel 16-7, um den Q-Zielwert y_val mit dem nächsten Paar aus Zustand und Aktion zu berechnen. Der einzige schwierige Teil ist hierbei, dass wir den Vektor `max_next_qu_values` mit dem Vektor `continues` multiplizieren müssen, um Q-Werte für Erinnerungen auf Null zu setzen, bei denen das Spiel zu Ende war. Anschließend führen wir eine Trainingsoperation aus, um die Vorhersagen des Online-DQN für Q-Werte zu verbessern. Schließlich kopieren wir das Online-DQN in regelmäßigen Abständen in das Ziel-DQN und speichern das Modell.



Leider ist das Training sehr langsam: Wenn Sie Ihren Laptop zum Trainieren verwenden, kann es Tage dauern, bevor Ms. Pac-Man etwas zustande bringt. Wenn Sie sich die Lernkurve der durchschnittlichen Belohnung pro Spiel ansehen oder den maximalen vom Online-DQN berechneten Q-Wert ermitteln und den Mittelwert dieser maximalen Q-Werte im Verlauf jedes Spiels verfolgen, fällt ein sehr starkes Rauschen auf. An einigen Punkten kann es eine lange Zeit überhaupt keinen erkennbaren Fortschritt geben, bis der Agent plötzlich lernt, einen längeren Zeitraum zu überleben. Wie oben erwähnt, können Sie in dieser Situation dem Modell so viel vorhandenes Wissen wie möglich zur Verfügung stellen (in Form von Vorverarbeitung, Belohnungen und so weiter). Sie können auch ein Bootstrapping des Modells durchführen, indem Sie es zunächst auf das Imitieren einer einfachen Strategie trainieren. In jedem Fall benötigt RL noch immer eine Menge Geduld und Feinabstimmung, aber das Endergebnis ist sehr spannend.

Übungen

1. Wie würden Sie Reinforcement Learning definieren? Wie unterscheidet es sich von gewöhnlichem überwachten oder unüberwachten Lernen?
2. Lassen Sie sich drei mögliche Anwendungen von RL einfallen, die in diesem Kapitel nicht erwähnt wurden. Was ist die jeweilige Umgebung? Was ist der Agent? Welche möglichen Aktionen gibt es? Was sind die Belohnungen?
3. Wofür steht die Discount-Rate? Kann sich die optimale Policy ändern, wenn Sie die Discount-Rate modifizieren?
4. Wie lässt sich die Leistung eines Agenten beim Reinforcement Learning messen?
5. Was ist das Credit-Assignment-Problem? Unter welchen Umständen tritt es auf? Wie können Sie es entschärfen?
6. Wozu ist ein Replay-Speicher gut?
7. Was ist ein off-Policy-RL-Algorithmus?
8. Verwenden Sie Policy Gradienten, um »BipedalWalker-v2« auf OpenAI Gym zu bewältigen.
9. Verwenden Sie Policy-Gradienten, um einen Agenten im berühmten Atari-Spiel *Pong* zu trainieren (*Pong-v0* in OpenAI Gym). Achtung: Eine einzelne Beobachtung reicht nicht aus, um Richtung und Geschwindigkeit des Balls zu bestimmen.
10. Wenn Sie etwa 100 USD übrig haben, können Sie sich einen Raspberry Pi 3 und einige billige Bauteile für Robotik zulegen. Installieren Sie TensorFlow auf dem Pi und toben Sie sich aus! Lesen Sie sich diesen unterhaltsamen Blogpost (<https://goo.gl/Eu5u28>) von Lukas Biewald durch oder schauen Sie sich GoPiGo oder BrickPi an. Warum sollten Sie nicht CartPole in der Realität nachbauen und den Roboter mit Policy-Gradienten trainieren? Oder Sie bauen eine Roboterspinne, die laufen lernt; geben Sie ihr jedes Mal eine Belohnung, wenn sie sich einem Ziel nähert (dazu benötigen Sie Sensoren zur Entfernungsmessung). Ihre Vorstellungskraft ist die einzige Grenze.

Lösungen zu diesen Übungen finden Sie in Anhang A.

Vielen Dank!

Bevor wir gemeinsam das letzte Kapitel dieses Buchs beenden, möchte ich Ihnen dafür danken, dass Sie es bis zum letzten Absatz gelesen haben. Ich hoffe, dass Sie so viel Spaß beim Lesen dieses Buch wie ich beim Schreiben hatten, und dass es für Ihre großen und kleinen Projekte nützlich sein wird.

Wenn Sie Fehler finden, schicken Sie diese bitte ein. Ganz allgemein würde ich mich freuen zu wissen, was Sie denken, bitte zögern Sie daher nicht, mich über O'Reilly oder über das GitHub-Projekt *ageron/handson-ml* zu kontaktieren.

Mein wichtigster Ratschlag an Sie ist zu üben und zu üben: Arbeiten Sie alle Übungen durch, falls Sie das nicht ohnehin schon getan haben, experimentieren Sie mit den Jupyter Notebooks herum und treten Sie Kaggle.com oder einer anderen ML-Community bei. Schauen Sie sich ML-Kurse an, lesen Sie Fachartikel, besuchen Sie Konferenzen und treffen Sie Experten. Sie können sich auch einige Themen anschauen, die dieses Buch ausgespart hat, beispielsweise Empfehlungssysteme, Clustering-Algorithmen, Algorithmen zur Erkennung von Anomalien und genetische Algorithmen.

Meine Hoffnung ist, dass dieses Buch Sie zum Entwickeln einer ML-Anwendung beflügelt, die uns allen nützt! Was wird es wohl für eine sein?

Aurélien Géron, 26. November, 2016

Lösungen zu den Übungsaufgaben



Lösungen zu allen Programmierübungen sind online in den Jupyter Notebooks auf <https://github.com/ageron/handson-ml> zu finden.

Kapitel 1: Die Machine-Learning-Umgebung

1. Beim Machine Learning geht es um das Konstruieren von Systemen, die aus Daten lernen können. Lernen bedeutet, sich bei einer Aufgabe anhand eines Qualitätsmaßes zu verbessern.
2. Machine Learning ist zum Lösen komplexer Aufgaben geeignet, bei denen es keine algorithmische Lösung gibt, zum Ersetzen langer Listen händisch erstellter Regeln, zum Erstellen von Systemen, die sich an wechselnde Bedingungen anpassen und schließlich dazu, Menschen beim Lernen zu helfen (z.B. beim Data Mining).
3. Ein gelabelter Trainingsdatensatz ist ein Trainingsdatensatz, der die gewünschte Lösung (das Label) für jeden Datenpunkt enthält.
4. Die zwei verbreitetsten Aufgaben beim überwachten Lernen sind Regression und Klassifikation.
5. Verbreitete unüberwachte Lernaufgaben sind Clustering, Visualisierung, Dimensionsreduktion und das Erlernen von Assoziationsregeln.
6. Reinforcement Learning funktioniert voraussichtlich am besten, wenn ein Roboter lernen soll, in unbekanntem Gelände zu laufen, da Reinforcement Learning für diese Art Aufgabe verbreitet ist. Die Aufgabe ließe sich auch als überwachte oder unüberwachte Aufgabe formulieren, diese Herangehensweise wäre aber weniger natürlich.
7. Wenn Sie nicht wissen, wie Sie die Gruppen definieren sollen, können Sie ein Clustering-Verfahren verwenden (unüberwachtes Lernen), um Ihre Kunden in Cluster jeweils ähnlicher Kunden zu segmentieren. Wenn Sie dagegen die gewünschten Gruppen bereits kennen, können Sie einem Klassifikationsalgo-

rithmus viele Beispiele aus jeder Gruppe zeigen (überwachtes Lernen) und alle Kunden in diese Gruppen einordnen lassen.

8. Spam-Erkennung ist eine typische überwachte Lernaufgabe: Dem Algorithmus werden viele E-Mails und deren Labels (Spam oder Nicht-Spam) bereitgestellt.
9. Ein Online-Lernsystem kann im Gegensatz zu einem Batch-Lernsystem inkrementell lernen. Dadurch ist es in der Lage, sich sowohl an sich schnell ändernde Daten oder autonome Systeme anzupassen als auch sehr große Mengen an Trainingsdaten zu verarbeiten.
10. Out-of-Core-Algorithmen können riesige Datenmengen verarbeiten, die nicht in den Hauptspeicher des Computers passen. Ein Out-of-Core-Lernalgorithmus teilt die Daten in Mini-Batches ein und verwendet Techniken aus dem Online-Learning, um aus diesen Mini-Batches zu lernen.
11. Ein instanzbasiertes Lernsystem lernt die Trainingsdaten auswendig; anschließend wendet es ein Ähnlichkeitsmaß auf neue Datenpunkte an, um die dazu ähnlichsten erlernten Datenpunkte zu finden und diese zur Vorhersage zu verwenden.
12. Ein Modell besitzt einen oder mehr Modellparameter, die festlegen, wie Vorhersagen für einen neuen Datenpunkt getroffen werden (z.B. die Steigung eines linearen Modells). Ein Lernalgorithmus versucht, optimale Werte für diese Parameter zu finden, sodass das Modell bei neuen Daten gut verallgemeinert kann. Ein Hyperparameter ist ein Parameter des Lernalgorithmus selbst, nicht des Modells (z.B. die Menge zu verwendender Regularisierung).
13. Modellbasierte Lernalgorithmen suchen nach einem optimalen Wert für die Modellparameter, sodass das Modell gut auf neue Datenpunkte verallgemeinert. Normalerweise trainiert man solche Systeme durch Minimieren einer Kostenfunktion. Diese misst, wie schlecht die Vorhersagen des Systems auf den Trainingsdaten sind, zudem wird im Falle von Regularisierung ein Strafterm für die Komplexität des Modells zugewiesen. Zum Treffen von Vorhersagen geben wir die Merkmale neuer Datenpunkte in die Vorhersagefunktion des Modells ein, wobei die vom Lernalgorithmus gefundenen Parameter verwendet werden.
14. Zu den Hauptschwierigkeiten beim Machine Learning gehören fehlende Daten, mangelhafte Datenqualität, nicht repräsentative Daten, nicht informative Merkmale, übermäßig einfache Modelle, die die Trainingsdaten underfitten und übermäßig komplexe Modelle, die die Trainingsdaten overfitten.
15. Wenn ein Modell auf den Trainingsdaten herausragend abschneidet, aber schlecht auf neue Datenpunkte verallgemeinert, liegt vermutlich Overfitting der Trainingsdaten vor (oder wir hatten bei den Trainingsdaten eine Menge Glück). Gegenmaßnahmen bei Overfitting sind das Beschaffen zusätzlicher Daten, das Vereinfachen des Modells (Auswählen eines einfacheren Algorith-

mus, Reduzieren der Parameteranzahl oder Regularisierung des Modells) oder das Verringern des Rauschens in den Trainingsdaten.

16. Ein Testdatensatz hilft dabei, den Verallgemeinerungsfehler eines Modells auf neuen Datenpunkten abzuschätzen, bevor ein Modell in einer Produktionsumgebung eingesetzt wird.
17. Ein Validierungsdatensatz wird zum Vergleichen von Modellen verwendet. Es ist damit möglich, das beste Modell auszuwählen und die Feineinstellung der Hyperparameter vorzunehmen.
18. Wenn Sie Hyperparameter mit den Testdaten einstellen, riskieren Sie ein Overfitting des Testdatensatzes. Der gemessene Verallgemeinerungsfehler ist dann zu niedrig angesetzt (Sie könnten in diesem Fall also ein Modell einsetzen, das schlechter funktioniert als erwartet).
19. Kreuzvalidierung ist eine Technik, mit der Sie Modelle vergleichen können (zur Parameterauswahl und zum Einstellen von Hyperparametern), ohne dass Sie einen separaten Validierungsdatensatz benötigen. Damit sparen Sie wertvolle Trainingsdaten ein.

Kapitel 2: Ein Machine-Learning-Projekt vom Anfang bis zum Ende

Die Lösungen finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 3: Klassifikation

Die Lösungen finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 4: Trainieren von Modellen

1. Wenn Sie einen Trainingsdatensatz mit Millionen Merkmalen haben, können Sie das stochastische Gradientenverfahren oder das Mini-Batch-Gradientenverfahren verwenden. Wenn die Trainingsdaten in den Speicher passen, funktioniert eventuell auch das Batch-Gradientenverfahren. Die Normalengleichung funktioniert jedoch nicht, weil die Komplexität der Berechnung schnell (mehr als quadratisch) mit der Anzahl Merkmale ansteigt.
2. Wenn die Merkmale in Ihrem Trainingsdatensatz sehr unterschiedlich skaliert sind, hat die Kostenfunktion die Gestalt einer länglichen Schüssel. Deshalb benötigen die Algorithmen für das Gradientenverfahren lange zum Konvergieren. Um dieses Problem zu beheben, sollten Sie die Daten skalieren, bevor Sie

das Modell trainieren. Die Normalengleichung funktioniert auch ohne Skalierung. Darüber hinaus können regularisierte Modelle mit nicht skalierten Merkmalen bei einer suboptimalen Lösung konvergieren: Weil die Regularisierung große Gewichte abstrahrt, werden Merkmale mit geringen Beträgen im Vergleich zu Merkmalen mit großen Beträgen tendenziell ignoriert.

3. Das Gradientenverfahren kann beim Trainieren eines logistischen Regressionsmodells nicht in einem lokalen Minimum stecken bleiben, weil die Kostenfunktion konvex ist.¹
4. Wenn das Optimierungsproblem konvex (wie bei der linearen oder logistischen Regression) und die Lernrate nicht zu hoch ist, finden sämtliche algorithmischen Varianten des Gradientenverfahrens das globale Optimum und führen zu sehr ähnlichen Modellen. Allerdings konvergieren das stochastische und das Mini-Batch-Gradientenverfahren nicht wirklich (es sei denn, Sie reduzieren die Lernrate), sondern springen um das globale Optimum herum. Das bedeutet, dass diese Algorithmen geringfügig unterschiedliche Modelle hervorbringen, selbst wenn Sie sie lange laufen lassen.
5. Wenn der Validierungsfehler nach jeder Epoche immer wieder steigt, ist die Lernrate möglicherweise zu hoch und der Algorithmus divergiert. Wenn auch der Trainingsfehler steigt, ist dies mit Sicherheit die Ursache, und Sie sollten die Lernrate senken. Falls der Trainingsfehler aber nicht steigt, overfittet Ihr Modell die Trainingsdaten, und Sie sollten das Trainieren abbrechen.
6. Wegen des Zufallselements gibt es weder beim stochastischen noch beim Mini-Batch-Gradientenverfahren eine Garantie für Fortschritte bei jeder Iteration. Wenn Sie also das Trainieren abbrechen, sobald der Validierungsfehler steigt, kann es passieren, dass Sie vor Erreichen des Optimums abbrechen. Es ist günstiger, das Modell in regelmäßigen Abständen abzuspeichern und das beste gespeicherte Modell aufzugreifen, falls es sich eine längere Zeit nicht verbessert (es also vermutlich den eigenen Rekord nicht knacken wird).
7. Die Trainingsiterationen sind beim stochastischen Gradientenverfahren am schnellsten, da es nur genau einen Trainingsdatenpunkt berücksichtigt. Es wird also normalerweise die Umgebung des globalen Optimums als Erstes erreichen (oder das Mini-Batch-Gradientenverfahren mit sehr kleinen Mini-Batches). Allerdings wird nur das Batch-Gradientenverfahren mit genug Trainingszeit auch konvergieren. Wie erwähnt, springen das stochastische und das Mini-Batch-Gradientenverfahren um das Optimum herum, es sei denn, Sie senken die Lernrate allmählich.
8. Wenn der Validierungsfehler deutlich höher als der Trainingsfehler ist, liegt es daran, dass Ihr Modell die Trainingsdaten overfittet. Dies lässt sich beheben, indem Sie den Grad des Polynoms senken: Ein Modell mit weniger Freiheits-

¹ Wenn Sie zwei beliebige Punkte der Kurve über eine gerade Linie verbinden, schneidet diese niemals die Kurve.

graden neigt weniger zu Overfitting. Sie können auch versuchen, das Modell zu regularisieren – beispielsweise über einen ℓ_2 -Strafterm (Ridge) oder einen ℓ_1 -Strafterm (Lasso), der zur Kostenfunktion addiert wird. Damit reduzieren Sie auch die Freiheitsgrade des Modells. Schließlich können Sie auch die Größe des Trainingsdatensatzes erhöhen.

9. Wenn der Trainingsfehler und der Validierungsfehler fast gleich und recht hoch liegen, liegt vermutlich Underfitting der Trainingsdaten vor. Es gibt also ein hohes Bias. Sie sollten daher den Hyperparameter zur Regularisierung α senken.
10. Schauen wir einmal:
 - Ein Modell mit etwas Regularisierung arbeitet in der Regel besser als ein Modell ohne Regularisierung. Daher sollten Sie grundsätzlich die Ridge-Regression der einfachen linearen Regression vorziehen.²
 - Die Lasso-Regression verwendet einen ℓ_1 -Strafterm, wodurch Gewichte auf exakt null heruntergedrückt werden. Dadurch erhalten Sie spärliche Modelle, bei denen alle Gewichte außer den wichtigsten null sind. Auf diese Weise können Sie eine automatische Merkmalsauswahl durchführen, wenn Sie ohnehin schon den Verdacht hegen, dass nur einige Merkmale wichtig sind. Wenn Sie sich nicht sicher sind, sollten Sie der Ridge-Regression den Vorzug geben.
 - Elastic Net ist grundsätzlich gegenüber der Lasso-Regression vorzuziehen, da sich Lasso in einigen Fällen sprunghaft verhält (wenn mehrere Merkmale stark miteinander korrelieren oder es mehr Merkmale als Trainingsdatenpunkte gibt). Allerdings gilt es einen zusätzlichen Hyperparameter einzustellen. Wenn Sie Lasso ohne das sprunghafte Verhalten verwenden möchten, können Sie einfach Elastic Net mit einer `l1_ratio` um 1 verwenden.
11. Wenn Sie Bilder als außen/innen und Tag/Nacht klassifizieren möchten, schließen sich die Kategorien nicht gegenseitig aus (d.h., alle vier Kombinationen sind möglich). Sie sollten daher zwei Klassifikatoren mit logistischer Regression trainieren.
12. Die Lösungen finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 5: Support Vector Machines

1. Der Grundgedanke bei Support Vector Machines ist, die breitestmögliche »Straße« zwischen den Kategorien zu fitten. Anders ausgedrückt, soll zwi-

2 Außerdem erfordert die Normalengleichung die Berechnung einer inversen Matrix, aber diese Matrix ist nicht immer invertierbar. Dagegen ist die Matrix für die Ridge-Regression immer invertierbar.

schen der Entscheidungsgrenze zwischen den beiden Kategorien und den Trainingsdatenpunkten eine möglichst große Lücke sein. Bei der Soft-Margin-Klassifikation sucht das SVM nach einem Kompromiss zwischen einer perfekten Trennung zwischen den Kategorien und der breitestmöglichen Straße (d.h., einige Datenpunkte dürfen auf der Straße liegen). Eine weiteres wichtiges Konzept ist die Verwendung von Kernels beim Trainieren nichtlinearer Datensätze.

2. Nach dem Trainieren eines SVM ist jeder Datenpunkt an der »Straße« (siehe vorige Antwort) ein *Stützvektor*, einschließlich des Straßenrands. Die Entscheidungsgrenze ist vollständig durch die Stützvektoren festgelegt. Jeder Datenpunkt, der *kein* Stützvektor ist (d.h. abseits der Straße liegt) hat darauf keinen Einfluss; Sie könnten diese entfernen, weitere Datenpunkte hinzufügen oder sie verschieben. Solange sie weg von der Straße bleiben, beeinflussen sie die Entscheidungsgrenze nicht. Zum Berechnen einer Vorhersage sind nur die Stützvektoren nötig, nicht der gesamte Datensatz.
3. SVMs versuchen, die breitestmögliche »Straße« zwischen den Kategorien einzufügen (siehe erste Antwort). Wenn also die Trainingsdaten nicht skaliert sind, neigt das SVM dazu, kleine Merkmale zu ignorieren (siehe Abbildung 5-2).
4. Ein SVM-Klassifikator kann den Abstand zwischen einem Testdatenpunkt und der Entscheidungsgrenze ausgeben, und Sie können diese als Konfidenzmaß interpretieren. Allerdings lässt sich dieser Score nicht direkt in eine Schätzung der Wahrscheinlichkeit einer Kategorie umrechnen. Wenn Sie beim Erstellen eines SVM in Scikit-Learn `probability=True` einstellen, werden die Wahrscheinlichkeiten mithilfe einer logistischen Regression auf den Scores der SVM kalibriert (das zusätzlich mit fünffacher Kreuzvalidierung auf den Trainingsdaten trainiert wird). Damit erhalten Sie auch für ein SVM die Methoden `predict_proba()` und `predict_log_proba()`.
5. Diese Frage betrifft nur lineare SVMs, da Kernel-SVMs nur die duale Form verwenden können. Die Komplexität der Berechnung der primären Form ist proportional zur Anzahl der Trainingsdatenpunkte m , während sie bei der dualen Form zu einer Zahl zwischen m^2 und m^3 proportional ist. Wenn es also Millionen Datenpunkte gibt, sollten Sie auf jeden Fall die primäre Form verwenden, weil die duale Form viel zu langsam wird.
6. Wenn ein mit einem RBF-Kernel trainierter SVM-Klassifikator die Trainingsdaten underfittet, gibt es möglicherweise zu viel Regularisierung. Um diese zu senken, müssen Sie `gamma` oder `C` erhöhen (oder beide).
7. Nennen wir die QP-Parameter für das Hard-Margin-Problem \mathbf{H}' , \mathbf{f}' , \mathbf{A}' und \mathbf{b}' (siehe Abschnitt »*Quadratische Programme*« auf Seite 159). Die QP-Parameter beim Soft-Margin-Problem enthalten m zusätzliche Parameter ($n_p = n + 1 + m$) und m zusätzliche Restriktionen ($n_c = 2m$). Sie lassen sich folgendermaßen definieren:

- \mathbf{H} entspricht \mathbf{H}' zuzüglich m Spalten mit Nullen auf der rechten Seite und m Zeilen mit Nullen auf der unteren Seite: $\mathbf{H} = \begin{pmatrix} \mathbf{H}' & 0 & \dots \\ 0 & 0 & \ddots \\ \vdots & & \ddots \end{pmatrix}$

- \mathbf{f} ist gleich \mathbf{f}' mit m zusätzlichen Elementen, die alle gleich dem Wert des Hyperparameters C sind.
- \mathbf{b} ist gleich \mathbf{b}' mit m zusätzlichen Elementen, die alle 0 sind.
- \mathbf{A} ist gleich \mathbf{A}' , mit einer zusätzlichen $m \times m$ -Identitätsmatrix \mathbf{I}_m auf der rechten Seite, $-\mathbf{I}_m$ direkt darunter, der Rest wird mit Nullen aufgefüllt:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}$$

Die Lösungen zu den Übungsaufgaben 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 6: Entscheidungsbäume

1. Die Tiefe eines ausbalancierten Binärbaums mit m Blättern ist $\log_2(m)^3$, aufgerundet. Ein binärer Entscheidungsbaum (der wie alle Bäume in Scikit-Learn nur binäre Entscheidungen trifft) ist nach dem Trainieren mehr oder weniger ausbalanciert und enthält ein Blatt pro Trainingsdatenpunkt, falls er ohne Einschränkungen trainiert wird. Wenn also der Trainingsdatensatz eine Million Datenpunkte enthält, hat der Binärbaum eine Tiefe von $\log_2(10^6) \approx 20$ (in der Praxis ein wenig mehr, da der Baum nicht perfekt ausbalanciert sein wird).
2. Die Gini-Unreinheit eines Knotens ist im Allgemeinen geringer als die des Elternknotens. Dies liegt daran, dass die Kostenfunktion des CART-Trainsalgorithmus jeden Knoten so aufteilt, dass die gewichtete Summe der Gini-Unreinheiten der Kinder minimal wird. Es ist aber möglich, dass ein Knoten eine höher Gini-Unreinheit als der Elternknoten erhält, solange dies durch eine geringe Gini-Unreinheit seines Geschwisterknotens ausgeglichen wird. Betrachten Sie beispielsweise einen Knoten mit vier Datenpunkten aus Kategorie A und 1 aus Kategorie B. Dessen Gini-Unreinheit beträgt $1 - \frac{1^2}{5} - \frac{4^2}{5} = 0.32$. Nehmen Sie an, dass der Datensatz eindimensional ist und die Datenpunkte in folgender Reihenfolge liegen: A, B, A, A, A. Es lässt sich nachweisen, dass der Algorithmus diesen Knoten nach dem zweiten Datenpunkt aufteilt und somit ein Kind mit den Datenpunkten A, B und eines mit den Datenpunkten A, A, A entsteht. Die Gini-Unreinheit des ersten Kinds beträgt $1 - \frac{1^2}{2} - \frac{1^2}{2} = 0.5$, was höher als die des Elternknotens ist. Dies wird dadurch kompensiert, dass

³ \log_2 ist der binäre Logarithmus $\log_2(m) = \log(m) / \log(2)$.

der zweite Knoten rein ist, die gesamte gewichtete Gini-Unreinheit beträgt damit $\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0 = 0.2$, was geringer als die Gini-Unreinheit des Elternknotens ist.

3. Wenn ein Entscheidungsbaum die Trainingsdaten overfittet, sollten Sie eventuell `max_depth` verringern, da diese Einschränkung das Modell regularisiert.
4. Entscheidungsbäume scheren sich nicht darum, ob die Trainingsdaten skaliert oder zentriert sind, dies ist eine ihrer angenehmen Eigenschaften. Wenn also ein Entscheidungsbaum die Trainingsdaten underfittet, ist Skalieren der Eingabemerkmale reine Zeitverschwendungen.
5. Die Komplexität der Berechnung beim Trainieren eines Entscheidungsbaums beträgt $O(n \times m \log(m))$. Wenn Sie also die Größe des Trainingsdatensatzes mit 10 multiplizieren, verlängert sich die Zeit zum Trainieren um den Faktor $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. Bei $m = 10^6$ beträgt $K \approx 11.7$, Sie können also mit einer Trainingsdauer von etwa 11.7 Stunden rechnen.
6. Vorsortieren der Trainingsdaten verkürzt das Training nur, wenn der Datensatz kleiner als einige Tausend Datenpunkte ist. Wenn er 100000 Datenpunkte enthält, wird das Trainieren mit der Einstellung `presort=True` deutlich langsamer.

Die Lösungen zu den Übungsaufgaben 7 und 8 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 7: Ensemble Learning und Random Forests

1. Wenn Sie fünf unterschiedliche Modelle trainiert haben und alle eine Relevanz von 95% erzielen, können Sie diese zu einem Ensemble kombinieren, was häufig zu noch besseren Ergebnissen führt. Wenn die Modelle sich sehr stark unterscheiden, funktioniert es noch besser (z.B. ein SVM-Klassifikator, ein Entscheidungsbaum, ein Klassifikator mit logistischer Regression und so weiter). Durch Trainieren auf unterschiedlichen Trainingsdaten lässt sich eine weitere Verbesserung erzielen (darum geht es beim Bagging und Pasting von Ensembles), aber es funktioniert auch ohne, solange die Modelle sehr unterschiedlich sind.
2. Ein Klassifikator mit Hard Voting zählt einfach nur die Stimmen jedes Klassifikators im Ensemble und wählt die Kategorie aus, die die meisten Stimmen erhält. Ein Klassifikator mit Soft Voting berechnet den Durchschnitt der geschätzten Wahrscheinlichkeiten für jede Kategorie und wählt die Kategorie mit der höchsten Wahrscheinlichkeit aus. Damit erhalten Stimmen mit hoher Konfidenz mehr Gewicht, was oft besser funktioniert. Dies gelingt aber nur,

wenn jeder Klassifikator zum Abschätzen von Wahrscheinlichkeiten in der Lage ist (z.B. bei SVM-Klassifikatoren in Scikit-Learn müssen Sie `probability=True` setzen).

3. Es ist möglich, das Trainieren eines Ensembles mit Bagging durch Verteilen auf mehrere Server zu beschleunigen, da jeder Prädiktor im Ensemble unabhängig von den anderen ist. Aus dem gleichen Grund gilt dies auch für Ensembles mit Pasting und Random Forests. Dagegen baut jeder Prädiktor in einem Boosting-Ensemble auf dem vorigen Prädiktor auf, daher ist das Trainieren notwendigerweise sequenziell, und ein Verteilen auf mehrere Server nutzt nichts. Bei Stacking-Ensembles sind alle Prädiktoren einer Schicht unabhängig voneinander und lassen sich daher parallel auf mehreren Servern trainieren. Allerdings lassen sich die Prädiktoren einer Schicht erst trainieren, nachdem die vorige Schicht vollständig trainiert wurde.
4. Bei der Out-of-Bag-Evaluation wird jeder Prädiktor in einem Bagging-Ensemble mit Datenpunkten ausgewertet, auf denen er nicht trainiert wurde (diese wurden zurückgehalten). Damit ist eine recht unbeeinflusste Evaluation des Ensembles ohne einen zusätzlichen Validierungsdatensatz möglich. Dadurch stehen Ihnen also mehr Trainingsdaten zur Verfügung, und Ihr Ensemble verbessert sich leicht.
5. Beim Erzeugen eines Baums in einem Random Forest wird beim Aufteilen eines Knotens nur eine zufällig ausgewählte Untermenge der Merkmale berücksichtigt. Dies gilt auch bei Extra-Trees, diese gehen aber noch einen Schritt weiter: Anstatt wie gewöhnliche Entscheidungsbäume nach dem bestmöglichen Schwellenwert zu suchen, verwenden sie für jedes Merkmal zufällige Schwellenwerte. Dieses zusätzliche Zufallselement wirkt wie eine Art Regularisierung: Wenn ein Random Forest die Trainingsdaten overfittet, könnten Extra-Trees besser abschneiden. Da außerdem Extra-Trees nicht nach dem bestmöglichen Schwellenwert suchen, lassen sie sich viel schneller trainieren als Random Forests. Allerdings sind sie beim Treffen von Vorhersagen weder schneller noch langsamer als Random Forests.
6. Wenn Ihr AdaBoost-Ensemble die Trainingsdaten underfittet, können Sie die Anzahl der Estimatoren steigern und die Regularisierung des zugrunde liegenden Estimators über dessen Hyperparameter verringern. Sie können auch versuchen, die Lernrate ein wenig zu erhöhen.
7. Wenn Ihr Gradient Boosting-Ensemble die Trainingsdaten overfittet, sollten Sie die Lernrate senken. Sie können auch Early Stopping verwenden, um die richtige Anzahl Prädiktoren zu finden (vermutlich haben Sie zu viele davon).

Die Lösungen zu den Übungsaufgaben 8 und 9 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 8: Dimensionsreduktion

1. Gründe zum Einsatz und Nachteile:

- Die Hauptgründe zum Einsatz von Dimensionsreduktion sind:
 - Die nachfolgenden Trainingsalgorithmen zu beschleunigen (in manchen Fällen sogar Rauschen und redundante Merkmale zu entfernen, wodurch das Trainingsverfahren eine höhere Leistung erbringt).
 - Die Daten zu visualisieren und Einblick in ihre wichtigsten Eigenschaften zu erhalten.
 - Platz zu sparen (Kompression).
 - Die wichtigsten Nachteile sind:
 - Es geht Information verloren, wodurch die Leistung nachfolgender Trainingsverfahren möglicherweise sinkt.
 - Sie kann rechenintensiv sein.
 - Sie erhöht die Komplexität Ihrer maschinellen Lernpipelines.
 - Transformierte Merkmale sind oft schwer zu interpretieren.
2. Der »Fluch der Dimensionalität« beschreibt den Umstand, dass in höherdimensionalen Räumen viele Schwierigkeiten auftreten, die es bei weniger Dimensionen nicht gibt. Beim Machine Learning ist eine häufige Erscheinungsform, dass zufällig ausgewählte höherdimensionale Vektoren grundsätzlich dünn besetzt sind, was das Risiko für Overfitting erhöht und das Erkennen von Mustern in den Daten erschwert, wenn nicht sehr viele Trainingsdaten vorhanden sind.
3. Sobald die Dimensionalität eines Datensatzes mit einem der besprochenen Algorithmen verringert wurde, ist es so gut wie immer unmöglich, den Vorgang vollständig umzukehren, weil im Laufe der Dimensionsreduktion Information verloren geht. Während es bei einigen Algorithmen (wie der PCA) eine einfache Prozedur zur reversen Transformation gibt, mit der sich der Datensatz recht nah am Original rekonstruieren lässt, ist dies bei anderen Algorithmen (wie T-SNE) nicht möglich.
4. Mit der Hauptkomponentenzerlegung (PCA) lässt sich die Anzahl der Dimensionen der meisten Datensätze erheblich reduzieren, selbst wenn sie stark nichtlinear sind, weil sie zumindest die bedeutungslosen Dimensionen verwerten kann. Wenn es aber keine bedeutungslosen Dimensionen gibt – wie beispielsweise beim Swiss-Roll-Datensatz –, geht bei der Dimensionsreduktion mittels PCA zu viel Information verloren. Sie möchten die Swiss Roll aufrollen, nicht platt quetschen.
5. Dies ist eine Fangfrage: Es kommt auf den Datensatz an. Betrachten wir zwei Extrembeispiele. Angenommen, der Datensatz besteht aus beinahe perfekt

aufgereihten Datenpunkten. In diesem Fall kann die Hauptkomponentenzerlegung den Datensatz auf nur eine Dimension reduzieren und trotzdem 95% der Varianz erhalten. Wenn dagegen der Datensatz aus perfekt zufällig angeordneten Punkten besteht, die überall in den 1000 Dimensionen verstreut sind, sind etwa 950 Dimensionen nötig, um 95% der Varianz zu erhalten. Die Antwort ist also, dass es auf den Datensatz ankommt und dass es eine beliebige Zahl zwischen 1 und 950 sein kann. Eine Möglichkeit, die intrinsische Dimensionalität des Datensatzes zu verdeutlichen, ist, die abgedeckte Varianz über der Anzahl der Dimensionen zu plotten.

6. Gewöhnliche Hauptkomponentenzerlegung ist Standard, sie funktioniert aber nur, wenn der Datensatz in den Speicher passt. Die inkrementelle PCA ist bei großen Datensätzen, die nicht in den Speicher passen, hilfreich, sie ist aber langsamer als die gewöhnliche PCA. Wenn der Datensatz in den Speicher passt, sollten Sie also die gewöhnliche Hauptkomponentenzerlegung vorziehen. Die inkrementelle PCA ist auch bei Online-Aufgaben nützlich, bei denen eine PCA bei neu eintreffenden Daten jedes Mal im Vorübergehen durchgeführt werden soll. Die randomisierte PCA ist nützlich, wenn Sie die Anzahl der Dimensionen erheblich reduzieren möchten und der Datensatz in den Speicher passt; in diesem Fall ist sie deutlich schneller als die gewöhnliche PCA. Schließlich ist die Kernel-PCA bei nichtlinearen Datensätzen hilfreich.
7. Intuitiv arbeitet ein Algorithmus zur Dimensionsreduktion dann gut, wenn er eine Menge Dimensionen aus dem Datensatz entfernt, ohne zu viel Information zu vergeuden. Dies lässt sich beispielsweise durch Anwenden der reversen Transformation und Bestimmen des Rekonstruktionsfehlers messen. Allerdings unterstützen nicht alle Verfahren zur Dimensionsreduktion die reverse Transformation. Wenn Sie die Dimensionsreduktion als Vorverarbeitungsschritt vor einem anderen Machine-Learning-Verfahren verwenden (z.B. einem Random Forest-Klassifikator), können Sie auch einfach die Leistung des nachgeschalteten Verfahrens bestimmen; wenn bei der Dimensionsreduktion nicht zu viel Information verloren geht, sollte der Algorithmus genauso gut abschneiden wie auf dem ursprünglichen Datensatz.
8. Es kann durchaus sinnvoll sein, zwei unterschiedliche Algorithmen zur Dimensionsreduktion in Reihe zu schalten. Ein verbreitetes Beispiel ist eine Hauptkomponentenzerlegung, um schnell eine große Anzahl unnützer Dimensionen loszuwerden und anschließend einen deutlich langsameren Algorithmus zur Dimensionsreduktion wie LLE zu verwenden. Dieser zweistufige Prozess führt vermutlich zur gleichen Qualität wie LLE für sich allein, benötigt aber nur einen Bruchteil der Zeit.

Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 9: Einsatzbereit mit TensorFlow

1. Die wichtigsten Vor- und Nachteile bei der Erzeugung eines Berechnungsgraphen gegenüber der direkten Berechnung sind:
 - Vorteile:
 - TensorFlow berechnet die Gradienten automatisch (über Autodiff im Reverse-Modus).
 - TensorFlow kümmert sich darum, Operationen in parallelen Threads auszuführen.
 - Es ist einfacher, das gleiche Modell auf unterschiedlichen Geräten auszuführen.
 - Die Introspektion ist einfacher – beispielsweise das Betrachten eines Modells in TensorBoard.
 - Nachteile:
 - Die Lernkurve ist steiler.
 - Das schrittweise Debuggen ist schwieriger.
2. Ja, die Anweisung `a_val = a.eval(session=sess)` ist wirklich zu `a_val = sess.run(a)` äquivalent.
3. Nein, die Anweisung `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` ist nicht zu `a_val, b_val = sess.run([a, b])` äquivalent. Die erste Anweisung führt den Graphen doppelt aus (einmal zur Berechnung von a, einmal zur Berechnung von b), die zweite hingegen nur einmal. Wenn eine dieser Operationen (oder abhängige Operationen) Nebeneffekte haben (z.B. eine Variable verändern, einer Queue ein Element hinzufügen oder eine Datei einlesen), werden sie unterschiedliche Auswirkungen haben. Wenn sie keine Nebeneffekte haben, ist das Ergebnis das gleiche, aber die zweite Anweisung ist schneller als die erste.
4. Nein, Sie können zwei Graphen nicht in der gleichen Session ausführen. Sie müssten dazu die Graphen zu einem einzelnen Graphen vereinigen.
5. Beim lokalen Ausführen von TensorFlow verwalten Sessions die Werte von Variablen. Wenn Sie also einen Graphen g mit einer Variable w erstellen, dann zwei Threads starten und in jedem Thread eine Session mit dem Graphen g öffnen, besitzt jede Session eine eigene Kopie der Variable w. Beim verteilten Ausführen von TensorFlow dagegen sind Variablen in vom Cluster verwalteten Containern abgelegt. Wenn also beide Sessions sich mit dem gleichen Cluster verbinden und den gleichen Container verwenden, teilen sie sich den Wert der Variablen w.
6. Eine Variable wird in dem Moment initialisiert, in dem Sie ihren Initializer aufrufen. Sie wird zerstört, sobald die Session endet. Beim verteilten Ausführen von TensorFlow existieren Variablen in Containern auf einem Cluster.

Dann zerstört das Schließen einer Session eine Variable nicht. Um eine Variable zu zerstören, müssen Sie ihren Container leeren.

7. Variablen und Platzhalter unterscheiden sich stark voneinander, Anfänger bringen sie aber leicht durcheinander:

- Eine Variable ist eine Operation, die einen Wert enthält. Wenn Sie die Variable ausführen, erhalten Sie ihren Wert. Bevor Sie eine Variable ausführen können, müssen Sie diese initialisieren. Sie können den Wert einer Variablen ändern (z.B. über eine Zuweisungsoperation). Sie hat einen Zustand: Die Variable behält ihren Wert über mehrere Durchläufe eines Graphen bei. Man verwendet Variablen normalerweise, um Modellparameter zu speichern, aber auch für andere Zwecke (z.B. die Schritte beim Trainieren mitzuzählen).
 - Platzhalter tun technisch gesehen nicht viel: Sie enthalten einfach nur Information über den Datentyp und die Abmessungen des repräsentierten Tensors, haben aber selbst keinen Wert. Wenn Sie versuchen, eine Operation zu evaluieren, die von einem Platzhalter abhängt, müssen Sie TensorFlow den Wert dieses Platzhalters mitteilen (über das Argument `feed_dict`), andernfalls erhalten Sie einen Ausnahmefehler. Platzhalter werden normalerweise dafür eingesetzt, während der Ausführungsphase Trainings- oder Testdaten an TensorFlow zu übergeben. Sie sind auch nützlich, um Werte an einen Zuweisungsknoten zu übergeben, um den Wert einer Variablen zu ändern (z.B. Gewichte des Modells).
8. Wenn Sie einen Graphen ausführen, der eine von einem Platzhalter abhängige Operation enthält, sie aber keinen Wert angeben, erhalten Sie einen Ausnahmefehler. Wenn die Operation von keinem Platzhalter abhängt, wird kein Fehler erzeugt.
9. Wenn Sie einen Graphen ausführen, können Sie den Ausgabewert einer beliebigen Operation angeben, nicht nur den Wert von Platzhaltern. In der Praxis ist dies aber eher selten (es ist z.B. dann nützlich, wenn Sie die Ausgabe von Frozen Layers sichern möchten, siehe Kapitel 11).
10. Sie können den Startwert einer Variablen beim Konstruieren des Graphen angeben. Wenn Sie später in der Ausführungsphase den Initializer dieser Variablen ausführen, wird sie initialisiert. Wenn Sie den Wert dieser Variablen während der Ausführungsphase ändern möchten, ist die einfachste Möglichkeit dazu, mit der Funktion `tf.assign()` einen Zuweisungsknoten anzulegen (während der Konstruktionsphase) und einen Platzhalter als Parameter zu übergeben. Während der Ausführungsphase können Sie die Zuweisungsoperation ausführen und den neuen Wert der Variablen über den Platzhalter angeben.

```
import tensorflow as tf
```

```
x = tf.Variable(tf.random_uniform(shape=(), minval=0.0, maxval=1.0))
```

```

x_new_val = tf.placeholder(shape=(), dtype=tf.float32)
x_assign = tf.assign(x, x_new_val)

with tf.Session():
    x.initializer.run() # Zufallszahl wird *jetzt* erzeugt
    print(x.eval()) # 0.646157 (eine Zufallszahl)
    x_assign.eval(feed_dict={x_new_val: 5.0})
    print(x.eval()) # 5.0

```

- Autodiff im Reverse-Modus (von TensorFlow implementiert) muss den Graphen nur zweimal abschreiben, um die Ableitungen der Kostenfunktion nach einer beliebigen Anzahl Variablen zu berechnen. Dagegen müsste Autodiff im Forward-Modus einmal pro Variable ausgeführt werden (also zehnmal, wenn wir Ableitungen nach zehn unterschiedlichen Variablen benötigen). Wie bei der symbolischen Differenzierung würde zum Berechnen der Gradienten ein zusätzlicher Graph aufgebaut werden, und der ursprüngliche Graph würde überhaupt nicht durchlaufen (außer zum Aufbau des Gradienten-Graphen). Ein hochoptimiertes System zur symbolischen Differenzierung könnte diesen neuen Graphen möglicherweise nur einmal abschreiben, um die Ableitungen nach allen Variablen zu berechnen, dieser neue Graph wäre verglichen mit dem ursprünglichen Graphen aber unsäglich komplex und ineffizient.
- Die Lösungen finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 10: Einführung in künstliche neuronale Netze

- Hier ist ein aus den ursprünglichen künstlichen Neuronen aufgebautes neuronales Netz, das $A \oplus B$ berechnet (wobei \oplus für das exklusive OR steht). Es macht sich den Umstand zunutze, dass $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. Es gibt weitere Lösungsmöglichkeiten – beispielsweise mithilfe von $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$ oder $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$ und so weiter.

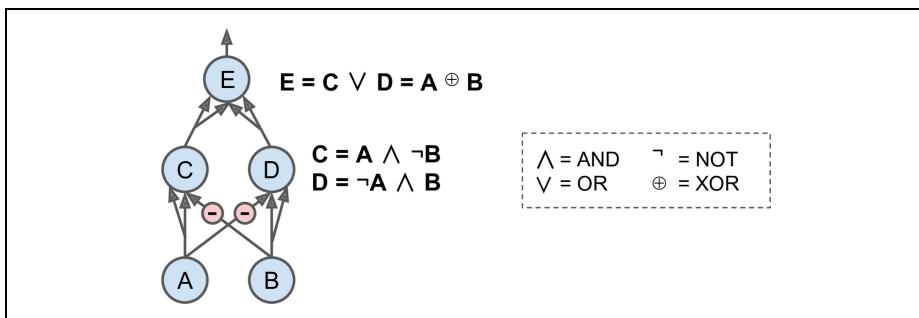


Abbildung A-1

- Ein klassisches Perzeptron konvergiert nur, wenn der Datensatz linear separierbar ist, und es ist nicht in der Lage, die Wahrscheinlichkeiten für Katego-

rien abzuschätzen. Ein Klassifikator mit logistischer Regression konvergiert dagegen sogar dann zu einer guten Lösung, wenn der Datensatz nichtlinear separierbar ist, und gibt Wahrscheinlichkeiten aus. Wenn Sie die Aktivierungsfunktion eines Perzeptron durch die logistische Aktivierungsfunktion ersetzen (oder bei mehreren Neuronen die Softmax-Aktivierungsfunktion) und es mit dem Gradientenverfahren trainieren (oder einem anderen Optimierungsalgorithmus, der eine Kostenfunktion wie die Kreuzentropie minimiert), ist das Netz zur Klassifikation mit logistischer Regression äquivalent.

3. Die logistische Aktivierungsfunktion war ein Hauptbestandteil beim Trainieren der ersten MLPs, da ihre Ableitung stets ungleich null ist, sodass das Gradientenverfahren stets bergab rollen kann. Wenn die Aktivierungsfunktion eine Stufenfunktion ist, kann sich das Gradientenverfahren nicht bewegen, da es überhaupt keine Steigung gibt.
4. Die Stufenfunktion, die logistische Funktion, der Tangens Hyperbolicus und die Rectified Linear Unit (siehe Abbildung 10-8). Weitere Beispiele wie ELU und Variationen der ReLU finden Sie in Kapitel 11.
5. Für das in der Frage beschriebene MLP nehmen wir an, es bestünde aus einer Eingabeschicht mit 10 Neuronen, gefolgt von einer verborgenen Schicht mit 50 künstlichen Neuronen und schließlich einer Ausgabeschicht mit 3 künstlichen Neuronen. Alle künstlichen Neuronen verwenden ReLU als Aktivierungsfunktion.
 - Die Abmessungen der Eingabematrix \mathbf{X} betragen $m \times 10$, wobei m für die Größe des Trainings-Batches steht.
 - Die Abmessungen des Gewichtsvektors der verborgenen Schicht \mathbf{W}_h betragen 10×50 , und die Länge des Bias-Vektors \mathbf{b}_h beträgt 50.
 - Die Abmessungen des Gewichtsvektors der Ausgabeschicht \mathbf{W}_o betragen 50×3 , und die Länge ihres Bias-Vektors \mathbf{b}_o beträgt 3.
 - Die Abmessungen der Ausgabematrix des Netzes \mathbf{Y} betragen $m \times 3$.
 - $\mathbf{Y} = \text{ReLU}(\text{ReLU}(\mathbf{X} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_o + \mathbf{b}_o)$. Die ReLU-Funktion setzt einfach jeden negativen Wert in der Matrix auf null. Außerdem wird der Bias-Vektor beim Addieren zu einer Matrix zu jeder einzelnen Zeile der Matrix addiert. Dies bezeichnet man als *Broadcasting*.
6. Um E-Mails als Spam oder Ham zu klassifizieren, benötigen Sie nur ein Neuron in der Ausgabeschicht eines neuronalen Netzes – das beispielsweise die Wahrscheinlichkeit für Spam anzeigt. Sie würden in der Ausgabeschicht dazu normalerweise die logistische Aktivierungsfunktion verwenden. Wenn Sie stattdessen die MNIST-Aufgabe bearbeiten, benötigen Sie zehn Neuronen in der Ausgabeschicht und müssten die logistische Funktion durch die Softmax-Funktion ersetzen, die mit mehreren Kategorien umgehen kann. Dabei erhalten Sie eine Wahrscheinlichkeit pro Kategorie. Wenn Ihr neuronales Netz wie

in Kapitel 2 Immobilienpreise vorhersagen soll, benötigen Sie ein Ausgabeneuron und überhaupt keine Aktivierungsfunktion in der Ausgabeschicht.⁴

7. Backpropagation ist ein Verfahren zum Trainieren künstlicher neuronaler Netze. Es berechnet zunächst die Gradienten der Kostenfunktion nach jedem Modellparameter (alle Gewichte und Biases) und führt dann einen Schritt im Gradientenverfahren mit diesen Gradienten aus. Dieser Schritt wird bei der Backpropagation normalerweise tausend- oder millionenfach ausgeführt, bis die Modellparameter zu Werten konvergieren, die die Kostenfunktion (hoffentlich) minimieren. Zur Berechnung der Gradienten verwendet das Backpropagation-Verfahren Autodiff im Reverse-Modus (auch wenn es bei der Einführung des Backpropagation-Verfahrens noch nicht so genannt wurde, es ist seitdem mehrmals neu erfunden worden). Autodiff im Reverse-Modus schreitet den Berechnungsgraphen vorwärts ab und berechnet den Wert jedes Knotens für den aktuellen Trainings-Batch und schreitet anschließend den Graphen rückwärts ab, wobei sämtliche Gradienten gleichzeitig berechnet werden (Details siehe Anhang D). Was ist also der Unterschied? Mit Backpropagation ist der gesamte Trainingsprozess eines künstlichen neuronalen Netzes über mehrere Backpropagation-Schritte gemeint, wobei in jedem einzelnen Gradienten berechnet und ein Schritt im Gradientenverfahren durchgeführt wird. Im Gegensatz dazu ist Autodiff im Reverse-Modus einfach eine Technik zum effizienten Berechnen von Gradienten, die zufällig vom Backpropagation-Verfahren verwendet wird.
8. Hier folgt eine Liste aller Hyperparameter, die Sie in einem einfachen MLP verändern können: Die Anzahl verborgener Schichten, die Anzahl Neuronen pro verborgene Schicht und die in jeder Schicht verwendete Aktivierungsfunktion.⁵ Im Allgemeinen ist die Aktivierungsfunktion ReLU (oder eine ihrer Varianten, siehe Kapitel 11) eine gute Standardeinstellung für die verborgenen Schichten. In der Ausgabeschicht sollten Sie bei binärer Klassifikation die logistische Aktivierungsfunktion verwenden, bei mehreren Kategorien die Softmax-Aktivierungsfunktion und bei Regression überhaupt keine Aktivierungsfunktion.
Wenn das MLP die Trainingsdaten overfittet, können Sie die Anzahl verborgener Schichten und die Anzahl der Neuronen darin reduzieren.
9. Weitere Lösungen finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

⁴ Wenn die vorherzusagenden Werte sich um viele Größenordnungen unterscheiden, sollten Sie anstatt der Zielgröße den Logarithmus der Zielgröße vorhersagen. Durch Berechnen der Exponentialfunktion aus der Ausgabe des Netzes erhalten Sie den geschätzten Wert (da $\exp(\log v) = v$).

⁵ In Kapitel 11 besprechen wir viele Techniken, die zusätzliche Hyperparameter einbringen: die Art der Initialisierung von Gewichten, Hyperparameter der Aktivierungsfunktion (z.B. die Stärke des Lecks beim Leaky ReLU), der Schwellenwert beim Gradient Clipping, die Art des Optimierungsverfahrens und dessen Hyperparameter (z.B. der Hyperparameter `momentum` beim Verwenden eines MomentumOptimizer), die Art der Regularisierung jeder Schicht sowie die Hyperparameter der Regularisierung (z.B. die Drop-out-Rate beim Drop-out) und so weiter.

Kapitel 11: Trainieren von Deep Learning-Netzen

1. Nein, alle Gewichte sollten unabhängig voneinander erzeugt werden; es sollten nicht alle den gleichen Startwert haben. Ein wichtiges Ziel beim zufälligen Erzeugen von Gewichten ist, Symmetrien aufzubrechen: Wenn alle Gewichte den gleichen Startwert haben, selbst wenn dieser ungleich null ist, besteht eine Symmetrie (d.h., sämtliche Neuronen einer Schicht sind äquivalent). Das Backpropagation-Verfahren ist nicht in der Lage, diese aufzubrechen. Anders ausgedrückt verhalten sich alle Neuronen mit gleichen Gewichten innerhalb einer Schicht wie ein einziges Neuron. Sie sind nur viel langsamer. Es ist praktisch unmöglich, dass eine derartige Anordnung zu einer guten Lösung konvergiert.
2. Es ist völlig in Ordnung, die Bias-Terme mit null zu initialisieren. Manchmal werden sie genau wie die Gewichte initialisiert, auch das ist in Ordnung; es macht keinen großen Unterschied.
3. Einige Vorteile der ELU-Funktion gegenüber der ReLU-Funktion sind:
 - Sie kann negative Werte annehmen, sodass die durchschnittliche Ausgabe eines Neurons in einer beliebigen Schicht normalerweise näher an 0 liegt als bei der ReLU-Aktivierungsfunktion (die niemals negative Werte ausgibt). Dies hilft beim Bekämpfen des Problems schwindender Gradienten.
 - Ihre Ableitung ist stets ungleich null, was das Problem der sterbenden Einheiten löst, das bei ReLU-Einheiten auftritt.
 - Sie ist überall glatt, während die Steigung der ReLU-Funktion bei $z = 0$ abrupt von 0 auf 1 springt. Solche abrupten Änderungen verlangsamen das Gradientenverfahren, weil es um $z = 0$ umherspringt.
4. Die ELU-Aktivierungsfunktion ist ein guter Ausgangswert. Wenn Ihr neuronales Netz so schnell wie möglich sein muss, können Sie stattdessen eine der Leaky-ReLU-Varianten verwenden (z.B. ein einfaches Leaky ReLU mit dem voreingestellten Hyperparameter). Die Einfachheit der ReLU-Aktivierungsfunktion macht diese für viele zur ersten Wahl, obwohl ELU und Leaky ReLU meist eine höhere Leistung erzielen. Allerdings kann es sich in manchen Fällen auszahlen, dass die ReLU-Aktivierungsfunktion exakt null ausgeben kann (z.B. siehe Kapitel 15). Der Tangens Hyperbolicus (\tanh) ist in der Ausgabeschicht nützlich, wenn Sie eine Zahl zwischen -1 und 1 ausgeben möchten, er wird aber heutzutage in verborgenen Schichten kaum noch verwendet. Die logistische Aktivierungsfunktion ist ebenfalls in der Ausgabeschicht nützlich, wenn Sie eine Wahrscheinlichkeit schätzen möchten (z.B. bei der binären Klassifikation). Auch sie findet selten in verborgenen Schichten Anwendung (es gibt Ausnahmen – beispielsweise in der codierenden Schicht eines Variational Autoencoders, siehe Kapitel 15). Schließlich wird die Softmax-Aktivierungsfunktion in der Ausgabeschicht von Klassifikatoren verwendet, um Wahrscheinlichkeiten sich gegenseitig ausschließender Kategorien auszugeben. Auch sie wird selten (falls überhaupt) in verborgenen Schichten eingesetzt.

5. Wenn Sie bei einem MomentumOptimizer den Hyperparameter `momentum` zu nah an 1 setzen (z.B. 0.99999), wird der Algorithmus voraussichtlich stark Geschwindigkeit aufnehmen, sich hoffentlich in etwa auf das globale Minimum zu bewegen und dann wegen seines Moments daran vorbeizischen. Dann bremst er ab, kehrt zurück, beschleunigt wieder, fliegt wieder zu weit und so weiter. Bis zur Konvergenz kann er auf diese Weise viele Male oszillieren, das Konvergieren dauert also insgesamt viel länger als mit einem kleineren Wert für `momentum`.
6. Ein dünn besetztes Modell (bei dem die meisten Gewichte null betragen) lässt sich erzeugen, indem Sie das Modell normal trainieren und dann winzige Gewichte auf null setzen. Zusätzlich können Sie beim Trainieren die ℓ_1 -Regularisierung anwenden, was den Optimierer in Richtung dünn besetzter Parameter drückt. Eine dritte Möglichkeit ist, die ℓ_1 -Regularisierung mit *Dual Averaging* durch die TensorFlow-Klasse `FTRL`Optimizer zu kombinieren.
7. Ja, das Trainieren wird durch Drop-out langsamer, meist etwa um den Faktor zwei. Es hat allerdings keinen Einfluss auf die Inferenz, da Drop-out nur beim Trainieren angeschaltet ist.

Die Lösungen zu den Übungsaufgaben 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 12: TensorFlow über mehrere Geräte und Server verteilen

1. Wenn ein TensorFlow-Prozess startet, beansprucht dieser sämtlichen verfügbaren Speicher aller sichtbaren GPUs. Wenn Sie also beim Starten eines TensorFlow-Programms einen `CUDA_ERROR_OUT_OF_MEMORY` erhalten, laufen vermutlich bereits andere Prozesse, die sich auf mindestens einer sichtbaren GPU den gesamten Speicher geschnappt haben (höchstwahrscheinlich ist es ein zweiter TensorFlow-Prozess). Um dieses Problem zu beheben, können Sie natürlich die anderen Prozesse anhalten und es noch einmal versuchen. Falls es notwendig ist, dass alle Prozesse simultan laufen, können Sie entweder jeden Prozess mit der Umgebungsvariable `CUDA_VISIBLE_DEVICES` auf einer anderen Recheneinheit platzieren oder TensorFlow so konfigurieren, dass es nur einen Teil des GPU-Speichers beansprucht, nicht den gesamten. Dazu erzeugen Sie einen `ConfigProto`, setzen dessen `gpu_options.per_process_gpu_memory_fraction` auf den Anteil des zu belegenden Speichers (z.B. 0.4) und verwenden diesen `ConfigProto` beim Öffnen einer Session. Als letzte Möglichkeit können Sie TensorFlow anweisen, Speicher erst dann zu belegen, wenn er benötigt wird, indem Sie die Option `gpu_options.allow_growth` auf `True` setzen. Allerdings ist diese letzte Option nicht empfehlenswert, da TensorFlow einmal belegten Speicher nicht wieder freigibt und es somit schwieriger wird, ein vorhersagbares Ver-

halten zu garantieren (es kann zu Race Conditions kommen, je nachdem welche Prozesse zuerst gestartet werden, wie viel Speicher diese beim Trainieren benötigen und so weiter).

2. Indem Sie eine Operation auf eine Recheneinheit pinnen, weisen Sie TensorFlow an, wo diese Operation bevorzugt platziert werden soll. Allerdings könnten einige Umstände TensorFlow an der Erfüllung Ihres Wunschs hindern. Zum Beispiel könnte es bei einer Operation keine Implementierung (einen *Kernel*) für diese Art Recheneinheit geben. In diesem Fall erzeugt TensorFlow standardmäßig einen Ausnahmefehler, Sie können aber optional auch auf die CPU ausweichen (dies nennt man *Soft Placement*). Ein anderes Beispiel sind Operationen, die eine Variable verändern; die Operation und die Variable müssen auf der gleichen Recheneinheit liegen. Der Unterschied zwischen Pinnen und Platzieren ist also, dass Sie beim Pinnen TensorFlow fragen (»Bitte platziere diese Operation auf GPU #1«), während Platzieren das ist, was TensorFlow tatsächlich tut (»Tut mir leid, ich weiche auf die CPU aus«).
3. Ja, alle Operationen werden auf der ersten GPU platziert, wenn Sie eine TensorFlow-Installation mit GPU-Unterstützung haben, die voreingestellte Platzierung verwendet, und wenn es für sämtliche Operationen einen GPU-Kernel gibt (d.h. eine Implementierung für GPUs). Wenn es allerdings für eine oder mehrere Operationen keinen GPU-Kernel gibt, erzeugt TensorFlow standardmäßig einen Ausnahmefehler. Wenn Sie TensorFlow so konfigurieren, dass es stattdessen die CPU verwendet (Soft Placement), werden sämtliche Operationen auf der ersten GPU platziert, außer denen ohne GPU-Kernel und sämtlichen Operationen, die mit diesen gemeinsam platziert werden müssen (siehe Antwort auf die vorige Frage).
4. Ja, wenn Sie eine Variable auf /gpu:0 pinnen, kann sie von auf /gpu:1 platzierten Operationen verwendet werden. TensorFlow kümmert sich automatisch darum, den Wert der Variablen von einer Recheneinheit auf die andere zu übertragen. Das Gleiche gilt für Recheneinheiten auf unterschiedlichen Servern (solange diese zum gleichen Cluster gehören).
5. Ja, zwei auf der gleichen Recheneinheit platzierte Operationen können gleichzeitig ausgeführt werden: TensorFlow kümmert sich automatisch darum, Operationen parallel auszuführen (in unterschiedlichen CPU-Cores oder GPU-Threads), solange keine der Operationen von der Ausgabe einer anderen abhängt. Außerdem können Sie mehrere Sessions in parallelen Threads (oder Prozessen) starten und in jedem Thread Operationen auswerten. Da die Sessions voneinander unabhängig sind, kann TensorFlow jede Operation aus einer Session mit einer beliebigen Operation aus einer anderen Session parallel auswerten.
6. Control Dependencies lassen sich zum Aufschieben einer Operation X auf einen Zeitpunkt nach Ausführen einiger anderer Operationen verwenden, selbst wenn diese Operationen zur Berechnung von X nicht erforderlich sind.

Dies ist vor allem dann nützlich, wenn X eine Menge Speicher belegt und Sie es nur später im Berechnungsgraphen benötigen oder wenn X eine Menge I/O-Kapazitäten beansprucht (beispielsweise den Wert einer auf einem anderen Gerät oder Server gespeicherten Variable benötigt) und Sie diese Operation nicht zeitgleich mit anderen I/O-lastigen Operationen ausführen möchten, um ein Erschöpfen der Bandbreite zu vermeiden.

7. Sie haben Glück! Beim verteilten Einsatz von TensorFlow sind die Variablen in vom Cluster verwalteten Containern gespeichert. Selbst wenn Sie die Session und Ihr Client-Programm beenden, sind die Modellparameter nach wie vor auf dem Cluster vorhanden. Sie müssen lediglich eine neue Session öffnen und das Modell speichern (stellen Sie sicher, dass Sie keine Initializer von Variablen aufrufen oder ein vorher erzeugtes Modell wiederherstellen, da dies Ihr wertvolles neues Modell zerstören würde!).

Die Lösungen zu den Übungsaufgaben 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 13: Convolutional Neural Networks

1. Folgendes sind die Hauptvorteile eines CNN gegenüber einem vollständig verbundenen DNN zur Bildklassifikation:

- Weil aufeinanderfolgende Schichten nur teilweise verbunden sind und Gewichte in großem Ausmaß wiederverwendet werden, enthält ein CNN viel weniger Parameter als ein vollständig verbundenes DNN, wodurch es schneller trainierbar ist, weniger zu Overfitting neigt und viel weniger Trainingsdaten erfordert.
- Wenn ein CNN einen Kernel erlernt hat, der ein bestimmtes Muster erkennt, kann es dieses Muster überall im Bild erkennen. Im Gegensatz dazu kann ein DNN, wenn es ein Muster an einem Ort erlernt hat, dieses nur am gleichen Ort wiedererkennen. Da Bilder meist wiederkehrende Muster enthalten, können CNNs bei der Verarbeitung von Bilddaten sehr viel besser und mit weniger Trainingsbeispielen als DNNs verallgemeinern, beispielsweise bei der Klassifikation.
- Schließlich verfügt ein DNN über keinerlei Wissen darüber, wie Pixel organisiert sind; es weiß nicht, dass benachbarte Pixel im Bild nah beieinanderliegen. Die Architektur eines CNN hat diese Art Vorwissen verinnerlicht. Die niedrigeren Schichten erkennen für gewöhnlich Muster in kleinen Bildausschnitten, die oberen Schichten kombinieren dagegen die Muster der unteren Schichten zu größeren Mustern. Dies funktioniert mit den meisten natürlichen Bildern gut, womit sich CNNs einen guten Vorsprung vor DNNs erarbeiten.

2. Rechnen wir aus, wie viele Parameter das CNN hat. Der erste Convolutional Layer enthält 3×3 Kernels, die Eingabe weist drei Kanäle auf (Rot, Grün und Blau), jede Feature Map enthält $3 \times 3 \times 3$ Gewichte sowie einen Bias-Term. Damit ergeben sich 28 Parameter pro Feature Map. Da dieser erste Convolutional Layer 100 Feature Maps enthält, gibt es insgesamt 2800 Parameter. Der zweite Convolutional Layer enthält 3×3 Kernels, und die Eingabe sind die 100 Feature Maps aus der vorigen Schicht. Damit enthält jede Feature Map $3 \times 3 \times 100 = 900$ Gewichte sowie einen Bias-Term. Da es 200 Feature Maps gibt, enthält diese Schicht $900 \times 200 = 180200$ Parameter. Der dritte und letzte Convolutional Layer enthält ebenfalls 3×3 Kernels, und seine Eingabe sind die 200 Feature Maps aus den vorigen Schichten. Also enthält jede Feature Map $3 \times 3 \times 200 = 1800$ Gewichte sowie einen Bias-Term. Da es 400 Feature Maps gibt, enthält diese Schicht insgesamt $1800 \times 400 = 720400$ Parameter. Rechnet man alles zusammen, hat dieses CNN $2800 + 180200 + 720400 = 903400$ Parameter.

Rechen wir nun aus, wie viel RAM dieses neuronale Netz (mindestens) beim Treffen einer Vorhersage für einen Datenpunkt benötigt. Dazu rechnen wir zunächst die Größen der Feature Maps in jeder Schicht aus. Da wir als Stride 2 und SAME-Padding verwenden, wird die horizontale und vertikale Größe jeder Feature Map in jeder Schicht durch 2 geteilt (aufgerundet, falls nötig). Da also die Eingabekanäle aus 200×300 Pixeln bestehen, haben die Feature Maps in der ersten Schicht die Größe 100×150 , die Feature Maps in der zweiten Schicht die Größe 50×75 und die Feature Maps in der dritten Schicht die Größe 25×38 . Da 32 Bits 4 Bytes entsprechen und der erste Convolutional Layer aus 100 Feature Maps besteht, nimmt die erste Schicht $4 \times 100 \times 150 \times 100 = 6$ Millionen Bytes ein (etwa 5.7 MB, da 1 MB = 1,024 KB und 1 KB = 1024 Bytes). Die zweite Schicht benötigt $4 \times 50 \times 75 \times 200 = 3$ Millionen Bytes (etwa 2.9 MB). Schließlich benötigt die dritte Schicht $4 \times 25 \times 38 \times 400 = 1520000$ Bytes (etwa 1.4 MB). Sobald eine Schicht berechnet wurde, kann der von der vorigen Schicht verwendete Speicher freigegeben werden, bei guter Optimierung sind also im RAM nur $6 + 3 = 9$ Millionen Bytes (etwa 8.6 MB) nötig (wenn die zweite Schicht soeben fertig berechnet, der von der ersten Schicht belegte Speicher aber noch nicht freigegeben wurde). Einen Moment! Wir müssen auch den von den Parametern des CNN belegten Speicher berücksichtigen. Wir haben oben 903400 Parameter ausgegerechnet, von denen jeder 4 Bytes belegt. Dadurch kommen noch einmal 3613600 Bytes hinzu (etwa 3.4 MB). Das gesamte nötige RAM beträgt (mindestens) 12613600 Bytes (etwa 12.0 MB).

Als Letztes berechnen wir die Mindestgröße des benötigten RAM beim Trainieren des CNN mit einem Mini-Batch aus 50 Bildern. Beim Trainieren verwendet TensorFlow das Backpropagation-Verfahren, bei dem sämtliche im Vorrücksdurchlauf berechneten Werte aufgehoben werden müssen, bis der Rückwärtsdurchlauf beginnt. Daher müssen wir das gesamte von allen Schichten benötigte RAM für einen Datenpunkt berechnen und diesen Wert

mit 50 multiplizieren! An dieser Stelle rechnen wir in Megabytes anstatt in Bytes weiter. Wir hatten oben berechnet, dass die drei Schichten jeweils 5.7, 2.9 und 1.4 MB pro Datenpunkt benötigen. Das sind insgesamt 10.0 MB pro Datenpunkt. Bei 50 Datenpunkten benötigen wir also 500 MB an RAM. Dazu kommt das für die Eingabebilder benötigte RAM, also $50 \times 4 \times 200 \times 300 \times 3 = 36$ Millionen Bytes (etwa 34.3 MB) sowie das für die Modellparameter benötigte RAM, die oben berechneten 3.4 MB. Das für die Gradienten nötige RAM ignorieren wir an dieser Stelle, da es im Verlauf des Backpropagation-Verfahrens im Rückwärtsdurchlauf nach und nach freigegeben wird). Wir erhalten insgesamt etwa $500.0 + 34.3 + 3.4 = 537.7$ MB. Und das ist wirklich nur eine optimistische untere Grenze.

3. Wenn Ihrer GPU beim Trainieren eines CNN der Speicher ausgeht, können Sie folgende fünf Dinge tun, um das Problem zu bekämpfen (oder eine GPU mit mehr RAM kaufen):
 - Die Größe der Mini-Batches verringern.
 - Die Dimensionalität verringern, indem Sie in einer oder mehreren Schichten einen größeren Stride einstellen.
 - Eine oder mehrere Schichten entfernen.
 - Floats mit 16 Bit anstatt mit 32 Bit verwenden.
 - Das CNN über mehrere Geräte verteilen.
4. Eine Max-Pooling-Schicht enthält überhaupt keine Parameter, wohingegen ein Convolutional Layer recht viele enthält (siehe vorige Fragen).
5. Ein *Local Response Normalization* Layer sorgt dafür, dass die am stärksten aktivierte Neuronen die Neuronen an der gleichen Position in benachbarten Feature Maps inhibieren, wodurch die Feature Maps dazu angehalten werden, sich unterschiedlich zu entwickeln und sich eine größere Bandbreite an Mustern aneignen. Er wird normalerweise in den unteren Schichten eingesetzt, um einen größeren Pool kleinteiliger Merkmale zu erhalten, auf den die nachgeschalteten Schichten aufbauen können.
6. Die wichtigsten Innovationen bei AlexNet gegenüber LeNet-5 sind, dass (1) es wesentlich größer und tiefer ist und (2) Convolutional Layers direkt aufeinanderstapelt, anstatt auf jedem Convolutional Layer einen Pooling Layer zu platzieren. Die wichtigste Innovation bei GoogLeNet ist die Einführung von *Inception-Modulen*, die ein weitaus tieferes Netz als bei früheren CNN-Architekturen bei weniger Parametern ermöglichen. Die wichtigste Neuerung von ResNet sind die Skip-Verbindungen, mit denen mehr als 100 Schichten möglich sind. Auch seine Einfachheit und Konsistenz können als innovativ gelten.

Die Lösungen zu den Übungsaufgaben 7, 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 14: Rekurrente neuronale Netze

1. Folgendes sind einige Anwendungsgebiete von RNNs:

- Bei einem Sequenz-zu-Sequenz-RNN: Wettervorhersage (oder Vorhersage einer beliebigen Zeitreihe), maschinelle Übersetzung (mit einer Encoder-Decoder-Architektur), Videos mit Untertiteln versehen, Umwandlung von Sprache zu Text, Erzeugen von Musik (oder anderen Sequenzen) und Identifizieren der Akkorde in einem Musikstück.
 - Bei einem Sequenz-zu-Vektor-RNN: Klassifizieren von Musikstücken nach Genre, Analysieren der Meinung einer Buchrezension, aus den Signalen von Gehirnimplantaten vorhersagen, an welches Wort ein aphasischer Patient denkt, aus bereits gesehenen Videos die Wahrscheinlichkeit vorherzusagen, mit der ein Nutzer einen bestimmten Film sehen möchte (dies ist eine vieler möglichen Implementierungen von *kollaborativen Filtern*).
 - Bei einem Vektor-zu-Sequenz-RNN: Bilder mit Untertiteln versehen, eine Playlist von Musikstücken aus einem Embedding des aktuellen Interpreten zu erstellen, eine Melodie anhand einer Parameterliste erstellen, Fußgänger auf einem Bild erkennen (z.B. einer Momentaufnahme der Kamera eines selbstfahrenden Autos).
2. Wenn Sie einen Satz Wort für Wort übersetzen, ist das Ergebnis in der Regel furchtbar. Beispielsweise bedeutet der französische Satz »Je vous en prie« übersetzt »Bitte schön«, aber wenn Sie ihn Wort für Wort übersetzen, erhalten Sie »Ich Sie im beten«. Es ist viel besser, zuerst den ganzen Satz zu lesen und ihn dann zu übersetzen. Ein einfaches Sequenz-zu-Sequenz-RNN würde unmittelbar nach Lesen des ersten Worts mit dem Übersetzen beginnen, wohingegen ein Encoder-Decoder-RNN zuerst den gesamten Satz liest und diesen erst dann übersetzt. Natürlich könnte man sich auch ein einfaches Sequenz-zu-Sequenz-RNN vorstellen, das einfach nur Stille ausgibt, wenn es sich des nächsten Worts nicht sicher ist (wie es menschliche Dolmetscher bei einer Simultanübersetzung tun).
3. Um Videos anhand der visuellen Inhalte zu klassifizieren, könnte eine mögliche Architektur ein Frame pro Sekunde extrahieren, jedes Frame in ein Convolutional Neural Network eingeben, die Ausgabe des CNN in ein Sequenz-zu-Vektor-RNN eingeben und schließlich die Ausgabe durch eine Softmax-Schicht leiten, sodass Sie die Wahrscheinlichkeiten sämtlicher Kategorien erhalten. Zum Trainieren würden Sie einfach die Kreuzentropie als Kostenfunktion verwenden. Wenn Sie auch das Audiosignal zur Klassifikation verwenden möchten, könnten Sie jede Sekunde des Audiosignals in ein Spektrogramm umwandeln, dieses Spektrogramm in ein CNN einspeisen und die Ausgabe des CNN in ein RNN eingeben (zusammen mit der entsprechenden Ausgabe des zweiten CNN).

4. Ein RNN mit der Funktion `dynamic_rnn()` anstatt `static_rnn()` aufzubauen, hat mehrere Vorteile:
 - Sie basiert auf der Operation `while_loop()`, mit der man während der Backpropagation den Speicher der GPU in den CPU-Speicher auslagern kann, um Fehler wegen unzureichenden Speicherplatzes zu vermeiden.
 - Sie ist einfacher verwendbar, da sie anstatt einer Liste Tensoren (pro Schritt) einen einzelnen Tensor als Eingabe und Ausgabe annimmt (die sämtliche Schritte abdecken). Es ist nicht nötig, `stack`, `unstack` oder `transpose` zu verwenden.
 - Sie erzeugt einen kleineren Graphen, der sich in TensorBoard leichter visualisieren lässt.
5. Um Eingabesequenzen unterschiedlicher Länge zu verarbeiten, ist es am einfachsten, den Parameter `sequence_length` beim Aufruf der Funktionen `static_rnn()` oder `dynamic_rnn()` zu setzen. Alternativ könnte man die kleineren Eingaben auf die Größe der längsten Eingabesequenz auffüllen (z.B. mit Nullen, dies kann schneller als die erste Möglichkeit sein, wenn die Längen der Eingabesequenzen einander ähnlich sind). Bei Ausgabesequenzen unterschiedlicher Länge können Sie den Parameter `sequence_length` verwenden, wenn Sie die Länge jeder Ausgabesequenz im Voraus kennen (beispielsweise wenn Sie jedes Frame in einem Video mit einem Score für das Ausmaß an Gewalt versehen möchten: Die Ausgabesequenz ist dann genauso lang wie die Eingabesequenz). Wenn Sie die Länge der Ausgabesequenz nicht kennen, können Sie folgenden Trick verwenden: Geben Sie stets eine Sequenz gleicher Länge aus, aber ignorieren Sie sämtliche Ausgaben nach einem End-of-Sequence-Token (auch beim Berechnen der Kostenfunktion).
6. Um das Trainieren und Ausführen eines Deep-RNN auf mehrere GPUs zu verteilen, ist es üblich, jede Schicht auf einer anderen GPU zu platzieren (siehe Kapitel 12).

Die Lösungen zu den Übungsaufgaben 7, 8 und 9 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 15: Autoencoder

1. Einige der wichtigsten Aufgaben, für die Autoencoder verwendet werden, sind Folgende:
 - Extrahieren von Merkmalen
 - Unüberwachtes Vortrainieren
 - Dimensionsreduktion
 - Generative Modelle
 - Erkennen von Anomalien (ein Autoencoder ist grundsätzlich schlecht im Rekonstruieren von Ausreißern)

2. Wenn Sie einen Klassifikator trainieren möchten und reichlich ungelabelte Trainingsdaten besitzen, aber nur wenige Tausend gelabelte Datenpunkte, können Sie zuerst einen Deep Autoencoder auf dem gesamten Datensatz trainieren (gelabelt und ungelabelt), anschließend dessen untere Hälfte für den Klassifikator verwenden (d.h. die Schichten bis zur und einschließlich der Schicht mit den Codings wiederverwenden) und den Klassifikator mit den gelabelten Daten trainieren. Wenn Sie wenige gelabelte Daten haben, sollten Sie die wiederverwendeten Schichten beim Trainieren des Klassifikators einfüren.
3. Dass ein Autoencoder die Eingabedaten perfekt rekonstruiert, bedeutet nicht unbedingt, dass es ein guter Autoencoder ist; es könnte auch einfach ein übervollständiger Autoencoder sein, der gelernt hat, die Eingaben in die Codings und von dort in die Ausgabe zu überführen. Selbst wenn die Schicht mit den Codings nur aus einem einzelnen Neuron besteht, könnte ein sehr tiefer Autoencoder lernen, jeden Trainingsdatenpunkt auf ein anderes Coding zu übertragen (z.B. der erste Datenpunkt könnte als 0.001 codiert sein, der zweite als 0.002, der dritte als 0.003 und so weiter). Damit könnte er lernen, den richtigen Trainingsdatenpunkt »auswendig« zu rekonstruieren. Damit würden die Eingabedaten perfekt rekonstruiert, ohne dass irgendein nützliches Muster in den Daten erlernt würde. In der Praxis ist ein derartiges Mapping unwahrscheinlich, es hebt aber den Umstand hervor, dass perfekte Rekonstruktionen keine Garantie dafür sind, dass der Autoencoder etwas Nützliches gelernt hat. Wenn die Rekonstruktionen dagegen sehr schlecht sind, können Sie sich sicher sein, dass der ganze Autoencoder nichts taugt. Um die Leistung eines Autoencoders zu evaluieren, können Sie den Verlust bei der Rekonstruktion bestimmen (z.B. den MSE, die mittlere quadrierte Differenz zwischen Ausgaben und Eingaben). Auch hier ist ein hoher Verlustbetrag bei der Rekonstruktion ein sicheres Zeichen für einen schlechten Autoencoders, aber ein niedriger Betrag ist keine Garantie für einen guten. Sie sollten den Autoencoder auch entsprechend seiner Anwendung evaluieren. Wenn Sie ihn beispielsweise zum unüberwachten Vtrainieren einsetzen möchten, sollten Sie auch die Leistung des Klassifikators auswerten.
4. Ein untvollständiger Autoencoder ist einer, dessen Coding-Schicht kleiner als die Ein- und Ausgabeschichten ist. Ist sie größer, handelt es sich um einen übervollständigen Autoencoder. Die Gefahr bei einem stark untvollständigen Autoencoder ist, dass er an der Rekonstruktion der Eingaben scheitert. Die Gefahr bei einem übervollständigen Autoencoder ist, dass er die Eingaben einfach in die Ausgaben kopiert, ohne irgendwelche nützlichen Merkmale zu erlernen.
5. Sie können die Gewichte einer Encoder-Schicht mit der entsprechenden Decoder-Schicht koppeln, indem Sie die Gewichte des Decoders den transponierten Gewichten des Encoders gleichsetzen. Damit sinkt die Anzahl der

Modellparameter auf die Hälfte, das Training konvergiert schneller und mit weniger Trainingsdaten, und das Risiko für Overfitting sinkt.

6. Um die von der unteren Schicht eines Stacked Autoencoders erlernten Merkmale zu visualisieren, können Sie die Gewichte jedes Neurons plotten, indem Sie jeden Gewichtsvektor auf die Größe eines Eingabebilds bringen (z.B. beim MNIST-Datensatz würden Sie einen Vektor der Größe [784] auf [28, 28] umformen). Die von höheren Schichten erlernten Merkmale lassen sich visualisieren, indem Sie die Trainingsdatenpunkte anzeigen, die jedes Neuron am stärksten aktivieren.
7. Ein generatives Modell ist ein Modell, das zufällig Ausgaben generiert, die den Trainingsdatenpunkten ähnlich sind. Beispielsweise könnte ein erfolgreich auf dem MNIST-Datensatz trainiertes generatives Modell dazu verwendet werden, zufällig realistische Bilder von Ziffern zu erzeugen. Die Ausgaben sind normalerweise ähnlich wie die Trainingsdaten verteilt. Da MNIST viele Bilder jeder Ziffer enthält, würde das generative Modell etwa die gleiche Anzahl Bilder für jede Ziffer liefern. Einige generative Modelle lassen sich parametrisieren – um beispielsweise nur bestimmte Ausgaben zu erzeugen. Variational Autoencoder sind ein Beispiel für einen generativen Autoencoder.

Die Lösungen zu den Übungen 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Kapitel 16: Reinforcement Learning

1. Reinforcement Learning ist ein Teilgebiet des Machine Learning, bei dem Agenten Aktionen in einer Umwelt so auszuführen lernen, dass sie über einen längeren Zeitraum maximale Belohnungen erzielen. Es gibt viele Unterschiede zwischen RL und gewöhnlichem überwachten und unüberwachten Lernen. Hier sind einige davon:

- Beim überwachten und unüberwachten Lernen ist das Ziel, Muster in den Daten zu entdecken. Beim Reinforcement Learning geht es darum, eine gute Policy zu finden.
- Im Gegensatz zum überwachten Lernen bekommt der Agent keine expliziten »richtigen« Antworten gezeigt. Er muss diese durch Versuch und Irrtum herausfinden.
- Im Gegensatz zum unüberwachten Lernen gibt es eine Art Überwachung in Form von Belohnungen. Wir sagen dem Agenten nicht, wie er seine Aufgabe ausführen soll, aber wir verraten ihm, wann er Fortschritte erzielt oder scheitert.
- Beim Reinforcement Learning muss der Agent die richtige Balance zwischen dem Erkunden seiner Umwelt zum Entdecken neuer Belohnungsmöglichkeiten und dem Nutzen bereits bekannter Belohnungsquellen

finden. Im Gegensatz dazu kümmern sich überwachte und unüberwachte Lernsysteme nicht um die Erkundung; sie verwenden einfach nur die erhaltenen Trainingsdaten.

- Beim überwachten und unüberwachten Lernen sind die Trainingsdatenpunkte normalerweise unabhängig voneinander (meistens sind sie durchmischt). Beim Reinforcement Learning sind aufeinanderfolgende Beobachtungen grundsätzlich *nicht* voneinander unabhängig. Ein Agent kann eine Weile im gleichen Gebiet einer Umwelt verweilen, bevor er weiterzieht, sodass aufeinanderfolgende Beobachtungen stark miteinander korrelieren. In einigen Fällen wird ein Replay-Speicher verwendet, um sicherzustellen, dass der Trainingsalgorithmus halbwegs unabhängige Beobachtungen erhält.
2. Hier sind einige mögliche Anwendungen für Reinforcement Learning, zusätzlich zu den in Kapitel 16 erwähnten:

Personalisierte Musik

Die Umwelt ist das personalisierte Webradio des Nutzers. Der Agent ist eine Software, die entscheidet, welcher Song dem Nutzer als Nächstes vorgespielt wird. Die möglichen Aktionen sind, einen Song aus dem Verzeichnis (und dabei einen Song auszuwählen, der dem Nutzer gefällt) oder Werbung abzuspielen (und eine Werbung auszuwählen, für die sich der Nutzer interessiert). Der Agent erhält jedes Mal eine kleine Belohnung, wenn der Nutzer sich einen Song anhört, und eine größere, wenn er einen Werbespot verfolgt. Er erhält eine negative Belohnung, wenn der Nutzer einen Song oder Werbespot überspringt, und eine stark negative, wenn er abschaltet.

Marketing

Die Umwelt ist die Marketingabteilung ihres Unternehmens. Der Agent ist die Software, die anhand von Profilen und früheren Kaufentscheidungen von Kunden entscheidet, an welche Kunden eine Mailkampagne gerichtet wird. Der Agent erhält eine negative Belohnung für die Kosten der Mailkampagne und eine positive Belohnung für den geschätzten, durch diese Kampagne generierten Gewinn.

Auslieferung von Produkten

Der Agent kontrolliert eine Flotte von Lieferfahrzeugen und entscheidet, was diese bei den Depots aufnehmen, wohin sie fahren, wo sie ausladen und so weiter. Er erhält positive Belohnungen für jedes pünktlich ausgelieferte Produkt und negative Belohnungen für verspätete Auslieferung.

3. Beim Abschätzen des Werts einer Aktion versuchen Reinforcement-Learning-Algorithmen normalerweise, alle durch diese Aktion erzeugten Belohnungen aufzusummen. Dabei erhalten unmittelbare Belohnungen ein höheres Gewicht und spätere ein geringeres (berücksichtigt wird, dass Aktionen die nahe Zukunft stärker beeinflussen als die ferne). Um dies zu modellieren, wird norma-

lerweise bei jedem Schritt eine Discountrate mit einberechnet. Beispielsweise würde bei einer Discountrate von 0.9 eine zwei Schritte in der Zukunft liegende Belohnung der Höhe 100 nur mit $0.9^2 \times 100 = 81$ beim Wert der entsprechenden Aktion berücksichtigt. Sie können sich die Discountrate als ein Maß dafür vorstellen, wie stark die Zukunft im Vergleich zur Gegenwart gewertet wird: Liegt sie sehr nah an 1, ist die Zukunft beinahe genauso viel wert wie die Gegenwart. Liegt sie nahe bei 0, zählen nur unmittelbare Belohnungen. Natürlich hat dies einen immensen Einfluss auf die optimale Policy: Wenn Sie der Zukunft einen Wert beimesse, können Sie eine Menge unmittelbare Schmerzen für die Aussicht auf eventuelle Gewinne auf sich nehmen. Wenn Sie der Zukunft dagegen keinen Wert beimesse, schnappen Sie sich einfach nur jede unmittelbare Belohnung, an der Sie vorbeikommen, und investieren nie in die Zukunft.

4. Um die Leistung eines Agenten beim Reinforcement Learning zu messen, können Sie einfach dessen Belohnungen aufsummieren. In einer simulierten Umgebung können Sie viele Episoden ausführen und sich die durchschnittliche Summe der Belohnungen ansehen (und sich Minimum, Maximum, Standardabweichung und so weiter ansehen).
5. Das Problem bei der Kreditzuweisung ist der Umstand, dass ein Agent beim Reinforcement Learning beim Erhalten einer Belohnung nicht direkt erfährt, welche seiner vorausgegangenen Aktionen zu dieser Belohnung beigetragen haben. Dies ist normalerweise der Fall, wenn zwischen der Aktion und der sich ergebenden Belohnung ein langer Zeitraum lag (z.B. bei Atari-Spiel *Pong* können zwischen dem Schlagen des Balls durch den Agenten und dem Erzielen eines Punkts mehrere Dutzend Zeitschritte liegen). Das Problem lässt sich verringern, indem man dem Agenten wenn möglich kurzfristigere Belohnungen bereitstellt. Dies erfordert Hintergrundwissen zur Aufgabe. Wenn wir beispielsweise einem Agenten das Schachspiel beibringen möchten, könnten wir ihm jedes Mal eine Belohnung geben, wenn er eine gegnerische Figur schlägt, anstatt nur Belohnungen für gewonnene Partien zu verteilen.
6. Ein Agent verbringt häufig eine Weile in der gleichen Region seiner Umwelt. Daher sind dessen Erfahrungen in diesem Zeitraum einander sehr ähnlich. Dies kann zu einem Bias im Lernalgorithmus führen. Die Policy kann dann auf diese Region der Umwelt optimiert sein, aber nicht so gut außerhalb funktionieren. Um dieses Problem zu beheben, können Sie einen Replay-Speicher nutzen. Anstatt nur die jüngsten Erfahrungen zum Lernen heranzuziehen, verwendet der Agent zum Lernen eine Sammlung Erfahrungen aus seiner jüngeren und früheren Vergangenheit. (Vielleicht träumen wir deswegen: um unsere Erfahrungen des Tags erneut abzuspielen und besser aus ihnen zu lernen?)
7. Ein Off-Policy-RL-Algorithmus lernt den Wert der optimalen Policy (d.h. die bei optimalem Verhalten des Agenten zu erwartende Summe der Beloh-

nungen unter Berücksichtigung der Discountrate), unabhängig vom tatsächlichen Verhalten des Agenten. Q-Learning ist ein gutes Beispiel für solch einen Algorithmus. Im Gegensatz dazu erlernt ein On-Policy-Algorithmus den Wert der vom Agenten tatsächlich ausgeführten Policy, was die Erkundung und Nutzung einschließt.

Die Lösungen zu den Übungsaufgaben 8, 9 und 10 finden Sie in den Jupyter Notebooks unter <https://github.com/ageron/handson-ml>.

Checkliste für Machine-Learning-Projekte

Diese Checkliste begleitet Sie durch Ihre Machine-Learning-Projekte. Sie besteht aus acht wesentlichen Punkten:

1. Klären Sie die Aufgabenstellung und betrachten Sie die Gesamtsituation.
2. Beschaffen Sie sich Daten.
3. Erkunden Sie die Daten, um daraus Erkenntnisse zu gewinnen.
4. Bereiten Sie die Daten so auf, dass Machine-Learning-Algorithmen die Muster darin leichter erkennen können.
5. Probieren Sie viele unterschiedliche Modelle aus und treffen Sie eine engere Auswahl.
6. Optimieren Sie Ihre Modelle und kombinieren Sie diese zu einer guten Lösung.
7. Stellen Sie Ihre Lösung vor.
8. Starten, beobachten und warten Sie Ihr System.

Natürlich sollten Sie diese Checkliste bei Bedarf anpassen.

Klären Sie die Aufgabenstellung und betrachten Sie die Gesamtsituation

1. Definieren Sie das Geschäftsziel.
2. Wie wird Ihre Lösung eingesetzt werden?
3. Was sind die bisherigen Lösungen oder Übergangslösungen (falls vorhanden)?
4. In welchen Bereich fällt die Aufgabe (überwacht/unüberwacht, online/offline und so weiter)?
5. Wie wird die Qualität der Lösung gemessen?
6. Liefert das Qualitätsmaß einen Beitrag zum Geschäftsziel?

7. Was ist die minimale zum Erreichen des Geschäftsziels nötige Leistung?
8. Welche vergleichbaren Aufgaben gibt es? Können Sie Erfahrung oder Werkzeuge wiederverwenden?
9. Ist menschliches Expertenwissen verfügbar?
10. Wie würden Sie die Aufgabe von Hand lösen?
11. Zählen Sie Annahmen, die Sie (oder andere) bisher getroffen haben.
12. Verifizieren Sie diese Annahmen, falls möglich.

Beschaffen Sie sich Daten

Hinweis: Automatisieren Sie so viel wie möglich, sodass Sie leicht an aktuellere Daten herankommen.

1. Zählen Sie die benötigten Daten und die benötigte Menge auf.
2. Finden und dokumentieren Sie, wie Sie sich die Daten beschaffen können.
3. Prüfen Sie, wie viel Platz die Daten benötigen.
4. Prüfen Sie gesetzliche Verpflichtungen und holen Sie nötigenfalls eine Erlaubnis ein.
5. Beschaffen Sie sich die Zugriffsrechte.
6. Erstellen Sie einen Arbeitsbereich (mit genug Speicherplatz).
7. Beschaffen Sie sich die Daten.
8. Wandeln Sie die Daten in ein Format um, das sich leicht manipulieren lässt (ohne die Daten selbst zu verändern).
9. Stellen Sie sicher, dass vertrauliche Information gelöscht oder geschützt wird (z.B. anonymisiert).
10. Prüfen Sie Größe und Typ der Daten (Zeitreihe, geografische Daten und so weiter).
11. Erstellen Sie einen Testdatensatz, legen Sie diesen beiseite und schauen Sie ihn niemals an (kein Schummeln!).

Erkunden Sie die Daten

Hinweis: Versuchen Sie, während dieser Schritte die Einschätzung eines Experten einzuholen.

1. Erstellen Sie zum Untersuchen eine Kopie der Daten (erstellen Sie nötigenfalls eine kleinere Teilmenge, die sich bequem bearbeiten lässt).
2. Erstellen Sie ein Jupyter Notebook, um Ihre Untersuchung zu dokumentieren.

3. Untersuchen Sie jedes Attribut und dessen Eigenschaften:
 - Name
 - Typ (kategorisch, int/float, begrenzt/unbegrenzt, Text, strukturiert und so weiter)
 - Prozentualer Anteil fehlender Daten
 - Menge und Art von Rauschen (stochastisch, Ausreißer, Rundungsfehler und so weiter)
 - Möglicherweise nützlich für die Aufgabenstellung?
 - Art der Verteilung (normalverteilt, einheitlich, logarithmisch und so weiter.)
4. Für überwachte Lernaufgaben identifizieren Sie das/die Zielattribut(e).
5. Visualisieren Sie die Daten.
6. Untersuchen Sie Korrelationen zwischen Attributen.
7. Untersuchen Sie, wie Sie die Aufgabe von Hand lösen würden.
8. Identifizieren Sie vielversprechende Transformationen, die Sie verwenden möchten.
9. Identifizieren Sie zusätzliche Daten, die nützlich sein könnten (gehen Sie zurück zu Abschnitt »Beschaffen Sie sich Daten«).
10. Dokumentieren Sie Ihre Erkenntnisse.

Aufbereiten der Daten

Hinweise:

- Arbeiten Sie mit einer Kopie der Daten (um die Originaldaten intakt zu halten).
 - Schreiben Sie Funktionen für sämtliche benötigten Datentransformationen. Fünf Gründe hierfür sind:
 - Sie können die Daten leichter aufbereiten, wenn Sie den nächsten Datensatz erhalten.
 - Sie können die Transformationen in zukünftigen Projekten anwenden.
 - Der Testdatensatz lässt sich bereinigen und aufbereiten.
 - Neue Datenpunkte lassen sich im laufenden Betrieb bereinigen und aufbereiten.
 - Ihre Entscheidungen bei der Aufbereitung lassen sich leichter als Hyperparameter betrachten.
1. Bereinigen der Daten:
 - Reparieren oder entfernen Sie Ausreißer (optional).

- Ergänzen Sie fehlende Werte (z.B. mit Null, Mittelwert, Median und so weiter) oder entfernen Sie die entsprechenden Zeilen (oder Spalten).

2. Merkmalsauswahl (optional):

- Entfernen Sie die Attribute, die keine nützliche Information für die Bearbeitung der Aufgabe enthalten.

3. Bei Bedarf Entwickeln von Merkmalen:

- Diskretisieren Sie kontinuierliche Merkmale.
- Zerlegen Sie Merkmale (z.B. Kategorien, Datum/Zeit und so weiter).
- Fügen Sie vielversprechende Transformationen von Merkmalen hinzu (z.B. $\log(x)$, \sqrt{x} , x^2 und so weiter).
- Aggregieren Sie Merkmale zu vielversprechenden neuen Merkmalen.

4. Skalieren von Merkmale: Standardisieren oder Normalisieren von Merkmalen.

Treffen Sie eine engere Auswahl vielversprechender Modelle

Hinweise:

- Wenn der Datensatz riesig ist, sollten Sie kleinere Trainingsdatensätze generieren, sodass Sie viele unterschiedliche Modelle in kurzer Zeit trainieren können (Ihnen sollte klar sein, dass dadurch komplexe Modelle wie große neuronale Netze oder Random Forests abgestraft werden).
 - Versuchen Sie wieder, diesen Schritt so weit wie möglich zu automatisieren.
1. Trainieren Sie schnell viele unterschiedliche Modelle (z.B. lineare Modelle, Naive Bayes, SVM, Random Forests, neuronale Netze und so weiter) mit den Standardeinstellungen.
 2. Messen und vergleichen Sie deren Leistung.
 - Verwenden Sie bei jedem Modell N -fache Kreuzvalidierung und berechnen Sie Mittelwert und Standardabweichung des Qualitätsmaßes aus den N Folds.
 3. Analysieren Sie die wichtigsten Variablen jedes einzelnen Algorithmus.
 4. Untersuchen Sie, welche Art Fehler die Modelle begehen.
 - Welche Daten würde ein Mensch verwenden, um diese Fehler zu vermeiden?
 5. Führen Sie eine kurze Runde Merkmalsauswahl und Entwicklung von Merkmalen durch.
 6. Wiederholen Sie die fünf vorigen Schritte ein- oder zweimal.

7. Erstellen Sie eine Auswahl der drei bis fünf vielversprechendsten Modelle, bevorzugen Sie dabei Modelle, die unterschiedliche Arten Fehler begehen.

Optimieren des Systems

Hinweise:

- Sie sollten bei diesem Schritt so viele Daten wie möglich verwenden, besonders gegen Ende der Optimierungsphase.
 - Automatisieren Sie wie immer so viel wie möglich.
1. Optimieren Sie die Hyperparameter mithilfe einer Kreuzvalidierung.
 - Behandeln Sie die Wahl Ihrer Datentransformation als weiteren Hyperparameter, besonders wenn Sie sich Ihrer Wahl nicht sicher sind (z.B. sollten Sie fehlende Werte mit null oder dem Median ersetzen? Oder die Zeilen einfach verwerfen?).
 - Wenn es nicht gerade sehr wenige Hyperparameter gibt, ziehen Sie die Zufallssuche einer Gittersuche vor. Wenn das Training sehr lange dauert, erwägen Sie ein Bayessches Optimierungsverfahren (z.B. mit Gauß-Prozess-Priors, wie von Jasper Snoek, Hugo Larochelle und Ryan Adams beschrieben (<https://goo.gl/PEFfGr>)).¹
 2. Probieren Sie Ensemble-Methoden aus. Eine Kombination Ihrer besten Modelle funktioniert oft besser, als sie einzeln auszuführen.
 3. Sobald Sie sich Ihres endgültigen Modells sicher sind, bestimmen Sie dessen Qualität mit dem Testdatensatz, um den Fehler der Verallgemeinerung abzuschätzen.



Verändern Sie Ihr Modell nach dem Bestimmen des Verallgemeinerungsfehlers nicht mehr: Sie würden damit lediglich die Testdaten overfitten.

Stellen Sie Ihre Lösung vor

1. Dokumentieren Sie Ihr Werk.
2. Erstellen Sie eine ansprechende Präsentation.
 - Heben Sie zu Beginn das Gesamtbild hervor.
3. Erklären Sie, warum Ihre Lösung zum Geschäftsziel beiträgt.

¹ »Practical Bayesian Optimization of Machine Learning Algorithms«, J. Snoek, H. Larochelle, R. Adams (2012).

4. Vergessen Sie nicht, auf dem Weg gewonnene interessante Erkenntnisse zu erwähnen.
 - Beschreiben Sie, was funktioniert hat und was nicht.
 - Zählen Sie Ihre Annahmen und die Beschränkungen Ihres Systems auf.
5. Stellen Sie sicher, dass Ihre wichtigsten Erkenntnisse durch eine ansprechende Visualisierung oder eingängige Aussagen untermauert werden (z.B. »das mittlere Einkommen ist hat den stärksten Einfluss bei der Vorhersage von Immobilienpreisen«).

Start!

1. Bereiten Sie Ihre Lösung für den Betrieb vor (schließen Sie die Eingabedaten für den Betrieb an, schreiben Sie Unit Tests und so weiter).
2. Schreiben Sie Code, der die Leistung Ihres Systems im laufenden Betrieb in regelmäßigen Abständen prüft und bei einem Abfall ein Alarmsignal auslöst.
 - Hüten Sie sich auch vor einem langsamen Zerfall: Modelle neigen dazu, zu »verfaulen«, wenn sich Daten weiterentwickeln.
 - Das Ermitteln der Qualität kann eine menschliche Interaktion erfordern (z.B. über einen Crowdsourcing-Dienst).
 - Behalten Sie auch die Qualität Ihrer Eingabedaten im Auge (z.B. Fehlfunktionen eines Sensors, der zufällige Werte verschickt, oder veraltete Ausgabedaten eines anderen Teams). Dies ist insbesondere bei Online-Lernsystemen wichtig.
3. Trainieren Sie Ihre Modelle regelmäßig mit aktuellen Daten (automatisieren Sie so viel wie möglich).

Das duale Problem bei SVMs

Um *Dualität* zu verstehen, müssen Sie zunächst die Methode der *Lagrange-Multiplikatoren* verstehen. Der generelle Ansatz ist, die Zielgröße einer Optimierung mit Nebenbedingungen in eine Optimierung ohne Nebenbedingungen zu überführen, indem die Nebenbedingungen in die Zielfunktion verschoben werden. Betrachten wir ein einfaches Beispiel. Nehmen wir an, Sie möchten die Werte von x und y finden, die die Funktion $f(x,y) = x^2 + 2y$ minimieren, wobei *Gleichheit als Nebenbedingung* vorliegt: $3x + 2y + 1 = 0$. Bei den Lagrange-Multiplikatoren definieren wir zunächst eine neue Funktion, die sogenannte *Lagrange-Funktion* (oder *Lagrangian*): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Jede Nebenbedingung (in diesem Fall eine) wird vom ursprünglichen Zieterm subtrahiert und mit einer neuen Variablen, dem Lagrange-Multiplikator, multipliziert.

Joseph-Louis Lagrange hat Folgendes gezeigt: Falls (\hat{x}, \hat{y}) eine Lösung zum Optimierungsproblem mit Nebenbedingungen ist, so muss ein $\hat{\alpha}$ existieren, für das $(\hat{x}, \hat{y}, \hat{\alpha})$ ein *stationärer Punkt* des Lagrange-Terms ist. Ein stationärer Punkt liegt vor, wenn sämtliche partiellen Ableitungen gleich null sind. Anders ausgedrückt, können wir Punkte finden, an denen alle diese Ableitungen gleich Null sind, indem wir die partiellen Ableitungen von $g(x, y, \alpha)$ nach x , y und α ; bilden; die Lösungen dieses Optimierungsproblems mit Nebenbedingungen (falls sie existieren) müssen unter diesen stationären Punkten zu finden sein.

$$\left\{ \begin{array}{l} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{array} \right.$$

In unserem Beispiel sind die partiellen Ableitungen:

Wenn alle diese partiellen Ableitungen gleich 0 sind,
gilt $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, woraus wir leicht
nachweisen können, dass $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$ und $\hat{\alpha} = 1$ gilt.

Dies ist der einzige stationäre Punkt, und da er der Nebenbedingung genügt, muss dies die Lösung des Optimierungsproblems mit Nebenbedingung sein.

Glücklicherweise lässt sich dieses Verfahren nicht nur auf Nebenbedingungen mit Gleichheitsoperator anwenden, sondern unter bestimmten Umständen (die in einem SVM vorliegen) auch auf *Nebenbedingungen mit Ungleichheit* (z.B. $3x + 2y + 1 \geq 0$). Die *allgemeine Lagrange-Funktion* für das Hard-Margin-Problem ist in Formel C-1 angegeben, wobei man die Variablen $\alpha^{(i)}$ als *Karush-Kuhn-Tucker-(KKT)-Multiplikatoren* bezeichnet. Diese müssen gleich oder größer null sein.

Formel C-1: Allgemeine Lagrange-Funktion für das Hard-Margin-Problem

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} \left(t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) - 1 \right)$$

mit $\alpha^{(i)} \geq 0$ mit $i = 1, 2, \dots, m$

Wie bei der Methode mit den Lagrange-Multiplikatoren können Sie die partiellen Ableitungen berechnen und die stationären Punkte ermitteln. Wenn es eine Lösung gibt, muss diese unter den stationären Punkten $(\hat{\mathbf{w}}, \hat{b}, \hat{\alpha})$ zu finden sein, die den *KKT-Bedingungen* entsprechen:

- Sie berücksichtigen die Nebenbedingungen des Problems:

$$t^{(i)} ((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}) \geq 1 \quad \text{mit } i = 1, 2, \dots, m.$$
- Für sie gilt $\hat{\alpha}^{(i)} \geq 0$ mit $i = 1, 2, \dots, m$.
- Entweder gilt $\hat{\alpha}^{(i)} = 0$, oder die i^{te} Nebenbedingung muss eine *aktive Nebenbedingung* sein, was bedeutet, dass für sie Gleichheit gilt: $t^{(i)} ((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}) = 1$. Diese Bedingung nennt man die *Complementary Slackness-Bedingung*. Sie impliziert, dass entweder $\hat{\alpha}^{(i)} = 0$ gilt, oder der i^{te} Datenpunkt auf der Grenze liegt (also ein Stützvektor ist).

Beachten Sie, dass die KKT-Bedingungen notwendige Bedingungen dafür sind, dass ein stationärer Punkt eine Lösung des Optimierungsproblems mit Nebenbedingungen darstellt. Unter bestimmten Umständen sind diese Bedingungen auch hinreichend. Glücklicherweise ist dies beim Optimierungsproblem in einer SVM der Fall, sodass jeder stationäre Punkt, der den KKT-Bedingungen entspricht, garantiert eine Lösung des Optimierungsproblems mit Nebenbedingungen darstellt.

Wir können die partiellen Ableitungen der allgemeinen Lagrange-Funktion nach \mathbf{w} und b mit Formel C-2 berechnen.

Formel C-2: Partielle Ableitungen der allgemeinen Lagrange-Funktion

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \alpha) = - \sum_{i=1}^m \alpha^{(i)} t^{(i)}$$

Wenn diese partiellen Ableitungen gleich 0 sind, so gilt Formel C-3.

Formel C-3: Eigenschaften der stationären Punkte

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} = 0$$

Wenn wir dieses Ergebnis in die Definition der allgemeinen Lagrange-Funktion einfließen lassen, verschwinden einige Terme, und wir erhalten Formel C-4.

Formel C-4: Duale Form des SVM-Problems

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \alpha) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

mit $\alpha^{(i)} \geq 0$ mit $i = 1, 2, \dots, m$

Das Ziel ist nun, den Vektor $\hat{\alpha}$ zu finden, der diese Funktion minimiert, wobei für alle Datenpunkte $\hat{\alpha}^{(i)} \geq 0$ gilt. Dieses Optimierungsproblem mit Nebenbedingungen ist das gesuchte duale Problem.

Sobald Sie das optimale $\hat{\alpha}$ gefunden haben, können Sie $\hat{\mathbf{w}}$ berechnen, indem Sie die erste Zeile von Formel C-3 verwenden. Um \hat{b} zu berechnen, können Sie sich zunutze machen, dass für einen Support-Vektor $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) = 1$ gilt. Wenn also der k te Datenpunkt ein Stützvektor ist (also $\alpha_k > 0$), können Sie ihn verwenden, um $\hat{b} = 1 - t^{(k)}(\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(k)})$ zu berechnen. Allerdings zieht man oft die Berechnung des Mittelwerts über alle Stützvektoren vor, um einen stabileren und genaueren Wert zu erhalten, wie in Formel C-5 angegeben.

Formel C-5: Abschätzung des Bias-Terms über die duale Form

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left[1 - t^{(i)} (\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}) \right]$$

ANHANG D

Autodiff

Dieser Anhang erklärt, wie das Autodiff-Feature in TensorFlow funktioniert und was es im Vergleich zu seinen Alternativen leistet.

Nehmen Sie an, Sie definieren eine Funktion $f(x,y) = x^2y + y + 2$ und möchten deren partielle Ableitungen $\frac{\partial f}{\partial x}$ und $\frac{\partial f}{\partial y}$ bilden, um beispielsweise das Gradientenverfahren (oder einen anderen Optimierungsalgorithmus) durchzuführen. Ihre Auswahlmöglichkeiten sind im Wesentlichen das Differenzieren von Hand, die symbolische Differenzierung, die numerische Differenzierung, Autodiff im Forward-Modus und schließlich Autodiff im Reverse-Modus. In TensorFlow ist die letztere Möglichkeit implementiert. Betrachten wir jede dieser Möglichkeiten etwas genauer.

Differenzierung von Hand

Unser erster Ansatz ist, einen Stift und ein Stück Papier in die Hand zu nehmen und mit unserem Wissen aus der Analysis die partiellen Ableitungen von Hand zu bilden. Für die soeben definierte Funktion $f(x,y)$ ist das nicht besonders schwierig; Sie benötigen lediglich fünf Regeln:

- Die Ableitung einer Konstanten ist 0.
- Die Ableitung von λx ist λ (wobei λ eine Konstante ist).
- Die Ableitung von x^λ ist $\lambda x^{\lambda-1}$, die Ableitung von x^2 ist also $2x$.
- Die Ableitung einer Summe von Funktionen ist die Summe der Ableitungen dieser Funktionen.
- Die Ableitung von λ multipliziert mit einer Funktion ist λ multipliziert mit deren Ableitung.

Aus diesen Regeln können Sie Formel D-1 herleiten:

Formel D-1: Partielle Ableitungen von $f(x,y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2 y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2 y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Bei komplexeren Funktionen wird dieser Ansatz sehr mühselig, und das Risiko, dabei Fehler zu machen, erhöht sich. Glücklicherweise lässt sich die von uns durchgeführte Herleitung der mathematischen Gleichungen für die partiellen Ableitungen durch einen Vorgang namens *symbolische Differenzierung* automatisieren.

Symbolische Differenzierung

Abbildung D-1 demonstriert die Funktionsweise der symbolischen Differenzierung anhand einer noch einfacheren Funktion, $g(x,y) = 5 + xy$. Der Graph der Funktionsgleichung ist auf der linken Seite gezeigt. Nach der symbolischen Differenzierung erhalten wir den Graphen auf der rechten Seite, der die partielle Ableitung $\frac{\partial g}{\partial x} = 0 + (0 \cdot x + y \cdot 1) = y$ darstellt (in ähnlicher Weise könnten wir auch die partielle Ableitung nach y ermitteln).

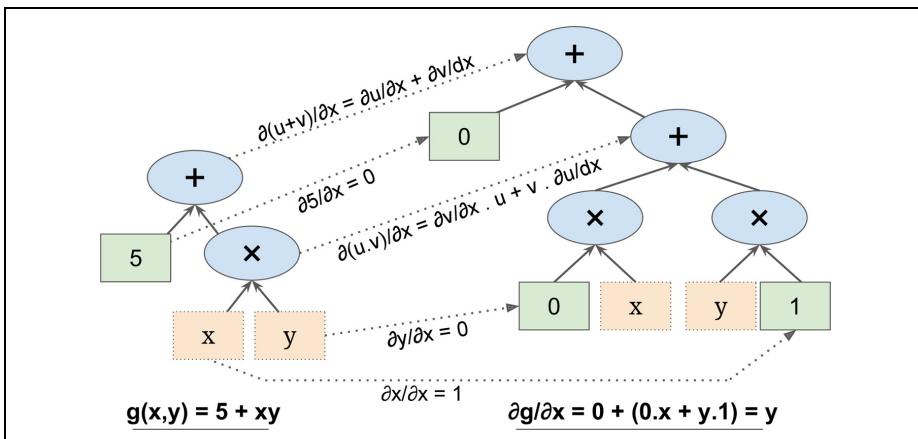


Abbildung D-1: Symbolische Differenzierung

Der Algorithmus beginnt, indem er die partiellen Ableitungen der Blätter im Graphen ermittelt. Der konstante Knoten (5) liefert die Konstante 0, da die Ableitung einer Konstanten stets 0 ist. Die Variable x liefert die Konstante 1, weil $\frac{\partial x}{\partial x} = 1$ ist, und die Variable y liefert die Konstante 0, weil $\frac{\partial y}{\partial x} = 0$ ist (würden wir nach der partiellen Ableitung nach y suchen, wäre es genau umgekehrt).

Nun haben wir alles erledigt, um im Graphen zum Knoten mit der Multiplikation in der Funktion g vorzurücken. Aus der Analysis wissen wir, dass die Ableitung eines Produkts von zwei Funktionen u und v der Gleichung $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times v$ entspricht. Wir können daher einen Großteil des Graphen auf der rechten Seite konstruieren, indem wir dort $0 \times x + y \times 1$ eintragen.

Schließlich können wir uns dem Knoten mit der Addition in der Funktion g widmen. Wie erwähnt, ist die Ableitung einer Summe von Funktionen die Summe der Ableitungen dieser Funktionen. Wir müssen daher lediglich einen Knoten mit einer Addition erstellen und ihn mit den bereits berechneten Knoten verbinden. Wir erhalten damit die korrekte partielle Ableitung:

$$\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$$

Dieses Verfahren lässt sich aber noch erheblich vereinfachen. Aus dem Graphen können mit einigen trivialen Schritten sämtliche unnötigen Operationen entfernt werden. Wir erhalten dann einen wesentlich kleineren Graphen mit nur einem Knoten:

$$\frac{\partial g}{\partial x} = y$$

In diesem Fall ist das Vereinfachen noch recht einfach, aber bei einer komplexeren Funktion kann die symbolische Differenzierung einen riesigen Graphen erzeugen, der sehr schwer zu vereinfachen ist und zu einer suboptimalen Leistung führt. Vor allem aber kann die symbolische Differenzierung nicht mit Funktionsdefinitionen in beliebigem Code arbeiten – beispielsweise scheitert das Verfahren an der folgenden in Kapitel 9 besprochenen Funktion:

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

Numerische Differenzierung

Die einfachste Lösung ist, numerisch eine Näherung der Ableitungen zu berechnen. Rufen Sie sich in Erinnerung, dass die Ableitung $h'(x_0)$ einer Funktion $h(x)$ am Punkt x_0 der Steigung der Funktion an diesem Punkt entspricht. Dies ist etwas präziser in Formel D-2 ausgedrückt.

Formel D-2: Ableitung einer Funktion $h(x)$ am Punkt x_0

$$\begin{aligned} h'(x) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\epsilon \rightarrow 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon} \end{aligned}$$

Möchten wir die partielle Ableitung von $f(x, y)$ nach x bei $x = 3$ und $y = 4$ berechnen, können wir daher einfach $f(3 + \epsilon, 4) - f(3, 4)$ berechnen und das Ergebnis durch ϵ teilen, wobei wir für ϵ eine sehr kleine Zahl verwenden. Genau das führt der folgende Code aus:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Leider ist das Ergebnis ungenau (und bei komplexeren Funktionen wird es noch schlimmer). Die korrekten Ergebnisse sind jeweils 24 und 10. Stattdessen erhalten wir:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Um beide partiellen Ableitungen zu berechnen, müssen wir $f()$ mindestens dreimal aufrufen (im obigen Code haben wir die Funktion viermal aufgerufen, dies ist jedoch optimierbar). Wenn es 1000 Parameter gäbe, müssten wir $f()$ mindestens 1001 Mal aufrufen. Bei großen neuronalen Netzen ist die numerische Differenzierung daher viel zu ineffizient.

Allerdings ist die numerische Differenzierung so einfach zu implementieren, dass sie sich hervorragend dafür eignet, die Implementierung mit anderen Methoden zu überprüfen. Wenn das Ergebnis z.B. dem Ihrer manuell abgeleiteten Funktion widerspricht, enthält Ihre Funktion vermutlich einen Fehler.

Autodiff im Forward-Modus

Autodiff im Forward-Modus ist weder eine numerische noch eine symbolische Differenzierung, aber in gewissem Sinne ein Kind dieser beiden. Das Verfahren beruht auf *dualen Zahlen*, die (kurioserweise) Zahlen der Form $a + b\epsilon$ sind, wobei a und b reale Zahlen sind. ϵ ist jedoch eine Infinitesimalzahl, für die $\epsilon^2 = 0$ (aber $\epsilon \neq 0$) gilt. Sie können sich die Dualzahl $42 + 24\epsilon$ in etwa wie $42.0000...000024$ mit einer

unendlichen Anzahl Nullen vorstellen (dies stellt natürlich eine Vereinfachung dar, die hier der Veranschaulichung dualer Zahlen dient). Im Speicher wird eine duale Zahl als ein Paar Floats repräsentiert. Beispielsweise wird $42 + 24\epsilon$ durch das Zahlenpaar (42.0, 24.0) dargestellt.

Duale Zahlen lassen sich addieren, multiplizieren und so weiter, wie Sie Formel D-3 entnehmen können.

Formel D-3: Einige Rechenoperationen mit dualen Zahlen

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Vor allem aber lässt sich nachweisen, dass $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$ gilt. Daher erhalten wir durch Berechnen von $h(a + \epsilon)$ sowohl $h(a)$ als auch die Ableitung $h'(a)$ in einem Arbeitsgang. Abbildung D-2 zeigt, wie Autodiff im Forward-Modus die partielle Ableitung von $f(x,y)$ nach x am Punkt $x = 3$ und $y = 4$ berechnet. Dazu müssen wir lediglich $f(3 + \epsilon, 4)$ berechnen; dabei erhalten wir eine duale Zahl, deren erste Komponente $f(3, 4)$ und deren zweite Komponente der Ableitung $\frac{\partial f}{\partial x}(3, 4)$ entspricht.

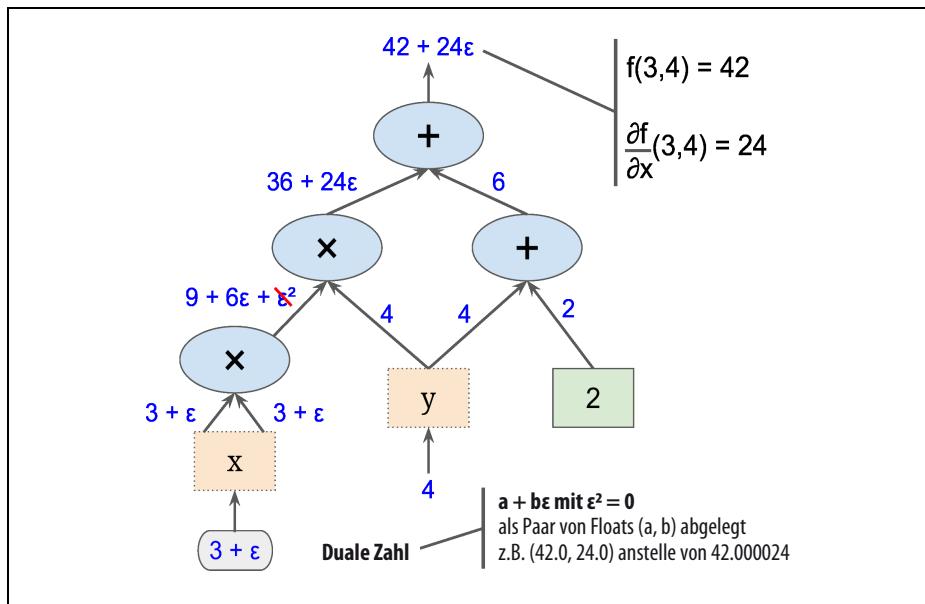


Abbildung D-2: Autodiff im Forward-Modus

Um $\frac{\partial f}{\partial y}(3, 4)$ zu berechnen, müssten wir den Graphen noch einmal durchlaufen, diesmal jedoch mit $x = 3$ und $y = 4 + \epsilon$.

Im Forward-Modus ist Autodiff also wesentlich genauer als die numerische Differenzierung, hat aber denselben Schöhnheitsfehler: Wenn es 1000 Parameter gäbe, müssten wir den Graphen 1000 Mal abschreiten, um alle partiellen Ableitungen zu berechnen. An dieser Stelle brilliert Autodiff im Reverse-Modus: Es kann sämtliche Ableitungen in nur zwei Durchläufen durch den Graphen berechnen.

Autodiff im Reverse-Modus

Die in TensorFlow implementierte Lösung ist Autodiff im Reverse-Modus. Das Verfahren schreitet den Graphen vorwärts ab (von den Eingaben zu den Ausgaben), um den Wert jedes Knotens zu berechnen. In einem zweiten Durchgang, diesmal in umgekehrter Richtung (von der Ausgabe zu den Eingaben), werden sämtliche partiellen Ableitungen berechnet. Abbildung D-3 veranschaulicht den zweiten Durchgang. Während des ersten Durchgangs wurden bereits die Werte sämtlicher Knoten berechnet, beginnend bei $x = 3$ und $y = 4$. Sie sehen diese Werte unten rechts in den jeweiligen Knoten (z.B., $x \times x = 9$). Die Knoten sind um der Klarheit willen von n_1 bis n_7 beschriftet. Der Ausgabeknoten ist $n_7: f(3,4) = n_7 = 42$.

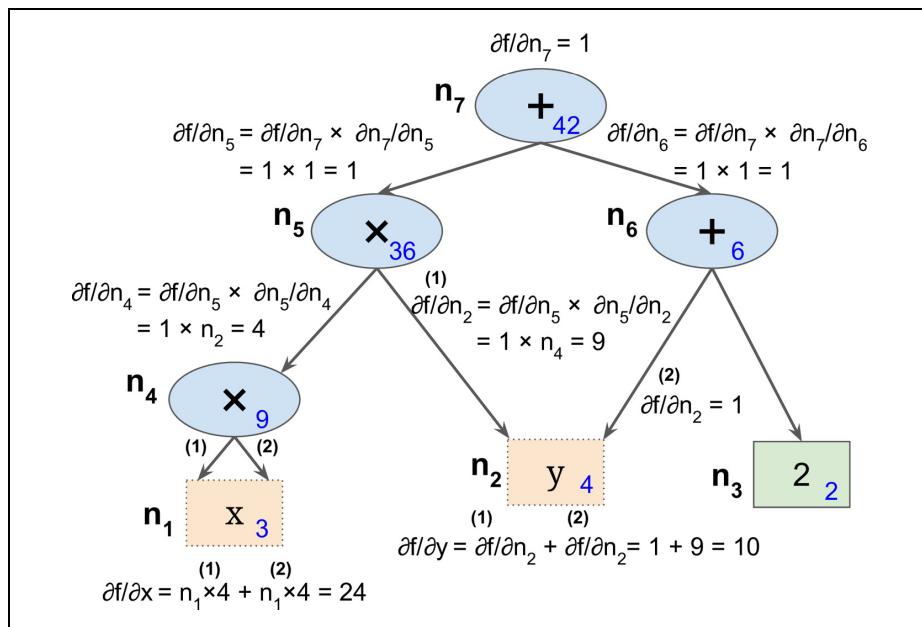


Abbildung D-3: Autodiff im Reverse-Modus

Die Grundidee ist, den Graphen schrittweise nach unten zu durchlaufen und dabei die partielle Ableitung von $f(x,y)$ nach dem jeweils folgenden Knoten zu berechnen, bis wir die Knoten mit den Variablen erreichen. Dafür stützt sich Autodiff im Reverse-Modus auf die Kettenregel in Formel D-4.

Formel D-4: Die Kettenregel

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Weil n_7 der Ausgabeknoten ist, ist $f = n_7$ trivialerweise $\frac{\partial f}{\partial n_7} = 1$.

Gehen wir im Graphen abwärts zu n_5 : Wie ändert sich f bei Änderung von n_5 ? Die Antwort ist $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. Wir kennen $\frac{\partial f}{\partial n_7} = 1$ bereits, also brauchen wir nur noch $\frac{\partial n_7}{\partial n_5}$. Da n_7 einfach die Summe $n_5 + n_6$ berechnet, erhalten wir $\frac{\partial n_7}{\partial n_5} = 1$, also ist $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Nun können wir mit dem Knoten n_4 fortfahren: Wie stark ändert sich f , wenn sich n_4 ändert? Die Antwort ist $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Wegen $n_5 = n_4 \times n_2$ erhalten wir $\frac{\partial n_5}{\partial n_4} n_2$, also ist $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

Dieser Vorgang setzt sich fort, bis wir das untere Ende des Graphen erreichen. An diesem Punkt haben wir sämtliche partiellen Ableitungen von $f(x,y)$ am Punkt $x = 3$ und $y = 4$ berechnet. In diesem Beispiel finden wir auf diese Weise heraus, dass $\frac{\partial f}{\partial y} = 10$ und $\frac{\partial f}{\partial x} = 24$ ist. Das klingt richtig!

Im Reverse-Modus ist Autodiff eine sehr mächtige und genaue Technik, besonders wenn es viele Eingabewerte und wenige Ausgaben gibt. Es sind nur ein Durchgang vorwärts und einer rückwärts nötig, um sämtliche partiellen Ableitungen der Ausgaben nach allen Eingaben zu berechnen. Vor allem kann die Methode mit beliebigen als Code definierten Funktionen umgehen. Sie kann auch nicht vollständig differenzierbare Funktionen bearbeiten, solange Sie die partiellen Ableitungen an differenzierbaren Punkten anfordern.



Wenn Sie eine neue Operation in TensorFlow implementieren und diese zu Autodiff kompatibel machen möchten, müssen Sie eine Funktion bereitstellen, die einen Teilgraphen zum Berechnen ihrer partiellen Ableitungen nach den Eingabewerten aufbaut. Nehmen wir an, Sie möchten eine Funktion schreiben, die das Quadrat ihrer Eingabe berechnet: $f(x) = x^2$. In diesem Fall müssten Sie die entsprechende Ableitungsfunktion $f'(x) = 2x$ bereitstellen. Beachten Sie, dass diese Funktion keinen numerischen Wert berechnet, sondern stattdessen einen Teilgraphen aufbaut, mit dem das Ergebnis (später) berechnet werden kann. Dies ist sehr hilfreich, weil Sie auf diese Weise Gradienten von Gradienten berechnen können (um zweite Ableitungen oder sogar noch höhere Ableitungen zu berechnen).

Weitere verbreitete Architekturen neuronaler Netze

In diesem Anhang möchten wir einen kurzen Überblick über einige historisch wichtige Architekturen neuronaler Netze geben, die heute deutlich seltener als mehrschichtige Perzeptrons (Kapitel 10), Convolutional Neural Networks (Kapitel 13), Recurrent Neural Networks (Kapitel 14) oder Autoencoder (Kapitel 15) eingesetzt werden. Sie werden aber häufig in der Fachliteratur erwähnt und werden in manchen Gebieten noch immer eingesetzt, sodass es sich lohnt, sie zu kennen. Wir werden außerdem *Deep Belief Nets* (DBNs) kennenlernen, die bis in die frühen 2010er der letzte Schrei waren. Sie werden noch immer sehr aktiv erforscht. Es ist daher möglich, dass sie in naher Zukunft zurückkehren, um sich zu rächen.

Hopfield-Netze

Hopfield-Netze wurden erstmalig von W. A. Little im Jahr 1974 entwickelt und erfuhren durch J. Hopfield im Jahr 1982 größere Beliebtheit. Sie sind *assoziative Speicher*: Man bringt ihnen zuerst einige Muster bei, und wenn sie dann ein neues Muster sehen, geben sie (hoffentlich) das ähnlichste erlernte Muster aus. Dadurch waren sie insbesondere zur Erkennung von Buchstaben nützlich, bevor sie von anderen Verfahren abgelöst wurden. Das Netz lässt sich trainieren, indem Sie ihm Bilder von Buchstaben zeigen (jedes binäre Pixel wird an ein Neuron geleitet). Wenn Sie ihm anschließend ein neues Bild eines Buchstabens zeigen, gibt es den ähnlichsten erlernten Buchstaben aus.

Hopfield-Netze sind vollständig verbundene Graphen (siehe Abbildung E-1); d.h., jedes Neuron ist mit jedem anderen verbunden. Die Bilder im Diagramm sind 6×6 Pixel groß, daher sollte das neuronale Netz 36 Neuronen (und 648 Verbindungen) enthalten. Der Klarheit wegen ist ein deutlich kleineres Netz dargestellt.

Der Trainingsalgorithmus folgt der Hebb'schen Lernregel: Bei jedem Trainingsbild wird das Gewicht zwischen zwei Neuronen erhöht, wenn jeweils beide korrespondierenden Pixel an oder aus sind, und gesenkt, falls ein Pixel an und das andere aus ist.

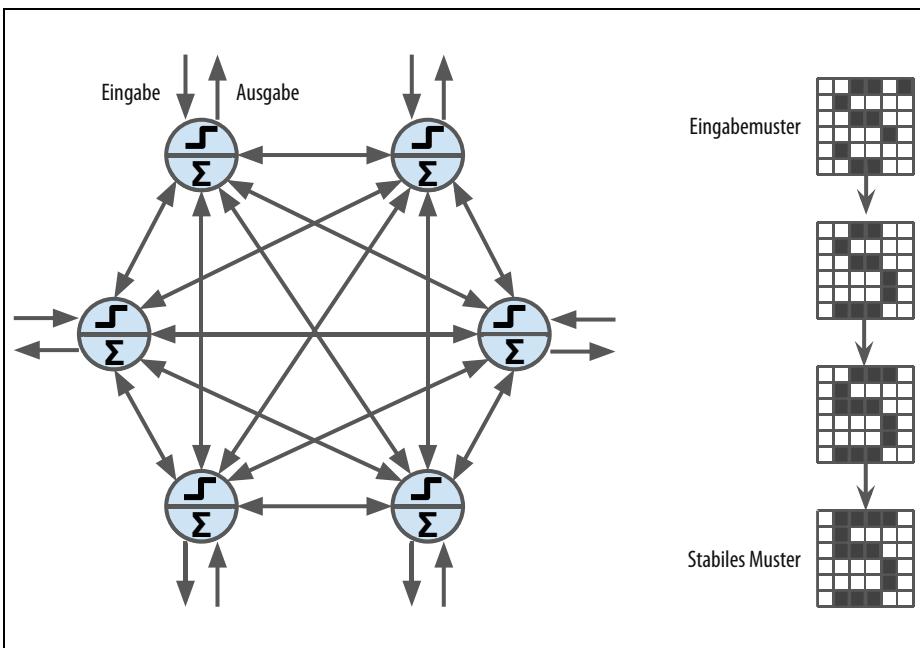


Abbildung E-1: Hopfield-Netz

Um dem Netz ein neues Bild zu zeigen, aktivieren Sie einfach die mit aktiven Pixeln korrespondierenden Neuronen. Das Netz berechnet dann die Ausgabe für jedes Neuron, und Sie erhalten ein neues Bild. Anschließend können Sie dieses Bild nehmen und den Prozess wiederholen. Nach einer Weile erreicht das Netz einen stabilen Zustand. Im Allgemeinen entspricht dieser dem Trainingsbild, das dem Eingabebild am ähnlichsten ist.

Bei Hopfield-Netzen gibt es eine sogenannte *Energiefunktion*. Die Energie wird von Iteration zu Iteration verringert, sodass das Netzwerk garantiert irgendwann einen stabilen niedrigen Energiezustand erreicht. Der Trainingsalgorithmus ändert die Gewichte derart, dass die Energiestufe der Trainingsmuster geringer ist. Damit wird es wahrscheinlich, dass sich das Netz in einer dieser energetisch günstigen Anordnungen stabilisiert. Leider können auch im Trainingsdatensatz nicht enthaltene Muster zu einem niedrigen Energiezustand führen, sodass das Netz sich bisweilen in einem nicht erlernten Zustand stabilisiert. Dies bezeichnet man auch als *unechte Muster* oder *spurious Patterns*.

Ein weiterer Nachteil von Hopfield-Netzen ist, dass sie nicht sehr gut skalieren – ihre Speicherkapazität entspricht grob 14% der Anzahl Neuronen. Um beispielsweise Bilder der Größe 28×28 zu klassifizieren, würden Sie ein Hopfield-Netz mit 784 vollständig verbundenen Neuronen und 306936 Gewichten benötigen. Solch ein Netz wäre lediglich in der Lage, etwa 110 Zeichen zu unterscheiden (14% von 784). Das sind eine Menge Parameter für ein kleines Gedächtnis.

Boltzmann Machines

Boltzmann Machines wurden im Jahr 1985 von Geoffrey Hinton und Terrence Sejnowski eingeführt. Wie Hopfield-Netze sind sie vollständig verbundene ANNs, basieren aber auf *stochastischen Neuronen*: Anstatt einer deterministischen Aktivierungsfunktion zur Bestimmung des Ausgabewerts geben diese Neuronen mit einer bestimmten Wahrscheinlichkeit 1 und andernfalls 0 aus. Die von diesen ANNs verwendete Wahrscheinlichkeitsfunktion beruht auf der Boltzmann-Verteilung (bekannt aus der statistischen Mechanik), daher der Name. In Formel E-1 finden Sie die Wahrscheinlichkeit, dass ein Neuron eine 1 ausgibt.

Formel E-1: Wahrscheinlichkeit, dass das i^{te} Neuron eine 1 ausgibt

$$p(s_i^{\text{(nächster Schritt)}} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{i,j} s_j + b_i}{T}\right)$$

- s_j ist der Zustand des j^{ten} Neurons (0 oder 1).
- $w_{i,j}$ ist das Gewicht der Verbindung zwischen dem i^{ten} und j^{ten} Neuron. Beachten Sie, dass $w_{i,i} = 0$ gilt.
- b_i ist der Bias-Term des i^{ten} Neurons. Dieser lässt sich implementieren, indem wir dem Netz ein Bias-Neuron hinzufügen.
- N ist die Anzahl Neuronen im Netz.
- T ist die *Temperatur* des Netzes; je höher die Temperatur, desto zufälliger ist die Ausgabe (umso näher liegt die Wahrscheinlichkeit an 50%).
- σ ist die logistische Funktion.

Neuronen in Boltzmann Machines lassen sich in zwei Gruppen einteilen: *Visible Units* und *Hidden Units* (siehe Abbildung E-2). Alle Neuronen arbeiten auf die gleiche stochastische Weise, aber die Visible Units sind diejenigen, die die Eingabe erhalten und von denen das Ergebnis ausgelesen wird.

Aufgrund seiner stochastischen Eigenschaften stabilisiert sich eine Boltzmann Machine nicht in einer bestimmten Anordnung, sondern wechselt ständig zwischen vielen Konfigurationen hin und her. Wenn man es lange genug laufen lässt, hängt die Wahrscheinlichkeit, eine bestimmte Konfiguration zu beobachten, lediglich von den Gewichten der Verbindungen und Bias-Termen ab, nicht von der ursprünglichen Konfiguration (wenn Sie analog dazu einen Kartenstapel lange genug mischen, hängt die Anordnung der Karten nicht vom Ausgangszustand ab). Sobald das Netz diesen Zustand erreicht, indem die ursprüngliche Konfiguration »vergessen« wurde, bezeichnet man dies als *thermisches Gleichgewicht* (auch wenn sich die Konfiguration ständig verändert). Durch entsprechendes Einstellen der Parameter, Erreichen des thermischen Gleichgewichts und anschließendes Beob-

achten des Zustands können wir ein breites Spektrum an Wahrscheinlichkeitsverteilungen simulieren. Dies bezeichnet man als *generatives Modell*.

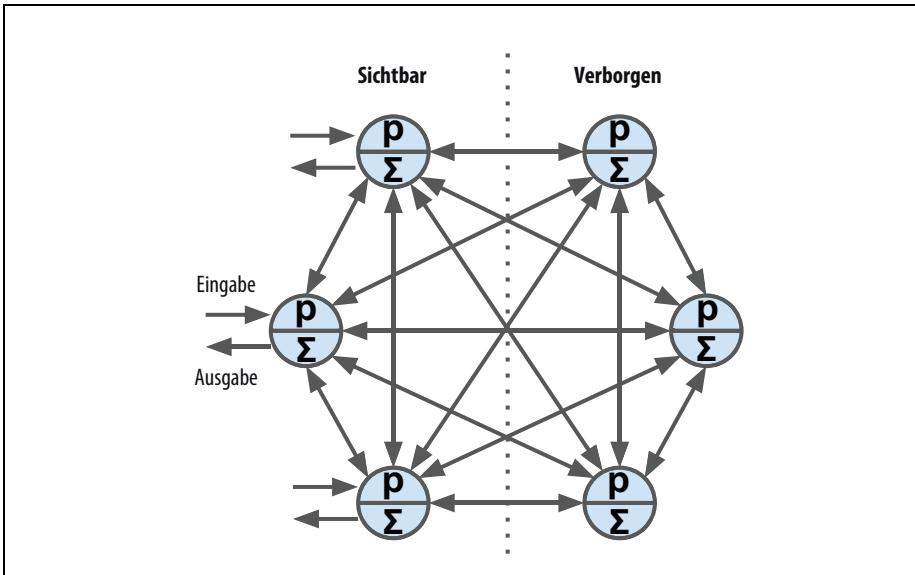


Abbildung E-2: Boltzmann Machine

Das Trainieren einer Boltzmann Machine besteht im Finden der Parameter, für die das Netz die Wahrscheinlichkeitsverteilung der Trainingsdaten approximiert. Wenn es beispielsweise drei Visible Units gibt und die Trainingsdaten 75% (0, 1, 1)-Tripel, 10% (0, 0, 1)-Tripel und 15% (1, 1, 1)-Tripel enthalten, können Sie die Boltzmann Machine nach dem Trainieren einsetzen, um zufällige Tripel mit in etwa der gleichen Wahrscheinlichkeitsverteilung zu generieren. Zum Beispiel wäre die Ausgabe in etwa 75% der Fälle ein (0, 1, 1)-Tripel.

Solch ein generatives Modell lässt sich unterschiedlich einsetzen. Wenn Sie es z. B. mit Bildern trainieren und dann ein unvollständiges oder verrauschtetes Bild in das Netz einspeisen, wird es das Bild automatisch »reparieren«. Sie können ein generatives Modell auch zur Klassifikation einsetzen. Fügen Sie einfach einige Visible Units hinzu, die die Kategorie des Trainingsbilds codieren (z. B. fügen Sie zehn Visible Units hinzu und aktivieren nur das 5. Neuron, wenn das Trainingsbild die Ziffer 5 enthält). Mit einem neuen Bild aktiviert das Netz dann automatisch die entsprechenden Visible Units, die der Kategorie des Bilds entsprechen (z. B. wird das 5. Neuron aktiviert, wenn das Bild eine 5 enthält).

Leider gibt es keine effizienten Techniken, um Boltzmann Machines zu trainieren. Es gibt aber recht effiziente Algorithmen, die entwickelt wurden, um *Restricted Boltzmann Machines* (RBMs) zu trainieren.

Restricted Boltzmann Machines

Eine RBM ist nichts weiter als eine Boltzmann Machine, bei der es keine Verbindungen zwischen Visible Units oder zwischen Hidden Units untereinander gibt, nur zwischen Visible und Hidden Units. Beispielsweise stellt Abbildung E-3 eine RBM mit drei Visible Units und vier Hidden Units dar.

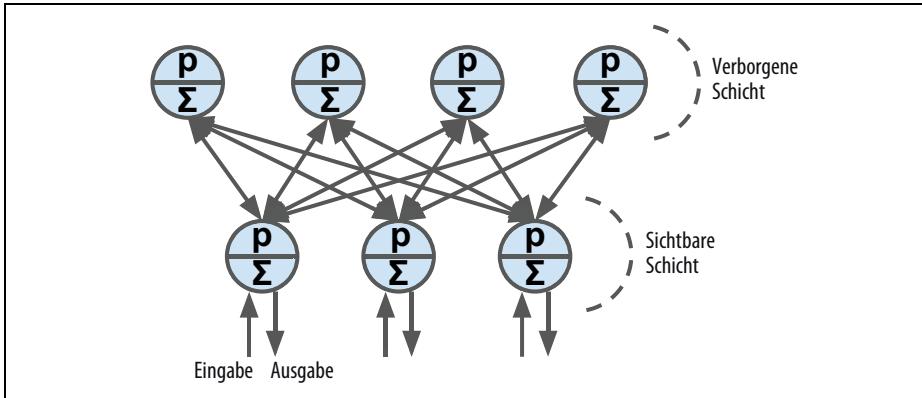


Abbildung E-3: Restricted Boltzmann Machine

Im Jahr 2005 wurde von Miguel Á. Carreira-Perpiñán und Geoffrey Hinton (<http://goo.gl/ZCP6Ir>)¹ ein sehr effizienter Trainingsalgorithmus namens *Contrastive Divergence* vorgestellt. Er funktioniert folgendermaßen: Der Algorithmus speist jeden Trainingsdatenpunkt \mathbf{x} in das Netz ein, indem er den Zustand der Visible Units auf x_1, x_2, \dots, x_n setzt. Anschließend wird der Zustand der Hidden Units über die oben beschriebene stochastische Gleichung berechnet (Formel E-1). Damit erhalten Sie den Hidden-Vektor \mathbf{h} (wobei h_i dem Zustand des i^{ten} Neurons entspricht). Anschließend berechnen Sie den Zustand der Visible Units mit derselben stochastischen Gleichung. Damit erhalten Sie einen Vektor \mathbf{x}' . Danach berechnen Sie wieder den Zustand der Hidden Units, wodurch Sie den Vektor \mathbf{h} erhalten. Nun können Sie mit der in Formel E-2 angegebenen Formel die Gewichte aktualisieren.

Formel E-2: Aktualisieren der Gewichte im Contrastive-Divergence-Algorithmus

$$w_{i,j}^{(\text{nächster Schritt})} = w_{i,j} + \eta (\mathbf{x} * \mathbf{h}^T - \mathbf{x}' * \mathbf{h}^T)$$

Der wesentliche Vorteil dieses Algorithmus ist, dass Sie nicht darauf warten müssen, dass das Netz ein thermisches Gleichgewicht erreicht: Es geht einfach nur vor, zurück, wieder vor, und das ist alles. Dadurch ist dieser Algorithmus früheren Algorithmen deutlich überlegen, und er war ein wichtiger Beitrag zu den ersten Erfolgen von Deep Learning mit multiplen gestapelten RBMs.

1 »On Contrastive Divergence Learning«, M. Á. Carreira-Perpiñán and G. Hinton (2005).

Deep Belief Nets

RBM s lassen sich in mehreren Schichten übereinanderstapeln; die Hidden Units des ersten RBM dienen als Visible Units für das RBM der zweiten Schicht und so weiter. Solch einen Stapel RBMs nennt man auch ein *Deep-Belief-Netz* (DBN).

Yee-Whye Teh, ein Schüler Geoffrey HINTONS, hatte beobachtet, dass sich DBNs mit dem Contrastive-Divergence-Algorithmus Schicht für Schicht trainieren lassen. Dabei beginnt man mit den unteren Schichten und arbeitet sich dann schrittweise zu den höheren Schichten vor. Dies führte zu dem bahnbrechenden Artikel, der 2006 eine Flutwelle im Deep-Learning-Bereich auslöste (<http://goo.gl/BcZQrH>).²

Wie RBMs lernen auch DBNs, die Wahrscheinlichkeitsverteilung der Eingabedaten ohne Überwachung zu reproduzieren. Sie sind aber viel besser darin, aus dem gleichen Grund, aus dem auch Deep Neural Networks den einfacheren Netzen überlegen sind: Realistische Daten bestehen häufig aus hierarchisch organisierten Mustern, und DBNs machen sich dies zunutze. Die niedrigeren Schichten erlernen einfache Merkmale in den Eingabedaten, und die höheren Schichten erlernen Merkmale auf einer höheren Ebene.

Wie RBMs sind auch DBNs grundsätzlich unüberwacht. Sie lassen sich aber auch überwacht trainieren, indem Sie einige Visible Units zum Repräsentieren der Labels hinzufügen. Eine weitere herausragende Eigenschaft von DBNs ist, dass sie sich in einem halbüberwachten Modus trainieren lassen. In Abbildung E-4 ist ein für halbüberwachtes Lernen konfiguriertes DBN dargestellt.

Zunächst wird RBM 1 ohne Überwachung trainiert. Es lernt die Merkmale der Trainingsdaten auf niedriger Ebene. Dann wird RBM 2 mit den Hidden Units von RBM 1 als Eingabe trainiert, auch diesmal ohne Überwachung: Es lernt Merkmale auf höherer Ebene (beachten Sie, dass die Hidden Units in RBM 2 nur aus den drei Neuronen auf der rechten Seite bestehen, nicht aus den Label-Neuronen). Weitere RBMs ließen sich auf diese Weise hintereinander schalten, aber Sie verstehen das Grundprinzip. Bisher war das Training zu 100% unüberwacht. Schließlich wird RBM 3 sowohl mit den Hidden Units von RBM 2 als Eingabe als auch mit zusätzlichen Visible Units trainiert, die die Labels repräsentieren (z.B. ein One-Hot-Vektor mit den Zielkategorien). Das Netz lernt, die abstrakteren Merkmale mit den Trainings-Labels zu assoziieren. Dies ist der überwachte Schritt.

Wenn Sie RBM 1 nach dem Training einen neuen Datenpunkt vorstellen, pflanzt sich dessen Signal zu RBM 2 und danach zu RBM 3 fort und erreicht schließlich die Neuronen mit den Labels; mit etwas Glück leuchtet dann das richtige Label auf. Auf diese Weise lässt sich ein DBN zur Klassifikation einsetzen.

² »A Fast Learning Algorithm for Deep Belief Nets«, G. Hinton, S. Osindero, Y. Teh (2006).

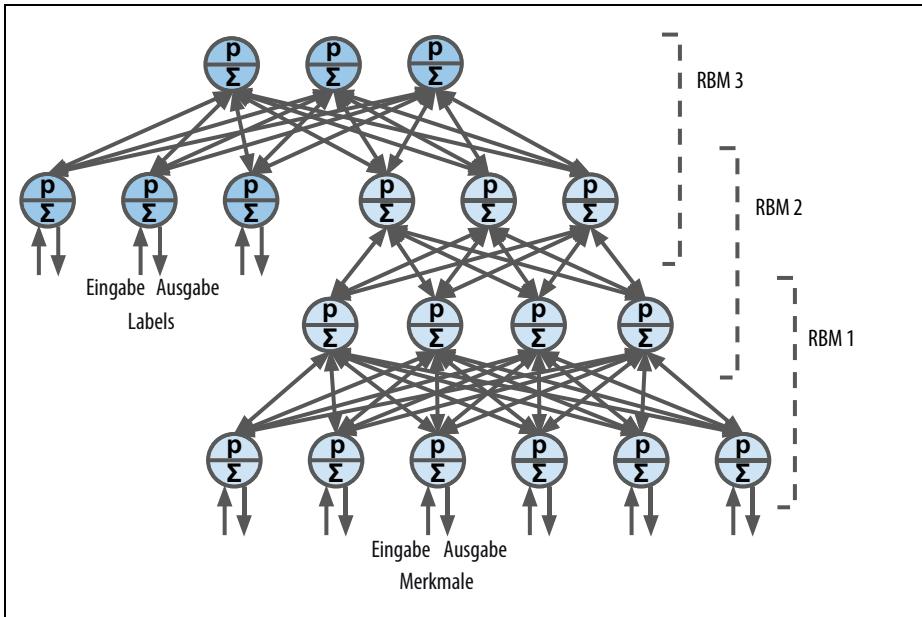


Abbildung E-4: Ein für halbüberwachtes Lernen konfiguriertes Deep Belief Network

Ein wesentlicher Vorteil dieses halbüberwachten Ansatzes ist, dass Sie nicht viele gelabelte Trainingsdaten benötigen. Wenn die unüberwachten RBMs gut funktionieren, ist nur eine geringe Anzahl gelabelter Trainingsdatenpunkte pro Kategorie notwendig. Genauso lernt ein Baby, Gegenstände ohne Überwachung zu erkennen, wenn Sie auf einen Stuhl zeigen und »Stuhl« sagen, kann das Baby das Wort »Stuhl« mit einer ganzen Klasse Objekte verbinden, die es bereits selbst erkennen kann. Sie müssen nicht auf jeden einzelnen Stuhl zeigen und »Stuhl« sagen; einige Beispiele reichen aus (gerade genug, sodass das Baby sicher sein kann, dass Sie tatsächlich den Stuhl meinen und nicht dessen Farbe oder einen seiner Bestandteile).

Interessanterweise funktionieren DBNs auch umgekehrt. Wenn Sie eines der gelabelten Neuronen aktivieren, pflanzt sich das Signal bis zu den Hidden Units von RBM 3 fort, dann weiter zu RBM 2 und RBM 1. Am Ende geben die Visible Units von RBM 1 einen neuen Datenpunkt aus. Dieser Datenpunkt sieht für gewöhnlich aus wie ein gewöhnlicher Vertreter der Kategorie, dessen Label Sie aktiviert haben. Diese generative Fähigkeit von DBNs ist ausgesprochen mächtig. Sie wurde z. B. eingesetzt, um Bilder automatisch zu beschriften: Ein DBN wird (ohne Überwachung) trainiert, um die Eigenschaften von Bildern zu erlernen, und ein zweites DBN wird (ebenfalls unüberwacht) trainiert, um Eigenschaften in Bildbeschriftungen zu erlernen (z. B. tritt »Auto« oft gemeinsam mit »Fahrzeug« auf). Anschließend wird auf beiden DBNs ein RBM aufgesetzt und mit einem Satz Bilder und deren Beschriftungen trainiert; es lernt, abstrakte Merkmale der Bilder mit abstrakten Merkmalen der Beschriftungen zu assoziieren. Wenn Sie dem DBN

anschließend das Bild eines Autos präsentieren, pflanzt sich das Signal durch das Netz bis zum RBM an der Spitze fort. Dann dringt es bis zum Beginn des DBN für die Beschriftungen vor, sodass Sie eine Beschriftung für das Bild erhalten. Wegen der stochastischen Eigenschaften von RBMs und DBNs ändert sich die Beschriftung zufällig, wird aber im Allgemeinen passend zum Bild sein. Wenn Sie einige Hundert Beschriftungen generieren, sind die am häufigsten erzeugten normalerweise gut.³

Selbstorganisierende Karten

Selbstorganisierende Karten (SOM) unterscheiden sich stark von allen anderen bisher betrachteten Arten neuronaler Netze. Sie erzeugen aus einem höher dimensionalen Datensatz eine Repräsentation mit wenigen Dimensionen, die allgemein zur Visualisierung, zum Clustern oder zur Klassifikation dient. Die Neuronen sind wie in Abbildung E-5 gezeigt über eine Karte verteilt (zur Visualisierung typischerweise in 2-D, es ist aber jede gewünschte Anzahl Dimensionen möglich). Jedes Neuron besitzt eine gewichtete Verbindung zu jedem Eingabewert (das Diagramm zeigt nur zwei Eingabewerte, aber deren Anzahl ist normalerweise sehr groß, da der Zweck einer SOM in der Dimensionsreduktion besteht.

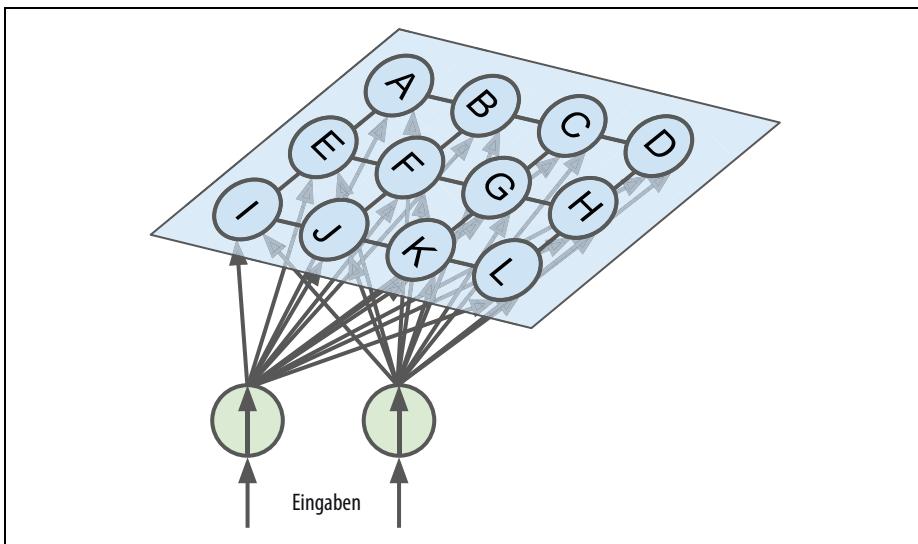


Abbildung E-5: Selbstorganisierende Karten

Ist das Netz erst einmal trainiert, können Sie einen neuen Datenpunkt eingeben. Dieser aktiviert nur ein Neuron (also einen Punkt auf der Karte): das Neuron, des-

³ Details und eine Demonstration finden Sie in diesem Video von Geoffrey Hinton: <http://goo.gl/7Z5-QiS>.

sen Gewichtsvektor zum Eingabevektor am ähnlichsten ist. Im Allgemeinen aktivieren im Eingaberaum ähnliche Datenpunkte nahe beieinanderliegende Neuronen auf der Karte. Das macht SOMs zur Visualisierung (insbesondere können Sie Cluster auf der Karte leicht identifizieren), aber auch für Anwendungen wie Spracherkennung so nützlich. Wenn beispielsweise jeder Datenpunkt die Tonaufnahme eines Vokals ist, werden unterschiedlich betonte Vokale »a« Neuronen im gleichen Gebiet der Karte aktivieren, während Aufnahmen des Vokals »e« Neuronen in einem anderen Gebiet aktivieren, und Geräusche dazwischen aktivieren in der Mitte gelegene Neuronen auf der Karte.



Ein wichtiger Unterschied zu den anderen in Kapitel 8 vorgestellten Verfahren zur Dimensionsreduktion ist, dass sämtliche Datenpunkte auf eine diskrete Anzahl Punkte in einem niedriger dimensionierten Raum projiziert werden (ein Punkt pro Neuron). Wenn es sehr wenige Neuronen gibt, sollte man diese Technik eher Clustering als Dimensionsreduktion nennen.

Der Trainingsalgorithmus ist unüberwacht. Er arbeitet, indem sämtliche Neuronen miteinander konkurrieren. Zuerst werden sämtliche Gewichte mit Zufallswerten initialisiert. Dann wird ein zufällig ausgewählter Datenpunkt in das Netz eingespeist. Alle Neuronen berechnen den Abstand zu ihren Gewichtsvektoren (dies ist anders als bei allen bisher betrachteten künstlichen Neuronen). Das Neuron mit dem kürzesten Abstand gewinnt und ändert sein Gewicht ein wenig in Richtung des Eingabevektors, sodass es zukünftige Wettbewerbe um ähnliche Datenpunkte noch wahrscheinlicher gewinnt. Es ermuntert auch seine Nachbarneuronen, deren Gewichte in Richtung des Eingabevektors zu ändern (diese ändern ihre Gewichte jedoch nicht so stark wie das Gewinnerneuron). Anschließend wählt der Algorithmus einen weiteren Datenpunkt aus und wiederholt den Vorgang wieder und wieder. Dieser Algorithmus führt dazu, dass sich nahe beieinandergelegene Neuronen nach und nach auf ähnliche Eingabedaten spezialisieren.⁴

4 Sie können sich eine Klasse Kinder mit ähnlichen Fähigkeiten vorstellen. Ein Kind ist vielleicht ein wenig besser im Basketball. Dies motiviert es, mehr mit seinen Freunden zu üben. Nach einer Weile wird diese Gruppe so gut im Basketball, dass ihnen die anderen Kinder nicht das Wasser reichen können. Das ist aber in Ordnung, weil die anderen Kinder sich auf andere Gebiete spezialisieren. Nach einer Weile besteht die gesamte Klasse aus kleinen Gruppen von Spezialisten.

Index

Symbolen

ℓ_∞ -Norm 39
 ℓ_0 -Norm 39
 ℓ_1 - und ℓ_2 -Regularisierung 305, 306
 ℓ_1 -Norm 40, 131, 140, 302, 305
 ℓ_2 -Norm 39, 129, 131, 140, 143, 305, 309
 ℓ_k -Norm 39

A

Abstimmverfahren unter Klassifikatoren 181
Achsenabschnitt 108
AdaBoost 192
AdaGrad 298
Adam-Optimierung 300, 302
adaptive Lernrate 299
adaptive Optimierung von Momenten 300
Agenten 446
Ähnlichkeitsfunktion 151
Ähnlichkeitsmaß 17
Aktionen, evaluieren 455
Aktivierungsfunktionen 257, 262
nicht sättigende 279
Aktoren 471
AlexNet, Architektur 373
Algorithmen
Daten vorbereiten für 60
genetische 448
Vorbereiten von Daten für 69
zur Visualisierung 10
AlphaGo 14, 253, 445, 461
Anaconda 41
Annahmen überprüfen 40
Anomalien erkennen 12
Anteil erklärter Varianz 214
Apples Siri 253
apply_gradients() 286, 458

Arbeitsverzeichnis 40
Area under the Curve (AUC) 94
array_split() 217
assign() 235
Assoziationsregeln, Lernen mit 12
assoziative Speicher 527
asynchrone Kommunikation 334
asynchrone Updates 353
atrous_conv2d() 382
Attribute 9
Aufgaben abstecken 35
Aufmerksamkeitsmechanismus 415
Ausgabeschicht 261
Aussagenlogik 254
Auswahl von Merkmalen 26
Autodiff 236, 519
Differenzierung von Hand 519
Forward-Modus 522
numerische Differenzierung 521
Reverse-Modus 524
symbolische Differenzierung 520
Autoencoder 419
Adversarial 441
Contractive 440
Denoising 432
effiziente Repräsentation von Daten 419
generative 436
generative stochastische Netze (GSN) 441
PCA mit einem unvollständigen linearen Autoencoder 421
probabilistische 436
Rekonstruktionen 420
Sparse 434
Stacked 423
Stacked Convolutional 440
überganz 432

- unvollständig 421
Variational 436
Visualisieren von Merkmalen 429
Winner-Take-All (WTA) 441
Average-Pooling Layer 371
avg_pool() 371
- B**
- Backpropagation 261, 275, 294, 430
Backpropagation through Time (BPTT) 395
Bagging und Pasting 185
 Out-of-Bag-Evaluation 187
 in Scikit-Learn 186
BasicLSTMCell 407
BasicRNNCell 403
batch_join() 345
batch_normalization() 284
batch() 345
Batch-Gradientenverfahren 116, 132
Batch-Learning 14
Batch-Normalisierung 282, 379
 mit TensorFlow 284
 Zusammenfassung 282
Belohnungen, in RL 446
Beobachtungsraum 454
Beschleunigter Gradient nach Nesterov
 (NAG) 297
Between-Graph-Replikation 347
Bias 267
Bias-Neuronen 258
Bias-Term 108
Bildklassifikation 371
binäre Klassifikatoren 83, 135
biologische Neuronen 254
Black-Box-Modelle 170
Blending 200
Boltzmann Machines *siehe* Restricted Boltzmann Machines (RBMs)
Boosting 191
 AdaBoost 192
 Gradient Boosting 195
Bootstrap-Aggregation *siehe* Bagging und Pasting
Bootstrapping 74, 185, 476
brew 202
- C**
- Caffe Model Zoo 293
CART-(Classification and Regression Tree-)Algorithmus 169, 176
 χ^2 -Test *siehe* Chi-Quadrat-Test
Chi-Quadrat-Test 174
- clip_by_value() 286
Cluster 328
Clustering, hierarchisches 10
Clustering-Algorithmen 10
Cluster-Spezifikation 328
Coderraum 437
Codings 419
Complementary Slackness-Bedingung 516
components_ 214
compute_gradients() 286, 457
concat() 376
config.gpu_options 322
ConfigProto 321
Contrastive Divergence 531
Control Dependencies 327
conv1d() 382
conv2d_transpose() 382
conv3d() 382
Convolution Kernels 371
Convolutional Neural Networks (CNNs) 359, 384
 Architekturen 371
 AlexNet 373
 GoogleNet 375
 LeNet5 372
 ResNet 378
 Convolutional Layer 361, 376, 382
 Feature Maps 364
 Filter 363
 Speicherbedarf 368
 Entstehung von 360
 Implementierung in TensorFlow 366
 Pooling Layer 369
Convolution-Kernels 363, 376
Coordinator, Klasse 342
Credit-Assignment-Problem 455
CUDA, Bibliothek 319
cuDNN, Bibliothek 319
- D**
- Data Augmentation 311
Data Mining 7
Data Snooping-Bias 49
DataFrame 60
Dataquest XVIII
Daten 45
 siehe auch Testdatensatz; Trainingsdaten
Annahmen treffen über 31
Arbeitsumgebung erstellen für 40
aus einer Queue holen 335
herunterladen 44, 45
in eine Queue stellen 334

- Korrelationen finden in 56
 Testdatensatzes erstellen 49
 Umgang mit realen Daten 33
 vorbereiten für Machine-Learning-Algorithmen 60, 68
 Datenaufbereitung 60
 Datenpipeline 36
 Datenstruktur 48
 Datenvizualisierung 53
 Decision Stumps 195
 Decision Trees 70
 Decoder 420
 deConvolutional Layer 382
 Deep Autoencoder *siehe* Stacked Autoencoder
 Deep Belief Nets (DBNs) 13, 532
 Deep Learning
 siehe auch Reinforcement Learning; TensorFlow
 Bibliotheken 228
 Einführung XVIII
 über XV
 Deep-Learning-Netze (DNNs) 275
 siehe auch Multi-Layer Perceptrons (MLP)
 instabile Gradienten 276
 Regularisierung 305
 Richtlinien zum Trainieren 312
 schnellere Optimierer für 295
 schwindende und explodierende Gradienten 275
 trainieren mit TensorFlow 265
 trainieren mit TF.Learn 264
 Wiederverwenden vortrainierter Schichten 287
 DeepMind 14, 253, 445, 468
 Deep-Q-Learning 468
 Ms. Pac Man, Beispiel 469
 Deep-Q-Netz 468
 Deep-RNNs 402
 Drop-out verwenden 404
 Schwierigkeiten bei langen Sequenzen 405
 Truncated Backpropagation through Time 406
 verteilen über mehrere GPUs 403
 Denoising Autoencoder 432
 dense() 267, 425
 Depth Concat Layer 376
 depthwise_conv2d() 382
 dequeue_many() 336, 338
 dequeue_up_to() 337
 enqueue() 336
 describe() 46
 device() 323
 device-Blöcke 331
 Differenzierung, automatische 229
 Dimensionsreduktion 12, 205, 419
 Ansätze zur
 Manifold Learning 210
 Projektion 207
 Auswählen der richtigen Anzahl Dimensionen 215
 und Datenvizualisierung 205
 Fluch der Dimensionalität 205
 Isomap 223
 LLE (Locally Linear Embedding) 221
 multidimensionale Skalierung 223
 PCA (Hauptkomponentenzerlegung) 211
 t-verteiltes Stochastic Neighbor Embedding (t-SNE) 223
 Discount-Rate 455
 DNNClassifier 264
 drop() 60
 Dropconnect 309
 dropna() 60
 Drop-out 272, 306, 309, 404
 dropout() 308
 Drop-out-Rate 307
 DropoutWrapper 405
 DRY (Don't Repeat Yourself) 245
 Dual Averaging 302
 duale Zahlen 522
 duales Problem 160
 Dualität 515
 dynamic_rnn() 393, 403, 415
 dynamische Platzierung 322, 324
 dynamische Programmierung 464
 dynamisches Aufrollen entlang der Zeitachse 393
- ## E
- Early Stopping 134, 272, 305
 Eingabeneuronen 258
 eingefrorene Schichten 291
 Einlesen mit mehreren Threads 342
 Elastic Net 133
 Embedded Reber Grammars 416
 embedding_lookup() 412
 Embeddings 411
 Encoder 420
 Encoder-Decoder 389
 End-of-Sequence-(EOS-)Token 394

Energiefunktion 528
Ensemble Learning 72, 76, 181
 Bagging und Pasting 185
 Boosting 191
 In-Graph- versus Between-Graph-Replikation 347
 Random Forests *siehe* Random Forests
 Stacking 200
 zufällige Patches und zufällige Subräume 188
Entropie als Maß für Unreinheit 173
Entscheidungsbäume 72, 179, 181
 Anzahl Kinder 169
 Binäräbäume 169
 Entscheidungsgrenzen 170
 geschätzte Wahrscheinlichkeiten für Kategorien 171
 GINI-Unreinheit 173
 Hyperparameter zur Regularisierung 173
 Instabilität bei 177
 Komplexität der Berechnung 172
 Random Forests *siehe* Random Forests
 Regressionsaufgaben 175
 trainieren und visualisieren 167
 Vorhersagen 168
Entscheidungsfunktion 89, 156
Entscheidungsgrenzen 89, 138, 143, 170
Episoden (beim RL) 452, 456, 459, 476
Epochen 120
 ϵ -greedy Policy 467, 473
 ϵ -insensitiv 154
Erkundungspolicies 467
Estimatoren 62
Euklidischer Abstand 39
eval() 238
explodierende Gradienten *siehe* Gradienten, schwindende und explodierende
Exponential Linear Unit (ELU) 280
Extra-Trees 190

F

F-1-Score 88
Falsch-positiv-Rate (FPR) 93
Fan-in 277, 279
Fan-out 277, 279
Feature Maps 363, 380
FeatureUnion 69
feed_dict 238
Feed-Forward-Netz (FNN) 263
Fehler, nicht reduzierbarer 128
Fehleranalyse 98

FIFOQueue 334, 337
fillna() 60
First-In-First-Out-Queues (FIFO) 334
fit_transform() 62, 67
fit() 61, 67, 217
Fitnessfunktion 20
Flaschenhals-Schichten 376
Fluch der Dimensionalität 207
 siehe auch Dimensionsreduktion
Folds 71, 83
Follow The Regularized Leader (FTRL) 302
Forget Gate 407
Fotodienste 13
Freiheitsgrade 28, 128
frühes Anhalten 198
functools.partial() 285, 423, 438

G

gamma, Wert 152
Gate Controller 408
Gaußsche RBF 151
Gaußscher RBF-Kernel 152, 162
Gedächtniszellen 350, 388
Genauigkeit 4, 84
geodätische Distanz 223
Gesetz der großen Zahl 183
Gesichtserkennung 102
get_variable() 247
Gewichte 267
 einfrieren 291
 GINI-Unreinheit 169, 173
Gittersuche 73, 151
Glättungsterme 299, 301, 438
Gleichgewicht zwischen Bias und Varianz 128
Gleichheit als Nebenbedingung 516
Gleichung mit geschlossener Form 107, 137
Global Average Pooling 378
global_step 473
global_variables_initializer() 231
Google 227
Google Images 253
Google Photos 13
GoogleNet, Architektur 375
gpu_options.per_process_gpu_memory_fraction 321
Gradient Boosted Regression Trees (GBRT) 195
Gradient Boosting 195
Gradient Descent (GD)
 Optimierungsverfahren 237
Gradient Tree Boosting 195

- GradientDescentOptimizer 268
Gradienten, schwindende und explodierende 275, 405
Batch-Normalisierung 282
Gradient Clipping 286
Initialisierung nach Glorot und He 276
nicht sättigende Aktivierungsfunktionen 279
Gradientenaufstieg 449
Gradientenverfahren (GD) 107, 112, 163, 275, 296, 298
automatische Berechnung von Gradienten 236
Batch GD 116, 132
Definition 112
lokales versus globales Minimum 114
manuelle Berechnung von Gradienten 235
Mini-Batch-GD 121, 238
Stochastisches Gradientenverfahren 118, 148
mit TensorFlow 235
Vergleich von Algorithmen 121
gradients() 236
graphviz 168
Greedy-Algorithmus 172
group() 473
GRÜ-(Gated-Recurrent-Unit-)Zelle 410
Grundlagen von Machine Learning
Arten von Systemen 7
Batch- und Online-Learning 14
instanzbasiertes versus modellbasiertes Lernen 17
überwachtes/unüberwachtes Lernen 8
Attribute 9
Beispiel Arbeitsablauf 22
Definition 4
Gründe für die Verwendung 4
Herausforderungen 22
Probleme mit Algorithmen 26
Probleme mit Trainingsdaten 26
Merkmale 9
Spamfilter-Beispiel 4
Testen und Validieren 30
Überblick 3
Workflow-Beispiel 18
Zusammenfassung 29
- H**
- Hailstone-Folge 420
halbüberwachtes Lernen 13
Hard-Margin-Klassifikation 146
Hard-Voting-Klassifikatoren 181
harmonischer Mittelwert 88
Hauptkomponente 212
Hauptkomponentenzerlegung (PCA) 211
als Komprimierungsverfahren 216
Anteil erklärter Varianz 214
Finden von Hauptkomponenten 212
inkrementelle PCA 217
Kernel PCA (kPCA) 218
Projektion auf d Dimensionen 213
randomisierte PCA 217
Scikit-Learn für 214
Varianz, erhalten 211
zur Komprimierung 217
Heaviside-Funktion 258
Hebbische Lernregel 259, 527
Hinge Loss-Funktion 164
Histogramm 47
Hold-out-Datensatz *siehe* Blending
Hopfield-Netze 527
Hyperebene 157, 210, 213, 223
Hyperparameter 28, 66, 73, 113, 151, 154
siehe auch Hyperparameter neuronaler Netze
Hyperparameter neuronaler Netze 270
Aktivierungsfunktionen 272
Anzahl verborgener Schichten 271
Neuronen pro verborgener Schicht 272
Hypothese 39
Manifold 210
Hypothesenfunktion 109
Hypothesis Boosting *siehe* Boosting
- I**
- Identitätsmatrix 130, 159
ILSVRC ImageNet Challenge 371
in_top_k() 268
Inception-Module 375
Inception-v4 381
Inferenz 22, 313, 369, 414
info() 45
Informationstheorie 173
In-Graph-Replikation 347
Initialisierung nach Glorot 276
Initialisierung nach He 276
Initialisierung nach Xavier 276
inkrementelles Lernen 16, 217
Input Gate 408
input_keep_prob 405
instanzbasiertes Lernen 17, 21
inter_op_parallelism_threads 327
InteractiveSession 231

Internal Covariate Shift-Problem 282
intra_op_parallelism_threads 327
inverse_transform() 220
isolierte Umgebung 41, 42
Isomap 223

J
Jobs 328
join() 329, 343
Jupyter 41, 42, 48

K
Kanten von Knoten 243
Karte-Kuhn-Tucker-(KKT-)Bedingungen 516
Kategorie, vorhergesagte 87
kategorische Merkmale 63
keep-Wahrscheinlichkeit 308
Keras 229
Kernel PCA (kPCA) 218
Kernels 150, 325
Kernel-SVM 161
Kernel-Trick 150, 152, 161, 218
k-fache Kreuzvalidierung 71, 85
Klassifikation versus Regression 8, 103
Klassifikatoren
 Abstimmverfahren 181
 auswerten 98
 binäre 83
 Fehleranalyse 98
 für die stochastische Gradientenverfahren (SGD) 84
 mehrere Ausgaben 103
 mehrere Kategorien 96
 mehrere Labels 102
 MNIST-Datensatz 81
 Qualitätsmaße 84
 Relevanz von 87
k-nächste-Nachbarn 22, 102
Komplexität der Berechnung 112, 153, 172
Konfusionsmatrix 86, 87, 98
Konnektionismus 260
Konvergenzrate 118
konvexe Funktion 114
Kopplung von Gewichten 424
Korrelationen, finden 56
Korrelationskoeffizient 56
Kostenfunktion 20, 39
 bei AdaBoost 193
 bei AdaGrad 299
 bei Autodiff 236
 bei der Batch-Normalisierung 285

Deep-Q-Learning 473
bei Elastic Net 133
bei der Gradientenmethode 199
beim Gradientenverfahren 107, 112, 116, 118, 275
Kreuzentropie 373
bei künstlichen neuronalen Netzen 264, 267
bei der Lasso-Regression 131
bei der linearen Regression 110, 114
bei der logistischen Regression 136
bei der Momentum Optimization 296
bei PG-Algorithmen 457
bei der Ridge-Regression 129
bei RNNs 395, 399
Stale Gradients und 354
bei Variational Autoencodern 438
in vortrainierten Schichten 295
kreative Sequenzen 401
Kreuzentropie 141, 264, 436, 457
Kreuzvalidierung 31, 71, 84
Kullback-Leibler-Divergenz 142, 434
künstliche neuronale Netze (KNNs) 253
 Boltzmann Machines 529
 Deep Belief Nets (DBNs) 532
 Evolution von 254
 Hopfield-Netze 527
 Optimieren von Hyperparametern 270
 Perzeptrons 257
 Selbstorganisierende Karten 534
 Trainieren eines DNN mit TensorFlow 265
 Überblick 253

L
l1_l2_regularizer() 306
LabelBinarizer 68
Labels 8, 37
Lagrange-Funktion 516
Lagrange-Multiplikator 515
Landmarken 151
Large-Margin-Klassifikation 145
Lasso-Regression 131
latenter Raum 437
latenter Verlust 438
Leaky ReLU 279
Learning-Algorithmus
 Deep-Q-Learning 468
LeNet-5-Architektur 361, 372
Lernrate 17, 113, 117
 vorgefertigte stückweise konstante 303
Leseoperationen 339

Levenshtein-Distanz 153
liblinear, Bibliothek 153
libsvm, Bibliothek 154
Linear Threshold Units (LTUs) 257
lineare Diskriminantenanalyse (LDA) 223
lineare Klassifikation mit SVMs 145
lineare Modelle
 Early Stopping 134
 Elastic Net 133
 Lasso-Regression 131
 lineare Regression *siehe* lineare Regression
 Regression *siehe* lineare Regression
 Regularisieren von Modellen *siehe* Regularisierung
 Ridge-Regression 129, 133
 SVM 145
lineare Regression 20, 69, 107, 122, 133
 Gradientenverfahren bei 112, 122
 Komplexität der Berechnung 112
 Lernkurven in 124
 mit dem stochastischen Gradientenverfahren (SGD) 121
 mit TensorFlow 233, 234
 Normalengleichung 110, 112
lineare SVM-Klassifikation 148
LinearSVR, Klasse 156
Lipschitz-stetig 115
LLE (Locally Linear Embedding) 221
load_sample_images() 366
Local Response Normalization 374
location invariance 369
log loss 137
log_device_placement 324
Loggen von Platzierungen 324
logistische Funktion 136
logistische Regression 9, 135
 Abschätzen von Wahrscheinlichkeiten 135
 Entscheidungsgrenzen 138
 Softmax-Regressionsmodell 140, 143
 trainieren und Kostenfunktion 136
lokale Sessions 332
lokales Wahrnehmungsfeld 360
LSTM (Long Short-Term Memory)-Zelle 406

M

Machine Learning
 Anwendungen XV
 Beispielprozess 33
 Checkliste für Projekte 35, 509

Großprojekte *siehe* TensorFlow
Ressourcen zu XVIII
Schreibweisen 38
make() 450
Manhattan-Metrik 40
Manifold Learning 210, 221
 siehe auch LLE (Locally Linear Embedding)
Manifold-Annahme/Hypothese 210
MapReduce 37
Margin-Verletzungen 147
Markov-Entscheidungsprozesse 461
Markov-Ketten 461
maschinelle Übersetzung *siehe* natürliche Sprachverarbeitung (NLP)
Maschinensteuerung *siehe* Reinforcement Learning (RL)
Master-Dienst 330
Maße für Unreinheit 169, 173
Matplotlib 41, 48, 93, 99
Max Margin Learning 295
max_norm_regularizer() 310
max_norm() 310
max_pool() 371
Max-Norm-Regularisierung 309
Max-Pooling Layer 369
mehrschichtige Perzeptrons (MLP) 253, 263, 454
 trainieren mit TF.Learn 264
Meinungsanalyse 385
memmap 217
Mercers Theorem 162
Merkmale 9
 siehe auch Datenstruktur
 entwickeln 26
 extrahieren 12
 Kombinationen von 58
 skalieren 66
 vorverarbeitete 48
 Wichtigkeit von 190
 Ziel 48
Merkmalsauswahl 75, 132, 191, 512
Merkmalserkennung 419
Merkmalsraum 218
Merkmalsvektor 39, 109, 156, 235
Merkmalszuordnung 219
Meta-Lerner *siehe* Blending
min_after_dequeue 337
Mini-Batches 15
Mini-Batch-Gradientenverfahren 121, 137, 238
minimize() 286, 291, 457, 473
Min-Max-Skalierung 66

- mittlere absolute Abweichung (MAE) 39
 mittlere quadratische Abweichung (MSE) 109, 235, 434
 mittlerer absoluter Fehler (MAE) 39
 mittleres Coding 437
 MNIST-Datensatz 81
 Modalwert 185
 Modellauswahl 19
 modellbasiertes Lernen 18
 Modelle
 analysieren 76
 evaluieren 30
 evaluieren auf dem Testdatensatz 77, 78
 generative 419, 530
 parameterfreie 173
 parametrische 174
 trainieren 20
 logistische Regression 143
 Modellparameter 115, 118, 135, 156, 232, 268, 395
 Definition 19
 Modell-Zoos 293
 Moment 300
 Momentum Optimization nach Nesterov 297, 298
 multidimensionale Skalierung (MDS) 223
 multinomial() 455
 multinomiale logistische Regression *siehe* Softmax-Regression
 MultiRNNCell 403
 multivariate Regression 37
- N**
- naive Bayes-Klassifikatoren 96
 Name Scopes 244
 natürliche Sprachverarbeitung (NLP) 411
 Encoder-Decoder-Netz zur maschinellen Übersetzung 413
 TensorFlow-Tutorials 411
 Word Embeddings 411
 Nebenbedingungen, aktive 516
 Netzwerk-Bandbreite, erschöpfen 354
 Netzwerktopologie 270
 neuron_layer() 267
 neuronale Netze als Policies 452
 Neuronen
 biologische 254
 logische Berechnungen mit 256
 next_batch() 269
 nichtlineare Dimensionsreduktion (NLDR) *siehe* Kernel PCA; LLE (Locally Linear Embedding)
 nichtlineare SVM-Klassifikation 149
 ähnlichkeitsbasierte Merkmale, hinzufügen 151, 152
 Gaußscher RBF-Kernel 152
 Komplexität der Berechnung 153
 mit polynomiellen Merkmalen 149
 polynomieller Kernel 150
 No-Free-Lunch-Theorem 31
 Normalengleichung 110
 Normalisieren 66
 normalisierte Exponentialfunktion 140
 Normalverteilung 437, 439
 Normen 39
 NP-vollständige Probleme 172
 Nullhypothese 174
 numerische Differenzierung 521
 NumPy 41
 NumPy-Arrays 64
 Nutzenfunktion 20
 Nvidia Compute Capability 318
 nvidia-smi 322
- 0**
- Offline-Learning 14
 off-Policy-Algorithmus 467
 One-Hot-Codierung 63
 One-versus-All-(OvA-)Strategie 96, 143, 165
 One-versus-One-(OvO-)Strategie 96
 Online-Learning 15
 Online-SVMs 163
 OpenAI Gym 449
 operation_timeout_in_ms 349
 Operationen pinnen 330
 Optimalitätsprinzip von Bellman 463
 Optimierer 295
 AdaGrad 298
 Adam-Optimierung 300, 302
 Beschleunigter Gradient nach Nesterov (NAG) 297
 Gradientenverfahren *siehe* GradientDescentOptimizer
 Momentum Optimization 296
 RMSProp 300
 Scheduling der Lernrate 302
 Optimierung mit Nebenbedingungen 158, 515
 optische Zeichenerkennung (OCR) 3
 Out-of-Bag-Evaluation 187
 Out-of-Core-Learning 16
 Out-of-Memory-(OOM-)Fehler 392
 OutOfRangeError 341, 343
 out-of-sample error 30

- Output Gate 408
`output_keep_prob` 405
 OutputProjectionWrapper 398
 Overfitting 26, 49, 147, 153, 174, 176, 272
 vermeiden durch Regularisierung 305
- P**
- PaddingFIFOQueue 338
 Pandas 41, 44
 `scatter_matrix` 57
 paralleles verteiltes Rechnen 317
 ein neuronales Netz pro Recheneinheit 346
 In-Graph- versus Between-Graph-Replikation 347
 mehrere Recheneinheiten auf einem Computer 318
 Control Dependencies 327
 das RAM der GPU verwalten 321
 Installation 318
 Operationen auf Recheneinheiten platzieren 322
 paralleles Ausführen 326
 mehrere Recheneinheiten auf mehreren Servern 328
 asynchrone Kommunikation mit Queues 334
 Master- und Worker-Dienste 330
 Öffnen einer Session 329
 Operationen auf Tasks pinnen 330
 Sharding von Variablen 331
 Trainingsdaten laden 339
 Zustände zwischen Sessions teilen 332
 parallelisierte Daten 351
 parallelisierte Modelle 349
 parallelisierte Daten 351
 asynchrone Updates 353
 Erschöpfen der Netzwerk-Bandbreite 354
 Implementierung in TensorFlow 355
 synchrone Updates 352
 parallelisierte Modelle 349
 Parameter Server (ps) 328
 Parametereffizienz 271
 Parametermatrix 140
 Parameterraum 115
 Parametervektor 109, 113, 136, 140
`partial_fit()` 217
 partielle Ableitung 116
 partielle Ableitungen erster Ordnung (Jacobians) 302
- partielle Ableitungen zweiter Ordnung (Hessians) 302
 Pearson 56
 Peephole-Verbindungen 409
 Performance Scheduling 303
`permutation()` 49
 Perzentil 46
 Perzeptron-Konvergenztheorem 259
 Perzeptrons 257, 263
 im Vergleich zu logistischer Regression 260
 trainieren 259
 PG-Algorithmen 456, 461
`pip` 41
 Pipeline-Konstruktor 67
 Pipelines 36
 Pipelines zur Transformation 67
 Platzhalter-Knoten 238
 Platzierungsverfahren *siehe* dynamische Platzierung
 Policy 447
 Policy-Gradienten 449
 siehe auch PG-Algorithmen
 Policy-Raum 448
 polynomielle Merkmale, hinzufügen 149, 150
 polynomielle Regression 108, 122
 Lernkurven bei 124, 128
 polynomieller Kernel 150, 162
 Pooling Layer 369
 Pooling-Kernel 369
 Power Scheduling 304
 Prädiktoren 8, 62
`predict()` 62
 Preloading von Trainingsdaten 339
 PReLU (parametrisierte Leaky ReLU) 280
 Pretty Tensor 229
 primales Problem 160
 Producer-Funktionen 345
 Projektion 207
 Pruning 174, 521
 p-Wert 174
 Python
 isolierte Umgebung in 41
 Notebooks in 42
 `pickle` 73
 `pip` 41
- Q**
- `q_network()` 471
 Q-Learning-Algorithmus 466
 aproximatives Q-Learning 467

- Deep-Q-Learning 468
Quadratische Programme (QP) 159
Qualitätsmaße 37
Konfusionsmatrix 86
Kreuzvalidierung 84
Relevanz und Sensitivität 88
ROC-(Receiver-Operating-Characteristic-)Kurve 92
Quantizing 355
Queries pro Sekunde (QPS) 347
QueueRunner 342, 344
Queues 334, 338
aus Tupeln 336
Daten aus einer Queue holen 335
Daten in eine Queue stellen 334
First-In-First-Out (FIFO) 334
PaddingFIFOQueue 338
RandomShuffleQueue 337
schließen 337
Q-Werte 464
Q-Wert-Iterationsalgorithmus 464
- R**
radiale Basisfunktion (RBF) 151
Random Forests 72, 96, 167, 178, 181, 189
Extra-Trees 190
Wichtigkeit von Merkmalen 190
random_uniform() 235
randomisierte Leaky ReLU (RReLU) 279
randomisierte PCA 217
RandomShuffleQueue 337, 341
rechtsschief 48
Recognition Network 420
Reconstruction Pre-Image 220
reduce_mean() 268
reduce_sum() 435, 438, 473
Regression 8
Entscheidungsbäume 175
versus Klassifikation 103
Regressionsmodelle
lineare 69
Regularisierung 27, 30, 128, 135
Data Augmentation 311
Drop-out 306
Early Stopping 134, 305
Elastic Net 133
Entscheidungsbäume 173
 ℓ_1 - und ℓ_2 -Regularisierung 305
Lasso-Regression 131
Max-Norm 309
nach Tikhonov 129
Ridge-Regression 129
- Shrinkage 196
REINFORCE-Algorithmen 456
Reinforcement Learning (RL) 13, 445
Aktionen 455
Beispiele für 446
Belohnungen, lernen zu optimieren 446
Credit-Assignment-Problem 455
Discount-Rate 455
Markov-Entscheidungsprozesse 461
neuronale Netze als Policies 452
OpenAI Gym 449
PG-Algorithmen 456
Policy-Suche 449
Q-Learning-Algorithmus 466
Suche nach Policies 447
Temporal Difference (TD) Learning 465
Rekonstruktionen 420
Rekonstruktionsfehler 216
Rekonstruktionsverlust 420, 436, 438
rekurrente neuronale Netze (RNNs) 385
Deep-RNNs 402
Ein- und Ausgabesequenzen 388
Erkundungspolicies 467
GRU-Zelle 410
LSTM-Zelle 406
natürliche Sprachverarbeitung (NLP)
411
in TensorFlow 390
Ausgabesequenzen mit unterschiedlicher Länge 394
dynamisches Aufrollen entlang der Zeitachse 393
Eingabesequenzen unterschiedlicher Länge 393
statisches Aufrollen entlang der Zeitachse 391
trainieren 395
Backpropagation through Time (BPTT) 395
kreative Sequenzen 401
Sequenz-Klassifikatoren 395
Vorhersage von Zeitreihen 397
rekurrente Neuronen 386
Gedächtniszellen 388
Relevanz 87
Relevanz und Sensitivität 88
F-1-Score 88
Relevanz-Sensitivitäts-Kurve (PR-Kurve)
94
Wechselbeziehung zwischen Relevanz und Sensitivität 89
ReLU (Rectified Linear Units) 245

relu(z) 267
 ReLU-Aktivierungsfunktion 379
 ReLU-Funktion 262, 272, 278
 render() 450
 Replay-Speicher 473
 replica_device_setter() 331
 request_stop() 343
 reset_default_graph() 232
 reset() 450
 reshape() 401
 Residual Learning 378
 Residual Network (ResNet) 293, 378
 Residual Units 379
 ResNet 378
 Resource Containers 332
 Restfehler 195
 restore() 240
 Restricted Boltzmann Machines (RBMs) 13, 293, 530
 reuse_variables() 248
 rgb_array 451
 Richtig-negativ-Rate (TNR) 93
 Richtig-positiv-Rate (TPR) 87, 93
 Ridge-Regression 129, 133
 RMSProp 300
 ROC-(Receiver-Operating-
 Characteristic-)Kurve 92
 Root Mean Square Error (RMSE) 109
 RReLU (randomisierte Leaky ReLU) 279
 run() 231, 349

S

Sampled Softmax 415
 save() 239
 Saver-Knoten 239
 Scheduling der Lernrate 119, 302
 Scheduling, exponentielles 303
 Schreibweisen 38
 Schrittweite 362
 schwache Lerner 182
 Schweigeverzerrung 25
 schwindende Gradienten *siehe* Gradienten,
 schwindende und explodierende
 Scikit Flow 228
 Scikit-Learn 41
 Bagging und Pasting in 186
 CART-Algorithmus 169, 176
 Design 61
 Einführung XVI
 Hyperparameter min_ und max_ 174
 Imputer 61
 Klassen zur SVM-Klassifikation 154
 Kreuzvalidierung 71

LinearSVR, Klasse 156
 MinMaxScaler 66
 PCA-Implementierung 214
 Perceptron-Klasse 259
 Pipeline-Konstruktor 67, 149
 randomisierte PCA 217
 Ridge-Regression mit 130
 SAMME 194
 SGDClassifier 84, 90, 97
 SGDRegressor 121
 sklearn.base.BaseEstimator 65, 68, 85
 sklearn.base.clone() 85, 135
 sklearn.base.TransformerMixin 65, 68
 sklearn.datasets.fetch_california_housing() 234
 sklearn.datasets.fetch_mldata() 81
 sklearn.datasets.load_iris() 138, 148, 167, 191, 259
 sklearn.datasets.load_sample_images() 366
 sklearn.datasets.make_moons() 149, 179
 sklearn.decomposition.IncrementalPCA 217
 sklearn.decomposition.KernelPCA 218, 220
 sklearn.decomposition.PCA 214
 sklearn.ensemble.AdaBoostClassifier 195
 sklearn.ensemble.BaggingClassifier 186
 sklearn.ensemble.GradientBoostingRegressor 196
 sklearn.ensemble.RandomForestClassifier 94, 97, 184
 sklearn.ensemble.RandomForestRegressor 72, 73, 189, 196
 sklearn.ensemble.VotingClassifier 184
 sklearn.externals.joblib 73
 sklearn.linear_model.ElasticNet 134
 sklearn.linear_model.Lasso 133
 sklearn.linear_model.LinearRegression 21, 62, 69, 112, 122, 123, 126
 sklearn.linear_model.LogisticRegression 138, 140, 142, 184, 219
 sklearn.linear_model.Perceptron 259
 sklearn.linear_model.Ridge 130
 sklearn.linear_model.SGDClassifier 84
 sklearn.linear_model.SGDRegressor 121, 131, 133
 sklearn.manifold.LocallyLinearEmbedding 221

sklearn.metrics.accuracy_score() 184, 188, 264
sklearn.metrics.confusion_matrix() 86, 98
sklearn.metrics.f1_score() 88, 103
sklearn.metrics.mean_squared_error() 70, 77, 125, 135, 198, 221
sklearn.metrics.precision_recall_curve() 90
sklearn.metrics.precision_score() 88, 92
sklearn.metrics.recall_score() 88, 92
sklearn.metrics.roc_auc_score() 94, 95
sklearn.metrics.roc_curve() 93, 94
sklearn.model_selection.cross_val_predict() 86, 90, 94, 98, 103
sklearn.model_selection.cross_val_score() 71, 84
sklearn.model_selection.GridSearchCV 73, 79, 98, 179, 219
sklearn.model_selection.StratifiedKFold 85
sklearn.model_selection.StratifiedShuffleSplit 52
sklearn.model_selection.train_test_split() 50, 71, 125, 179, 198
sklearn.multiclass.OneVsOneClassifier 97
sklearn.neighbors.KNeighborsClassifier 102, 105
sklearn.neighbors.KNeighborsRegressor 22
sklearn.pipeline.FeatureUnion 68
sklearn.pipeline.Pipeline 67, 127, 148, 219
sklearn.preprocessing.Imputer 61, 69
sklearn.preprocessing.LabelBinarizer 64, 68
sklearn.preprocessing.LabelEncoder 63
sklearn.preprocessing.OneHotEncoder 64
sklearn.preprocessing.PolynomialFeatures 123, 127, 130, 149
sklearn.preprocessing.StandardScaler 67, 98, 115, 129, 146, 148, 152, 235, 264
sklearn.svm.LinearSVC 147, 149, 154, 156, 165, 168
sklearn.svm.LinearSVR 155
sklearn.svm.SVC 148, 150, 154, 156, 165, 184
sklearn.svm.SVR 79, 156
sklearn.tree.DecisionTreeClassifier 174, 179, 186, 189, 195
sklearn.tree.DecisionTreeRegressor 70, 167, 175, 195
sklearn.tree.export_graphviz() 168
StandardScaler 115, 235, 264
TF.Learn 228
User Guide XVIII
score() 62
Selbstorganisierende Karten (SOMs) 534
Semantic Hashing 442
Sensitivität 87, 93
separable_conv2d() 382
sequence_length 393, 415
Sequenzen 385
Shannons Informationstheorie 173
Shortcut-Verbindungen 378
show_graph() 244
Shrinkage 196
shuffle_batch_join() 345
shuffle_batch() 345
sigmoid_cross_entropy_with_logits() 436
sigmoide Funktion 136
Simulated Annealing 119
simulierte Umwelt *siehe* OpenAI Gym
Singular Value Decomposition (SVD) 213
Skip-Verbindungen 312, 378
sklearn.neighbors.KNeighborsRegressor 22
Slack-Variable 158
Smoothing-Term 283
Soft Placement 325
Soft Voting 184
Soft-Margin-Klassifikation 146
Softmax-Funktion 140, 263
Softmax-Regression 140
Source-Ops 234, 326
Spamfilter 3, 8
spärliche Modelle 132, 302
Spärlichkeitsverlust 434
Sparse Autoencoder 434
sparse_softmax_cross_entropy_with_logits() 268
Sparse-Matrix 64
Spezifität 93
Spiele *siehe* Reinforcement Learning (RL)
Spracherkennung 6
Sprachverarbeitung (NLP) 385
spurious Patterns 528
stack() 392
Stacked Autoencoder 423
Implementierung mit TensorFlow 423
Kopplung von Gewichten 424
Trainieren mehrerer nacheinander 426
unüberwachtes Vortrainieren mit 430

- Visualisieren der Rekonstruktionen 428
Stacked Denoising Autoencoder 430, 432
Stacked Denoising Encoder 432
Stacked Generalization *siehe* Stacking
Stacking 200
Stale Gradients 353
Standardisierung 66
StandardScaler 68, 235, 264
starke Lerner 182
state_is_tuple 404, 407
states, Tensor 394
static_rnn() 391, 415
stationärer Punkt 515, 517
statische Signifikanz 174
statisches Aufrollen entlang der Zeitachse 391
Stemming 106
step() 451
sterbende ReLUs 279
Stichproben-Bias 26, 51
Stichprobenrauschen 25
Stichprobenverzerrung 25
Stimmerkennung 359
stochastische Neuronen 529
stochastische Policy 448
stochastisches Gradient Boosting 199
stochastisches Gradientenverfahren (SGD) 118, 148, 260
 Klassifikator 131
 trainieren 137
Strafen *siehe* Belohnungen, in RL
stratifizierte Stichprobe 51, 85
String Kernels 153
string_input_producer() 345
Stützvektoren 146
Subdifferentiale 164
Subgradientenvektor 133
subsample 199
Suchraum 75, 270
Support Vector Machines (SVMs) 96, 145
 duales Problem 160, 515
 Entscheidungsfunktion und Vorher-sagen 156
 Kernel-SVM 161
 lineare Klassifikation 145
 Mechanismen von 156
 nichtlineare Klassifikation 149
 Online-SVMs 163
 Quadratische Programme (QP) 159
 SVM-Regression 154
 Zielfunktionen beim Trainieren 157
svd() 213
symbolische Differenzierung 236, 520
synchrone Updates 352
Systeme zum autonomen Fahren 385
- T**
- Tangens hyperbolicus (htan-Aktivierungs-funktion) 262, 272, 276, 278, 387
target_weights 415
Tasks 328
tatsächliche Kategorie 87
Temporal Difference (TD) Learning 465
Tensor Processing Units (TPUs) 319
TensorBoard 229
TensorFlow 227
 Ausführungsphase 232
 Autodiff 236, 519
 Batch-Normalisierung mit 284
 Control Dependencies 327
 Convolutional Layers 382
 Convolutional Neural Networks und 366
 Daten in den Trainingsalgorithmus einspeisen 238
 Denoising Autoencoder 433
 Drop-out mit 308
 dynamische Platzierung 322
 einen ersten Graphen erstellen und ausführen 230
 einfache Platzierung 322
 Einführung XVI
 Gradientenverfahren mit 235
 Graphen, verwalten 232
 Installation 230
 Konstruktionsphase 232
 l1- und l2-Regularisierung mit 305
 Lebenszyklus des Werts von Knoten 232
 Lineare Regression mit 233
 Max-Norm-Regularisierung mit 309
 Max-Pooling Layer in 370
 Modelle speichern und wiederherstellen 239
 Modell-Zoos 293
 Modularität 245
 Momentum Optimization in 297
 Name Scopes 244
 neuronale Netze als Policies 454
 NLP Tutorials 411, 414
 nützliche Funktionen 345
 Operationen (Ops) 233
 Optimierungsverfahren 237
 parallelisierte Daten und 355
 Python-API

Ausführung 269
Konstruktion 265
Verwenden des neuronalen Netzes
 270
Queues *siehe* Queues
RNNs in *siehe* rekurrente neuronale
 Netze (RNNs)
Scheduling der Lernrate in 304
Sparse Autoencoder mit 435
 und Stacked 423
Teilen von Variablen 247
TensorBoard 240
tf.abs() 305
tf.add_n() 245, 248, 306
tf.add_to_collection() 310
tf.add() 245, 305
tf.assign() 235, 290, 309, 491
tf.bfloat16 355
tf.bool 308
tf.cast() 268, 396
tf.clip_by_norm() 309
tf.clip_by_value() 286
tf.concat() 314, 376, 454, 458
tf.ConfigProto 321, 324, 349, 496
tf.constant_initializer() 247
tf.constant() 232, 323, 327, 329
tf.container() 333, 356, 490
tf.contrib.layers.l1_regularizer() 306, 310
tf.contrib.layers.l2_regularizer() 306, 424
tf.contrib.layers.variance_scaling_initializer()
 278, 397, 424, 438, 454,
 458, 471
tf.contrib.learn.DNNClassifier 264
tf.contrib.learn.infer_real_valued_
 columns_from_input() 264
tf.contrib.rnn.BasicLSTMCell 407, 409
tf.contrib.rnn.BasicRNNCell 391, 396,
 398, 400, 407
tf.contrib.rnn.DropoutWrapper 404
tf.contrib.rnn.GRUCell 411
tf.contrib.rnn.LSTMCell 409
tf.contrib.rnn.MultiRNNCell 403
tf.contrib.rnn.OutputProjection-
 Wrapper 398
tf.contrib.rnn.RNNCell 403
tf.contrib.rnn.static_rnn() 391, 415, 502
tf.contrib.slim module 228, 383
tf.contrib.slim.nets module (nets) 383
tf.control_dependencies() 327
tf.decode_csv() 340, 344
tf.device() 323, 330, 403
tf.exp() 438
tf.FIFOQueue 334, 336, 340, 344
tf.float32 234, 491
tf.get_collection() 289, 306, 310, 424,
 471
tf.get_default_graph() 232, 241
tf.get_default_session() 231
tf.get_variable() 247, 290, 306
tf.global_variables_initializer() 231, 235
tf.global_variables() 288
tf.gradients() 236
tf.Graph 230, 232, 240, 339, 347
tf.GraphKeys.GLOBAL_VARIABLES
 289
tf.GraphKeys.REGULARIZATION_
 LOSSES 306, 424
tf.GraphKeys.TRAINABLE_VARIABLES
 291, 471
tf.group() 472
tf.int32 325, 341, 393, 396, 412, 473
tf.int64 265
tf.InteractiveSession 231
tf.layers.batch_normalization() 284
tf.layers.dense() 267
TF.Learn 264
tf.log() 436, 438, 454, 458
tf.matmul() 234, 245, 265, 390, 425,
 428, 433
tf.matrix_inverse() 234
tf.maximum() 245, 247, 282
tf.multinomial() 454, 458
tf.name_scope() 244, 246, 265, 267, 427
tf.nn.conv2d() 366
tf.nn.dynamic_rnn() 392, 396, 398, 400,
 403, 415, 502
tf.nn.elu() 281, 424, 438, 454, 458
tf.nn.embedding_lookup() 412
tf.nn.in_top_k() 268, 396
tf.nn.max_pool() 370
tf.nn.relu() 265, 398, 400, 471
tf.nn.sigmoid_cross_entropy_with_
 logits() 436, 439, 457
tf.nn.sparse_softmax_cross_entropy_
 with_logits() 267, 396
tf.one_hot() 473
tf.PaddingFIFOQueue 338
tf.placeholder_with_default() 433
tf.placeholder() 238, 491
tf.random_normal() 245, 390, 433, 438
tf.random_uniform() 235, 239, 412, 491
tf.RandomShuffleQueue 337, 341, 344
tf.reduce_mean() 235, 244, 267, 305,
 396, 422, 426, 428, 433, 436,
 473
tf.reduce_sum() 305, 436, 439, 473

tf.reset_default_graph() 232
 tf.reshape() 400, 471
 tf.RunOptions 349
 tf.Session 231, 491
 tf.shape() 433, 438
 tf.square() 235, 244, 399, 422, 424, 426,
 428, 433, 436, 438, 473
 tf.stack() 340, 344, 392
 tf.string 340, 344
 tf.summary.FileWriter 241
 tf.summary.scalar() 241
 tf.tanh() 390
 tf.TextLineReader 340, 344
 tf.to_float() 457
 tf.train.AdamOptimizer 301, 396, 399,
 422, 424, 425, 436, 439, 457,
 473
 tf.train.ClusterSpec 328
 tf.train.Coordinator 342
 tf.train.exponential_decay() 304
 tf.train.GradientDescentOptimizer 237,
 268, 286, 297, 301
 tf.train.MomentumOptimizer 237, 297,
 304, 313, 356, 494
 tf.train.QueueRunner 342
 tf.train.replica_device_setter() 331
 tf.train.RMSPropOptimizer 300
 tf.train.Saver 239, 268, 383, 405, 458,
 473
 tf.train.Server 328
 tf.train.start_queue_runners() 345
 tf.transpose() 234, 392, 425
 tf.truncated_normal() 265
 tf.unstack() 391, 401, 502
 tf.Variable 230, 491
 tf.variable_scope() 247, 290, 333, 397,
 471
 tf.zeros() 265, 390, 425
 Truncated Backpropagation through
 Time 406
 Überblick 227
 Visualisieren von Graphen und
 Lernkurven 240
 Wiederverwenden vortrainierter
 Schichten 288
 TensorFlow Serving 347
 tensorflow.contrib 264
 Testdatensatz 30, 49, 83
 Testen und Validieren 30
 TextLineReader 340
 Text-Merkmale 63
 tf.layers.conv1d() 382
 tf.layers.conv2d_transpose() 382
 tf.layers.conv2d() 471
 tf.layers.conv3d() 382
 tf.layers.dense() 278, 284
 tf.layers.separable_conv2d() 382
 TF.Learn 228, 264
 tf.nn.atrous_conv2d() 382
 tf.nn.depthwise_conv2d() 382
 tf.nn.elu() 281
 TF-slim 228
 thermisches Gleichgewicht 529
 Thread Pools (inter-Op/intra-Op, in
 TensorFlow 326
 threshold, Variable 247
 Tiefenradius 374
 toarray() 64
 Toleranz-Hyperparameter 154
 Trainieren 284, 405
 Trainieren von Modellen 107
 Lernkurven beim 124
 lineare Regression 107, 108
 logistische Regression 135
 polynomiale Regression 108, 122
 Überblick 107
 Trainingsdaten 4
 irrelevante Merkmale 26
 laden 339
 minderwertige 26
 mischen 83
 nicht repräsentative 24
 Overfitting 26
 Underfitting 29
 unzureichende Menge 23
 Trainingsdatenpunkt 4
 Trainingsdatensatz 4, 30, 53, 60, 69
 Kostenfunktion von 136
 Transfer Learning *siehe* Wiederverwenden
 vortrainierter Schichten
 transform() 62, 68
 Transformer 62
 Transformer, eigene 65
 transpose() 391
 Trefferquote 87
 Truncated Backpropagation through Time
 406
 Tupel 336
 t-verteiltes Stochastic Neighbor Embedding
 (t-SNE) 223

U

übervollständige Autoencoder 432
 überwachtes Lernen 8
 Umwelt, im Reinforcement Learning 446,
 467, 472

unbalancierte Datensätze 86
Underfitting 29, 70, 153
Ungleichheit als Nebenbedingung 516
univariate Regression 37
unstack() 391
Unterstichproben 369
unüberwachtes Lernen 10
 Algorithmen zur Dimensionsreduktion 12
 Algorithmen zur Visualisierung 10, 12
 Clustering 10
 Erkennen von Anomalien 12
 Lernen von Assoziationsregeln 10, 12
unüberwachtes Vortrainieren 293, 430
Upsampling 382

V

Validierungsdatensatz 31
Value Iteration 463
value_counts() 46
variable_scope() 247
Variablen, teilen 247
variance_scaling_initializer() 278
Varianz
 erhalten 211
 erklärte 215
 Gleichgewicht zwischen Bias und Varianz 128
Variational Autoencoder 436
Verallgemeinerungsfehler 30
verborgene Schichten 261
verschachtelte device-Blöcke 331
verteilte Sessions 332
verteiltes Rechnen 227
VGGNet 381
Visualisierung 240
visueller Cortex 360
vorgetäuschter X-Server 451
Vorhersagen 86, 156, 168
vorverarbeitete Merkmale 48

W

Wahrscheinlichkeiten, schätzen 135, 171
Warmup-Phase 354
while_loop() 393
White-Box-Modelle 170
Wiederverwenden vortrainierter Schichten 287
 andere Frameworks 290
 Caching eingefrorener Schichten 291
 Einfrieren der unteren Schichten 291
 Hilfsaufgaben 294
 Modell-Zoos 293
 obere Schichten 292
 TensorFlow, Modell 288
 unüberwachtes Vortrainieren 293
Worker 328
worker_device 331
Worker-Dienst 330
Wrapper einer Zelle 398
Wurzel der mittleren quadratischen Abweichung (Root Mean Square Error, RMSE) 37

Y

YouTube 253

Z

Zeitreihen 385
Zerfall, exponentieller 285
Zero Padding 362, 368
Zielfunktionen beim Trainieren 157
Zielgröße 48
zufällige Initialisierung 113, 117, 119, 276
zufällige Patches und zufällige Subräume 188
Zufallssuche 75, 270
Zugewinn an Information 173
Zustands-Aktions-Werte 464
Zustandswert, optimaler 463

Über den Autor

Aurélien Géron arbeitet als Consultant für Machine Learning. Als ehemaliger Mitarbeiter von Google hat er von 2013 bis 2016 das YouTube-Team zur Klassifikation von Videos geleitet. Er war von 2002 bis 2012 Gründer und CTO von Wifirst, einem führenden Wireless ISP in Frankreich; 2001 war er Gründer und CTO von Polyconseil, der Firma, die inzwischen den Carsharing-Dienst Autolib' verwaltet.

Davor war er als Ingenieur in verschiedenen Bereichen tätig: Finanzen (JP Morgan und Société Générale), Verteidigung (das Department of Defense in Kanada) und Gesundheit (Bluttransfusionen). Er hat einige technische Bücher veröffentlicht (zu C++, WiFi und Internetarchitekturen) und war Dozent für Informatik in einer französischen Ingenieurschule.

Sonstige wissenswerte Dinge: Er hat seinen drei Kindern beigebracht, mit den Fingern binär zu zählen (bis 1023), hat Mikrobiologie und Evolutionsgenetik studiert, bevor er sich der Softwareentwicklung zugewandt hat, und sein Fallschirm ging bei seinem zweiten Absprung nicht auf.

Über den Übersetzer

Dr. Kristian Rother arbeitet als Trainer für Forschungsmethoden, Datenverarbeitung und Kommunikation. Er ist in der Python-Community und im Redeclub Berliner.Rhetorik. Salon aktiv, um seine Fähigkeiten stetig weiterzuentwickeln. Kristian war bis 2011 als Wissenschaftler in der Bioinformatik aktiv, wo er Python-Software zum Modellieren der 3-D-Strukturen von Makromolekülen entwickelt hat. Er hat 2006 an der HU Berlin promoviert.

Kolophon

Das Tier auf dem Cover von *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow* ist ein Kleinasiatischer Feuersalamander (*Salamandra infraimmaculata*), eine Amphibie, die im Nahen Osten zu finden ist. Die Haut ist schwarz mit großen gelben Flecken auf Rücken und Kopf, die als Warnfarben Fressfeinde fernhalten sollen. Ausgewachsene Tiere können bis zu 30 Zentimeter groß werden, meist werden sie aber 19 bis 24 Zentimeter lang.

Kleinasiatische Feuersalamander leben in subtropischen Strauch- und Waldregionen in der Nähe zu Flüssen und Feuchtgebieten. Sie verbringen den Großteil ihres Lebens auf dem Land, legen ihre Eier aber im Wasser ab. Die Nahrung besteht aus Insekten, Würmern, kleinen Krebstierchen und manchmal auch anderen Salamandern. Männchen können 23 Jahre alt werden, Weibchen immerhin 21 Jahre.

Auch wenn der Kleinasiatische Feuersalamander noch nicht zu den gefährdeten Amphibien gehört, so ist die Art in ihrem Bestand doch bedroht. Flussregulierungen, Dämme und die Verschmutzung des Wassers stören die Fortpflanzung. Des Weiteren hat das Aussetzen von Moskitofischen (Koboldkäpfchen) zur weiteren Reduktion der Populationen geführt. Diese Fische, die zur Bekämpfung von Stechmückenlarven in Flüssen eingesetzt wurden, um die Malaria einzudämmen, fressen genauso gern auch junge Salamander.

Viele der Tiere auf den O'Reilly-Covern sind vom Aussterben bedroht. Doch jedes einzelne von ihnen ist für den Erhalt unserer Erde wichtig. Wie man bedrohten Arten helfen kann, erfahren Sie auf animals.oreilly.com.

Der Umschlagsentwurf dieses Buchs basiert auf dem Reihenlayout von Edie Freedman und stammt von Randy Comer und Michael Oreal, die hierfür einen Stich aus *Wood's Illustrated Natural History* verwendet haben. Auf dem Cover verwenden wir die Schriften URW Typewriter und Guardian Sans, als Textschrift die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed, und die Nichtproportionalschrift für Codes ist LucasFonts TheSans Mono Condensed.